Exokernel-based Distributed Operating System for RISC-V Microcontrollers

by

Zakhar Semenov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

April 2025

Author .....
Department of Computer Science
2025.04.30

Certified by .....
Nursultan Kabylkas
Assistant Professor
Thesis Supervisor

Certified by .....
Sain Saginbekov
Assistant Professor
Thesis Supervisor

Certified by .....
Akhan Almagambetov
Associate Professor
Thesis Supervisor

Accepted by .....
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

Exokernel-based Distributed Operating System for RISC-V Microcontrollers

by

Zakhar Semenov

Submitted to the Department of Computer Science
on 2025.04.30, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

The industrial adoption of microcontrollers is growing drastically. Advancements in the field of Internet of Things lead to developments of large-scale automation systems such as smart homes, smart cities, etc. The centralized architecture of such systems faces challenges when it comes to scaling. A decentralized operating system injected into each of the nodes can enhance performance and provide low maintenance for scaling. Orbit - a distributed operating system kernel is proposed to create a unified framework for the execution of tasks on a network of nodes. This framework is a foundation for the distributed operating system for various microcontrollers under the RISC-V instruction set architecture. The methodology involves an evaluation of the kernel design based on exokernel principles that integrates an interrupt-driven scheduler. The anticipated results of this study are a lightweight, efficient and flexible kernel capable of task distribution across a network of nodes with heterogeneous hardware. Future works may focus on the integration of advanced security methods and addressing dynamic reconfiguration of the network in the event of topology changes.

Thesis Supervisor: Nursultan Kabylkas
Title: Assistant Professor

Thesis Supervisor: Sain Saginbekov
Title: Assistant Professor

Thesis Supervisor: Akhan Almagambetov
Title: Associate Professor

Acknowledgments

I am deeply grateful to my family and friends for their support, patience and understanding throughout this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid growth of microcontroller-based systems has created various automated environments with which people interact every day, including smart homes, healthcare devices, and smart cities. Such systems utilize sensors and actuators connected to microcontrollers (MCUs) to record data and react to changes in the environment. Microcontrollers are often chosen because of their low cost, small size, and energy efficiency. However, as the scale of such systems increases, limitations of microcontrollers such as low computational capabilities and limited memory become apparent. Hence, the common solution in Internet of Things (IoT) systems is the offloading of computations and storage of data to some remote centralized infrastructure (fog or cloud computing).

There exists an alternative approach that can help solve the limitations of scaling, fault tolerance, and reliability. Instead of introducing a centralized agent who manages all the nodes, the nodes can manage communication with each other through distributed algorithms in the form of a distributed operating system. A distributed operating system helps to consolidate management of the nodes and create a framework of execution in distributed environment. This decentralized approach allows nodes to distribute computations while maintaining coordination with others and execute independent tasks [9, 21, 2].

The implementation of this distributed system includes design and development of operating system kernel, set up inter-process and inter-node communication, and

develop distributed applications for running in the node network.

To mitigate scarce microcontroller resources, this study proposes the exokernel-based design, which provides an architecture-agnostic programming interface while limiting abstraction overhead. Exokernel presents a paradigm shift in operating system architecture design by minimizing abstractions and providing the application with more direct access to hardware resources.

The interprocess communication mechanism is considered the heart of the distributed system because it is the main method available for the nodes to communicate with each other and maintain the system in a consistent state [4]. The design of the IPC must be efficient and robust, as it directly affects the performance of the system [4].

Applications are designed to perform specific tasks and make their functionality available to the network as services. For example, if a node is given a complex task that requires more resources than it can provide, it can offload the task to other nodes, which might be more efficient in the execution of the given specific operations.

This study contributes Orbit - a distributed operating system for RISC-V microcontrollers based on exokernel and written in Rust. The main features of Orbit are lightweight design, versatility, and scalability [19].

Chapter 2

Related works

2.1 Background

2.1.1 Exokernel

The exokernel is a special type of operating system kernel. Compared to more conventional kernel types such as monolithic, microkernel, or hybrid kernel, the main feature of exokernel is that it allows an application to communicate with hardware of the computer in a more direct way, bypassing several layers of abstractions that are present in modern kernels.

Exokernel has two main mechanisms: resource management and resource protection. The meaning of these mechanisms can be broken down into the following: resource management gives the exokernel mechanism to separate hardware resource management from higher-level abstractions. Instead of the traditional approach where the operating system provides a set of abstractions like files, processes, and memory, exokernel exposes raw hardware resources (such as CPU, memory, disks, peripherals, etc.) to applications. This allows applications to have direct control over these resources, enabling them to implement their own resource management policies.

Resource protection maintains control over the protection and security of the system. Although applications have direct access to hardware resources, the exokernel ensures that one application cannot interfere with or access the resources of another

application without proper permissions [13, 5]. This protection mechanism is crucial for maintaining the integrity of the system and preventing unauthorized access or process interference.

Furthermore, to allow applications to benefit from abstractions, exokernel can be extended via library operating systems (LibOS). These LibOS provide a common interface to hardware resources to reduce code repetition when, for example, multiple applications use the same classes of methods to abstract from the hardware.

This exokernel approach was pioneered by Engler, Kaashoek, and O'Toole in the late 1990s with the ExOS project, an operating system based on exokernel, which demonstrated the potential of exokernel to reduce software overhead and optimize resource utilization [5]. However, despite such advances, there has been limited research on the other benefits and use cases of exokernel. Therefore, this study proposes exploring plausible advantages of exokernel in a constrained concurrent environment such as a distributed microcontroller network.

2.1.2 Operating systems

In the broader context of distributed operating system (DOS) research, efforts have focused on systems with remotely connected nodes, enabling communication and resource sharing to function as a unified entity. In the early days of DOS research, such systems as Amoeba, Sprite, and Helios laid the foundation for distributed file systems [22]. However, most of DOS architectures take the microkernel approach. Moreover, developed solutions focus on general-purpose computers and lack adaptation for constrained devices such as microcontrollers.

Communication in a distributed network of constrained devices requires thorough consideration to achieve efficiency. Existing communication protocols such as TCP or UDP are widely used, but introduce high overhead in the system and may be inefficient for microcontroller networks. The operating system for limited devices should include a low-latency, resource-efficient interprocess communication protocol. Due to the different configuration of available microcontrollers, the decision of the exact protocol relies on the availability of network resources of a given MCU because,

for example, not all of them may have wireless network capabilities. Therefore, a custom IPC protocol may be developed as part of this study.

However, distributed protocols are complex and can hide potential security or reliability risks if not implemented and tested properly. Therefore, researchers of the OpenComRTOS operating system focused on formal specification and verification during the development[16]. This is the second generation of OS for the Rosetta spacecraft. To improve the design of the OS and potentially decrease its size, the authors decided to use the model-based approach. That is, they applied formal methods to verify different properties of the system. Consequently, this approach resulted in a ten-fold reduction in code size and a significant improvement in performance.

2.1.3 RISC-V ISA

The RISC-V instruction set architecture (ISA) is specified in two volumes: Unprivileged and Privileged Architecture [6, 7].

Unprivileged Architecture

The unprivileged architecture explains the base integer instruction set which includes `RV32E`, `RV32I`, `RV64I`, and `RV128I`. The base instruction set defines commands for integer arithmetic, branching, load/store operations, jumps, etc. Extensions present additional commands, which can be implemented by the vendor. Extensions can provide operations for multiplication and division, floating-point arithmetic, atomics, bit manipulations, etc.

Privileged Architecture

To ensure the security of the system, the RISC-V architecture introduces several security mechanisms. They are Privilege Modes and Physical Memory Protection (PMP).

Privilege modes allow OS developers to restrict access to some registers and memory locations in different scenarios. The RISC-V Instruction Set Manual Volume

II presents three modes of operation: Supervisor, Machine, and User modes [7]. Although there are three modes specified by the instruction set architecture, most microcontroller vendors implement at most two of them (that is, machine and user modes) or only the user mode.

Table 2.1: RISC-V CSRs [7]

CSR	Description
marchid	Architecture number register
mimpid	Hardware implementation numbering register
mstatus	Status register
misa	Hardware instruction set register
mtvec	Exception base address register
mscratch	Machine mode staging register
mepc	Exception program pointer register
mcause	Exception cause register
mtval	Exception value register
pmpcfg<i>	PMP unit configuration register
pmpaddr<i>	PMP unit address register
dcsr	Debug control and status registers
dpc	Debug mode program pointer register

Aside from regular 32 registers, RISC-V privileged machine mode adds Control State Registers (CSR). These registers are used to maintain security and increase awareness of the state of the system. CSRs specified by the RISC-V ISA are described in Table 2.1.

