Evaluation of Rust usage in space applications by developing BSP and RTOS targeting SAMV71

Executive Summary Report

ROS-N7S-ESR-001 rev. 1.0

N7 SPACE SP. Z O.O.

Prepared by	Date and Signature
Seweryn Ścibior	
Verified by	
Filip Demski	
Approved by	
Michał Mosdorf	

Table of Contents

Change Record

Issue	Date	Change
1.0	2024-01-26	Initial release

1 Introduction

1.1 Purpose, scope, and content

This document provides Executive Summary Report, an extract from project's Final Report, for the 'Evaluation of Rust usage in space applications by developing BSP and RTOS targeting SAMV71' project. It presents all results, findings, and conclusions.

1.2 Project motivation and objectives

The main objective of the activity was to evaluate the usage of Rust programming language in space applications, by providing an RTOS targeting ARM Cortex-M7 SAMV71 microcontroller, a BSP (Board Support Package) and a Demonstration Application.

RTOS is implemented in the form of an executor instead of a classic scheduler. The scope of this project doesn't include preemption. This executor runs tasklets, which are fine-grained units of computation, that execute a processing step in a finite amount of time.

Basic functionality required to create a working system is provided – tasklet priorities, recurring and time-based execution, as well as communication facilities such as queues and events. Additionally, execution time statistics are reported to facilitate scheduling analyses and discovery of real-time related issues.

Creating a real time operating system validates Rust features mentioned in the section above in practice, evaluates Rust viability in space applications and additionally checks compatibility with ECSS software development process.

The main focus of the BSP part of the project was to provide a minimal set of functionalities for peripherals required to create the RTOS and interact with the board as well as example sensors.

In the second part of the activity, a small demonstration application software was developed. This demonstration provided input to a Lessons Learned report, describing the encountered issues, potential problem and improvement areas, usage recommendations and proposed way forward.

2 Terms, definitions, and abbreviated terms

This document acronyms and abbreviations are listed here under.

BSP	Board Support Package
HAL	Hardware Abstraction Layer
N7S	N7 Space
PAC	Peripheral Access Crate
ROS	Rust Operating System
RTOS	Real Time Operating System

3 Work logic

Figure 1 2below shows activity work logic and work packages.

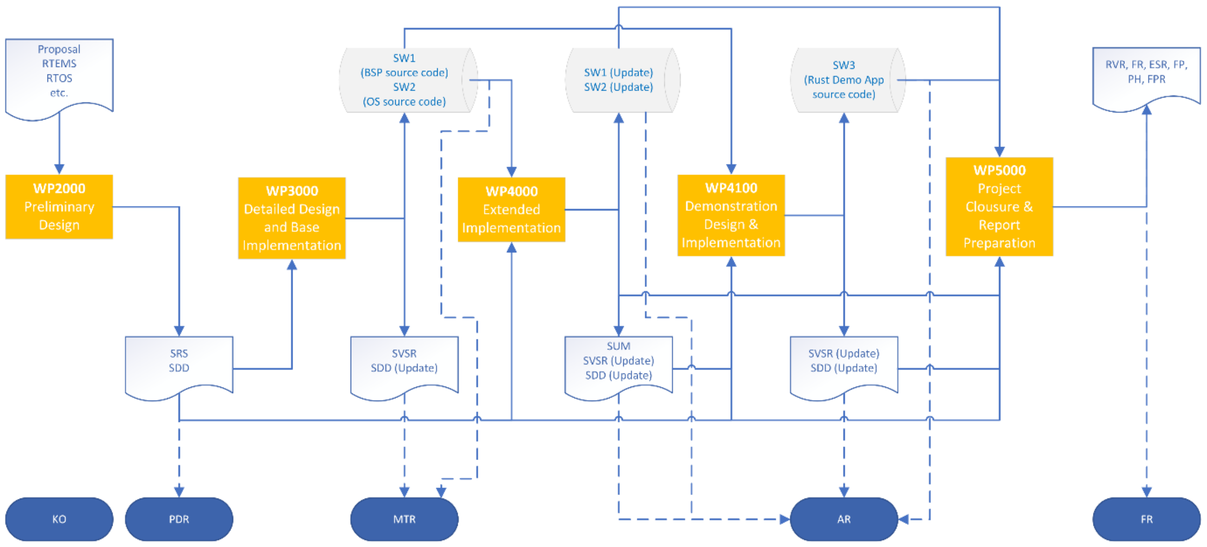

Figure 1 2– Detailed project work logic

4 Activity Overview

The project was kicked-off on the 1st of February 2023 and begun with requirements consolidation, preliminary software design preparation and choice of I/O drivers. That phase of the project ended with Preliminary Desing Review (PDR), held on 4th of April 2023.

