Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTMENT OF SYSTEMS AND COMPUTER ENGINEERING

Investigation and Application of the Rust Programming Language in Spacecraft Embedded Software

Internship Report to fulfil the Master's degree in Informatics Engineering

Specialization in Software Engineering

Author

Carolina Batista Veríssimo Lopes

Supervisor

Francisco José Baptista Pereira

Advisor in the organisation Critical Software S.A.

Daniel Chichorro de Carvalho

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, november 2025

RESUMO

A linguagem de programação Rust tem ganho destaque no desenvolvimento de sistemas críticos devido à sua abordagem inovadora para garantir segurança e eficiência na gestão de memória. Em contraste, linguagens tradicionais como C e C++ continuam amplamente utilizadas neste domínio, apesar de introduzirem riscos associados à gestão manual de recursos e à ocorrência de vulnerabilidades de segurança. No setor espacial, onde os requisitos de fiabilidade e conformidade são especialmente rigorosos, estas limitações tornam-se particularmente relevantes. Este trabalho insere-se num estudo pioneiro realizado em colaboração com a Critical Software, tendo como objetivo avaliar a viabilidade da adoção de Rust no contexto espacial. Para tal, foi analisada a arquitetura da plataforma Karvel, um on-board software modular desenvolvido em C, e foi implementado o seu módulo de gestão de memória em Rust, totalmente integrado com os restantes módulos escritos em C. Através desta abordagem, foi possível realizar uma análise comparativa entre as duas versões, considerando métricas como número de linhas de código, complexidade, desempenho e consumo de memória.
Os resultados obtidos permitem identificar vantagens e desafios na utilização de Rust em sistemas críticos, contribuindo para fundamentar futuras decisões de adoção tecnológica na indústria espacial e reforçando a investigação sobre o papel das linguagens de programação seguras neste setor.

Palavras-chave: Rust, Sistemas Críticos, Embedded Software, Sistemas Espaciais, Interoperabilidade C/Rust, Gestão de memória, Análise Comparativa.

ABSTRACT

The Rust programming language has gained increasing relevance in the development of critical systems due to its innovative approach to ensuring memory safety and efficiency. In contrast, traditional languages such as C and C++ remain widely used in this domain, despite their susceptibility to memory management errors and security vulnerabilities. In the space sector, where reliability and compliance requirements are particularly stringent, these limitations raise critical concerns. This dissertation is part of a pioneering study conducted in collaboration with Critical Software, aiming to assess the feasibility of adopting Rust in space applications. To this end, the architecture of Karvel, a modular on-board software platform developed in C, was analysed, and its memory management module was re-implemented in Rust and fully integrated with the remaining C modules. This approach enabled a comparative analysis between the two implementations, based on metrics such as lines of code, complexity, performance, and memory usage. The results provide insights into the advantages and challenges of applying Rust in critical systems, supporting future decisions on its adoption within the space industry and contributing to the ongoing research on memory-safe programming languages in this domain.

Index Terms: Rust, Critical Systems, Embedded Software, Space Systems, C/Rust Interoperability, Memory Management, Comparative Analysis.

DEDICATION

Dedico este trabalho à Luísa Maria Lopes e ao José Eduardo Lopes,
que me deram asas para voar.
Este trabalho é tão vosso quanto meu.

ACKNOWLEDGEMENT

Quero expressar a minha sincera gratidão à Critical Software pela oportunidade de realizar este estágio, que me proporcionou tantos conhecimentos e experiências valiosas.

Agradeço ao Mauro Gameiro, por ter me dado a oportunidade de integrar neste projeto e, tanto a ele como ao Gustavo Dinis pela revisão do relatório.

Um especial agradecimento ao Daniel Chichorro, o meu orientador que, com a sua boa disposição e dedicação, guiou-me e apoiou-me ao longo deste percurso. Onde tive o privilégio de aprender muito com ele.

Agradeço também à equipa de SpaceForce, onde tive o privilégio de conhecer pessoas incríveis que, para além de me acolherem bem mostraram-me que afinal não sei jogar matrecos tão bem como pensava.

Ao professor Francisco Pereira, agradeço o acompanhamento continuo e sugestões ao longo de todo o estágio.

Ao Instituto Superior de Engenharia de Coimbra, pela oportunidade de realizar este estágio numa empresa tão reconhecida.

A todos os professores da licenciatura e de mestrado, deixo um agradecimento especial por todo o conhecimento transmitido, que foi essencial para a concretização deste trabalho.

A nível pessoal quero agradecer aos meus pais, Luísa e José, pelo apoio incondicional e pela presença constante em todos os momentos.

À minha família, com especial agradecimento às minhas primas Viviana e Marta por estarem presentes e por serem um exemplo.

Aos meus amigos da universidade, em especial à Beatriz Duro e ao Pedro Praça, pelo companheirismo ao longos destes anos.

Ao meu amigo de infância Tiago e às minhas amigas da terrinha: Kah M., Kah C., Guida, Carol e Katy, uma grande obrigada por estarem presentes ao longo destes anos e por me fazerem tão feliz.

Por último, mas muito importante, quero deixar um enorme obrigada ao Rodrigo, que esteve ao meu lado todos os dias, apoiando-me, motivando-me e partilhando comigo esta experiência académica de forma tão especial.

INDEX

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface
Asw	Application Software
BSP	Board Support Package
CCN	Average Cyclomatic Complexity
CCSDS	Consultative Committee for Space Data Systems
CFDP	File Delivery Protocol
cFS	Core Flight System
CPU	Central Processing Unit
DLR	Deutsches Zentrum für Luft- und Raumfahrt
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
FFI	Foreign Function Interface
fws	Flight Software Code
GDB	GNU Debugger
HDSW	Hardware-Dependent Software
HTML	HyperText Markup Language
ID	Identifier
LEO	Low Earth Orbit
MISCRA-C	Motor Industry Software Reliability Association – C
NASA	National Aeronautics and Space Administration
NLOC	Number of Lines of Code
OBCP	On-Board Control Procedures
OBSW	On-Board Space Software

ONCD	Office of the National Cyber Director
OS	Operating Systems
PhD	Philosophiae Doctor
POSIX	Portable Operating System Interface
PUS	Packet Utilization Standard
RAM	Random Access Memory
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
SDD	Software Design Description
SRS	Software Requirements Specification
std	Standard Library
TC	Telecommand
TM	Telemetry
USA	United States of America

1 INTRODUCTION

The Rust programming language has gained significant industry adoption, particularly in critical systems, due to its innovative approach to ensuring safety and memory efficiency. The development of these systems faces growing challenges, especially with the increasing complexity of the functionalities to be developed, where traditional approaches are becoming less viable, requiring the search for new solutions to meet more stringent requirements, while always promoting the quality and security that critical systems demand [1]. Traditionally, the C and C++ languages have dominated the development of these systems due to their maturity and ability to interact directly with hardware. This choice is also justified by the fact that organizations in the space sector, including European Space Agency (ESA), often adopt coding standardization guidelines such as Motor Industry Software Reliability Association – C (MISRA-C). These guidelines, which are well known, aim to ensure security, reliability and compliance in space projects. However, these languages have suffered from memory safety issues, i.e. manual resource management leading to undefined behaviour or security vulnerabilities. Rust, a promising option launched by Mozilla in 2015, has demonstrated performance levels comparable to C and C++, but with compile-time security guarantees. Rust is notable for its unique memory management system, ownership, which removes the programmer's responsibility to explicitly manage resources, as well as the use of a garbage collector, which can cause additional performance and responsiveness issues in critical systems [2]. This evidence has become even more relevant and was highlighted in a 2022 White House report, where a United States of America (USA) government agency, the Office of the National Cyber Director (ONCD), decreed that programmers should use memory-safe languages such as Rust instead of C and C++ [3].

The exploration and use of space is undergoing a transformation, characterized by an increasing diversity of spacecraft, ranging from large, sophisticated systems such as space stations, constellations or space telescopes, to smaller, more accessible and flexible platforms such as CubeSats. These compact satellites, particularly prevalent in Low Earth Orbit (LEO), exemplify the trend towards miniaturization and democratization of space technology. With much higher affordability and shorter development cycles, the number of CubeSats has doubled in the last 4-5 years. Many spacecrafts rely on software developed in the C language, which is known for its performance and mature ecosystem, but not for its security features. This reliance raises critical concerns, especially as these systems play an increasingly important role in scientific research, communications and Earth observation. Despite the high profile of this language, there is still a need to investigate the viability of its use in critical systems [2]. In this context, an internship promoted by the master's degree in informatics engineering at Coimbra Institute of Engineering aims to explore the adoption of Rust in such systems, with a particular focus on the space sector.

1.1 Internship Objectives

This internship aims to investigate the application of the Rust programming language in the development of embedded software for spacecraft, with a focus on its use in critical systems. The aim is to conduct an in-depth analysis of the language's particularities, identifying its advantages, limitations and potential benefits compared to the C language.

It also includes an analysis of the architecture and design of Karvel, a software platform developed by Critical Software in C language. Karvel is an on-board space software (OBSW) platform designed to be modular, allowing it to be easily used in different projects. It enables faster software development and cost-effective solutions, eliminating the need to start a new project from scratch. The platform allows programmers to create applications without needing to know the underlying operating system and hardware, as it abstracts the hardware layer (Board Support Package (BSP), drivers) and interactions with the operating system (scheduling, memory management).

The main objective of the internship can be systematized into three steps:

1. **Integrate:** Before starting further development, the Rust module was integrated into the existing C system. This step ensured that the project could compile correctly, recognize the Rust module, and establish communication between Rust and C.
2. **Implementation:** After integration, the module's functionality was developed in Rust while maintaining interoperability with the C modules.
3. **Analyse:** Compare both implementations, in C and Rust, comparatively, based on metrics such as number of lines of code, complexity, performance, and memory usage, to draw conclusions about the feasibility and advantages of adopting Rust in this context.

1.2 Contributions

This internship contributes to my own capabilities through empirical knowledge and academic research. For Critical Software, it provides a practical evaluation of a module implemented in Rust, fully integrated with other modules written in C. This work will allow for a practical evaluation of the advantages and disadvantages of using Rust in critical systems, producing concrete results that may support future decisions about its adoption in the Karvel platform.

From a personal point of view, this internship represented a unique opportunity to participate in a research project applied to a real case, gaining professional experience in a leading company and deepen my skills in embedded systems programming. It also offered the opportunity to work with an emerging language such as Rust,

understand its integration with C, and explore the particularities of development for critical systems in the space sector, characterised by high standards and strict norms. In addition, it allowed me to apply tools and working methodologies used in a professional environment, including version control systems and structured development processes.

1.3 Work Methodology

To develop the proposed work, presented in Annex A, a 10-month schedule was defined, which can be seen in Figure 1, where the phasing of the various tasks is represented, as well as the expected individual duration of each one:

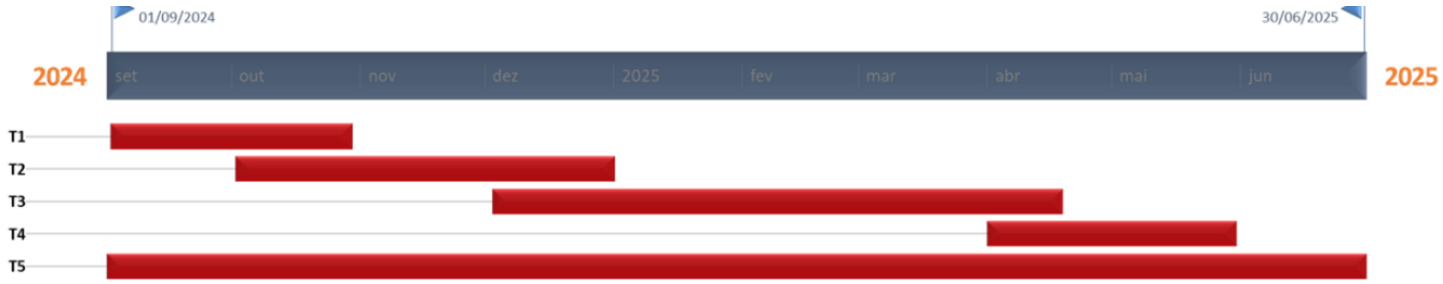


Figure 1 - Task schedule [Annex A]

The main activities to be developed within the scope of the project are:

T1: Analysis of the Rust programming language.

T2: Analysis of the architecture and design of Karvel.

T3: Development of Karvel functionalities in the Rust programming language.

T4: Comparison of the two implementations (C and Rust) of Karvel.

T5: Writing of the dissertation on the above topics, as well as the preparation of two accompanying presentations, one for academic assessment and the other, more technical, for internal presentation at Critical Software.

1.4 Critical Software

Founded in 1998, Critical Software, headquartered in Taveiro, Coimbra, was established by João Carreira, Gonçalo Quadros, and Diamantino Costa, who were at the time Philosophiae Doctor (PhD) students in Computer Engineering at the Faculty of Science and Technology of the University of Coimbra. The company specializes in the development of critical systems, that is, software

whose failure could have significant negative consequences, including safety risks, financial losses, or disruptions to essential services [4].

Currently, Critical Software employs around 1450 people. In addition to its headquarters in Coimbra, it has offices in Porto, Viseu, Lisbon, Southampton (United Kingdom), Munich (Germany), Boston (US), and Sunnyvale (US) [4].

The company operates in several sectors, including aviation, defence and security, automotive, energy, financial services, railway, medical devices, and space [4]. Within the company, this internship was developed within the space sector, specifically in the SpaceForce team.

1.5 Document structure

This document is structured to provide a clear and progressive understanding of the integration of the Rust programming language into the Karvel platform.

Chapter 2 – Rust for Space explores the Rust language, covering concepts such as ownership, references, borrowing, and lifetimes, which allow Rust to ensure memory safety without relying on a garbage collector, while avoiding the need for manual memory management as in C. The chapter also includes several case studies from organizations such as ESA, NASA, and N7 Space, illustrating the growing adoption of Rust in space applications.

Chapter 3 – Karvel describes the Critical's on-board software platform called Karvel, its architecture, build system, configurator and modular structure. This chapter provides the necessary foundation to understand how a Rust module can be integrated into Karvel.

Chapter 4 – Integration of Rust Code into Karvel details the process of integrating Rust and C code. It covers tools such as CMake, Corrosion, Bindgen, cBindgen, CXX, and Libloading, as well as the specific implementation of the memory manager module in Rust. The chapter also discusses challenges encountered throughout the development process.

Chapter 5 – Verification focuses on the verification and validation of the developed work. It emphasizes unit testing, including the mock-based approach, coverage analysis with grcov and lcov, identification of errors in the original C codebase, and the cross-compiling process.

Chapter 6 – Memory Manager Comparison: C and Rust Implementations present a comparative analysis between the C and Rust implementations of the memory manager. It includes static analysis results, complexity and line count metrics, profiling with Renode, and a comparison of total execution time.

Chapter 7 – Conclusion summarizes the results, challenges, and lessons learned throughout the internship. It reflects on the role of Rust in critical systems and within the Karvel platform and identifies potential directions for future work.

2 RUST FOR SPACE

In this chapter, and as a starting point for the internship, a study of the Rust programming language was conducted, always using the C language as a reference. In a second phase, related articles were reviewed and analysed, focusing on how Rust has been studied and applied, particularly in the space sector.

Rust has emerged as a promising language for the developing of critical systems in space environments, which have traditionally been dominated by C. While C is efficient and widely used, its lack of memory safety guarantees makes it vulnerable to critical failures that could compromise the reliability of space missions. In this context, Rust presents itself as a robust alternative, eliminating, at compile time, common C errors such as buffer overflows and null pointer dereference [1] [2].

Its innovative memory management approach, based on the ownership model and other language features, ensures greater safety and predictability during execution [2]. A comparative study between Rust and C, conducted as part of the internship, is included in Annex B. This study highlights the differences between the two languages.

2.1 The importance of Rust in Space

In this section, the memory safety system, lifetimes, and borrow checkers offered by Rust will therefore be detailed, given that this represents one of the main limitations recognized in C [1]. This limitation manifests both in conventional systems and in critical space systems, where it deserves special attention due to limited memory resources and the severe constraints on correcting failures post-launch, which are costly and risky. In this context, Rust can represent a significant solution for improving safety and reliability [2].

To understand how Rust improves safety in memory management, two fundamental concepts must first be clarified, since they will be extensively used throughout this section: stack and heap.

- Stack: The stack is a memory region used to store function calls and local variables. Its operation is automatic, being managed by the system in a fast and efficient way. Memory is allocated in contiguous blocks within the call stack, and its size is known before execution. When a function is called, local variables are allocated, and they are released as soon as the function ends, without requiring the programmer to handle allocation or deallocation manually. However, the stack has a limited size and is not suitable for storing large data structures or data that must remain valid after a function has finished executing [5].

- Heap: The heap is the area of memory used for dynamic allocation, where variables can be created at runtime and persist independently of the function that allocated them. This flexibility comes at the cost of slower access and more complex management. However, industry standards typically restrict or even forbid dynamic memory allocation in the context of space [5].

Ownership: Memory management

The heap provides ample space to store data during program execution, but this space is not infinite. Therefore, it is essential to ensure that dynamically allocated memory is freed when it is no longer needed. There are three approaches to memory management: explicit allocation, garbage collection, and ownership.

Manual Memory Management in C

In explicit allocation, as in C, the programmer is directly responsible for managing memory. Functions such as malloc (to allocate) and free (to deallocate) are used to manually control the lifecycle of memory blocks. This approach gives the programmer a high degree of control over how memory is used, which can be advantageous. However, this freedom also introduces significant risks. It is easy to make mistakes such as invalid memory access, memory leaks, or double freeing of blocks, which can compromise the program's stability and safety [6].

A simple example is sowed in the following code:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {

    int *number = malloc(sizeof (int));

    if (number == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    *number = 42;
    printf("Number value: %d\n", *number);

    free(number);

    return 0;
}
```

Automatic Memory Management

In contrast, languages that use garbage collection, such as Java or Python, automate memory management. Although this simplifies development, it introduces some unpredictability in performance, since the garbage collector decides when and how memory is freed. This unpredictability is not ideal for critical systems, as the garbage collector might run at moments when performance is crucial [6].

Memory Management in Rust: Ownership

Rust offers a unique approach compared to other languages, based on the concept of ownership, without relying on a garbage collector or requiring manual memory deallocation. Variables are responsible for freeing their own resources according to a set of rules [7].

The first rule of Rust's ownership system states that every value has one and only one owner at a time. This means that when a value is assigned to another variable, ownership is moved, and the original variable can no longer be used [7].

```
1. fn main(){
2.    let my_string = String::from(“Rust”);
3.    let other_string = my_string;
4.    println!("{}", my_string);    // An error in this line
5. }
```

This code fails with the following compiler error:

```
 1. error[E0382]: borrow of moved value: `my_string`
 2.  --> src/main.rs:4:16
 3.   |
 4. 2 | let my_string = String::from("Rust");
 5.   |     --------- move occurs because `my_string` has type `String`, which does not
implement the `Copy` trait
 6. 3 | let other_string = my_string;
 7.   |                    --------- value moved here
 8. 4 | println!("{}", my_string);
 9.   |                ^^^^^^^^^ value borrowed here after move
10.   |
11.   = note: this error originates in the macro `$crate::format_args_nl` which comes from the
expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
12. help: consider cloning the value if the performance cost is acceptable
13.   |
14. 3 | let other_string = my_string.clone();
15.   |                             ++++++++
```

In this code, the string “Rust” is initially created and owned by the variable *my_string*. However, when it is assigned to *other_string*, ownership of the heap-allocated data is moved to the new variable. As a result, *my_string* becomes invalid, and trying to use it afterward and will cause a compile-time error. This behaviour is visualized in the Figure 2, which illustrates what would break Rust's first ownership rule [7].

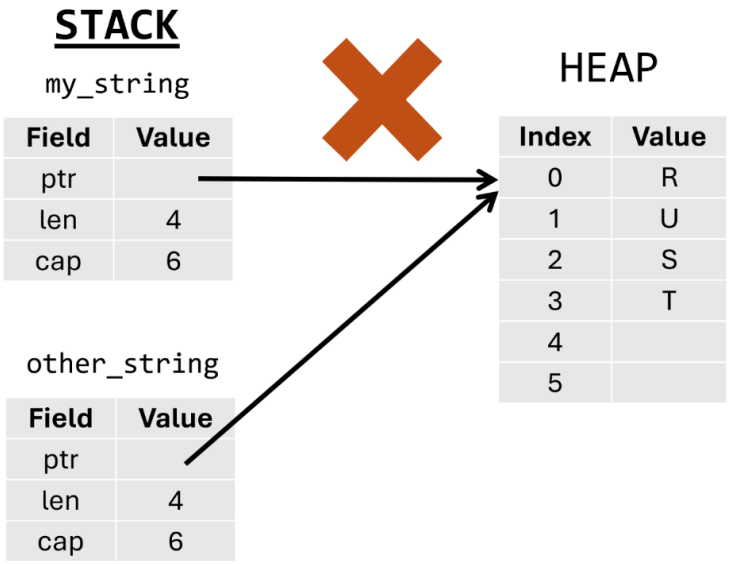


Figure 2 - Behaviour that does not comply with Rust's first ownership rule

In this case, Rust's ownership rule is not respected. This issue can easily be fixed by using the *clone()* method, which creates a copy of the string, allowing both variables to own its data independently [7]. For example:

```
1. fn main(){
2.    let my_string = String::from("Rust");
3.    let other_string = my_string.clone();
4.    println!("{}", my_string);
5. }
```

This way, both *my_string* and *other_string* have their own separate copies of the string "Rust" [7]. The memory layout is illustrated in Figure 3.

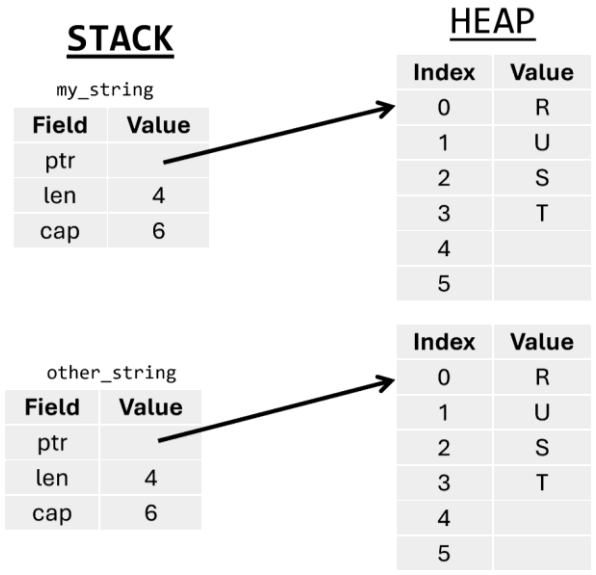


Figure 3 - Memory layout of *my_string* and *other_string*

At first glance, the following code may appear to violate Rust's ownership rules, yet it compiles and executes without any errors:

```
1. fn main() {
2.   let x : i32 = 1;
3.   let y = x;
4.   println!("Value of x: {}", x);
5. }
```

This occurs because *i32* implements the copy trait, meaning its value is duplicated rather than moved during assignment. Consequently, *x* remains valid after being assigned to *y* [7]. Figure 4 illustrates the allocation on the stack and the absence of heap usage:

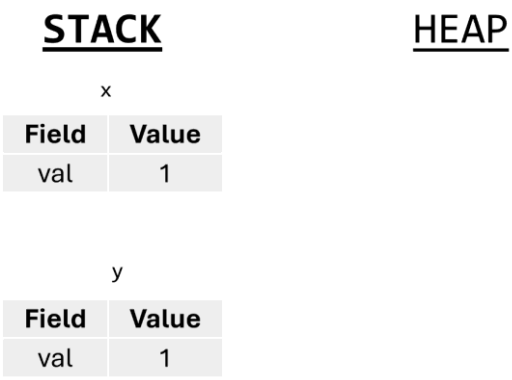


Figure 4 - Stack allocation of *x* and *y*

The second rule is about when the owner variables go out of scope, the value is dropped. For this rule the following code explains this very well [7]:

```
1. fn main(){
2.   let my_string = String::from("Rust");
3.   rust_function(my_string);
4.   println!("My String = {}", my_string);
5. }
6.
7. fn rust_function(input: String){
8.    println!("My String = {}", input)
9. }
```

This code fails to compile because *my_string* is moved to *rust_function*, transferring its ownership. As a result, *my_string* becomes invalid after the move, and any further use causes a compiler error [7]:

```
1. 1.  error[E0382]: borrow of moved value: `my_string`
2.   --> src/main.rs:4:31
3.    |
4. 2 |    let my_string = String::from("Rust");
5.    |        --------- move occurs because `my_string` has type `String`, which does not
implement the `Copy` trait
6. 3 |    rust_function(my_string);
7.    |                  --------- value moved here
8. 4 |        println!("My String = {}", my_string);
9.    |                                   ^^^^^^^^^ value borrowed here after move
10.   |
11. note: consider changing this parameter type in function `rust_function` to borrow instead if
owning the value isn't necessary
12.   --> src/main.rs:7:26
13.    |
14. 7 |     fn rust_function(input: String){
15.    |        -------------        ^^^^^^ this parameter takes ownership of the value
16.    |        |
17.    |        in this function
18.    = note: this error originates in the macro `$crate::format_args_nl` which comes from the
expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
19. help: consider cloning the value if the performance cost is acceptable
20.    |
21. 3 |    rust_function(my_string.clone());
22.    |                           ++++++++
```

In the main function, the memory layout is initially structured in a specific way, with the *my_string* variable allocated on the stack and its contents on the heap, as shown in Figure 5.

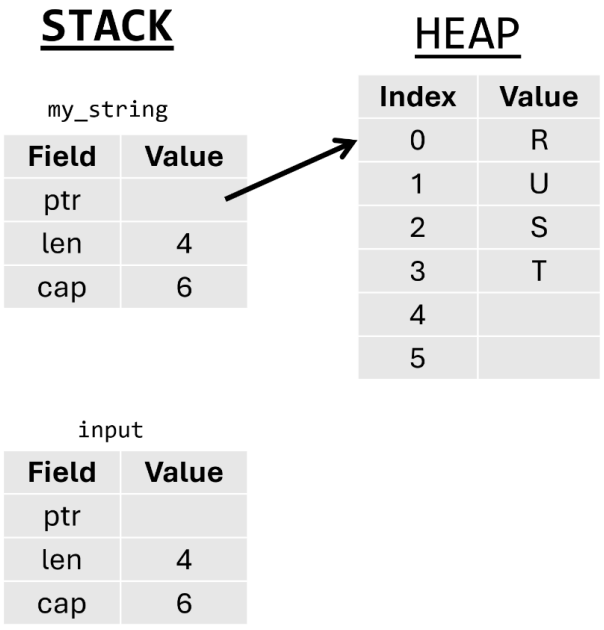


Figure 5 - Memory layout of *my_string* on stack and heap

Once the variable is passed to the function, ownership is transferred, and the memory layout changes accordingly, as illustrated in Figure 6.

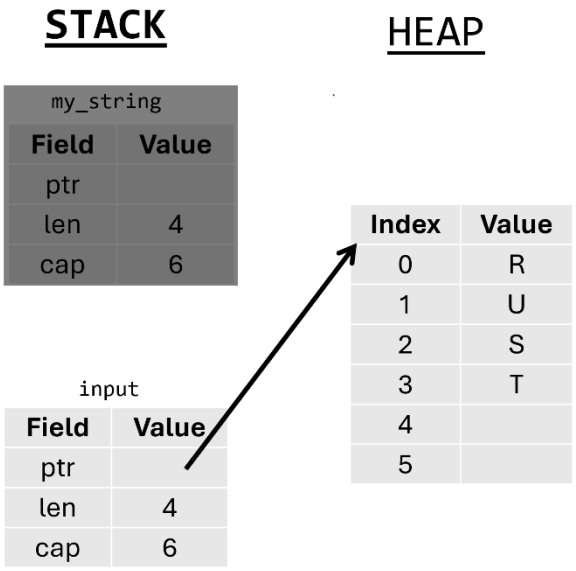


Figure 6 - Memory layout after transferring ownership of *my_string*

The *rust_function* now owns the variable. When the function scope ends, the variable is dropped, deallocating the heap memory. As a result, the main function no longer has access to *my_string*.

To solve this problem the following code works

```
1. fn main(){
        let my_string = String::from("Rust");
2.      let my_string = rust_function(my_string);
3.      println!("My String = {}", my_string);
4. }
5. fn rust_function(input: String) -> String {
6.      println!("My String = {}", input)
7.      input
8. }
```

This approach gives safe and efficient memory management. But have the disadvantages of requiring understanding of ownership.

2.1.1 References and Borrowing: Mechanisms for Safe Memory Access in Rust

However, transferring ownership of a value, such as when passing it to a function, is not always the intended behaviour. In such cases, it is possible to borrow the value instead, which allows access to the data without transferring ownership. This mechanism is implemented through references, which form a key part of Rust's memory management model [7].