Furthermore, vendors who implement the RISC-V core may include additional CSRs. The main difference between CSRs and regular registers is that CSRs can be read or modified only in machine mode. For this reason RISC-V ISA adds CSR-specific instructions such as `csrr` to read CSR, `csrw` to write CSR, etc. If machine CSRs are accessed in user mode, the CPU core raises an exception. These registers play a valuable role in the control flow and security of the operating system implemented in this study.

Figure 2-1: pmpcfg register format [7]

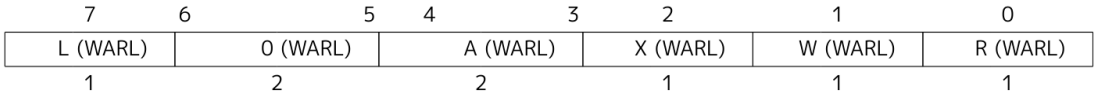
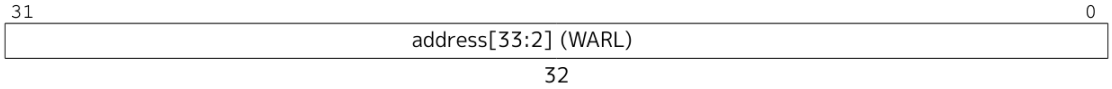


Figure 2-2: pmpaddr register format [7]



PMP is a special part of the CPU core that manages the memory accesses of the programs. PMP consists of two types of registers: PMP configuration register (*pmpcfg*) and PMP address register (*pmpaddr*). The content of these registers is presented in Figures 2-1 and 2-2.

Table 2.2: pmpcfg register field description

Field	Description
R	read permission
W	write permission
X	execute permission
A	configuration encoding
L	locking

The *pmpcfg* register has 5 fields which are described in Table 2.2. Notice that bits 5-6 are always set to 0.

The A field determines the type of encoding of the PMP address. There are 4 types of PMP encoding presented in Table 2.3. These types determine how a PMP register captures a memory region. OFF encoding means that a PMP register is not enabled. TOR encoding of a PMP register works with the preceding PMP register to form an address range of arbitrary size. If the PMP register with TOR encoding is

the first PMP entry, it forms a region from 0 to the address given in the PMP register; otherwise, the previous PMP register determines the bottom of the address range. NA4 is used to protect a 4-bytes memory regions, while NAPOT allows protection of memory regions whose sizes are powers of two [7].

Table 2.3: PMP A field encoding types [7]

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, $\geq$ 8 bytes

In summary, the PMP mechanism plays a crucial role in regulating memory access permissions at the CPU level. A detailed understanding of these registers is essential in configuring memory protection in RISC-V-based systems.

2.2 Study Comparison

The development of exokernel-based distributed operating systems combines two study fields, OS design and distributed systems, each having their own history and challenges. This review of the literature covers the main work in exokernel design, distributed algorithms, and networking in constrained environments. Analysis of each area reveals research gaps and shows where the contribution of this study is focused.

Table 2.4: Comparison of existing operating system design features

OS	Exokernel	Distributed IPC	Rust	Microcontroller support	Modularity	RISC-V support
Linux [15]	✗	✗	✗	✓	✗	✓
Windows [23]	✗	✗	✗	✗	✗	✗
ExOS [10]	✓	✗	✗	✗	✗	✗
L4 [14]	✗	✗	✗	✓	✗	✓
Minix [20]	✗	✗	✗	✗	✓	✓
Barrelfish [1]	✗	✓	✗	✗	✓	✗
TockOS [11]	✗	✗	✓	✓	✓	✓
BernRTOS [8]	✗	✗	✓	✓	✓	✓
OpenComRTOS [16]	✗	✓	✗	✓	✓	✓
This study	✓	✓	✓	✓	✓	✓

For implementation, this work proposes the use of the Rust programming language. Rust is a relatively new system programming language that allows writing equally fast code, compared to C. Furthermore, due to Rust's design oriented to memory safety (safety through ownership) and compile-time type verification, it guarantees the memory safety and robustness of the code. Burtsev et al. claim that the use of Rust in kernel implementation can enforce security through the ownership system and heap isolation [3]. Furthermore, Rust's special ownership system allows for a straightforward implementation of the exokernel resource protection methods.

There have already been developments in implementation of OS using the Rust programming language. For example, TockOS demonstrates the design of a secure embedded OS with an emphasis on connectivity, energy efficiency [11]. Although it does not implement the exokernel approach, it innovates in separating trusted and unreliable kernel code into separate, secured capsules [11].

BernRTOS is another microcontroller operating system developed using the Rust programming language. Similarly to TockOS, the BernRTOS kernel follows a microkernel approach. That is, the kernel manages memory protection and process scheduling. To demonstrate modular design, BernRTOS provides an opportunity to use third-party components for system enhancement. For example, users are allowed to use hardware abstraction layers (HAL) or board support packages (BSP) alongside the kernel to address application needs on a specific board computer [8].

It is evident that Rust is extensively applied in the area of embedded systems. However, research on the exokernel-based distributed operating system developed in Rust remains limited.

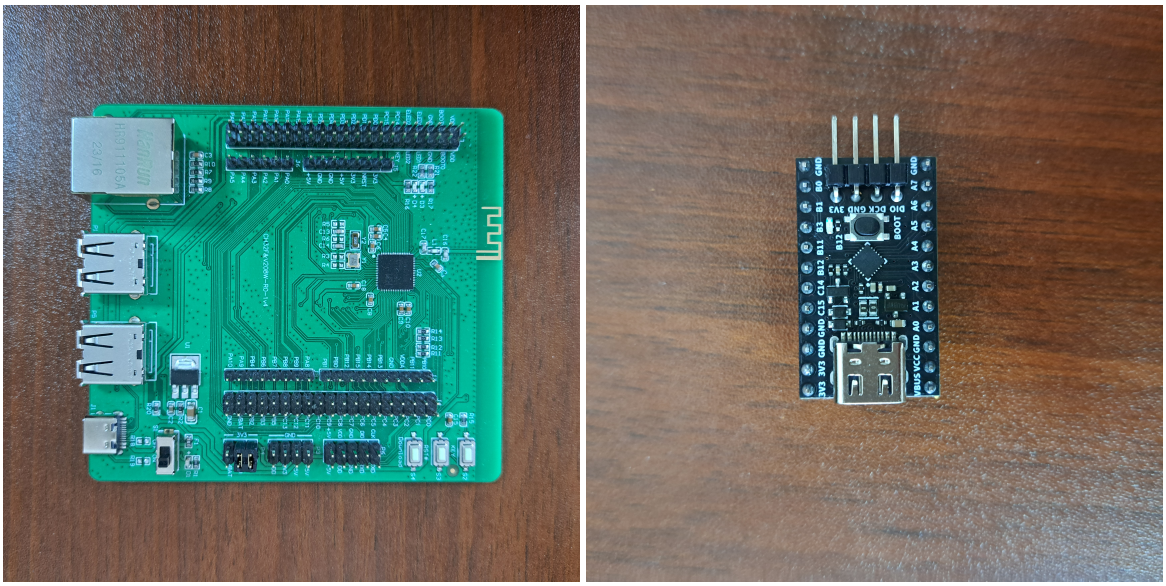
Table 2.4 compares the availability of the features of this study with other operating systems. It shows significant differences with general-purpose operating systems such as Linux and Windows. Other operating systems are more related to the work of this study, with TockOS, BernRTOS, and OpenComRTOS being the closest examples to the domain of this project. However, this study innovates by taking different approaches in kernel design and inter-node communication.

Chapter 3

Methodology

3.1 Development setup

During the development of Orbit, two WCH-produced microcontrollers were used. They are: `CH32V208WBU6` (Figure 3-1a) and `c32x035` (Figure 3-1b).



(a) `CH32V208WBU6` (b) `CH32X035`

Figure 3-1: Microcontrollers used in development

The Table 3.1 illustrates the differences between these microcontrollers. As shown, the `CH32V208WBU6` MCU offers greater performance and resources compared to its

counterpart. Nevertheless, the operating system kernel must remain compact and lightweight to accommodate the constraints of memory-limited MCUs.

