HAW HAMBURG

BACHELOR THESIS

Tom Hert

Porting RIOT OS to the RP2350: An Exploration of a Heterogeneous Architecture and Programmable I/O

Faculty of Computer Science and Digital Society

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Tom Hert

Porting RIOT OS to the RP2350: An Exploration of a Heterogeneous Architecture and Programmable I/O

Bachelor thesis submitted for examination in Bachelor's degree

in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Faculty of Computer Science and Digital Society
at University of Applied Science Hamburg

Supervising examiner: **Prof. Dr. Thomas C. Schmidt**
Second examiner: **Prof. Dr. Franz Korf**

Submitted on: January 15, 2026

Tom Hert

Title of thesis

Porting RIOT OS to the RP2350: An Exploration of a Heterogeneous Architecture and Programmable I/O

Keywords

RISC-V, RIOT OS, ARM, Embedded Systems, Operating System

Abstract

A recent development in the field of embedded systems is the emergence of heterogeneous architectures, which combine multiple types of processors on a single chip. The Raspberry Pi RP2350 is one such architecture, combining two ARM Cortex M33 and two Hazard3 RISC-V cores, along with a Programmable Input/Output (PIO) subsystem. Currently, RIOT OS, a popular operating system for embedded devices, does not support such architectures.

This thesis, explores the challenges and opportunities involved in porting RIOT OS to the RP2350. It focuses on understanding the architecture, implementing the necessary low-level support, and evaluating the advantages of such a system, including multicore processing in embedded applications.

Tom Hert

Thema der Arbeit

Portierung von RIOT OS auf den RP2350: Eine Untersuchung einer heterogenen Architektur und programmierbarer I/O

Stichworte

RISC-V, RIOT OS, ARM, Eingebettete Systeme, Betriebssystem

Kurzzusammenfassung

Eine aktuelle Entwicklung in der Welt der eingebetteten Systeme ist das Vorhandensein heterogener Architekturen, die mehrere Prozessortypen auf einem einzigen Chip

kombinieren. Der Raspberry Pi RP2350 ist eine solche Architektur, die zwei ARM Cortex M33- und zwei Hazard3 RISC-V-Kerne sowie ein PIO Subsystem kombiniert. Derzeit werden solche Architekturen in RIOT OS, einem beliebten Betriebssystem für eingebettete Geräte, nicht unterstützt.

Diese Arbeit untersucht die Herausforderungen und Chancen der Portierung von RIOT OS auf den RP2350, wobei der Schwerpunkt auf dem Verständnis der Architektur, der Implementierung der erforderlichen Low-Level-Unterstützung und der Bewertung der Vorteile eines solchen Systems liegt. Dabei ist auch ein Fokus auf die Möglichkeiten die Multi-Core Verarbeitung in eingebetteten Anwendungen bietet.

Contents

List of Figures

List of Tables

Listings

Abbreviations

CLIC **– Core Local Interrupt Controller**

ISR **– Input Shift Register**

LPOSC **– Low Power Oscillator**

NVIC **– Nested Vectored Interrupt Controller**

OSR **– Output Shift Register**

PLIC **– Platform Level Interrupt Controller**

ROSC **– Ring Oscillator**

XOSC **– Crystal Oscillator**

board **– MCU Development Board**

IoT **– Internet of Things**

IRQ **– Interrupt Request**

ISA **– Instruction Set Architecture**

ISR **– Interrupt Service Routine**

LR/SC **– Load-Reserved/Store-Conditional**

MCU **– Microcontroller (Unit)**

OS **– Operating System**

OTP **– One-Time Programmable**

PIO **– Programmable Input/Output**

PMP **– Physical Memory Protection**

PSM **– Power State Machine**

RISC **– Reduced Instruction Set Computing**

1 Introduction

1.1 Motivation

In recent years, ARM has dominated the embedded systems industry, offering relatively fast Microcontroller Units (MCUs) such as the Cortex-M series, at comparatively low prices.

The emergence of RISC-V has challenged that dominance by offering an open alternative, which allows anyone to design and manufacture their own RISC-V-based MCUs without paying licensing fees to ARM. This has led to a surge of innovation in the field of MCUs, with many new designs and architectures being developed [2].

To ease developers into the usage of RISC-V based devices, Raspberry Pi released the RP2350 MCU. This MCU combines the legacy and wide adoption of the ARM Cortex-M33 with the flexibility of the new RISC-V architecture using a Hazard3 open-source core [3].

This heterogeneous dual-core design is unified through an MCU architecture that emphasizes a shared environment, including common board peripherals, memory, and programmable I/O (PIO) blocks. This makes the RP2350 a unique platform for experimenting with heterogeneous architectures in the embedded systems world.

RIOT OS is an open-source operating system designed for low-end IoT devices. It is known for its modularity, efficiency, and broad hardware support. It is designed to be hardware-agnostic and portable across different architectures and boards [4]. Still, RIOT OS currently does not support MCUs with heterogeneous architectures such as the RP2350.

1.2 Objective

This thesis explores the process of porting RIOT OS to the RP2350 MCU, leveraging its unique dual-core architecture to enhance the capabilities of RIOT. The goal is to implement a functional port that allows RIOT OS to run seamlessly on the RP2350, taking advantage of its heterogeneous architecture while maintaining the modularity and efficiency that RIOT OS prides itself on [4].

The main objective of this work is to create a unified abstraction layer for both architectures that allows seamless switching between RISC-V and ARM with minimal code redundancy. This entails exploring methods of integrating with the existing codebase of RIOT while also conforming to the unique peculiarities of the RP2350, such as the custom interrupt controller which the Hazard3 RISC-V processor includes.

In this thesis, we will also take a first glance at multicore processing within RIOT OS, exploring how a heterogeneous dual-core architecture can be utilized in an embedded operating system context. The objective of this thesis is to have a functional RIOT OS port for the RP2350 that can serve as a foundation for future work and exploration of heterogeneous architectures in embedded systems.

1.3 Outline

Section 2 provides the relevant background information on the RP2350 architecture, and relevant concepts, such as heterogeneous architectures and programmable I/O, as well as its multicore processing. It also gives an introduction to the RIOT operating system and design principles.

A review of related work in Section 3 follows next. In it, both related academic work and existing implementations of the RP2350 on other operating systems and libraries are discussed, with differences and similarities in our approach and goals being explained.

In Section 4 the thesis analyzes the RP2350 in more detail to explore the requirements and design considerations that are relevant for the porting process. We examine the boot process, multicore startup sequence, and interrupt system in detail. We also explore RP2350-specific details such as the picobin image format and Hazard3 custom extensions.

After diving into these details, we then describe the implementation of the port in Section 5. Detailing the steps taken to implement low-level support for the RP2350 architecture, including clock configuration. Describing the approach that was taken to

implement multicore support and a unified abstraction for both architectures. We also discuss our approach to integrating the RP2350 interrupt controller with the existing RIOT interrupt handling system.

In Section 6, we evaluate the functionality and performance of the RIOT OS port on the RP2350. Showcasing the benefits that a second core can bring to an embedded operating system. We also compare differences in performance and size between the ARM and RISC-V cores when running RIOT-OS.

Finally, Section 7 wraps up the thesis by summarizing the key findings and contributions of this work. We reflect on the challenges faced during the porting process and how they were addressed. We also discuss the implications of our work for the future of RIOT OS and heterogeneous architectures in embedded systems. Followed by a discussion of potential future directions and improvements that can be made based on the work of this thesis in Section 8.

2 Background

2.1 RISC-V

RISC-V is an open-standard Instruction Set Architecture (ISA) based on established principles of Reduced Instruction Set Computing (RISC) [5].

The RISC design philosophy aims to simplify the processor design by using a small set of simple and general instructions. This allows for easier implementation, lower power consumption, and higher performance by moving complexity from the hardware to the software, such as compilers and assemblers that can optimize instruction usage.

RISC-V expands on this concept by being open and extensible, allowing anyone to design, manufacture, and sell RISC-V processors without any licensing agreements. This has led to a wide range of adoptions, from small MCUs to high-performance processors, including the Hazard3 open-source core used by the RP2350.

RISC-V is still a relatively new architecture compared to architectures such as ARM and x86. However, it has gained significant traction in recent years, experiencing a 276.8% growth from 2022 to 2023 with market analysts such as the SHD Group forecasting continuing rapid growth over the next decades [6].

2.2 ARM-M

The ARM-M family comprises RISC processors designed by ARM Holdings plc. Unlike RISC-V, ARM is a proprietary architecture, requiring companies to license the technology from ARM Holdings plc when used in their products[1].

[1]ARM licensing information can be found here (Accessed 30.10.2025): https://www.arm.com/products/licensing

The ARM-M family is designed for low-power, cost-efficient processors, making it ideal for embedded systems and IoT devices. It features a simplified instruction set, a limited number of registers, and other optimizations intended for embedded systems.

The ARM-M architecture also includes various security features, such as TrustZone technology, which allows for the creation of secure and non-secure execution environments, as has been explored for RIOT OS in "Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS"[7].

While RISC-V is gaining traction in the embedded systems industry [6], ARM-M remains a dominant architecture for low-power embedded systems due to its maturity, extensive ecosystem, and wide range of available tools and libraries, including the Cortex-M series of processors, such the RP2350 Cortex-M33 core.

2.3 RIOT OS

RIOT OS is an open-source Operating System (OS) for low-end embedded devices in the Internet of Things (IoT). It is vendor-neutral and lightweight on as little as 3.2 kB of ROM and 2.8 kB of RAM under minimal configurations [4].

The focus of RIOT on modularity and modifiability facilitates the easy integration of new features, including new MCUs and boards. Accompanied by a comprehensive set of tutorials and documentation to help new users get started with the OS[2].

RIOT also offers a vast number of packages that can be utilized by newly ported boards after the initial setup. Currently, RIOT does not support heterogeneous architectures or multicore systems.

RIOT does, however, support both ARM and RISC-V architectures, including a comprehensive abstraction for architecture specific common code, which makes it a good candidate for the RP2350.

2.3.1 RIOT OS Support for New MCUs

RIOT OS adopts a modular approach to peripheral and module support. Each peripheral or module is implemented as a separate driver that can be included or excluded depending on the target board or MCU [4, Chapter 6].

[2]RIOT OS documentation can be found here (Accessed 30.10.2025): https://guide.riot-os.org/

The minimal support level RIOT OS requires of any added MCU is a bootable system capable of running a basic application, preferably including threading, though under special circumstances, a single-threaded system is also acceptable [4, Chapter 5].

From there, additional peripherals and most essential functionality can be added incrementally, such as interrupts, timers, GPIO, and UART.

2.3.2 RIOT Principles

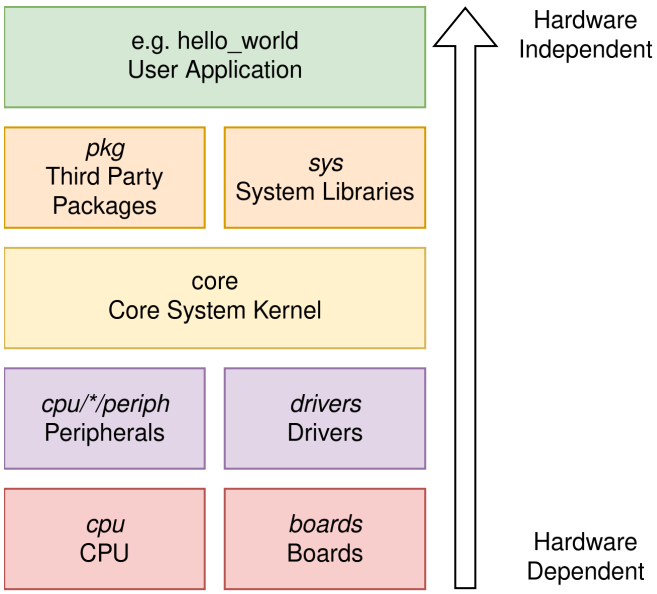

Figure 1: RIOT OS modular architecture showcasing the kernel, drivers, and applications. Going from highly hardware-dependent modules to hardware-agnostic ones.

RIOT separates hardware-dependent code from hardware-agnostic code through a layered architecture where each layer only interacts with the layer directly above or below it, rarely crossing layers as seen in Figure 1.

In essence, hardware-dependent code is limited to the MCU `cpu` drivers and `boards` board. The MCU driver, depending on the MCU and board, provides access to peripherals `periph` such as GPIO, timers, UART, SPI, and I²C. The board file then maps these peripherals to physical pins and configures any board-specific settings, such as UART baud rate or LED active high/low.

These components use a common peripheral API `drivers` / `periph` defined by RIOT OS. This API enables hardware-agnostic code, such as network stacks, file systems, and

applications, to use these peripherals without knowledge of the underlying hardware [4, Chapter 6c].

Another useful side effect of this modularity is that swapping out hardware becomes easier. Provided the new hardware has a RIOT OS MCU driver and a board file, the remainder of the system usually remains unchanged.

RIOT also offers third-party packages through the `pkg` directory. These packages can be added to a RIOT OS project through a simple `USEPKG` directive in the `Makefile`, enabling the integration of new functionality without the need to modify the core RIOT OS codebase. Examples include libraries, such as `lvgl` for graphical user interfaces, `micropython` for Python scripting support, or `tinyusb` for USB support[3].

First-party packages are also modular and can be included from the `sys` directory. This includes essential functionality such as networking, file system, and cryptography.

The `core` directory contains the RIOT OS kernel and essential services such as the scheduler, memory management, and inter-process communication. This layer is hardware-agnostic and can run on every supported MCU[4].

RIOT ensures that all the aforementioned layers are well tested through a comprehensive suite of unit and integration tests. This testing helps to maintain the stability and reliability of the system as new features and MCU support are added, and existing components are modified[5].

2.4 RP2350

2.4.1 RP2350 Overview

The RP2350 is a low-cost MCU developed by Raspberry Pi. The Raspberry Pi Pico 2 serves as the reference board for the RP2350. Throughout this thesis, the terms *RP2350* and *Raspberry Pi Pico 2* are used interchangeably.

The RP2350 features both a dual-core ARM Cortex-M33 and a dual-core Hazard3 RISC-V processor, which can be switched between while retaining full access to peripherals and memory. Both processors run at 150MHz on the Pico 2 board [3, Chapter 1]. The

[3] A comprehensive list of available packages can be found here (Accessed 30.10.2025): https://github.com/RIOT-OS/RIOT/tree/master/pkg

[4] More information about the RIOT OS structure can be found here (Accessed 30.10.2025): https://guide.riot-os.org/general/structure/

[5] CI can be accessed here (Accessed 29.10.2025): https://ci.riot-os.org/details/branch/master

Pico 2 board includes 520 kB of SRAM, 4 MB of onboard QSPI flash, two UARTs, two SPI controllers, two I2C controllers, 24 PWM channels, 26 GPIO pins, and three PIO subsystem blocks, each supporting four state machines [8].

The RP2350 is the first heterogeneous architecture developed by Raspberry Pi [9]. It succeeds the RP2040, which includes dual ARM Cortex-M0+ cores, 264kB of SRAM, and the first version of PIO [10]. RIOT OS already includes support for the RP2040, however, the RP2350 drastically changes the architecture by introducing RISC-V cores and a more advanced Cortex-M33 core, thus requiring a new port.

