Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

Academic Year 2024/2025

Graduation Session October 2025

Exploring RUST-based AUTOSAR-compliant Operating Systems for Embedded Processors

Supervisors:

Dr. Corrado DE SIO
Prof. Sarah AZIMI
Prof. Luca STERPONE

Candidate:

Federico RIVOIRA

Abstract

Electronics and software are essential components in the modern automotive industry, where safety, correctness, and real-time performance are critical requirements. Traditional automotive software relies on C/C++ due to its high performance, but its unrestricted syntax and manual memory management make it highly error-prone, increasing development costs and risks, especially for large projects. Rust is a promising alternative, a mature programming language that offers comparable performance and eliminates entire classes of memory and concurrency errors through strict compile-time rules. The language is gaining traction in the context of embedded systems, supported by an expanding ecosystem of projects and libraries.

This thesis aims to demonstrate the feasibility of using Rust in real-time, safety-critical embedded software through the development of a Rust-based operating system compatible with the widely adopted AUTOSAR standard. The project targets a development board featuring the real-time profile Cortex-R52 processor and is also compatible with QEMU. The core component of the system is a priority-based scheduler driven by a hardware timer on the CPU. AUTOSAR objects that define more advanced features such as synchronization mechanisms between tasks and cores, and configurable interrupt routines were implemented, along with a comprehensive set of APIs to expose them to the user code, which remains in C/C++ for compatibility.

The final outcome includes build tools, examples, and documentation. It establishes a robust foundation that demonstrates the practical applicability of Rust to real-time, safety-critical embedded systems. By providing a working prototype aligned with AUTOSAR principles, the project offers a starting point that can be extended and refined for future research and industrial adoption.

Acknowledgements

I would like to express my sincere gratitude to Dumarey for making this project possible and for the support provided throughout its development. I am especially grateful to Salvatore Parisi, Roberto Morone, and Filippo Parisi for their guidance and valuable insights.

I am deeply thankful to my supervisors, Corrado De Sio, Sarah Azimi and Luca Sterpone, for their guidance and support during this research. I am particularly grateful to Corrado De Sio, who directly oversaw this project and was consistently available and supportive, making a significant impact on both my work and personal growth.

I would also like to extend my heartfelt thanks to Emanuele Messina, who worked alongside me on this project. His dedication, collaboration, and expertise were essential to the success of this thesis.

Table of Contents

Chapter 1

Introduction

The increasing complexity of modern vehicles has made electronics and software indispensable components of the automotive industry. Advanced driver-assistance systems (ADAS), infotainment platforms, and electronic control units rely heavily on embedded software to guarantee performance, reliability, and safety. In such systems, meeting real-time constraints and ensuring predictable behavior are fundamental requirements, as any malfunction may compromise both system integrity and passenger safety.

Traditionally, the automotive software stack has been dominated by C and C++, languages valued for their efficiency and close interaction with hardware. However, their low-level nature comes with inherent drawbacks. Aspects such as unrestricted pointer manipulation, manual memory management, and the absence of strict concurrency safety mechanisms make these languages prone to subtle bugs and undefined behavior. As systems grow in scale and complexity, these issues increase development costs, extend testing cycles, and raise certification challenges for the required safety standards.

Rust has emerged as a promising alternative for developing embedded and safety-critical applications. It combines performance comparable to C/C++ with a strong focus on memory safety, concurrency correctness, and software reliability. By enforcing strict compile-time checks, Rust prevents entire classes of runtime errors, such as null pointer dereferencing, data races, and buffer overflows. Over the past few years, the language has gained significant traction in the embedded domain, supported by a growing ecosystem of tools, crates, and community-driven projects that simplify cross-compilation, hardware abstraction, and real-time development.

1.1 Motivation

In traditional C development, software safety is typically ensured through strict coding conventions, such as MISRA C, that programmers must manually adhere to or through static analysis performed by sophisticated and often costly tools. These approaches, while effective to a degree, are prone to human error and require significant effort to maintain compliance throughout the development lifecycle. Rust has been proven to reduce the reliance on such practices through its built-in safety mechanisms. [1]

The adoption of Rust offers significant advantages in the development of inherently complex, safety-critical software. Studies in the context of real-time operating systems have shown that up to 54% of reported vulnerabilities originate from memory corruption issues, which the safety model of Rust effectively mitigates. [2]

However, despite these clear benefits, the transition to Rust in production environment, and particularly in the automotive sector, has been slow. This is largely due to the absence of domain-specific solutions, standardized development methodologies, and established best practices.

Broader adoption of Rust in domains such as the automotive industry, along with emerging efforts to introduce the language in neighboring sectors such as aerospace [3, 4], could accelerate its use across the embedded systems field and software development in general. This convergence of adoption across safety-critical domains is expected to contribute to improved reliability and security in critical applications,.

1.2 Goal

This thesis aims to demonstrate the feasibility of using Rust in the development of real-time, safety-critical embedded software by exploring the existing ecosystem of tools, libraries, and methodologies, and by developing an operating system compatible with the AUTOSAR standard, widely adopted in the development of automotive ECUs.

The project targets a real-time hardware platform based on the ARM Cortex-R52 processor, representative of products currently used in the industry. To facilitate integration with existing software, and for adherence with the specification, the system is designed to maintain compatibility with C applications, allowing it to be used alongside existing codebases and tools. At the same time, the kernel itself is fully developed in Rust, providing stronger guarantees of safety and reliability. This approach represents a balanced compromise, simplifying adoption in established workflows while leveraging the key advantages of Rust to enhance software robustness.

By delivering a fully functional prototype aligned with AUTOSAR design principles, and by formulating practical development guidelines, this work ultimately seeks to build a solid foundation for the integration of Rust in industrial automotive software workflows and to encourage its broader adoption in the embedded systems domain.

1.3 Outline of the thesis

The thesis work has been divided into the following chapters:

- **Background:** Provides an overview of related topics, including embedded systems, Rust, operating systems, and the AUTOSAR platform.
- **State of the Art:** Reviews existing AUTOSAR-based solutions and resources for embedded systems development in Rust.
- **Development Methodology:** Describes the development process of the operating system, including initial experiments, implementation of AUTOSAR features, and the build system.
- **Testing and Evaluation:** Explains the testing strategies, deployment and performance evaluation on the target board.
- **Conclusion and Future Works:** Concludes the thesis by summarizing the work, key design decisions, contributions, and future improvements.

Chapter 2

Background

The first part of this chapter offers an overview of embedded systems, beginning with the main hardware components and placing particular emphasis on the Cortex-R52, followed by an outline of the software development principles and workflow.

The second part shifts to the software domain, providing a review of the features of Rust, a discussion of the essential concepts of operating systems for embedded devices, and an introduction to the AUTOSAR platform.

2.1 Embedded Systems

Embedded systems are special-purpose computing systems designed to interact closely with external electronic components, such as displays, sensors, and actuators. They are often built to perform specific tasks with high efficiency and reliability. Common examples include smart appliances, bank ATMs, video game consoles, industrial robots, network routers, and automotive ECUs.

Unlike general-purpose computing systems, embedded systems are typically constrained in terms of processing power, memory, and energy consumption, requiring careful hardware and software design. Depending on the application domain, the design priorities may emphasize different aspects, such as power efficiency, real-time performance, and security making these systems highly optimized for their intended functions.

2.1.1 System on a Chip (SoC)

A System on a Chip (SoC) is an integrated circuit that incorporates all the essential components of a computing system on a single chip, including CPU cores, RAM, flash memory, and various peripherals. By integrating these elements into a single package, SoCs achieve high levels of power efficiency and compactness, making

them ideal for embedded systems, mobile devices, and other applications where size and energy consumption are critical considerations.

Microcontrollers are a subset of SoCs that integrate a low-power processor core. They are optimized for control tasks, low energy consumption, and cost efficiency. They typically have limited computational power, smaller memory, and fewer advanced features compared with high-performance CPUs.

Unlike modular designs, where individual components are connected separately, SoCs trade off some flexibility and upgradability for a smaller footprint, reduced power usage, and improved communication speed between components. This approach enables manufacturers to deliver highly optimized and cost-effective solutions for a wide range of computing tasks.

2.1.2 Printed Circuit Board (PCB)

Printed Circuit Boards (PCBs) are flat boards made of insulating materials with copper pathways that connect electronic components. They can have one or more layers, with connection between layers made with through-holes, small copper-plated drilled holes that carry signals and power.

In the context of embedded systems, PCBs are commonly referred to simply as boards, and they host the SoC along with supporting components, including the power supply system, memory, and ports.

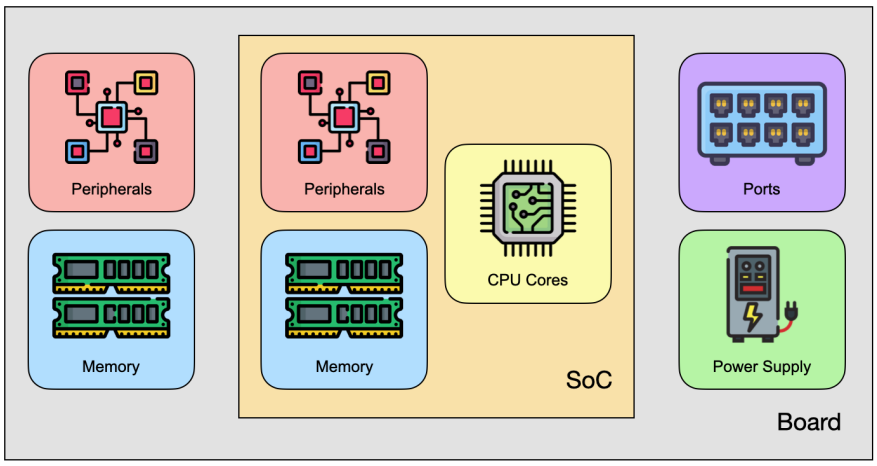

Figure 2.1: Diagram showing the relationship between board, SoC, and CPU cores.

A specific category is represented by development boards, which are intended for evaluation and testing and typically provide a wide range of components. In contrast, production devices usually employ custom PCBs that are tailored to the

requirements of the final product.

2.1.3 Peripherals

Peripherals are hardware modules that extend the functionality of the CPU core. They are controlled by means of registers on the module that are used both for configuration and to exchange data. These registers are accessed through standard read and write operations at specific, reserved memory addresses with a technique known as memory mapping.

Peripherals signal events to CPU cores through interrupts, which are first handled by a controller module. When an interrupt occurs, the normal instruction flow is suspended and a configured handler function is executed. Each interrupt has a priority level that determines the order in which interrupts are handled. Higher priority interrupts can interrupt lower priority ones, allowing nested execution.

ARM architectures distinguish between two types of peripherals:

- **Private Peripherals:** tightly coupled with the core, typically integrated with the CPU.
- **Shared Peripherals:** accessible to the whole system, typically installed on the SoC or the board.

Configurable Software Generated Interrupts (SGIs) are also supported. These interrupts are generated with specific instructions and share the same behavior as traditional hardware interrupts [5].

This project focuses on two peripherals: the Universal Asynchronous Receiver-Transmitter (UART), and the Generic Timer.

- **UART:** shared peripheral which converts data into a serial stream that can be read from a USB port for debugging purposes.
- **Generic Timer:** private peripheral used to generate periodic interrupts.

2.1.4 Cortex-R52

The ARM Cortex-R52 [6] is a high performance processor designed for time-sensitive and safety-critical applications. It implements the ARMv8-R 32-bit architecture and integrates a set of hardware modules commonly found in other ARM processors, including the following:

- **Vector Floating Point (VFP):** coprocessor that provides hardware acceleration for floating-point arithmetic operations.

- **Memory Protection Unit (MPU):** configurable unit that enforces memory access permissions on non-overlapping memory regions.
- **Generic Interrupt Controller (GIC):** centralized component that prioritizes, routes, and forwards peripheral interrupts to the appropriate CPU core.