Table 3.1: Comparison of `CH32V208WBU6` and `c32x035` MCUs

MCU	`CH32V208WBU6`	`CH32X035`
Core	Qingke V4C	Qingke V4B
RISC-V Extensions	IMAC	IMAC
PMP registers	4	4
Privilege modes	Machine, User	Machine, User
Peripherals	53	30
Memory	FLASH 144 KiB, RAM 32 KiB	FLASH 62 KiB, RAM 20 KiB

Additionally, the WCH-Link debugger was used to debug the operating system. With custom version of the OpenOCD tool, the debugger allowed to read microcontroller's registers and investigate memory locations at runtime. This was particularly useful in preventing unspecified behavior from the OS, such as infinite loops or data corruption.

Lastly, a logic analyzer with 8 channels and a 24 MHz frequency was used to record microcontroller signals and evaluate performance relative to time.

Debugger and logic analyzer are shown in Figures 3-2a and 3-2b, respectively.

(a) `WCH-Link` debugger

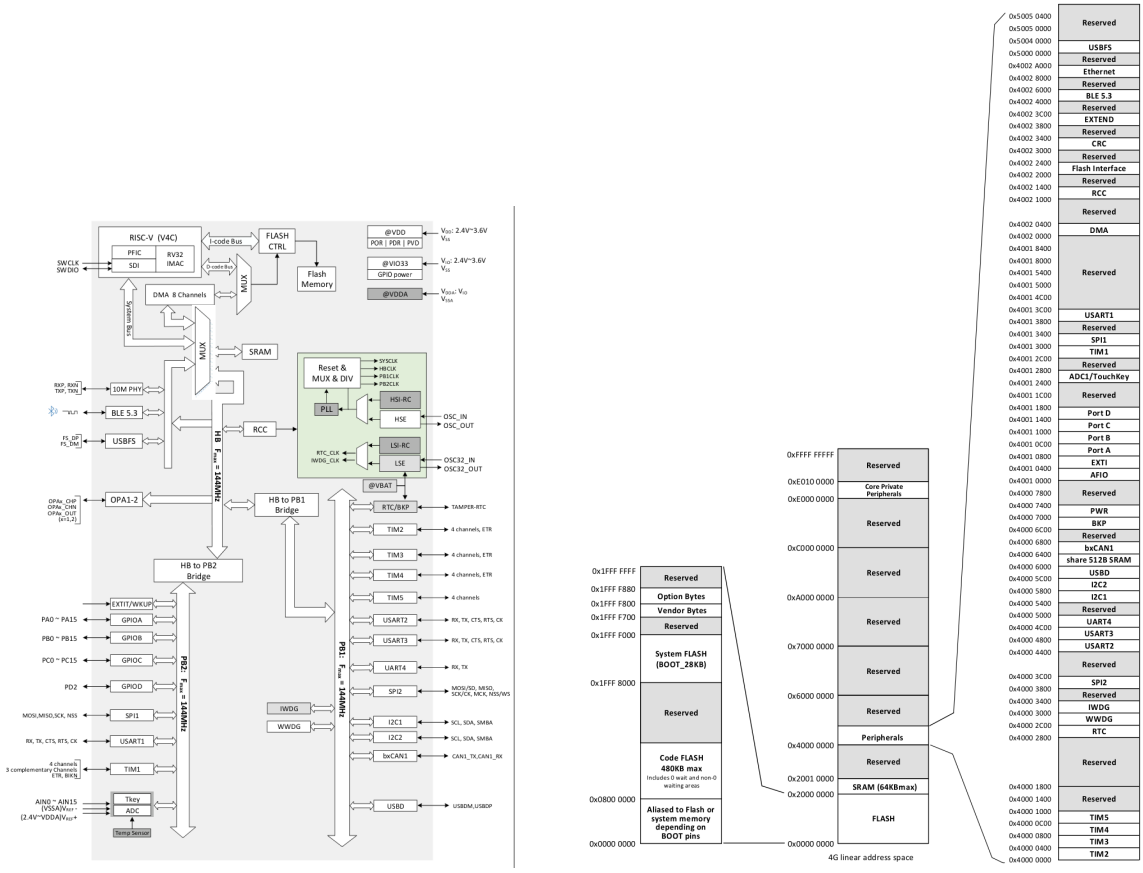
(b) Logic analyzer

Figure 3-2: Additional development tools

3.2 Microcontroller anatomy

Microcontrollers combine multiple small system components. Identifying these components and how their characteristics differ from one MCU to another can help structure the operating system architecture to improve the versatility and modularity.

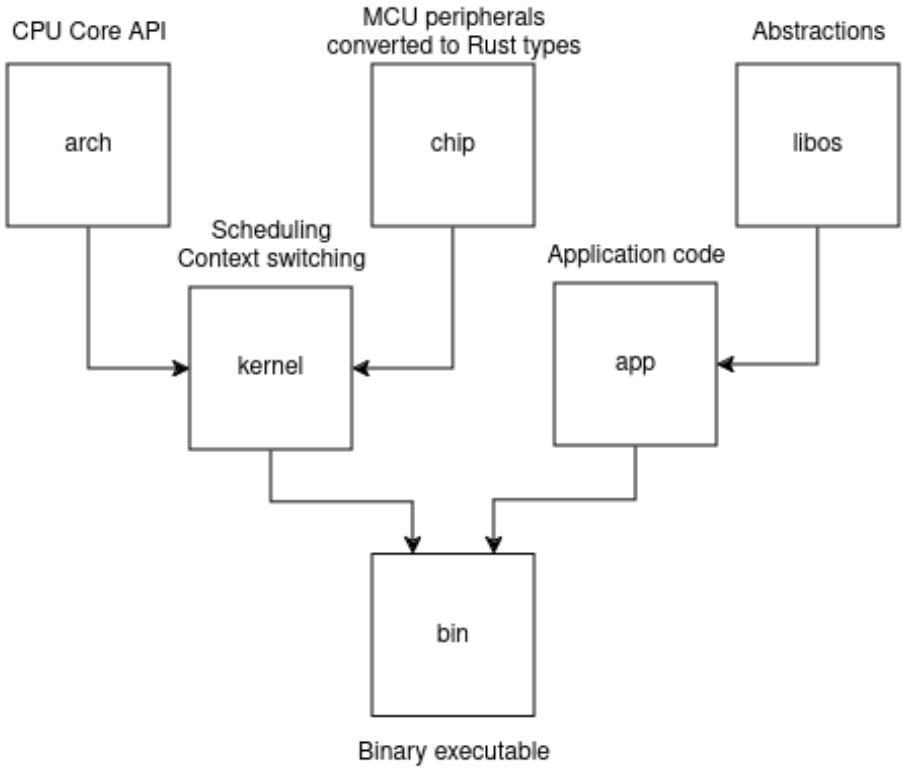
Figure 3-3 illustrates two perspectives on the microcontroller structure: system architecture and memory. The architecture of the system allows us to see how different components of the MCU are related to each other and which buses can access them. In Figure 3-3a we can see the CPU core, flash memory, clocks, and various peripherals. In Figure 3-3b the precise memory regions of each component are illustrated.



(a) `CH32V208WBU6` system architecture (b) `CH32V208WBU6` memory image

Figure 3-3: `CH32V208WBU6` anatomy [18]

Figure 3-4: Project structure



3.3 Project structure

This study takes a structured approach in the design, development, and implementation of a distributed operating system based on exokernel. The project is divided into multiple packages to manage different microcontroller components and create a generic foundation for the operating system kernel. The structure of the project is illustrated in Figure 3-4.

Each package is responsible for a specific use and provides abstractions of different components of the MCU or operating system.

Such a structure allows one to build an operating system for various microcontrollers. It is achieved using a meta-programming technique common for many languages called conditional compilation. In Rust, conditional compilation is provided through special package metadata called *features*. The features in Rust allow the developer to include/exclude a package module, a function, or a block of code to/from the compilation process [12]. Each package of Orbit contains some features. The arch

crate can be compiled based on CPU core while other components are compiled based on the MCU type.

Throughout each package, the linker scripts are used to manipulate memory layout of the final binary. For example, it is preferred for the application code to be located in a continuous memory region rather than being scattered across the memory because we want to use only one PMP register to capture this region. Additionally, linker scripts can insert static variables into memory locations, which can be later imported to Rust code. This is used, for example, to give a unique name to the main function of an application and be able to jump to the function by that name. This approach is particularly beneficial in embedded systems, where resource constraints and memory efficiency are essential.

