# RUST FOR SPACE APPLICATIONS AND RTEMS

The good, the bad and the ECSS

DLR Institute of Software Technology, Department for Flight Software

Jan.Sommer@dlr.de
Andreas.Lund@dlr.de
Hany.Abdelmaksoud@dlr.de
Tamara.GutierrezRojo@dlr.de

DLR

# ESA activity: cRustacea in Space

**Evaluate Rust for spacecraft onboard software development regarding**

- Execution on embedded targets
- Execution on RTEMS operating system

WP01

- Developer friendliness
- Integration with legacy C-code

WP02

- ECSS standard conformity
- Qualification effort

WP03

# WP01: Questions investigated

**Is Rust viable for our targets?**

- Can we create cross-compilers for our hardware targets, i.e. Zynq (and later probably Leon/Noel)?
- Can we create cross-compilers for our used operating systems i.e. RTEMS?
- And both at the same time?
- Which Rust features are then available for us?
  - Multithreading?
  - std library?
  - Unit test execution?

A small orange crab holding a white sign clipart

# WP01: Explored options for Rust application on RTEMS

Build Rust toolchain for Zynq (bare metal) → Determine correct target flags → Build [no_std] application for RTEMS → Port stdlib to RTEMS → Build Rust application for RTEMS

(Im-)Possible alternative

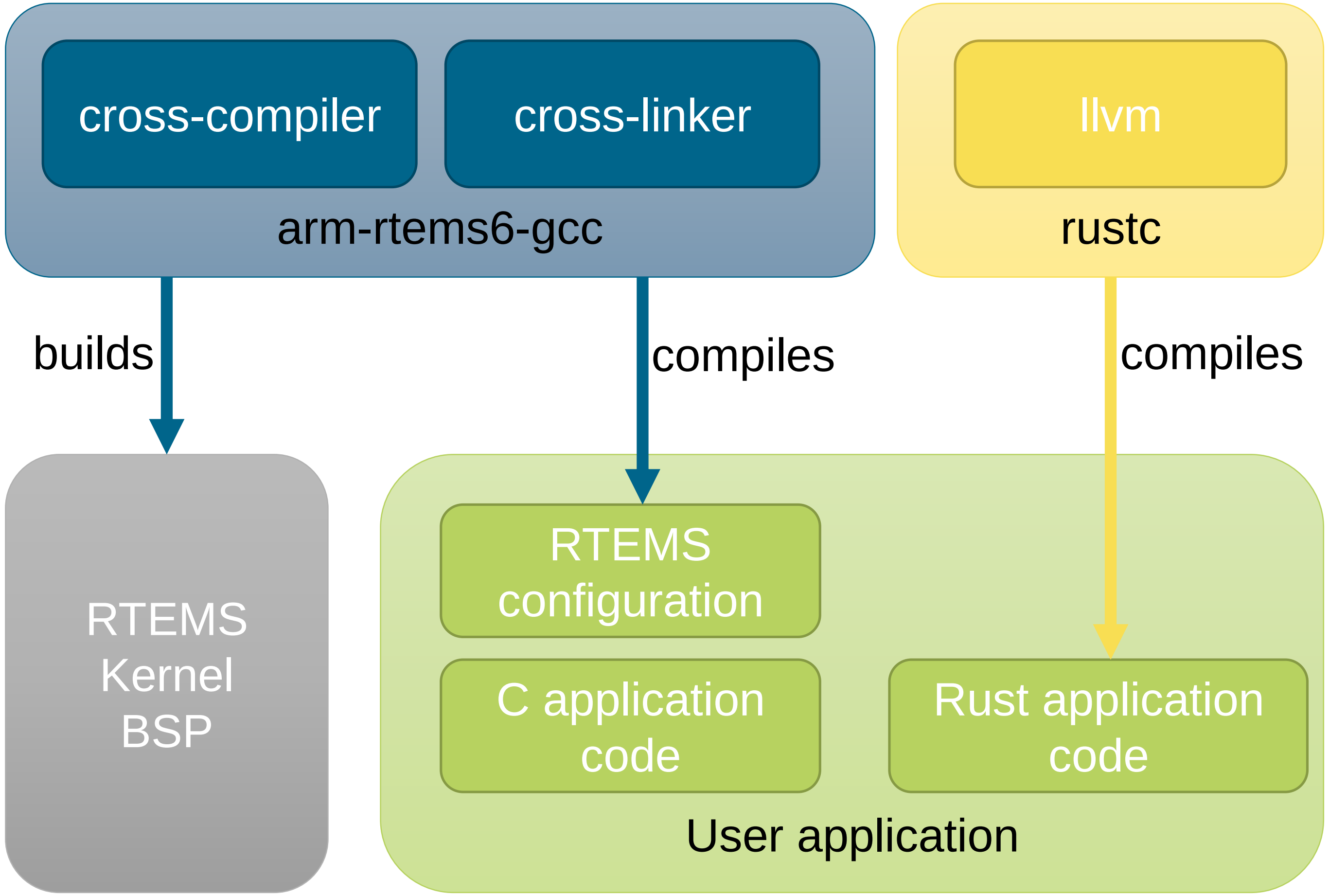
Check out GCC Front-End for Rust → Try to build gcc cross-compiler for RTEMS → Build Rust application for RTEMS