```rust
fn main(){
  let my_string = String::from("Rust");
  rust_function(&my_string);
  println!("My String = {}", my_string);
}

fn rust_function(input: &String) -> usize {
  println!("My String = {}", input)
  let len = input.len();
  Len
}

```

Figure 7 provides an illustration of this concept.

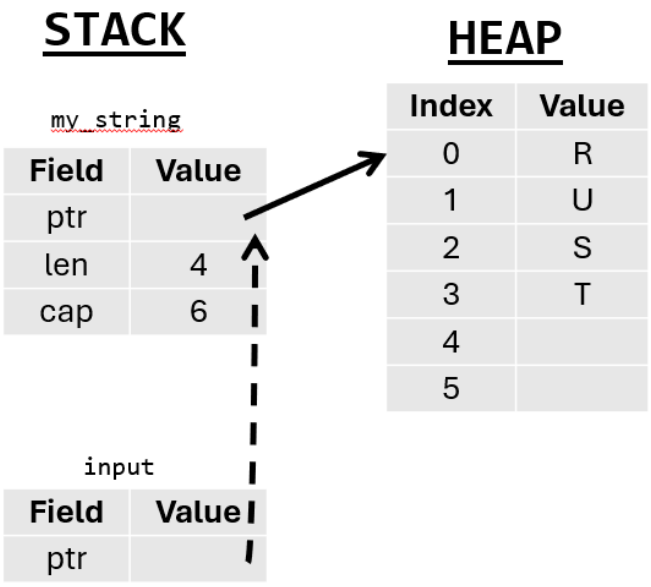


Figure 7 - Representation of Rust's borrowing mechanism, where references allow access to data without transferring ownership.

Rust imposes a strict set of rules regarding the use of references, which are essential to guarantee memory safety at compile time [7]. In particular:

1. At any given time, a value can have either:
    - One mutable reference *(&mut T)*, or
    - Any number of immutable references *(&T)*, but not both simultaneously.

The Figure 8 provides a visual summary of the rule.

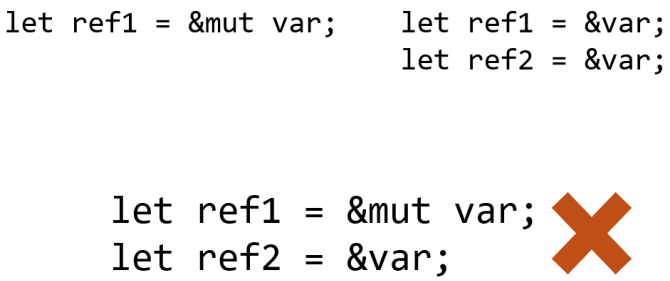


Figure 8 - A visual representation of the rules for using references in Rust, showing exclusive mutable access versus shared immutable access.

2. References must always be valid. That is, they must never point to deallocated or invalid memories. This eliminates the risk of dangling references.

2.1.2 Lifetimes: Ensuring Safe Memory Access in Rust

In Rust, lifetimes are a compile-time mechanism that ensure references are always valid while in use. By explicitly tracking the scope of each reference, the compiler can prevent issues such as dangling references, where memory is accessed after being deallocated. This validation is performed by the borrow checker, which uses *lifetime annotations* (e.g., *'a*, *'b*, *'c*) [8].

A valid example of lifetime annotation is presented in the following Figure 9.

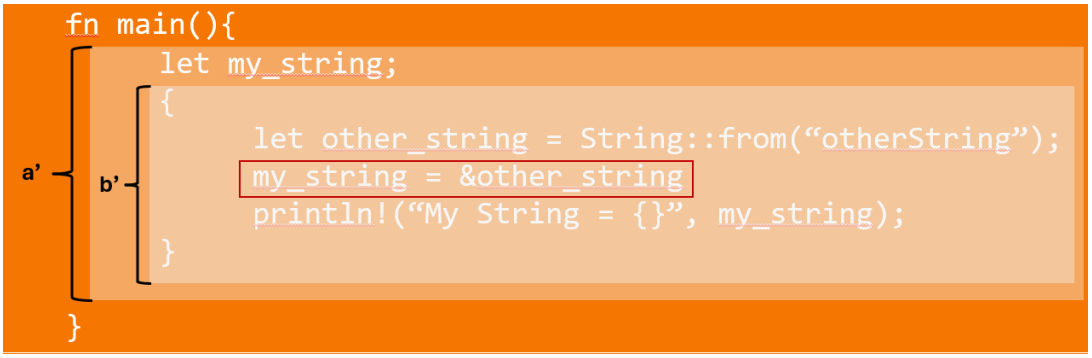


Figure 9- Illustration of a function with explicit lifetime annotations.

Lifetime annotations are not always required in Rust. They are only necessary when the compiler is unable to infer the lifetimes of the variables automatically. In such cases, Rust applies a set of inference rules, which allow the compiler to deduce lifetimes in function signatures without explicit annotations. These rules are as follows [8]:

1. Each input parameter that is a reference is assigned its own lifetime.

```
fn this_is_an_example <‘a,’b,’c>(x: &’a str, y: &’b str, z: &’c str) -> &str
{
    //…
}
```

2. If there is exactly one input lifetime, assign it to all output lifetimes

```
fn this_is_an_example<‘a>(x: &’a str) -> &’a str
{
  //…
}

fn this_is_an_example<‘a,’b>(x: &’a str, y: &’b str) -> &?<str
{
  //…
}
```

3. If there is a *&self* or *&mut* self-input parameter, its lifetime will be assigned to all output lifetime

```
fn this_is_an_example<‘a,’b>(x: &’a self, y: &’b str) -> &’a str
{
    //…
}
```

Even after applying the lifetime rules, there are cases where the Rust compiler cannot determine the correct lifetime of references [8]. The following example illustrates this:

```
fn main(){
  let result;
  let my_string = String::from(“Hello Rust!”);
  let other_string =  String::from(“Hello World!”);
  result = check_rust(&my_string, &other_string);
  println!(“My String = {}”, result);
}

fn check_rust(x: &str, y:&str) -> &str {
      if x.len() > y.len(){
                x
      } else{
                y
      }
}
```

In this case, the function *check_rust* returns a reference, but the compiler is unable to infer whether the returned reference comes from *x* or *y*, since both are independent input references and none of the lifetime rules apply [8].

As a result, the following compilation error is produced:

```
Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:37
  |
9 |   fn check_rust(x: &str, y:&str) -> &str {
  |                    ----    ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not
say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 |   fn check_rust<'a>(x: &'a str, y:&'a str) -> &'a str {
  |                ++++     ++         ++          ++
```

To resolve this error, it is necessary to add explicit lifetime annotations that indicate that both input references and the return value share the same lifetime [8]:

```
fn main(){
  let result;
  let my_string = String::from("Hello Rust!");
  let other_string =  String::from("Hello World!");
  result = check_rust(&my_string, &other_string);
  println!("My String = {}", result);
}

fn check_rust <'a> (x: &'a str, y:&'a str) -> &'a str
{
   if x.len() > y.len(){
     x
   } else{
     y
   }
}
```

By adding the lifetime parameter *'a*, the function signature now clearly indicates that the returned reference will be valid if both *x* and *y* are valid for the lifetime *'a*, satisfying the borrow checker's requirements [8].

2.2 Overview of Rust for Space

The Rust programming language has established itself by offering robust security and performance guarantees, attracting considerable interest from various industries, including space. In this context, organizations such as the ESA, N7 Space and National Aeronautics and Space Administration (NASA) have investigated the potential of Rust to fulfil the rigorous demands of safety-critical space systems.

2.2.1 Case Study I: ESA Project

The *ESA* project "Crustacea in Space - Co-Operative Rust and C Embedded", carried out by *Deutsches Zentrum für Luft- und Raumfahrt (DLR)* - German Aerospace Centre between 2022 and 2024, investigated the viability of Rust for the development of embedded software in space missions. The focus was on evaluating Rust's integration into critical systems, particularly those using the Real-Time Executive for Multiprocessor Systems (RTEMS), a widely used open-source real-

time operating system (RTOS). This evaluation is increasingly relevant as new space missions demand more complex functionality, making traditional approaches less viable. Rust offers a potential solution to meet these stringent requirements while maintaining high standards of quality and safety. Previous studies have investigated the viability and performance of Rust for implementing space-related applications. While Rust has performed well on various hardware architectures and operating systems (OS), several barriers need to be overcome before adoption for space missions, such as [1]:

- Rust needs to compile on common OS and processor architectures used in space missions.

- Rust needs to interact with existing C code to enable a gradual transition in development.

- Rust must comply with existing product quality and assurance standards, such as European Cooperation for Space Standardization (ECSS).

The study developed three ECSS Packet Utilization Services (Function Management, Parameter Management, and Housekeeping) in both Rust and C. These were compared with a focus on differences in memory management, code readability, and writability. The ownership system stood out for bringing significant improvements where it demonstrated greater robustness in avoiding problems such as buffer overflows. The results indicated that Rust not only met the *ESA's* strict standards, with a focus on the *ECSS-E-ST-80C* but also offered reliability and safety advantages for space missions. Rust has thus been identified as a promising solution, with comparable performance to C, but with compile-time safety guarantees [1].

2.2.2 Case Study II: NASA Project

NASA also had initiatives between 2020 and 2021 with the study "Rust in *cFS*: Prevent Bugs with Memory-Safe Programming". The study focused on using Rust within the Core Flight System (cFS), assessing its impact on safety and reliability in flight-critical systems, and feasibility of integrating Rust with cFS [9].

The aim of the study was to make flight software more reliable and affordable, allowing groups with limited budgets to reduce the negative impact of errors, such as loss of scientific data or even loss of the satellite [9].

The results showed that integration with Rust brought benefits, making flight controllers more reliable and less expensive. This was due to compile-time checks that eliminated early common errors in C, such as null pointers, double free and use after free errors, contributing to better software safety and robustness [9].

2.2.3 Case Study III: *N7 Space* Project

The study "Evaluation of Rust usage in space applications" [5], carried out by *N7 Space* between 2023 and 2024, aims to boost the study of the language by developing

a *RTOS* in Rust targeting *ARM Cortex-M7 SAMV71* microcontroller, and a *BSP*. The *RTOS* will be made available as open source and developed for *CubeSat* missions. The system is based on simplicity, portability, low memory consumption and the use of mature Rust features. To obtain these features, the scheduler will execute *tasklets* instead of traditional threads. The *tasklets* can be used for share stack, avoid context switches and provide constrained concurrency patterns that are more predictable than traditional threads [10].

The implemented *RTOS* includes essential functionalities such as execution priorities, communication by queues and events, and analysis of execution times to optimize performance in real time. In addition to evaluating the viability of Rust for critical systems, the operating system also aims to evaluate the viability with *ESA* standards [10].

The results of the evaluation focus on the strengths of memory security, compatibility with *ESA* standards, high-performance capabilities, and the documentation available with great support from the community. The weaknesses identified were the learning curve, building times, some tools and libraries aren't as mature as traditional, and the stability of language features [10].

2.2.4 Case Study IV: Bringing Rust to Safety-Critical Systems in Space

The study "Bringing Rust to Safety-Critical Systems in Space" [1] also identified problems with memory security and manual resource management in C that lead to undefined behaviour and security vulnerabilities, which are increasing as the sector evolves, mainly due to the increase in the number of smaller and cheaper satellites (*CubeSats*), and with more frequent launches, creating the need for more robust, secure and reliable software [2].

This study identifies that in addition to space systems, Rust has been highlighted in large projects such as Android, where memory security vulnerabilities decreased from 76% to 35% between 2019 and 2022. These benefits could also bring benefits to the space sector, even though theoretical studies have been done, Rust has not yet been adopted widely in the industry due to concerns about the maturity of the tools and formal specifications. This work also noted that, while Rust has security and reliability advantages, its use is still limited in real-world contexts due to the need to integrate it with existing systems, as it would be impractical to completely replace the C code [2].

The study presents recommendations for the development of safety-critical space systems. Present an overview of ongoing efforts to adapt Rust to safety-critical systems and how Rust provides robustness. Introduce a procedure for partially rewriting C-based systems in Rust. Rewrite a case study of an open source satellite communication protocol. Three undiscovered vulnerabilities were identified and fixed [2].

2.2.5 Final Summary

All the above studies identified the potential of the Rust programming language to enhance the safety of critical systems and serve as a viable alternative for space industry development. Table 1 summarizes the main objectives, benefits, and challenges identified in each of the case studies. However, the studies also emphasize that further research is required, and that the language still presents challenges due to its immaturity [1] [2] [9] [10].

Table 1 - Overview of Rust Case Studies in Space Applications

Project	Objective	Benefits	Challenges
ESA	Evaluate Rust for embedded software in space missions	It met ECSS standards, offering greater memory management robustness and being comparable to C with compile-time safety.	There is a need for compatibility with common architectures and operating systems, and for gradual integration with existing C code.
NASA	Make flight software more reliable and cost-effective	Reduced common memory errors for safer, more affordable software.	Gradual integration with existing systems and a learning curve.
N7 Space	Develop an RTOS in Rust for CubeSats	Memory safety, compliance with ESA standards, high performance and good documentation.	Learning curve, build times, maturity of tools and libraries, and resource stability.
Bringing Rust to Safety-Critical Systems in Space	Highlight C security issues and Rust benefits	Vulnerabilities identified and fixed. Recommendations for partial migration.	Adoption is limited due to the immaturity of the tool and the need for integration with existing systems.

3 KARVEL

Within the scope of this internship and considering the advantages previously discussed regarding the adoption of Rust in space systems, it is relevant to analyse its application in real-world contexts. This will be carried out using the Karvel platform, developed by Critical Software. Karvel is an OBSW platform, whose name is inspired by the Portuguese maritime explorations of the 15th and 16th centuries enabled by the caravel. In the same way, Karvel is designed as a mission-critical software platform that supports and enables space exploration.

The advantage of Karvel is to provide a platform with a reusable core that can be emplaced in different projects. Using Karvel it is possible to develop mission software without starting from scratch that makes the development three times faster, and four times less expensive.

The platform allows developers to create applications (modules) without requiring deep knowledge of underlying OS and hardware, as it abstracts the hardware layer (BSP, drivers) and the interactions with the OS (scheduling, memory management).

Karvel is developed using the language C and functionally tested using the C++ language. It has the capability to run in different OS, including RTOS such as RTEMS (version 4 to 6), VxWorks (version 6 to 7), FreeRTOS, as well as Linux.

As a truly multi-platform framework, Karvel supports compilation for various architectures, including SPARC (LEON3 and LEON4), PowerPC (P2020), RISC-V (Generic and NOEL-V), ARM Cortex-A72 and A9 (e.g., Xilinx Zynq), NVIDIA (Jetson Xavier NX) and Beyond Gravity's cOBC.

It also supports a wide range of hardware interfaces, such as MIL-STD-1553, SpaceWire, RS-422, RS-485, and CAN. In terms of communication protocols, Karvel is compatible with CCSDS, ECSS PUS (A&C), CubeSat Space Protocol, and CFDP.

Thanks to this versatility, Karvel can be adapted to meet the specific requirements of different platforms and mission profiles.

3.1 Architecture

Karvel's architecture is composed of three layers, ordered from lower to higher abstraction: the low-level layer, the middleware layer and application layer. Each of these layers contains mission specific components, reusable modules, hardware specific and third-party.

The low-level layer only contains the bootloader and is the first thing to run on the computer. Its role is to load and start application modules, which are designed to

run as threads or tasks, and prepare modules for use as libraries. Some missions need a more specific functionality that doesn't work with simple bootloaders so in those cases the bootloader starts the boot software that contains these specific requirements. Figure 10 represents the bootloader specific.

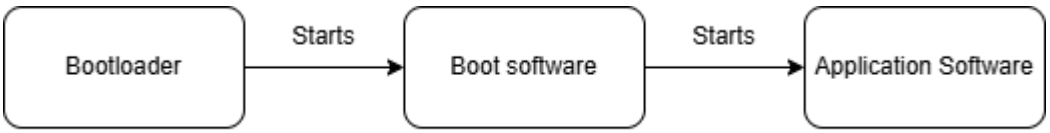


Figure 10 - Bootloader with specific Boot Software

Figure 11 represents the bootloader without any specific requirements.

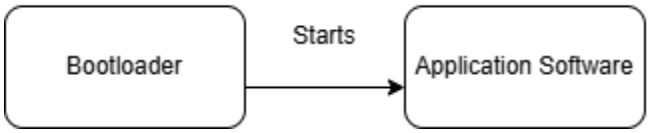


Figure 11 - Bootloader without specific Boot Software

The middleware, as the name suggests, acts as an intermediary between the application and the hardware. It serves as an interface that receives an intention from the application and executes it using the OS's available methods.

The middleware layer is composed of the components:

- The BSP makes it possible for the OBSW to work in a specific hardware environment, integrated with the embedded OS.
- Hardware-dependent Software (HDSW) provides the application software (Asw) with an interface to easily access those features which are directly supported by hardware.
- IO Drivers is a software driver for specific hardware interfaces.
- RTOS.
- RTOS Wrapper abstracts the application layer software components from the RTOS enhance the functionality and manageability.

The layers in the application layer are composed of reused components, such as:

- File Delivery Protocol (CFDP) implements the Consultative Committee for Space Data Systems (CCSDS) CFDP compliant with CCSDS 727.0-B-5.

- Data Pool ensures consistent data, integrity, and accessibility across different modules, being essential for reliable data handling and real-time data updates.
- On-Board Control Procedures (OBCPs) implementation complies with ECSS-E-ST-70-01C.
- Packet Utilization Standard (PUS) Library, that implements the PUS services that complements the CCSDS Space Packet Protocol (CCSDS 133.0-B-1) and comply with ECSS-E-ST-50 and ECSS-E-ST-70-41 (versions A and C) standards, by defining the application-level interface between ground and space.
- Scheduler Manager.
- File System: Data block-based file system.

And by mission specific like:

- Initialization: Initializes the asw.
- Thermal Control: Equipment temperature control.
- Time Manager.
- FDIR: Failure, Detection, Isolation and Recovery management activities.
- Power Manager: Equipment's power manager.
- Mode Manager: Manages internal modes and performs actions during mode transitions.
- GNC SW: Guidance, Navigation and Control functionality of Spacecraft.
- Comms Manager: Communications management activities.
- Equipment Manager: Instruments, equipment and sub-systems management.

This layer simplifies tailoring for specific use cases and mission requirements. Developers can easily choose which modules to include or exclude during the build process, a mechanism that will be detailed in the next chapter.

Karvel was designed to be reusable between different space missions, so its architecture is modular. The division of all modules is based on the initial approach adopted during the ClearSpace-1[1] mission [11]. Figure 12 represents the Karvel architecture.

[1] The ClearSpace-1 mission, led by ESA, will be the first active debris removal project: it aims to reach an uncontrolled object in space, capture it, and guide it to a controlled re-entry into Earth's atmosphere.

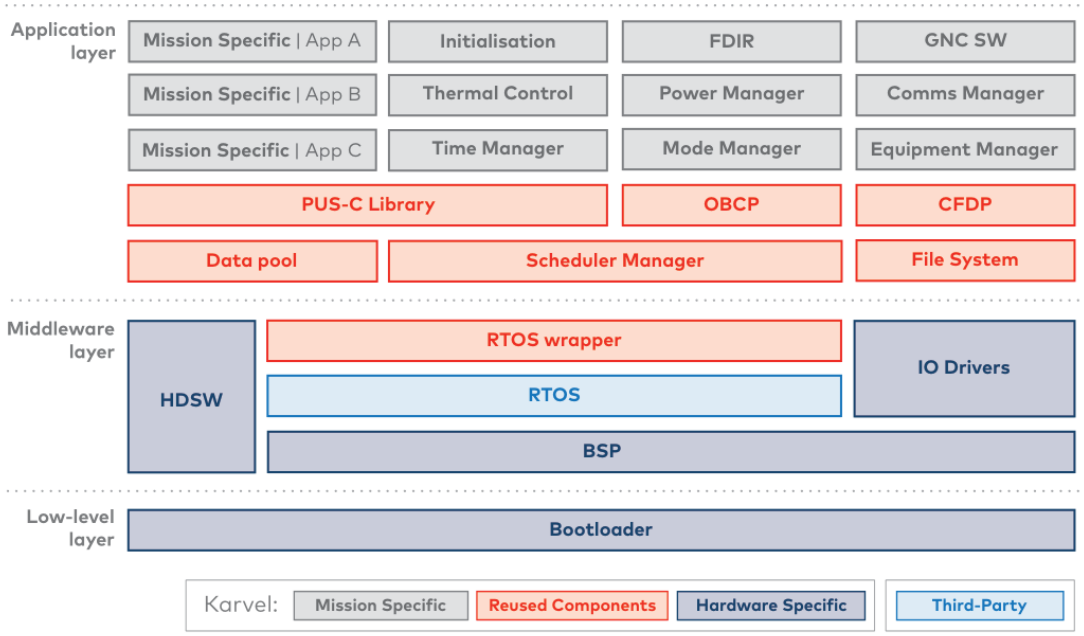


Figure 12- Karvel Architecture

3.2 Build System and Configurator

Karvel uses CMake as its build system. It allows modular configurations and gives the capability to change settings and create targets without having to change the software itself.

CMake is an open source, cross-platform tool used to automate the software build process. It generates platform-specific build files [12]. CMake does this by reading configuration files called *CMakeLists.txt* and producing the appropriate build scripts for the chosen compiler and platform.

In the configurator, it is possible to find the starting point for the system compilation in the CMakeList.

The build system is designed to find everything in the project and will try to find all folders with a CMakeList and automatically add them to the build process.

3.3 Modules

A module in Karvel is composed of the following components that Figure 13 shows.

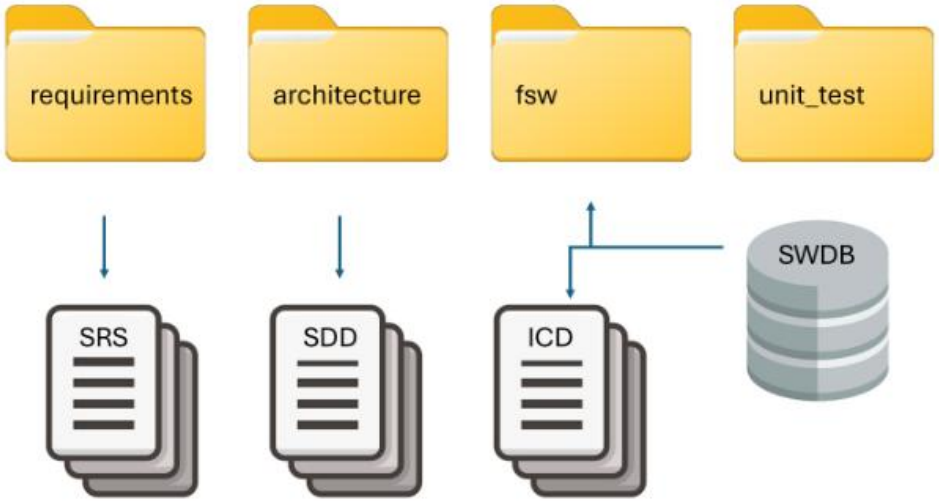


Figure 13- Karvel's module structure

The first is requirements, which contain the software requirements that are part of the Software Requirements Specification (SRS) for the given component.

The second is architecture, which is part of the Software Design Description (SDD), where is described the static and dynamic software architecture usually contains diagrams.

The third is the Flight Software Code (fws), which includes the main source code of the module. It is divided into a public folder, which contains the header files, and a private folder, where the implementation files are located.

The fourth is unit tests that contain the unit tests for verifying the correct implementation of the code and getting test coverage. And follow the rules that the unit test should have at least 100% statement coverage and 100% branch coverage.

And finally, CMakeLists is a file to manage and build all the dependencies and subdirectories of the module.

After gaining a global overview of the Karvel platform and becoming familiar with the Rust language, it becomes possible to advance to the investigation of its integration. This process seeks to understand how a Rust module can be incorporated into the platform, and which mechanisms enable its communication with the other modules in C.

4 INTEGRATION OF RUST CODE INTO KARVEL

After analysing the Karvel platform and understanding its internal organization, developed in C, it becomes necessary, before the code implementation, to define and formalize how Rust can be integrated and how both languages can communicate. This chapter provides a detailed description of the process of integrating Rust code into the Karvel platform, making use of Corrosion and CMake to ensure compatibility with the existing build system. It also analyses different techniques for enabling interoperability between Rust and C, including manual bindings, automated tools such as Bindgen and cBindgen, as well as the use of the cxx crate.

4.1 CMake and Corrosion

The way in which Rust could be integrated into Karvel was explored. A significant example is Android 13, where some of the new functionalities began to be implemented in Rust, resulting in a reduction of security vulnerabilities from 75% to 35% [13]. However, this approach goes beyond the scope of this study, which focuses specifically on improving memory safety in the Karvel platform.

Another alternative could involve rewriting the entire Karvel platform directly in Rust [14]. However, this option proves to be unfeasible, as Karvel is a mature and complex platform, making a full conversion too time-consuming and costly.

The other approach that was studied, and which is the most appropriate strategy, is the gradual adoption of Rust, starting with the conversion of specific modules [14].

As previously mentioned, Karvel uses CMake as its build system. However, to support the integration of modules written in Rust, the process must be adapted. For this purpose, Corrosion is used, which is a tool that acts as a bridge between CMake and Rust's Cargo build system [15].

Corrosion allows CMake to automatically recognize and build Rust dependencies (importing executables, static and dynamic libraries). In this way, it becomes possible to integrate Rust modules into an existing C project without the need to restructure the entire build system [15].

Internally, Corrosion uses the *Cargo.toml* file as the entry point for the Rust project. Through this file, it identifies the project structure, required dependencies, libraries to generate, and compilation methods [15].

The following example shows an excerpt from a *Cargo.toml* file used in this context:

```
[package]
name = "example"
version = "0.1.0"
edition = "2021"

[dependencies]

[lib]
crate-type=["staticlib"]
```

To use Corrosion in projects that use CMake version 3.19 or higher, as is the case of Karvel, which uses version 3.22.1, the *FetchContent* method can be used to automatically integrate Corrosion [15].

This mechanism downloads Corrosion during the project configuration phase and allows it to be used as if it were included locally as a subdirectory, without the need to clone it manually [15].

A block of code was added to the Rust module's *CMakeLists.txt*. This was done to do this:

```
include(FetchContent)

FetchContent_Declare(
    corrosion
    GIT_REPOSITORY https://github.com/corrosion-rs/corrosion.git
    GIT_TAG v0.5
)

FetchContent_MakeAvailable(corrosion)
```

Source: From Corrosion documentation

4.2 Approaches to the Integration of C and Rust Modules

Since a gradual approach compatible with CMake was chosen, enabling communication between Rust and C became necessary. To identify the most suitable integration method, a study was conducted considering several approaches: manual integration, Bindgen and cBindgen for automatic generation of bindings, cxx for safer interoperability, and libloading for dynamic linking. Each method was analysed in terms of safety, ease of use, and compatibility with the existing build system.

4.2.1 Manual Integration

The first method was manual, where Rust provides a Foreign Function Interface (FFI). The FFI is a mechanism that allows a language to call or be called by functions from other languages, which facilitate integration. For Rust code to be called from C, it is necessary to define a "foreign function" in Rust, using the keyword extern and the attribute *#[no_mangle]*. This latter attribute instructs the compiler not to

change the function name during the compilation process, ensuring the name stays readable and usable from the C side [14].

By default, the Rust compiler applies name mangling, a process that modifies function names to include additional information, to avoid collisions during linking across multiple crates [16]. However, this behaviour is not desired in cross-language interoperability scenarios, as it would render the function names unreadable or inaccessible externally. To prevent this, *#[no_mangle]* is used, which preserves the exact function names, making them usable from C code [14].

In the *lib.rs* file, an example is shown in the following code.

```
1. #[no_mangle]
2. pub extern "C" fn add(a: i32, b: i32) -> i32 {
3.     a + b
4. }
```

Afterwards, it is necessary to compile the Rust code as a shared or static library. This requires specifying the type of library to be generated, either a *.so* (shared) or *.a* (static) file, in the *Cargo.toml* file. Rust supports different types of libraries for linking [17].

The following code shows how to configure *Cargo.toml* to generate a static library:

```
1. [lib]
2. crate-type=["staticlib"]
```

This designation is commonly used in embedded systems. It means that a static library is created, which can be used in a C/C++ program as a *.a* library [14]. The following code shows an example of its usage:

```
1. #include <stdio.h>
2.
3. int add(int a, int b);
4.
5. int main() {
6.     int result = add(5, 3);
7.     printf("Result: %d\n", result);
8.     return 0;
9. }
```

4.2.2 Using C in Rust code

To call C functions from Rust code, the extern keyword must be used. It is also necessary to choose between the *libc* crate [14] and the *std::os::raw* module [18], which provide C-compatible types. While *std::os::raw* is more limited, it is often used instead because it is lightweight and doesn't depend on external crates. The types provided include, for example, *c_int* and *c_char*.

The following code shows an example of a C function that will later be used in Rust:

```c
#include <stdio.h>

int add(int a, int b){
    return a + b;
}
```

The following code shows how Rust will call the add function:

```rust
extern "C" {
  fn add(x: i32, y: i32) -> i32;
}

fn main() {
  let result = unsafe { add(4, 5) };
  println!("Result {}", result);
}
```

4.2.3 Bindgen

For large projects, doing this manually becomes exhausting and inefficient. To address this, Rust provides tools to automatically generate FFI bindings using Bindgen and cBindgen [14]. To use this approach effectively, the project should be organized as follows:

```
module_in_rust/
    CMakeLists.txt
    Cargo.toml
    build.rs
    in_binding.h
    out_binding
    src/
        lib.rs
```

Bindgen generates Rust FFI bindings to C. This tool analyses the header of the C file (*.h*) and creates corresponding Rust code. To use this tool, it is necessary to add the corresponding dependencies to *Cargo.toml* as shown in the following code [14]:

```
[build-dependencies]
bindgen = "0.69.4"
```

The following code creates a C header containing structures and methods to be used by Rust:

```c
struct Example {
  int field1;
  double field2;
}

int example_function(int param);
```

By using Bindgen, the following Rust file is automatically generated:

```rust
use std:os:raw::{c_int, c_double};

#[repr(C)]
pub struct Point {
    pub field1: c_int,
    pub field2: c_double,
}

extern "C" {
    pub fn example_function(param: c_int) -> c_int;
}
```

After this, the Bindgen-generated file can be included in *lib.rs*, enabling the use of C functions directly from Rust.

```rust
include!("bindings.rs");

fn main(){
    let result = unsafe { add(5, 3) };
    println!("The result: {}", result);
}
```

4.2.4 cBindgen

In turn, cBindgen generates C FFI bindings for Rust. In this case, the Rust code is analysed, and C header files are generated. To use cBindgen, it is necessary to add the dependency to the *Cargo.toml*, as shown in the following code [14]:

```
[build-dependencies]
cbindgen = "0.26.0"
```

For this purpose, the functions intended to be used in Rust must be declared as follows:

```rust
#[no_mangle]
pub extern “C” fn add( a: i32, b: i32) -> i32 {
    a + b
}
```

Subsequently, cBindgen generates the following header:

```
#include <stdint.h>
int32_t add(int32_t a, int32_t b);
```

The following Figure 14 shows the Bindgen and cBindgen:

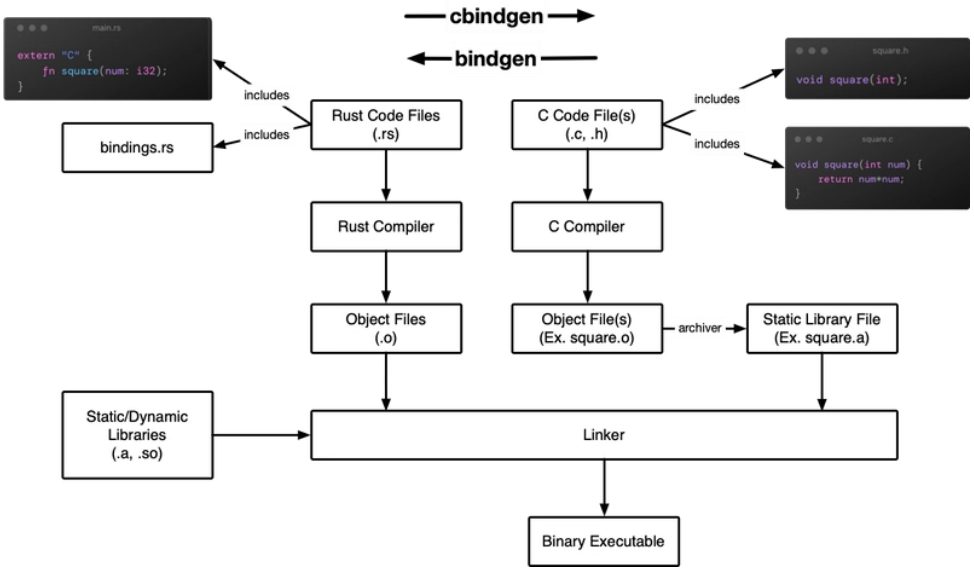


Figure 14 – Workflow of Bindgen and cBindgen
Source: Rust FFI and bindgen: Integrating Embedded C Code in Rust [14]

4.2.5 CXX

Another approach studied was the cxx library, which presents itself as a safe mechanism enabling secure interoperability between Rust and C++, allowing bidirectional calls between the two languages. The cxx uses a declarative approach through the *#[cxx::bridge]* macro, where functions and types to be shared between Rust and C++ are explicitly defined [19] [20]. Figure 15 illustrates a diagram of how cxx operates.

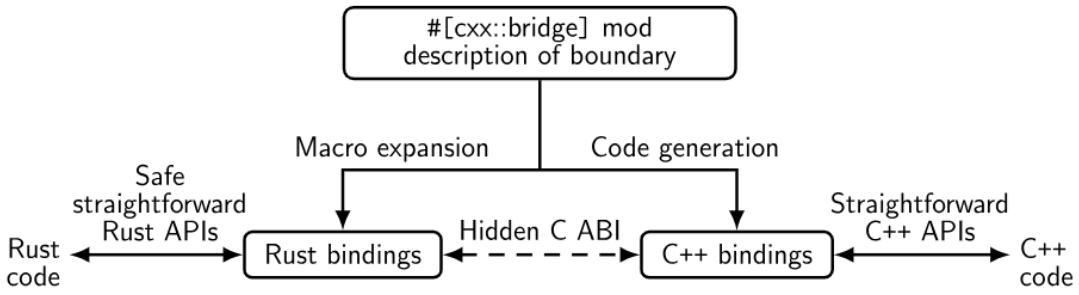


Figure 15 - Schematic representation of the cxx mechanism
Source: CXX — safe interop between Rust and C++ [20]

To use this approach, the dependency must be added to *Cargo.toml* [20], as demonstrated in the following code:

```
[dependencies]
cxx = "1.0"
```

Source: cxx-rs github [21]

The following project structure can be considered for implementing cxx:

```
demo/
    include/
        blobstore.h
    src/
        blobstore.c
        main.rs
    Cargo.toml
    build.rs
```

Source: CXX — safe interop between Rust and C++ [20]

Using a single Rust module, it is possible to have both FFI bindings. To define this module, the *#[cxx::bridge]* attribute is used, which serves as a bridge between the languages by creating a shared interface. Within this module, Rust definitions are placed inside the extern *"Rust"* block, and C++ definitions are placed inside the extern *"C++"* block [20].

The following code illustrates an example of that module, which would be located in the *main.rs* file.

```rust
#[cxx::bridge]
mod ffi {
    // Any shared structs, whose fields will be visible to both languages.
    struct BlobMetadata {
        size: usize,
        tags: Vec<String>,
    }

    extern "Rust" {
        // Zero or more opaque types which both languages can pass around but
        // only Rust can see the fields.
        type MultiBuf;

        // Functions implemented in Rust.
        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        // One or more headers with the matching C++ declarations. Our code
        // generators don't read it but it gets #include'd and used in static
        // assertions to ensure our picture of the FFI boundary is accurate.
        include!("demo/include/blobstore.h");

        // Zero or more opaque types which both languages can pass around but
        // only C++ can see the fields.
        type BlobstoreClient;

        // Functions implemented in C++.
        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(&self, parts: &mut MultiBuf) -> u64;
        fn tag(&self, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    } }
```

Source: CXX — safe interop between Rust and C++ [20]

```rust
// build.rs

fn main() {
    cxx_build::bridge("src/main.rs")  // returns a cc::Build
        .file("src/demo.cc")
        .std("c++11")
        .compile("cxxbridge-demo");

    println!("cargo:rerun-if-changed=src/main.rs");
    println!("cargo:rerun-if-changed=src/demo.cc");
    println!("cargo:rerun-if-changed=include/demo.h");
}
```

Source: CXX — safe interop between Rust and C++ [20]

This approach allows all unsafe functions to be encapsulated inside a single block, instead of calling C functions individually within unsafe blocks. Besides this, it also eliminates the need for additional wrappers and performs static analysis, which reduces the risk of errors during compilation and, unlike Bindgen, does not automatically generate unsafe bindings.

One disadvantage of this approach is that it operates at a lower level than Bindgen, requiring manual configuration of modules. However, once configured, the entire process is handled transparently.

In the context of the Karvel project, this approach proves particularly advantageous as it promotes safer and more effective integration between Rust and C++. Nonetheless, it is important to note that *cxx* is specifically designed for interoperability with C++ and doesn't offer direct support for C. Despite attempts to extend its use to C, the tool is inherently limited to C++ interoperability.

4.2.6 Libloading

Libloading is a Rust crate that enables loading dynamic libraries (e.g., *.so* files on Linux) at runtime. This approach allows interaction with libraries that are not statically linked to the binary, offering flexibility and eliminating the need to generate bindings [22].

As with the previous approaches, it is necessary to add the following dependency to *Cargo.toml*:

```toml
[dependencies]
libloading = "0.8"
```

Source: Crate libloading [22]

Libloading allows a library to be loaded dynamically, and its functions accessed through pointers, using a safe Rust API [22]. The crate provides two main components:

- **Library**: Represents the loaded library.

- **Symbol**: Represents a function or variable exported by the library.

```rust
fn call_dynamic() -> Result<u32, Box<dyn std::error::Error>> {
    unsafe {
        let lib = libloading::Library::new("/path/to/liblibrary.so")?;
        let func: libloading::Symbol<unsafe extern fn() -> u32> = lib.get(b"my_func")?;
        Ok(func())
    }
}
```

Source: Crate libloading [22]

However, in safety-critical systems, the use of dynamic libraries is not recommended, as these introduce dependencies and runtime decisions that compromise predictability and reliability. For this reason, it was deemed unsuitable for the case study.

After studying how Rust can be integrated with C in Karvel, the decision was made to use Bindgen and cBindgen. The next step was to move to the practical phase.

4.3 Memory Manager

For applying Bindgen and cBindgen, this phase involved setting up and integrating a Rust module within Karvel before implementing its internal functionality. As an initial test, a simple "hello world" module was created to verify that the integration was functioning correctly.

Following this, the module selected for conversion to Rust is the Memory Manager. Which is an integral module of Karvel that manages memory operations. The memory manager is responsible for handling memory operations requested by other applications in a cyclic manner. Provides the capability to load memory as requested by other applications, dumping memory contents for analysis or backup purposes, performing checksum operations to verify memory integrity and managing or processing memory operation requests. Memory managers provide a public interface to be used by other applications to create requests.

The component maintains an operation queue and multiple data queues to manage and process memory requests efficiently. The operation queue is a linked list that contains the following attributes:

- A pointer to the next element in the operation queue, according to the order in which they were received.
- The status of the operation that can be free, ready, operation successful, operation unsuccessful, reviewed or in progress.
- The error code records the execution that is used during the execution of the requests
- The operation identifier (ID) uniquely identifies the request.

- The operation type can be copying file, move file, write file, read file, checksum memory, dump memory, and load memory.
- The data index that contains the position in the data queue.

The data queue is implemented as an array that stores the specific information associated with each request. Alongside the data, each array cell includes a flag indicating whether the cell is currently in use. This flag is essential for managing the array efficiently during runtime: as the application progresses, cells are dynamically marked as either occupied (true) or free (false). This allows the system to reuse memory slots without the need to dynamically allocate or shift elements in the array. Figure 16 shows the operation queue and data queues.

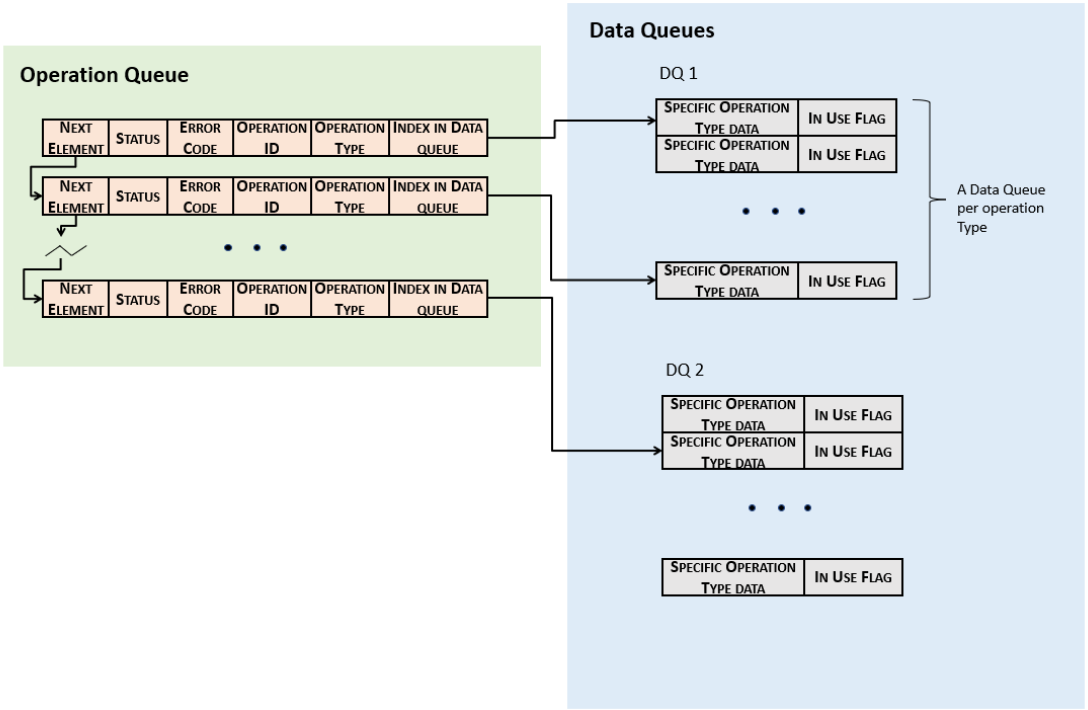


Figure 16- Operation Queue and Data Queue

4.4 Integrating the Memory Manager Module into Karvel

The *Karvel2Rust* repository is used for developing the Rust module. Subsequently, the *memory_manager_rust* module was added to the Karvel in *aws* specific folder. For this purpose, the Rust folder was organized as follows:

- memory_manager_rust
  - cheaders/
    - c_header.h
  - src/
    - lib.rs
  - build.rs
  - Cargo.toml
  - CMakeLists.txt
  - wrapper.h
  - in_bindings.h
  - out_bindings.h

Although Karvel defines a specific folder structure for its C modules, this convention was not strictly followed for the Rust implementation, as Rust projects follow a different organizational model centred around Cargo. Nonetheless, a *CMakeLists.txt* file was included to ensure that the module could still be correctly integrated into Karvel's CMake based build system.

The *src* directory holds the implementation of the memory manager module along with its associated unit tests.

The *Cargo.toml* file is mandatory in a Rust project because it is the manifest written in the *TOML* format. It is used by Cargo and defines everything needed to compile, test and run the project [23].

This file contains metadata such as package information (name and version), dependencies (crates used), build configuration (e.g. development or release profile), optional features (that can enable or disable specific parts of the code) [23].

For the memory manager module, the *Cargo.toml* file is presented below. Its contents will be explained in detail in the following sections.

```
[package]
name = "memory_manager_rust"
version = "0.1.0"
edition = "2021"

[lib]
crate-type=["staticlib"]

[build-dependencies]
bindgen = "0.69.4"
cbindgen = "0.26.0"

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"

[profile.test]
panic = "abort"

[dependencies]
heapless = "0.8"
panic-halt = "1.0.0"

[features]
std = []
file_system = []
context_manager = []

[dev-dependencies]
rusty-fork = "0.3.0"
```

As previously explained, the communication between C and Rust is established through Bindgen and cBindgen. To enable this interaction, for Bindgen it is necessary to define a C headers file, which includes all the header files containing the structures, constants and functions that the Rust code needs to access from the C modules [24]. As an example of the folder:

```
cheaders
    a. context_manager_interface.h
    b. file_system_interface.h
    c. pus_svc_6_interface.h
```

To streamline the binding generation process, a wrapper file is created, which includes all the necessary headers from the *cheaders* folder. This wrapper acts as a unified entry point for Bindgen [24]. An example of such a wrapper is:

```
#include "cheaders/context_manager_interface.h”
#include "cheaders/file_system_interface.h”
#include "cheaders/pus_svc_6_interface.h”
```

The wrapper is referenced in the *build.rs* script at the root of the project, which is automatically executed by Cargo during the build process, before the rest of the crate

is built. The script uses Bindgen to parse the wrapper and generate the corresponding Rust FFI bindings at compile time, making the C functions and types accessible from the Rust side [24]. The code below illustrates a typical use of this setup.

```
let input_bindings = bindgen::Builder::default()
                      .header("wrapper.h")
                      .generate()
                      .expect("Unable to generate input bindings");
input_bindings
      .write_to_file("in_bindings.rs")
      .expect("Couldn't write input bindings!");
```

The *bindgen::Builder::default*, creates a builder instance with the default settings, where it is possible to customize the Builder with additional configuration. The *.header* method specifies the path to the C header file previously mentioned, from which the Rust bindings will be generated. The *.generate* triggers the generation of bindings, where this operation returns a Result object. And finally, the *.expect* that extracts the Result information and in case of error the message "Unable to generate input bindings" is displayed and the build process is terminated [24]. Finally, write the input bindings into the file *in_bindings.rs*.

Different from Bindgen, cBindgen uses the *build.rs* file to automatically generate a header file during the build process [14]. The code shows the way.

```
let crate_dir = env::var("CARGO_MANIFEST_DIR").unwrap();
let mut config: cbindgen::Config = Default::default();
config.language = cbindgen::Language::C;
cbindgen::generate_with_config(&crate_dir, config)
        .unwrap()
        .write_to_file("out_bindings.h");
```

First, the *CARGO_MANIFEST_DIR* macro is used to retrieve the path to the project's *Cargo.toml* file. Then, a *cbindgen::Config* object is created with default values. This configuration is customizable, and in this case, it is modified to specify that the output header should be written in the C language. After that, the *cbindgen::generate_with_config* function is invoked to generate the bindings. This operation returns a Result type, which is obtained by unwrapping and stored in the file named *out_bindings.h* if successful, otherwise the program goes into panic mode [25].

Finally, *CMakeLists.txt*, which is mandatory for the build system, contains the following lines:

```
cmake_minimum_required(VERSION ${CMAKE_MINIMUM_VERSION)
set (MODULE_NAME memory_manager_rust)
set_property(GLOBAL APPEND PROPERTY MISSION_LIBS "memory_manager_rust")
```

After organizing the module, it was necessary to incorporate it into Karvel. To do that, the *memory_manager_rust* had to be added to the configuration file. The structure of the configuration is represented in the following line.