The packages in Figure 3-4 interact cohesively to form the layered structure of the operating system. Each package is responsible for a distinct layer of functionality, ranging from low-level hardware abstractions to high-level application logic. Together, these components to produce a complete and functional system binary, suitable for deployment on the target microcontroller platform. This separation of concerns aligns with best practices in operating system design, where isolation between kernel services, hardware interfaces, and application logic enhances reliability and security. The following subsections provide a detailed exposition of each package, outlining their responsibilities, interdependencies, and roles within the overall system architecture.

3.3.1 Arch

The arch contains code for specific instruction set architecture and microcontroller core.

Different microcontrollers accommodate different cores. which may have various features. For example, WCH MCU core Qingke V4 implements `IMAC` extensions of RISC-V ISA, has vendor-specific interrupt controller called Programmable Fast Interrupt Controller (PFIC), 4 PMP registers, a total of 256 exceptions, etc. Qingke V2, a less powerful CPU core, on the other hand, implements `EC` extensions and does

not have PMP registers.

In summary, the arch package encapsulates information about the CPU core and provides generic API to the core for the OS kernel.

3.3.2 Chip

The chip package is responsible for wrapping the chip peripherals. Information about peripherals available for a chip, their registers, and memory layout is distributed through CMSIS-SVD files [17]. This information can be easily converted to Rust types through the *svd2rust* tool, where each peripheral is converted to a Rust structure with the peripheral registers being the structure's fields.

Since every microcontroller has different amounts of flash and RAM memory, the chip package also defines the MCU memory layout to produce a linker script. For `CH32V208WBU6` the memory linker script is defined as follows:

```
MEMORY
{
    FLASH : ORIGIN = 0x00000000, LENGTH = 144K
    RAM   : ORIGIN = 0x20000000, LENGTH = 32K
}
```

, where the exact location and length of flash and RAM are indicated.

The package also provides a generic API for the OS kernel.

3.3.3 Kernel

It is the main package of the operating system that is responsible for peripheral initialization, MCU clock configuration, application management, and communication between the distributed operating system's nodes.

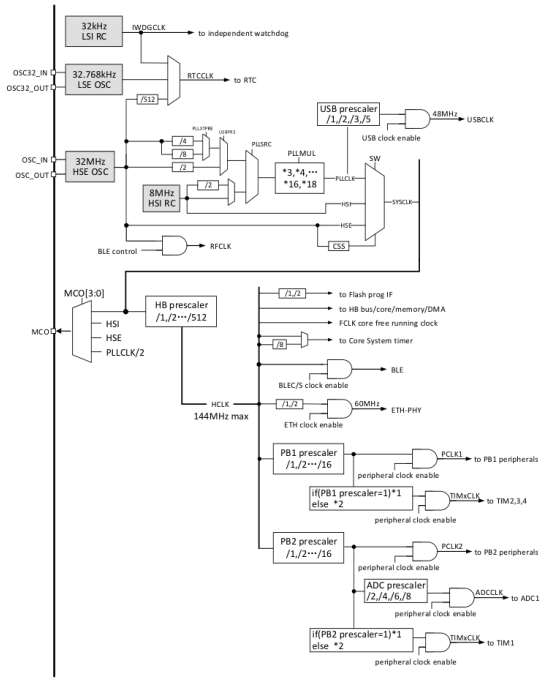
The kernel structure is illustrated in the following code:

```
pub struct Kernel<'k> {
    context: Context,
    apps: [MaybeUninit<AppContainer<'k, PMP>>; APPS],
    running: Option<usize>,
```

```
    ports: [MaybeUninit<Port<'k>>; PORT_NUM],
    pub(crate) peripherals: MaybeUninit<Peripherals>,
    pub core: Core<PMP>,
    pub clock: Clocks,
}
```

The *context* field, which also appears in each application, contains the values of the registers. The *apps* field includes an array of application containers, illustrated in Figure 3-6. The *running* field shows the index of the current running application. The *peripherals* field is a struct of MCU peripherals obtained from the `chip` package. The *core* field provides API to the functions of the CPU core defined in the `arch` package. The *clock* field contains the configuration of the MCU clock, which is used to enable peripherals.

Figure 3-5: `CH32V208WBU6` clock tree [18]



There are multiple settings of clock configuration. For example, the MCU clock can be driven by an internal or external high-speed clock (HSI or HSE). Moreover, the frequency of the clock can be multiplied or divided to achieve a certain value. The reference manual for a microcontroller should indicate possible clock values. In

case of CH32V208WBU6 the clock tree is illustrated in Figure 3-5.

```
pub(crate) enum Message {
    Invoke,
    Reply,
    Busy,
    Unknown,
}
```

The ports field contains an array of the main data channels for the microcontroller network. It is responsible for handling incoming data in the form of *messages*. Message is a special enumerator that creates the foundation for distributed communication.

Currently, Message can represent values shown in the listing above. `Busy` message is used to indicate whether the node is unable to accept the message due to being occupied by processing another message. `Invoke` invokes the application on the MCU by the application name. Applications are able to spawn more messages through system calls. If the application is absent on the node, it can respond with the `Unknown` message. Thus, the kernel port creates the foundation for the event loop that schedules applications based on the message received from the network nodes.

The kernel port is capable of parsing messages with arbitrary length but less than 32 bytes. However, messages must be terminated by the `\0` value.

At this stage of development, the clocks of both `CH32V208WBU6` and `CH32X035` are configured to run on HSI with the smallest available frequency of 8 MHz.

There are some peripherals that should not be exposed to applications because they play a key role in the system functioning. Such peripherals are Reset and Clock Control (RCC) which manages the clocks, PFIC which manages interrupts, Power Control (PWR) which controls low-power modes, and power supply of the MCU. Therefore, to restrict access peripherals, the kernel provides an API to claim a peripheral on a request by an application. The API utilizes Rust's type system to expose peripherals. The mechanism of hiding behavior of types that do not implement a particular interface is called *trait bound* [12]. The Orbit kernel uses special zero-sized trait called *Claimable* to mark peripherals which are exposed by the kernel.

Thus, if the kernel implements the Claimable trait for a peripheral, it is exposed, otherwise, applications will not be able to request the peripheral through kernel API.

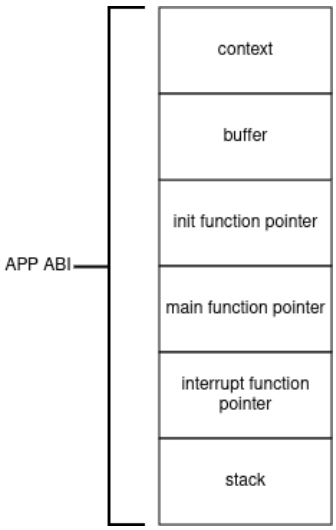
3.3.4 Library Operating System

LibOS in terms of exokernel are special libraries that mimic functionalities of existing operating systems such as Linux, Minix, etc. However, because of the low memory capacity of microcontrollers, it is not possible to fit such a vast amount of abstractions in a regular MCU. Therefore, for this study libOSes play a simpler role. Abstractions used in multiple applications can be combined in a form of library to increase code usability and reduce code space.

However, from a memory layout point of view, the libOS code may not appear near the application code. In this case, an additional PMP register should be occupied to give access to read and execute the libOS code. In systems with a low number of PMP registers, this would be unfortunate. To overcome this, library code can be inlined in the application code to save the PMP register for an additional peripheral.

3.3.5 Application

Figure 3-6: Application container



Orbit introduces a special Application Binary Interface (ABI) for its applications,

illustrated in Figure 3-6. Each application should be represented by a static Rust struct (structure) that contains the application context - set of all CPU registers. Context is necessary to save the application state in case of a CPU interrupt. When an interrupt occurs, the CPU moves to a special location called the trap handler and changes the privilege mode from user to machine but saves the program counter value of the interrupted application in the `mepc` CSR. Hence, the kernel needs to save the application context to handle the interrupt and return to the application afterward.

In addition, the application should have its own stack. The size of the stack is determined by the developer. The minimum requirement for the stack is 4 bytes or 1 word for the RISC-V32 architecture. Lastly, each application must have a buffer in which the kernel will pass the application arguments. Further, developers can add additional data to the application structure according to their needs.