Currently the project is in a too early stage to be useful for our work. Still worth to follow future developments.

https://rust-gcc.github.io/

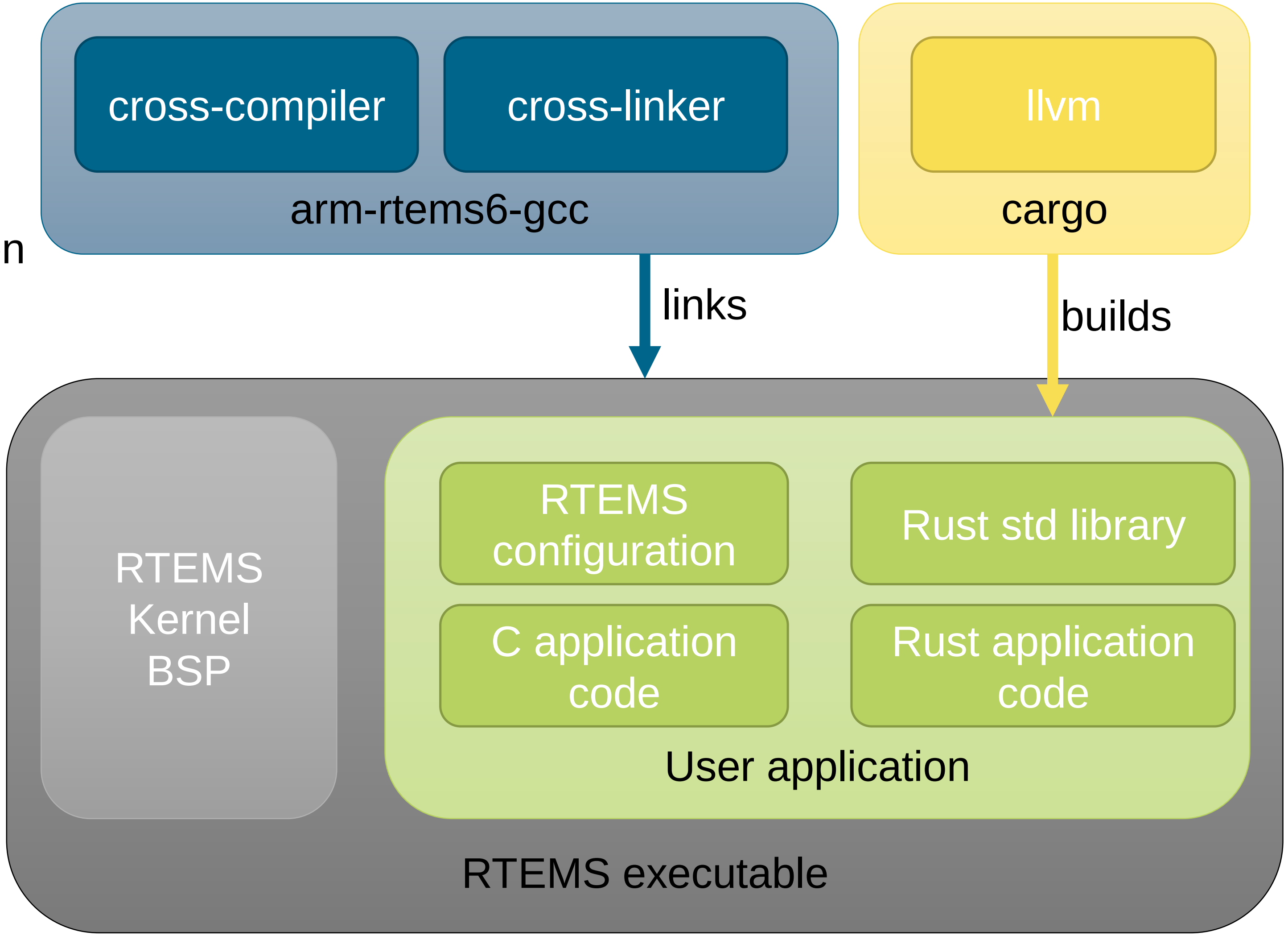
# WP01: Build #[no_std] application for RTEMS