```
1. { .DevId = 26,. Type = LDR_ObjectType_APP, .SymbolName =
"Memory_Manager_Main”,    .SymbolAddress = (void*) &Memory_Manager_Main,     .ModuleName =
"memory_manager_rust",   .Priority = 250, .StackSize = 16384, .CpuAffinity = 0 }
2.
3. extern void Memory_Manager_Main();
```

This gives the possibility to define the features of the applications or libraries. Where the *DevId* field is a unique number defined by the user. The *Type* field specifies whether the module should be treated as an application or a library. The *SymbolName* is the name of the main function for the application, while the *SymbolAddress* stores the reference to this main function. The *ModuleName* field identifies the module. The *Priority* field defines the initial priority of the task. The *StackSize* item specifies the amount of memory allocated for the task's stack during its creation, ensuring sufficient space for function calls and local variables. Finally, the *CpuAffinity* field indicates the Central Processing Unit (CPU) core to which the task is bound.

After adding the new module, it was necessary to remove the memory manager in C so that the steps were the same as when adding it. Once it was removed, there was a need to change the memory manager in all modules that were dependent on it. To do this, changes were made to *CMakeLists.txt*.

```
1. set (MODULE_DEPENDENCIES
2.      framework_al
3.      memory_manager -> memory_manager_rust
4.      Converters
5.      pus_library
6. )
```

4.5 Implementation of the memory manager in Rust

Following the successful integration of the Rust module into Karvel, interoperability between C and Rust was achieved. The Memory Manager, implemented in Rust, is now invoked by Karvel tasks, alongside the other C tasks. Once the integration had been validated, the project proceeded to the implementation phase.

This section provides a detailed account of the Memory Manager development. It begins with adapting the module to a *no_std* environment, then describes how the entry point was designed and integrated into Karvel's cyclic task execution. Next, it covers the creation of a public interface and C bindings for communication with other modules, followed by the implementation of operation and data queues, using the singleton pattern to safely manage shared mutable state. Finally, it discusses conditional compilation for optional modules and how various issues encountered during development were addressed.

4.5.1 Development in a *no_std* Environment

Embedded Systems are used in a variety of environments, each with its own OS characteristics, and this introduces some limitations when producing code for this

type of system. Compared to personal computers, embedded systems have several constraints. They typically feature a single CPU, while modern computers have multiple processing cores. The clock speed in embedded systems is usually measured in megahertz, whereas in computers it commonly reaches gigahertz. Additionally, embedded systems may have as little as 16 KB of Random Access Memory (RAM), while personal computers often have several gigabytes. These limitations arise mainly due to constraints in size, power consumption, and heat dissipation. As a result, these types of systems require low-level programming [26].

There are two general classifications of embedded systems programming. Hosted environments, which are close to a normal computer environment, where there aren't many restrictions because of the system interface, such as Portable Operating System Interface (POSIX). This type of environment provides interaction with various systems such as file systems, networking, memory management and threads [26].

Then there are bare metal environments, where the OS runs directly on the hardware. This means that no code is loaded before the application starts and without this it is not possible to use standard libraries [26].

The std crate is Rust's standard library (std), which assumes that the program is compiled on an OS with the characteristics we find in computers and provides a set of functionalities commonly used in these OSs, such as threads, files, sockets and processes. As well as providing abstractions of the OS, it also provides a runtime which, among other things, takes care of setting up stack overflow protection, handling command line arguments, and spawning the main thread before the main function of a program is invoked [26].

Instead of using the *libstd* crate, it is possible to use the *libcore* crate, which makes no assumptions about the system where the program is running on and excludes features that aren't desirable on embedded systems, such as memory allocation. It should be an alternative to *libstd*, as it also provides primitive language Application Programming Interfaces (*APIs)* such as floats, characters, slices, but has no APIs for anything involving memory allocation and I/O [26].

In summary, omitting the std in Rust restricts access to various functionalities typically provided by it, including:

- String manipulation (*String*).
- Common collections (*Vec*, *HashMap*, etc.).
- Input/output macros (*println*!, *eprintln*!).

Figure 17 provides an overview of functionalities available in *no_std* environments compared to those requiring the std.

feature	no_std	std
heap (dynamic memory)	*	✓
collections (Vec, BTreeMap, etc)	**	✓
stack overflow protection	✗	✓
runs init code before main	✗	✓
libstd available	✗	✓
libcore available	✓	✓
writing firmware, kernel, or bootloader code	✓	✗

* Only if you use the `alloc` crate and use a suitable allocator like alloc-cortex-m.

** Only if you use the `collections` crate and configure a global default allocator.

** HashMap and HashSet are not available due to a lack of a secure random number generator.

Figure 17 -Comparison of Features Between std and *no_std* Environments
Source: *no_std* Rust Environment [26]

The *#![no_std]* attribute was added at the top of the *lib.rs* file to remove the dependency on the Rust std [27].

```
1. #![no_std]
2. #![no_main]
```

Additionally, the *#![no_main]* attribute was also included, as in a *no_std* environment the standard main function is not used as the program entry point, contrary to the default assumptions in Rust.

After compiling the program, the following error was encountered:

```
1.  error: `#[panic_handler] ` function required, but not found
```

This error indicates that, in a *no_std* environment, the panic handler is not defined by default, like as the std's default panic behaviour is no longer available. Therefore, it is necessary to implement a custom mechanism to handle panic situations. For this purpose, the *panic-halt* crate [28] was used, which simply halts the execution when a panic occurs. To include this crate in the project, the following line was added to the *Cargo.toml* file:

```
1. panic-halt = "1.0.0"
```

In addition, the following line was added at the beginning of the *lib.rs* file:

```
1. external crate panic_halt;
```

Once the panic handler was defined, a new linker error appeared during compilation:

```
1. /usr/bin/ld: x64-linux/asw/memory_manager_rust/libmemory_manager_rust.a(core-
a88b4ab71963f9fd.core.46aa9df3d3dcdeb1-
cgu.0.rcgu.o):(.data.DW.ref.rust_eh_personality[DW.ref.rust_eh_personality]+0x0): undefined
reference to `rust_eh_personality'
2. collect2: error: ld returned 1 exit status
3. gmake[3]: *** [CMakeFiles/x64-linux_cpu1.dir/build.make:164: x64-linux_cpu1] Error 1
4. gmake[2]: *** [CMakeFiles/Makefile2:999: CMakeFiles/x64-linux_cpu1.dir/all] Error 2
5. gmake[1]: *** [Makefile:91: all] Error 2
6. make: *** [Makefile:33: all] Error 2
```

This issue occurs because the compiler is still expecting a personality function for stacking unwinding (*rust_eh_personality*), which is part of Rust's exception handling used in environments where unwinding is supported, like the std. However, since the project targets a *no_std* environment with panic abort semantics, stack unwinding should be disabled altogether [27].

To resolve this, it is necessary to explicitly instruct the compiler to use abort as the panic strategy [27]. This can be achieved by including the following in the *Cargo.toml* file:

```
1. [profile.dev]
2. panic = "abort"
3. [profile.release]
4. panic = "abort
```

And in *lib.rs*:

```
1. #[no_mangle]
2. pub extern "C" fn rust_eh_personality() {}
```

4.5.2 Cyclic Task Integration and Entry Point Function

To begin the development of the memory manager module, one of the system requirements is that the application must run in a cyclic manner, continuously processing incoming requests. For this purpose, the following function defines the main entry point for the memory manager module:

```
#[no_mangle]
pub extern "C" fn Memory_Manager_Main() {

// Initialsize the Memory Manager
let _status = MemoryManager_Init();

//Initialize the Memory Interface
unsafe {
  MEM_Init();
}

// Wait for the system to be ready
unsafe {
  ES_AL_WaitForSystemReady();
}

// Loop while the application check is running
while unsafe { ES_AL_RunAppCheck() } {
// Process cyclic operations for the Memory Manager module
  MemoryManager_CyclicOperationsProcess();

// Exit the application
unsafe {
  ES_AL_ExitApp();
}
}
```

It is important to note that the *Memory_Manager_Main* function is declared using the *pub extern "C"*, it is allowed to be accessed by other modules within Karvel. This is important because the memory manager is part of a task managed by the system's scheduler, which requires direct access to the module's entry point.

This function initializes the memory manager, including the creation of a mutex, which is used for synchronization purposes. As the application runs cyclically, continuously receiving and executing requests from an operation queue, it may also receive new requests asynchronously from other modules at any time. This behaviour requires proper synchronization mechanisms to ensure safe concurrent access. In this case, Karvel has its own mechanism for handling each type of OS.

Next, the scrubbing mechanism is initialized. This component is responsible for periodically performing memory clean-up or validation operations to ensure the integrity and reliability of memory usage over time. Scrubbing reads the memory, sequentially and in a loop, forcing the hardware to recognize errors and thus triggering correcting measures. Once these initial steps have been completed, the application signals to the scheduler that it is ready to run by entering the appropriate application phase.

Next, the Memory Interface module is initialized. Although this module has not converted. However, it interacts with the memory manager module and is responsible for executing memory-related requests. This module is designed as a separate component from the Memory Manager and is intended to be architecture-specific, as it directly interfaces with the various types of memory present in the target system.

Finally, the system enters a wait state until the RTOS signals that it is ready to proceed with memory management operations. Only then does the application start its main cyclic execution loop.

Figure 18 illustrates the workflow of the *Memory_Manager_Main* function:

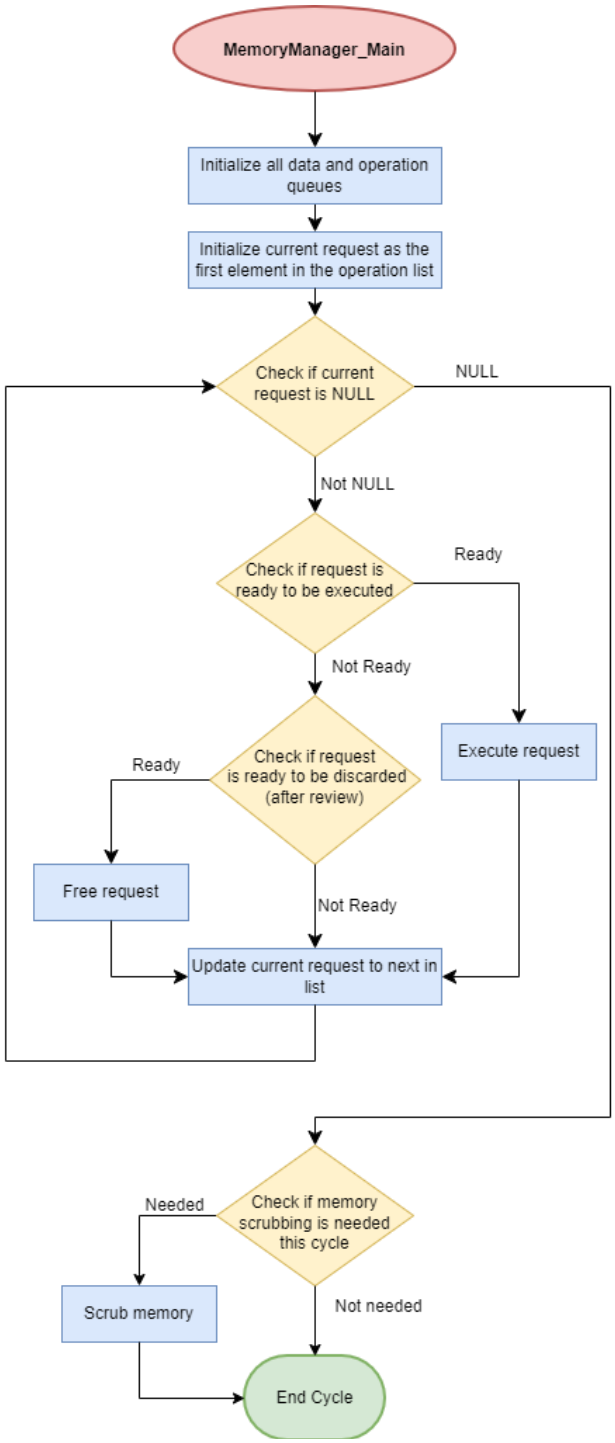


Figure 18 - Memory Manager Workflow

However, an unforeseen problem has arisen: the code has not been compiled. When analysing the error, the compiler couldn't find the *memory_interface* functions, even though they were in the *wrapper.h* file. The code generated by the bindings was analysed and the functions were not present, suggesting that there was a problem because the other modules had generated the FFI correctly. After a deep analysis it was concluded that the memory manager (implemented in Rust) was the only one that depended on the memory interface (implemented in C), which meant that it wasn't compiled and didn't generate the *memory_interface.a* and consequently the bindings weren't generated. To fix this, it was necessary to force the inclusion of the module in CMake.

```
1. target_link_options(memory_manager_rust INTERFACE -Wl,--whole-archive
${CMAKE_BINARY_DIR}/${SYSTEM}-${PLATFORM}/hw_interfaces/memory_interface/libmemory_interface.a -
Wl,--no-whole-archive)
```

4.5.3 Public Interface and C Bindings Generation

At this point, it was necessary to define a public interface that could be accessed by the other modules and that would be the target of the cBinding. To do this, all the structures and functions *create* had to be public, as shown in the following code.

```
1.  // An example of a structure
2.  #[repr(C)]
3.  #[derive(Eq, Ord, PartialEq, PartialOrd, Debug, Copy, Clone)]
4.  pub struct mm_operation_t {
5.      pub Status: u8, // Status of this operation request
6.      pub ErrorCode: u8, // Error code
7.      pub Type: u8,   // Type of operation (copy, move, load...)
8.      pub DataIndex: u16, // Index of data on the respective data structure
9.      pub OpId: u16, // Operation ID
10.     pub TaskId: u32,  // Task ID of the task that called this
11. }
12. // An example of a function
13. [no_mangle]
14. pub extern "C" fn MemoryManager_CreateGeneralMemoryLoadRequest(
15.     OpId: &mut u16,
16.     Type: u8,
17.     LoadAddress: *mut c_void,
18.     LoadMemoryId: u8,
19.     Length: address_length_t,
20.     LoadData: *const u8,
21.     Checksum: bool,
22. ) -> u8 {
23.   // ...
24. }
```

The result is the C code present in the *out_bindings.h* file, to which C modules have access.

```c
typedef struct mm_operation_t {
  uint8_t Status;
  uint8_t ErrorCode;
  uint8_t Type;
  uint16_t DataIndex;
  uint16_t OpId;
  uint32_t TaskId;
} mm_operation_t;

uint8_t MemoryManager_CreateGeneralMemoryLoadRequest(uint16_t *OpId,
                                                     uint8_t Type,
                                                     void *LoadAddress,
                                                     uint8_t LoadMemoryId,
                                                     address_length_t Length,
                                                     const uint8_t *LoadData,
                                                     bool Checksum);
```

To organize the project, instead of C, where the code is divided into *.h* and *.c* files, Rust divides the code into modules [29]. Instead of writing all the code in a *lib.rs* file, this modular structure is used. Each module groups together related code. For example, the *memory_manager_create_operations.rs* file contains all the code related to creation operations. When other modules need to access the functionality provided by this module, it is necessary to:

```rust
pub mod memory_manager_create_operations;
use crate::memory_manager_create_operations::*;
```

4.5.4 Linked List Implementation in a *no_std* Environment

To implement the operation queue, in conformity with the architecture described above, a linked list was used, like in the C version. However, it was decided not to develop a manual implementation of the linked list, as is common in C. This is because an own implementation would require careful management of pointers and memory, which would imply the use of unsafe blocks in Rust, which would compromise the safety guarantees offered by the language.

Given the limitations of the *no_std* environment, the *SortedLinkedList* structure provided by the *heapless* crate [30] was chosen. This alternative allows us to benefit from safe implementation.

The *heapless* crate provides data structures with static allocation, compatible with systems that do not support dynamic memory allocation. It is important to note that the generic *LinkedList* implementation has been removed from recent versions of *heapless*, with *SortedLinkedList* being the currently supported alternative [30]. This requires the following dependency to be added to the *Cargo.toml* file:

```toml
[dependencies]
heapless = "0.8"
```

And in the *lib.rs* file, the following module needed to be imported:

```rust
use heapless::sorted_linked_list::{SortedLinkedList, Min};
```

The order of the list is determined by the type of parameter, *Min* for ascending order and *Max* for descending order. In this case, since the requirements specify that the requests should be sorted on a first-come, first-served basis, and assuming that the number of requests is incremental, the ordering was configured based on *Min*.

Different from the C implementation, which uses global variables, a safer approach was chosen by not using global variables and encapsulating the linked list in a structure, as shown in the following code:

```
1. [derive(Debug)]
2. struct MM_OperationRequest {
3.     // Operation requests (Linked List)
4.     MM_AvailableRequest: SortedLinkedList<mm_operation, LinkedIndexUsize, Min,
MAX_NR_MEMORY_OPERATION>,
5 }
```

The following function is used to initialize it:

```
1. fn Memory_Manager_Init() -> MM_OperationRequest {
2. let mut mm_AvailableRequest = SortedLinkedList::<mm_operation, LinkedIndexUsize, Min,
MAX_NR_MEMORY_OPERATION>::new_usize();
3.
4. mm_OperationRequest = MM_OperationRequest {
5.    MM_AvailableRequest: mm_AvailableRequest,
6.   };
7. }
```

Despite the initial advantages, this approach had significant limitations. In particular, segmentation faults occurred. The most critical limitation appeared when, at some stage of the program, it was necessary to remove or change elements from the linked list, operations not supported by *SortedLinkedList*, which only allows removing elements at the beginning or end of the list.

These limitations resulted in a reformulation of the architecture, which motivated an investigation of the most appropriate data structure. As a result, the linked list was replaced by the *Vec* structure, also provided by crate *heapless* [30]. This alternative ensures compatibility with *no_std* environments and allows elements to be removed and modified.

Consequently, the architecture has been updated to incorporate this solution, as shown in Figure 19:

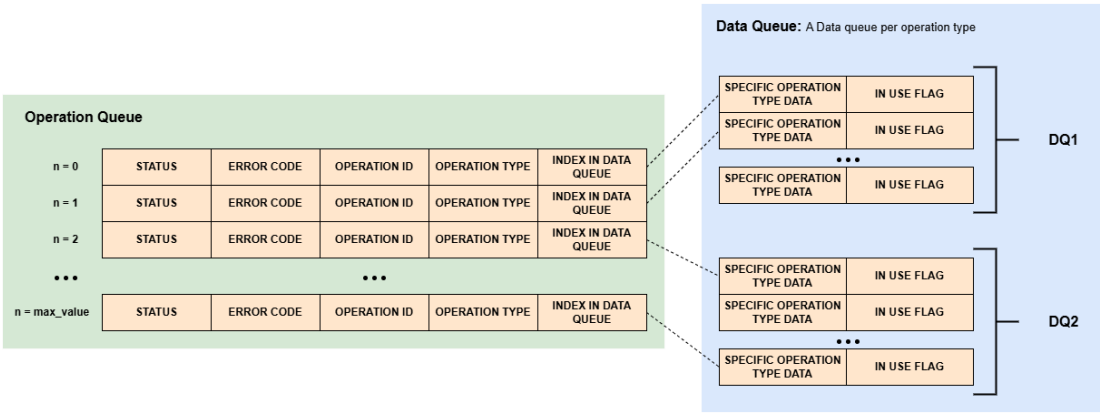


Figure 19 - Memory manager diagram for Rust implementation

The request creation process follows the workflow illustrated in Figure 20.

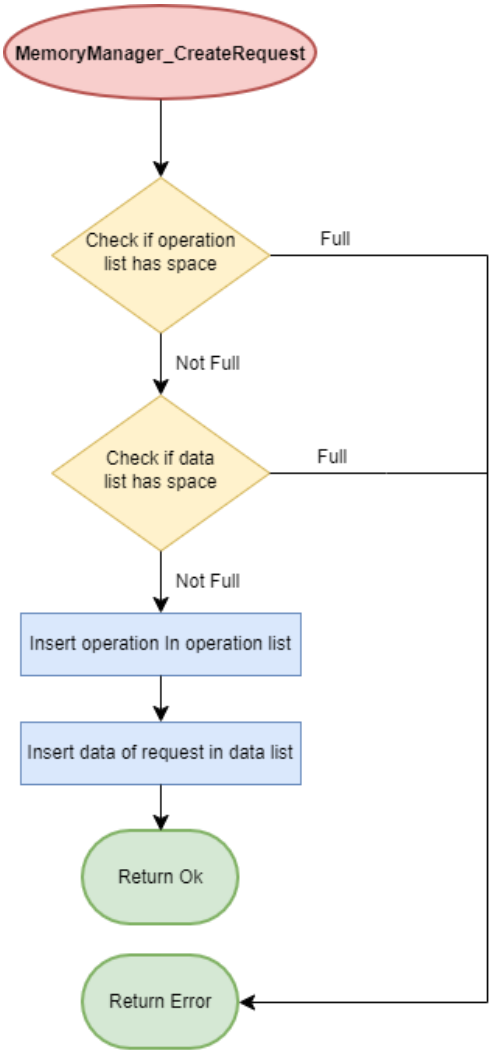


Figure 20- Request Creation Workflow

4.5.5 Global Variables and the Singleton Approach

As previously mentioned, the inability to avoid using global variables comes from the fact that the queues need to be accessed in different parts of the code, both for their creation and during the execution of operations. This need makes it impossible to avoid the problem of using global variables. Although the use of global variables is not recommended in either C or Rust, in C their use is uncontrolled, while in Rust the compiler requires additional security guarantees, because access to mutable data must be secure.

In this case, mutability is necessary because queues are dynamic structures that receive and remove requests over time. Since the Rust compiler cannot guarantee safe mutable access to these global structures, access requires the use of unsafe blocks. This scenario is not ideal because, in addition to the need for unsafe code, the global variable remains visible throughout the program, limiting the effectiveness of the compiler's borrow checker.

To this purpose, the singleton pattern has been implemented, which guarantees the existence of a single instance of the structure throughout the entire program, with the program going into panic mode if a second instance is created [31]. This approach has a runtime cost because it requires encapsulation, but this initial cost allows the above problems to be solved.

Furthermore, by encapsulating the methods, they can only be accessed if there is a property or reference to the structure to which they belong.

So, all the queues have been implemented and encapsulated as shown in the following code:

```rust
static mut MM_OPERATION_REQUEST: Option<MM_OperationRequest> = Some (MM_OperationRequest {
    MM_OperationRequestQueue: Vec::new(),
    IdOperation: 0,
});

#[derive (Debug)]
struct MM_OperationRequest {
    MM_OperationRequestQueue: Vec<mm_operation_t, MAX_NR_MEMORY_OPERATION>,
    IdOperation: u16,
}
impl MM_OperationRequest {
    fn get_instance() -> &'static mut MM_OperationRequest {
        unsafe {
            MM_OPERATION_REQUEST
                .as_mut()
                .expect("MM_OperationRequest instance is not initialized")
        }
    }

    fn get_operation_id(&mut self) -> u16 {
        if self.MM_OperationRequestQueue.is_empty() {
            self.IdOperation = 0;
        } else {
            self.IdOperation += 1;
        }

        self.IdOperation
    }
    fn clear_operation_queue(&mut self) {
        self.MM_OperationRequestQueue.clear();
    }

```

This encapsulation does not eliminate the need for unsafe blocks, but it mitigates the associated risks by promoting a safe and organized access pattern to the global variable and reducing the propagation of unsafe through the code.

4.5.6 Design and Management of Data Queues

For the other queues, it was decided to use arrays, like the C implementation, as these structures don't require complex operations like the operation queue. In addition, the *Vec* type was avoided because removing elements implies moving the rest, not changing the order in which the requests arrive, but simply removing the elements that are empty. This introduces a performance penalty. Since data queues don't need this flexibility, the use of static arrays proves sufficient and more efficient.

Similarly to the operation queue, global variables and the singleton pattern were used to ensure controlled access to the structure and persistence of state during system execution. The load data queue example demonstrates this approach:

```
static mut MM_AddressLoadData: Option<MM_AddressLoadData> = None;

/** Array with load data information */
struct MM_AddressLoadData {
    LoadDataQueue: [mm_address_load_t; MAX_NR_MEMORY_LOAD],
    AddressLoadIndex: u16,
}

impl MM_AddressLoadData {
    fn get_instance() -> &'static mut MM_AddressLoadData {
        unsafe {
            if MM_AddressLoadData.is_none() {
                MM_AddressLoadData = Some(MM_AddressLoadData {
                    LoadDataQueue: [mm_address_load_t {
                        InUse: false,
                        LoadAddress: ptr::null_mut(),
                        Length: 0,
                        Data: [0; LOAD_SIZE],
                        LoadMemoryId: 0,
                        Checksum: false,
                    }; MAX_NR_MEMORY_LOAD],
                    AddressLoadIndex: 0,
                });
            }
            MM_AddressLoadData.as_mut().unwrap()
        }
    }
}
```

4.5.7 Segmentation Fault Issue: Causes and Solutions

During development, a segmentation fault problem was identified related to the *MM_AddressLoadData* structure, specifically the *Data* field in the *LoadDataQueue* array. This field was defined as a size of *LOAD_SIZE = 1000*. This size caused a segmentation fault during execution. However, when the *LOAD_SIZE* value was reduced to 100, the error no longer occurred, clearly indicating that the problem was related to the program exceeding the limits of the stack.

The cause of the error was related to the fact that, although in a *no_std* environment, the instance initialized within the *MemoryManager_Init()* function via the *get_instance()* method forced the creation of a temporary allocation on the stack. This approach resulted in undefined behaviour and execution failures due to the size of the structure.

To solve this limitation and avoid future failures, it was decided to initialize the variable directly in its declaration, using the *static mut*, ensuring that the structure is statically allocated, instead of being created temporarily in the stack. This approach assures that the instance is created safely and remains available throughout the execution of the program. As shown in the code below:

```
 1.  static mut MM_ADDRESS_LOAD_DATA: Option<MM_AddressLoadData> =Some(MM_AddressLoadData{
 2.     LoadDataQueue: [mm_address_load_t {
 3.         InUse: false,
 4.         LoadAddress: ptr::null_mut(),
 5.         Length: 0,
 6.         Data: [0; LOAD_SIZE as usize],
 7.         LoadMemoryId: 0,
 8.         Checksum: false,
 9.     }; MAX_NR_MEMORY_LOAD],
10.     AddressLoadIndex: 0,
11. });
12.
13. /** Array with load data information */
14. struct MM_AddressLoadData {
15.     LoadDataQueue: [mm_address_load_t; MAX_NR_MEMORY_LOAD],
16.     AddressLoadIndex: u16,
17. }
18. impl MM_AddressLoadData {
19.     fn get_instance() -> &'static mut MM_AddressLoadData {
20.         unsafe {
21.             MM_ADDRESS_LOAD_DATA
22.                 .as_mut()
23.                 .expect("MM_AddressLoadData instance is not initialized")
24.         }
25.     }
26. }
```

With this change, the *MM_AddressLoadData* is now allocated statically, and the *get_instance()* method is restricted to accessing the previously initialized instance, removing the segmentation error problem.

4.5.8 Generic Data Queue Implementation

All data queues are managed based on the attribute, for example in the case of the load data queue, the *AddressLoadIndex*, which is incremented when a new request is added to the array. This index acts as a circular counter, allowing the array to be used continuously. To control the occupancy of each element in the queue is used as the *InUse* flag, a *bool* field present in each element of the array, which indicates if the respective element is available for use.

Given that the logic of updating an element is common to all queues, Rust generics were chosen. This approach allows the behaviour to be generalized, avoiding code duplication and the need to define a specific update function for each queue. To allow generalization, it was defined as follows:

```rust
pub trait DataQueueGeneric {
    fn increments_Index(&mut self);
    fn get_Index(&mut self) -> u16;
}

static mut MM_ADDRESS_LOAD_DATA: Option<MM_AddressLoadData> = Some(MM_AddressLoadData{
    LoadDataQueue: [mm_address_load_t {
        InUse: false,
        LoadAddress: ptr::null_mut(),
        Length: 0,
        Data: [0; LOAD_SIZE as usize],
        LoadMemoryId: 0,
        Checksum: false,
    }; MAX_NR_MEMORY_LOAD],
    AddressLoadIndex: 0,
});

/** Array with load data information */
struct MM_AddressLoadData {
    LoadDataQueue: [mm_address_load_t; MAX_NR_MEMORY_LOAD],
    AddressLoadIndex: u16,
}

impl DataQueueGeneric for MM_AddressLoadData {
    fn increments_Index(&mut self) {
        self.AddressLoadIndex += 1;

        if usize::from(self.AddressLoadIndex) == MAX_NR_MEMORY_LOAD {
            self.AddressLoadIndex = 0;
        }
    }

    fn get_Index(&mut self) -> u16 {
        self.AddressLoadIndex
    }
}

impl MM_AddressLoadData {
    fn get_instance() -> &'static mut MM_AddressLoadData {
        unsafe {
            MM_ADDRESS_LOAD_DATA
                .as_mut()
                .expect("MM_AddressLoadData instance is not initialized")
        }
    }
}
```

In this case, the *increments_Index* and *get_Index* methods are declared in the trait without default implementation, leaving it up to each concrete structure, such as the data queue, to define the corresponding logic. This strategy makes it possible to adapt the behaviour of each queue to its particularities, while maintaining a common interface.