To unify application ABI and prevent developers from manually inserting all the necessary data into the application structure, Orbit provides macros which expand the code during compilation. For instance, an example of an application that blinks an LED is shown in the following code listing:

```
use orbit_common_proc_macro::{app_init, app_interrupt, app_main,
    orbit_app};
use orbit_kernel::{arch::interface::timer::Timer, chip::pac::GPIOB};

use crate::{app_stack, KERNEL};

app_stack!(64, "blinky");

#[orbit_app(GPIOB)]
pub struct Blinky {}

impl Blinky {
    #[app_init("blinky")]
    pub fn init(&mut self) {
        let gpiob = unsafe { self.gpiob.assume_init_mut() };
        gpiob.modify(|p| {
            p.cfghr.write(|w| unsafe { w.bits(0b0101) });
```

```
17              p.bshr.write(|w| unsafe { w.bits(1 << 8) });
18          });
19      }
20
21      #[app_interrupt("blinky")]
22      pub fn interrupt(&mut self) {}
23
24      #[app_main("blinky")]
25      pub fn main(&mut self) {
26          let gpiob = unsafe { self.gpiob.assume_init_mut() };
27
28          gpiob.modify(|p| {
29              p.bshr.write(|w| unsafe { w.bits(1 << 24) });
30              unsafe { KERNEL.core.timer.delay(100000) };
31              p.bshr.write(|w| unsafe { w.bits(1 << 8) });
32          });
33      }
34  }
```

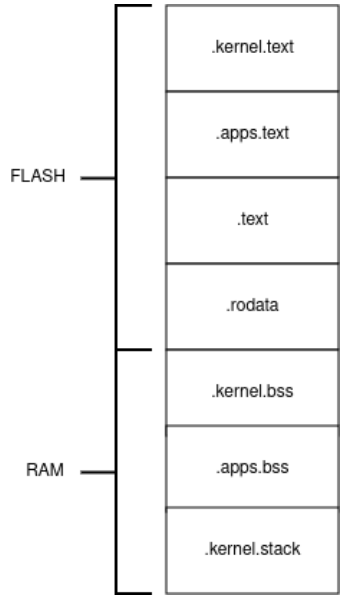
The application uses multiple macros: `orbit_app`, `app_init`, `app_interrupt`, `app_main`, `app_stack`. `orbit_app` is used to create the application structure, configure the context and buffer, and register MCU peripheral. In this example, the application claims GPIO port B. Later, the application is capable of configuring peripheral registers in the init method and performing manipulations with it in interrupt and main methods.

During compilation, the macros used in this application are expanded. The expanded version is shown in Appendix B.1. As can be seen, by using macros we can eliminate the amount of code needed to write an application while preventing the developer from corrupting application sections critical for ABI.

3.3.6 Bin

Lastly, the bin package is used to produce the final binary image. It uses multiple linker scripts produced by the previous packages to unite memory regions. In the

Figure 3-7: Final binary image



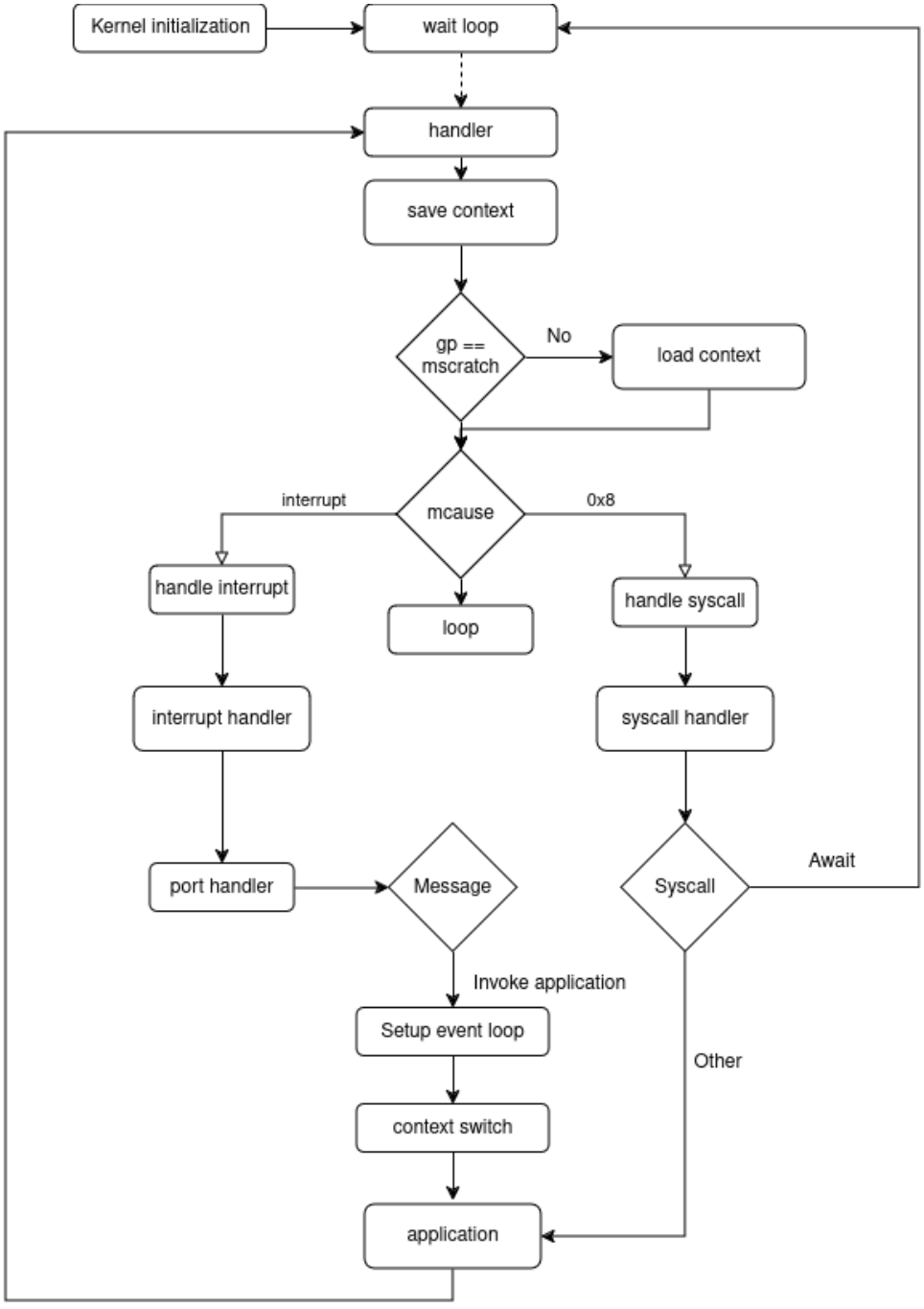
end, the binary takes the shape illustrated in Figure 3-7 with the memory address increasing from top to bottom. The code is contained in the `.text` section while uninitialized data is included in `.bss` sections. The applications code as well as the application data are separated from the kernel to define the memory region of each application and configure the PMP registers when the application runs.

Similarly to application code, the bin package also uses macros to shrink the code and eliminate repetitions. For this, the bin package used the `orbit_main` macro which takes application types as arguments. The example of a binary source code for the `ch32v208wbu6` microcontroller is shown in Appendix B.2, while the expanded code is shown in Appendix B.3. During compilation, the bin package parses the application types from the `orbit_main` macro and creates linker scripts for each application.

3.4 Event loop

Applications in this operating system are driven by interrupts. This means that code is executed whenever the CPU receives an interrupt assigned to a peripheral allocated to the application. When this happens, the interrupt is handled by the exception handler and invokes the application.

Figure 3-8: Operating system's event loop



The Exception (Trap) handler is a special mechanism to react to exceptions and interrupts. RISC-V ISA describes different types of exceptions that may cause system malfunction. They include: instruction address misalignment, load/store access error, illegal instruction, environment call, etc. Each exception and interrupt has a specific exception code. There are two types of exception handlers: direct and vectored. The direct exception handler sets the program counter to the memory address inside the *mtvec* CSR. The vectored exception handler does the same but adds offset of the exception code multiplied by 4. For example, a "load instruction access address misalignment" exception has an exception code 4. If *mtvec* is set to the memory location of $0x800$, the vector exception handler will jump to the memory address $0x800 + 4 * 0x4 = 0x810$.