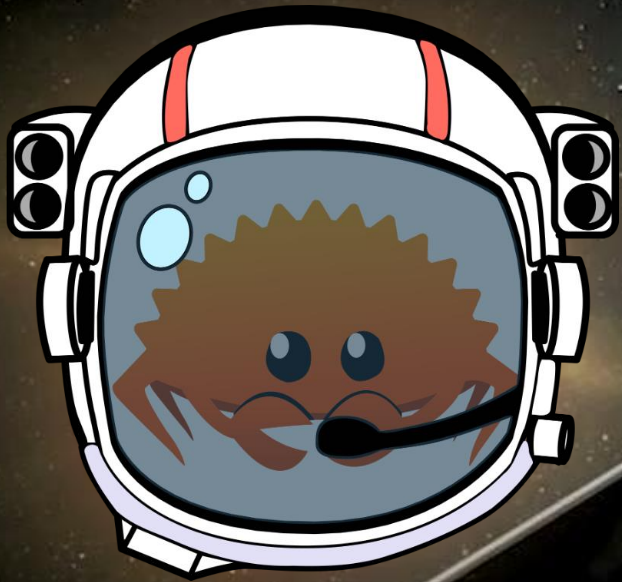
- Use the standard rustc compiler for compiling user application
  - Use #[no_std], #[no_main]
  - Export Rust functions to extern C
- Compile parts separately
  - Init C-code for RTEMS with `gcc` cross-compiler
  - Rust code with `rustc` compiler
  - Call exported Rust functions from C



# WP01: Porting stdlib to RTEMS

- `armv7-rtems-eabihf`
  - New Tier 3 target in Rust compiler
  - Includes `std` library
  - Update of RTEMS documentation soon
- Cargo builds RTEMS application
  - Determines compile flags
  - Compiles C and Rust code
  - Links final binary