```rust
pub fn MemoryManager_UpdateRequestPointer<T>(OpId: &mut u16, Type: u8, DataQueue: &mut T)
where
    T: DataQueueGeneric,
{
    let mut task_id: u32 = 0;
    let mm_operation_request = MM_OperationRequest::get_instance();

    unsafe {
        ES_AL_GetAppID(&mut task_id);
    }

    *OpId = mm_operation_request.get_operation_id();

    let operation = create_mm_operation(
        *OpId,
        STATUS_REQUEST_READY.try_into().unwrap(),
        REQUEST_SUCCESSFULLY_CREATED.try_into().unwrap(),
        Type,
        DataQueue.get_Index(),
        task_id,
    );

    let _ = mm_operation_request.MM_OperationRequestQueue.push(operation);

    DataQueue.increments_Index();
}

```

The *MemoryManager_UpdateRequestPointer* function shows how the abstraction provided by the *DataQueueGeneric* trait makes it possible to operate on different data queues in an undifferentiated way.

4.5.9 Conditional Compilation for File System and Context Manager

The memory manager module has conditional compilation, depending on the presence of the File System or Context Manager modules, which introduce specific functionalities. In C, this configuration is traditionally done as follows:

```rust
ifdef FILE_SYSTEM
//....
endif

ifdef CONTEXT_MANAGER
//....
#endif
```

The availability or non-availability of these modules is specified in the *project_vars.yml* file, where it can take the values *'yes'* or *'no'*. For example, for the File System the variable is *FILE_SYSTEM_PRESENT* and for the Context Manager it is *CONTEXT_MANAGER_PRESENT*.

```rust
FILE_SYSTEM_PRESENT: yes
CONTEXT_MANAGER_PRESENT: no
```

For conditional compilation to work on the Rust side, this information needs to be passed to the Rust code. To do this, it used *cfg*, a Rust feature that allows parts of the code to be included or excluded depending on certain conditions, such as OS, architecture, or the presence of a feature [32].

In the example below, the function is only compiled if the *file_system* feature is enabled:

```
#[cfg(feature = "file_system")]
[no_mangle]
pub extern "C" fn MemoryManager_CreateFileWriteReadRequest(
    OpId: &mut u16,
    Type: u8,
    MemoryId: u8,
    FilePtr: *mut FS_File_t,
    Data: *mut c_void,
    DataLength: address_length_t,
    Offset: address_length_t,
) -> u8 {
     // …
}

```

The features are defined by using variables in *project_vars.yml*. These variables are accessible from all the *CMakeLists.txt* files.

Based on this, the following code has been added to *CMakeLists.txt*:

```
set(CARGO_FEATURES "")
if("${FILE_SYSTEM_PRESENT}" STREQUAL "yes")
    list(APPEND CARGO_FEATURES "file_system")
endif()

if("${CONTEXT_MANAGER_PRESENT}" STREQUAL "yes")
    list(APPEND CARGO_FEATURES "context_manager")
endif()
```

Despite having access to the variables in CMake, it is still necessary to send them to the Rust compiler. To do this, it uses *corrosion_import_crate*, where variables are passed as features:

```
corrosion_import_crate (
 MANIFEST_PATH "Cargo.toml"
 FEATURES ${CARGO_FEATURES}
)
```

If, for example, *FILE_SYSTEM_PRESENT* has the value '*yes*', then the cargo build will be compiled as follows:

```
 cargo build --features "file_system”
```

For this to work, the features must be declared in the *Cargo.toml* file.

```
[features]
file_system = []
context_manager = []
```

It was also detected that the functions defined with *#ifdef FILE_SYSTEM* in the C code were not generated by Bindgen. To solve this, it was necessary to include the argument in the *build.rs* configuration:

```
.clang_arg("-DFILE_SYSTEM")
```

In this way, by defining the file system as a macro and ensuring that the functions within these clauses are included in the generated header through the binding process, we guarantee that these functions are correctly recognized in the C code and exposed on the Rust side.

4.5.10 Bindings: Header Generation Issue

During the binding generation process, issues arose related to circular dependencies between headers. Specifically, the modules *pus_svc_6_interface* and *file_system_public* depend on the file *memory_manager_interface.h*, which, in this new flow, no longer exists and is instead dynamically generated by cBindgen.

To solve this issue, it was necessary to reverse the order from the initial approach: the header from cBindgen had to be created first, so it could then be used as input for the creation of the bindings.

This change, in turn, introduced a new problem: some functions were being generated twice. The functions *Memory_Manager_MainRust* and *rust_eh_personality* are exported in *out_bindings.h* file, but since this same file is later processed by Bindgen, those functions are generated again, causing linking conflicts.

To mitigate this problem, it was necessary to explicitly block the generation of these functions during the binding process in *build.rs*, as follows:

```
1. blocklist_function("Memory_Manager_MainRust")
2. blocklist_function("rust_eh_personality")
```

4.5.11 cBinding Generation Problem: Forcing Structure Export

Another issue identified, this time related to cBindgen, was that, unlike functions, the structures were not being correctly generated in the header. To resolve this limitation, it was necessary to explicitly force the export of these structures and global variables in the *build.rs* file, using the following configuration:

```
 1. config.export.include = vec![
 2.     "mm_operation".to_string(),
 3.     "mm_file_copy_move_t".to_string(),
 4.     "mm_file_write_read_t".to_string(),
 5.     "mm_nvm_write_read_t".to_string(),
 6.     "mm_nvm_checksum_t".to_string(),
 7.     "mm_address_copy_t".to_string(),
 8.     "mm_address_checksum_t".to_string(),
 9.     "mm_address_dump_t".to_string(),
10.     "mm_address_load_t".to_string(),
11.     "MM_NrRequestCompleted".to_string(),
12.     "MM_NrRequestReady".to_string(),
13.     "MM_FirstRequest".to_string(),
14.     "MM_LastRequest".to_string(),
15.     "MM_AvailableRequest".to_string(),
16. ];
```

With this approach, it was ensured that all the required structures and variables were included in the header generated by cBindgen, allowing for proper integration with the C modules.

Up to this point, the integration has been successfully achieved. Despite the challenges encountered throughout the implementation process, these were effectively overcome, leading to the successful completion of the memory manager module in Rust. The module is currently interfacing with C code, ensuring the transmission of memory requests and their proper execution.

4.6 Additional Contribution: The karvel-sys Crate

A crate called *karvel-sys* was developed, inspired by the approach adopted by *Linux Kernel* [33], with the aim of automatically generating bindings between Rust and C for Karvel. This infrastructure aims to isolate the binding layer, facilitating its reuse and integration by other modules that need to interact with the Karvel system core written in C.

The structure of *karvel-sys* is as follows:

1. Cheaders/
    - Headers.h
2. Src/
    - lib.rs
    - print_no_std.rs
3. Bindigns_helper.h
4. Build.rs
5. Cargo.toml
6. In_bindigins.rs

Unlike the approach described in another chapter, where the generation of bindings was embedded directly in the module, here *karvel-sys* is isolated, functioning as an independent abstraction layer. Thus, any module that wishes to interact with the system can simply add a dependency to *karvel-sys*, taking advantage of the generated bindings.

One of these features is the *tlch_log!* macro, defined in the *print_no_std* module, which allows logging to the console, even in restricted environments where std is not available. This macro directly invokes a C function (*_TLCH_Log_impl*), providing a safe interface in Rust for issuing log messages.

The *lib.rs* file is configured for embedded environments and reflects the following code:

```rust
#![no_std]
#![no_main]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(non_upper_case_globals)]
#![allow(improper_ctypes)]

include!("../in_bindings.rs");

pub mod print_no_std;
pub use print_no_std::*;

#[no_mangle]
pub extern "C" fn rust_eh_personality() {}

use core::panic::PanicInfo;
```

The *print_no_std.rs* defines the macro *tlch_log!* as follows:

```rust
1.  #[repr(C)]
pub enum TlchLoggerMsgType  {
    Info,
    Warn,
    Error,
    Debug,
}

#[macro_export]
macro_rules! tlch_log {
    ($logtype:expr, $fmt:literal $(, $arg:expr)* $(,)?) => {{
        use core::ffi::c_char;

        extern "C" {
            fn _TLCH_Log_impl(
                logtype: LoggerMsgType,
                context: bool,
                file: *const c_char,
                line: i32,
                function: *const c_char,
                fmtstr: *const c_char,
                ...
            );
        }

        // All these strings are static and null-terminated (\0)
        const FILE: &[u8] = concat!(file!(), "\0").as_bytes();
        const FUNC: &[u8] = concat!(core::module_path!(), "\0").as_bytes();
        const FMT: &[u8] = concat!($fmt, "\0").as_bytes();

        unsafe {
            _TLCH_Log_impl(
                $logtype,
                true,
                FILE.as_ptr() as *const c_char,
                line!() as i32,
                FUNC.as_ptr() as *const c_char,
                FMT.as_ptr() as *const c_char,
                $($arg),*
            );
        }
    }};
}
```

To use crate, it was necessary to remove the entire bindings generation section and include *karvel-sys* in *memory_manager* as follows. In *lib.rs*:

```
1. 1.  #[cfg(not(test))]
2. use karvel_sys::*;
3.
4. #[macro_use]
5. extern crate karvel_sys;
```

And add the following code to *Cargo.toml*:

```
1. [dependencies]
2. heapless = "0.8"
3. panic-halt = "1.0.0"
4. karvel-sys = { path = "../../karvel-sys" }
```

Given the critical nature of the systems under consideration, the next step consists of performing tests on the code developed in Rust, with the aim of ensuring that its behaviour fully corresponds to the expected outcomes. This stage will be addressed in the following chapter.

5 VERIFICATION

After the implementation was completed, the verification phase of the Rust module began. The main objective of this phase is to ensure that the code meets the defined requirements, exhibits consistent behaviour, and can be safely integrated into Karvel. This chapter first presents the unit tests, which validated the internal functionality of the module and measured its coverage to verify compliance with the established coverage goals for the internship (100% statement coverage and 100% branch coverage). Next, it describes the integration of the Rust unit tests with those already existing in Karvel, including the necessary adaptations and the unification of metrics with the C modules. Finally, it discusses the validation tests, highlighting the main challenges encountered, as well as the process of changing the target to enable execution in a simulator.

5.1 Unit tests

For unit test development, the Karvel convention was initially followed, where unit tests are placed in a folder separate from the source code. However, when attempting to access internal methods, they were not recognized. After several unsuccessful attempts to include the methods in the test file, it was concluded, according to Rust conventions, that unit tests are typically written in the same file as the source code or in a separate file located in the same directory. The *tests/* folder is reserved for integration tests, which do not have direct access to non-public elements of Rust modules [34].

5.1.1 Challenges with cargo test in a *no_std* Environment

After solving the initial barrier related to the location of the test file, a file named *coverage_memory_manager.rs* was created in the same directory as the source code, following the convention used in the Karvel project.

However, when trying to run the tests using the *cargo test* command, a new problem arose related to conflicts between the libraries used for the *no_std* environment and the *std*. The following error was displayed:

```
 1. error[E0152]: duplicate lang item in crate `std` (which `memory_manager_rust` depends on):
`panic_impl`
 2.   |
 3.   = note: the lang item is first defined in crate `panic_halt` (which `memory_manager_rust`
depends on)
 4.   = note: first definition in `panic_halt` loaded from /home/cblopes/rust-
things/karvel2rust/asw_specific/memory_manager_rust/target/debug/deps/libpanic_halt-
491dca9c61f38ab4.rlib
 5.   = note: second definition in `std` loaded from /home/cblopes/.rustup/toolchains/stable-
x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/lib/libstd-f6265b21db1f990f.so,
/home/cblopes/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-
linux-gnu/lib/libstd-f6265b21db1f990f.rlib
 6.
 7. For more information about this error, try `rustc --explain E0152`.
 8. warning: `memory_manager_rust` (lib test) generated 1 warning
 9. error: could not compile `memory_manager_rust` (lib test) due to 1 previous error; 1 warning
emitted
```

This error occurs because the *memory_manager_rust* crate was configured with the *#![no_std]* attribute, using the *panic_halt* crate as the *panic_handler* implementation. However, when running tests with *cargo test*, the compiler automatically tries to include the *std*, which also defines a *panic_impl*. This results in a conflict due to the double definition of the same lang item, which is not allowed by the Rust compiler.

After deeper analysis, it was confirmed that the *cargo test* command depends on the *std* by default. To work around this problem, it was decided to run unit tests in a Linux environment, independent of the actual target environment for the code. This approach is like the strategy used for C code on the Karvel.

To follow this approach, the previously mentioned *cfg* attribute was used to distinguish between the *no_std* environment and the test environment, where the *std* is required. To make this possible, it was necessary to declare the *std* feature in the *Cargo.toml* file:

```
1. [features]
2. std = []
```

Then, conditional compilation was applied using *#[cfg(test)]* and *#[cfg(not(test))]*. For the test environment, the *#[cfg(test)]* attribute was used, and the *std* was explicitly imported:

```
1. #[cfg(test)]
2. extern crate std;
```

For the *no_std* environment, the *#[cfg(not(test))]* attribute was used, and the inverse configuration was applied to enable *no_std* and to prepare the code properly for deployment in embedded systems.

```
1. #![cfg_attr(not(test), no_main)]
2.
3. #[cfg(not(test))]
4. #[no_mangle]
5. pub extern "C" fn rust_eh_personality() {}
6.
7. #[cfg(not(test))]
8. extern crate panic_halt;
```

This setup ensures that the code remains compatible with embedded, *no_std* environments while still supporting standard unit testing on Linux. As a result, it is now possible to run unit tests without conflicts.

Following the resolution of the configuration challenges, it was then possible to proceed with the implementation of unit tests. To define a unit test in Rust, it is sufficient to annotate the function with *#[test]*. In the case of feature-specific tests, such as those for *file_system* or *context_manager*, the workflow is similar: conditional compilation is applied using *#[cfg(feature = "file_system")]*, allowing tests to be executed only when the corresponding feature is enabled.

The tests were organized based on similar functionalities, always aligned with the previously defined requirements. For example, the tests were divided according to the functionality of each data queue, specifically execution and creation. All tests follow a common structure composed of three phases:

- Setup: Initialization of variables, data structures, and necessary mocks.
- Execution: Invocation of the function or method under test.
- Verification: Use of *assert_eq!* to validate the expected results.

Below is a representative unit test example, illustrating the described structure and validating the functionality of creating a data queue:

```
#[test]
fn Test_MemoryManager_CreateGeneralMemoryLoadRequest() {

     // Setup

    let max_data: u16 = MAX_NR_MEMORY_LOAD.try_into().unwrap();

    let mm_operation_request = MM_OperationRequest::get_instance();

    let mm_address_load_t = MM_AddressLoadData::get_instance();

    /*
     *
     * @step Create a Load Request -> Execution
     *
     */
    create_data_queue_request(REQUEST_TYPE_LOAD_MEMORY_AREA.try_into().unwrap(), false);

    /*
     * @check Check Number of AddressLoadIndex Increased to one -> Verification
     *
     */
    assert_eq!(mm_address_load_t.AddressLoadIndex, 1);
    // (…)
}
```

5.1.2 Mock-Based Testing Approach

As previously mentioned, some functionalities of the Rust module depend on functions implemented in C. However, since the goal is to test the memory manager module in isolation, without depending on functionalities from other modules, the

use of stubs provided by the Karvel platform was considered as is already done for C tests.

However, after several unsuccessful attempts to integrate these stubs with the Rust code, due to the need to replicate the same integration process used in C and the inability to link correctly to the functions in the test environment. It was decided to implement mocks directly in Rust. This approach allows for controlled and independent simulation of the expected behaviour of external functions.

To achieve this, the file *mock_function_unit_tests.rs* was created in the same directory as the tests, where a module marked with the *#[cfg(test)]* attribute was defined. This module contains the implementation of the mock functions, with the following structure:

```
#[cfg(test)]

pub mod mock_c_bindings {
    #[no_mangle]
    pub fn ES_AL_RunAppCheck() -> bool {
        true
    }

// ...
}
```

5.1.3 Crate rusty: Purpose and Application

After running several tests, it was identified that previous tests were affecting the state of subsequent ones. For example, when testing the addition of a new element to the operation queue, the element was rejected because the queue was already full. However, since this was a new test, the expected behaviour was for the queue to be empty and for the element to be accepted as the first one. This undesired behaviour occurred because requests created in previous tests were still in the operation queue, compromising the isolation of the test.

To address this issue, the *rusty_fork* crate was used. It allows each test to run in a separate process by forking the main process before each test execution. This ensures that any changes, particularly to global variables, don't affect subsequent tests. With this approach, each test runs in a clean, isolated environment, ensuring reliable results and preventing interference between tests [35].

To use this crate, it was added to the development dependencies in the *Cargo.toml* file:

```
[dev-dependencies]
rusty-fork = "0.3.0"
```

Then, at the beginning of each test file, the following line was included:

```
use rusty_fork::rusty_fork_test;
```

Finally, all tests are encapsulated by the *rusty_fork_test!* macro, as shown in the example below:

```
rusty_fork_test! {
  #[test]
  fn test01 () {//...}
  fn test02 () {//...}
  //...
}
```

5.1.4 Code Coverage Analysis with *grcov* and *lcov*

To ensure the quality and robustness of the developed code, it was essential to measure test coverage. Test coverage helps identify which parts of the code were actually exercised during testing, highlighting untested areas that may hide potential issues [36].

There are several metrics used to evaluate test coverage. Line coverage checks that each line of code was executed at least once during testing. Region coverage focuses on larger blocks, such as functions, and checks that they were executed. Finally, branch coverage evaluates if all possible decision paths in the code, such as those in an if statement, were correctly executed [36].

To generate unit test coverage in Rust and given that Karvel uses *gcov* and *lcov* for C code, an equivalent solution was searched for Rust, with the goal of eventually integrating C and Rust test coverage into a single report. For this purpose, the *grcov* tool was selected.

Initially, a local script was created in the module directory. The procedure is based on the following steps:

```
cargo clean

export RUSTFLAGS="-C instrument-coverage"
export LLVM_PROFILE_FILE="./target/debug/memory_manager-%p-%m.profraw"

# cargo test

cargo test --features "file_system context_manager"

grcov . -s . --binary-path ./target/debug/ -t lcov --branch --ignore-not-existing --keep-only "src/**" --ignore "src/coverage_memory_manager.rs" --ignore "src/mock_C_function_unit_tests.rs"  -o ./target/debug/memory_manager.lcov

genhtml -o ./target/debug/coverage_report ./target/debug/memory_manager.lcov --branch-coverage --function-coverage
```

To enable code coverage in Rust, the *'-C instrument-coverage'* flag is used. This is a native coverage instrumentation provided by the Rust compiler. It is widely used for source-based code coverage and instructs the compiler to embed coverage metadata into the compiled binary. This instrumentation allows the collection of detailed execution data during tests, such as how many times each line, region, or branch of code is executed [36].

During test execution, coverage data is stored in a *.profraw* file. This file is essential for generating the final coverage report. The use of the *%p* (process ID) and *%m* (binary name) placeholders in the file name helps to avoid duplicate names when tests are run in parallel [36].

Additionally, unit tests can be executed conditionally by enabling only specific module features. For example, the following command runs the tests with the *file_system* and *context_manager* features enabled:

```
cargo test --features "file_system context_manager"
```

This works because the tests follow the same logic previously mentioned, being compiled only when the corresponding feature is enabled:

```
#[cfg(feature = "context_manager")]
#[test]
Test_MemoryManager_CreateNvmReadWriteRequest(){
    //…
}
```

To generate the coverage report, the *grcov* tool was used. It was first installed in the project using the command:

```
cargo install grcov
```

The *grcov* is used to convert the *.profraw* files generated during test execution into a coverage report in *lcov* format. The output is a *.lcov* file that consolidates all the collected coverage data [36] [37].

Then, the *genhtml* tool is used to transform the *.lcov* file into a HyperText Markup Language (HTML) report. The *--branch-coverage* and *--function-coverage* options ensure that the report includes detailed metrics on branch and function coverage, respectively [37].

5.1.5 Issues Identified During Unit Testing

During the development and execution of unit tests for the *memory_manager* module, a few unexpected behaviours were identified. These problems were identified, analysed, and resolved.

The first bug found was related to the execution of unit tests not finishing. This occurred because *memory_manager* is a cyclic application, designed to run continuously. However, during the tests, this execution prevented the test from being completed, resulting in a block.

To solve this problem, it was necessary to introduce a conditional statement into the main loop, allowing the cycle to be interrupted after the first iteration in a test environment. For that, the following code was added:

```
// Loop while the application check is running
  while unsafe { ES_AL_RunAppCheck() } {
     // Process cyclic operations for the Memory Manager module
     MemoryManager_CyclicOperationsProcess();

      #[cfg(test)]
      break;
}
```

During the execution of unit tests for the file system write operation, an issue was identified related to *MemoryManager_GetAddressPointerAndLength* function. This function is responsible for calculating the memory address and the length of data to be read or written in a file operation, based on the provided parameters.

The declaration of the function is the following:

```
pub fn MemoryManager_GetAddressPointerAndLength(
    NrBlocks: u32,
    StartingOffset: u32,
    LoopCounter: u32,
    CurrentRequest: &mut mm_operation_t,
    StartingBlockPtr: *mut FS_Block,
) -> (address_length_t, address_length_t) {
// ...
}

```

The bug was caused by mistakenly passing the *start_block_ptr* pointer instead of the correct *starting_block_index* to the final parameter of the function. Both pointers were declared in the caller's context, but they pointed to different things: *start_block_ptr* was used for other context and had the value *null*, while *starting_block_index* held the valid reference to the initial file block required by the function.

This kind of error is particularly insidious because it does not produce a compile-time warning, both pointers are of the same type. As a result, the function accessed invalid memory, leading to immediate test failures and incorrect behaviour. After correcting the function call to use *StartingBlockPtr*, the address calculation logic behaved as expected, returning valid addresses and lengths for the write cycles.

5.2 Cross-compiling

Considering that the initial goal of the project was to ensure code portability, all development was directed to allow its execution not only on Linux systems but also on other platforms, including bare metal environments. The choice of environment became obvious, given that it was intended to run validation tests and the available boards, with the chosen one to be the *RTEMS* OS, based on the *SPARC* (*LEON3*) architecture.

The intended target, *sparc-unknown-none-elf*, is only available for *no_std* environments and is categorized as *Tier 3*. This means that support for it is experimental and may

be discontinued or broken at any time. There are also no official buildings provided by the Rust team for this target, which requires manual configuration and for which continuous tests are not performed [38].

Although the documentation recommends using the nightly version, several attempts were made to compile for *SPARC* using the stable version. However, due to persistent errors, it was necessary to resort to nightly running the validation tests, even though this was not the ideal option. It turned out that the *SPARC* compilation process was more complicated than expected, with numerous errors arising along the way that were only resolved after multiple attempts. The first step was to add the nightly version to *rustup* with the following command [39]:

```
1. rustup toolchain add nightly
```

With this, the following toolchains became available, as shown in Figure 21.