The RP2350 is designed for low-power applications and is suitable for use in a wide range of embedded systems, including IoT devices, wearables, and home automation systems. At the time of writing, three public revisions of the RP2350 exist: `RP2350 A2`, `RP2350 A3`, and `RP2350 A4`, released in July 2025. These mostly contain bug fixes and security improvements [3, Appendix C]. This thesis is based on revision `RP2350 A3` of the RP2350, as revision `A4` was released after the initial research phase.

2.4.2 Hazard3

Hazard3 is a three-stage pipelined RISC-V processor used by Raspberry Pi in the RP2350 MCU. It was designed by Luke Wren and is open-source [16]. The Hazard3 includes various extensions that introduce new CSRs and instructions, as listed in Table 1.

Although the Hazard3 is designed with modularity in mind, this thesis assumes that all of the above extensions are present, given that they are all implemented by the RP2350 [3, Chapter 3.8].

2.4.2.1 Physical Memory Protection (PMP)

Physical Memory Protection (PMP) is a security feature of RISC-V that allows the definition of memory regions with specific access permissions [14, Chapter 3.7].

Although the Hazard3 supports 16 PMP regions [1, Chapter 3.3], the RP2350 implementation is configured for only eight PMP regions at 32-byte granularity, followed by three hard-wired regions [3, Chapter 10.4].

The RIOT implementation of PMP follows the ISA specification, where only 16 or 64 regions are supported [17, Chapter 2.2.4].

2.4.3 Cortex-M33

The Cortex-M33 is the first three-stage pipelined Armv8-M based processor and stands as one of the more powerful ARM MCUs [18].

Extension	Description
RV32I v2.1	Base integer instruction set with 32-bit registers [5]
M v2.0	Integer multiplication and division instructions [5]
A v2.1	Atomic memory operations [5]
C v2.0	Compressed 16-bit instructions for reduced code size [5]
Zicsr v2.0	CSR read/write instructions [5]
Zifencei v2.0	Instruction-fetch fence for self-modifying code [5]
Zba v1.0.0	Address generation bit manipulation instructions [11]
Zbb v1.0.0	Basic bit manipulation instructions [11]
Zbc v1.0.0	Carry-less multiplication instructions [11]
Zbs v1.0.0	Single-bit manipulation instructions [11]
Zbkb v1.0.1	Bit manipulation for cryptography [12]
Zcb v1.0.3-1	Code size reduction with additional compressed instructions [13]
Zcmp v1.0.3-1	Push/pop and double move compressed instructions [13]
Machine ISA v1.12	Machine-mode privileged instructions [14]
Debug v0.13.2	External debug support [15]
Xh3bextm	Custom bit extraction multiple instructions (`h3.bextm`, `h3.bextmi`)
Xh3irq	Custom interrupt controller
Xh3pmpm	Custom CSRs for M-mode Physical Memory Protection (PMP) enforcement
Xh3power	Custom power management with `msleep` CSR and hint instructions

Table 1: RISC-V extensions supported by Hazard3

The RP2350 supports both Secure and Non-Secure states through the ARM TrustZone technology [3, Chapter 3.7.2]. This thesis focuses on the Non-Secure state of the Cortex-M33, as RIOT does not have a merged integration of this technology [7], and the Hazard3 exclusively supports Non-Secure mode. Thus, the Non-Secure mode is the only common denominator between the two architectures.

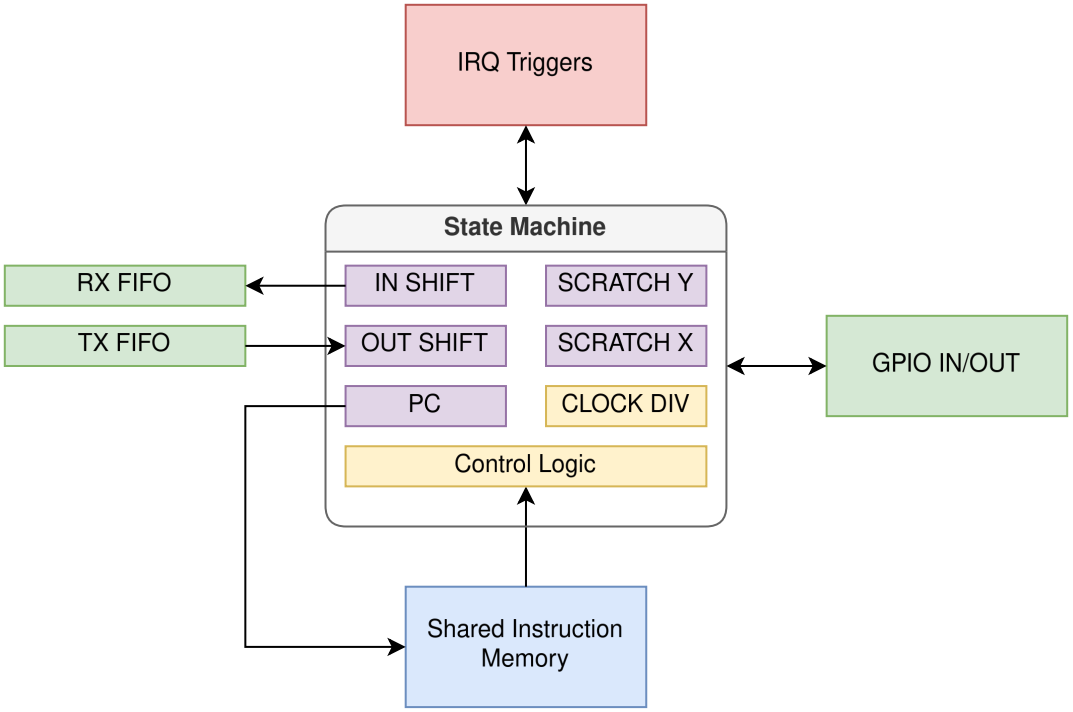

Figure 2: Overview of a PIO state machine. Showcasing the shared instruction memory, access to the FIFO buffer, and interrupts.

2.4.4 Programmable Input/Output (PIO)

Programmable Input/Output (PIO) is a distinctive feature of the Raspberry Pi Pico MCU family. It was first introduced in the RP2040 and has been updated in the RP2350 [3, Chapter 11.1.1].

The RP2350 contains three identical PIO blocks. Each block includes four state machines programmable in a custom assembly language. The state machines can operate independently or in parallel, allowing complex I/O operations to be offloaded from the main processors.

In total, the PIO assembly language has nine instructions, as explained in Table 2, that, when combined, allow for fairly complex operations, such as generating precise waveforms, handling serial protocols, or bit-banging custom interfaces. [3, Chapter 11.4].

Each state machine (Figure 2) can read and write to a FIFO buffer, which can be used to communicate with the main processors. In total, each state machine has eight 32-bit buses, by default configured as four inputs and four outputs. This design allows for flexible communication between the state machines and the main processors.

Instruction	Description
`JMP`	Jump to address if condition is true
`WAIT`	Stall until condition is met (GPIO/pin/IRQ/jmppin)
`IN`	Shift data from source into Input Shift Register
`OUT`	Shift data from Output Shift Register to destination
`PUSH`	Push ISR contents to RX FIFO
`PULL`	Pull data from TX FIFO into OSR
`MOV`	Move data between registers
`IRQ`	Set or clear IRQ flags
`SET`	Set pins or register to immediate value

Table 2: PIO assembly instructions

For high bandwidth operations, the RP2350 supports eight unidirectional 32-bit buses, allowing for eight input or eight output buses exclusively [3, Chapter 11.5.3].

Each state machine can also trigger and respond to interrupts. In total, there are eight IRQ flags shared among all state machines. State machines can both trigger and wait for these IRQ flags [3, Chapter 11.4.11].

In total, each state machine has four registers:

- The `X` and `Y` registers are general-purpose registers that can be used for arithmetic and logic operations.
- The Input Shift Register (ISR) and Output Shift Register (OSR) are used for serial data input and output operations.

The instruction memory is shared between all state machines in a block. Holding a total of 32 instructions per block.

PIO runs on the system clock. This would, however, be too fast for most I/O operations. To mitigate this, each state machine has a configurable clock divider that can be used to slow down the execution of instructions.

The clock divider modifies the number of clock cycles that count as one execution cycle of the state machine, instead of reducing the clock frequency [3, Chapter 11.5.5].

2.4.5 Inter-Processor Communication

The RP2350 features a few different mechanisms that enable synchronization and communication between its cores.

2.4.5.1 Spinlocks

The RP2350 includes 32 hardware spinlocks and an additional 32 software locks for Secure mode. Each spinlock is a single flag bit that can be set or cleared by either core. If a core tries to acquire a lock that is already held by the other core, it will spin in a loop until the lock is released [3, Chapter 3.1.4].

2.4.5.2 Atomic Memory Operations

The RP2350 supports atomic access to SRAM based on the Armv8-M Global Exclusive Monitor mechanism. The implementation covers nearly all atomic RISC-V operations as defined in the atomicity PMA specification, except for the Load-Reserved/Store-Conditional (LR/SC) `RsrvEventual` option [3, Chapter 2.1.6].

Load-Reserved/Store-Conditional (LR/SC) is a pair of instructions used in RISC-V to implement read-modify-write operations. The LR instruction loads a value from memory and marks the address as "reserved". The SC instruction attempts to store a new value to the same address, but only if it is still marked as reserved (i.e., no other writes have occurred to that address since the LR). If the store is successful, it indicates that the operation was atomic; otherwise, it fails, and the operation must be retried [5].

There are three support levels for LR/SC PMA reservability:

- `RsrvNone`: No LR/SC operations are supported (locations are not reservable)
- `RsrvNonEventual`: LR/SC operations are supported, but the reservation may be lost
- `RsrvEventual`: LR/SC operations are supported and guarantee eventual success

The RISC-V Privileged Architecture specification recommends support for `RsrvEventual` and states that `RsrvNonEventual` support should include fallback mechanisms when lack of progress is detected [14, Chapter 3.6.3.2].

Raspberry Pi justifies not supporting `RsrvEventual` by noting that while artificial scenarios without progress guarantees can be theoretically constructed, practical implementations with properly bounded atomic sequences typically complete quickly without requiring additional fallback mechanisms [3, Chapter 2.1.6].

2.4.5.3 Doorbell

The RP2350 features a core-local doorbell interrupt (identified as `SIO_IRQ_BELL` at IRQ 26) that can be triggered by either core or by itself. This mechanism enables event signaling between cores in scenarios where event ordering is not critical or where multiple events can be processed within a single interrupt handler [3, Chapter 3.1.6].

2.4.5.4 Inter-Processor FIFOs

The primary inter-processor communication mechanism consists of two hardware FIFOs, each 32 bits wide and four elements deep. Each FIFO is readable by one core and writable by the other. The FIFOs support interrupt generation when non-empty (for the reading core) or non-full (for the writing core) [3, Chapter 3.1.5].

These FIFOs are utilized by both the RP2350 bootloader and the multicore startup procedure, as discussed further in Section 5.3

3 Related Work

This chapter reviews existing work relevant to the implementation of RIOT OS support for the RP2350. It examines prior efforts in porting operating systems to embedded architectures, security analyzes of the RP2350 platform, and alternative operating systems and frameworks that support the RP2350. These works provide context for the design decisions and implementation approaches taken in this thesis.

3.1 Inferno OS on ARMv7-M

A relevant contribution in this field is the masters thesis "Porting Inferno OS to ARMv7-M and Cortex-M7" by Petter Duus Berven [19]. In that work, the author ported the Inferno operating system, a distributed non-real-time OS derived from Plan 9, to the ARMv7-M architecture used in Cortex-M MCUs.

Plan 9 is an operating system developed at Bell Labs in 1992, designed with a focus on distributed computing and simplicity [20]. In Plan 9 only the most core and essential components are part of the kernel[6], while most other functionality is run outside the kernel[7]. Inferno OS, similar to RIOT OS, allows a fairly modular design, with a small kernel.

The thesis focused on extending the custom compiler toolchain of Inferno to generate ARM Thumb instructions, implementing low-level hardware support for the Teensy 4.1 board, and adapting core kernel components. In his thesis, Berven highlights common challenges including incomplete compiler backends, limited instruction-set coverage, and the need to redesign low-level exception handling, memory layout, and startup code.

While Berven's project targeted a homogeneous ARM-based environment, this work is relevant to the current thesis as it provides insights into the complexities of porting an

[6]Called 'devices' in Plan 9 [20].
[7]Called 'servers' in Plan 9 [20].

operating system to a different architecture, which can inform the approach taken for porting RIOT OS to the RP2350.

3.2 Security through Transparency: Tales from the RP2350 Hacking Challenge

In "Security through Transparency: Tales from the RP2350 Hacking Challenge" [21], the authors discuss the security aspects of the RP2350 architecture that were found in the process of a hacking challenge organized by Raspberry Pi Ltd. They analyze various vulnerabilities and attack vectors, providing insights into the security challenges associated with heterogeneous architectures. In it, the authors discuss attacks on the One-Time Programmable (OTP) Power State Machine (PSM), vector boot and signature verification to bypass secure boot.

While the main focus of the paper is on exploring the security of the RP2350 and methods to defeat it, the paper provides a comprehensive overview of the underlying hardware and boot sequence of the RP2350, which aids in understanding the low-level initialization of the RP2350 when implementing RIOT OS support.

Although this thesis does not focus on security aspects, as discussed in Section 2.4.3, RIOT OS TrustZone support is out of scope and not yet merged into mainline RIOT OS, understanding the security features of the RP2350 remains important for future work building upon this thesis.

3.3 Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices

In "Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices" [17], Bennet Blischke explores the use of the RISC-V Physical Memory Protection (PMP) unit in the context of constrained IoT devices running RIOT OS. In his thesis, Blischke implements data execution prevention and thread stack overflow detection using the RISC-V PMP, evaluating its effectiveness and performance impact.

While this does not directly relate to the RP2350 porting effort, his work serves as a foundation for demonstrating the benefits of enabling RIOT OS support on the RP2350 platform, as it allows leveraging the RISC-V PMP features on the Hazard3 core with minimal additional effort.

One of the conclusion of this work is that most existing PMP implementations do not properly comply with the specifications, we will extend the findings of this work in Section 6 when evaluating the PMP implementation of the Hazard3 core in the RP2350.

3.4 ArielOS

In "Ariel OS[8]: An Embedded Rust Operating System for Networked Sensors & Multi-Core Microcontrollers," the authors present ArielOS a new operating system for embedded devices, written in Rust. It aims to provide a safe and secure environment for IoT applications, including support for multicore MCUs [22]. While still retaining much of the design philosophy of RIOT OS, ArielOS focus on multicore systems differentiates it from RIOT OS.

ArielOS includes support for the RP2350 from the beginning, making it an interesting point of comparison for this thesis. It was designed with the RP2350 and similar systems in mind [22, Chapter 1]. RIOT OS, on the other hand, was designed at a time when single-core 8 bit and 16 bit MCUs were fairly common in the embedded space [4, Section 2], which is something that ArielOS does not target or support.

Under the hood, ArielOS differs significantly from RIOT OS. It leverages the pre-existing Rust ecosystem for embedded systems, using libraries such as Embassy as building blocks for the operating system [22, Chapter 5]. This is in contrast to RIOT OS, which implements most of its functionality from scratch in C.

Embassy is an asynchronous runtime for embedded systems in Rust, providing abstractions for concurrency and hardware access[9]. In the case of the RP2350, ArielOS uses Embassy for the underlying hardware access, binding the implementation offered by the `embassy_rp` crate to its own abstractions[10].