On ARM architectures, the following CPU registers are defined:

- **General-Purpose Registers (R0-R15):** General-purpose registers used for data storage and operations. R15 serves as the Program Counter (PC).
- **Link Register (LR):** Stores the return address for function calls and interrupts.
- **Stack Pointer (SP):** Points to the current top of the stack memory, where the compiler typically allocates variables.
- **Program Counter (PC):** Holds the address of the next instruction to be executed.
- **Current Program Status Register (CPSR):** Contains flags for the current processor state, including condition flags and interrupt disable bits.

The VFP uses a separate set of registers for floating point operations. The stack pointer and link register are banked, meaning that each processor mode has its own separate copy of these registers. This allows the processor to switch quickly between modes, such as during exceptions or interrupts, without overwriting the values used in another mode.

The Cortex-R52 was chosen as the target architecture for this project due to its real-time capabilities, determinism, and safety features, which make it well-suited for applications in the automotive sector. The target board hosts the ST Stellar SR6P6 SoC, which integrates six Cortex-R52+ cores in a multi-cluster configuration.

2.1.5 PYNQ Boards

PYNQ is a lineup of development boards produced by AMD that feature ARM Cortex-A and Cortex-R CPU cores along with FPGAs. The development workflow is based on the Vitis software suite. The PYNQ-Z2 board with the Cortex-A9 was used as a hardware reference in the early stages of the project to compile and debug programs. Plans were made to move to the PYNQ-ZU with the Cortex-R5 due to its similarity to the target architecture.

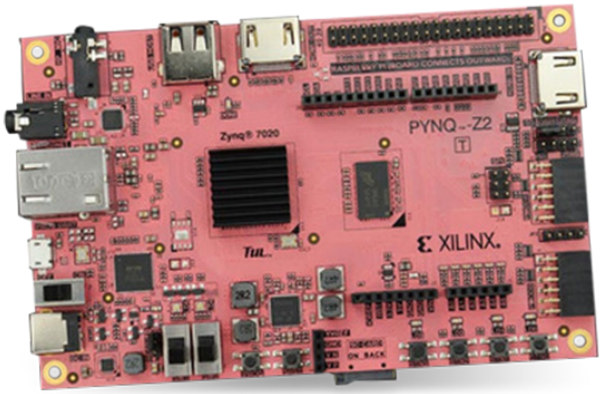

Figure 2.2: The PYNQ-Z2 development board.

2.2 Embedded Software Development

In embedded systems development, the program is typically written and built on a host machine, a general-purpose computer running an operating system such as Windows or Linux. The host provides the computational resources and development tools needed to create code efficiently. The software produced on the host is not executed directly on the same system but is instead designed to run on a target platform, which is a specific board or SoC with its own hardware architecture. Because the host and target architectures differ, software cannot be compiled natively for the target processor. Instead, a cross-compilation process is used, where the compiler running on the host generates executable code for a different instruction set architecture (ISA). The compiler must be aware of the target architecture, including its instruction set, calling conventions, and memory model. Cross-compilation ensures that the generated binary is compatible with the target hardware while allowing the developer to benefit from the performance and flexibility of the host system. The collection of programs installed on the host that enables cross-compilation is known as the toolchain. A typical embedded toolchain includes a compiler, assembler, linker, debugger, and additional utilities for code analysis or binary inspection. The GNU Arm Embedded Toolchain is one of the most common examples, though many vendors provide custom versions optimized for their devices. Compiled programs are often stored in the Executable and Linkable Format (ELF), a standardized file format that contains machine code, data sections, and metadata such as symbol tables and relocation information. Once built, the program must be transferred from the host machine to the memory of target system with a process known as flashing. This process is performed

through a hardware debug interface on the board, such as JTAG, using specific tools provided by the board manufacturer or third-party vendors.

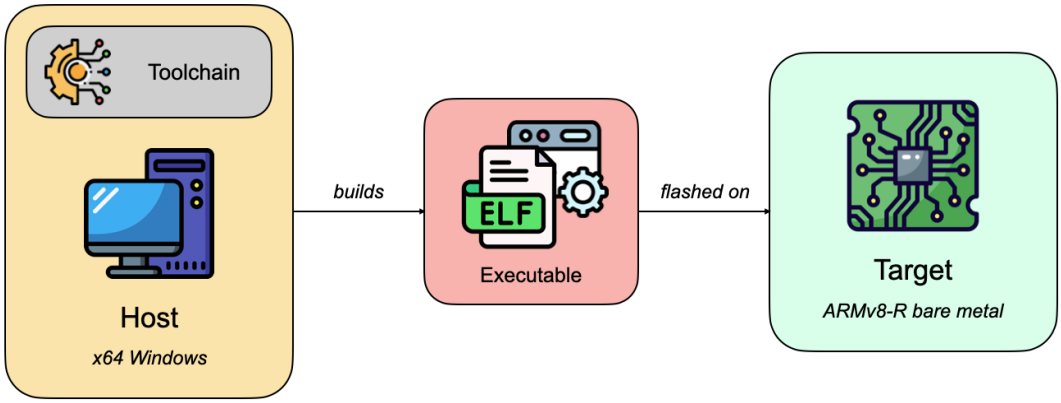

Figure 2.3: The typical embedded software development workflow.

2.2.1 Bare Metal

A bare metal program runs directly on the hardware without the support of an operating system or runtime environment. All hardware resources, such as the CPU, memory, and peripherals, are controlled explicitly by the program itself. This approach is typical in deeply embedded applications where the software performs a single, well-defined function and timing or resource constraints make the use of an OS unnecessary. System-level software such as bootloaders, kernels, or firmware components are also inherently bare metal, as they execute before any operating system is initialized. Bare metal programs do not have access to standard libraries or operating system services. In the Rust ecosystem, for instance, such programs are typically built with the `no-std` attribute, which disables dependencies on the standard library and allows the use of lightweight core libraries. These provide only essential functionality such as basic data types and mathematical operations, while system-level features like file I/O, dynamic memory allocation, and multithreading must be implemented explicitly.

Memory management in a bare metal environment is entirely under application control. The layout of memory, including sections for code, data, and stack, is defined in a linker script, which specifies how program sections are mapped to physical memory regions. Since dynamic allocation can introduce non-deterministic behavior and is often unavailable on small devices, data structures are typically allocated statically at compile time. If heap allocation is required, it must be implemented manually, usually by defining a custom memory allocator that interfaces

directly with the available RAM.

The startup procedure of a bare metal system is handled by a small assembly or startup file that executes immediately after reset. This code performs minimal hardware initialization, sets up the stack pointer, initializes CPU registers, and configures the vector table containing interrupt and exception handlers. It may also copy data from flash to RAM and clear uninitialized memory sections before transferring control to the main program entry point, usually defined as the `main` function in C or Rust.

2.2.2 Software Development Kit (SDK)

The Software Development Kit (SDK) is a comprehensive software package provided by the manufacturer of a board, designed to simplify development and accelerate the creation of applications. It provides a collection of functions, libraries, and abstractions that make it easier to interact with CPU cores, memory subsystems, and hardware peripherals such as timers, communication interfaces, and sensors. The SDK typically includes low-level boot code to initialize the processor and essential hardware modules, device drivers for standard and board-specific peripherals, build tools to compile and link programs, documentation describing the hardware and software interfaces, and example programs demonstrating common usage patterns. By providing these resources, the SDK allows developers to quickly familiarize themselves with the board, test hardware functionality, and prototype applications without needing to implement low-level routines from scratch, making it a critical component in embedded software development.

The SDK of the target board, entirely written in C, was fully integrated in the software stack. This integration enabled direct access to vendor-provided drivers both from the kernel and from the application, while the Rust micro-architecture crate was used to interact with CPU cores and configure modules such as the GIC and MPU.

2.2.3 QEMU

QEMU is an open-source emulator that provides full-system emulation of complete hardware platforms, reproducing processors, memory subsystems, and peripheral modules entirely in software. Unlike instruction set simulators that emulate only the CPU, QEMU models the behavior of an entire board, enabling developers to run and debug complete embedded applications without access to physical hardware. This makes it a fundamental tool in early-stage development, testing, and continuous integration for embedded systems.

Internally, QEMU operates using two main execution modes: system emulation and user-mode emulation. System emulation reproduces the entire hardware

platform and is typically used for operating system or firmware development, whereas user-mode emulation runs individual programs compiled for a different architecture within the host OS. In both cases, QEMU employs dynamic binary translation, converting target instructions into equivalent sequences for the host CPU at runtime. This approach maintains accurate architectural behavior.

An important feature of QEMU is its integration with the GNU Debugger (GDB) through a remote debugging interface. This allows developers to connect to an emulated target as if it were a physical device, enabling capabilities such as setting breakpoints, stepping through instructions, inspecting registers, and examining memory regions. Combined with symbol information from the compiled ELF binary, this feature provides deep visibility into program execution and simplifies fault diagnosis during development.

QEMU supports a wide range of processor architectures, including ARM, RISC-V, x86, PowerPC, and MIPS, as well as numerous development boards and SoC configurations. Among these, the ARM MPS3-AN536 platform featuring a dual-core Cortex-R52 processor is particularly relevant for real-time and safety-critical applications. This board model provides an accurate representation of the core architecture, memory regions, and essential peripherals such as timers, UART interfaces, and interrupt controllers, making it suitable for testing embedded kernels and operating systems.

Overall, QEMU offers a flexible and cost-effective platform for developing, testing, and debugging embedded software. Its combination of hardware accuracy, cross-architecture support, and integration with standard toolchains makes it an indispensable component of modern embedded development workflows.

During this project, software emulation with QEMU was employed extensively in the early development phases, when the target board was not yet available. The emulator provided a stable and repeatable environment for testing the system. This approach accelerated development by enabling rapid iteration, debugging, and functional validation without requiring access to physical hardware. Although QEMU cannot reproduce precise real-time behavior, it offered sufficient accuracy to verify system logic and ensure architectural consistency. The final version of the OS maintains full compatibility with QEMU, which continues to serve as a regression-testing and prototyping platform alongside the target hardware.

2.3 Rust

Rust is a low-level, general-purpose programming language first released in 2012, designed with a strong focus on memory safety, reliability, and concurrency. Its core design philosophy aims to provide the performance and control typical of system programming languages while eliminating classes of memory-related errors

that often lead to instability and security vulnerabilities.

A central feature of Rust is the borrow checker, a component of the compiler that enforces strict ownership and lifetime rules. These rules ensure that instructions interact with memory safely, preventing common issues such as null pointer dereferences, data races, and use-after-free errors. Because this verification is performed entirely at compile time through static analysis, the resulting binaries do not incur additional runtime overhead. As a result, Rust achieves performance comparable to other low-level languages such as C, while offering stronger guarantees of safety and correctness.

Rust development is supported by a modern toolchain that simplifies the build and maintenance process. Cargo, the Rust package manager and build system, automates project configuration, dependency management, and compilation. It also supports custom build scripts written in Rust and provides access to a large ecosystem of reusable code modules and libraries, known as crates, which can be integrated to extend functionality. Complementing this, rust-analyzer acts as a language server that assists developers by providing context-aware suggestions, inline diagnostics, and error highlighting to improve productivity and code quality.

Rust was selected for this project because its emphasis on safety and correctness aligns with the requirements of real-time and safety-critical embedded systems. These properties make it particularly suitable for developing a reliable, secure, and maintainable kernel that integrates low-level control with modern software engineering practices.

2.3.1 Foreign Function Interface (FFI)

The Rust Foreign Function Interface (FFI) provides a standardized mechanism for interoperability between Rust and other programming languages. It defines conventions for function calling, symbol linkage, and data type representation, ensuring that code written in different languages can communicate safely and efficiently. Through the FFI, existing libraries that are not natively implemented in Rust can be integrated into a Rust project, allowing developers to reuse mature, well-tested components. Conversely, Rust functions can also be exposed to other languages, enabling mixed-language systems where Rust modules coexist with legacy or platform-specific code.