The next phase of the project focused on the implementation of a basic executor capable of running tasklets and the minimal BSP required by the Rust runtime, detailed software design and software validation. That project phase finished with Mid-Term Review (MTR) on 22nd of August 2023.

After the MTR, the implementation was continued, focusing on developing the agreed drivers required for communication and data acquisition, as well as on execution time monitoring facilities for the system and a demonstration application development. Implementation phase ended with Acceptance Review (AR), held on 9th of January 2024.

The last project phase was dedicated to Rust in Space Viability Report preparation and was finished on 30th of January 2024 with Final Review (FR).

5 Software implementation results

5.1 Aerugo Real-Time Operating System

Aerugo is a real-time operating system written in Rust, developed during this activity. It targets SAMV71 microcontroller based on the 32-bit ARM Cortex-M7 processor. RTOS is implemented in the form of an executor instead of a classic scheduler and doesn’t support preemption. Its core design was inspired by purely functional programming paradigm and transputers architecture.

Instead of traditional tasks based on threads, the executor runs tasklets, which are fine-grained units of computation, that execute a processing step in a finite amount of time. They can be used to share stack, avoid context switches, and provide constrained concurrency patterns which may be more predictable than the ones which can be created using traditional threads.

Each tasklet is assigned one user function, whether it is performing some operation on received data, or reacting to an event. That concept fits with the constraints enforced by Rust and formalizes the patterns already used in flight software.

5.2 Board Support Package

The BSP was designed in line with the standards in the embedded Rust community. It consists of two parts. First is PAC – Peripheral Access Crate – a library that provides safe and structured access to the registers and peripherals of a microcontroller. This crate allows the developer to interact with hardware components while leveraging the language's safety features. It was generated by the svd2rust tool, which takes SVD file provided by the hardware manufacturer and outputs Rust structures. The second part is HAL – Hardware Abstraction Layer which uses PAC to provide an abstract and uniform interface for the hardware. This layer serves as an intermediary, enabling developers to interface with hardware in a standardized and modular manner, abstracting away low-level intricacies for better code readability, maintainability, and portability.

6 Validation overview

6.1 Hardware setup

Validation was performed on a hardware setup based on the development board for SAMV71Q21 microcontroller, consisted of the board with debugger connector (JTAG) attached to Raspberry Pi and made available over N7S intranet. The Raspberry Pi was also used as an interface to various external peripherals via dedicated dongles. Those interfaces were used for integration testing of Aerugo and BSP. The Continuous Integration system was set up on GitHub Actions to cyclically run unit tests. Integration tests were run manually by the developers on the actual hardware.