Referring to Figure 1 as explained in Section 2.3.2, this means that compared to RIOT OS, ArielOS still offers abstraction layers, but the underlying implementation of `cpu`, `drivers`, and peripherals is provided by Embassy rather than being originated from the OS[11]. ArielOS then provides core system services such as multicore task scheduling, inter-

[8] ArielOS can be found here (Accessed 28.10.2025): https://github.com/ariel-os/ariel-os

[9] Embassy can be found here (Accessed 28.10.2025): https://github.com/embassy-rs/embassy

[10] embassy_rp can be found here (Accessed 30.10.2025): https://github.com/embassy-rs/embassy/tree/main/embassy-rp/

[11] The implementation of the RP2350 in ArielOS can be found here (Accessed 30.10.2025): https://github.com/ariel-os/ariel-os/tree/main/src/ariel-os-rp

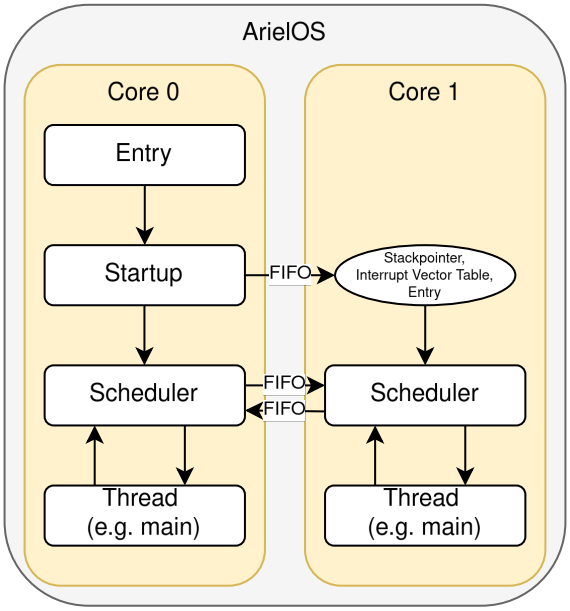

Figure 3: ArielOS Scheduler Architecture. After startup, Core 0 initializes the system and starts Core 1 via the FIFO (See Section 2.4.5.4). Then, both cores run the same scheduler, triggered through FIFO messages from either core to handle task scheduling.

process communication, and memory management on top of Embassy, similar to how RIOT OS builds its core services on top of its own hardware abstractions.

The scheduler in ArielOS is designed as a continuation of the tickless real-time scheduler of RIOT OS with preemptive priority scheduling, extended to support multicore systems. The exploration of multicore scheduling is further expanded in the original master's thesis "Multicore Scheduling and Synchronization on Low-Power Microcontrollers using Embedded Rust" by Elena Frank in which she explores the design and implementation of a multicore scheduler for `RIOT-rs`, which later evolved into ArielOS [23].

ArielOS utilizes a global scheduling approach, as shown in Figure 3, where tasks are not bound to a specific core (though they can have a core affinity if desired), together with a shared mutually-exclusive kernel design [22, Chapter 5D]. The authors argue that such a global scheduling approach is acceptable on IoT MCUs given the low number of cores and limited parallelism [22, Chapter 5B].

On the RP2350, it uses the same process to start both cores as described in Section 5.3. After the startup process, it uses the FIFO to pass scheduler invocations between the two cores [22, Chapter 6B]. Specifically, when the scheduler needs to be invoked in a multicore

system, ArielOS uses a global spinlock through the RP2350 FIFO to synchronize a global critical section for all cores [22, Chapter 5F].

ArielOS represents a different approach to supporting the RP2350 compared to the work done in this thesis as it modifies the concept of the RIOT OS scheduler to support multicore systems directly, rather than building around the existing RIOT OS scheduler design, which is inherently single-core. Nonetheless, given its similarity to the RIOT OS scheduler, ArielOS provides a valuable comparison for the multicore design decisions made in this thesis and how they can be improved in future work. ArielOS also demonstrates that supporting the RP2350 in an embedded operating system is feasible and can provide a solid foundation for further exploration of heterogeneous architectures in embedded systems.

3.5 Pico SDK

Pico SDK is the official software development kit for the Raspberry Pi Pico[12]. Compared to RIOT OS or other operating systems, the Pico SDK is designed solely for the Pico series, similar to other vendor SDKs such as esp-idf from Espressif[13].

The SDK aims to be a development framework rather than a full operating system, providing low-level access to the hardware and basic libraries for common tasks, but not including the benefits that come with a full operating system.

Given that this is a vendor SDK, it offers the widest support for RP2350 hardware features. It uses a CMake build system for building applications. The user must manually specify which modules to include in their application, similar to how RIOT OS allows users to select modules at compile time.

3.5.1 Abstraction

One of the core themes of this thesis is the abstraction of architectural differences between the ARM and RISC-V cores in the RP2350. The Pico SDK uses compile-time flags to differentiate between the two architectures.

It also shares a common abstraction layer between the RP2040 and RP2350, mostly sharing headers and higher-level libraries. The RP2040 was the predecessor of the RP2350, sharing most peripherals, though having a different purely ARM Cortex-M0+

[12]PicoSDK can be found here (Accessed 28.10.2025): https://github.com/raspberrypi/pico-sdk
[13]esp-idf can be found here (Accessed 28.10.2025): https://github.com/espressif/esp-idf

dual-core architecture. While not supporting all peripherals, RIOT OS does also offer support for the RP2040.

While this approach works well for a vendor SDK, it lacks the modularity and flexibility of RIOT OS. The Pico SDK approach is hardware-dependent by design, making it less suitable for applications that require portability across different architectures and boards.

Throughout the technical specification and documentation of the RP2350, the Pico SDK is often referenced to explain hardware details, making it an important resource when implementing support for the RP2350 in RIOT OS. This influenced various parts of the implementation described in Section 5.

3.6 ZephyrOS

In contrast to the Pico SDK, ZephyrOS is a full-fledged operating system for embedded devices, very similar to RIOT OS[14]. ZephyrOS is maintained by the Linux Foundation and has a large community of contributors.

While RIOT OS is historically largely developed in the context of academic research through volunteer contributions, ZephyrOS is backed by major industry players such as Google, Meta, ARM, Intel, Texas Instruments, Nordic, STMicroelectronics, and others[15].

This allows ZephyrOS to have vastly larger support for hardware platforms, architectures, and features compared to RIOT OS, supporting 881 boards as of October 2025[16].

While working on this thesis, ZephyrOS added support for the RP2350, including Hazard3 support, by the end of September 2025[17]. While this did not directly influence the work done in this thesis, it supported decisions made throughout the implementation process, as the approach in ZephyrOS aligned with the approach taken in this thesis regarding the abstraction of architectural differences between the ARM and RISC-V cores and handling the Hazard3 xh3irq interrupt controller (see Section 5.2).

[14]ZephyrOS can be found here (Accessed 30.10.2025): https://github.com/zephyrproject-rtos/zephyr

[15]ZephyrOS project members (Accessed 28.10.2025): https://www.zephyrproject.org/project-members/

[16]ZephyrOS supported boards (Accessed 30.10.2025): https://docs.zephyrproject.org/latest/boards/index.html

[17]ZephyrOS RP2350 Hazard support PR (Accessed 30.10.2025): https://github.com/zephyrproject-rtos/zephyr/pull/89758

4 Analysis and Design

In this chapter we analyze potential design considerations and requirements for porting RIOT OS to the RP2350 MCU.

This includes the bootup sequence, interrupt controller, clock system, and threading model. This gives us a good framework to then implement the necessary low-level support and ensure that RIOT OS can effectively utilize the capabilities of the RP2350.

4.1 Bootup Sequence

4.1.1 Bootrom

The RP2350 features a built-in bootrom that is executed on power-up [3, Chapter 5.2.2]. This bootrom is stored directly at `0x0` in a 32 kB immutable memory region and flashed onto the device during manufacturing [3, Chapter 4.1].

The bootrom has a wide range of responsibilities, from boot slot selection to rollback protection, intended for secure applications [3, Chapter 5]. However, for the context of this thesis, the most relevant parts of the bootrom are:

- Image and partition definition using picobin blocks
- Bootloader
- Architecture switching
- Core 0 boot code
- Core 1 launch preparations

4.1.2 Flashing

This thesis focuses on two primary flashing methods for the RP2350: OpenOCD and Picotool. These cover the most common use cases for flashing RIOT OS onto the RP2350, either through a SWD debugger or through USB mass storage device mode, thus offering flexibility for different development setups.

The RIOT build system abstracts the flashing process through the `PROGRAMMER` variable, allowing users to select their preferred flashing method through the build system, making it easy to switch between different flashing methods without dealing with the underlying implementation details.

An MCU must simply declare their supported programmers, and offer relevant config options, such as the OpenOCD target file.

4.1.2.1 OpenOCD / Debugprobe

OpenOCD (Open On-Chip Debugger) is a popular open-source tool for debugging and programming embedded devices using JTAG/SWD interfaces, including acting as a remote target for GDB[18].

The RP2350 allows for flashing and debugging through a SWD interface. Raspberry Pi recommends using their Debugprobe[19] or a secondary Raspberry Pi Pico 1/2 as a SWD debugger flashed with a debugprobe firmware[20] [24, Appendix A].

OpenOCD support for the RP2350 is, as of October 2025, still not integrated into any mainline OpenOCD release, requiring a custom build with RP2350 support[21]. RIOT OS already has support for OpenOCD as a flashing method, thus, as long as the custom OpenOCD build is used, no additional changes are required to support OpenOCD flashing for the RP2350.

4.1.2.2 Picotool

`picotool` is a custom command-line tool developed by Raspberry Pi for interacting with Raspberry Pi Pico series over USB[22]. RIOT already had support for `elf2uf2`, a predecessor to `picotool`, used for converting ELF files to the UF2 format used by the RP2040. `Picotool` support is thus a natural extension of this existing functionality and can reuse a significant portion of the existing integration codebase within RIOT OS.

UF2 (USB Flashing Format) is a file format developed by Microsoft designed for flashing MCUs over USB mass storage device mode. It allows users to simply drag and drop firmware files onto the device when it appears as a USB drive[23].

[18] OpenOCD official website (Accessed 03.11.2025): http://openocd.org/

[19] Debugprobe product link (Accessed 30.10.2025): https://www.raspberrypi.com/products/debug-probe/

[20] The debugprobe firmware can be downloaded here (Accessed 28.10.2025): https://github.com/raspberrypi/debugprobe

[21] The RP2350 OpenOCD fork can be found here (Accessed 23.10.2025): https://github.com/raspberrypi/openocd

[22] Picotool official repository (Accessed 30.10.2025): https://github.com/raspberrypi/picotool

[23] UF2 repository (Accessed 28.10.2025): https://github.com/microsoft/uf2

Upon selecting `picotool` as the `PROGRAMMER`, the RIOT build system will automatically clone and build `picotool`. It then converts the compiled binary into the UF2 format, verifies that the UF2 file is valid, and finally uses `picotool` to flash the UF2 file onto the RP2350 over USB.

4.1.3 Image and Partition Definition (Picobin)

4.1.3.1 Background

For the bootrom to load an application from flash, it needs to understand the specifics of the image in question, given the heterogeneous nature of the RP2350. To allow such specifications the bootrom uses picobin blocks [3, Chapter 5.1.4].

The block described in Table 3 should be placed within the first 4kB of flash memory. The bootrom will parse this block to determine which image to load based on the current architecture and security state of the core [3, Chapter 5.1.5.1].

A single binary can have multiple picobin blocks to support different architectures and security states. The `Next Block Pointer` field indicates the relative position.

Field	**Description**
Block Start Marker	Marks the beginning of a picobin block (Magic Value)
Item Type	Type of items in the block (e.g. Image Definition)
Item Size	Block size in words
Image Type Flags	• Bit 0-3: Image Type (e.g. Executable, Data) • Bit 4-5: Security (e.g. Secure, Non-Secure) Bit 6-7: Reserved • Bit 8-10: CPU Architecture (e.g. RISC-V, ARM) Bit 11: Reserved • Bit 12-14: Chip (e.g. RP2350, RP2040) • Bit 15: Try Before You Buy (TBYB) Image
Last Item Marker	Marks the last item in the block
Last Item Size	Size of last item in words
Next Block Pointer	Relative pointer to next block (0 = self = no more blocks)
Block End Marker	Marks the end of the picobin block (Magic Value)

Table 3: Picobin block structure for RP2350 image definition

4.1.3.2 Design Considerations

Incorporating a picobin block into the RIOT OS build process involves a few key design considerations. First, the build system must be able to generate the picobin block with the correct fields based on the target architecture and security settings. This includes setting the appropriate `Image Type Flags` to indicate whether the image is for ARM or RISC-V and then link that binary blob into the final firmware image.

To achieve this, we must first look into how RIOT OS handles the build process and most notably how it manages linker scripts for different architectures.

The CPU defines a custom linker script under the `ldscripts` directory. In the case of the ARM version of the RP2350, this file would be `ldscripts/rp2350_arm.ld`. However, since both the ARM and RISC-V versions use an even higher abstraction layer through the `cortexm_common` and `riscv_common` module respectively, the actually important linker script is provided by these common modules, thus `ldscripts/rp2350_arm.ld` simply uses the `INCLUDE` directive to include the relevant common linker script.

This, however, poses a challenge as the aforementioned common linker scripts do not provide any hooks for adding custom sections, such as the picobin block. As such, there are two different approaches to solve this problem:

- Modify the common linker scripts to include hooks for custom sections.
- Create a new linker script specifically for the RP2350 that includes the picobin block.

Since these common modules are used by multiple MCUs, modifying them could potentially introduce issues for other platforms. In the case of the `cortexm_common` module, the linker script is fairly lengthy and complex.

4.2 Interrupt Controller

Interrupt handling on the RP2350 is complex due to the heterogeneous design. Each architecture has its own interrupt controller, with the ARM cores using a NVIC and the Hazard3 cores using the XH3IRQ controller [3, Chapter 3.8.4.2].

To facilitate cross-architecture compatibility, the RP2350 keeps the identical Interrupt Requests (IRQs) numbers for both, including support for platform-specific interrupts on both architectures [3, Chapter 3.2].

The RISC-V Machine-mode timer interrupt `SIO_IRQ_MTIMECMP`, for instance, is a standard privileged interrupt for RISC-V RV32I. However, both on the Hazard3 and Cortex-M33, the IRQ is mapped to the value `29` and is functional [3, Chapter 3.1.8].

In total, the RP2350 defines 52 IRQ signals. The first 46 IRQ signals are connected to peripheral interrupt sources, while the remaining 6 are intentionally hardwired to 0 for forceful self-interrupts via software [3, Chapter 3.2].

4.2.1 Nested Vectored Interrupt Controller (NVIC)

NVIC is a nested vectored interrupt controller designed by ARM for their Cortex-M series of processors. It provides a flexible and efficient way to manage interrupts, allowing for prioritization and preemption of ISRs. It also handles context saving and restoring during interrupt handling. The NVIC design uses a vector table to map IRQ numbers to their corresponding ISR addresses, in which the first entries are reserved for system exceptions, followed by device specific interrupts [25].

On the Cortex-M33, the NVIC allows up to 480 interrupts to be managed with a preemption level of 0 to 255, whereby a lower level signals a higher priority [26]. As opposed to the custom XH3IRQ controller on the Hazard3 core, the NVIC on the RP2350 Cortex-M33 follows the standard ARM design without any modifications.