Functions and data structures that belong to the FFI domain are declared using the `extern` keyword, which specifies the expected calling convention and linkage behavior. External symbols defined in the foreign language must be declared in Rust through bindings, which act as a bridge between the two environments. These bindings can be written manually by the developer or generated automatically during the build process using dedicated tools. The `bindgen` utility is commonly employed for this purpose, as it parses C header files and produces corresponding

Rust bindings, reducing inconsistencies and manual errors.

Interfacing across language boundaries introduces challenges related to memory management and safety. Because ownership and lifetime checks of Rust do not extend beyond the FFI boundary, developers must ensure that data alignment, allocation, and deallocation follow the conventions of both languages. Neglecting these aspects can result in undefined behavior or memory corruption, undermining the safety guarantees that Rust normally enforces.

In this project, the integration of C code from the SDK was performed using the `bindgen` crate to generate Rust bindings automatically from the provided headers. The FFI was also used in the opposite direction, allowing selected Rust functions that form part of the operating system API to be accessible from application code written in C. This bidirectional integration enabled seamless communication between system-level components implemented in Rust and application logic developed in C.

2.4 Operating System

An Operating System (OS) is a layer of software that supervises the execution of applications and manages the hardware resources of a computing device. It provides fundamental services such as task scheduling, memory management, and input-output handling, acting as an intermediary between the user application and the hardware. External programs executed under the control of the OS are referred to as user applications, while the core module responsible for resource management and control flow is known as the kernel.

In general-purpose systems such as desktop computers, user applications are developed and compiled as independent programs, which the OS executes and isolates from one another. In contrast, embedded systems typically adopt a more integrated structure. User code is organized as individual functions, referred to as tasks, which are compiled together with the operating system into a single executable binary. The OS provides a lightweight runtime environment, handling the scheduling and synchronization of these tasks while maintaining a small memory footprint suitable for constrained hardware. At the core of the system, the kernel is periodically updated through the system tick, a hardware-generated interrupt that defines the basic time unit of the OS. The tick enables the kernel to maintain control over task execution and to perform time-dependent operations.

Each active task is assigned a dedicated stack memory region, which stores its local variables during execution. The OS also leverages the CPU execution levels, or privilege modes, to distinguish user tasks from kernel operations. Typically, user tasks run in an unprivileged mode, whereas the kernel executes in one or more privileged modes with full access to system resources. The memory unit,

such as the MPU, can be configured to enforce memory access restrictions between these modes, ensuring that each task operates only within its assigned stack and memory region. This separation improves system stability and prevents unintended interference between tasks or with kernel memory. Interaction between user tasks and the kernel occurs through system calls, which allow user code to request privileged services such as task management, event handling, or timing operations. On ARM architectures, these calls are implemented using Supervisor Calls (SVCs), which trigger a controlled switch from user mode to privileged mode. The kernel then executes the requested service and restores the previous execution context before returning control to the task. To support this mechanism, the OS preserves the current state of the CPU registers, collectively known as the context, whenever an interrupt or SVC is processed. Context saving and restoring are typically implemented in assembly language, using the stack to store register values temporarily. This ensures that the interrupted task can resume execution precisely from the point where it was suspended, maintaining system consistency.

2.4.1 Scheduler

The scheduler is a core component of the kernel responsible for managing the lifecycle and execution of tasks. Its primary role is to determine which task should run at a given time and on which CPU core, ensuring that all tasks share system resources efficiently while meeting timing and priority requirements. When multiple tasks are active simultaneously, the scheduler coordinates their execution over the available cores according to predefined rules. This process, known as scheduling, can follow various policies, such as static or dynamic priority levels, round-robin rotation, or time slicing, depending on the requirements of the system.

During normal operation, the scheduler may suspend the execution of a task to allow another one to run. This mechanism, referred to as preemption, enables high-priority tasks or time-critical operations to take control of the CPU when necessary. Preemption involves a context switch where the current state of the running task, represented by the contents of its CPU registers, is saved so that execution can later resume from the same point. The scheduler then restores the context of the next selected task, effectively transferring control of the processor. To support this mechanism, the kernel maintains dedicated data structures that store the context and state information of suspended tasks. These structures ensure that each task can be paused, resumed, or terminated without loss of data or corruption of shared resources. Efficient management of these transitions is essential for achieving deterministic behavior in real-time operating systems, where predictable response times are often as important as raw performance.

2.4.2 Synchronization

Synchronization mechanisms are essential in multitasking systems to prevent conflicts when multiple tasks attempt to access shared resources, such as memory regions, data structures, or hardware devices. The portion of code that performs such access is known as a critical section, where only one task should be allowed to execute at a time. Ensuring exclusive access within this section prevents data inconsistencies and unintended behavior. When two or more tasks concurrently access a shared resource without proper synchronization, a race condition occurs, leading to unpredictable and often non-deterministic results.

In addition to preventing conflicts, synchronization also enables coordination between tasks that must cooperate to achieve a common objective. In such cases, the progress of one task may depend on the outcome of another, requiring mechanisms that control the order of execution and the exchange of signals or data. These mechanisms ensure that dependent operations occur in a consistent and predictable sequence, which is particularly important in real-time and safety-critical environments.

The operating system provides specific abstractions to manage synchronization and protect the execution of critical sections. These abstractions ensure safe access to shared resources and reliable coordination among tasks. In AUTOSAR OS, synchronization mechanisms are represented by events, resources, and spinlocks. Events are typically used for signaling and task coordination, resources protect access to shared data, and spinlocks support synchronization across multiple cores. All these features are accessible to user tasks through supervisor calls, which provide controlled interaction between user code and kernel-level synchronization primitives.

2.4.3 Real-Time Operating System (RTOS)

A Real-Time Operating System (RTOS) is a specialized type of embedded OS designed to guarantee predictable and deterministic behavior, meeting strict timing and real-time constraints. RTOSs are commonly employed in domains where timely execution is critical, including industrial automation, medical devices, aerospace, and automotive systems. Unlike general-purpose operating systems, an RTOS focuses on minimizing latency and ensuring that high-priority tasks execute within their required time windows.

Key features of an RTOS include deterministic preemptive scheduling, typically based on task priority levels, and low-overhead context switching that allows the system to respond quickly to interrupts and external events. Efficient interrupt handling is crucial for maintaining responsiveness and for coordinating the execution of multiple concurrent tasks without violating real-time constraints. Many RTOSs also provide lightweight synchronization mechanisms, timing services, and inter-task

communication primitives to facilitate safe and predictable interaction between tasks.

In addition to AUTOSAR OS implementations, which are widely used in automotive and safety-critical contexts, FreeRTOS represents one of the most broadly adopted small-footprint RTOSs. FreeRTOS is particularly well-suited for resource-constrained systems, including IoT devices, sensor networks, and microcontroller-based applications. It offers a minimal yet flexible kernel, supporting preemptive scheduling, queues, semaphores, and software timers. Its simplicity and portability make it a popular choice for developers who require a lightweight, deterministic operating system without the complexity of a full-featured embedded OS.

2.5 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a global alliance of automotive, electronics, and software firms founded in 2003 to define a common software architecture for ECUs and to promote reuse, standardization, and interoperability across vendors. Over time, AUTOSAR released two main platforms to cover different use cases: the Classic Platform (CP) and the Adaptive Platform (AP). The Adaptive Platform targets high-performance, service-oriented ECUs for applications such as automated driving and connected cars, while the Classic Platform remains the standard choice for ECUs with real-time and safety constraints. This project focuses on the Classic Platform architecture, which is organized into three main software layers:

- **Application Layer:** automotive software, mostly independent from the CPU architecture.
- **Runtime Environment (RTE):** middleware that handles the communication between software components and provides an interface to lower layers for applications.
- **Basic Software (BSW):** low level modules not directly related to the automotive application, including the OS.

In the Classic Platform, applications share the same memory space, with optional support for MPU-based memory protection. AUTOSAR also provides a methodology for system configuration, described via XML (ARXML) files, enabling automation of RTE and BSW generation and strong toolchain interoperability. The AUTOSAR OS specification is an evolution over the earlier OSEK standard. It retains backward compatibility but extends functionality by adding multicore support, timing extensions, and stricter safety features. In particular, release R24-11 of the AUTOSAR OS was selected as the reference for this project. By adopting

the Classic Platform, this project aligns with a widely accepted industry standard for real-time ECUs, leveraging the AUTOSAR architecture, methodologies, and specification guarantees to guide the design and implementation of a Rust-based OS for automotive applications.

Chapter 3

State of the Art

This chapter analyzes existing solutions in the domains of Rust for embedded systems and AUTOSAR-compliant operating systems, including open-source implementations that informed the development of the project and resources for embedded systems in Rust, some of which were integrated in the project. The final section discusses the current state of the Rust community and the ongoing initiatives aimed at incorporating the language into established industry standards.

3.1 AUTOSAR OS Implementations

The available implementations of systems and platforms based on the AUTOSAR specification can be divided in three categories:

- **Commercial Solutions:** Developed by companies such as ETAS and Vector, these are fully supported, feature-complete AUTOSAR implementations, but require licensing fees.
- **Consortium-developed Reference Products:** Developed under the AUTOSAR partnership (e.g. OpenERIKA), mainly as reference or educational implementations, often with limited scope and not always intended for production use.
- **Open-source Implementations:** A few projects exist, such as ERIKA Enterprise and Arctic Core, but they are no longer actively maintained and may not include the latest AUTOSAR features.

Companies typically rely on commercial solutions, which offer support for several hardware architectures and integration with the tools commonly used in automotive software development workflows.

The two main open-source solutions were analyzed, providing an insight into the design and implementation of the modules and features of an AUTOSAR OS.

- **Arctic Core** AUTOSAR-compliant platform developed by ARCCORE, later acquired by Vector in 2018. The source code of the project is released under the GPLv2 license with support for the MPC5xxx and STM32 series microcontrollers. Although Arctic Core is a complete implementation of the standard, the publicly available versions date back to 2014 and do not include important features such as multicore support.
- **ERIKA Enterprise** RTOS developed by Evidence. Although ERIKA is not a direct implementation of the AUTOSAR platform, it includes many features defined in the AUTOSAR OS specification, including recent extensions. This system is OSEK/VDX certified, provides explicit multi-core support and has been deployed in production across automotive and white-goods applications. The Enterprise version, however, is no longer actively maintained as the development shifted towards the OpenERIKA project.

All existing implementations of AUTOSAR-compliant operating systems are written in C. This is in large part because C is the predominant language in the automotive industry, and because the specification itself assumes the use of the language for both the kernel and the API exposed to application code. However, Rust represents a viable alternative for implementing an AUTOSAR OS. Through its support for interoperability with C via the FFI, a Rust-based kernel can seamlessly interact with existing C code and libraries, ensuring adherence to the specification. At the same time, using Rust allows the kernel to benefit from features that enhance safety and reliability, providing a foundation for more robust and maintainable embedded software.

3.2 Rust Crates

Rust is supported by a vast ecosystem of projects and resources for embedded system. In 2024, more than 11.000 crates were already available, including hardware support packages, drivers, utility crates, and systems [2].

A specific category is represented by micro-architecture crates, which serve a similar purpose to a SDK, but more limited in scope. The focus is on functionalities that are specific to the CPU based on its architecture, including direct access to registers and operation of peripherals integrated with the CPU. Typically, the boot code is also provided. The `cortex-m` crate is a notable example. All features are implemented through zero-cost idiomatic abstractions. This approach simplifies the development process, enabling the use of Rust in place of assembly instructions and

providing a direct overview of the low-level features of the CPU and its modules. The development of the `cortex-ar` crate specific for Cortex-R processors started during the early stages of this project. This crate and `arm-gic`, a separate crate that focuses on the GIC module, were integrated with the project and proved essential for managing the complexity of the architecture.