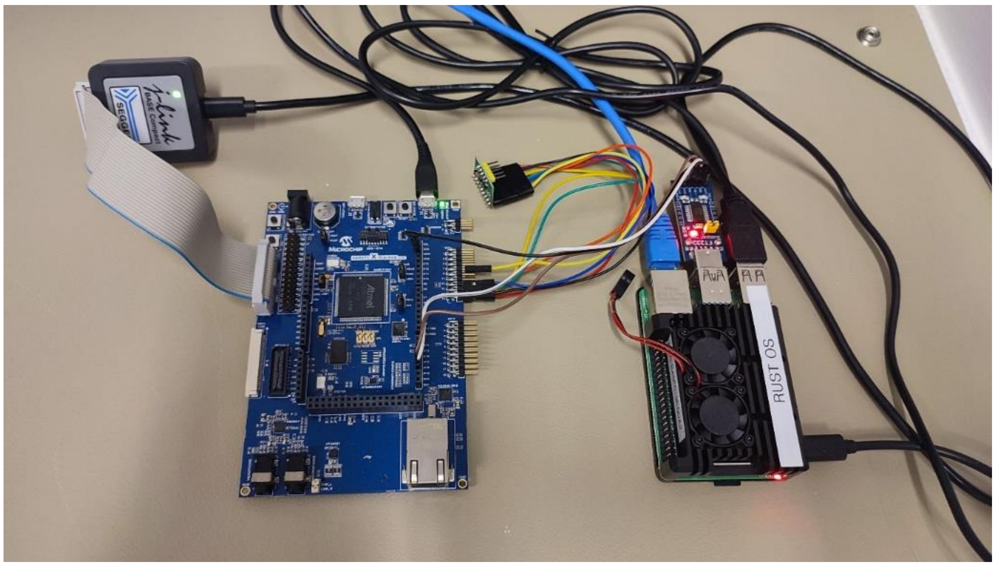

Figure 2 – Complete connected SAMV71Q21 Aerugo RTOS test setup

7 Rust in Space Viability Report – summary

7.1 Strong and weak sides of Rust

Rust's strength lies in its dedication to memory safety, a crucial aspect in systems and embedded programming. The language achieves this through a well-designed ownership system and borrowing mechanism. By enforcing strict rules during compilation, Rust helps prevent common memory-related errors like null pointer dereferences and data races, which often lead to vulnerabilities or hard to find bugs in languages with manual memory management like C and C++.

Beyond its commitment to memory safety, Rust delivers high-performance capabilities, making it a compelling choice for systems and embedded programming. The language's focus on zero-cost abstractions allows developers to express powerful constructs without sacrificing runtime efficiency. This is especially important for working with low-powered devices which are the most prevalent in the space industry. Rust's ownership system, which enables precise control over memory allocation and deallocation, contributes significantly to its performance benefits. In comparison to C and C++, traditional choices for embedded programming, Rust's performance characteristics are notable. While C and C++ offer low-level control over hardware resources, Rust introduces modern features and abstractions without compromising on execution speed. Furthermore, Rust's ecosystem actively embraces performance optimization, with a focus on minimizing runtime overhead. The language's ability to seamlessly interface with existing C libraries provides a way to use well-established and well-tested performance-critical libraries without having to rewrite everything from scratch.

Rust supports built-in documentation tests, and examples. The language's tooling, such as `cargo` and `rustdoc`, enables developers to seamlessly integrate documentation directly into their codebase. Documentation tests not only serve as a form of living documentation but also ensure that examples provided in the documentation are kept up-to-date and accurate. Additionally, the inclusion of documentation examples allows developers to understand and implement functionalities quickly. This emphasis on comprehensive and easily accessible documentation contributes to a developer-friendly ecosystem, enhancing code understanding, and promoting good programming practices.

Rust's strength extends beyond its language features to encompass an active ecosystem, supported by an engaged community. The language's ecosystem is marked by a growing repository of libraries, frameworks, and tools that cater to diverse needs, fostering a collaborative and modern development environment. The active participation of the Rust community is a cornerstone of this ecosystem's vitality. Developers benefit from a wealth of resources, including forums, documentation, and open-source contributions, making it easier to share knowledge, troubleshoot challenges, and discover best practices. The community's commitment to inclusivity and knowledge-sharing ensures that Rust remains accessible to developers of varying experience levels.

Despite compelling features, the broader integration of Rust into space development faces several challenges. Addressing those challenges is an important factor in considering the feasibility of using it in space applications.

The learning curve associated with Rust stands out as a prominent factor influencing its adoption. Rust's unique ownership system, designed to guarantee memory safety without a garbage collector, demands a paradigm shift for developers accustomed to memory management model from languages like C, C++, or Ada. Understanding and mastering concepts such as ownership, borrowing, and lifetimes is often perceived as challenging, requiring developers to invest significant time and effort to become proficient in Rust.