4.2.2 XH3IRQ Controller

To minimize architectural differences, the Hazard3 core includes an interrupt controller extension called XH3IRQ. This extension adds a new set of Control and Status Registers (CSRs) and instructions to the core that enable an interrupt handling mechanism similar to ARM NVIC.

To facilitate this, the XH3IRQ controller adds 6 custom CSRs, as described in Table 4.

CSR	Description
`meiea`	Machine External Interrupt Enable Array (enables/disables interrupts)
`meipa`	Machine External Interrupt Pending Array (status of interrupts)
`meifa`	Machine External Interrupt Force Array (force interrupts)
`meipra`	Machine External Interrupt Priority Array (priority levels for interrupts)
`meinext`	Machine External Interrupt Next (pointer to next highest priority pending interrupt)
`meicontext`	Machine External Interrupt Context (Saves/informs about context during interrupt handling)

Table 4: XH3IRQ custom CSRs for interrupt management [1, Chapter 4.1]

The XH3IRQ controller handles enabling, status, priority, and forced pending through a window system, as shown in Table 5.

Bits	Name	Description
31:16	window	16-bit read/write window into the external interrupt array (1 bit per interrupt)
15:5	-	Reserved
4:0	index	Write-only, self-clearing field (no value is stored) used to control which window of the array appears in the window

Table 5: XH3IRQ CSRs register fields for Interrupt Pending Array (`meipa`), Interrupt Enable Array (`meiea`), Force Interrupt Array (`meifa`)

This window system, described in Table 5 allows the XH3IRQ controller to manage 512 interrupts while only using a 32-bit CSR. Thus, at 1 bit per interrupt, each window can manage 16 interrupts with 32 total windows [1, Chapter 3.8.1].

Bits	Name	Description
31:16	window	16-bit read/write window into the external interrupt array (4 bits per interrupt)
15:5	-	Reserved
6:0	index	Write-only, self-clearing field (no value is stored) used to control which window of the array appears in window

Table 6: XH3IRQ CSRs register fields for the Interrupt Priority Array (`meipra`)

To allow 16 preemption levels, the interrupt priority array CSR uses a 7-bit index instead with a 4-bit value per interrupt, as described in Table 6. Thus, each window can manage 4 interrupts with a total of 128 windows [1, Chapter 3.8.4].

The XH3IRQ controller supports two operational modes: direct and vectored. In direct mode, the interrupt handler address is fixed, and all interrupts jump to the same handler. In vectored mode, each interrupt can have its own handler address [1, Chapter 4.1].

The XH3IRQ controller also includes a context-saving mechanism that allows the current execution context to be saved and restored when handling interrupts. This is done using the `meicontext` CSR and can optionally be enabled [1, Chapter 3.8.6].

Once the context is saved, the interrupt handler can be executed. After the handler is finished, the context can be restored and execution can continue from where it was interrupted through a `mret` call after completing the ISR [1, Chapter 3.2.9].

This is similar to the way ARM NVIC handles context saving and restoring during interrupt handling, as both controllers automatically save the execution context when an interrupt occurs, allowing the processor to jump to the interrupt handler. After the handler completes, the original context gets restored, and execution resumes from the point of interruption. Additionally, both support preemption priorities, for which higher-priority interrupts can interrupt lower-priority ones, ensuring critical tasks are handled promptly.

4.2.3 Design Considerations

When designing the interrupt handling for the RP2350 port in RIOT OS, the goal is to create a unified abstraction layer that could handle interrupts for both architectures seamlessly. At the same time though, conforming to existing interrupt handling mechanisms of RIOT for both architectures. Specifically, this means that we first needed to understand the existing interrupt handling mechanisms for both architectures in RIOT OS.

4.2.3.1 Cortex-M Interrupt Handling

The `cortexm_common` module in RIOT already includes support for the NVIC, thus the design considerations for the RP2350 port were mostly about ensuring that the implementation we intend to provide for the Hazard3 XH3IRQ controller conforms to the existing design patterns used in the `cortexm_common` module, at least in a way that allows architecture-agnostic code to work seamlessly across both architectures.

In RIOT the NVIC implementation uses a macro-based approach, in which the cpu module provides an extension to the interrupt vector table. Each vector that should be included in the final vector table gets a `.vector` label assigned using the `__attribute__((used,section(".vectors." # x )))` attribute. These attributes are then collected at the linking stage using a `KEEP(*(SORT(.vectors*)))` and sorted based on the assigned `x` value.

This way `cortex_common` ensures that common Cortex-M IRQ handlers can be defined in a platform-agnostic way, while still allowing platform-specific handlers to be defined in the respective cpu module.

4.2.3.2 RISC-V Interrupt Handling

The interrupt handler of the `riscv_common` module functions in a fairly different way compared to the `cortexm_common` module. The `riscv_common` module uses a single trap handler function that manages all interrupts and exceptions, opposed to the direct vector table jumps typical to NVIC. Depending on the enabled systems, such as Platform Level Interrupt Controller (PLIC) or Core Local Interrupt Controller (CLIC), the trap handler then passes the interrupt to the relevant sub-handler.

Given that the XH3IRQ controller needs a custom handling mechanism and the Cortex-M does not use a custom interrupt controller, the design consideration here was to implement the XH3IRQ handler in a way that fits into the existing trap handler mechanism of the `riscv_common` module while still allowing the RP2350 interrupt vector to be defined similarly to the vector table that the NVIC uses for direct jumps.

4.2.3.3 Abstracting

Thus, the final design proposal for the interrupt handling abstraction uses an interrupt vector table similar to the one used by the NVIC on the ARM side, while on the RISC-V side the trap handler function checks whether the interrupt originated from the XH3IRQ controller and then looks up the relevant handler in the vector table to call, as shown in Figure 4.

While this does introduce some overhead that the XH3IRQ controller theoretically could avoid through direct vector jumps (See Section 4.2.2), this design allows seamless integration into the existing interrupt handling mechanisms of RIOT for both architectures while still allowing architecture-agnostic code to define ISR handlers in a unified way. The trap handler of the RISC-V implementation also goes beyond the scope of direct vector jumps the XH3IRQ controller handles, as it also deals with the scheduler, ecalls, faults and context switching.

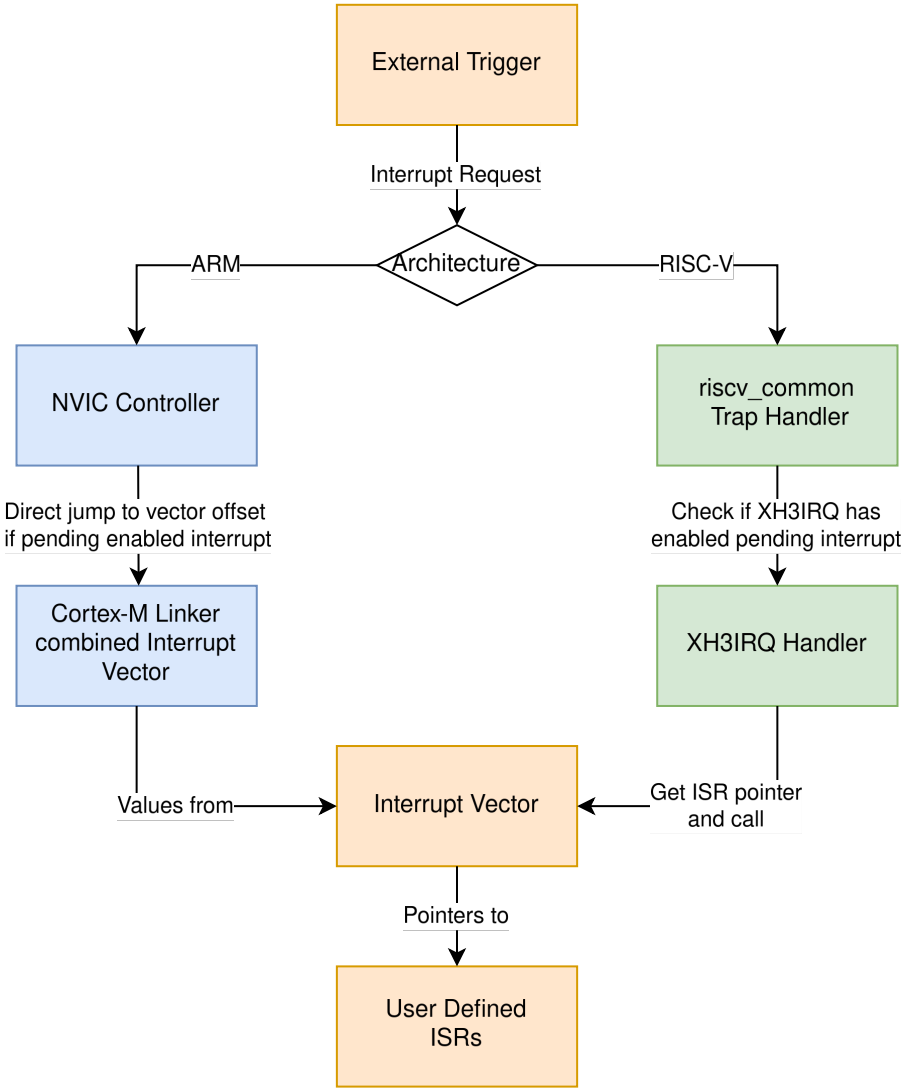

Figure 4: Diagram showing the design proposal for the route a hardware interrupt takes through the abstraction layers. Starting from the external trigger to the final user-defined ISR handler. Orange boxes are shared/common, blue boxes are ARM, green boxes are RISC-V.

Thus, the alternative of replacing the entire RISC-V interrupt handling mechanism within RIOT with a direct vector jump system for the RP2350 would have introduced significant complexity and maintenance burden of two competing interrupt handling mechanisms that would both need to be maintained in the future.

4.3 Clocks

The RP2350 provides a flexible clocking system that allows for multiple clock sources and configurations. The main internal clock sources are the ROSC, XOSC, and LPOSC [3, Chapter 8.1.2].

These clock sources then get routed through a series of dividers to allow for a wide range of clock frequencies for internal components, such as the system clock used for processors and memory, the peripheral clock used by UART and SPI, or the reference clock used by the watchdog and timers [3, Chapter 8.1].

4.3.1 Ring Oscillator (ROSC)

On startup, the ROSC is used as the main clock source. Since hardware revision `RP2350 A3` the ROSC operates at a randomized frequency on each power cycle to improve glitching attack resistance. The intended nominal frequency provided to the reference clock by the ROSC is 11 MHz, but due to the aforementioned randomization, it can vary largely. Without randomization, the RP2350 guarantees a speed in-between 4.6 MHz and 19.6 MHz, depending on the operating voltage and temperature.

On revision `RP2350 A2` ROSC is set to a randomized frequency between 4.6 MHz and 24.0 MHz. The `RP2350 A3` and later revisions quadruple the ROSC frequency by reducing the divisor of the system clock to 2. This increases the standard range of the system clock to 18.4 MHz to 96.0 MHz. Given that this is substantially higher than the nominal frequency of 11 MHz, `RP2350 A3` and later revisions increase the divisor of the reference clock to compensate [3, Chapter 8.3.1].

Due to the volatility of the ROSC frequency, it is not suitable for applications that require a stable clock source. Therefore, while not technically required, Raspberry Pi recommends to switch to the XOSC after the initial boot sequence [3, Chapter 8.3.4].

4.3.2 Crystal Oscillator (XOSC)

The XOSC on the RP2350 uses an external 12 MHz `ABM8-272-T3` ceramic SMD crystal to provide a stable clock source. This is the recommended clock source for most applications, especially those that require precise timing [3, Chapter 8.2.1]. It should be noted that the RP2350 has a specified XOSC support range of 1 MHz to 50MHz if a different crystal is used [3, Chapter 8.2.1].

To allow XOSC to stabilize, it is advisable to wait for at least 1 ms after enabling it before switching the system clock to it. This can be done through a specialized startup delay timer set within the `CTRL_ENABLE` register [3, Chapter 8.2.3].

4.3.2.1 XOSC Counter

The XOSC COUNT register is relevant to this thesis as it allows for accurate hardware-based delays by counting the number of XOSC cycles. Given the stable 12 MHz frequency of the XOSC, this allows for precise timing without relying on software-based delays that can be affected by interrupts [3, Table 603].

4.3.3 Low Power Oscillator (LPOSC)

To enable low power operation while the core is dormant, the RP2350 includes a low power oscillator (LPOSC) running at a nominal 32.768 kHz. Compared to the XOSC, there is no configuration required to use the LPOSC. When the system detects that the XOSC is powered down for low power operations, it will automatically switch to the LPOSC to keep the always-on logic running [3, Chapter 8.4].

4.3.4 Design Considerations

When designing the clock system support for the RP2350 port in RIOT OS, the startup flow needed to be considered carefully. After evaluating all available clock sources, we decided to implement the clocks as shown in Figure 5.

IoT devices are often used in scenarios where battery and by that power consumption are a limiting factor. In works such as “Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices” by Michel Rottleuthner [27], it has been shown that energy awareness can significantly improve the battery life of IoT devices. In the work, the authors propose an energy-aware design that allows the system to adapt its performance based on the current energy budget. This includes dynamically adjusting the clock speed to balance performance and power consumption. While an implementation as presented in the work is out of scope for this thesis, allowing for the user to easily change the clock speed is relevant and important for the RP2350 port in RIOT OS and lays the groundwork for future energy-aware designs.

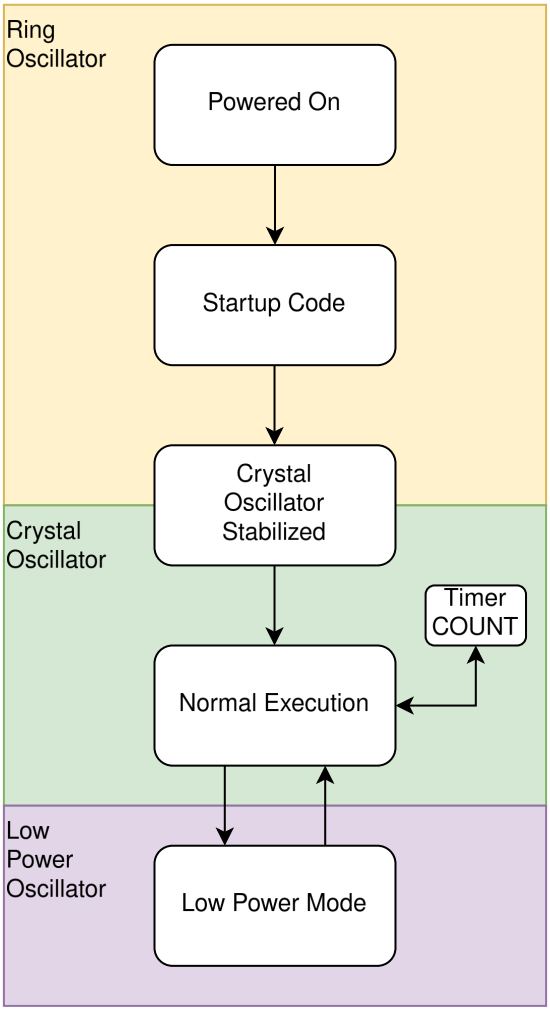

Figure 5: Proposed clock startup sequence for RP2350 port in RIOT OS. First, while running via ROSC, the XOSC is enabled. After a delay to allow it to stabilize, the system clock is switched to the XOSC for stable operation.