Utility crates offer functionalities that are normally provided by system libraries but without relying on the abstractions of an operating system. For this reason, they are often referred to as *no-std* crates. The supported features include dynamic memory management, network stacks, mathematical operations, and data storage mechanisms. Such crates are useful both in bare-metal programming and in system development, where they can extend the kernel with additional capabilities. Their use enables developers to build complex functionality in environments that lack standard library support, while keeping control over resource usage and hardware interaction.

Among the most widely used utility crates in Rust embedded development, `embedded-hal` provides standardized interfaces for peripherals, enabling hardware-independent driver and application code. This crate defines a set of standardized traits for GPIO, I2C, SPI, UART, and timers, serving as the foundation for many higher-level libraries and operating systems in the Rust embedded ecosystem. By implementing these traits, developers can write portable drivers and applications that work across different architectures without modification.

3.3 Rust Projects

Embedded operating systems and real-time frameworks fully developed in Rust are increasingly available, and studies have analyzed their design, performance, and safety characteristics, highlighting their strengths, weaknesses, and potential application domains [7]. Many of these solutions, however, are primarily aimed at low-power microcontroller devices and may not meet the strict timing and feature requirements of real-time applications. Others are not complete operating systems, but rather frameworks or libraries intended to simplify the development of embedded software. In contrast, industrial sectors such as automotive require domain-specific, production-ready systems that comply with established standards like AUTOSAR. These standards define precise requirements for task scheduling, memory protection, communication, and system reliability, and currently there are no Rust-based solutions that fully satisfy them. This section provides an overview of some of the most relevant Rust-based embedded OSs and frameworks, illustrating the current state of the ecosystem.

- **Tock:** The most prominent embedded OS fully developed in Rust. This open source project targets Cortex-M and RISC-V architectures with an

emphasis on power efficiency and security. Tock relies on the safety features of Rust to provide a multiprogramming environment for microcontrollers, isolating software faults and supporting application workloads written in any language [8]. While Tock is not an RTOS, efforts to extend it with real-time capabilities and other features have been made both in research projects [9] and in commercial products like OxidOS.

- **Ariel OS:** The first embedded OS in Rust with support for multicore preemptive scheduling on microcontrollers. Ariel integrates components from the Embassy framework and other utility crates to provide a complete execution environment with features such as networking, storage, and cryptography. [10]

- **Hubris:** A microkernel style OS designed for deeply embedded and safety critical systems. Hubris follows a strictly static and minimal architecture with a kernel of about 2000 lines of Rust code. It uses preemptive scheduling, and all tasks, memory regions, and priorities are defined at compile time, with no dynamic allocation or runtime task creation allowed. This system relies on hardware memory protection to isolate tasks, running nearly all code in unprivileged mode and reducing the trusted computing base.

- **Drone:** Designed for writing real-time applications in Rust, this OS supports a flexible concurrency model with lightweight stackless tasks and optional stackful tasks for blocking code. A CLI utility simplifies project setup, configuration, and building across different microcontroller targets. The project is no longer actively developed.

- **RTIC:** A lightweight concurrency framework for real-time applications. RTIC leverages hardware interrupt controllers to manage preemptive scheduling with minimal software overhead. It employs the Stack Resource Policy (SRP) to ensure compile-time guarantees against data races and deadlocks, promoting safe concurrency without the need for locks or semaphores. RTIC is well-suited for bare-metal applications requiring deterministic behavior and efficient resource utilization. [11]

3.4 Working Groups and Standardization

Rust Embedded is an official working group of the Rust project that focuses on improving the experience of using Rust on embedded system. This organization develops projects, produces manuals and curates an extensive list of resources which includes crates, drivers, and whole operating systems.

The Safety-Critical Rust Consortium, established in 2024 by the Rust Foundation, brings together organizations from various domains, including AdaCore, Arm,

HighTec, and Ferrous Systems. Its objective is to promote the responsible adoption of Rust in regulated and safety-critical environments by developing guidelines, tools, and collaborations with existing certification standards. Among the members of the consortium, Ferrous Systems contributes with the Ferrocene compiler, a Rust toolchain designed for qualification in safety-critical applications.

Within the automotive domain, AUTOSAR has introduced preliminary support for Rust applications, referred to as ARA applications, in the Adaptive Platform. Automotive software companies such as Vector and HighTec have also presented proof-of-concept integrations that demonstrate the coexistence of Rust components with AUTOSAR-compliant software, highlighting the growing interest in Rust for automotive development. [12]

The presence of active working groups and ongoing projects developed in Rust shows a positive trend and the initiatives led by companies and standardization organizations to ensure compliance with safety and certification standards represent a key step toward integrating Rust into safety-critical fields. However, the lack of domain-specific solutions, particularly AUTOSAR-compliant systems, still limits its broader industrial adoption.

Chapter 4

Development Methodology

This chapter outlines the development process, beginning with the initial experiments and progressing through the implementation of the operating system to its deployment on the target board. Core concepts from the AUTOSAR specification are examined, with particular attention to the strategies adopted for implementing key features. The chapter concludes by describing the build process of the system.

4.1 Preliminary Experiments and Environment Setup

The work commenced with a preliminary investigation focused on three main objectives:

- **Toolchain Setup:** Verifying the functionality of cross-compilation for ARM embedded platforms in bare-metal mode, and ensuring proper integration of the GDB debugger to enable effective program analysis and testing.
- **C Interoperability:** Establishing the integration of C libraries within the development workflow, using automated tools to facilitate linking and maintain compatibility between C and Rust modules.
- **Development Environment:** Selecting and configuring a suitable development environment to begin the development process, given that the target board was not yet available during the early stages of the project.

4.1.1 Toolchain Setup

The experimentation involved the compilation and debugging of simple Rust programs intended for embedded systems, executed on the PYNQ-Z2 development

board. This board integrates a Cortex-A processor and was selected primarily because it was readily available and provided a straightforward means of hardware testing during the initial phase of the project. The presence of an accessible JTAG debugging interface, which could be reached through a micro USB connection, significantly simplified the setup process and allowed a smooth debugging experience. Although the board was not identical to the final target hardware, it represented an adequate platform for practical experimentation and validation of essential concepts on real embedded equipment. The PYNQ family also includes models based on Cortex-R processors, which share several architectural and tooling similarities with the intended target architecture. Consequently, working with the PYNQ-Z2 facilitated an early understanding of the development and debugging tools that are commonly employed across this product line.

A minimal Rust project was implemented to conduct these preliminary tests. The program executed a series of basic arithmetic operations designed to verify the runtime behavior of the compiler and to confirm correct execution flow on the target hardware. After configuring the Rust compiler toolchain, the project was cross-compiled to produce an ELF executable suitable for embedded platforms. The resulting binary was subsequently transferred to the development board using the Xilinx Software Command-Line Tools (XSCT), which are distributed as part of the Vitis suite. Program execution was then validated through GDB by launching the code in debug mode and examining the state of variables during execution, thereby confirming the correct functioning of both compilation and debugging processes.

4.1.2 C Interoperability

Following this initial validation, the `bindgen` crate was utilized to generate Rust bindings for a C library, thus enabling seamless interaction with external functions and data structures implemented in C. This process required compiling the C source code separately as a static library using a designated version of the GCC compiler configured for the ARM bare-metal environment. The generated bindings allowed Rust programs to directly reference the functions, constants, and data types defined in the C library, which facilitated the integration of existing C components within the Rust-based workflow. A similar approach was later used when integrating the SDK with the kernel. In that case, however, the compiler included in the SDK is used to ensure compatibility with the provided toolchain. When the kernel is compiled together with user-level C code, the process is reversed: the operating system is first built as a library and then linked with the C application.

4.1.3 Development Environment

At this stage of development, the target hardware board was not yet available, so software emulation was identified as one of the possible approaches to be explored in order to support early testing and functional validation. Version 9 of QEMU was utilized to emulate the MPS3-AN536 board featuring a dual-core Cortex-R52 CPU. The Rust toolchain for the ARMv8-R architecture required the use of the nightly compiler release, together with specific build configurations to successfully compile the core library for this target. To verify the correctness of the setup, the UART Driver project developed by Ferrous Systems was initially employed as a reference. In the QEMU environment, the UART peripheral is automatically redirected to the standard output, which allowed debug messages to be displayed directly through the terminal. Portions of this reference project were later integrated within the kernel implementation to supply UART functionality in QEMU, where no dedicated SDK was available for the MPS3-AN536 platform.

At this point, the decision was to be made between QEMU and the PYNQ-ZU board featuring the Cortex-R5. QEMU was ultimately selected, primarily because it provided the same CPU architecture as the target board and offered greater flexibility within the development workflow. The emulator enabled quick iteration by allowing rapid testing and deployment of new builds with minimal setup overhead. This capability significantly accelerated the development cycle, as programs could be executed and debugged almost instantly within a consistent software environment. The PYNQ-ZU platform, on the other hand, presented additional layers of complexity due to its more sophisticated hardware architecture, which incorporates an FPGA and requires dedicated configuration and tool support.

The QEMU-based development environment was progressively refined with scripts and supporting tools to streamline the build and execution workflow. Initially, launching QEMU and GDB required lengthy terminal commands specifying the board and configuration parameters. To simplify this process, Windows batch files were first employed, later replaced by Visual Studio Code task definitions that offered greater flexibility and maintainability. GDB integration within the IDE, along with a UART visualizer for both cores based on the MultiTail utility, enabled real-time monitoring of system behavior and output, improving debugging efficiency and overall development productivity. Lastly, the integration of Rust crates that abstract low-level architectural features, such as `cortex-ar` and `arm-gic`, together with the definition of the project requirements provided the groundwork to begin the development phase.

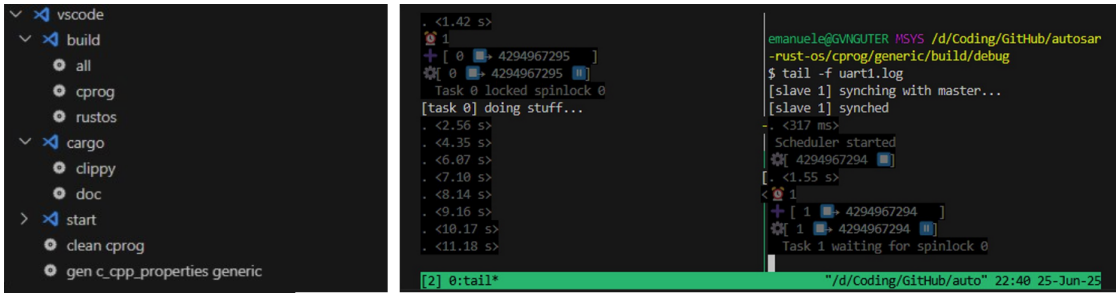

Figure 4.1: Buttons to launch build tasks (on the left) and the UART visualizer showing output from two cores (on the right) in Visual Studio Code.

4.2 Functional Requirements

The system requirements were defined based on the essential properties of an RTOS and the characteristics of the intended application domain. Compatibility with C was identified as a critical condition to enable potential adoption in industrial environments, where legacy systems and existing development workflows rely heavily on C-based toolchains. The architecture of the target board, designed for real-time responsiveness, also played an important role in shaping the design process. It influenced optimization strategies, especially in areas related to timing precision and resource efficiency.

The requirements applied during the development of the system are summarized below.

- **Real-time Performance:** The system is lightweight and performance-oriented, with emphasis on the efficiency of core mechanisms such as context switching and interrupt handling. These elements are optimized to minimize delays and ensure deterministic behavior under varying workloads.

- **Rust and C Interoperability:** The kernel is implemented in Rust to increase safety and reliability while maintaining full compatibility with user code and the SDK, both developed in C. This interoperability allows developers to integrate existing C modules without major modifications, facilitating a smoother migration path.