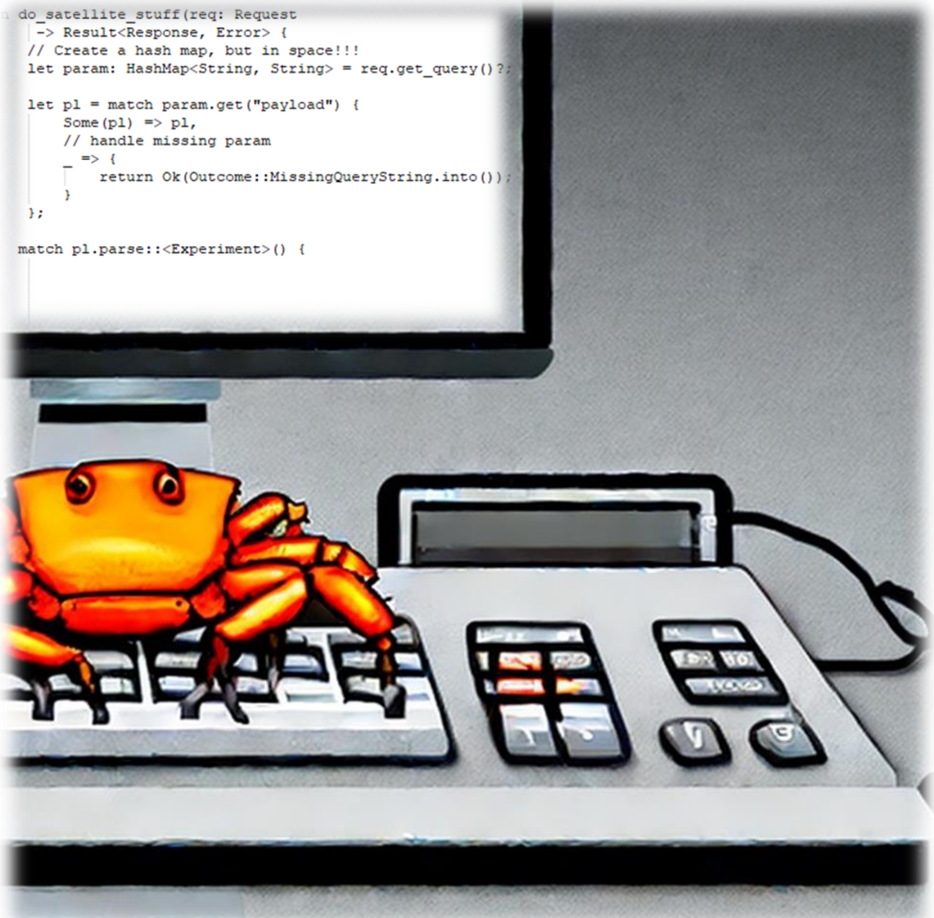


# WP02: ON-BOARD APPLICATION EXAMPLE

# WP02: Questions investigated

## Is a move to Rust worth it?

- Are the Rust language features really that beneficial?
- How does Rust code integrate with our development infrastructure (gitlab, JUnit tests, Doxygen, static analysis)
- How does Rust integrate with existing code (C/C++)
- How much effort is it to learn Rust as a C/C++ developer?