4.4 Multi-Core Support

4.4.1 Background

When RIOT was initially designed, it was built around the concept of a single core with a few MHz of processing power [4, Chapter 2].

The RP2350 and most other modern IoT MCUs however, significantly exceed these initial assumptions. Thus, in order to make use of these new capabilities, we first must

understand how RIOT handles threading and scheduling in its current form and then adapt our approach accordingly.

RIOT uses a fixed-priority fixed-preemption scheduling model. Each thread is only interrupted through IRQs, otherwise, threads execute to completion [4, Chapter 5b]. The main approaches to multi-core scheduling in the IoT OS field are global scheduling and partitioned scheduling. In global scheduling, one singular task queue is shared across all cores, and tasks are distributed onto available cores. In partitioned scheduling each core has its own task queue, and tasks are assigned to specific cores [22, Chapter 2].

ArielOS adapts this scheduling model to a multi-core environment by implementing a global scheduler (a single scheduler managing threads across all cores) that can distribute tasks across both cores, as explained in Section 3.4.

While changing the scheduler of RIOT to a similar design as in Figure 3 is theoretically possible, scheduler modifications were avoided in the design process. Fitting such a critical code change into the scope of this thesis exceeded the scope, given the complexity of multi-core scheduling and integrating it into the existing architecture of RIOT.

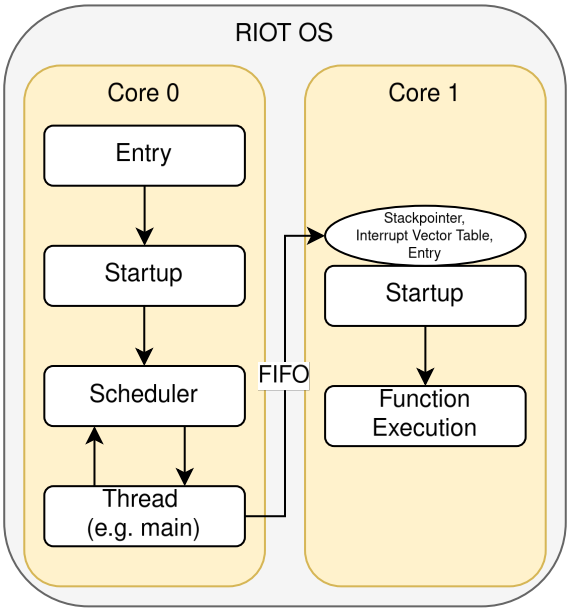

Figure 6: Proposed “Worker Core” multi-core model for RIOT OS. The main core (Core 0) offloads specific tasks to the secondary core (Core 1) which runs them independently without any scheduler intervention.

4.4.2 Design Proposal

Thus, a method was required to start both cores and have them run independently without any scheduler intervention. To achieve this, the secondary core is effectively isolated from the main RIOT OS environment as a worker core. It executes only what is directly assigned to it through inter-core communication mechanisms, as shown in Figure 6.

Naturally, this design does come with drawbacks compared to Figure 3 (See Section 3.4) would not have. The user is forced to design their application around this limitation, compared to a scheduler-based approach where the user can simply trust the scheduler to distribute tasks across both cores.

On the other hand, this design significantly reduces the complexity and maintenance burden of the implementation, as the entire multi-core logic can be contained within the RP2350 cpu module, which was the deciding factor for this design choice.

4.5 Programmable Input/Output (PIO)

The official pico sdk uses a `pioasm` assembler tool to assemble PIO instructions. Since RIOT aims to be vendor neutral, integrating the `pioasm` tool directly into RIOT is not ideal. However, any user wanting to use the `pioasm` tool should still be able to do so easily. The output format of the `pioasm` tool itself is however not compatible with RIOT as it also produces additional functions that rely on the Pico SDK. However, for our use case, we only require the raw assembled binary data to be loaded into the PIO memory.

Given the relatively small amount of instructions that can be stored in the PIO memory (32 instructions per state machine), it can be reasoned that programming PIO using C macros is feasible for most use cases. Thus, we propose a design where the user can define their PIO programs using C macros that directly encode the required instructions into binary data, as shown in Figure 7. The developer would then simply manually execute the required setup at runtime to load the assembled binary into the PIO memory and launch the state machines [3, Chapter 11.2.1].

One notable design goal with this is that PIO should integrate into the existing RIOT GPIO driver abstractions so the user can easily switch between using standard GPIO pins and PIO-controlled pins without changing their application code significantly and reducing code duplication, thus increasing maintainability. The PIO support we aim to provide is meant as a foundational layer for future more advanced PIO abstractions, such

as a dedicated PIO driver that can manage state machines, load programs, and handle interrupts in a more user-friendly way.

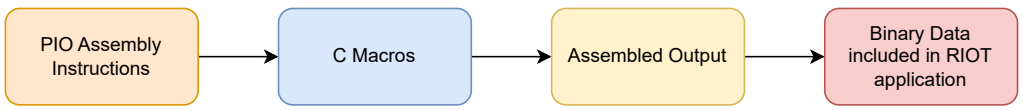

Figure 7: Proposed design for integrating `pioasm` into the RIOT build system. The `pioasm` tool is built from the Pico SDK and then used to assemble PIO assembly files into raw binary data that can be included in the RIOT build process.

5 Implementation

In this chapter, we discuss the implementation of the RP2350 port in RIOT OS. Going through the various notable components that were implemented, including the module setup, interrupt handling, multicore support, and clock configuration.

The full source code of the implementation can be found on GitHub for Picotool[24], ARM support[25], Multicore[26] and RISC-V support[27].

5.1 Module Setup and Abstraction

RIOT OS already has support for both ARM and RISC-V architectures in the form of shared common folders.

In order to avoid code duplication, the RP2350 includes a secondary common abstraction module. This module, referred to as `rp2350_common`, contains nearly all the code that is shared between both architectures. This includes peripheral drivers, riot-specific definitions, and general initialization functions.

5.1.1 Module Structure

The standard procedure of adding new MCUs or boards to RIOT OS is to create a CPU-specific module within the `cpu` directory and a board specific module within the `boards` directory. These modules then include the common code from the `common` directory. They can also offer peripheral support from the `periph` directory, such as GPIO or UART drivers.

[24]The picotool integration PR (Accessed 21.10.2025): https://github.com/RIOT-OS/RIOT/pull/21269

[25]The ARM RP2350 port PR (Accessed 21.10.2025): https://github.com/RIOT-OS/RIOT/pull/21545

[26]The Multicore code (Accessed 21.10.2025): https://github.com/AnnsAnns/RIOT/blob/pico2_riscv/cpu/rp2350/core.c

[27]The RISC-V and Interrupts RP2350 port PR (Accessed 21.10.2025): https://github.com/RIOT-OS/RIOT/pull/21753

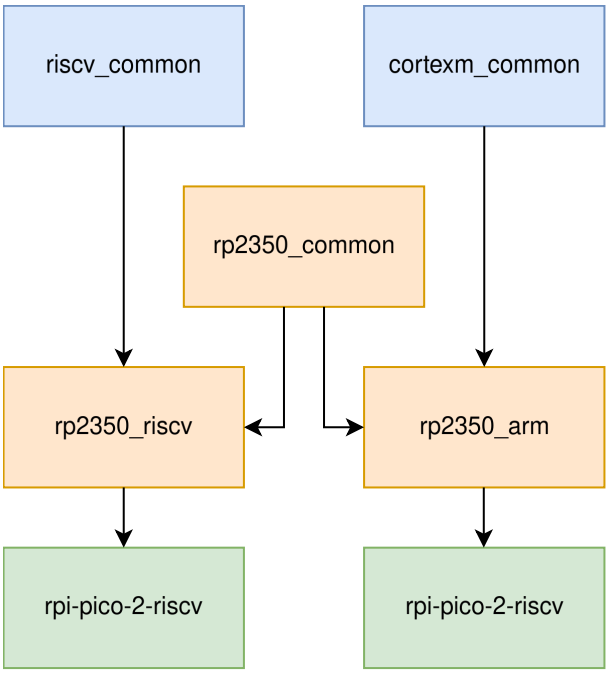

Figure 8: RIOT OS RP2350 module folder structure. Blue denotes architecture-specific modules. Orange denotes CPU module definitions. Green denotes board module definitions.

The general layout of the implementation of the RP2350 support in RIOT can be seen in figure Figure 8. The rpi-pico-2 board includes its architecture-specific module, which then includes the shared `rp2350_common` module.

5.1.2 Build System Architecture Abstraction

The shared `rp2350_common` configures itself based on the defined architecture, either ARM or RISC-V, through defines provided by the RIOT build system, including the import of the architecture-specific common folder.

Architecture-specific function calls are handled through abstraction layers via define flags provided by the RIOT build system. For example, IRQ enabling is done through the `rp_irq_enable` function, which inlines the appropriate function call based on the current architecture, as shown in Listing 1. Using `static inline` allows the compiler to optimize away the function call overhead, resulting in efficient architecture specific code generation while still maintaining a clean and centralized abstraction.

This centralized design allows for the architecture-specific code to be kept to a minimum. The CPU module itself has to create the `cpu_init` function, which is called by the

```c
/**
 * @brief     Enable the given IRQ
 * @param[in] irq_no IRQ number to enable
 */
static inline void rp_irq_enable(uint32_t irq_no)
{
#ifdef RP2350_USE_RISCV
    xh3irq_enable_irq(irq_no);
#else
    NVIC_EnableIRQ(irq_no);
#endif
}
```

Listing 1: Example of architecture-specific IRQ enabling through abstraction in `rp2350_common`.

RIOT kernel during startup. The ARM implementation only requires Listing 2 within the architecture-specific CPU module.

This function then calls both the architecture-specific initialization function and the shared `rp2350_init` initialization function (Listing 3), which handles the initialization of the RP2350 itself.

5.1.3 Build System

The build system of RIOT is built upon the `Makefile` system. Each module can provide a `Makefile.include`, `Makefile.dep`, `Makefile.features`, and a general `Makefile` file, which is then included by the RIOT build system when the module is used. This allows for easy configuration of the build system, including the addition of source files, include paths, and defines.

```c
#include "cpu.h"
#include "periph_cpu.h"

void cpu_init(void)
{
    cortexm_init();
    rp2350_init();
}
```

Listing 2: Example of the ARM `cpu_init` function within the `rp2350_common` module.

```c
/**
 * @brief Initialize the CPU, set IRQ priorities, clocks, and peripherals
 */
void rp2350_init(void)
{
    /* Reset GPIO state */
    gpio_reset();
    /* Reset clock to default state */
    clock_reset();
    /* initialize the CPU clock */
    cpu_clock_init();
    /* initialize the early peripherals */
    early_init();
    /* trigger static peripheral initialization */
    periph_init();
    /* initialize the board */
    board_init();
}
```

Listing 3: The shared `rp2350_init` function within the `rp2350_common` module. Initializes clocks, GPIO, and peripherals.

The main purpose of the per-CPU module is to provide the correct values for the architecture. For example, the RISC-V version needs to define that it supports the RISC-V specific xh3irq and PMP peripherals as shown in Listing 4.

```
CPU_CORE := rv32imac
CPU_FAM     := RP2350
CPU_MODEL   = rp2350_hazard3

FEATURES_PROVIDED += periph_pmp
FEATURES_PROVIDED += periph_xh3irq

include $(RIOTCPU)/rp2350_common/Makefile.features
include $(RIOTCPU)/riscv_common/Makefile.features
```

Listing 4: Example of the RISC-V CPU module `Makefile` including the shared `rp2350_common` and `riscv_common` feature files.

5.1.4 Picobin Integration

To avoid the maintenance burden of a custom linker script, we chose the design in Section 4.1.3 to modify the common linker script to include a hook for the picobin section using the `KEEP` directive to ensure that the picobin block is not discarded during the linking process, but not included when building for other platforms, being an optional addition.

The picobin block must be located within the first 4kB of flash memory. To ensure this, we place the picobin block directly after the interrupt vector table, which is typically located at the beginning of the flash memory region, as shown in the excerpt from the modified linker script in Listing 5.

```
sfixed = .;
_isr_vectors = DEFINED(_isr_vectors) ? _isr_vectors : . ;
KEEP(*(SORT(.vectors*)))
KEEP(*(SORT(.picobin_block*))) /* Keep picobin block used by RP2350 */
*(.text .text.* .gnu.linkonce.t.*)
```

Listing 5: Excerpt of modified linker script from `cortexm_common` module to include picobin block

We do a similar modification (Listing 6) for the RISC-V linker script within the `riscv_common` module to ensure that both architectures support picobin when building for the RP2350. Since RISC-V does not have a vector table at the start of flash, we simply place the picobin block at the beginning.

```
.text           :
{
  KEEP(*(SORT(.picobin_block*)))
  *(.text.unlikely .text.unlikely.*)
  *(.text.startup .text.startup.*)
  *(.text .text.*)
  *(.gnu.linkonce.t.*)
} >flash AT>flash :flash
```

Listing 6: Excerpt of modified linker script from `riscv_common` module to include picobin block

The picobin block itself is a pure Assembly file called `picobin.s` that the RIOT build system automatically includes when building the RP2350 as described in Listing 7.

```
.section .picobin_block, "a" /* "a" means "allocatable" (can be moved by the
linker) */

/* PICOBIN_BLOCK_MARKER_START */
.word 0xffffded3
    /* ITEM 0 START based on 5.9.3.1 */
    .byte 0x42 /* (size_flag == 0, item_type ==
PICOBIN_BLOCK_ITEM_1BS_IMAGE_TYPE) */
    .byte 0x1 /* Block Size in words */
    /* image_type_flags (2 bytes) [See 5.9.3.1 / p419] */
    /* 15 -> 0 (1 for "Try before you buy" image */
    /* 12-14 -> 001 (RP2350 = 1) */
    /* 11 -> 0 (Reserved) */
    /* 8-10 -> 001 (EXE_CPU_ARM == 000) || (EXE_CPU_RISCV == 001) */
    /* 6-7 -> 00 (Reserved) */
    /* 4-5 -> 10 (2) EXE Security */
    /* 0-3 // 0001 IMAGE_TYPE_EXE */
    .hword 0b0001000100100001
    /* ITEM 0 END see 5.1.5.1 for explanation and 5.9.5.1 for the value /
structure */
    .byte 0xff /* PICOBIN_BLOCK_ITEM_2BS_LAST */
    .hword 0x0001 /* Size of the item in words (predefined value) */
    .byte 0x00 /* Padding */
    /* Next Block Pointer */
    .word 0x00000000 /* Next block pointer (0 means no more blocks) */
/* PICOBIN_BLOCK_MARKER_END */
.word 0xab123579 /* Marker for the end of the picobin block */
```

Listing 7: Assembly code for the picobin block used in RP2350 builds, based on the definitions in Section 4.1.3.

If in the future this structure needs to be appended, e.g., to support both RISC-V and ARM with a single binary, this file can be easily modified to include multiple picobin blocks as needed based on the settings explained in Section 4.1.3.

5.2 Interrupt Handling

To facilitate an easy abstraction to registering ISRs for both architectures, `rp2350_common` provides a shared vector table full of function pointers. Given that typical programs will not want to define all 51 ISRs, all entries are initialized to a default handler that causes a core panic.

To make these functions rewritable, all of them are defined with the `weak` and `alias` attributes. When the compiler sees a function with the same name defined elsewhere, it will use that function instead of the default one.