The learning curve mentioned above is a significant aspect to consider when migrating from C to Rust. In C, developers wield explicit control over memory allocation and deallocation, allowing for flexibility but also opening the door to common pitfalls such as buffer overflows and dangling pointers. As previously mentioned, Rust's ownership system and borrowing mechanisms prevent those pitfalls by design, but this design is quite different from the design of C language. Moreover, the learning curve in transitioning from C to Rust extends beyond memory management to encompass Rust's borrow checker and lifetime system. In C, developers enjoy a more permissive model, allowing shared mutable references and pointers, which can lead to subtle bugs and challenges in concurrent programming. Rust's borrow checker, however, ensures memory safety by restricting certain operations, unfortunately also introducing a layer of complexity for developers unfamiliar with those constraints. Furthermore, in C, error handling often involves return codes, and managing concurrent operations typically relies on low-level threading primitives. Rust introduces a more systematic and expressive error handling mechanism with the Result and Option types, along with the concept of ownership, to facilitate robust error management.

The Rust toolchain, while robust, faces certain challenges that impact its seamless integration into development workflows. Rust's emphasis on strict compiler checks and static analysis contributes to a rigorous development environment, ensuring code reliability. However, this can also lead to longer build times, particularly for large projects, which may hinder the iterative development process. Additionally, the availability and maturity of certain development tools and libraries in the Rust ecosystem compared to more established languages can present obstacles. Although Rust community actively expands the ecosystem, including the embedded field, providing either support for existing tools (i.e. using OpenOCD for executing code on the actual hardware) or creating new (i.e. svd2rust used to generate PAC structures from SVD files provided by the hardware manufacturer).

Compiler support for different hardware targets is another facet influencing Rust's adoption landscape. Rust's commitment to zero-cost abstractions and performance is evident in its emphasis on generating efficient machine code. However, challenges arise in achieving broad compatibility across various hardware architectures. While Rust's LLVM-based compiler provides a solid foundation for producing optimized code, ensuring consistent and comprehensive support for diverse platforms remains an ongoing effort. Projects targeting specialized hardware (i.e. SPARC-based LEON family of processors) may encounter limitations or even lack of support whatsoever. Compiler support for hardware targets in Rust operates within a framework of different tiers. The Rust project categorizes platforms into different tiers based on the level of support provided by the compiler. Tier 1 ensures the highest level of support, with regular CI builds and running a full test suite. However, as one moves to lower tiers, the level of support may vary, with some platforms being community-maintained or having limited features. Toolchain (`thumbv7em-none-eabihf`) for SAMV71 (ARM architecture) processor used in this project is placed in the Tier 2, although no problems regarding support for this platform were encountered during the development. All of the written code was successfully compiled and then run on the hardware.

Another aspect influencing the adoption of Rust revolves around the availability and stability of language features. While Rust boasts a range of cutting-edge features designed to enhance developer productivity and code expressiveness, a significant number of these features remain exclusive to the nightly compiler or are labeled as experimental. This means that they are not yet considered stable or suitable for production use. Certain features deemed highly beneficial for developers have lingered in a state of limbo for an extended duration, despite their evident utility. Some of these features, with the potential to significantly enhance the language's capabilities, face challenges in reaching stabilization or full incorporation into the stable release. Adding to that are features present in languages like C or C++ which are missing in Rust, such as the ability to use `#define` to set a value for a constant via compiler flags. In this project that had to be worked around with environment variables which a custom parser for them, as Rust doesn't yet support non-string values yet.

7.2 Project's lessons learned

During this activity Rust programming language was used to create a real-time operating system and a HAL targeting SAMV71 microcontroller. This work gave feedback for this report which goal was to determine whether Rust is a suitable choice for further investing.

During the development of Aerugo, we've learned that Rust with its ecosystem can be a good tool for creating critical applications. As a language, it was difficult to grasp at first, but after some time the experience of working with Rust got considerably better. It introduces new concepts, like borrow checking and move-semantics-by-default, that usually allow to improve application's safety and performance in comparison to other types of languages. Rich core library proved to be a very useful tool in the process of creating Aerugo, providing not only the most basic utilities – like fundamental types, traits, and simple memory operations support – but also higher-level utilities, like iterators and Unicode strings support, with a very rich implementation. This is a very big step-up from standard C library, as it saves a lot of development time, however, it must be kept in mind that the output binary size is closely dependent on the amount of used features. This can be a downside in environments with very constrained memory resources, but on the other hand Rust follows the C++ philosophy of "you don't pay for what you don't use", which allows it to run in those environments by stripping the unnecessary features away.