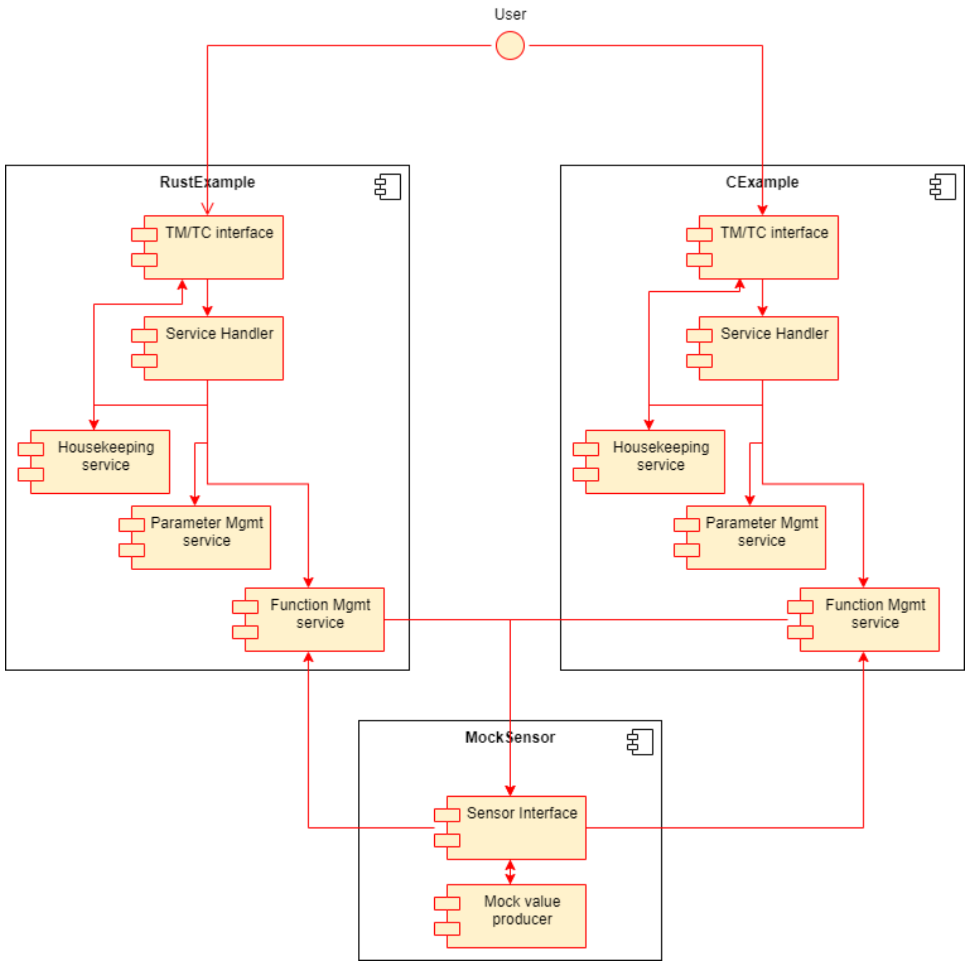


A small orange crab sitting on a keyboard of a desktop pc clipart

# WP02: Use case example

- Comparison RUST vs. C
  - Collecting differences in development
  - Compare the reliability (e.g. by Valgrind)

- Minimal example for common satellite application
  - Command & Data Handling → PUS services:
    - Service 3 Housekeeping
    - Service 8 Function Management
    - Service 20 Parameter Management
  - Mock Sensor in C



# Comparison - Research

We compared the implementation of three ECSS Packet Utilization Services in Rust and C.
Our comparison focused on the following key points:

- **Memory safeness and consumption**
    - Memcheck
    - Heap consumption
- Developer friendliness
    - **Readability**
        - Syntax and Expressiveness
        - Ownership and Borrowing
        - Lifetimes
        - Error Handling
    - **Writability**
        - Learning Curve
        - Tooling and Ecosystem
        - Debugging and Testing
        - Code Safety

RUST vs. C clipart

# Developer Friendliness / Readabiltiy

- **Syntax and Expressiveness:**
  - **C:** Traditional syntax with explicit control structures like if statements and for loops.
  - **Rust:** Modern syntax with pattern matching (match) and iterators for concise, readable, and maintainable code.

- **Ownership and Borrowing:**
  - **C:** Manual memory management, prone to dangling pointers and undefined behavior.
  - **Rust:** Ownership and borrowing rules ensure safe memory management, preventing data races and enhancing reliability but can be complex at first.

- **Lifetimes:**
  - **C:** No explicit lifetime management, risking unsafe memory access in concurrency.
  - **Rust:** Explicit lifetime annotations ensure memory safety, especially for concurrent programming, adding complexity but guaranteeing data validity.

- **Error Handling:**
  - **C:** Relies on return codes and structs, leading to unstructured error handling.
  - **Rust:** Uses the Result type for structured error handling, making code more predictable and interfaces clearer.

A small orange crab reading a book clipart

# Developer Friendliness / Writability

- **Learning Curve:**
  - **C:** Easier for basics but requires deep understanding for concurrency and memory management.
  - **Rust:** Steeper learning curve due to ownership rules, but compiler aids learning. Takes more time initially.

- **Tooling & Ecosystem:**
  - **C:** Mature ecosystem; manual integration for libraries and tools.
  - **Rust:** Modern tooling (cargo) for simplified package management and rapid ecosystem growth.

- **Debugging:**
  - **C:** Uses GDB; requires deep memory and hardware knowledge due to manual memory management.
  - **Rust:** Also uses GDB but benefits from compile-time checks and safer memory management, reducing runtime errors.

A small orange crab writing a book clipart

# Developer Friendliness / Writability(2)

A small orange crab writing a book clipart

- **Testing:**
  - **C:** Lacks built-in testing; requires external frameworks.
  - **Rust:** Built-in testing framework integrated with cargo, simplifying testing processes.