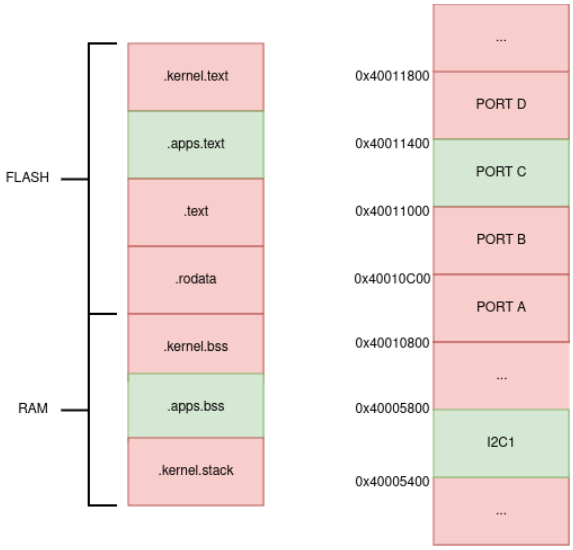
Orbit implements a direct exception handler. The kernel determines the interrupt by exception code and invokes the application to which the interrupt is assigned.

The event loop of the operating system is illustrated in Figure 3-8. When the kernel handles control to the application and vise versa, the registers of one entity need to be stored to load registers of another entity. This procedure is performed in the functions *save context* and *load context*. After the *save context* procedure the exception is handled in the kernel context.

When the kernel calls an application, it configures `mepc` and `mstatus` CSRs. `mepc` holds the pointer to the main function of the application. When kernel loads application's registers and calls the `mret` instruction, which changes privilege mode from machine to user and jumps to the memory location in `mepc` CSR.

When the application finishes running, it produces a *system call* (syscall) via the `ecall` instruction which calls the higher privilege mode environment by jumping to the exception handler in machine mode. The kernel provides multiple system calls for applications. They include: `Return`, `NumPorts`, `SendAll`, `Await`. The `Return` syscall is emitted at the end of application execution; `NumPorts` gives the application the number of ports available on the MCU; `SendAll` allows the application to distribute messages with some data to neighboring nodes via available ports; `Await` stops the application execution until all neighboring nodes reply to the message.

Figure 3-9: PMP configuration for application



3.5 Security

The kernel assigns the PMP configuration for each application. To ensure safety, one PMP is used to access the application code in the **.text** memory region with configuration that allows read and execute; and one for the application data in the **.bss** memory region with configuration for reading and writing. The kernel mostly uses the NAPOT encoding to secure memory regions of 1024 bytes because most peripherals' registers are aligned to this value. A possible PMP configuration is illustrated in Figure 3-9. There, the green blocks represent the memory regions to which an application has permission to execute or read/write.

3.6 Use Cases

The Orbit kernel provides a framework to perform distributed operations across the network of nodes. Hypothetical use cases of such framework are diverse and increase proportionally with the scale of the network of nodes. In small-scale configurations, the framework can support applications such as distributed sensing, actuator coordination, and real-time control in embedded systems. At larger scales, it can enable more complex scenarios, including decentralized data aggregation, federated learning

across edge nodes, fault-tolerant control systems, and collaborative decision-making in autonomous systems.

Chapter 4

Results

4.1 Theoretical soundness

The developed system has rigorous design principles, which integrate advanced concepts in operating system architecture and distributed embedded systems. The system attempts to solve fundamental challenges in resource-constrained devices with limited memory, computational power, and peripheral interfaces. The exokernel approach reduces the system complexity associated with traditional monolithic and microkernel architectures, allowing applications to directly manage resources.

The system design follows a structured model of distributed computing, leveraging inter-node communication. The message passing approach enables reactive communication between the nodes, providing a shared resource to exchange common state and application data.

In addition, the Rust programming language employed in system design provides memory safety guarantees. Moreover, Rust's ownership and borrowing model alongside zero-cost abstractions ensures that memory management is efficient and safe. Having this as a foundation reduces the risks of common low-level errors, such as buffer overflows or invalid memory access, which are particularly problematic in embedded systems.

4.2 Architectural design

Architecture of the system follows a modular and layered design, which allows the separation of concerns between peripheral initialization, distributed communication, and application execution. In addition, it allows similar microcontrollers to reuse common components. For example, if they have identical CPU core or peripheral registers. In the latter case, the common abstractions can be isolated into a library operating system to increase modularity.

The system incorporates application isolation through hardware memory protection achieved by utilizing RISC-V physical memory protection registers. This mechanism protects application code from accessing restricted memory areas and peripheral registers from being altered by other applications.

4.3 Performance and size

The performance evaluation of this system must be considered based on multiple divisions that reflect the main goals of the design. They include minimal overhead, fast execution, safe peripheral access, and scalable distributed coordination. Given the system's unique position as being distributed exokernel targeting RISC-V microcontrollers, traditional OS benchmarks must be adapted to emphasize latency, determinism, and resource utilization rather than the raw throughput or multitasking performance, which are often used to evaluate general-purpose operating systems.

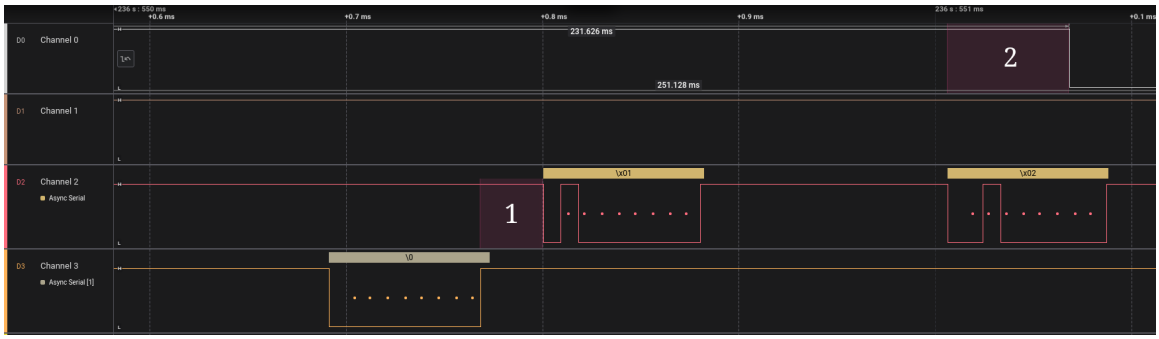
The main benchmark on the local system level is the context switch overhead. Since the Orbit kernel avoids many abstraction levels and does not implement complex system services, the performance of switching between applications is increased. Additionally, Rust's zero-cost abstractions ensure that the kernel or application API does not provide additional overhead beyond what is explicitly implemented.

Table 4.1: Average measurements of message handling and context switch

Measurement	Duration (μs)
Message handling	31
Context switch	59

Peripheral acquisition is only relevant during kernel initialization. At this stage, the kernel allocates certain peripherals to applications. If there happens to be a request on an already claimed peripheral, the kernel will abandon the application. After kernel initialization, applications are isolated from each other and cannot access each other's peripherals.

Figure 4-1: Experiment evaluating context switch performance and message delivery reliability



In distributed dimension, performance evaluation focuses on inter-node communication latency and message delivery reliability. To measure these attributes, the following experiment was conducted: in a system consisting of two nodes, one sends messages to invoke the application, which toggles the GPIO pin, on the other. The time measurements were captured using a logic analyzer and presented in Figure 4-1. There are two regions in Figure 4-1 denoted as **1** and **2**. They represent the time to react to an incoming message and the time to perform a context switch and run an application that pulls a GPIO to ground, respectively. This scenario has lasted for 15 minutes. The average values of both metrics are shown in Table 4.1.

The experiment has been carried out on the hardware that was used for development, namely, `ch32v208wbu6` and `ch32x035` microcontrollers with both having a CPU

frequency of 8 MHz. Since the maximum frequency is 144 MHz for `ch32v208wbu6` and 48 MHz for `ch32x035`, the overall performance of the MCUs can be improved by increasing the CPU frequency.

Another experiment can be carried out to evaluate performance gains from distributing some calculation across two nodes. The experiment consists of calculation three expressions of form

$$\text{(N1 Op1 N2) Op2 (N3 Op3 N4)} \tag{4.1}$$

where $N\#$ are hexadecimal numbers and $Op\#$ is any from the operators `+`, `-`, `*`, `/`.