- **AUTOSAR OS Features:** The fundamental features defined by AUTOSAR OS are implemented and made available to user code through a C-language API of SVC functions. This approach allows a familiar interaction model for developers used to AUTOSAR environments while maintaining the safety guarantees offered by Rust in the kernel.

- **SDK and Drivers:** The SDK for the target board is fully integrated with both the user applications and the kernel. The kernel directly manages essential hardware modules such as GIC, MPU, WDG, and UART, ensuring consistent control and efficient communication between software layers.
- **Board Optimizations:** The system is developed with attention to the hardware characteristics of the target board, including its memory layout and peripheral interfaces. This allows specific optimizations that improve performance and reduce overhead where practical.

4.3 System Architecture

The architecture of the OS follows a simplified AUTOSAR-like design that maintains a clear separation between kernel services, user-level code, and other modules. At the core of the system, the kernel provides two main functions: task scheduling and interrupt management.

The scheduler is responsible for the execution of tasks, which are defined as C functions combined with their configuration parameters. Each task is assigned a priority that determines its position within the scheduling policy. The policy is based on fixed priority levels, ensuring that higher-priority tasks can preempt lower-priority ones whenever required to meet real-time constraints. This preemptive approach guarantees that critical operations receive immediate CPU attention, reducing latency and improving responsiveness under load.

Another essential component of the kernel is the interrupt manager, which configures and controls the GIC module of the CPU. This module enables flexible prioritization and routing of interrupts across multiple CPU cores, supporting scalable real-time processing. The interrupt manager handles both kernel and AUTOSAR interrupts. Kernel interrupts correspond to system-level events such as the system tick and communication between cores, while AUTOSAR interrupts belong to the application domain and are defined in a similar way to tasks, with corresponding configuration and behavior.

Interaction between the application and the kernel is achieved through a dedicated API exposed as a C header file. This API provides essential services such as task activation, termination, and synchronization. By following a familiar C-based interface, the system allows developers to build and integrate user applications without needing to understand the internal details of the kernel implementation. Access to vendor-provided drivers for the target board is available through the SDK, which connects user applications with hardware resources in a consistent way.

System configuration takes place at compile time through a custom XML file. This configuration file is parsed automatically to generate both kernel code and

the data structures required by the application. The compile-time configuration is essential because dynamic memory management is not implemented, which means all data must be statically allocated before runtime. This approach also improves efficiency, as it removes the need for runtime allocation and reduces memory fragmentation. The resulting data structures, referred to as AUTOSAR objects, represent the configured tasks, interrupts, and other resources. During runtime, the kernel uses these objects to store state information, schedule activities, and manage system execution according to the defined configuration.

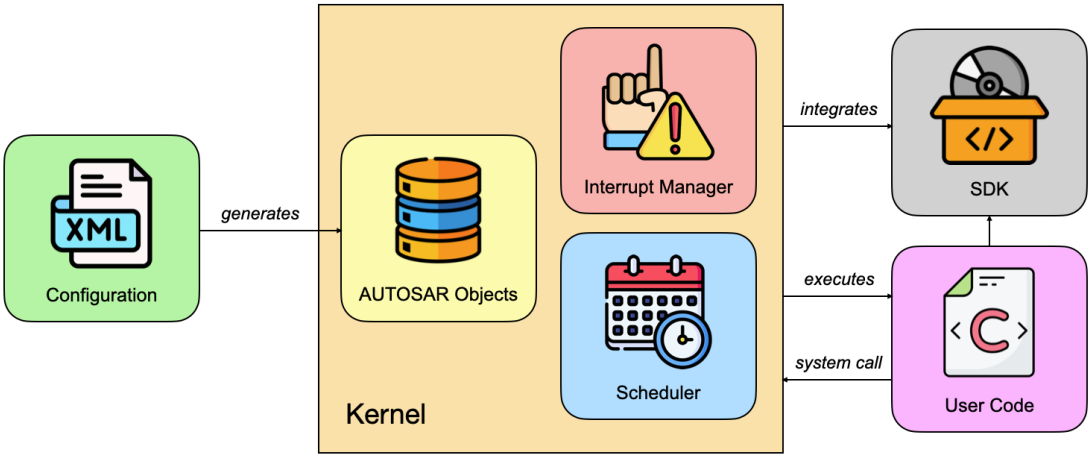

Figure 4.2: Representation of the system architecture

4.4 API Functions

Selected kernel features are made accessible to the application through a dedicated API, which provides the set of functions defined in the specification. These functions serve as the main interface between user applications and the kernel, allowing user-level code to request kernel services in a controlled and predictable manner. The implementation of these functions relies on supervisor calls, which enable a secure and well-defined transition between user and privileged execution modes. Each call is processed by a dedicated SVC handler within the kernel. The handler interprets the service request and dispatches it to the corresponding internal routine. During the execution of a supervisor call, interrupts are temporarily disabled to prevent interference from concurrent events. This mechanism ensures that kernel operations complete atomically and that the system remains in a consistent state after each call. The key functions associated with each kernel feature are presented in the following sections.

Most of the functions return a value of type `StatusType`, which indicates the outcome of the requested operation. This return type provides a range of specific error codes that describe different failure conditions, such as invalid parameters, access to unavailable resources, or function calls made in an incorrect system state. For brevity, the return type is not repeated in the individual function descriptions that follow.

Some API functions operate across CPU cores. By default, these functions are executed synchronously, meaning that control is returned to the caller only after the requested operation has been completed on the target core. In specific cases, asynchronous variants of these functions are also available. These allow the application to continue execution while the requested operation proceeds in the background, which can improve responsiveness and parallel efficiency under certain workloads.

4.5 Task Scheduling

The scheduling process is organized around the concept of a schedule table, which defines when individual tasks are activated. Each schedule table specifies precise activation times that determine the temporal behavior of the system. Durations within the table are defined relative to updates received through the system tick, commonly referred to as ticks. Each tick represents a fixed unit of system time, and the table uses these increments to define expiry points, specific offsets from the start of the table where one or more tasks are activated. This structure provides deterministic task activation and makes system timing predictable.

The specification allows multiple schedule tables to coexist, enabling them to be executed in sequence or under defined conditions. However, in practice, the most common use case involves a single schedule table that repeats continuously. This table automatically restarts once it reaches the end of its cycle, providing a periodic and stable scheduling framework. The project adopted this simplified but representative approach to streamline the development of the scheduler while maintaining the core functionality needed for real-time applications.

Each task is associated with a priority value that is defined statically in the system configuration. This value determines the relative importance of the task in the scheduling process. Although priorities remain fixed under normal conditions, they can be temporarily modified through synchronization protocols described in the following sections. The priority has a direct influence on the order of task execution and on how the system reacts to new activations. When a task with higher priority becomes ready to execute, it immediately preempts any task with lower priority that is currently running. As a result, at any given moment, the running task usually corresponds to the active task with the highest priority. This

preemptive scheduling model ensures responsiveness and guarantees that operations with strict timing requirements are executed without delay.

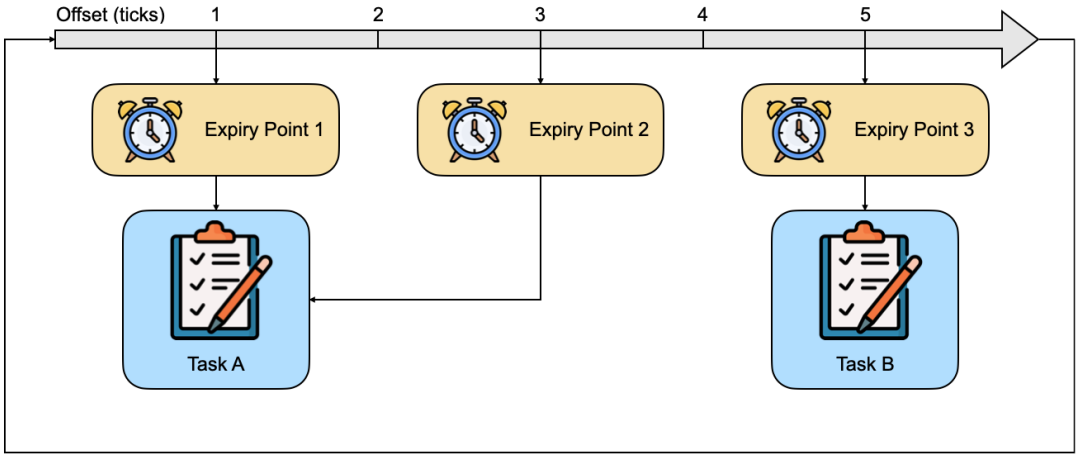

Figure 4.3: Example of a schedule table.

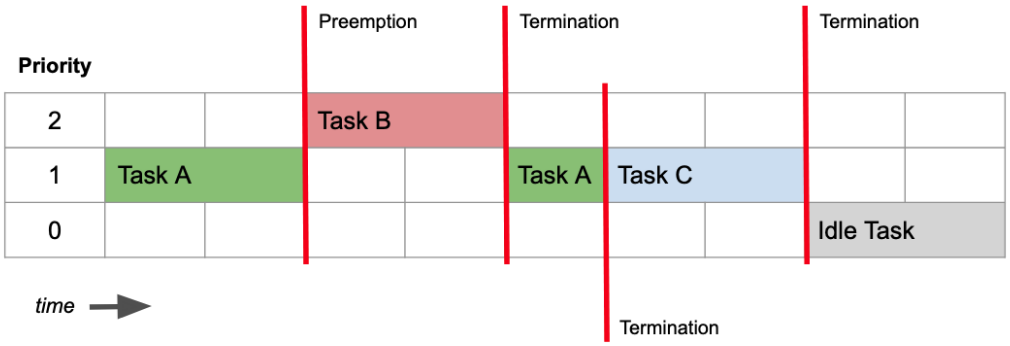

Figure 4.4: Example of priority-based scheduling. The running task is represented.

4.5.1 Task State

Tasks can be in one of the following states:

- **Suspended:** inactive or terminated, can be activated at a later time.
- **Ready:** already activated and ready to be executed.

- **Running:** currently in execution, can terminate or be preempted by a task of higher priority.

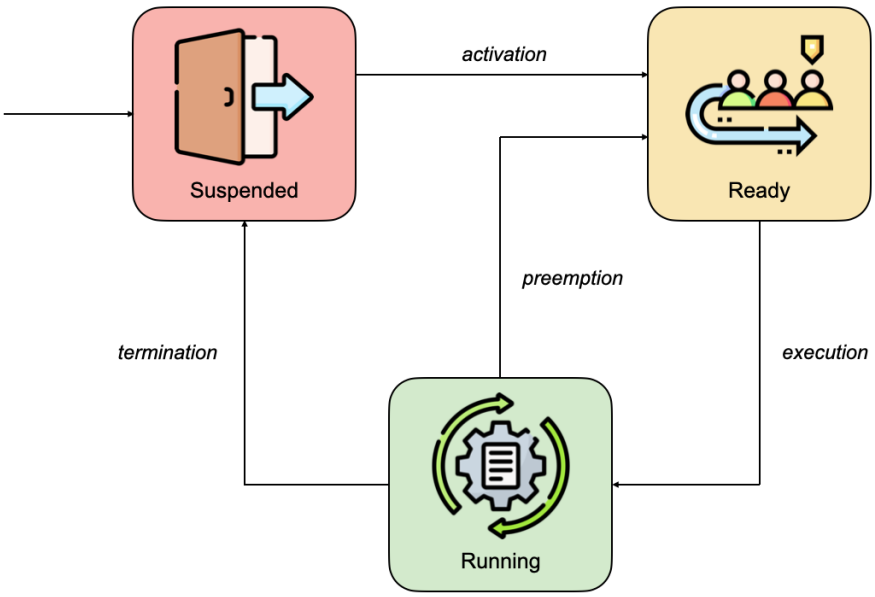

Figure 4.5: States of a basic task.