```
cblopes@PT-PF3V564G:~/karvel-in-rust/karvel_generic_instance$ rustup show
Default host: x86_64-unknown-linux-gnu
rustup home:  /home/cblopes/.rustup

installed toolchains
--------------------
stable-x86_64-unknown-linux-gnu (active, default)
nightly-x86_64-unknown-linux-gnu
```

Figure 21 - Available toolchains

Since it is a *Tier 3* target, it is not possible to run commands such as:

```
1. rustup target add sparc-unknown-none-elf
```

As mentioned earlier in the Karvel section on toolchains, the available toolchains are in */opt*, , as shown in Figure 22.

```
cblopes@PT-PF3V564G:~$ cd /opt/
cblopes@PT-PF3V564G:/opt$ ls
arm-none-eabi  rtems-6-arm-xilinx_zynq_a9  rtems-6-powerpc-qoriq_e500  rtems-6-sparc-gr712rc-smp-4
containerd     rtems-6-powerpc-qemuppc     rtems-6-riscv-noel32imafd  rtems-6-sparc-ut699-uni-0
```

Figure 22 - Toolchains available in /opt

Initially, a *.cargo/config.toml* file was created solely to avoid having to manually run the following command [38]:

```
1. cargo build --target sparc-unknown-none-elf
```

As previously mentioned, manual configuration is required for *Tier 3* targets. In the *.cargo/config.toml* file, the toolchain components were specified manually using the provided linker provided in */opt* [38].

This allows cargo build to work without needing to specify the *--target* flag every time and enables integration with make as usual.

Here is the configuration that was added to *.cargo/config.toml* [38]:

```
[target.sparc-unknown-none-elf]
linker = "/opt/rtems-6-sparc-gr712rc-smp-4/bin/sparc-rtems6-gcc"

rustflags = [
"-Ctarget-cpu=leon3",
# The linker is a gcc compatible C Compiler
"-Clinker-flavor=gcc",
"-Clink-arg=-mcpu=leon3",
# Rust needs libatomic.a to satisfy Rust's compiler-builtin library
"-Clink-arg=-latomic",
]

[unstable]
build-std = ["core"]
```

The core library is enabled here, which is necessary in *no_std* environments since precompiled standard libraries are not available for *Tier 3* targets. In addition to *.cargo/config.toml*, the *project.vars* file from the Karvel required the following values to ensure correct cross-compilation and simulator configuration:

```
PLATFORM: "gr712rc"
SYSTEM: "sparc-rtems6"
SIMULATOR: "sis"
PLATFORM_INTERFACE: eth_interface
```

With the toolchain installed, it was necessary to switch to the nightly version before building. To do this, the following command was run:

```
rustup override set nightly
```

This ensures that the active toolchain in the project directory is the nightly version, which is required to compile for the *sparc-unknown-none-elf* target, as shown in Figure 23.

```
cblopes@PT-PF3V564G:~/karvel-in-rust/karvel_generic_instance$ rustup show
Default host: x86_64-unknown-linux-gnu
rustup home:  /home/cblopes/.rustup

installed toolchains
--------------------
stable-x86_64-unknown-linux-gnu (default)
nightly-x86_64-unknown-linux-gnu (active)
```

Figure 23 - Nightly toolchain configured for the sparc-unknown-none-elf target

When attempting to run Karvel, the following error occurred:

```
 1. error: failed to run custom build command for `memory_manager_rust v0.1.0
(/home/cblopes/karvel-in-rust/karvel_generic_instance/asw/memory_manager_rust)`
 2.
 3. Caused by:
 4.   process didn't exit successfully: `/home/cblopes/karvel-in-
rust/karvel_generic_instance/build/./cargo/build/debug/build/memory_manager_rust-
3ef6a43b60a3e45a/build-script-build` (exit status: 101)
 5.   --- stderr
 6.   /usr/include/stdint.h:26:10: fatal error: 'bits/libc-header-start.h' file not found
 7.
 8.   thread 'main' panicked at build.rs:53:10:
 9.   Unable to generate input bindings: ClangDiagnostic("/usr/include/stdint.h:26:10: fatal
error: 'bits/libc-header-start.h' file not found\n")
10.   note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
11. gmake[3]: *** [sparc-rtems6-gr712rc/asw/memory_manager_rust/CMakeFiles/_cargo-
build_memory_manager_rust.dir/build.make:70: sparc-rtems6-
gr712rc/asw/memory_manager_rust/CMakeFiles/_cargo-build_memory_manager_rust] Error 101
12. gmake[2]: *** [CMakeFiles/Makefile2:1236: sparc-rtems6-
gr712rc/asw/memory_manager_rust/CMakeFiles/_cargo-build_memory_manager_rust.dir/all] Error 2
13. gmake[1]: *** [Makefile:91: all] Error 2
14. make: *** [Makefile:33: all] Error 2
15.
```

This error occurs because bindgen tries to use the system headers (in this case, the Linux ones), which are not available in the *SPARC* compilation environment.

To solve the problem, it was decided to deactivate the automatic generation of bindings for the *SPARC* architecture, keeping it active only for builds in the Linux environment. To do this, the logic was updated in the *build.rs* file as follows:

```rust
fn main() {
    let target = env::var("TARGET").unwrap();

    if target == "x86_64-unknown-linux-gnu" {

        // => cbindgen
        let crate_dir = env::var("CARGO_MANIFEST_DIR").unwrap();
        let mut config: cbindgen::Config = Default::default();

        config.include_guard = Some("OUT_BINDINGS_H".to_string());
        config.includes = vec!["../../karvel-sys/cheaders/file_system_interface.h".to_string()];
        config.export.include = vec![
            "mm_operation".to_string(),
            "mm_file_copy_move_t".to_string(),
            "mm_file_write_read_t".to_string(),
            "mm_nvm_write_read_t".to_string(),
            "mm_nvm_checksum_t".to_string(),
            "mm_address_copy_t".to_string(),
            "mm_address_checksum_t".to_string(),
            "mm_address_dump_t".to_string(),
            "mm_address_load_t".to_string(),
            "MM_NrRequestCompleted".to_string(),
            "MM_NrRequestReady".to_string(),
            "MM_LastRequest".to_string(),
            "MM_AvailableRequest".to_string(),
        ];
        config.language = cbindgen::Language::C;

        cbindgen::generate_with_config(&crate_dir, config)
            .expect("Unable to generate output bindings")
            .write_to_file("out_bindings.h");

    } else {
        // For sparc targets, we skip the bindgen/cbindgen generation
        // as the target is not supported by bindgen
        // and cbindgen
        println!("cargo:warning=Skipping bindgen/cbindgen for target: {}", target);
    }
}
```

After this change, the code compiled successfully and the **.exe* executable was generated, as shown in Figure 24.

Figure 24 - Successful compilation and generation of the *.exe executable

As intended, the executable can now be loaded into the simulator, allowing the validation tests to be executed. After setting up the simulator, the test application was visualized as shown in Figure 25.

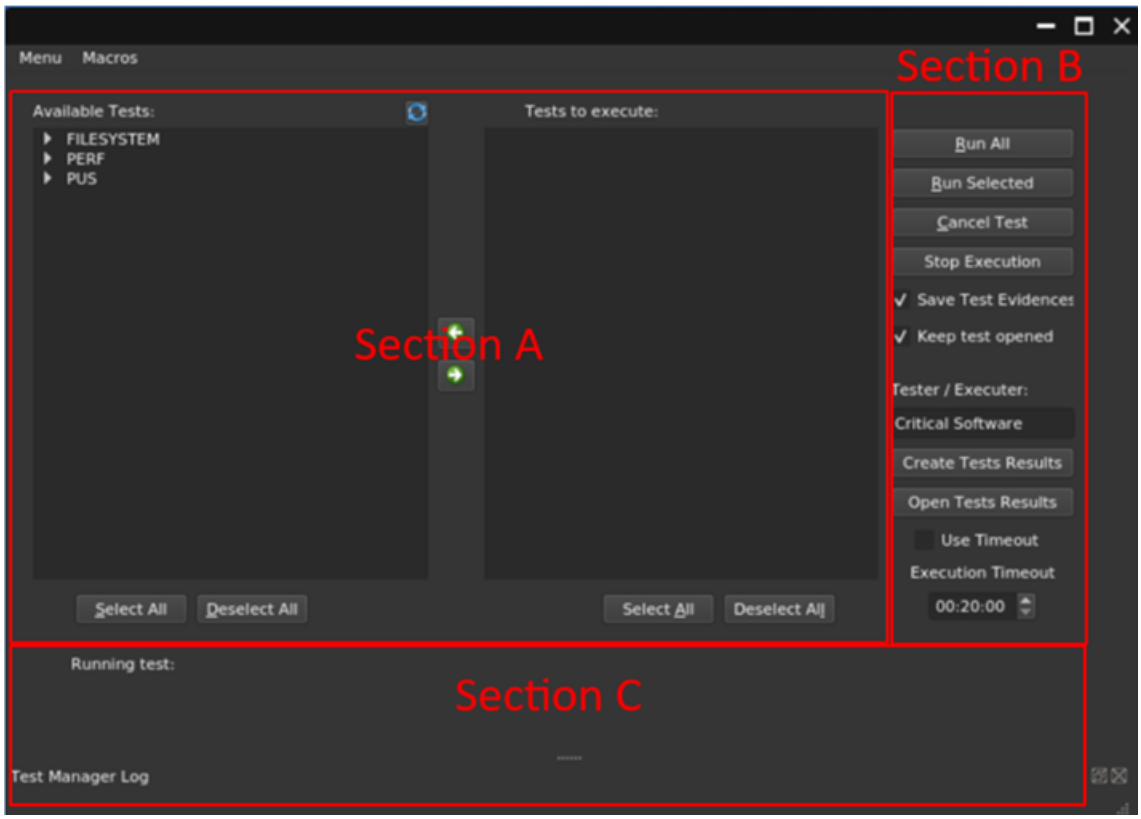


Figure 25 - Sections of the validation simulator interface

It is possible to select the desired tests, in this case services 6[2] [40] and 23[3] [40]. After running the tests and reviewing the generated reports, an issue was identified:

```
1. SIOCGIFADDR: tap0: No such device
```

This problem in the simulator made its use unfeasible, as the issue could not be resolved within the scope of this work. Consequently, an alternative approach was adopted to perform the validation tests.

In Karvel, whenever a pull request is made to the development branch, a Jenkins pipeline is triggered. To enable this, branches were updated and conflicts resolved to create the pull request for development. After a few hours, the pipeline failed during the validation tests step, as shown in Figure 26.

[2] The PUS Service 6, defined in ECSS-E-ST-70-41C, provides subservices for onboard memory management, including memory load, dump, and verification operations, as well as corresponding reports of execution. It is essential for software updates and data management in space missions.
[3] The PUS Service 23, defined in ECSS-E-ST-70-41C, provides onboard file management capabilities, including file upload, download, deletion, and manipulation. It supports missions requiring transfer and management of large data volumes, such as scientific or image data.

Figure 26 - Jenkins pipeline execution and failure during validation tests

Where it ended with the following error:

```
1. Cannot contact karvel-build: hudson.remoting.RequestAbortedException:
java.nio.channels.ClosedChannelException
2. Gracefully stopping... (press Ctrl+C again to force)
3. canceled
4. make: *** [.ci/pipeline/Makefile.docker:102: set-khronosim] Error 130
5. wrapper script does not seem to be touching the log file in /datadrive/PR-41-1@tmp/durable-
dc18322d
6. (JENKINS-48300: if on an extremely laggy filesystem, consider
Dorg.jenkinsci.plugins.durabletask.BourneShellScript.HEARTBEAT_CHECK_INTERVAL=86400)
7. script returned exit code -1
```

This error is not related to the code or testing, but rather to a communication failure between Jenkins and the build. It is considered a sporadic problem external to development. It was necessary to investigate a new alternative.

After several attempts to run the validation tests, both remotely and locally, it was concluded that a different approach would be necessary. It was therefore decided to integrate the module into a project where the validation of services 6 and 23 had already been confirmed as functional.

This required migrating the platform from *SPARC* to *RISCV-RTEMS6* with the *NOEL32IMAFD* architecture and adapting the Rust environment to the new toolchain, which, like the previous one, is also available in the nightly version.

First, it was necessary to add the new architecture to the *project_vars.yml* file.

```
1. PLATFORM: "noel32imafd"
2. SYSTEM: "riscv-rtems6"
```

Next, it was necessary to adapt the *CMakeLists.txt* file of the *memory_manager* module to correctly configure the toolchain, as shown in the example below:

```
 1.  if("${SYSTEM}" STREQUAL "riscv-rtems6" AND "${PLATFORM}" STREQUAL "noel32imafd")
 2.      message(STATUS "Setting Rust toolchain to nightly for ${SYSTEM}-${PLATFORM}")
 3.      execute_process(
 4.          COMMAND rustup override set nightly

 5.          WORKING_DIRECTORY ../ implementation_com_obc

 6.          RESULT_VARIABLE RUSTUP_RESULT
 7.      )
 8.
 9.      if(NOT RUSTUP_RESULT EQUAL 0)
10.          message(FATAL_ERROR "Failed to set rustup override to nightly. Required for riscv
target.")
11.      endif()
12.
13.      corrosion_set_env_vars(memory_manager_rust
14.      "CARGO_BUILD_TARGET=${Rust_CARGO_TARGET}"
15.      "CARGO_TARGET_RISCV_LINKER=/opt/rtems-6-riscv-noel32imafd/bin/riscv-rtems6-gcc"
16.      "CARGO_UNSTABLE_BUILD_STD=true"
17.      )
18.
19.      corrosion_set_cargo_flags(memory_manager_rust "-Zbuild-std=core")
20. endif()
```

After resolving minor conflicts resulting from integration with the mission, the programme began to panic, indicating a possible error on the Rust code side. To debug, Renode with GNU Debugger (GDB*)* was used. It was therefore necessary to prepare the execution environment to allow Renode to be launched with the following command:

```
1. renode --console -e "include @./emulators/renode/config/scripts/hpm-riscv-debug.resc"
```

Through GDB, it was confirmed that the error did happen in the Rust code, since the program went into panic mode during execution. The bug was found in the following line:

```
1. //Get the file address pointer
2.      let mut file_address: address_length_t = unsafe {
3.          (*FAT[mm_file_write_read.WriteReadDataQueue[CurrentRequest.DataIndex as
usize].MemoryId
4.              as usize]
5.              .wrapping_add((*StartingBlockPtr).FATBlockID.try_into().unwrap()))
6.          .BlockPtr
7.      };
8.
```

After analysis, it was found that the cause of the panic was that the *StartingBlockPtr* pointer was *null* value. When attempting to access its *FATBlockID* field, the programme failed, resulting in a panic when attempting to *.unwrap()* an invalid value.

The use of *.unwrap()* was initially adopted because the Rust compiler suggested it in cases where explicit conversion handling was not performed. However, *.unwrap()* should be avoided in the *no_std* context, as it assumes that the value is always present, which is not always guaranteed, as illustrated in Figure 27.

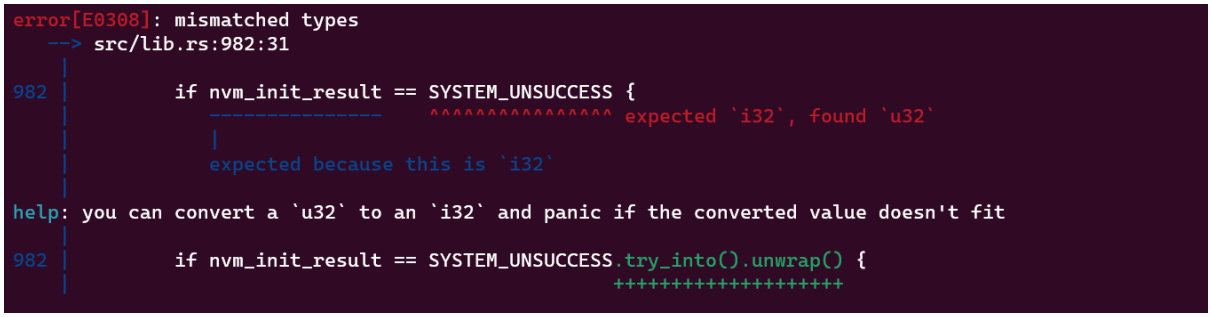


Figure 27 - Compilation error and compiler recommendation regarding *.unwrap()* usage

To resolve the panic issue initially detected, the code was adjusted to check whether the *StartingBlockPtr* pointer was *null* before accessing its content. The new version of the line in question is now:

```
1. 1.  //Get the file address pointer
2.     let mut file_address: address_length_t = if StartingBlockPtr.is_null() {
3.         0
4.     } else {
5.         unsafe {
6.             match (*StartingBlockPtr).FATBlockID.try_into() {
7.                 Ok(fat_block_id) => {
8.                     (*FAT[mm_file_write_read.WriteReadDataQueue[CurrentRequest.DataIndex as
usize]
9.                         .MemoryId as usize]
10.                        .wrapping_add(fat_block_id))
11.                    .BlockPtr
12.                }
13.                Err(_) => 0,
14.            }
15.        }
16.    };
```

However, a second problem also related to *StartingBlockPtr* arose in the following line:

```
1. starting_block_ptr = unsafe { (*starting_block_ptr).Next };
```

Since the pointer could be null, that line was replaced with:

```
1. if !starting_block_ptr.is_null() {
2.     starting_block_ptr = unsafe { (*starting_block_ptr).Next };
3. }
```

With these corrections, the following error appeared on the C code side. It was found that the *file_system* and *context_manager* modules were directly accessing the *memory_manager* request list and making changes. However, this structure, originally implemented as a linked list in C, was converted to a *Vec* in Rust.

This mismatch between the current implementation and the expectations of the C code led to a bug that was difficult to detect and debug. This opportunity was taken to encapsulate and improve the internal logic by reinforcing the isolation between modules.

New functions were then added to the *lib.rs* file that allow services to interact with the list of operations in a controlled manner. In this way, service 6 can iterate through the operation requests without directly accessing the internal structure, leaving that responsibility exclusively to the *memory_manager*.

This encapsulation prevents unintended changes by the C code, ensuring greater robustness and consistency in the data.

```
1.
2. #[no_mangle]
3. pub extern "C" fn MM_GetFirstRequest() -> *const mm_operation_t {
4.     let mm_operation_request = MM_OperationRequest::get_instance();
5.     if !mm_operation_request.MM_OperationRequestQueue.is_empty() {
6.         &mm_operation_request.MM_OperationRequestQueue[0] as *const mm_operation_t
7.     } else {
8.         core::ptr::null()
9.     }
10. }
11.
12. #[no_mangle]
13. pub extern "C" fn MM_UpdateRequestStatus(OpId: u16, new_status: u8) {
14.
15.     let mm_operation_request = MM_OperationRequest::get_instance();
16.
17.     if let Some(operation) = mm_operation_request
18.         .MM_OperationRequestQueue
19.         .iter_mut()
20.         .find(|op| op.OpId == OpId)
21.     {
22.         operation.Status = new_status;
23.
24.     } else {
25.         // println!("Operation with OpId {} not found.", OpId);
26.     }
27. }
28. #[no_mangle]
29. pub extern "C" fn MM_NextRequest(OpId: u16) -> *const mm_operation_t {
30.     let mm_operation_request = MM_OperationRequest::get_instance();
31.     if let Some(pos) = mm_operation_request
32.         .MM_OperationRequestQueue
33.         .iter()
34.         .position(|op| op.OpId == OpId)
35.     {
36.         if pos + 1 < mm_operation_request.MM_OperationRequestQueue.len() {
37.             &mm_operation_request.MM_OperationRequestQueue[pos + 1] as *const mm_operation_t
38.         } else {
39.             core::ptr::null()
40.         }
41.     } else {
42.         core::ptr::null()
43.     }
44. }
```

With the code compilation complete, the binary was run on the board with service 23, subservice 40 test, as this also included some service 6 tests. However, the test failed with the following error:

```
1. Performing EXCEPTION: Packet execution not completed or timeout achieved. Unable to check
responses.
```

This error suggested in service 6 tests that something was not working correctly between service 6 and the *memory_manager* module in Rust, since the tests passed successfully with the C version. As the error was not very descriptive and access to

the board was limited (shared with other team members), it was decided to adopt an alternative approach to facilitate the debugging process.

It was decided to manually send the Telecommand (TCs[4] [41] [42]) used in the validation, using the Renode simulator and the picocom terminal connected to the *UART* device *(/tmp/nanohpm-uart5*). To do this, it was necessary to extract the TCs from the validation report and convert them to a format compatible with Renode, using the *rs422_interface* (rather than *the cants_interface_noel32imafd*, which is not supported by Renode).

TCs are instructions sent from the ground segment to the satellite to perform various operations. These operations can range from simple tasks, such as executing a simple action, to complex sequences involving multiple subsystems. TCs are essential for the control and management of the satellite, enabling ground operators to interact with and configure the satellite's onboard systems in real-time or as scheduled activities. The handling of TCs involves their reception, validation, decoding, and execution, ensuring that the satellite performs the desired actions accurately and efficiently.

To send commands TCs via *UART* in the Renode simulator, each TC must be formatted in a specific structure that Renode can interpret correctly. This structure involves encapsulating the TC data with a defined header and footer, as well as indicating the message size. The process is as follows:

• Original TC example (hexadecimal data):

18 14 C0 7C 00 0F 2F 17 09 00 00 03 44 44 52 04 64 69 72 31 23 8F

This TC has 22 bytes in format expected by Renode.

For Renode to recognise this TC correctly, it must be packaged with the following structure:

\xAA\xAA\x01\<LEN>\<TC_bytes>\x70\x4A

• Where:

• \xAA\xAA – A fixed header marks the beginning of the message.

• \x01 – Channel ID

• <LEN> – Length of TC data, in hexadecimal (in this case x00\x16, because the TC has 22 bytes).

• <TC_bytes> – The actual bytes of the TC.

• \x70\x4A – A fixed footer marks the end of the message.

---

[4] Telecommand (TC): refers to the set of instructions sent from the ground segment to a spacecraft or space system, intended to control, configure, or command its operations. Telecommands are encoded, validated, and transmitted according to standardized protocols to ensure operational integrity and security.

Complete command line example:

```
1. printf"\xAA\xAA\x01\x00\x16\x18\x14\xC0\x7C\x00\x0F\
x2F\x17\x09\x00\x00\x03\x44\x44\x52\x04\x64\x69\x72\x31\x23\x8F\x70\x4A" >> /tmp/nanohpm-uart4
```

This process made it possible to identify the problem with the help of GDB in Renode. It was found that requests from service 6 were not being changed to *completed* status, causing them to accumulate until exceeding the maximum size of the internal array. This led to the exception that was detected during execution.

Further analysis revealed that the tests still followed the previous logic, which was based on the condition that a request would be considered as completed:

```
1.  if ((TaskPtr->Task.RequestId - TaskPtr->Task.RequestCounter) == current_request->OpId)
```

In this logic, *RequestId* corresponds to the main request identifier, while *RequestCounter* represents the number of operations that still need to be completed. This formula is used to calculate the *OpId* of the completed operation. However, the new implementation introduces changes that mean the *OpId* is no longer managed by *pus_6* but coincides directly with the *RequestId* of service 6. This direct correspondence removes the need for the previous calculation, making this check redundant and causing incorrect behaviour in the tests.

After resolving the previous problem, the test continued to fail. However, it was found that the report generated was identical to the C version, indicating that the behaviour was aligned. Next, tests for service 6 were evaluated, revealing that TCs 6,6 and 6,10 had failed with a *wrong length* error. It was discovered that the Telemetry (TM[5] [41] [42]) received from the memory manager was the incorrect size. Analysis revealed that the request was being sent incomplete due to the global variables *current_request_tm_size* and *current_request_tm_index* being changed to local. This occurred because two requests were generated for the same order, one with the *LastRequest* flag set to *false* and the other to *true*. However, as the global variables were used to accumulate the size and index each time the function was called, they were reset to zero each time, which was not the intended result. Consequently, the information from the two requests was not accumulated, and only the size and index of the second request (with *LastRequest* = *true*) were stored and sent to Service 6. Consequently, the TM was generated with only the final information, ignoring the partially processed content, resulting in an incorrect size. The following code illustrates the described logic:

[5] Telemetry (TM) refers to the process of collecting, encoding, transmitting, and receiving data from a space system (such as a satellite) for the purpose of monitoring its status and performance.