(a) Scenario 1

(b) Scenario 2

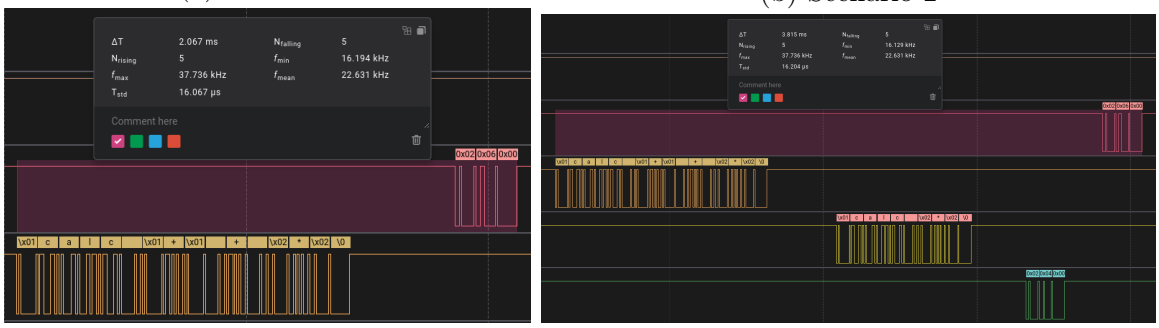


Figure 4-2: Offloading experiment

The expression 4.1 requires first calculating expressions in parentheses and then apply the *Op2* operator on the results of these expressions. These procedures were performed in two scenarios. In the first scenario, only `CH32V208WBU6` is used to calculate the expressions. In the second scenario, `CH32V208WBU6` offloads the second expression in parenthesis to `CH32X035` while calculating the first expression in parentheses. Upon receiving the result from `CH32X035`, `CH32V208WBU6` calculates the final value of the expression.

The results show that time scenarios one and two are $2.07ms$ and $3.82ms$ on average out of 10 measurements, respectively. Samples of the measurements of both scenarios are illustrated in Figures 4-2a and 4-2b.

The distribution of the calculation to other nodes requires additional time for the

(a) Message transmission time

(b) Message transmission and response reception time

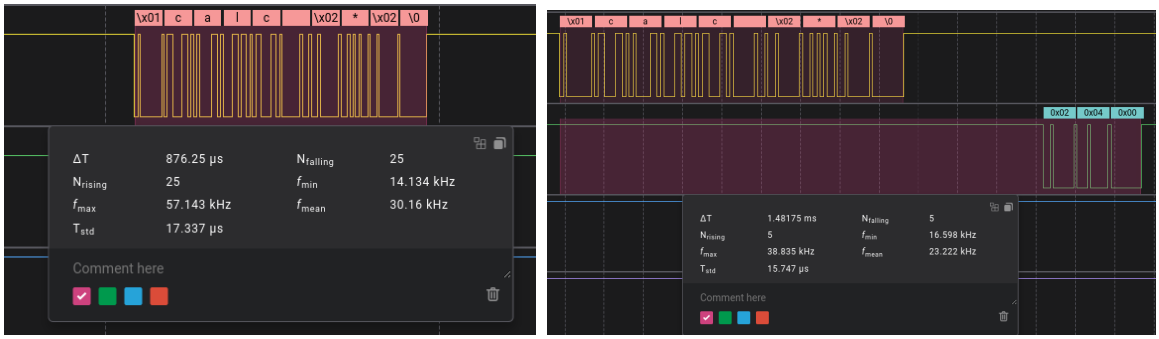


Figure 4-3: Message offloading time

transmission of the task and the reception of its result. These transmissions add to overall calculation time (Figure 4-3). Hence, the first scenario performs better than the second one. However, offloading computations allows performing other tasks while awaiting the response. Therefore, it is a more flexible approach on the scale of the node network.

Regarding binary size, the final kernel for both microcontrollers takes up approximately `3.5` KiB, including both FLASH and RAM memory. The `calc` application's size is `1.7` KiB.

Chapter 5

Conclusion

This thesis presented the design, implementation, and performance evaluation of a distributed exokernel-based operating system for RISC-V microcontrollers. Motivated by the growing need for decentralized coordination and resource sharing in embedded systems, this study challenges conventional monolithic and microkernel designs by embracing exokernel principles, minimizing abstraction overhead, and delegating resource management to applications. The OS is implemented in Rust to take advantage of the language's type system and memory safety features.

The architectural choices and implementation techniques used in this thesis were evaluated using both analytical and empirical methods. The results demonstrate that the system achieves low latency peripheral access and minimal kernel overhead.

5.1 Limitations

The limitations of this work arise mainly from design trade-offs to achieve minimality and modularity as well as time constraints. Although this distributed exokernel demonstrates strong practical feasibility, some limitations must be acknowledged in terms of application flexibility, security, and scalability.

First, applications are currently constrained by the kernel infrastructure. Although the kernel is capable of running the application on an external message request, applications lack proper handling of arguments, return values, and system calls.

In addition, dynamically uploading an application to the kernel is not supported. Instead, all applications must be statically compiled in the OS binary image.

Second, the kernel does not apply security measures to the distributed communication layer. Therefore, messages handled by the kernel can cause unpredictable consequences if they are exploited aggressively. For example, initiating flooding attacks could potentially overload node's kernel or even disconnect a portion of a network.

Third, due to the lack of time and variety of microcontrollers, this work was unable to properly evaluate the performance of large network of microcontrollers. Although this experiment would help identify weak sports in distributed communication and local node performance, it may provide an opportunity for further research.

5.2 Future work

Future work may extend this research in multiple directions. From a systems perspective, efforts could focus on dynamic node discovery, application migration, and cross-node memory sharing mechanisms. In addition, work could be done to improve the security in decentralized communication through access control policies and fault tolerance. For example, consensus algorithms such as Raft or Paxos could be implemented to maintain the shared state of the MCU network.

This work could also be extended by studying power consumption of Orbit kernel during various modes of operation, as power consumption is considered one of the main limitations of embedded devices.

Another area of contribution is infrastructure. To increase the usefulness of the Orbit kernel, it needs to support other microcontroller families. This may include other MCUs from WCH as well as microcontrollers from other vendors such as Espressif, Artinchip, GigaDevice, or Renesas. As such, Orbit would be capable of uniting more diverse hardware devices and be generally available for distributed systems research activities or industrial use cases.

Appendix A

Abbreviations

- ABI - application binary interface
- BSP - board support package
- CSR - control and status register
- CPU - central processing unit
- GPIO - general purpose input-output
- GPIOB - general purpose input-output port B
- HAL - hardware abstraction layer
- HSI - high-speed internal
- HSE - high-speed external
- IoT - internet of things
- IPC - inter-process communication
- MCU - microcontroller unit
- PMP - physical memory protection
- PFIC - programmable fast interrupt controller