Before the schedule table loop begins, a specific group of tasks referred to as startup tasks can be executed. These tasks carry out initialization routines that prepare the application for normal operation by setting up essential components and verifying the system state. Once the system is ready, the scheduler proceeds with the regular execution of the schedule table. When no other task is in the ready state, the scheduler executes an idle task that remains permanently available and never terminates. The idle task can be employed for basic power management or for background operations that do not require strict timing.

Tasks can also be activated through the API by other tasks or by interrupt routines. An activation is accepted only when the target task is currently in the suspended state, which ensures that each activation follows a defined and consistent lifecycle. This rule prevents conflicts and keeps the kernel state predictable during task management.

The OS typically runs a limited set of tasks repeatedly with the goal of ensuring deterministic behavior. The main source of variability is the handling of interrupts generated from external sources, which may delay the execution of tasks. Applications introduce large time margins to avoid overlapping activations of the

same task. This behavior, known as overrun, is reported by the OS when a new activation occurs while the task is in the wrong state.

`ActivateTask(TaskID task)`
Transfers the specified task, which can be defined on any core, from the *Suspended* state into the *Ready* state. The asynchronous version `ActivateTaskAsyn is also available`.
`TerminateTask()`
Terminates the current task.
`ChainTask(TaskID task)`
Terminates the current task and activates the specified task with a single operation.

Table 4.1: API functions related to the scheduling of tasks.

4.5.2 Implementation Details

In order to support context switching between tasks, tasks are associated with a Task Control Block (TCB) which defines the required fields to store the values of all CPU registers, including the VFP coprocessor registers used for floating point calculations. During a context switch, the scheduler saves and restores the content of these registers to preserve task state.

The scheduler maintains a record of ready tasks through a dedicated queue structure. In the initial implementation, this queue was built as a linked list in which tasks were sorted by their priority. Although functional, this approach produced higher management costs when the number of tasks increased. In order to improve performance in applications that handle many concurrent tasks, the linked list was later replaced with a structure composed of individual queues, one for each priority level. This design allows faster insertion and retrieval operations, which directly benefits response time in the scheduling process.

Inside the ready queue, the system distinguishes between tasks that were newly activated and those that were preempted. This distinction is represented by two internal ready states. It is important because different preparation steps must be carried out before a task begins or resumes execution, depending on its state:

- **Newly activated tasks:** the stack pointer and program counter are initialized in the task context.
- **Preempted tasks:** the task context is restored from the Task Control Block (TCB).

The allocation of stack memory for tasks follows the assumption that execution occurs in a defined order determined by priority. Because preempted tasks have lower priority and resume only after the running task finishes, their stacks remain inactive during this period. As a result, the stack of a newly activated task can be allocated above the stacks of other preempted tasks without causing overlap or corruption. This approach makes efficient use of limited memory resources while preserving safe task isolation.

The MPU is reconfigured dynamically each time the running task changes. The reconfiguration is based on the stack region assigned to the task to ensure separation between memory spaces. The MPU enforces isolation according to the maximum stack size value defined for each task in the configuration. Since stack-level protection is not required in all cases, this mechanism can be optionally disabled to simplify execution and improve performance in systems where strict isolation is not necessary.

4.6 Multi-core Support

Since both the target board and the MPS3 in QEMU feature multiple CPU cores, the system was designed to support multi-core functionality from the early stages of the project.

QEMU supports Symmetric Multi-Processor (SMP) mode, in which the two cores of the MPS3 execute instructions from the same executable while sharing a common RAM space. In this execution model, conditional branches in the program distinguish the behavior of each core, allowing them to perform separate operations. One of the first steps in initialization is to allocate distinct stack regions for each core to avoid memory overlap and preserve isolation. On ARM architectures, the Multiprocessor Affinity Register (MPIDR) is used to identify the core currently executing instructions. This register contains several fields that vary depending on the implementation, but it is generally used to enumerate individual CPU cores and clusters.

4.6.1 Synchronization Mechanisms

Shared memory regions provide the basis for synchronization mechanisms between cores. Within the kernel, two main mechanisms are used: barriers and spinlocks.

- **Barriers** are points in the program where each core must wait until all other cores reach the same location before continuing execution. This ensures that all cores progress in a coordinated manner.
- **Spinlocks** control access to shared resources by allowing only one core to

use the resource at a time. Cores that attempt to access a locked resource repeatedly check the lock and wait until it is released.

Both mechanisms are implemented using Rust abstractions for atomic types. In the case of barriers, a counter keeps track of the number of cores that have reached the synchronization point, allowing execution to continue only when all cores have arrived. Spinlocks rely on a variable that records which core currently holds the lock, preventing simultaneous access and ensuring safe coordination between cores. This approach provides deterministic behavior and minimizes the risk of race conditions in multi-core execution.

4.6.2 Inter-core Communication

A mechanism for Inter-core Communication (ICC), referred to as the core mailbox, was also implemented. Each core is associated with a mailbox, which relies on a spinlock to protect a shared memory region where data can be exchanged. When two cores need to communicate, the sender acquires the spinlock and writes data on the mailbox of the receiver, then sends a SGI to notify the receiver. When the interrupt is processed, the receiver reads the data and releases the lock. Mailboxes also support a response mode, in which the receiver sends data back to the sender. In this mode, a barrier causes the sender to wait until the receiver processes the message and writes the response in its own mailbox. The sender then reads the response and releases the lock. This mechanism is used to implement API functions that operate across cores, such as the `ActivateTask` function.

SGIs are also used to propagate faults or panic events from one core to all other cores. This ensures that critical errors are communicated system wide and allows coordinated responses to maintain safety and consistency across cores.

4.6.3 Multi-core Scheduling

In AUTOSAR, the first core is conventionally designated as the master core, and each core runs its own instance of the operating system with a dedicated scheduler. To maintain synchronization between schedulers, the Generic Timer of the master core is configured to generate the system tick, which is forwarded to all other cores through SGIs. During system startup, all cores synchronize using a barrier to ensure that all startup tasks are executed before the schedule tables begin operation.

In the system configuration, tasks and expiry points are defined as part of a single global schedule table. In practice, however, they are distributed across the individual schedulers running on different cores. The parser generates identifiers that embed the core number in the upper bits, which are set through masking and bitwise operations. This mechanism allows the system to identify the core to which each task is assigned. The same method is applied to other AUTOSAR objects.

4.7 Interrupt Management

The interrupt manager is the kernel component responsible for the configuration and management of the GIC module. At the initialization it configures all interrupts used by the kernel, including the system tick and the SGIs used for cross-core communications. Application-related interrupts, represented by AUTOSAR Interrupt Service Routines (ISRs), are also configured and managed by the interrupt manager to ensure correct execution within the operating system.

4.7.1 Interrupt Service Routines (ISRs)

Interrupt Service Routines are functions defined by the application that are executed inside interrupt handlers under the control of the kernel. The AUTOSAR specification defines two categories of ISR:

- **Category 1 (ISR1):** functions associated with high priority interrupts that do not have access to SVCs.
- **Category 2 (ISR2):** can interact with the kernel through SVCs, potentially resulting in task activations or preemption of the running task.

The operating system keeps track of the type of operation currently being executed using a field known as the call level. This field indicates whether a task, ISR1, ISR2, or another kernel operation is in progress. The call level is used to enforce access restrictions in supervisor calls and to guide the execution of specific operations in their implementation.

In practice, ISRs are implemented as system wrappers that invoke the configured application functions within the interrupt handlers. For ISR1, the wrapper only preserves the task context and sets the call level, ensuring minimal overhead and predictable execution. For ISR2, the wrapper includes an additional check to determine whether the running task should be preempted, and the scheduler is invoked if necessary to manage task switching.

ISRs can be interrupted by other interrupts, allowing nested execution. Because Category 1 ISRs are generally assigned higher priority values, they complete before any ISR2 when nesting occurs. For nested ISR2, the scheduler is not invoked immediately upon termination. Instead, potential preemption of the running task is performed only after the outermost interrupt handler finishes, ensuring correct task sequencing and avoiding unnecessary context switches.

The interrupt manager also provides mechanisms that are used to implement the API functions related to interrupts.

`DisableInterruptSource(ISRID isr)`
Disables the interrupt associated with the ISR. `EnableInterruptSource` is used to enable the interrupt.
`DisableAllInterrupts()`
Disables all interrupts, including AUTOSAR and kernel interrupts. Used with `EnableAllInterrupts` to re-enable the interrupts at a later time.
`SuspendAllInterrupts()`
Variant of `DisableAllInterrupts` that supports multiple sequential calls. Used with `ResumeAllInterrupts`. `SuspendOSInterrupts` and the corresponding *Resume* function are used to manage kernel interrupts separately.

Table 4.2: API functions related to interrupts.

4.7.2 Implementation Details

The GIC module provides 256 priority levels for interrupts and supports nested execution. Special care was required when implementing nested interrupts, as the link register could otherwise be overwritten, causing incorrect return addresses. To prevent this, interrupt routines are executed in the privileged system mode, which preserves the original register values through the use of banked registers. This approach ensures reliable nested execution while maintaining consistent kernel state.

Because the GIC has separate interfaces for each core, initialization is performed individually for every core. ISRs are configured during system initialization and enabled after the startup phase, when the schedule tables begin execution. In the system configuration, each ISR is defined with the data necessary to configure the interrupt in the GIC, including the interrupt number (INTID), an enable flag, and the priority. Category 1 ISRs are enforced to have higher priority values to simplify the handling of preemption in nested scenarios. Category 2 ISRs allow nested execution by explicitly re-enabling interrupts within their wrappers.

4.8 Other Features

This section presents features that are part of the specification and extend the functionality of the system, providing synchronization and execution mechanisms.

4.8.1 Events

Events provide a mechanism for synchronization and signaling between tasks. They are associated with extended tasks, a variant of tasks that supports an additional *Waiting* state. In this state, the execution of the task is suspended until one or more events are set by other tasks or interrupt routines. This allows tasks to coordinate their execution and respond to specific conditions or signals without continuously polling for changes. Because multiple events can be referenced together, bit masks are used to represent groups of events efficiently. A task can wait for a combination of events, resuming execution only when all the required events in its mask are set. This allows complex synchronization patterns between tasks, such as waiting for multiple conditions to be met simultaneously.

`WaitEvent(EventMask events)`
Puts the current task in the *Waiting* state until the events are set.
`SetEvent(TaskID task, EventMask events)`
Sets the events for the specified task, which can be defined on any core. The asynchronous version `SetEventAsyn` is also available.
`ClearEvent(EventMask events)`
Clears the events for the current task.

Table 4.3: API functions related to events.

In the implementation, each extended task is associated with two mask fields: one records the events required for the task to resume execution, while the other is updated when events are set. This distinction is necessary because events remain set until they are explicitly cleared, and a single field would not be sufficient to track all changes reliably.

Stack memory allocation for extended tasks requires a different approach compared with standard tasks, because they do not always follow the traditional priority-based execution scheme. In this project, fixed memory sections are pre-allocated during OS initialization, ensuring that each extended task has sufficient stack space regardless of its execution order or waiting behavior.

In typical applications, events are used to implement producer-consumer patterns, signal the completion of hardware operations, or synchronize tasks that depend on external inputs. For example, one task may wait for a sensor reading to complete, while another task sets the corresponding event once the data is available. This approach allows the application to remain responsive and deterministic, avoiding unnecessary CPU usage while tasks wait for relevant conditions.

4.8.2 Resources

Resources provide a mechanism for synchronization, commonly used to protect critical sections from interference by other tasks or ISRs running on the same core. They ensure that shared data or hardware is accessed safely and consistently, preventing race conditions and inconsistent state.

