Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

→ DISCOVERY

cRustacea in Space - Co-operative RUST and C embedded applications in Space - Theory and Practice

Executive summary

Early technology development

Related OSIP Campaign
New concepts for onboard software development

Affiliation: DLR - German Aerospace Center (Prime)

Activity summary:
This activity investigates the feasibility of using Rust as a programming language for onboard software development in space missions. A port of the Rust standard library to RTEMS was successfully implemented, demonstrating Rust's potential for efficient and reliable onboard software development.
The research also focuses on quality assurance aspects relevant to ESA ECSS standards, including documentation, testing, and product metrics. A working prototype of an onboard system was developed and tested using Rust, enabling thorough evaluation and improvement of the language's suitability for space mission applications. The results show that Rust complies with different aspects of the ECSS-E-ST-80C standard, offering improved safety and reliability in space missions.

Publishing Date: 2024-07-31
Contract Number: 4000140242
Implemented as ESA Initial Support for Innovation

→ THE EUROPEAN SPACE AGENCY
ESA Discovery & Preparation
From breakthrough ideas to mission feasibility. Discovery & Preparation is laying the groundwork for the future of space in Europe
Learn more on www.esa.int/discovery
Activity Page: https://activities.esa.int/4000140242
Deliverables published on https://nebula.esa.int

1. Introduction

Onboard software development for space applications demands high standards in determinism, code quality, and product assurance. As new missions require increasingly complex functionalities, traditional approaches become less feasible. To meet these stringent requirements while ensuring software quality and safety, new approaches are being explored.

Rust, a promising language, offers performance levels comparable to C programs with safety guarantees at compile time and parallel execution features. However, the maturity of Rust's development tools needs investigation for supporting complex space missions.

2. Project Background

This project is a stepping stone to allow space-qualified software systems (or parts of them) to be developed in Rust for real-life missions. Previous studies have explored Rust's feasibility and performance for implementing space-related applications. Although Rust has shown its versatility across various hardware architectures and operating systems, several roadblocks need to be overcome before serious adoption in space missions:

1. Rust needs to compile on common operating systems used in space missions as well as target hardware.
2. Rust must interact seamlessly with existing C code to be adopted by key players. A gradual transition to Rust development is essential.
3. Rust must be qualifiable with existing quality and product assurance standards, such as the ECSS standard family.

This activity therefore looked at the current state of the Rust ecosystem for the RTEMS real-time operating system and tracked the effort to enable it to run Rust code. Additionally, it compared the developer experience for Rust development with the efforts needed for a similar application developed in C under the premise that both languages start "from scratch" without any existing development framework. Finally, the aspects of product assurance in the framework of ECSS standards were investigated. That means the applicability of standard requirements, the availability of needed quality metrics and how to obtain them.

3. Methodology

This project addressed each of these roadblocks by investigating:

1. Porting Rust to the RTEMS operating system
2. Developing a working prototype for testing and improvement
3. Investigating product assurance aspects in the framework of ECSS standards

3.1 Porting Rust to the RTEMS operating system

The RTEMS RTOS is widely used in the European space industry and several ESA funded activities have contributed key improvements to the operating system, e.g. the implementation of Symmetric Multiprocessing (SMP) support as well as the preparation of pre-qualified version RTEMS-QDP. As RTEMS itself is a FOSS project the wider RTEMS community benefitted from those funded activities.

In order to enable the Rust compiler to compile Rust code for RTEMS targets, first a target platform was selected. The Xilinx Zedboard was selected because the corresponding RTEMS board support package (BSP) is among the best supported, the target can be simulated by an emulator and real hardware is readily available.
Next, the correct configuration for the Rust compiler were derived which produce machine code compatible with the RTEMS kernel for the selected target and which could be linked into an executable binary.
This initial simple setup could only provide the very basic language features of Rust and would not support many of the desired safety related features. Therefore, further additions to the Rust compiler and standard libraries were investigated to enable higher-level language features.
With this information we added RTEMS bindings to the Rust-libc and ported the Rust standard library to RTEMS. These additions were contributed to the open Rust community project.

3.2 Developing a working prototype for testing and improvement

This activity implements three ECSS Packet Utilization Services—Function Management, Parameter Management, and Housekeeping—in both Rust and C to compare the two languages. Both implementations interact with a mock-up sensor developed in C and follow the same architecture. The comparison focuses on differences in memory management, code readability, and writability, with a particular focus on the experience of learning Rust for developers familiar with C. Key factors include Rust's ownership model for memory safety, the impact of syntax and language constructs on code understanding, and the ease of adapting to Rust's tooling and modern features.

3.3 Investigating product assurance aspects in the framework of ECSS standards

This work package gives an overview and analysis of key language features impacting product assurance objectives, specifically within the context of ECSS-Q-ST-80C standard for missions at different levels of criticality. The interplay between these requirements and software technology is examined, with particular attention to how they can be addressed taking into consideration Rust language, focusing on memory safety and concurrent task execution. In addition, unit testing, static analysis and code coverage tools are implemented for the project application. Furthermore, a separate evaluation is also presented, highlighting specific practical software technology aspects related to ECSS-Q-ST-80C that were considered during this activity.