For example, if the user wants to define a ISR for the UART0 peripheral, they can define a function with the name `isr_uart0` and the compiler will use that function instead of the default one.

5.2.1 RISC-V Interrupt Handling

On initialization, the riscv_common startup function `riscv_init` sets the standard trap entry point through the `mtvec` CSR to the `trap_entry` function. This function is then called on every interrupt or exception (commonly referred to as traps in RISC-V).

The `trap_entry` then saves the stack and calls the `trap_handler` function, which then handles the actual interrupt. On other RISC-V devices, this function would then call the handler for PLIC or CLIC, however, since the RP2350 uses the custom XH3IRQ controller, it was necessary to implement our own handler.

To make future ports of Hazard3-based devices easier, the port implements the handler within the `riscv_common` module itself. This allows for easier reuse of the code in future projects, thus reducing implementation effort.

Similar to PLIC and CLIC, the XH3IRQ controller can be enabled through the common RIOT `periph` abstraction layer. Any device that runs on RISC-V and includes the XH3IRQ controller can include the `periph_xh3irq` feature in its CPU module and get support for the XH3IRQ controller.

When enabled, the `trap_handler` checks the `xh3irq_has_pending` function whether the Machine Interrupt Pending CSR has any pending interrupts, as shown in Listing 8. If this is the case, the `trap_handler` in Listing 9 then calls `xh3irq_handler`, which uses the shared vector table to call the appropriate ISR for the pending interrupt depending on the highest priority written within the `MEINEXT` CSR.

```
/**
 * Hazard3 has internal registers to individually filter which
 * external IRQs appear in meip. When meip is 1,
 * this indicates there is at least one external interrupt
 * which is asserted (hence pending in mieipa), enabled in meiea,
 * and of priority greater than or equal to the current
 * preemption level in meicontext.preempt.
 */
#define MEIP_OFFSET 11

/*
* Get MEIP which is the external interrupt pending bit
* from the Machine Interrupt Pending Register address
*/
uint32_t mip_reg = read_csr(0x344);
uint32_t meip = bit_check32(&mip_reg, MEIP_OFFSET);
```

Listing 8: Checking the Machine Interrupt Pending CSR for pending interrupts in `trap_handler`.

```
/*
* Get MEINEXT at 0xbe4, which is the next highest interrupt to handle (Bit
2-10).
* This will also automatically clear the interrupt (See 3.8.6.1.2.)
*
* Contains the index of the highest-priority external interrupt
* which is both asserted in meipa and enabled in meiea, left-
* shifted by 2 so that it can be used to index an array of 32-bit
* function pointers. If there is no such interrupt, the MSB is set.
*/
uint32_t meinext = (read_csr(0xBE4) >> MEINEXT_IRQ_OFFSET) & MEINEXT_MASK;

void (*isr)(void) = (void (*)(void)) vector_cpu[meinext];
```

Listing 9: Fetching the highest priority pending interrupt from the `MEINEXT` CSR and calling the appropriate ISR from the shared vector table in `xh3irq_handler`.

5.2.2 ARM Interrupt Handling

As with the RISC-V interrupt handling, the port aims to conform to the existing RIOT-OS abstractions as closely as possible.

The `cortexm_common` module already includes the necessary setup for the NVIC, including the default vector table and the `cortexm_init` function, which is called during startup to initialize the NVIC.

To allow amendments to the vector table, `cortexm_common` uses an attribute system to give all vector table amendment arrays the `section(".vectors." # x )` attribute that the linker script can then sort and properly place within the final binary.

5.3 Multicore Implementation

The very first step is to wake up the secondary core. The secondary core remains dormant after the initial boot sequence and expects a specific sequence of events for wake up. For that, the RP2350 needs to release the reset state of the core. This can be done by a simple write to the `FRCE_ON` register of the PSM, followed by polling the `DONE` register of the PSM till the software has a confirmation that the reset has completed [3, Chapter 7.4.4].

At this point, the secondary core is in a known state, awaiting further instructions. The port then uses the inter-processor FIFOs described in Section 2.4.5.4 to send the necessary startup information to the secondary core to boot it up. In total, the startup sequence sends six 32-bit values to the secondary core in the order specified in Table 7.

Value	Description
1-3	`0`, `0`, `1`
4	Pointer to ISR vector
5	Initial stack pointer
6	Entry point address (Trampoline function)

Table 7: Values sent to the secondary core via inter-processor FIFO during boot.

A trampoline function is a small piece of code that sets up the environment for the actual function to be called, thus allowing for more complex setups, such as setting up the stack or registers before jumping to the actual function. Thus the handler functions also writes the function and argument to the stack of the secondary core before sending the stack pointer value. To send these values, it follows a specific sequence of steps to ensure that the secondary core receives them correctly. After Listing 10 has completed, the secondary

core should be awake and running the trampoline function at the provided entry point address.

The trampoline function then calls the architecture-specific initialization function, pops both values from the stack, and jumps to the provided entry point function with the provided argument. In the design of this entry function interface, we decided to conform to the way threads are started in RIOT OS. In essence, this means that the entry function needs to have a signature of `void *(*core_1_fn_t)(void *arg)`. The current implementation is designed to offload a specific, blocking task to the secondary core as described in the analysis in Section 4.4.

```
uint32_t seq = 0;
/** We iterate through the cmd_sequence till we covered every param
 * (seq does not increase with each loop, thus we need to while loop this) */
while(seq < 6) {
    uint32_t cmd = cmd_sequence[seq];
    /* If the cmd is 0 we need to drain the READ FIFO first*/
    if(cmd == 0) {
        /* Drain READ FIFO till it is empty */
        while(SIO->FIFO_ST & 1<<SIO_FIFO_READ_VALID_BIT) {
            (void) SIO->FIFO_RD; /* Table 39 FIFO_RD*/
        };
        fifo_unblock_processor();
    }
    /* Check whether queue is full */
    while (!(SIO->FIFO_ST & 1<<SIO_FIFO_SEND_READY_BIT)) {
        /* Wait for queue space */
    }
    SIO->FIFO_WR = cmd; /* Write data since we know we have space */
    fifo_unblock_processor(); /* Send event */
    /* This is eq. to the SDK multicore_fifo_pop_blocking_inline*/
    /* We check whether there are events */
    while(!(SIO->FIFO_ST & 1<<SIO_FIFO_READ_VALID_BIT)) {
        /* If not we wait */
        fifo_block_processor();
    };
    /* Get the event since this is our response */
    volatile uint32_t response = SIO->FIFO_RD;
    /* move to next state on correct response (echo-d value)
     * otherwise start over */
    seq = cmd == response ? seq + 1 : 0;
};
```

Listing 10: Sequence to send the necessary boot values to the secondary core via inter-processor FIFO. First, draining the read FIFO if the value to send is `0`, then sending the value and waiting for an echoed response before proceeding to the next value. On each incorrect response, the sequence is restarted.

5.4 Implementation of Clocks

The RP2350 provides multiple clock sources, initially running from the ROSC (See: Section 4.3.1). The implementation provides a clock initialization function within `rp2350_common` that handles the switch to the more stable XOSC (See: Section 4.3.2) and then switch the system clock, reference clock, and other clocks to the desired frequencies.

This function first initializes the XOSC by setting the appropriate bits within the `XOSC CTRL` register to enable it and waits for it to stabilize. The RP2350 uses 12-bit magic value codes for this to protect against accidental writes. These differ depending on the desired frequency range of the crystal being used [3, Chapter 8.2.8]. After configuring the startup delay timer based on the crystal frequency and desired stabilization time [3, Chapter 8.2.4], the XOSC can be enabled and polled until stable.

At this point, the initialization sequence configures the PLL to run off the XOSC as the reference clock. The feedback and post divider values are calculated based on the desired VCO frequency and final PLL output frequency of 125 MHz [3, Chapter 8.1.6.1]. The port then sets the system clock to run off the PLL output and the peripheral clock to run through the lower line provided by the system clock. The complete clock initialization sequence is encapsulated in the `cpu_clock_init()` function within `rp2350_common`, as shown in Listing 3.

To allow for modifications to the set clock speed, the port provides all these values as `#define` flags, allowing for easy adjustments to the clock speed if the user desires a different configuration, e.g. to save power by running at a lower frequency as discussed in Section 4.3.4. The implementation then asserts that any entered values are within the valid ranges specified in the RP2350 datasheet to avoid misconfigurations that could lead to undefined behavior or at worst hardware damage.

5.5 Programmable Input/Output (PIO) Support

PIO support requires some modification to existing RIOT OS drivers, most notably the GPIO driver. PIO state machines can have direct access to GPIO pins, which requires the GPIO driver to configure the pins accordingly. In RIOT OS, the GPIO driver `gpio_init` function takes two arguments, the pin number and the mode. The mode is defined as a set of flags that configure the pin as input/output, pull-up/down, etc., however, PIO functionality serves as an additional mode.

The easiest way to do this is to redefine these aforementioned flags to include PIO functionality. The RIOT design already considered such scenarios and wrapped the definition with `#ifndef HAVE_GPIO_MODE_T` guards, allowing us to redefine the `gpio_mode_t` enum within the `rp2350_common` module, adding additional flags for PIO0/PIO1 respectively. We can then simply check for these flags within the `gpio_init` function and configure the pin accordingly for PIO functionality by setting the appropriate bits within the `PIO_CTRL` register of the RP2350. The user can then configure the desired pins as PIO0/PIO1 and use the existing PIO driver to configure and use the PIO state machines as needed.

5.5.1 Abstracting PIO Instruction Generation

To facilitate usage of PIO within RIOT OS, we implemented an abstraction layer for generating PIO instructions. This layer provides a set of functions that allow users to create PIO programs without needing to write raw PIO assembly code. Specifically, we use C Macros to define common PIO instructions, making it easier to construct PIO programs programmatically, as shown in Listing 11 for the uncondition `JMP` jump instruction.

This ensures that the generated instructions are correct and reduces the likelihood of errors when writing PIO programs as compared with writing raw binary values.

```
#define PIO_JMP_COND_ALWAYS     (0)     /**< Always jump */
/**
 * @brief JMP instruction encoding
 *
 * Set program counter to address if condition is true.
 *
 * @param[in] cond      Condition (PIO_JMP_COND_*)
 * @param[in] addr      Target address (0-31)
 */
#define PIO_JMP(cond, addr) \
    (0b0000000000000000 | (((cond) & 0b111) << 5) | ((addr) & 0b11111))
/**
 * @brief JMP - unconditional jump
 *
 * @param[in] addr  Target address (0-31)
 */
#define PIO_JMP_ALWAYS(addr)    PIO_JMP(PIO_JMP_COND_ALWAYS, (addr))
```

Listing 11: Example of C macros to generate PIO instructions, specifically the `JMP` instruction with conditional and unconditional variants.

The resulting macro output can then be directly written into the instruction memory of the PIO state machines. Alternatively, if users want to use the pico sdk PIO assembler, they can still do so by including the necessary headers. This way, our own PIO support remains vendor agnostic and does not rely on the pico sdk while still allowing users to leverage existing tools if desired.

5.5.2 PIO Usage Example

Through the abstractions provided, a simple PIO program such as Listing 12 can be created with minimal effort fully within RIOT OS, without a need to write raw PIO assembly code or binary values. The code in the example initializes a PIO program that generates a square wave on GPIO0 by setting the pin high and low in a loop. While not a complex example, it demonstrates the ease of use provided by the PIO abstraction layer within RIOT OS.

```
static const uint16_t squarewave_program_instructions[] = {
    PIO_SET_PINDIRS(1),
    PIO_SET_PINS(1),
    PIO_SET_PINS(0),
    PIO_JMP_ALWAYS(1),
};

int main(void) {
    // Load instructions
    for (uint32_t i = 0; i < 4; ++i) {
        *(&PIO0->INSTR_MEM0 + i)
            = squarewave_program_instructions[i];
    }
    // Set the Clock Divider for SM0
    PIO0->SM0_CLKDIV = (uint32_t) (1.0f * (1 << 16)); //12.5 MHz
    // Configure the Pin Control for SM0
    PIO0->SM0_PINCTRL = (1 << PIO_SM0_PINCTRL_SET_COUNT_LSB) |
        (0 << PIO_SM0_PINCTRL_SET_BASE_LSB);
    // Initialize GPIO0 for PIO usage
    gpio_init(0, GPIO_PIO0);
    // Set SM0 to enabled
    atomic_set(&PIO0->CTRL, 1 << PIO_CTRL_SM_ENABLE_LSB);
}
```

Listing 12: Example of using the PIO abstraction layer to create a simple square wave generator on GPIO0 using PIO0. The program sets the pin high and low in a loop, creating a square wave output. The GPIO pin is initialized for PIO usage using the modified GPIO driver.

6 Evaluation

In this chapter, we evaluate the implementation of the RP2350 port in RIOT OS. We assess the practical performance characteristics of the port and verify if the design goals outlined in Section 4 and Section 5 have been achieved.

The evaluation starts with the multicore support implementation, where we measure the performance benefits of the "worker core" design discussed in Section 4.4. This is followed by a detailed code size comparison between the ARM Cortex-M33 and RISC-V Hazard3 architectures to validate the efficiency of the unified abstraction layer.

Finally, we demonstrate the ecosystem benefits of the RIOT integration, including the access to the comprehensive testing infrastructure and support for high-level languages.

6.1 Multicore Support

6.1.1 Methodology

To evaluate the multicore support implemented in Section 5, we designed a series of tests to demonstrate the functionality and performance benefits of utilizing both cores, even in a more limited fashion as currently implemented.

As a simple demonstration of the functionality of both cores working in tandem, we implemented a dual-core GPIO test application Listing 13. Both cores toggle separate GPIO pins as fast as possible. While this is a simple test, it effectively demonstrates that both cores can operate independently and concurrently in scenarios where, for example, vast amounts of data need to be transferred over GPIO.

As a point of comparison, we also implemented a single-core version, Listing 13, of the same application, where only one core toggles both GPIO pins sequentially. The hypothesis is that the dual-core version should be twice as fast as the single-core version, assuming both cores can operate at full speed. We run both tests on the Cortex-M33 core, measuring the toggling frequency of each GPIO pin using an oscilloscope.

```c
#define PIN_14 14u
#define PIN_15 15u

/* Single-core GPIO toggling both pins
sequentially */
int main(void) {
    gpio_init(PIN_15, GPIO_OUT);
    gpio_init(PIN_14, GPIO_OUT);
    uint32_t selected_pin = PIN_15;
    while (1) {
        selected_pin = (selected_pin ==
PIN_15) ? PIN_14 : PIN_15;
        gpio_set(selected_pin);
        gpio_clear(selected_pin);
    }
    return 0;
}
```

```c
/* This function runs on core 1 */
void* core1_main(void *arg) {
    (void)arg;
    gpio_init(PIN_14, GPIO_OUT);
    while (1) {
        gpio_set(PIN_14);
        gpio_clear(PIN_14);
    }
    return NULL;
}
/* This function runs on core 0 */
int main(void) {
    /* This will start core 1 and run
core1_main on it */
    core1_init(core1_main, NULL);
    gpio_init(PIN_15, GPIO_OUT);
    while (1) {
        gpio_set(PIN_15);
        gpio_clear(PIN_15);
    }
    return 0;
}
```

Listing 13: Single-core GPIO toggling both pins sequentially (left) and dual-core GPIO toggling both pins in parallel (right).