When a resource is acquired, the runtime priority of the owning task or ISR is raised according to the Priority Ceiling Protocol (PCP). The new value, known as the priority ceiling of the resource, is defined as the maximum priority among all tasks and ISRs that may access the resource. Raising the priority to the ceiling prevents other tasks or ISRs with conflicting access from executing within the critical section, since their priorities are lower or equal. Once the resource is released, the owner's priority is restored to its previous value, which may result in preemption if higher priority tasks become ready.

Multiple resources may be acquired in succession. In these cases, the highest priority ceiling among the acquired resources is applied, and resources must be released in last-in, first-out (LIFO) order. All resources must be released before a task or ISR can terminate or, for an extended task, wait for events. In the implementation, both tasks and ISRs track the resources they acquire using a queue, which allows the system to verify that all resources are released when required. This disciplined approach ensures that resources remain available to other tasks and ISRs, preventing deadlock and maintaining predictable system behavior.

`GetResource(ResourceID resource)`
The current task or ISR acquires the resource.
`ReleaseResource(ResourceID resource)`
The current task or ISR releases the resource, which must be its most recently acquired resource.

Table 4.4: API functions related to resources.

In typical applications, resources are used to protect access to shared hardware peripherals, communication buffers, or data structures that are modified by multiple tasks or interrupt routines. For example, a task updating a shared sensor buffer may acquire a resource to prevent an ISR from reading inconsistent data at the same time. Similarly, multiple tasks writing to a status variable can use a resource to serialize access, ensuring correct ordering and avoiding corruption.

ISRs and tasks may access the same resources. Normally, ISRs can interrupt tasks regardless of their priority levels. However, when a resource is shared between tasks and ISRs, interrupts that would cause conflicting access must be prevented while a task already owns the resource. To apply the priority ceiling protocol

to ISRs, they are assigned priority values that are comparable to task priorities, allowing the system to compute a consistent ceiling value for each resource. In this project, interrupts are managed by the GIC, which uses a scheme where higher priority interrupts have numerically lower values. To reconcile this difference, priority mapping functions translate GIC interrupt priorities into values compatible with the ceiling protocol. When a resource is acquired, interrupts with lower priority than the ceiling are temporarily masked using the priority mask field in the GIC, ensuring that access to the resource remains exclusive and consistent.

Tasks and ISRs statically define the list of resources they may access in the system configuration. This allows the ceiling priority for each resource to be computed at compile time when the configuration file is parsed.

4.8.3 Spinlocks

Spinlocks provide a synchronization mechanism designed specifically for use across multiple CPU cores. They can be acquired and released similarly to resources, but the underlying mechanism is different. Because a spinlock may not be immediately available, a task or ISR attempting to acquire it may enter a busy-wait loop until the lock is free. In this project, AUTOSAR spinlocks are implemented using the spinlock abstraction provided by the kernel, which relies on Rust atomic types for safe and efficient inter-core synchronization.

`GetSpinlock(SpinlockID spinlock)`
The current task or ISR acquires the spinlock. This function returns after the spinlock becomes available.
`TryToGetSpinlock(SpinlockID spinlock, SpinlockResult* result)`
The current task or ISR acquires the spinlock if available. This function always returns immediately.
`ReleaseSpinlock(SpinlockID spinlock)`
The current task or ISR releases the spinlock, which must be its most recently acquired spinlock.

Table 4.5: API functions related to spinlocks.

Multiple spinlocks may be acquired in succession. To prevent deadlock scenarios, a fixed lock acquisition order is defined in the system configuration and enforced by the kernel. Once acquired, spinlocks must be released in LIFO order, and all spinlocks must be released before waiting or termination.

Spinlocks are intended solely for inter-core synchronization, and attempting to acquire a spinlock already held by the current core results in an error. They

are particularly useful for protecting critical sections that involve shared data or hardware accessed by multiple cores, where traditional resource locks cannot guarantee exclusivity. However, because spinlocks rely on busy-waiting, their use should be restricted to short critical sections. Prolonged busy-waiting could waste CPU cycles and negatively affect real-time performance.

In typical applications, spinlocks are used to guard shared communication buffers, control registers, or other data structures that are modified by tasks or ISRs on different cores.

4.8.4 Alarms

Alarms provide a mechanism for executing actions at specific times, based on counters. Counters are numerical values that increase either through a hardware timer or by software increments. Each alarm is associated with a counter and can be configured to expire when the counter reaches a predetermined value. When an alarm expires, the operating system executes one of the following actions:

- **Task Activation** of a task on the same core.
- **Event Setting** on a task on the same core.
- **Callback** of a function provided by the application.

`SetRelAlarm(AlarmID alarm, Ticks increment, Ticks cycle)`
Sets the alarm to expire after `increment` ticks on its first activation and subsequently every `cycle` ticks. If `cycle` is zero, the alarm expires only once.
`CancelAlarm(AlarmID alarm)`
Cancels the alarm so that it will no longer be updated.
`GetAlarm(AlarmID alarm, Tick* ticks)`
Stores in `ticks` the number of ticks remaining until the alarm expires.

Table 4.6: API functions related to alarms.

In this project, the implementation of alarms was simplified and restricted to those associated with the system tick. The scheduler is responsible for tracking all active alarms. During each tick, after updating the schedule table, the scheduler updates the state of alarms to ensure timely execution of their associated actions.

4.9 Build Process

The build process of the system requires multiple steps and operations where files are processed and modules written both in Rust and C are linked. This process is supported by build tasks in the IDE, makefiles, and Rust build scripts, which together ensure a reproducible and automated compilation flow.

Build tasks provide a user-friendly entry point for developers, prompting the selection of the application project, the target board, and the build mode, either debug or release. Internally, these tasks invoke terminal commands that start the Rust build process through Cargo and execute additional scripts. The Rust build script runs on the host system and uses crates that depend on system libraries. The following operations are performed before the kernel is compiled:

1. **Assembly Build:** Assembly files, including the boot code and routines for saving and restoring the CPU context, are compiled first.

2. **SDK Binding:** Selected modules of the SDK are specified in a JSON file. The file is parsed using the `serde_json` crate and then bindings for these modules are generated through a build module based on the `bindgen` crate.

3. **Configuration Parsing:** The XML file containing the system configuration is parsed through a build module based on the `roxmltree` crate to generate Rust code representing the AUTOSAR objects required by the application. Additionally, a C header file is created containing symbols used in user code to reference these objects, such as task IDs passed as parameters in supervisor calls.

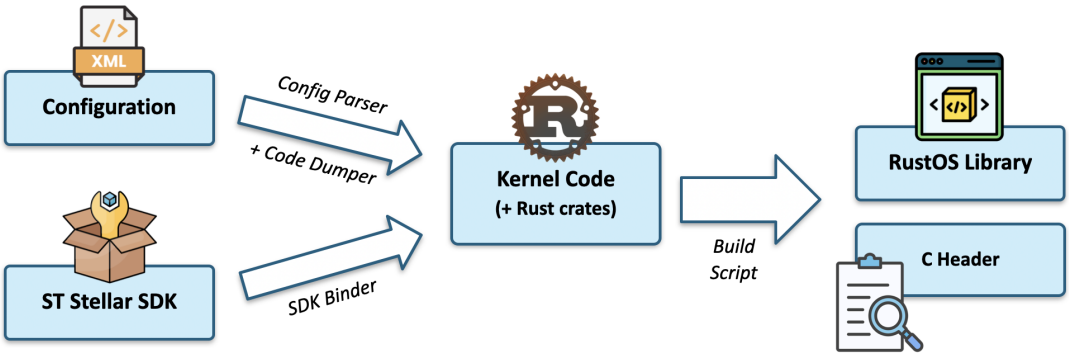

Figure 4.6: Steps to build the kernel as a static library.

After these steps, the kernel is compiled as a static library using `rustc`, the Rust compiler. In a second stage, the application project, written in C, is compiled using either the ARM version of GCC or the compiler provided with the SDK. Finally, the application is linked together with the SDK and the OS library to produce the final ELF executable. This staged approach ensures that dependencies between assembly, Rust, and C modules are correctly resolved and that the resulting binary is fully functional on the target platform.

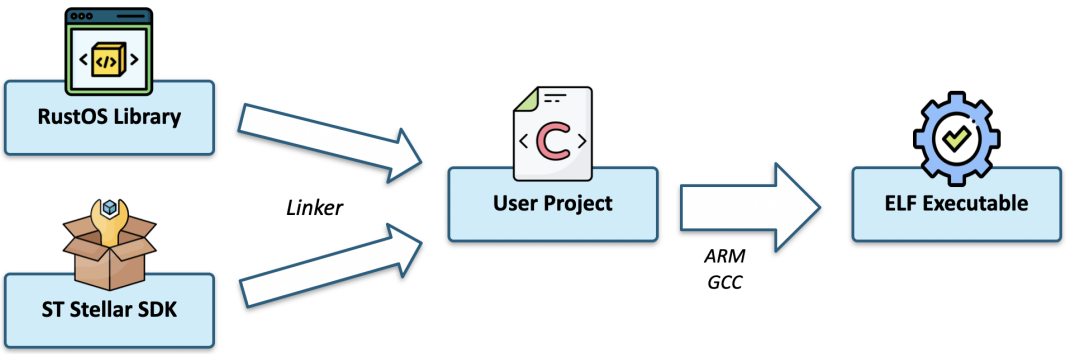

Figure 4.7: Steps to build the final ELF executable.

Chapter 5

Testing and Validation

This chapter is divided in two parts. The first part presents the strategies used to test the system and verify its functionality, while the second part addresses the modifications needed to deploy the system on the target board and presents the measurements used to evaluate performance across different scenarios.

5.1 Testing Procedure

Since the functionality of the system is tied to runtime execution and asynchronous events, unit tests would not be sufficient or practical. The solution adopted for testing employs small dedicated applications aimed at verifying individual features of the system during execution.

The first step for the development of test applications was a modification to the build system that enabled the use of individual application projects, simply referred to as projects, contained in the folder of the system. This made the development of applications more flexible and enabled the creation of separate projects for the various features to test.

A second step was the implementation of a macro for assertions, similar to that of the C standard library. This macro checks a condition and, if it is not verified, performs a system call that halts execution and prints a message relying on the `panic` macro of Rust. This allowed the immediate detection and reporting of errors, even in applications with complex behavior.

Test projects were designed to validate the system by checking that API functions produced the expected results under different conditions and that tasks and routines executed in the correct order according to their configuration and the runtime state. In practice, user code performs mathematical operations on shared variables while using OS mechanisms such as priorities, locks, and events to enforce execution order and protect data integrity. The results are then checked for correctness, which is

achieved only if the operations are executed in the expected order.

The real-time behavior of the developed programs was observed and verified thanks to the assistance of simple mechanisms such as direct feedback from the on-board LEDs and output via the UART interface.

These test projects were primarily developed for QEMU due to its flexibility and ease of use, providing a regression-testing environment in which tests can be executed after system updates to verify functionality and compatibility with existing features. This approach proved instrumental in identifying and correcting programming errors throughout the development process.

5.2 Porting to the Target Hardware

While QEMU provided an accurate emulation of the target CPU architecture, several adjustments were required to deploy the system on the target board and obtain the expected behavior.

5.2.1 Build Tools and SDK Integration

The first step was the integration of the build tools supplied with the SDK. Although the SDK includes its own makefile, it could not be used directly, as certain operations that are typically automated by the vendor-provided IDE had to be implemented explicitly. In practice, additional directives and symbol redefinitions were required to incorporate the necessary modules. For this reason, the makefile provided with the SDK was incorporated into a custom makefile, which integrates these modifications.

The kernel also relies on selected modules from the SDK. Linking the entire SDK would have necessitated generating bindings for all components, which was not practical. To address this, only the required modules are linked. A dedicated build module, referred to as the SDK binder, was developed for this purpose, using the `bindgen` crate.