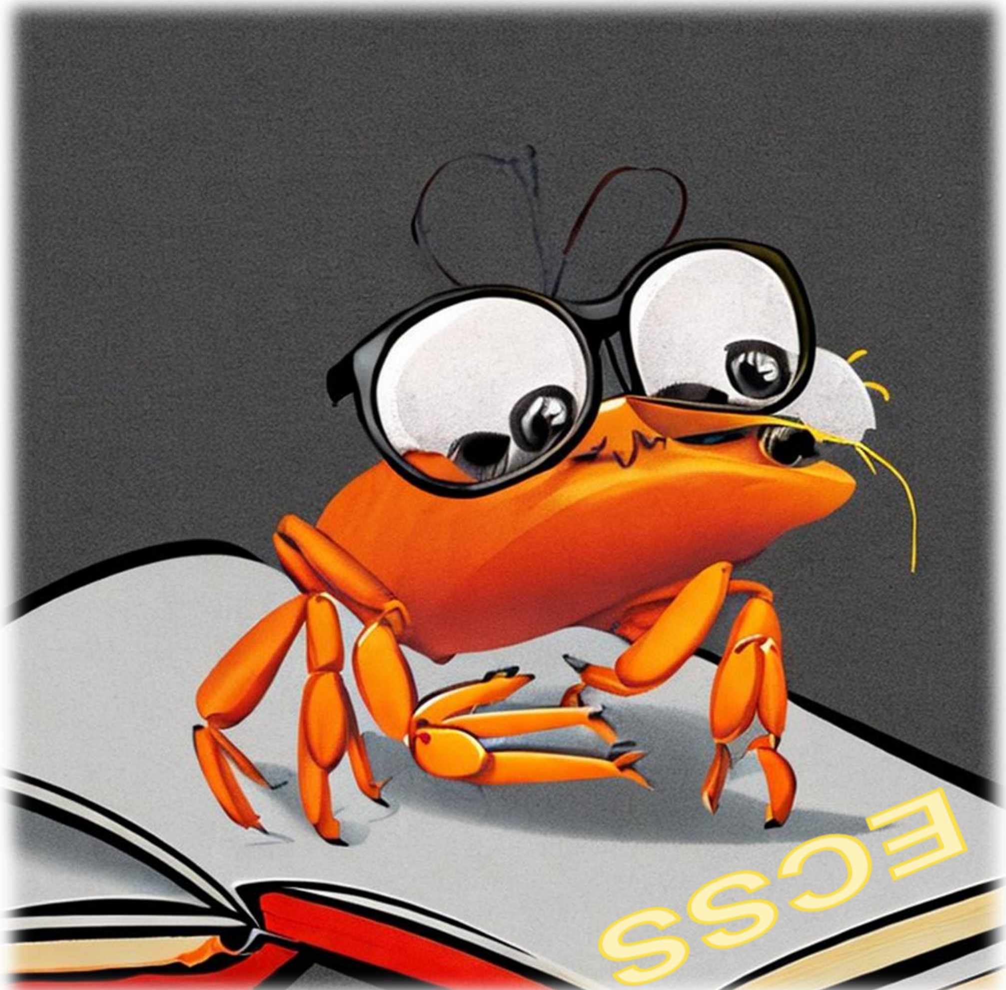
- **Code Safety:**
  - **C:** Programmer-managed safety; prone to undefined behavior and vulnerabilities.
  - **Rust:** Enforces compile-time safety, preventing many common issues for more secure code.

# WP03: PA ASPECTS OF CRITICAL SOFTWARE IMPLEMENTED IN RUST

# WP03: Questions investigated

**How difficult is it to qualify Rust Code?**

- How easy is it to extract code metrics from Rust?
- Are all metrics from C needed?
- Is it possible to qualify Rust code according to ECSS standards?
- Does using Rust simplify the qualification process?



A small orange crab wearing glasses sits on a big book cartoon

# WP03: Applicable Standard documents

- ECSS-Q-ST-80C
  - 29 relevant requirements identified


ECSS-Q-ST-80C
6 March 2009
EUROPEAN COOPERATION
ECSS
FOR SPACE STANDARDIZATION
Space product assurance
Software product assurance

- ECSS-E-ST-40C
  - 14 relevant requirements identified


ECSS-E-ST-40C
6 March 2009
EUROPEAN COOPERATION
ECSS
FOR SPACE STANDARDIZATION
Space engineering
Software

# WP03: Testing and Collection of Code Metrics

- Unit testing, static analysis and coverage tools were analyzed and surveyed

| Tasks | Surveyed tools |
|---|---|
| Build | cargo |
| Static analysis | rustc<br>rust-clippy |
| Unit tests | cargo-test<br>cargo-nextest |
| Code coverage | **llvm-cov (instrumentation-based)**<br>clang (source-based)<br>**cargo-tarpaulin**<br>grcov<br>kcov |