- PWR - power control
- RCC - reset and clock control

Appendix B

Source code snippets

B.1 Blinky application expanded during compilation

```
pub mod blinky_v208 {
    use orbit_common_proc_macro::{app_init, app_interrupt, app_main,
    orbit_app};
    use orbit_kernel::{arch::interface::timer::Timer, chip::pac::
   GPIOB};
    use crate::{app_stack, KERNEL};
    #[used]
    #[link_section = ".blinky.bss"]
    pub static mut STACK: [usize; 64] = [0; 64];
    use crate::application::Application;
    use core::{
        arch::{asm, naked_asm},
        mem::MaybeUninit, sync::atomic::compiler_fence,
    };
    use orbit_kernel::{
        application::Context, claim::{Claim, Claimed,
   KernelPeripherals},
        port::{RINGBUF_SIZE, RingbufType, ringbuf::RingBuf},
    };
    #[used]
    #[unsafe(no_mangle)]
```

```
19      #[unsafe(link_section = ".blinky.bss.struct")]
20      static mut BLINKY: MaybeUninit<Blinky> = MaybeUninit::uninit();
21      #[repr(C, align(4))]
22      pub struct Blinky {
23          context: Context,
24          _buf: RingBuf<RINGBUF_SIZE, RingbufType>,
25          gpiob: MaybeUninit<Claimed<'static, GPIOB>>,
26      }
27      impl Blinky {
28          #[unsafe(link_section = ".blinky.text")]
29          pub fn _init(&mut self) {
30              unsafe extern "C" {
31                  static _app_blinky_text_start: usize;
32                  static _app_blinky_text_end: usize;
33                  static _app_blinky_bss_start: usize;
34                  static _app_blinky_bss_end: usize;
35                  static _app_blinky_text_main: usize;
36                  static _app_blinky_bss_struct: usize;
37              }
38              let provides = unsafe {
39                  &_app_blinky_text_end as *const usize as usize
40                      | &_app_blinky_text_start as *const usize as
    usize
41                      | &_app_blinky_text_end as *const usize as usize
42                      | &_app_blinky_bss_start as *const usize as
    usize
43                      | &_app_blinky_bss_end as *const usize as usize
44                      | &_app_blinky_text_main as *const usize as
    usize
45                      | &_app_blinky_bss_struct as *const usize as
    usize
46              };
47              self.context = Context::new();
48              self.context.t0 = provides;
49              compiler_fence(core::sync::atomic::Ordering::SeqCst);
50              self.context.t0 = 0;
```

```
51              self.context.sp = unsafe {
52                  STACK.last().unwrap_unchecked() as *const usize as
    usize + 0x4
53              };
54              self.context.gp = &self.context as *const Context as
    usize;
55              self.context.ra = Self::ecall as *const fn() as usize;
56              self._buf.buf.iter_mut().for_each(|i| *i = RingbufType::
    default());
57          }
58      }
59      impl Application for Blinky {
60          #[inline(never)]
61          #[unsafe(link_section = ".blinky.text")]
62          fn _main(&mut self) {
63              unsafe { asm!("li a0, 0;li a1, 0;") };
64          }
65          #[inline(never)]
66          #[unsafe(link_section = ".blinky.text")]
67          fn context(&self) -> Context {
68              self.context
69          }
70          #[naked]
71          #[unsafe(link_section = ".blinky.text.ecall")]
72          extern "C" fn ecall() {
73              unsafe { asm!("ecall") };
74          }
75      }
76      impl Blinky {
77          #[unsafe(link_section = ".blinky.text")]
78          pub fn init(&mut self) {
79              self._init();
80              {
81                  let gpiob = unsafe { self.gpiob.assume_init_mut() };
82                  gpiob
83                      .modify(|p| {
```

```
                        p.cfghr.write(|w| unsafe { w.bits(0b0101) })
    ;
                        p.bshr.write(|w| unsafe { w.bits(1 << 8) });
                    });
            }
        }
        #[unsafe(link_section = ".blinky.text.interrupt")]
        pub fn interrupt(&mut self) {
            {}
            unsafe { asm!("li a0, -1; li a1, 0;") };
        }
        #[inline(never)]
        #[unsafe(link_section = ".blinky.text.main")]
        pub fn main(&mut self) {
            let output = {
                let gpiob = unsafe { self.gpiob.assume_init_mut() };
                gpiob
                    .modify(|p| {
                        p.bshr.write(|w| unsafe { w.bits(1 << 24) })
    ;
                        unsafe { KERNEL.core.timer.delay(100000) };
                        p.bshr.write(|w| unsafe { w.bits(1 << 8) });
                    });
            };
            self._main();
        }
    }
}
```

B.2 Binary source code for ch32v208wbu6

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

use orbit_app::{blinky_v208::Blinky, calc::Calc};
```

```rust
use orbit_bin::orbit_main;

orbit_main!(Blinky, Calc);
```

B.3 Binary source code for ch32v208wbu6 (expanded)

```rust
#![feature(prelude_import)]
#![no_std]
#![no_main]
#[prelude_import]
use core::prelude::rust_2024::*;
#[macro_use]
extern crate core;
use orbit_app::{blinky_v208::Blinky, calc::Calc};
use orbit_bin::orbit_main;
use orbit_common_proc_macro::orbit_main_attribute;
use orbit_kernel::kernel::Kernel;
use orbit_app::application::Application;
unsafe extern "Rust" {
    static mut KERNEL: Kernel<'static>;
    static mut BLINKY: Blinky;
    static mut CALC: Calc;
}
#[unsafe(no_mangle)]
#[unsafe(link_section = ".text.bin")]
unsafe fn main() -> ! {
    KERNEL.clock.freeze();
    BLINKY.init();
    KERNEL
        .add_application(
            0usize,
            unsafe { &BLINKY as *const Blinky as usize },
            Blinky::main as usize,
            Some(Blinky::interrupt as usize),
            BLINKY.context(),
```

```
        );
    CALC.init();
    KERNEL
        .add_application(
            1usize,
            unsafe { &CALC as *const Calc as usize },
            Calc::main as usize,
            Some(Calc::interrupt as usize),
            CALC.context(),
        );
    KERNEL.initialize();
    {
        #[cold]
        #[track_caller]
        #[inline(never)]
        const fn panic_cold_explicit() -> ! {
            ::core::panicking::panic_explicit()
        }
        panic_cold_explicit();
    };
}
```

Bibliography

[1] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, Big Sky, MT, USA, 2009.

[2] U. Brinkschulte, A. Bechina, F. Picioroaga, and E. Schneider. Distributed real-time computing for microcontrollers-the osa+ approach. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 169–172, Washington, DC, USA, 2002.

[3] A. Burtsev, V. Narayanan, Y. Huang, K. Huang, G. Tan, and T. Jaeger. Evolving operating system kernels towards secure kernel-driver interfaces. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 166–173, New York, NY, USA, 2023.

[4] Hamed Dinari. Inter-process communication (ipc) in distributed environments: An investigation and performance analysis of some middleware technologies. *International Journal of Modern Education and Computer Science (IJMECS)*, 12(2):36–52, 2020.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. *SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[6] A. Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. RISC-V International, version 20240411 edition, 2024. [Online]. Available: `https://riscv.org/technical/specifications`. [Accessed: 20-Mar-2025].

[7] A. Waterman et al. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, version 20240411 edition, 2024. [Online]. Available: `https://riscv.org/technical/specifications`. [Accessed: 20-Mar-2025].

[8] A. Galliker. Bern rtos: Kernel architecture and development framework.

[9] Branden Ghena, Jean-Luc Watson, and Prabal Dutta. Embedded oss must embrace distributed computing. In *Next Generation Operating Systems for Cyber-Physical Systems (NGOSCPS)*, Montreal, Canada, April 2019.

[10] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1995.

[11] G. J. Kambhampati, B. C. Kuszmaul, and T. W. Grossman. Tock: A secure embedded operating system for microcontrollers. In *Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW)*, pages 1–5, San Jose, CA, USA, 2017.

[12] Steve Klabnik and Carol Nichols. The rust programming language, 2025. Accessed: 20-Mar-2025.

[13] Tim Leschke. Achieving speed and flexibility by separating management from protection: Embracing the exokernel operating system. *SIGOPS Operating Systems Review*, 38(4):5–19, 2004.

[14] J. Liedtke. On microkernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250, Copper Mountain, CO, USA, 1995.

[15] Linux Archives. Linux kernel source code archives. [Online]. Available: `https://www.kernel.org`. [Accessed: 20-Mar-2025].

[16] H. Mampaey, E. A. de Kock, and P. J. M. Havinga. Opencomrtos: A portable message-passing multi-processor rtos. In *Proceedings of the International Workshop on Hardware/Software Co-Design (CODES/ISSS)*, pages 73–77, 2005.

[17] Trevor. Martin. *The designer's guide to the Cortex-M processor family : a tutorial approach*. Gale eBooks. Elsevier/Newnes, Amsterdam ;, 1st edition edition, 2013.

[18] Nanjing Qinheng Microelectronics. *CH32F/V20x_ V30x_ V31x Reference Manual*, 2023. Accessed: 20-Mar-2025.

[19] Zakhar Semenov. orbit. `https://github.com/Wooker/orbit`, 2025. GitHub repository.

[20] A. S. Tanenbaum, A. S. Woodhull, and A. S. Herder. *Operating Systems: Design and Implementation*. Prentice-Hall, 3rd edition, 2006.

[21] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, MA, 5th edition, 2022.

[22] Andrew S. Tanenbaum, René van Renesse, Henri E. Bal, Johannes J. Jansen, Guido van Rossum, J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Computers and Communications*, 14(6):25–35, 1991.

[23] Microsoft Windows. Windows kernel source code archives. [Online]. Available: `https://www.microsoft.com/en-us/windows` [Accessed: 20-Mar-2025].