5.2.2 Runtime Behavior

The runtime behavior of QEMU differs significantly from that of the target board because of the multi-core execution model and its associated memory layout. Unlike the MPS3, where all cores execute instructions from a single executable stored in shared memory, the target board provides separate memory modules for each core, where individual programs are stored to improve performance. A shared memory module is also available, but it offers slower access and is intended exclusively for data that must be explicitly shared between cores. This architectural difference required modifications to both the configuration parser module and the build tools.

Since the build process produces separate executables rather than a single ELF, the data structures generated from the configuration, along with their accessor functions, are distributed so that each core has access only to the objects defined for it. Synchronization variables and data structures associated with shared peripherals are placed in the shared memory section so that they can be accessed by any core.

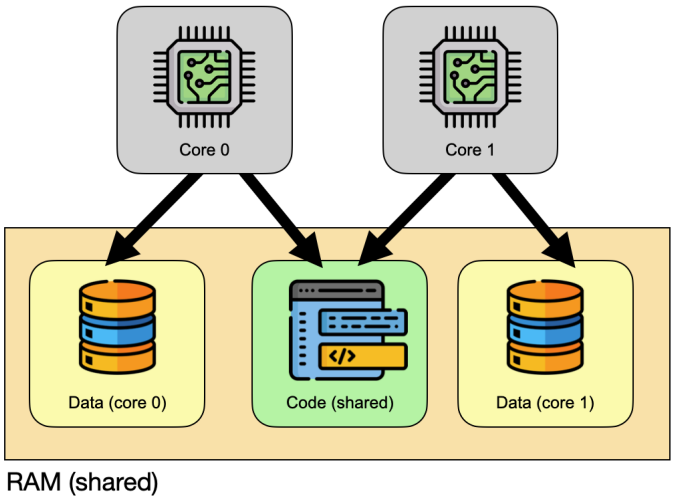

Figure 5.1: Memory layout of the MPS3 board in QEMU.

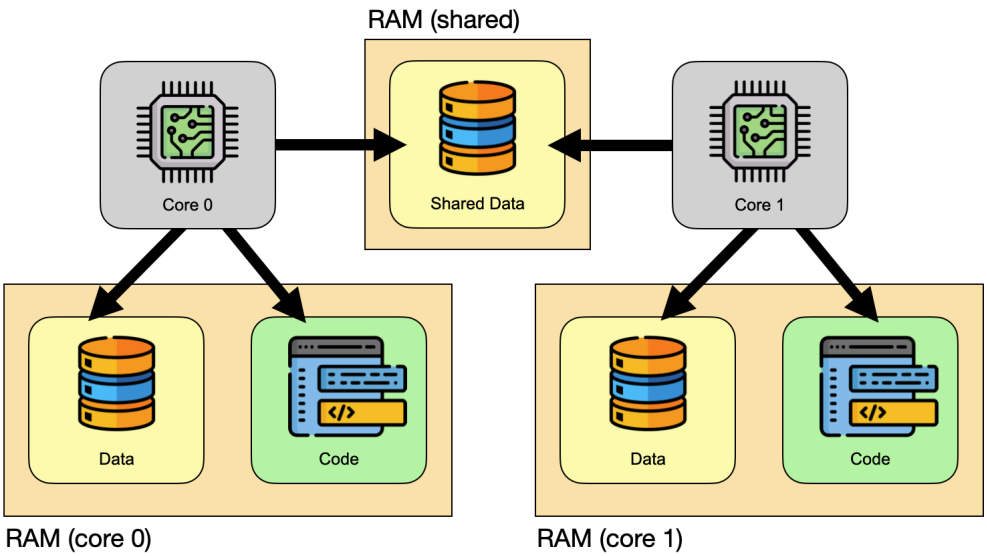

Figure 5.2: Memory layout of the target board.

5.3 Performance Evaluation

The performance evaluation process focused on measuring the duration of core low-level operations such as context switching between tasks and interrupt handling. Although this scope may appear limited, these operations are highly impactful due to their frequency, and their efficient implementation is critical for minimizing delays and achieving acceptable real-time performance.

Measurements were conducted on applications executed on the target board. Execution was monitored using the Lauterbach TRACE32 software, with cycle-accurate timing obtained via the Embedded Trace Macrocell (ETM) trace chart. Observed values were generally consistent. However, values reported refer to the maximum times.

Two applications were employed for the evaluation. In the first, two tasks alternated execution, producing context switches. In the second, a long-running high-priority task was active while other tasks were activated but not immediately executed.

The following values were observed:

Tick Interrupt	1µs
Tick + Task Activation	1.3µs
Tick + Task Activation + Preemption (by new task)	1.8µs
Task Termination + Preemption (by new task)	1.1µs
Task Termination + Preemption (by preempted task)	1.5µs

- **Tick Interrupt:** Duration of the interrupt when no tasks are activated from the schedule table. The operations performed are limited to saving and restoring the context and processing the interrupt.
- **Task Activation:** In this case a lower priority task was activated from an expiry point of the schedule table and inserted in the queue of ready task.
- **Task Termination:** When a task terminates, the scheduler is invoked within the SVC without waiting for the tick and a task from the ready queue is scheduled (or the idle task if there no other tasks).
- **Preemption (by new task):** When the running task is suspended to start a new task, only few registers such as the program counter and the stack pointer are set in the context so that the execution can begin from the entry point of the task.

- **Preemption (by preempted task):** When the running task is suspended to resume another task that was previously preempted, the whole context of the task is restored from its TCB.

In practical applications, typical values for the resolution of the system, which are used to configure the tick interval, are in the range of milliseconds. In comparison, the observed durations for the execution of system operations are negligible, leaving ample margin for task execution and ensuring that delays in handling external events remain acceptable.

Chapter 6

Conclusion and Future Works

This chapter concludes the thesis by providing an overview of the work conducted and the results achieved. It summarizes the key design decisions and the main contributions of the project, while also highlighting potential improvements, optimizations, and future developments that could further enhance or extend the system, with the ultimate goal of industrial adoption.

6.1 Conclusion

This project aimed to explore the use of Rust in embedded systems through the development of an AUTOSAR-based RTOS for the ARM Cortex-R52. Rust was selected for its memory-safety and reliability features, while interoperability with C, the de facto industry standard, was implemented from the beginning to support industrial adoption. Development and testing were largely performed through software emulation of the target architecture using QEMU.

The development process began with the implementation of fundamental features of the system, including deterministic, priority-based task scheduling and interrupt management. Subsequent work introduced multi-core support and MPU-based memory protection, which played a central role in shaping the architecture of the system. In addition, synchronization and execution mechanisms defined in the AUTOSAR OS specification were implemented. The required functions were made available to the user application through an API, implemented via SVCs and provided in a C header file.

The system was tested using dedicated applications designed to verify the runtime behavior of its various features. Lastly, the project was deployed to the target board to collect performance measurements. This process required

code adaptations due to differences in multi-core execution and memory layout compared with the QEMU environment. The observed results met the typical timing requirements of real-time applications.

In conclusion, the project demonstrates the practical feasibility of using Rust for real-time, safety-critical embedded systems. The resulting implementation provides a self-contained software platform that can be extended and refined to support future research and industrial adoption.

The combination of features and tools integrated within the Rust ecosystem, effective software emulation through QEMU, and the automation capabilities provided by the Visual Studio Code IDE demonstrate that open-source solutions can serve as highly reliable and efficient alternatives in a field traditionally dominated by proprietary technologies.

Although the amount of available resources, particularly in specialized domains, is still more limited than that of C, the Rust community has proven to be highly active and responsive. Observing the rapid development of several related open-source projects, some of which were integrated into this work during its execution, and even contributing to their evolution, has been highly encouraging. These experiences reinforce the belief that Rust is rapidly maturing and is likely to become a prominent choice for embedded and safety-critical development in the near future.

6.2 Future Works

While the developed system represents a fully functional prototype and a strong foundation, several limitations remain that offer opportunities for improvement. Future work could focus on achieving better compliance with the specification and improving the development experience. The subsequent points outline potential paths for further improvement:

- **AUTOSAR OS Features:** Some of the implemented features are only partially compliant with the specification, as simplified versions were developed. This is the case for schedule tables and alarms. Other features, such as hooks, were not implemented However, the system architecture was designed to facilitate their future integration.

- **Platform Integration:** Although the system currently operates as a standalone solution, it has not been tested as part of the full AUTOSAR platform. Integration with industry-standard development tools, such as the Vector DaVinci suite, would ensure compatibility with typical workflows. This would require adopting the standardized configuration format and adapting the existing parser accordingly.

- **AUTOSAR Scalability Class 3:** This specification, typically required for high-end ECUs in industrial contexts, defines advanced features such as timing and memory protection mechanisms. Since the system already supports some of the features, such as memory protection, are already available, the implementation of the missing features would be feasible and it would support industrial adoption.
- **Rust Application:** The system is designed to work with user code written in C. Allowing the use of Rust for the application code alongside C would support broader adoption of Rust in embedded development. This would require modifications to the build system.

Bibliography

[1] André Pinho, Luis Couto, and José Oliveira. «Towards Rust for Critical Systems». In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036 (cit. on p. 2).

[2] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Kumar Machiry. «Rust for Embedded Systems: Current State and Open Challenges». In: *Proceedings of the 2023 ACM Conference on Computer and Communications Security (CCS 2023)*. New York, NY, USA: ACM, 2024, pp. 2296–2310. DOI: 10.1145/3658644.3690275 (cit. on pp. 2, 19).

[3] Lukas Seidel and Julian Beier. «Bringing Rust to Safety-Critical Systems in Space». In: *2024 Security for Space Systems (3S)*. 2024, pp. 1–8. DOI: 10.23919/3S60530.2024.10592287 (cit. on p. 2).

[4] James Curbo and Gregory Falco. *Alcyone: A Blueprint for Secure Rust Flight Software*. July 2025. DOI: 10.13140/RG.2.2.17022.91202 (cit. on p. 2).

[5] ARM. *ARM GIC Fundamentals*. https://developer.arm.com/documentation/198123/0302/Arm-GIC-fundamentals. Accessed: 2025-10-10 (cit. on p. 6).

[6] ARM. *Cortex-R52 Product Support*. https://developer.arm.com/Processors/Cortex-R52. Accessed: 2025-10-10 (cit. on p. 6).

[7] Thibaut Vandervelden, Ruben De Smet, Diana Deac, Kris Steenhaut, and An Braeken. «Overview of Embedded Rust Operating Systems and Frameworks». In: *Sensors* 24.17 (2024). DOI: 10.3390/s24175818 (cit. on p. 20).

[8] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. «Multiprogramming a 64kB Computer Safely and Efficiently». In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*. New York, NY, USA: ACM, 2017, pp. 234–251. DOI: 10.1145/3132747.3132786 (cit. on p. 21).

[9] I. Culic, A. Vochescu, and A. Radovici. «A Low-Latency Optimization of a Rust-Based Secure Operating System for Embedded Devices». In: *Sensors* 22.22 (Nov. 2022), p. 8700. DOI: 10.3390/s22228700 (cit. on p. 21).

[10] 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)*. 2025, pp. 241–245. DOI: 10.1109/DCOSS-IoT65416.2025.00040 (cit. on p. 21).

[11] Zakaria Madaoui, Henri Lunnikivi, Pawel Dzialo, and Per Lindgren. «Towards modularity of the Rust RTIC real-time scheduling framework». In: *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*. 2024, pp. 1–7. DOI: 10.1109/NorCAS64408.2024.10752441 (cit. on p. 21).

[12] Thor Myklebust, Christian Askeland, and Espen Helle. «Enhancing Software Safety Through Programming Languages: A Study of Rust». In: June 2025. URL: https://www.researchgate.net/publication/392736748_Enhancing_Software_Safety_Through_Programming_Languages_A_Study_of_Rust (cit. on p. 22).