4. Key Findings

Rust support for RTEMS including porting the Rust standard library can be achieved with reasonable effort. The before mentioned Xilinx Zedboard BSP is now available as an official Tier 3 target as part of the Rust compiler. This makes it easier for interested parties get a working cross-compiler installed for further testing and development. With the initial port integrated into the official sources it is now also much easier to add further improvements as well as RTEMS ports for new targets. Also, it is possible to use the Rust build system “*cargo*” to build a working RTEMS executable.
The implementation of the three ECSS Packet Utilization Services in Rust and C was examined, with a focus on memory safeness, readability, and writability. The memory analysis showed that both languages showed minimal memory leaks in non-complex functions but tended to exhibit leaks in more intricate code, though these were relatively low compared to allocated memory. Regarding memory consumption, both implementations performed similarly, with Rust's House Keeping service showing a slight deviation likely due to specific implementation details. Overall, language choice did not significantly impact outcomes for this small-scale project, emphasizing the importance of developer awareness in managing memory issues. In terms of readability, C's

traditional syntax is familiar but verbose and error-prone, while Rust's modern features enhance clarity and maintainability despite initial complexity with ownership and borrowing concepts. Rust's focus on safety and structured error handling with the Result type contrasts with C's ad-hoc approaches, promoting better software development practices over time. In writability, C offers simplicity but demands expertise in memory management and concurrency, whereas Rust, once mastered, facilitates safe and efficient code writing supported by modern tooling like Cargo. Rust's compile-time safety checks mitigate runtime errors, contrasting with C's need for vigilant programming to prevent vulnerabilities.
Finally, it was found that 29 from ECSS-Q-ST-80C and 14 requirements from ECSS-E-ST-40C could potentially be impacted by Rust features related to software product assurance, quality assurance and engineering requirements. The analysis revealed direct or indirect effects on these requirements. The second activity involves the application and evaluation of various tools for static analysis, unit testing and code coverage.

- *rustc* and *rust-clippy* were applied for static analysis, with *rust-clippy* offering additional lints and a feature to create custom metrics. In this regard, its capabilities are more complete to fulfill a more exhaustive inspection of the source code.
- *Cargo-test* and *cargo*-nextest were integrated for unit testing evaluation, with cargo-*test* currently providing more guarantees in terms of platform support, availability and maintainability, but with worse performance compared to what *cargo-nextest* aims.
- For code coverage, both *tarpaulin* and instrumentation-based approaches were tested. Both coverage tools rely on LLVM, which increases dependability and may affect maintainability. However, *tarpaulin* capabilities are limited since, at the moment, it only provides line test coverage.

5. Conclusion

The goal of this project was to enable the use of Rust as a viable option for onboard software development in space missions. The results show that with reasonable effort, Rust can be used on top of the well-known RTEMS real-time operating system, and compared to C, it provides similar developer experience while offering additional benefits such as safety guarantees at compile time. Furthermore, the project has given an overview of the application of ECSS standards for product assurance when using Rust in terms of its features and the surveyed tools. The availability of needed quality metrics and how to obtain them were also investigated in the context of software quality assurance.
The findings of this project will help pave the way for the use of Rust in space missions, enabling more efficient and reliable onboard software development.
While C remains a robust choice for systems programming, known for its performance and low-level control, Rust's advanced safety features and modern practices make it a compelling alternative, particularly in safety-critical applications. However, Rust presents a steep learning curve for developers accustomed to C. This complexity primarily arises from Rust's ownership model and strict compile-time checks, which enforce memory safety and prevent data races but require a significant shift in mindset. Despite this challenge, Rust's potential for broader adoption is supported by its growing ecosystem and community, making it an attractive option for future development where performance and safety are crucial.

6. Future Work

With the initial port of the Rust standard library ported to RTEMS and available to the broader community the next step would be to test a wider array of Rust features on the RTEMS operating system and stabilize the port. Furthermore, additional ports for other RTEMS target architectures,

especially the more space related ones like the upcoming Leon and RISCV processor generation would be of great importance. Many parts of this activity can be re-used for this goal. Also, the qualification of the Rust compiler and standard libraries should be considered to fulfill the quality assurance goal of future space projects.
A more thorough examination of other available tools is warranted, coupled with an in-depth analysis of their performance characteristics. Such an investigation could provide further insight and knowledge necessary for defining the software product assurance requirements being investigated.
To demonstrate further Rust's ability to compete with standard programming languages, as C, we recommend to continue the development of typical space segment applications. An entire implementation of the PUS standard in Rust as Open Source followed by a demonstrator mission would show that Rust is suitable as programming language for critical software components. Furthermore, the European space industry could benefit from such an open source implementation.