Rust's ownership system gives the developer an easy way of controlling object's lifetimes and changing their ownership, and the borrow checker protects the developer from breaking its rules during compilation. It also unlocks new design patterns, like typestate pattern, giving developers more ways of defining object's behaviors. Implementing communication interfaces (like UART or SPI) using a typestate pattern proved to be relatively easy, and that implementation prevents the user from misusing the interface by forcing it into potentially invalid state, or misconfiguring. It's a very desirable property for critical software – if the developer can't break the driver while using it, the probability of introducing a critical bug in the code is drastically reduced by default.

The traits system proved to be a very robust alternative to the object-oriented inheritance system. Built-in traits work not only as ordinary interfaces but can also work as markers for Rust's type system and change the low-level behavior of type's values (for example, implementing `Copy` trait allows the value of a type to be implicitly copied, instead of being moved, which may be desirable for small types or type wrappers).

During the development, one of the biggest issues with Rust was the compilation time. Compilation times of Rust are known to be long, in part due to the number of safety checks performed. Rust performs incremental compilation by default, reducing the severity of this issue in day-to-day operations, but full rebuilds took considerably more time compared to builds of similarly sized projects written in C or C++. This mostly affects the CI/CD pipelines, as the code is often built from scratch there. Additional code checks also take considerable amounts of time, comparable to the time of compilation.

The existence of out-of-the-box, easy-to-use tools for testing and documenting the project's code is one of the most important properties of Rust language and its ecosystem in context of space applications. Cargo, the Rust package manager, proved to be an invaluable tool for this project's management. Not only did it make managing the project and its dependencies easy, but also provided utilities for testing and generating the documentation. Built-in support for unit and integration tests made creating those relatively simple and quick – additional tools for integration testing on embedded hardware were required, but Cargo was used as the final runner to generate uniform reports from all the tests, and it was possible to integrate it with external tools. Cargo can generate documentation in HTML form by extracting the comments from code. It also supports running code blocks directly from documentation – this can be used to verify whether the example embedded in documentation is still valid, and those code blocks can become test cases themself. Providing the documentation for the code can be enforced in project's configuration – Rust compiler can throw an error or a warning on undocumented item. It's also worth mentioning that all of the tools in Rust ecosystem are well documented, using the documentation generated with Cargo and community-standardized tools (`rustdoc`, `rustbook`).

Aerugo design was influenced by FreeRTOS – the idea of message queues connected to tasks, and tasks being initialized by code at initialization was based on it. FreeRTOS was also chosen as an inspiration due to being relatively simple and lightweight – which we consider very important for space-grade RTOS.

7.3 The way forward

Considering those conclusions there are a few ways forward that could be taken regarding the system that was created in this activity. One way would be to implement asynchronous executor. This was considered to be included in the scope of this activity, but due to the maturity of the async functionality in Rust and the budget it had to be omitted. Now, as the base of the system is completed, adding such functionality could greatly enhance system usability. Asynchronous execution would be a fitting replacement for the preemption, as it allows tasks to voluntarily yield control, enabling efficient multitasking without relying on preemptive context switches. This not only enhances responsiveness in concurrent operations but also simplifies code by eliminating the need for explicit synchronization mechanisms, aligning with Rust's focus on safety and performance.

Another possibility is to further develop SAMV71 HAL. During this activity only the basic peripherals could be implemented, and there are many remaining, which are of a common use in space industry like MCAN, SDRAMC, GMAC or TWIHS. As the HAL is separate from the system itself, adding more features would greatly improve its usability in other projects. SAMV71 is widely used in space industry but creating HAL for its rad-hard version – SAMRH71 – could also be considered.

The goal was to create RTOS that is complete in core features and allows to make working space systems, which was also validated with the demonstration application which was also a part of this activity. As a next step it could be qualified according to ECSS standard to the category B. That would further examine Rust suitability for space applications and give feedback on how well Rust safety features could contribute to fulfilling criticality requirements.

8 Conclusions

The viability of Rust for space applications represents an important exploration within the area of mission-critical software development. The distinctive attributes of Rust make it a compelling candidate for use in space applications, where reliability and safety are the most important factors. As space missions increasingly require sophisticated, reliable, and efficient software systems, Rust's emphasis on safety, performance, and modern language features could make it a great fit in the space software area.