6.1.2 Results

We can observe the results of both tests using an oscilloscope in Figure 9. The sequential toggling of both pins in the single-core version results in a lower frequency signal, as expected. In contrast, the dual-core version shows both pins toggling at a higher frequency and in parallel, confirming that both cores are functioning correctly and independently, given the simultaneous toggling of both GPIO pins.

The average signal period of each GPIO pin in the single-core test is approximately 560 nanoseconds. In the dual-core test, the average signal period is around 288 nanoseconds.

In total, we can see a performance improvement of approximately 94.94% when utilizing both cores in parallel compared to a single core handling both tasks sequentially.

Figure 9: Oscilloscope captures showing single-core GPIO toggling (left) and dual-core GPIO toggling (right). Yellow (Top) is PIN 14, Blue (Bottom) is PIN 15. Single core average period: 560 ns, Dual core average period: 288 ns.

6.1.3 Discussion

Given the design choices explained in Section 4.4 and the differences in approach to the multi-core scheduler of Ariel OS as discussed in Section 3.4, we can conclude that while the current implementation demonstrates the feasibility of multi-core processing on the RP2350 within RIOT OS, there is significant room for improvement.

Ariel OS and other operating systems that have been designed with multi-core support from the ground up, implement more sophisticated scheduling algorithms that can better utilize the capabilities of both cores. This includes load balancing, inter-core communication mechanisms, and more efficient context switching as discussed in “Multicore Scheduling and Synchronization on Low-Power Microcontrollers using Embedded Rust” by Elena Frank [23].

Comparing the results here with those of the master’s thesis by Elena Frank, she achieves a performance improvement of 84% when utilizing both cores on the RP2040 in a CPU-bound workload. In the thesis, she calculated π using the Leibniz formula, sending calculation results between the cores, demonstrating that even a more advanced scheduler can achieve similar performance improvements in CPU-bound workloads while also utilizing more advanced features such as inter-core communication [23].

While inter-core communication is feasible with the current implementation using the methods described in Section 2.3, we provide no higher level abstractions for it. One of the key objectives stated in Section 1.2 of this thesis is the creation of a unified architecture abstraction layer that allows seamless switching between the ARM Cortex-M33 and Hazard3 RISC-V architectures within RIOT OS.

Both architectures can run RIOT OS with multi-core support, and the basic functionality of utilizing both cores has been demonstrated, which is something RIOT OS did not support before, which was a key objective of this thesis. Yet the limitations of the current multi-core implementation, particularly in terms of scheduling and inter-core communication, indicate that there is still future work to be done when comparing to multi-core operating systems, such as Ariel OS.

6.2 Code Size Comparison

To evaluate the efficiency of our implementation and compare the two architectures supported by the RP2350, we performed a detailed code size analysis of a minimal multicore application. This analysis provides insights into the memory footprint required for each architecture and helps identify optimization opportunities.

6.2.1 Methodology

We compiled a minimal dual-core GPIO application, Listing 13, for both the ARM Cortex-M33 and RISC-V Hazard3 architectures. The application uses basic peripheral drivers (UART, GPIO) and demonstrates multicore functionality, making it representative of a typical embedded application on the RP2350.

RIOT provides a memory usage analysis tool called `cosy`, which we used to extract detailed size information from the compiled binaries[28]. The measurements were performed with the standard RIOT build configurations for each architecture to ensure consistency. The builds were also done within the standard RIOT docker environment. RIOT ran in the default `-Os` optimization level, which enables all `-O2` optimizations except those that increase code size. The chosen stack size here is the default size allocated by RIOT OS for each core depending on the architecture, which can be configured by the user.

6.2.2 Results

6.2.2.1 ARM Cortex-M33

The ARM build produces a binary with a total text section size of 7,610 bytes. Table 8 shows the breakdown by major components across all memory sections.

[28]RIOT Cosy Repository (Accessed 04.01.2026): https://github.com/RIOT-OS/cosy/

Component	Text (bytes)	Data (bytes)	BSS (bytes)	Total (bytes)
cpu	2576	-	2064	4640
core	1972	2	1754	3728
pkg	1768	-	-	1768
boards	1012	-	-	1012
newlib	96	-	-	96
sys	86	-	-	86
app	60	-	-	60
unspecified	20	-	-	20
drivers	16	-	-	16
fill	4	2	2	8
Total (Stacks)	**7610**	**4**	**3820**	**11434**
Total (No Stacks)	**7610**	**4**	**236**	**7850**

Table 8: ARM memory section breakdown by component

A noticeable contribution to the larger BSS section comes from the default stack allocation of 1536 bytes for each core and an additional 512 bytes for the isr_stack in RIOT OS on ARM Cortex-M33. Thus, a total of 3584 bytes are allocated for stacks by default.

Table 9 shows the detailed breakdown of the `cpu` text section, revealing the contributions from the shared `rp2350_common` module, the `cortexm_common` module, and the ARM-specific `rp2350_arm` module.

Module/Symbol	Size (bytes)
rp2350_common	1394
cortexm_common	1168
rp2350_arm	14
Total	**2576**

Module/Symbol	Size (bytes)
periph	784
vectors.o	224
core.o	164
clock.o	88
xosc.o	68
cpu.o	66
Total	**1394**

Table 9: ARM cpu text section breakdown by module (left) and rp2350_common text section breakdown (right)

Component	Text (bytes)	Data (bytes)	BSS (bytes)	Total (bytes)
cpu	2765	208	1300	4273
core	2201	-	1754	3955
pkg	2168	-	-	2168
sys	156	-	-	156
newlib	150	-	-	150
examples	66	-	-	66
boards	40	-	-	40
unspecified	20	-	-	20
fill	15	-	2	17
drivers	6	-	-	6
Total (Stacks)	**7587**	**208**	**3056**	**10851**
Total (No Stacks)	**7587**	**208**	**240**	**8035**

Table 10: RISC-V memory section breakdown by component

6.2.2.2 RISC-V Hazard3

The RISC-V build produces a slightly more compact binary with a total text section size of 7,587 bytes. Table 10 shows the breakdown by major components.

On RISC-V, the default stack allocation is 1280 bytes for each core and 256 bytes for the idle stack, leading to a total of 2816 bytes allocated for stacks by default. Table 11 shows the detailed breakdown of the `cpu` text section, revealing the contributions from `riscv_common`, the shared `rp2350_common` module, and the RISC-V-specific `rp2350_riscv` module. The larger data section can be explained by the XH3IRQ interrupt vector table we use to abstract the interrupt controller differences, which is stored in the data section (32-bit pointers for 52 interrupts = 208 bytes).

Table 11 provides a breakdown of the `cpu` module and the `rp2350_common` module for RISC-V, showing the individual object file contributions.

6.2.3 Analysis

The comparison reveals several important characteristics of both architectures:

Text Section (Code Density): Both architectures show remarkably similar text section sizes, with ARM at 7,610 bytes and RISC-V at 7,587 bytes, a difference of only 23 bytes (0.3%). This indicates that our unified abstraction layer effectively minimizes

Module/Symbol	Size (bytes)
riscv_common	1441
rp2350_common	1308
rp2350_riscv	16
Total	**2765**

Module/Symbol	Size (bytes)
periph	876
core.o	178
clock.o	94
cpu.o	78
xosc.o	64
vectors.o	18
Total	**1308**

Table 11: RISC-V cpu text section breakdown by module (left) and rp2350_common text section breakdown (right)

architecture-specific overhead, allowing both architectures to achieve comparable code density.

Data Section: The RISC-V build shows a notably larger data section (208 bytes vs 4 bytes on ARM). This difference stems from the RISC-V architecture storing the interrupt vector tables in the data section for interrupt controller compatibility purposes.

BSS Section (RAM Usage): When excluding stack allocations, both architectures show comparable BSS usage (236 bytes on ARM vs 240 bytes on RISC-V). The total BSS difference (3,820 bytes on ARM vs 3,056 bytes on RISC-V) is primarily due to differing default stack sizes. These stack sizes are configurable by the user based on application requirements, thus a direct comparison may not reflect actual application memory usage scenarios.

Shared Components: Both architectures benefit from the modular design of RIOT OS. The shared `rp2350_common` module contributes 1,394 bytes on ARM and 1,308 bytes on RISC-V, with the difference primarily in peripheral driver implementations and vector table handling. The `pkg` module shows sizes of 1,768 bytes on ARM vs 2,168 bytes on RISC-V, attributed to architecture-specific library optimizations.

CPU Module: The architecture-specific modules (`cortexm_common` at 1,168 bytes vs `riscv_common` at 1,441 bytes) reflect the differing interrupt handling and context switching mechanisms inherent to each architecture. Notably, the chip-specific modules (`rp2350_arm` at 14 bytes and `rp2350_riscv` at 16 bytes) are minimal, demonstrating the effectiveness of the unified abstraction layer.

Total Memory Footprint: The overall memory footprint is 11,434 bytes for ARM and 10,851 bytes for RISC-V (including stacks), or 7,850 bytes vs 8,035 bytes when excluding

stack allocations. This represents a difference of less than 2.4% in either direction, confirming that the choice between architectures does not significantly impact memory requirements.

These results validate our implementation approach and demonstrate that both architectures provide viable options for RP2350 development. The choice between ARM and RISC-V can be made based on other factors such as toolchain preferences, debugging capabilities, or specific peripheral requirements, as the memory overhead differences are minimal. Even more so when considering the 520 kB of SRAM and 4 MB of onboard QSPI flash on the Rasperry Pi Pico 2.

6.2.4 Comparison with Pico SDK

To provide additional context, we compared the code size of our RIOT OS implementation with that of the official Raspberry Pi Pico SDK for the RP2350. For that, we implemented Listing 13 in the Pico SDK and compiled through their default build system. We followed the exact methodology outlined in the "Getting Started with Raspberry Pi Pico" guide, including compiling using the Microsoft Visual Studio Code Pico extension [24]. The default core stack size on Pico SDK is 2048 bytes.

Table 12 shows the memory usage comparison between the Pico SDK and RIOT OS for the same dual-core GPIO application on the ARM Cortex-M33 core. We can see a significant difference in memory usage, with the Pico SDK requiring 40,847 bytes compared to 8,362 bytes in RIOT OS. This substantial difference can be attributed to the overhead included by the Pico SDK, which may not be necessary for all applications, including a few fairly massive newlib components used within the RP2350 initialization of the SDK and not required by the written application code.

Memory Section	**Pico SDK**	**RIOT OS**
Code (.text)	24688 bytes	7610 bytes
Data (.data/.rodata)	17838 bytes	4 bytes
Zero-initialized (.bss)	2417 bytes	3820 bytes
Overall Memory Usage		
Total	**44943 bytes**	**11434 bytes**
Total (No Core Stacks)	**40847 bytes**	**8362 bytes**

Table 12: Memory comparison between Pico SDK and RIOT OS for dual-core GPIO application on the ARM Cortex-M33 cores

This indicates that the RP2350 port in RIOT OS follows the design goal of RIOT to be a lightweight operating system suitable for resource-constrained embedded systems, while still providing essential features and abstractions for application development and offers a more efficient memory footprint compared to the vendor SDK.

6.3 Benefits of RIOT on RP2350

The idea of integrating the RP2350 into RIOT OS was driven by the potential benefits that RIOT OS could offer to such a MCU, including the vast ecosystem of supported libraries, protocols and unit/integration tests that RIOT OS provides.

RIOT allowed us to abstract away many critical low-level details that would require significant design and implementation efforts to implement from scratch, including threading, cryptography, scheduling and other core OS functionalities. Any user willing to use RIOT on the RP2350 can now leverage these well-tested components without needing to reimplement them.

Comparing it to the official vendor `picosdk` SDK mentioned in Section 3, RIOT OS provides a considerable advantage in terms of available features and libraries, making it a valuable option for developers looking to build applications on the RP2350, without the vendor-lock-in of using the own SDK by Raspberry Pi. Examples include network stacks such as CoAP through `unicoap`, graphic drivers through `lvgl` or even Web Assembly support through `wamr`.

6.3.1 Unit Tests / Integration Tests

Through the integration with RIOT, we can leverage the existing unit and integrations tests, through the entire process of porting RIOT OS to the RP2350. Each commit made to the port could be verified against the existing test suite, including tests against all examples and the core/sys modules of RIOT OS.

The Murdock CI Runner of RIOT enables this by building the entire suite on large server clusters, reducing the time it would take to run the tests locally and by that enabling rapid iteration during development[29].

Most notably it builds both the ARM and RISC-V versions of each test, ensuring that both architectures are always tested in parallel.

[29]One example test suite run (11.10.2025) showing all tests passing on the RP2350 (Accessed 14.11.2025): https://ci.riot-os.org/details/01fef4c5e621454aa199aa339fec965b

6.3.2 Rust Integration

Through RIOT OS support for Rust and C++, we were able to easily integrate these languages into our RP2350 port. In the example of Rust, all that was required was to add the appropriate target specifications. The existing RIOT build system then takes care of compiling and linking the Rust code into the final binary, allowing us to leverage existing Rust libraries and tools within our RP2350 applications.

The Rust integration of RIOT is compatible with all existing code written for the RP2350, including the initialization, peripheral drivers and interrupt handling. Thus, while this support was trivial to add for the RP2350 port, it stands as an example of the benefits that RIOT OS provides. Comparing it to the official Pico SDK which does not support Rust, the user is able to easily switch between these languages without leaving their existing codebase behind.

Looking into the binary size of a simple "Hello World" application, written in both C and Rust, we can see that the Rust version is only slightly larger than the C version, as shown in Figure 10.

Aside from the access to the language itself, the RIOT Rust integration also provides access to the existing embedded Rust ecosystem, including the `embassy` async framework

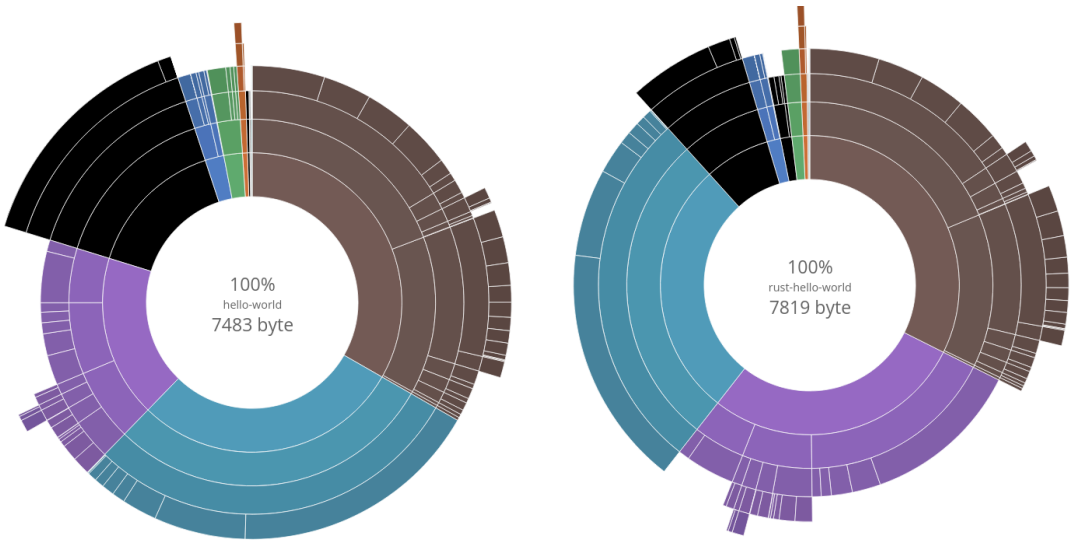

Figure 10: Binary size comparison of a "Hello World" application written in C (left) and Rust (right) for the RP2350 RISC-V Hazard3 cores running RIOT OS. C TEXT size: 7483 bytes, Rust TEXT size: 7819 bytes. The different colors represent different modules, such as the core, rp2350_common or pkg that contribute to the final binary size.

that is also used by Ariel OS (see Section 3.4). This allows users to leverage the benefits of async programming on the RP2350 within RIOT OS, opening up new possibilities for application design and architecture.