# WP03: Key Findings


- Rust’s strong focus on compilation allows us to:
  - Use *rustc* as static code analysis tool → might reduce effort compared to C
  - Get instrumentation-based code coverage
- Impact on qualification process
  - Memory safety and Rust‘s focus on concurrency are features that may be beneficial
  - *cargo-test* provides robust **support**, **availability** and **documentation** for different platforms, and ensures **maintainability**
  - *rust-clippy* offers more metrics to measure code quality
  - *tarpaulin‘s scope* is currently limited to line coverage and *LLVM instrumentation-based* coverage provides branch coverage at an unstable level (impact on req. 6.2.3.2)

# Summary

- Rust and stdlib can be ported to RTEMS and added as a proper target
  - Port needs to stabilize. Ideally aim for a Tier 2 target
  - Add further architectures, like Leon/Noel processor family

- Rust can support developers in writing memory safe and concurrent code
  - Existing/qualified C code can still be used side-by-side → Gradual adoption possible
  - Needs some getting used to lifetimes and ownership rules

- Rust provides necessary tools for ECSS qualification
  - Many relevant requirements can be fulfilled, some need further evaluation
  - Rust ecosystem provides means to extract relevant metrics

# Wishlist and Future Work

- RTEMS QDP
  - Add the RTEMS POSIX API as it is used by the Rust port
  - Evaluate and qualify the Rust compiler for RTEMS
  - How to do schedulability analysis with RTEMS?
- Community Discussion about ECSS Qualification
  - Regarding static analysis metrics provided by the Rust compiler and *rust-clippy*
  - More thorough examination of other available tools for code coverage metrics (including branch coverage)
- Try out Rust in scoped real project
  - Including qualification according to ECSS


**We are happy to contribute**

# WP03: Relevant Requirements from ECSS-Q-ST-80C

## How difficult is it to qualify?

- Is it possible to qualify Rust code according to ECSS standards?

| ECSS-Q-ST-80C Requirement | Impact from tools usage | Future actions for complete evaluation |
|---|---|---|
| 5.2.1.1 | Indirect | N/A |
| 5.2.1.3 | Indirect | N/A |
| 5.2.1.4 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 5.2.5.1 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 5.3.1 | Partial | Evaluation on technology maturity aspects from the tools usage |
| 5.3.2.1 | Partial | Evaluation on technology maturity aspects from the tools usage |
| 5.3.2.2 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 5.6.1.2 | Partial | Further analysis on other code coverage tools and unit testing tools' performance |
| 5.6.2.1 | Partial | Further analysis on performance |
| 6.2.2.2 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 6.2.2.3 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 6.2.3.2 | Partial | Further analysis on other code coverage tools and unit testing tools' performance. |
| 6.2.3.4 | Indirect | N/A |
| 6.2.5.1 | Indirect | N/A |
| 6.2.6.1 | Indirect | N/A |

# WP03: Relevant Requirements from ECSS-Q-ST-80C

## How difficult is it to qualify?

- Is it possible to qualify Rust code according to ECSS standards?

| ECSS-Q-ST-80C Requirement | Impact from tools usage | Future actions for complete evaluation |
|---|---|---|
| 6.2.6.2 | Indirect | N/A |
| 6.2.7.2 | Indirect | N/A |
| 6.2.7.3 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 6.2.7.4 | Indirect | N/A |
| 6.2.7.7 | Indirect | N/A |
| 6.3.4.1 | Indirect | N/A |
| 6.3.4.2 | Indirect | N/A |
| 6.3.4.4 | Indirect | N/A |
| 6.3.4.5 | Partial | Evaluation on Rust compiler and static analysis metrics |
| 6.3.5.32 | Indirect | N/A |
| 6.3.7.2 | Indirect | N/A |
| 7.1.4 | Direct | Evaluation on Rust compiler and static analysis metrics. Further analysis on other code coverage tools and unit testing tools' performance |
| 7.1.5 | Partial | Further analysis on other code coverage tools |
| 7.2.2.1 | Indirect | N/A |