It should be noted that currently the Rust integration of RIOT does not support the Cortex-M33 cores of the RP2350, due to limitations in the existing RIOT Rust support. However, this could be added in the future, given the existing support for ARM Cortex-M architectures in the Rust embedded ecosystem.

6.3.3 Accessing Third Party Libraries

Unless a third party library has board/cpu-specific code, it can be used on the RP2350 without any modifications. The user can simply specify any library with `USEPKG` or `USEMODULE` in their application Makefile, and the RIOT build system will take care of the rest. This is a significant advantage over using vendor-specific SDKs, where the user would often need to manually port or adapt third-party libraries to work with the specific SDK and hardware.

A good example of this is the usage of modules such as `stdio_uart`. Without RIOT, the user would need to implement their own STDIO over UART functionality, or adapt an existing library to work with the RP2350 hardware. With RIOT, the user can simply include the `stdio_uart` module in their application, and it will work out of the box on the RP2350. Such functionality has proven to be very useful during the entire development process, allowing quick iterations over the code that matters, rather than spending time on boilerplate code to get basic functionality working.

This ease of integration with libraries significantly lowers the barrier to entry for developers looking to build applications on the RP2350, allowing them to focus on their application logic rather than low-level hardware details.

6.3.4 PMP Support

The RIOT implementation of PMP support (see Section 2.4.2.1) based on the work of Bennet Blischke complies with the official PMP specifications [17]. In the testing of the PMP implementation on the RP2350 Hazard3 cores, we observed that the PMP does not function as expected. Specifically, the RP2350 only supports eight PMP regions instead of the standard 16 or 64 regions the specification allows.

This limitation hinders the effective use of the existing PMP implementation in RIOT OS, as it requires vendor specific adjustments to function correctly on the RP2350, though the core implementation remains compliant with the RISC-V PMP specification.

However, Errata `RP2350-E6` breaks the PMP specification conformity. The standard ordering for PMP permissions is X, W, R (execute, write, read). The Hazard3 incorrectly interprets the ordering as R, W, X (read, write, execute) [3]. The Hazard3 core v1.1 revision fixed this issue in April 2024[30].

Raspberry Pi decided to not include this fix in newer RP2350 revisions, instead opting to keep the errata. Based on the errata description, it can be assumed that this was done to maintain compatibility with existing software, as it states that the issue was fixed through "Documentation". This means that the PMP implementation in RIOT OS will not work correctly on any RP2350 device.

[30]The commit fixing the issue (Github, Accessed 03.11.2025): https://github.com/Wren6991/Hazard3/commit/7d370292b00f5bab846a1702ee24cc41179d631e

7 Conclusion

The embedded systems landscape is diversifying with the emergence of RISC-V as an open alternative to established ARM architectures. The Raspberry Pi RP2350 microcontroller offers both ARM Cortex-M33 and Hazard3 RISC-V cores in a single device, allowing developers to choose between architectures without changing hardware. However, operating system support for such heterogeneous platforms requires careful abstraction to maintain portability across different processor architectures.

The contribution of this thesis comprises several parts. First, a unified abstraction layer was implemented through the `rp2350_common` module, allowing applications to compile for either ARM or RISC-V without modifications. The abstraction uses compile-time flags and inline wrappers to handle architecture-specific differences transparently.

Second, support for the Hazard3 XH3IRQ custom interrupt controller was integrated into the `riscv_common` module through the `periph_xh3irq` feature. This enables future Hazard3-based devices to reuse the implementation. The integration maintains compatibility with existing interrupt handling patterns of RIOT while abstracting interrupt controller specifics between both architectures.

Third, multicore support was implemented using a worker-core model, in which the secondary core executes tasks independently of the main RIOT scheduler. Additional contributions include picobin image format integration through modifications of common linker scripts, support for OpenOCD and Picotool flashing methods, configurable clock management, and support for the RP2350 Programmable Input/Output (PIO) subsystem.

The evaluation showed that both architectures achieve comparable binary sizes with similar code distributions. Rust integration demonstrates the benefits of integrating the RP2350 into RIOT, with a minimal increase in binary size for Rust applications. Integration with the RIOT CI system ensures ongoing validation through comprehensive test suites for both architectures.

Some limitations were identified. The Hazard3 core has non-standard PMP permission bit ordering (Errata `RP2350-E6`), preventing standard RISC-V PMP implementations from functioning correctly. The worker-core multicore model requires explicit task management rather than transparent scheduling, severely limiting applicability for thread-intensive workloads, compared to other multicore operating systems.

The RP2350 port provides RIOT users with access to an affordable dual-architecture development platform. The port enables use of RIOT ecosystem of network stacks, cryptographic libraries, and third-party packages on both ARM and RISC-V without vendor lock-in. The unified abstraction layer establishes design patterns applicable to future heterogeneous platforms.

8 Outlook

8.1 TrustZone-M and Security Features

The Cortex-M33 core of the RP2350 supports TrustZone-M. In her master thesis, Lena Boeckmann integrated TrustZone-M on the Cortex-M33 into RIOT [7]. This could be extended to the RP2350 port in the future. The author of the thesis even suggests the RP2350 as an interesting target for future work, due to the presence of both ARM security features, such as TrustZone-M and PMP, on the RISC-V core. While this thesis focused on getting a functional port of RIOT OS running on the RP2350, this can be seen as a stepping stone towards the aforementioned research into security features on heterogeneous architectures.

8.2 Heterogeneous Core Utilization

Another interesting avenue for future work is to explore the potential of using both the RISC-V and Cortex-M33 cores in a Core0 and Core1 configuration, where Core0 (Cortex-M33) handles security-sensitive tasks through the Secure Mode support, while Core1 (RISC-V) manages less critical operations. The official Raspberry Pi documentation hints at this possibility but does not provide concrete examples or implementations, warning that it could be challenging on the software side [3, Chapter 3.9.2].

8.3 USB Support

In 2024, a `periph_usb` RIOT driver for the RP2040 was drafted. This could be finished and adapted to work on the RP2350 as well[31]. The `periph_usb` driver would allow RIOT

[31]The periph_usb driver draft (Accessed 27.10.2025): https://github.com/RIOT-OS/RIOT/pull/20817

applications running on the RP2350 to utilize its USB functionality, including UART over USB, which would lower the barrier to entry for developers wanting to experiment with RIOT on the RP2350.

8.4 Advanced Multi-Core Features

In this thesis, we have only scratched the surface of multicore processing within RIOT OS. Future work could explore more advanced multicore features, such as inter-core communication mechanisms, load balancing, and task scheduling across cores. Currently, we avoid the scheduler, future work could explore how to extend the existing RIOT scheduler to be multicore aware, allowing it to distribute tasks between the two cores more effectively in a hardware-agnostic abstraction.

Glossary

CSR: Control and Status Registers (CSRs) are special-purpose registers in RISC-V processors that control various aspects of the processor operation and provide status information. These are privileged registers, meaning they can only be accessed in certain privilege modes (e.g., Machine mode). CSRs are used for tasks such as configuring interrupts, managing memory protection, and controlling performance counters [14].

RV32I: The base integer instruction set for 32-bit RISC-V processors. It includes basic arithmetic, logical, control flow, and memory access instructions. It is the foundation for all RISC-V implementations and can be extended with optional instruction set extensions for additional functionality.

first-party: In the context of software packages, "first-party" refers to packages or components that are developed and maintained by the original creators or maintainers of the software platform itself. In contrast, "third-party" packages are developed by external contributors or organizations not directly affiliated with the original software platform.

PLL: A Phase-Locked Loop (PLL) is an electronic circuit that generates a stable output clock signal by synchronizing with a reference clock signal. It is commonly used in MCUs to provide higher frequency clocks derived from a lower frequency source, enabling precise timing and frequency control for various system components.

PMA: The Physical Memory Attributes (PMA) specification defines how different types of memory regions behave in terms of caching, buffering, and ordering. It provides guidelines for memory access to ensure correct operation and performance optimization [14].

three-stage pipelined: A three-stage instruction pipeline that improves performance by overlapping instruction execution. The stages are:

- **Fetch** – fetches instructions from memory and performs predecoding
- **Execute** – decodes and executes instructions and handles control flow
- **Memory** – completes memory operations and writes results back to registers.

VCO: A Voltage-Controlled Oscillator (VCO) is an electronic oscillator whose output frequency is controlled by an input voltage. In the context of PLLs, the VCO generates a clock signal that can be adjusted based on the feedback from the PLL to maintain synchronization with the reference clock.

Bibliography

[1] L. Wren, "Hazard3 Datasheet." 2024. Accessed: Oct. 09, 2025. [Online]. Available: https://raw.githubusercontent.com/Wren6991/Hazard3/refs/heads/stable/doc/hazard3.pdf

[2] K. R. Raghunathan, "History of Microcontrollers: First 50 Years," *IEEE Micro*, vol. 41, no. 6, pp. 97–104, 2021, doi: 10.1109/MM.2021.3114754.

[3] Raspberry Pi Ltd, "RP2350 Datasheet." 2025. Accessed: Oct. 09, 2025. [Online]. Available: https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf

[4] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, Mar. 2018, doi: 10.1109/JIOT.2018.2815038.

[5] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA." Dec. 2019. Accessed: Oct. 09, 2025. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[6] The SHD Group, "RISC-V Market Analysis 2024: Abridged Report." Accessed: Oct. 22, 2025. [Online]. Available: https://theshdgroup.com/wp-content/uploads/2024/01/RISC-V-Market-Analysis-2024-Abridged-Report-2.pdf

[7] L. Boeckmann, "Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS," Master's thesis, HAW Hamburg, 2025. Accessed: Oct. 13, 2025. [Online]. Available: https://inet.haw-hamburg.de/thesis/completed/ma_lena_boeckmann.pdf

[8] Raspberry Pi Ltd, "Raspberry Pi Pico 2 Product Brief." Nov. 2024. Accessed: Oct. 10, 2025. [Online]. Available: https://datasheets.raspberrypi.com/pico/pico-2-product-brief.pdf

[9] E. Upton, “Raspberry Pi Pico 2, our new $5 microcontroller board, on sale now.” Accessed: Oct. 09, 2025. [Online]. Available: https://www.raspberrypi.com/news/raspberry-pi-pico-2-our-new-5-microcontroller-board-on-sale-now/

[10] Raspberry Pi Ltd, “RP2040 Datasheet.” 2020. Accessed: Oct. 09, 2025. [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[11] RISC-V International, “RISC-V Bit-Manipulation ISA Extensions, Version 1.0.0-38-g865e7a7.” June 28, 2021. Accessed: Oct. 16, 2025. [Online]. Available: https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0-38-g865e7a7.pdf

[12] RISC-V International, “RISC-V Crypto Extensions, Scalar Specification, Version 1.0.1.” 2023. Accessed: Oct. 16, 2025. [Online]. Available: https://github.com/riscv/riscv-crypto/releases/download/v1.0.1-scalar/riscv-crypto-spec-scalar-v1.0.1.pdf

[13] RISC-V International, “RISC-V Zc Code-Size Reduction Extension, Version 1.0.3-1.” 2024. Accessed: Oct. 16, 2025. [Online]. Available: https://github.com/riscv/riscv-code-size-reduction/releases/download/v1.0.3-1/Zc-v1.0.3-1.pdf

[14] RISC-V International, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.” Oct. 2025. Accessed: Oct. 10, 2025. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-6f1d759-2025-10-10/riscv-privileged.pdf

[15] RISC-V International, “RISC-V External Debug Support Specification.” Dec. 2019. Accessed: Oct. 16, 2025. [Online]. Available: https://riscv.org/wp-content/uploads/2024/12/riscv-debug-release.pdf

[16] L. Wren, “Hazard3: 3-stage RV32IMACZb* processor with debug.” Accessed: Oct. 09, 2025. [Online]. Available: https://github.com/Wren6991/Hazard3

[17] B. Blischke, “Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices,” Bachelors thesis, HAW Hamburg, 2023. Accessed: Oct. 23, 2025. [Online]. Available: https://inet.haw-hamburg.de/thesis/completed/ba_bennet_blischke.pdf

[18] P. B. Hemanthkumar, F. T. Josh, S. R. Anireddy, and R. Venkatesan, “Introduction to ARM processors & its types and Overview to Cortex M series with deep explanation of each of the processors in this Family,” in *2022 International Conference on Computer Communication and Informatics (ICCCI)*, IEEE, Jan. 2022, pp. 1–8. doi: 10.1109/ICCCI54379.2022.9740768.

[19] P. D. Berven, "Porting Inferno OS to ARMv7-M and Cortex-M7," Master's thesis, Norwegian University of Science, Technology (NTNU), 2022. Accessed: Oct. 20, 2025. [Online]. Available: https://hdl.handle.net/11250/3034870

[20] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," in *UKUUG Summer*, Buntingford, Herts: United Kingdom UNIX systems User Group, July 1990, pp. 1–9.

[21] M. Muench, A. Cullen, K. C. Courdesses, T. '. Roth, and A. Zonenberg, "Security through Transparency: Tales from the RP2350 Hacking Challenge," in *USENIX WOOT Conference on Offensive Technology (WOOT'25)*, USENIX Association, 2025.

[22] E. Frank, K. Schleiser, R. Fouquet, K. Zandberg, C. Amsüss, and E. Baccelli, "Ariel OS: An Embedded Rust Operating System for Networked Sensors & Multi-Core Microcontrollers," in *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, IEEE, June 2025, pp. 241–245. doi: 10.1109/dcoss-iot65416.2025.00040.

[23] E. Frank, "Multicore Scheduling and Synchronization on Low-Power Microcontrollers using Embedded Rust," Master's thesis, Freie Universität Berlin, 2024.

[24] Raspberry Pi Ltd, "Getting started with Raspberry Pi Pico-series: C/C++ development with Raspberry Pi Pico-series and other Raspberry Pi microcontroller-based boards." Accessed: Oct. 23, 2025. [Online]. Available: https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf

[25] J. Yiu, "Chapter 7 - Exceptions and Interrupts," *The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors (Third Edition)*. Newnes, Oxford, pp. 229–272, 2014. doi: https://doi.org/10.1016/B978-0-12-408082-9.00007-5.

[26] STMicroelectronics, "STM32 Cortex-M33 MCUs and MPUs Programming Manual (PM0264)." Mar. 2025. Accessed: Oct. 14, 2025. [Online]. Available: https://www.st.com/resource/en/programming_manual/pm0264-stm32-cortexm33-mcus-programming-manual-stmicroelectronics.pdf

[27] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 3, Mar. 2021, doi: 10.1145/3441643.

Declaration of Independent Processing

I hereby certify that I wrote this work independently without any outside help and only used the resources specified. Passages taken literally or figuratively from other works are identified by citing the sources.

Place	Date	Original Signature