```
1. fn execute_checksum_operation(CurrentRequest: &mm_operation){
 2.
 3.    let mut current_request_tm_size: u32 = 0;
 4.      let mut current_request_tm_index: u32 = 0;
 5.
 6.     (…)
 7. if mm_address_checksum_t.ChecksumDataQueue[CurrentRequest.DataIndex as usize].LastInRequest
{
 8.          unsafe {
 9.              PUS_SVC_6_9_Send_TM(
10.                  &mut CURRENT_REQUEST_TM_SIZE as *mut _,
11.                  &mut CURRENT_REQUEST_TM_INDEX as *mut _,
12.                  mm_address_checksum_t.ChecksumDataQueue[CurrentRequest.DataIndex as
usize].SourceId,
13.)};} }
```

Following the completion of the validation phase, unit tests ensured full coverage (100%). After resolving the issue identified during the validation tests, these tests were executed, showing results consistent with those obtained in C tests, although not all cases were successful. At this point, the memory manager module in Rust is fully tested and functioning as expected. This allows proceeding to the comparison phase between Karvel with a Rust module and Karvel implemented in C, as both modules are now in equivalent conditions for analysis.

6 MEMORY MANAGER COMPARISON: C AND RUST IMPLEMENTATIONS

This chapter presents a comparison between the memory manager module implemented in C and in Rust. The analysis covers several aspects, including static analysis, code size and complexity metrics, execution profiling, and total execution times.

6.1 Static analysis

A static analysis was performed on the C code to identify errors, and to check if these errors also manifested themselves in the Rust version, or if they were mitigated or eliminated by the language's guarantees.

Additionally, a direct comparison was made between the two codes in terms of the number of lines of code, structure and complexity.

6.1.1 Line Count

The Tokei tool was used to analyse the size and complexity of the C and Rust code. Understanding these metrics is essential in critical systems, as more complex or extensive code increases the risk of errors and makes verification and maintenance more difficult **[43] [44]**. The results for C and Rust are shown in Figures 28 and 29, respectively.

Result in C:

```
cblopes@PT-PF3V564G:~/ioendurance/implementation_com_obc/asw$ tokei memory_manager
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 C                       6         3267         1721         1098          448
 C Header                3          219           71          119           29
 CMake                   1           45           42            0            3
 Markdown                1           22            0           15            7
 ReStructuredText        4          462          336            0          126
===============================================================================
 Total                  15         4015         2170         1232          613
===============================================================================
```

Figure 28 - Code size and complexity metrics for the C implementation

Result in Rust:

```
cblopes@PT-PF3V564G:~/ioendurance/implementation_com_obc/asw$ tokei memory_manager_rust/
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 C Header                1          311          200           21           90
 CMake                   1          102           57           22           23
 TOML                    1           38           27            2            9
-------------------------------------------------------------------------------
 Rust                   11         6310         3948         1259         1103
 |- Markdown             2            3            0            3            0
 (Total)                            6313         3948         1262         1103
===============================================================================
 Total                  14         6761         4232         1304         1225
===============================================================================
```

Figure 29 - Code size and complexity metrics for the Rust implementation

Analysis of Results

The analysis shows that the Rust implementation almost doubles the number of lines of code compared to the C implementation (4.232 vs. 2.170). This is mainly because, in C, variable access is simpler and more direct, whereas Rust enforces stricter access controls. In C, global variables can be freely accessed and modified, while in Rust, access is controlled through traits and the Singleton pattern. Consequently, in C, other modules can directly manipulate the linked list, whereas the Rust memory manager module exposes only public methods, encapsulating internal structures and preventing unintended external modifications.

Furthermore, Rust enforces checks that are not required in C, enhancing robustness. This comparison highlights a clear distinction between the two languages: C favours expressive efficiency, resulting in more compact but less safe code, while Rust is more verbose, providing greater safety, robustness, and better structural organization.

6.1.2 Complexity analysis

To provide a comprehensive evaluation of the code, its complexity was measured using the Lizard tool. This tool was applied to both the memory manager module in C and its equivalent in Rust. This process served to complement the static analysis. The objective of this analysis is to evaluate cyclomatic complexity, number of lines of code (NLOC), number of parameters, and average function size. These factors are directly related to maintainability and susceptibility to errors [45].

Results in C

The execution of Lizard on the memory manager in C involved 26 functions, distributed across 7 files, with a total of 795 NLOC, an average cyclomatic complexity (CCN) of 4.4, and an average function length of 26.8 lines. Even though

none of the thresholds defined by the tool were exceeded (i.e. complexity > 15, length > 1000, or NLOC > 1,000,000), two functions are worthy of particular attention due to their high cyclomatic complexity:

- *MemoryManager_ExecuteRequest* (CCN = 14, 58 lines)
- *MemoryManager_FreeRequest* (CCN = 14, 52 lines)

These functions, which are associated with memory request management, exhibit more complex control structures and therefore carry a higher risk of errors and greater maintenance difficulty. The results of this study are presented in Annex C.

Result in Rust

In the Rust implementation, the memory manager contained 45 functions across 7 files, with a total of 1,956 NLOC, an average cyclomatic complexity of 3.1, and an average function length of 36 lines. In this instance, the established complexity thresholds were not exceeded. The functions with the highest complexity were:

- *MemoryManager_ExecuteRequest* (CCN = 14, 75 lines)
- *MemoryManager_FreeRequest* (CCN = 13, 70 lines)

Although these functions have values similar to the C version, the overall average cyclomatic complexity in Rust is lower (3.1 vs. 4.4), suggesting a better distribution of logic among functions. This difference may be related to the implementation choices adopted. For example, in Rust it was possible to encapsulate behaviours using traits, a feature not available in C, which helped reduce individual function complexity and increase code modularity. The results of this study are presented in Annex D.

6.2 Renode Profiling

One of the ways used to compare the implementations in C and Rust was through profiling provided by Renode. As the main objective was to activate the *memory_manager* module, it was decided to send the TCs from services 6 and 23.

As the number of TCs was high, a Python script was created that through the test reports (in HTML format) and searches for the TCs, placing them in the previously defined format so that Renode can interpret them correctly, as mentioned in the previous section.

```python
from bs4 import BeautifulSoup
import re
from pathlib import Path

def generate_uart_command(hex_data_str):
    hex_data = bytes.fromhex(hex_data_str.strip())
    prefix = b"\xAA\xAA\x01"
    length = len(hex_data).to_bytes(2, byteorder='big')
    suffix = b"\x70\x4A"
    full_cmd = prefix + length + hex_data + suffix
    return 'printf "' + ''.join(f"\\x{b:02X}" for b in full_cmd) + '" >> /tmp/nanohpm-uart4'

def extract_commands_from_html(html_path):
    html = Path(html_path).read_text(errors="ignore")
    soup = BeautifulSoup(html, "html.parser")
    rows = soup.find_all("tr")
    commands = []
    seen_hex_data = set() # Avoid duplicates

    for row in rows:
        cells = row.find_all("td")
        texts = [cell.get_text(strip=True) for cell in cells]

        # Check if any cell contains 'DATA_FROM_TEST.PAYLOAD'
        payload_found = False
        for text in texts:
            if 'DATA_FROM_TEST.PAYLOAD' in text:
                payload_found = True
                break

        if payload_found:
            # Look for hex data in any cell of this row
            for text in texts:
                # Match hex patterns like "18 14 C0 7C 00 0F 2F 17 09 00 00 03 44 44 52 04
64 69 72 31 23 8F"
                if re.fullmatch(r'([0-9A-Fa-f]{2}\s*)+', text.strip()):
                    hex_data = text.strip()
                    if hex_data not in seen_hex_data:  # Avoid duplicates
                        seen_hex_data.add(hex_data)
                        cmd = generate_uart_command(hex_data)
                        commands.append(cmd)
                    break

    return commands

if __name__ == "__main__":
    import sys
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage: python extract_uart_commands.py REPORT_FILE.html [OUTPUT_FILE.sh]")
        print("If OUTPUT_FILE.sh is not specified, 'commands.sh' will be used")
        exit(1)

    html_file = sys.argv[1]
    output_file = sys.argv[2] if len(sys.argv) == 3 else "commands.sh"

    commands = extract_commands_from_html(html_file)
    with open(output_file, "w") as f:
        f.write("#!/bin/bash\n\n")
        for cmd in commands:
            f.write(cmd + "\n")

    print(f"{len(commands)} commands saved to {output_file}")
```

The generated script allows the TC's to be sent sequentially, avoiding manual work. It receives as parameters the name of the report in HTML format, and optionally the name of the script to be generated. The script is executed as follows:

```
python3 extract_uart_commands.py s6_10_C.html s6_10_C.sh
```

The output confirms the number of TCs extracted:

```
9 commands saved to s6_10_C.sh
```

The resulting *s6_10_C.sh* file includes the *printf* commands with the correctly formatted TCs:

```
#!/bin/bash
printf
"\xAA\xAA\x01\x00\x1D\x18\x14\xC0\x7C\x00\x16\x2F\x06\x02\x00\x00\x01\x01\x01\x00\x00\x00\x00\x00\x00\x04\x0D\x08\x22\x6A\xB9\x8F\xDE\x03\x70\x4A" >> /tmp/nanohpm-uart4
printf
"\xAA\xAA\x01\x00\x17\x18\x14\xC0\x83\x00\x10\x2F\x06\x05\x00\x00\x01\x01\x01\x00\x00\x00\x00\x00\x01\xED\x09\x4F\x70\x4A" >> /tmp/nanohpm-uart4
(…)
```

After generating the scripts containing the TC's to be sent, the test environment where Renode would be executed was defined. The objective was to simulate the execution of the system to collect performance metrics for later comparison between implementations in C and Rust [46].

```
renode --console -e "include @./emulators/renode/config/scripts/hpm-riscv-debug.resc"
```

With the environment loaded, the profiling system was activated on the Renode terminal [46]:

```
(machine-0)  machine EnableProfiler "/home/cblopes/ioendurance/Result_Rust/S6_10.dump"
```

Next, the system was started up [46].

```
(machine-0) start
```

On a second terminal, the previously generated script was executed, which contained the relevant TCs:

```
./s6_10.sh
```

After approximately 10 seconds of execution, considered sufficient time for sending and processing the TCs, the simulation was completed:

```
(machine-0) quit
```

With the *.dump* files generated, the metrics were analysed using the metrics-visualizer tool provided by Renode. This tool allows graphs to be generated from profiling, facilitating comparison of behaviour between the C and Rust versions of the system [46].

The graphs were generated using the following command:

```
python3 tools/metrics_analyzer/metrics_visualizer/metrics-visualizer.py   --no-dialogs -o
/home/cblopes/ioendurance/output_dir   /home/cblopes/ioendurance/S6_10.dump
```

The graphs obtained are presented in Annex E. However, no significant differences can be observed between them. This outcome can be explained by the fact that Karvel is composed of multiple modules, meaning that a modification limited to the memory manager module does not have a noticeable impact on the overall system behaviour.

6.3 Total execution time

The execution time of the validation tests was also considered in the comparative analysis between the C and Rust implementations. For each test, the total execution time and the respective result (Pass or Fail) were recorded, allowing us to evaluate not only performance but also functional consistency between the two versions, as summarized in Table 2.

Table 2 - Execution times and results of validation tests for C and Rust implementations

Test	C		Rust	
-	Execution Duration (Minutes: seconds. milliseconds)	Overall Result	Execution Duration (Minutes: seconds. milliseconds)	Overall Result
S6_0010	00:27.205	Pass	00:23.213	Pass
S6_0012	00:29.202	Pass	00:25.204	Pass
S6_0014	00:23.206	Pass	00:23.212	Pass
S6_0020	00:29.206	Pass	00:31.206	Pass
S23_0040	01:13.207	Pass	01:09.212	Pass

In general, the execution times of the Rust version were slightly lower than those of the C version. However, a higher time was observed for test *S6_0020*. It is important to note that the functional results (pass/fail) were consistent for both versions, which reinforces the reliability of the Rust reimplementation for validation testing.

7 CONCLUSION

The objective of this chapter is to provide a comprehensive reflection on the work conducted during the internship at Critical Software, discussing the main challenges and difficulties encountered, the key findings regarding the use of Rust within Karvel and in space-critical systems, and outlining potential directions for future development and research.

With the conclusion of this internship, Karvel now includes a memory manager module implemented in Rust. This module is not part of the main development branch but is maintained in a dedicated branch for the Rust implementation, due to the limitations that will be discussed in this chapter. The module has been tested and compared with its C counterpart, generally showing good results when compared to the C implementation.

7.1 Challenges and Difficulties

The use of Rust constituted one of the main challenges of this work, given that the language was initially unknown and required a steep learning curve. At the same time, development for embedded systems was also a new area, which added an additional layer of complexity. The intrinsic complexity of Karvel, characterized by extensive codebase, also represented a significant barrier. Moreover, in the initial phase of the project, Karvel was undergoing structural changes which brought moments of instability, resulting in compilation errors that delayed the start of the work.

After overcoming these initial difficulties, new obstacles arose during the validation testing phase, causing delays and hindering the integration of the solutions developed, compromising the smoothness of the validation process. The instability of the platform required the reformulation of approaches, delaying the progress of the work and limiting the depth and consistency of the comparative results obtained.

Despite these adversities, the work undertaken allowed for the acquisition of valuable knowledge in Rust, embedded development, and integration of complex systems, while at the same time providing Critical Software with a fully functional integration between C and Rust, designed in a modular way to allow the addition of new modules if needed, with the memory manager serving as a practical example.

7.2 Rust in Critical Systems and in Karvel

Rust has proven to be a promising language, but still relatively immature in the space context. One of its main strengths lies in the rigorous checks enforced by the compiler, which require the programmer to consider all possible scenarios. This

mechanism directly contributes to reducing typical C errors, increasing software safety and reliability.

Within the scope of this internship, progress could have been smoother if the delays related to the validation tests had not taken so long to overcome. This delay directly affected the possibility of performing more detailed and comprehensive comparisons. It also became evident that the Karvel platform itself, due to its high complexity, has not yet reached sufficient maturity to fully support a smooth integration of Rust.

During the investigation, it was observed that Rust preserves safety by enforcing robust programming practices that C does not impose. However, integration with Karvel revealed some challenges, particularly regarding the use of global variables and pointers, areas where Rust does not simplify the process, but which simultaneously demonstrate its conceptual advantage over C, by discouraging practices prone to errors.

Nevertheless, there are disadvantages that limit the immediate adoption of Rust in space systems. On one hand, Rust is still not officially recognized in the standards for space software development, such as the ECSS standards, which continue to favour C and Ada. This lack of formal recognition restricts its use in space missions. On the other hand, its relative immaturity sometimes requires the use of features available only in nightly, which is not recommended for critical systems. For Rust to compete on equal footing with C, it is essential that its most relevant features become available in stable versions.

In summary, Rust has proven to be a safe and promising language, capable of significantly reducing common C errors and enhancing the reliability of critical systems. However, integration with the Karvel platform still faces practical challenges, as the platform's current maturity limits the systematic conversion of modules to Rust. The study demonstrated that, although Rust has great potential to enhance the safety and robustness of Karvel, its use in critical space missions still requires formal validation and platform adaptation.

7.3 Future work

As a future work perspective, the implementation of unit tests in Rust using the existing C *stubs* is highlighted, eliminating the need for duplication and allowing direct reuse of the previously defined stubs. Regarding the *karvel-sys* crate, it will be necessary to optimize the creation of wrappers, which is currently performed manually due to the dispersion of Karvel headers across multiple directories. Future reorganization of these headers will allow this process to be automated, removing the need to manually duplicate headers into a specific folder.

Another relevant objective is the adaptation of the infrastructure to enable validation on alternative platforms, such as *RTEMS* on the *LEON3* architecture and Linux, complementing the currently used Renode environment. This expansion would allow leveraging internal Karvel tools, such as the *performance_app*, providing additional performance metrics, extremely important for critical systems, that were not possible within the scope of this study.

REFERENCES

[1] DLR - German Aerospace Center (Prime), “cRustacea in Space - Co-operative RUST and C embedded applications,” *cRustacea in Space - Co-operative RUST and C embedded applications,* p. 5, 31 Jul 2024.

[2] L. S. a. J. Beier, “Bringing Rust to Safety-Critical Systems in Space,” *Bringing Rust to Safety-Critical Systems in Space,* p. 8, 28 May 2024.

[3] G. Gross, “White House urges developers to dump C and C++,” InfoWorld, 27 Fev 2024. [Online]. Available: https://www.infoworld.com/article/2336216/white-house-urges-developers-to-dump-c-and-c.html.

[4] Critical Software, “We are Critical Software,” [Online]. Available: https://criticalsoftware.com/en/our-world/culture-history.

[5] geeksforgeeks, “Stack vs Heap Memory Allocation,” geeksforgeeks, 26 Feb 2025. [Online]. Available: https://www.geeksforgeeks.org/dsa/stack-vs-heap-memory-allocation/.

[6] S. Romanazzi, “From Manual Memory Management to Garbage Collection,” *From Manual Memory Management to Garbage Collection,* p. 20, Jul 2018.

[7] C. N. a. C. K. w. c. f. t. R. C. Steve Klabnik, “4. Understanding Ownership,” in *The Rust Programming Language*, Online, 2018, p. 554.

[8] C. N. a. C. K. w. c. f. t. R. C. Steve Klabnik, “10. Generic Types, Traits, and Lifetimes,” in *The Rust Programming Language*, Online, 2018, p. 554.

[9] NASA, “Rust in cFS: Prevent Bugs with Memory-Safe Programming,” TechPort, 2021 Sep 2021. [Online]. Available: https://techport.nasa.gov/projects/96767.

[10] N7 Space, “Evaluation of Rust usage in space applications by developing BSP,” *Evaluation of Rust usage in space applications by developing BSP,* p. 10, 07 Nov 2023.

[11] ESA - European Space Agency, “ClearSpace-1,” ESA - European Space Agency, n.d. n.d. n.d.. [Online]. Available: https://www.esa.int/Space_Safety/ClearSpace-1.

[12] Kitware, “CMkae,” cmake.org, 2024. [Online]. Available: https://cmake.org/.

[13] J. V. Stoep, “ Security Blog,” Google, 1 Dec 2022. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.

[14] O. Hiari, “Rust FFI and bindgen: Integrating Embedded C Code in Rust,” The Embedded Rustacean, 16 Jan 2023. [Online]. Available:

https://blog.theembeddedrustacean.com/rust-ffi-and-bindgen-integrating-embedded-c-code-in-rust#heading-introduction.

[15] A. Gaspar, “Corrosion documentation,” [Online]. Available: https://corrosion-rs.github.io/corrosion/.

[16] S. Colbrooke, “The mangle in Rust language,” Medium, 29 Jul 2023. [Online]. Available: https://medium.com/@comsamtom/the-mangle-in-rust-language-512ee113dd73.

[17] R. Community, “Linkage,” The Rust Reference, n.d. n.d. n.d.. [Online]. Available: https://doc.rust-lang.org/reference/linkage.html.

[18] Rust Community, “Module std::os::raw,” n.d. n.d. n.d.. [Online]. Available: https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/std/os/raw/index.html.

[19] Rust Community, “Crate cxx,” docs.rs, n.d. n.d. n.d.. [Online]. Available: https://docs.rs/cxx/latest/cxx/.

[20] Rust Community, “CXX — safe interop between Rust and C++,” cxx.rs, n.d. n.d. n.d.. [Online]. Available: https://cxx.rs/#cxx--safe-interop-between-rust-and-c.

[21] Rust Community, “cxx-rs github,” n.d. n.d. n.d.. [Online]. Available: https://github.com/wmatthews-google/cxx-rs.

[22] Rust Community, “Crate libloading,” docs.rs, n.d. n.d. n.d.. [Online]. Available: https://docs.rs/libloading/latest/libloading/.

[23] Rust community, “Cargo Reference,” The Cargo Book, n.d. n.d. n.d.. [Online]. Available: https://doc.rust-lang.org/cargo/reference/manifest.html.

[24] Rust Community, “Library Usage with build.rs,” The bindgen User Guide, n.d n.d n.d. [Online]. Available: https://rust-lang.github.io/rust-bindgen/library-usage.html.

[25] Rust Community, “cbindgen User Guide github,” n.d. n.d. n.d.. [Online]. Available: https://github.com/mozilla/cbindgen/blob/main/docs.md.

[26] Rust Community, “no_std Rust Environment,” The Embedded Rust Book, n.d. n.d. n.d.. [Online]. Available: https://docs.rust-embedded.org/book/intro/no-std.html.

[27] Rust Community, “The smallest #![no_std] program,” The Embedonomicon, n.d. n.d. n.d.. [Online]. Available: https://docs.rust-embedded.org/embedonomicon/smallest-no-std.html.

[28] Rust Community, “Crate panic_halt,” docs.rs, n.d. n.d. n.d.. [Online]. Available: https://docs.rs/panic-halt/1.0.0/panic_halt/.

[29] F. GA, “Learning Rust: 15 - How you can organize your Rust code with "Modules",” Dev.to, 1 Oct 2023. [Online]. Available: https://dev.to/fadygrab/learning-rust-15-how-you-can-organize-you-rust-code-with-modules-2c28.

[30] Rust Community, “Crate heapless,” docs.rs, n.d. n.d. n.d.. [Online]. Available: https://docs.rs/heapless/latest/heapless/.

[31] Rust Community, “Singletons,” The Embedded Rust Book, n.d. n.d. n.d.. [Online]. Available: https://docs.rust-embedded.org/book/peripherals/singletons.html.

[32] Rust Community, “cfg,” Rust By Example, n.d. n.d. n.d.. [Online]. Available: https://doc.rust-lang.org/rust-by-example/attribute/cfg.html.

[33] The kernel development community, “linux github,” github, n.d. n.d. n.d.. [Online]. Available: https://github.com/torvalds/linux.

[34] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “11. Writing Automated Tests,” in *The Rust Programming Language*, Online, n.d., p. 554.

[35] J. Lingle, “rusty-fork,” crates.io, 2018. [Online]. Available: https://crates.io/crates/rusty-fork.

[36] G. Ganesh, “Robust Rust: How Code Coverage Powers Rust Software Quality,” 13 Feb 2024. [Online]. Available: https://medium.com/@gnanaganesh/robust-rust-how-code-coverage-powers-rust-software-quality-417ef3ac2360.

[37] Rust Community, “grcov github,” github, n.d. n.d. n.d.. [Online]. Available: https://github.com/mozilla/grcov.

[38] Rust Community, “Platform Support,” The rustc book, n.d. n.d. n.d.. [Online]. Available: https://doc.rust-lang.org/nightly/rustc/platform-support.html.

[39] docs.rtems.org, “Bare Metal Rust with RTEMS,” docs.rtems.org, 22 Jan 2025.. [Online]. Available: https://docs.rtems.org/docs/6.1/user/rust/bare-metal.html#build-and-run-on-sparc.

[40] ECSS, “Space engineering Telemetry and telecommand packet,” ECSS, 2016 Apr 2016. [Online]. Available: https://ecss.nl/wp-content/uploads/2016/06/ECSS-E-ST-70-41C15April2016.pdf.

[41] ESA, “Telemetry & Telecommand,” ESA, n.d. n.d. n.d.. [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Telemetry_Telecommand.

[42] ECSS, “Space engineering Space segment operability,” ECSS, 5 JuL 2024. [Online]. Available: https://ecss.nl/wp-content/uploads/2024/07/ECSS-E-ST-70-11C-Rev.1-DIR1(5July2024).pdf.

[43] Rust Community, “tokei github,” tokei, n.d. n.d. n.d.. [Online]. Available: https://github.com/XAMPPRocky/tokei.

[44] Rust Community, “Crate tokei,” docs.rs, n.d. n.d. n.d.. [Online]. Available: https://docs.rs/tokei/latest/tokei/.

[45] Rust Community, “lizard github,” github, n.d. n.d. n.d.. [Online]. Available: https://github.com/terryyin/lizard.

[46] Renode, “Metrics analyzer,” Renode, 17 Oct 2025. [Online]. Available: https://renode.readthedocs.io/en/latest/basic/metrics.html.

[47] Rust, “Rust,” Rust-lang, n.d. n.d. n.d.. [Online]. Available: https://rust-lang.org/.

[48] A. Perugini, “Evaluation of the Parallel,” *Evaluation of the Parallel,* p. 95, Oct October.

[49] Rust, “Install Rust,” Rust-lang, n.d. n.d. n.d.. [Online]. Available: https://rust-lang.org/tools/install/.

[50] Rust Community, “The Manifest Format,” The Cargo Book, n.d. n.d. n.d.. [Online]. Available: https://doc.rust-lang.org/cargo/reference/manifest.html.

[51] A. Merced, “Introduction to Cargo and cargo.toml,” Dev, 5 Nov 2024. [Online]. Available: https://dev.to/alexmercedcoder/introduction-to-cargo-and-cargotoml-2l86.

[52] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “3. Common Programming Concepts,” in *The Rust Programming Language*, Online, 2018, p. 554.

[53] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “6. Enums and Pattern Matching,” in *The Rust Programming Language*, Online, 2018, p. 554.

[54] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “9. Error Handling,” in *The Rust Programming Language*, Online, 2018, p. 554.

[55] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “19. Patterns and Matching,” in *The Rust Programming Language*, Online, 2018, p. 554.

[56] C. N. a. C. K. w. c. f. t. R. C. by Steve Klabnik, “5. Using Structs to Structure Related Data,” in *The Rust Programming Language*, Online, 2018, p. 554.

ANNEXES

Annexes

ANNEX A: INTERNSHIP PROPOSAL

Investigação e aplicação da linguagem de programação Rust em software embebido de naves espaciais

SUMÁRIO

Rust é uma linguagem de programação desenhada com o objetivo de ser "segura, concorrente e prática", mas diferente de outras linguagens seguras, por exemplo, Rust não usa “garbage colector”, possui suporte nativo ao WebAssembly, entre outras funcionalidades.

A linguagem de programação Rust baseia-se nos seguintes princípios: segurança sem coletor de lixo, concorrência sem disputa de dados e abstração sem overhead. Estes princípios fazem com que Rust seja uma opção rápida para ser usada não só em aplicações de baixo nível bem como em projetos de alto nível.

Muitas das piores vulnerabilidades de segurança de sempre tiveram como causa principal problemas com a segurança da memória. As linguagens de baixo nível mais antigas dão aos programadores muito poder, mas isso aumenta o risco de um código com erros causar graves repercussões. Independentemente disso, a utilização de linguagens com segurança de memória - como Rust, Python e JavaScript - tem vindo a aumentar.

Neste ano de 2024, Rust foi notícia (website: https://www.whitehouse.gov/oncd/briefingroom/2024/02/26/press-release-technical-report/) quando uma agência governamental dos EUA, a ONCD (*Office of the National Cyber Director*), decretou que os programadores devem usar linguagens seguras em termos de memória, como Rust, ao invés de C e C++.

A exploração e utilização do Espaço está a sofrer uma transformação, caracterizada por uma diversidade crescente de naves espaciais, que vão desde sistemas grandes e sofisticados como estações espaciais, constelações ou telescópios espaciais, até plataformas mais pequenas, mais acessíveis e flexíveis, como os CubeSats.

Estes satélites compactos, particularmente predominantes na Órbita Terrestre Baixa (LEO), exemplificam a tendência para a miniaturização e democratização da tecnologia espacial. Com uma acessibilidade significativamente mais elevada e ciclos de desenvolvimento mais curtos, levaram à duplicação do número de CubeSats nos últimos 4-5 anos. Muitas naves espaciais dependem de software desenvolvido em linguagem C, que é conhecida pelo seu desempenho e ecossistema maduro, mas não pelas suas características de segurança. Esta confiança levanta preocupações críticas, especialmente porque estes sistemas desempenham papéis cada vez mais vitais na investigação científica, na comunicação e na observação da Terra.

OBJECTIVO DO PROJECTO DE ESTÁGIO

O objetivo deste projeto de estágio passa pela inclusão do estagiário na investigação sobre a utilização da linguagem Rust em software embebido de naves espaciais.

Numa primeira fase, a ideia será o aluno investigar as particularidades da linguagem de programação Rust e fazer uma apresentação dos resultados.

Numa segunda fase, a ideia será o aluno analisar a arquitetura e design do software de controlo de naves espaciais da CRITICAL "KARVEL" (implementada em linguagem de programação C).

Numa terceira fase, a ideia será o aluno desenvolver funcionalidades do "KARVEL" utilizando a linguagem de programação Rust.

Por fim deverá ser feita uma análise das duas implementações (i.e., linguagem C vs Rust) a vários níveis como por exemplo: número de linhas de código utilizadas, complexidade do código, análise da performance, análise da utilização de memória, e será necessário escrever um relatório desta análise.

PLANO DE TRABALHOS

As principais atividades a desenvolver no âmbito do projeto de dissertação são:

- T1: Análise da linguagem de programação Rust;
- T2: Análise da arquitetura e design do "KARVEL";
- T3: Desenvolver funcionalidades do "KARVEL" em linguagem de programação Rust;
- T4: Comparar as duas implementações (C e Rust) do "KARVEL";
- T5: Escrita da dissertação sobre os temas acima abordados, assim como a preparação de duas apresentações anexas, uma para a avaliação académica e outra, mais técnica, para apresentação interna na Critical Software.

PERFIL DO ESTAGIÁRIO

A Critical Software procura um aluno que apresente conhecimentos base em linguagens de programação Rust, C e Python bem como tecnologias como o UML. Este deve também ter o conhecimento dos processos de desenvolvimento de software em metodologias Agile e Waterfall. É considerado uma mais-valia se o candidato estiver familiarizado com uma, ou mais, das normas: ECSS e MISRA.
Na escolha do candidato, a Critical Software tem em conta, não só as suas competências técnicas como as competências comportamentais, sendo que as duas categorias de competências são avaliadas com uma importância semelhante. Espera-se igualmente que o candidato esteja motivado para integrar o programa formativo e de acompanhamento proposto pela Critical Software.

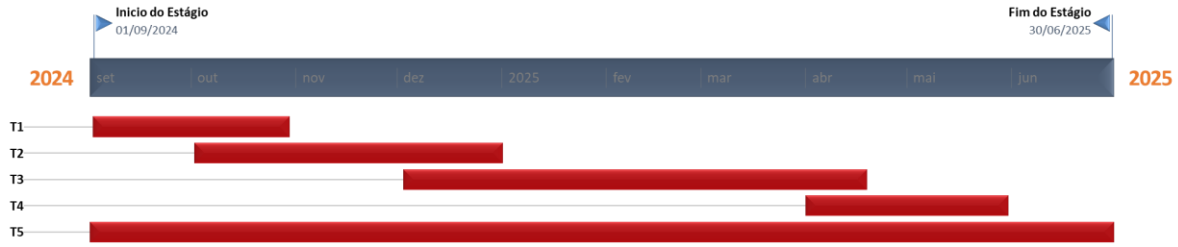
Note-se que é condição para a atribuição do projeto de dissertação a realização de uma entrevista com os candidatos interessados.

DURAÇÃO E CALENDÁRIO

O início, duração e fim de atividades são flexíveis de forma a adaptar-se às condições definidas pela instituição de ensino e serão acordados entre a Critical Software e a

Instituição de Ensino, atendendo em primeiro lugar aos imperativos do calendário escolar.

Para efeitos de planeamento e escalonamento preliminar será assumida uma duração de 6 meses propondo o seguinte calendário:



LOCAL DE TRABALHO

A dissertação será executada nas instalações da empresa Critical Software em Coimbra, Porto ou Lisboa. Poderá ser necessário ter capacidade de deslocação pontual entre sites da Critical Software em Portugal para algumas atividades.

ORIENTAÇÃO

O projeto de Dissertação será orientado por um engenheiro da Critical Software em complementaridade à orientação fornecida pela instituição de ensino.

O aluno integrará a equipa Critical Software tendo acesso a todo o programa formativo, de acompanhamento e de avaliação de desempenho inerente a este programa.

CONFIDENCIALIDADE

A informação transmitida pela Critical Software no âmbito do projeto de Dissertação, incluindo documentos técnicos ou de gestão, diagramas, código ou outra informação relevante deve ser tratada com a máxima confidencialidade. O candidato a quem for atribuído o projeto de Dissertação deve assinar um acordo de obrigação de confidencialidade (NDA, Non Disclosure Agreement).

ANNEX B: RUST LANGUAGE

The Rust Programming Language: Fundamentals and Comparison with C

The Rust programming language was officially released on May 15, 2015, and after nearly a decade of continuous development, it has reached version 1.91.0, released on October 30, 2025. Rust source files typically use extensions such as *.rs* and *.rlib*. The choice of Rust in the context of this study is motivated by three key aspects that make it particularly suitable for critical systems and embedded environments: performance, reliability, and productivity [47].

Rust's performance is characterized by the absence of a traditional runtime environment and garbage collector. This enables highly efficient and predictable execution, which is essential in systems with strict timing and resource constraints. Regarding reliability, the language introduces a strict type of system and an innovative ownership model, which together provide strong guarantees of memory safety and concurrency. These mechanisms allow a wide range of errors to be detected at compile time, significantly reducing the risk of runtime failures. On the productivity front, Rust offers a smooth learning curve supported by high-quality official documentation and an intuitive compiler known for its detailed, context-aware error messages [47].

One of Rust's most notable features is its approach to memory safety, which is arguably one of the most significant vulnerabilities in systems developed using low-level languages such as C. These traditional languages, by granting direct control over memory management, introduce significant risks such as invalid accesses, memory leaks, and race conditions. Rust was specifically designed to address these issues by introducing the concept of *ownership*, a unique mechanism not found in other languages. This model will be explored in detail throughout the chapter, as it forms one of the foundational pillars that establish Rust's reputation as a safe and reliable programming language [7].

The purpose of this chapter is therefore to introduce the Rust programming language, highlighting its core features and explaining how they contribute to the goals of safety, efficiency, and robustness. This introduction will be conducted in direct comparison with the C language, in order to emphasize the conceptual and practical differences between the two, and to justify the growing interest in Rust as a modern and secure alternative for systems programming.

Installing Rust: rustup and Essential Components

To start programming in Rust, it is necessary to install the Rust toolchain. This can be done using *rustup*, a command-line tool that installs and manages the Rust compiler and associated tools [48] [49].

On most Unix-based systems, *rustup* can be installed by executing the following command [49]:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Once installed, the following command can be used:

```
rustup --version
rustup 1.28.1
```

After the initial setup is complete, *rustup* is not used frequently, except when updating or installing a new toolchain. It is particularly useful for cross-compiling on platforms that are not yet supported in the stable release [48].

In addition to installing the toolchain manager itself, *rustup* also installs the essential components required for Rust development, such as the standard library (*rust-std*), which provides core functionality for Rust programs [48] [49].

It also installs *rustc*, the Rust compiler. However, this is not commonly used. Instead, Cargo is used, which is also installed via *rustup*. Cargo is Rust's package manager and build system, handling downloading dependencies, compiling code, and running tests. It provides a more user-friendly interface than using *rustc* directly, relying on a *Cargo.toml* file located in the root of each Rust project to manage metadata and dependencies [48] [49].

The *Cargo.toml* file is organized into different sections, such as [50] [51]:

- [package]: general project metadata, including the package name, project version, authors and the rust edition.
- [dependencies]: managing project dependencies, by listing the external libraries and specifying the name and the version. Cargo will then automatically download the dependencies.
- [dev-dependencies]: like dependencies, but used only for development and testing.
- [features]: allows conditional compilation, including dependencies, or enabling parts of the code.
- [profile]: allows customization of build settings for different profiles, such as development or release.

The following code shows an example [50] [51]:

```
[package]
name = "project_name"
authors = [“my_name”]
version = "0.1.0"
edition = "2021"

[profile.dev]
panic = "abort"

[profile.release]
```

```
panic = "abort"

[dependencies]
example = "0.8"

[dev-dependencies]
example = “0.8”

[features]
default = []
feature1 = []
```

Variables and Mutability: Concepts and Usage

Different to C, where all variables are mutable by default, in Rust variables are immutable by default. Although it is possible to declare mutable variables using the *mut* keyword [52].

Considering a scenario where a segment of code assumes that a particular variable's value remains unchanged, while a subsequent segment modifies it. Such behaviour may cause the initial segment to function incorrectly. This issue can be particularly challenging to identify and resolve, especially if the modification occurs conditionally, for example within an if-else clause. The following code demonstrates how variables are declared in Rust [52]:

```
fn main() {
    const PI: f64 = 3.1415;
    let mut y = 20;
    let x = 10;
    y += 1;
    x += 1;
}

-------------------------------------OUTPUT-----------------------------------------
error[E0384]: cannot assign twice to immutable variable `x`
  --> src/main.rs:6:1
   |
 4 | let x = 10;
   |     - first assignment to `x`
 5 | y += 1;
 6 | x += 1;
   | ^^^^^^ cannot assign twice to immutable variable
   |
 help: consider making this binding mutable
   |
 4 | let mut x = 10;
   |     +++
```

This results in a compile-time error, since the previous code attempts to modify the value of an immutable variable. It is also possible to see how Rust compiler identifies the error and suggests a solution.

Rust Data Types: Definition and Usage

Each value in Rust is assigned a data type, enabling the compiler to determine at compile time how that data will be handled. Data types in Rust are organized into two subsets: scalar types and compound types [52].

Scaler Types

Scalar types represent a single value. Within this category are the following types: integers, floating-point numbers, booleans, and characters. In Rust, these data types are stored on the stack [52]. There are some significant differences compared to the C language, which will be discussed throughout this section.

Integer

An integer is a number without a fractional component. The use of integers in Rust highlights certain language features that promote greater safety compared to C. The compiler needs to know, at compile time, the amount of memory required to store the variable, emphasizing strict control over the code and preventing common errors such as buffer overflows [52].

```
1. fn main() {
2.   let mut x: u8 = 255;
3.   x += 1;
4. }
5.
6. ------------------------------------OUTPUT-----------------------------------------
7. error: this arithmetic operation will overflow
8.  --> src/main.rs:5:5
9.   |
10. 5 |     x += 1;
11.   |     ^^^^^^ attempt to compute `u8::MAX + 1_u8`, which would overflow
12.   |
13.   = note: `#[deny(arithmetic_overflow)]` on by default
14.
```

Figure B-1 presents the integer types available in Rust.

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Figure B-1 - Integer Types in Rust

Source: The Rust Programming Language [52]

In Rust, it is necessary to specify whether a value is signed or unsigned, followed by its size in bits. If this specification is omitted, the default type is i32. Signed values are identified with the prefix "i", while unsigned values use the prefix "u". For example, the type u8 means that the value stored in the variable ranges from 0 to 255 ($2^n - 1$, where n is the number of bits). In the case of i8, the first bit is reserved to store the sign, and the remaining bits represent values from (-2^7) to 2^7-1, which corresponds to the range (–128) to 127 [52].

Additionally, Rust provides the isize and usize types, whose sizes are adjusted according to the system architecture (32-bit or 64-bit) on which the program is executed [52].

Floating-Point

A floating-point type is a number with decimal points. In this case, Rust's floating-point types have two types: f32 and f64, which are 32 bits and 64 bits in size respectively. By default, the type is f64, and all are signed [52].

```
1. fn main() {
2.   let x: f64 = 2.0;
3.   let y: f32 = 3.0;
4. }
```

Boolean

The boolean data type in Rust is like that in other programming languages, storing values of true or false. It is often used in logical conditions and control flow [52].

```
1. fn main(){
2.   let f : bool = false;
3.   let g : bool = true;
4. }
```

Arithmetic Operations

As in other languages, Rust supports mathematical operations such as addition (+), subtraction (-), multiplication (*), division (/), and remainder (%). However, unlike C, Rust uses truncation rather than rounding during division [52].

```
1. let truncated = 5 / 3; // Integer division: Result is 1
2. let division = 10 / 3; // Float division: Result is approximately 3
3.
```

Furthermore, Rust avoids ambiguities by requiring operators to have compatible types, as illustrated in the following code [52]:

```
1. fn main() {
```

```
 2.   let x = 10;
 3.   let y = 2.7;
 4.   let z = x / y;
 5.   println!("{}", z);
 6. }
 7. ------------------------------------OUTPUT-----------------------------------------
 8. error[E0277]: cannot divide `{integer}` by `{float}`
 9. --> src/main.rs:5:14
10. |
11. 5 |     let z = x/y;
12. |             ^ no implementation for `{integer} / {float}`
13.
```

The previous code shows an error caused by an attempt to perform an operation between variables of different types, which is not allowed in Rust without explicit conversion. This limitation exists because the compiler cannot safely infer the resulting data type, for example, whether it should be an integer or a floating-point number. To ensure type consistency and safety, an explicit cast to a common type must be performed before carrying out the operation [52]:

This kind of casting isn't recommended in Rust, as it transfers the responsibility for ensuring correctness from the compiler to the programmer [52]:

```
1. // Using Rust
2. fn main(){
3.   let x = 10;
4.   let y  = 2.7;
5.   let z as f64 = x/y;
6.   println!("{}", z);
7. }
8.
```

However, this kind of verification does not exist in C. As shown in the following code, C allows operations between incompatible types, which can lead to unexpected behaviour [52]:

```
1. // Using C
2. void main(){
3.   int x = 10;
4.   float y  = 2.7;
5.   int z = x/y;
6.   printf("z = %d", z);
7. }
```

Character

Rust uses the char type to represent Unicode characters, occupying four bytes in memory. This approach enables the storage of a much broader set of characters compared to languages like C, where the char type is limited to one byte and only represents ASCII characters [52].

```
1. let letter: char = 'A';
2. let other_letter: char = 'À';
```

Compound Types

Compound types allow multiple values to be grouped under a single data type. In Rust, there are two primary compound types: tuples and arrays. Both are stored on the stack and have fixed sizes [52].

Tuples

Tuples are not available in C, but in Rust they allow grouping multiple values of different types into a single structure defined at the moment of declaration. Once created, the number of elements in a tuple is fixed and cannot be increased or decreased. The following code provides an example of tuple declaration and how its values can be accessed [52].

```
1.  fn main(){
2.    let tup : (i32, f64, u8) = (500,6.4, 1);
3.    println!(“{}”, tup.0);
4.  }
```

Arrays

Arrays in Rust are similar to those in the C language, having a fixed size and being stored on the stack. However, unlike C, Rust arrays include strict bounds checking at compile time, enhancing memory safety and preventing invalid memory access. This feature represents an important safety mechanism by eliminating unexpected behaviour. If the array's defined boundary is exceeded, the compiler will raise a compile-time error, thus preventing errors that may be difficult to detect and resolve [52].

```
 1. fn main() {
 2.    let array: [i32; 4] = [1, 2, 3, 4];
 3.    println!("{}", array[5]);
 4. }
 5. ------------------------------------OUTPUT-----------------------------------------
 6. error: this operation will panic at runtime
 7. --> src/main.rs:3:20
 8. 3 |     println!("{}", array[5]);
 9. |                     ^^^^^^^^ index out of bounds: the length is 4 but the index is 5
10. = note: `#[deny(unconditional_panic)]` on by default
11. error: could not compile `playground` (bin "playground") due to 1 previous error
12.
```

Repetition with Loops

To perform repetitive tasks, Rust provides three primary loops: loop, while, and for [52].

Loop

Unlike C, Rust has a construct called loop, which is an infinite repetition structure. It continuously executes the code block until it is explicitly interrupted. The only way to terminate its execution is by using the break statement. Furthermore, loop allows returning a value directly at the point where break is invoked, as demonstrated in the following example [52]:

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Source: The Rust Programming Language [52]

While

The while loop repeatedly executes a block of code as long as a specified condition remains true. In Rust, the while loop is similar to C but with some key differences [52].

One important distinction is that Rust doesn't allow non-boolean expressions as loop conditions. In C, it is valid to use an integer expression as a condition, where any non-zero value is considered true [52]:

```c
void main() {
 int x = 10;
   while (x) {
     x--;
   }
}
```

However, in Rust, this results in a compilation error, because the language enforces strict type checking. The condition must be explicitly of type bool (for example x!=0) [52]:

```rust
fn main() {
 let mut x = 10;
   while x {
     x -= 1;
   }
}
 ------------------------------------OUTPUT-----------------------------------
error[E0308]: mismatched types
 --> src/main.rs:4:7
  |
4 | while x { // ERRO: expected `bool`, found integer
  |       ^ expected `bool`, found integer
```

For

The *for loop* in Rust differs significantly from the traditional *for* in C. While in C the *for* is structured with three explicit components requiring manual control: initialization, condition, and increment. In Rust the for loop is based on the concept of iterators, allowing direct traversal over ranges of values. This approach eliminates

common errors in C, such as array bounds overruns or missing index updates. An example is shown in the following code [52]:

```
1. fn main() {
2.     let a = [10, 20, 30, 40, 50];
3.
4.     for element in a {
5.         println!("the value is: {element}");
6.     }
7. }
```

Source: The Rust Programming Language [52]

Functions declaration

To declare functions in Rust, the keyword "fn" is used, followed by the definition of parameters and the return type [52]. The following example illustrates this syntax:

```
1. fn rustExample (a: i32, b: i32) -> i32 {
2.     a + b
3. }
```

In this code, the return type is specified by "-> i32". It is important to note that the expression "a + b" doesn't end with a semicolon, unlike in C. In Rust, an expression followed by a semicolon is treated as a statement that returns the unit type "()", which would cause a mismatch with the function signature requiring a return type of i32. Alternatively, though less commonly used, it is also possible to explicitly return a value using the return keyword, such as "return a + b;" [52].

Enum Definition

Enums allow representing distinct values that belong to the same set. In C, an enum associates symbolic names with integer numbers, improving code readability but without adding information beyond those numbers. In contrast, in Rust, enums are much more versatile, besides identifying variants by name, each variant can store different data, effectively acting as compound types that combine enumerations with structures [53]. For example, the following code illustrates this capability:

```
1. enum IpAddr {
2.     V4(u8, u8, u8, u8),
3.     V6(String),
4. }
5.
6. let home = IpAddr::V4(String::from(127, 0, 0, 1"));
7.
8. let loopback = IpAddr::V6(String::from("::1"));
9.
```

Source: The Rust Programming Language [53]

In this example, the *IpAddr* enum can hold either an *IPv4* address, consisting of four numbers, or an *IPv6* address, represented by a string. This approach enables representing different formats without the need to create separate struct, thereby increasing code flexibility [53].

Result: A Type for Recoverable Errors

Not all errors should cause a program to panic. For example, if an attempt to open a file fails because the file does not exist, a better approach than panicking might be to create the file instead. To handle such situations, Rust provides the Result type, which is an enumeration representing the outcome of an operation that may fail. It is defined as Result<T, E>, where T is the type returned on success and E is the error in case of failure [54].

This type of approach forces programmers to explicitly handle both successful and unsuccessful outcomes, since the compiler requires all possible cases. This improves the robustness of the software, preventing silent error propagation and making the control flow more predictable [54].

```
1. enum Result<T, E> {
2.   Ok(T),
3.   Err(E),
4. }
```

Source: The Rust Programming Language [54]

Option: Representing Nullable Values Safely

The Option type in Rust represents the presence or absence of value. It is a generic enumeration defined as Option<T>, where T is the type of the value that may be present. It has two variants [53]:

- Some(T): indicates that the value of type T is present.
- None: indicates the absence of value.

This structure replaces the use of null pointers commonly found in C, promoting safer handling of optional values. Since the compiler enforces explicit handling of the None case, it helps prevent one of the most common software errors: dereferencing null pointers [53].

```
1. enum Option<T> {
2.   None,
3.   Some(T),
4. }
```

Source: The Rust Programming Language [52]

Control Flow in Rust

In Rust, control flow is achieved using if/else statements and the match expression.

Match

The match is a powerful control flow feature in Rust that enables exhaustive pattern matching. By requiring all possible cases to be explicitly handled, it helps to prevent logical errors and undefined behaviour. In contrast, C relies on the switch statement, which doesn't enforce completeness. This can lead to subtle bugs, such as missing

cases or unintended fall-through, unless it is managed carefully using break statements [55] [48].

A simple example in Rust:

```
fn main() {
    let number = 13;
    match number {
        // Match a single value
        1 => println!("One!"),
        // Match several values
        2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
        // Match an inclusive range
        13..=19 => println!("A teen"),
        // Handle the rest of cases
        _ => println!("Ain't special"),
}
```

The use of the underscore in match statements may seem unfamiliar to C developers. However, this is one of Rust's match statement's key strengths, it ensures that all possible cases are covered [55] [48].

If the match statement doesn't cover all possible cases, the compiler will raise an error:

```
Compiling playground v0.0.1 (/playground)
error[E0004]: non-exhaustive patterns: `i32::MIN..=0_i32`, `4_i32`, `6_i32` and 3 more not covered
  --> src/main.rs:3:11
   |
3  |     match number {
   |           ^^^^^^ patterns `i32::MIN..=0_i32`, `4_i32`, `6_i32` and 3 more not covered
   |
   = note: the matched value is of type `i32`
help: ensure that all possible cases are being handled by adding a match arm with a wildcard pattern as shown, or multiple match arms
   |\
9  ~         13..=19 => println!("A teen"),
10 ~         _ => todo!(),
   |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `playground` (bin "playground") due to 1 previous error
```

If else

Like in C, if allows decision-making based on conditions. However, in Rust, if is an expression, not just a statement. This means it can return a value, enabling direct assignments and more concise code [52]. For example:

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

Source: The Rust Programming Language [52]

Struct: Definition and using in Rust

A struct in Rust is a way to define custom compound types by grouping different pieces of data under a single name like in C. The following example illustrates a struct declaration in Rust [56]:

```
1. struct User {
2.     active: bool,
3.     username: String,
4.     email: String,
5.     sign_in_count: u64,
6. }
```

Source: The Rust Programming Language [56]

To instantiate this structure in Rust, the following syntax is used:

```
1. fn main() {
2.     let user1 = User {
3.         active: true,
4.         username: String::from("someusername123"),
5.         email: String::from("someone@example.com"),
6.         sign_in_count: 1,
7.     };
8. }
```

Source: The Rust Programming Language [56]

And to access these values, the following syntax is used:

```
1. println!("Username: {}", user1.username);
2. println!("Email: {}", user1.email);
```

Source: The Rust Programming Language [56]

Rust besides C it is possible to associate functionalities with structures through *impl* blocks, where methods and constructor functions, allowing behaviour and data to be encapsulated [56]. This approach, which doesn't exist in C, resembles the style of object-oriented languages, promoting better code organization and readability.

```
 1. #[derive(Debug)]
 2. struct Rectangle {
 3.     width: u32,
 4.     height: u32,
 5. }
 6.
 7. impl Rectangle {
 8.     fn area(&self) -> u32 {
 9.         self.width * self.height
10.     }
11. }
12.
13. fn main() {
14.     let rect1 = Rectangle {
15.         width: 30,
16.         height: 50,
17.     };
18.
19.     println!(
20.         "The area of the rectangle is {} square pixels.",
21.         rect1.area()
22.     );
23. }
```

Source: The Rust Programming Language [56]

Traits

In Rust, as can be seen in the structures using the *impl* block, there are some features associated with object-oriented programming, namely the ability to associate methods with structures. However, Rust does not support inheritance between structures. Instead, it relies on composition, which is achieved through the use of traits. Traits define behaviours that can be implemented by different types. A trait functions similarly to an interface in other languages, allowing the description of a set of methods that a type can or must implement, thus ensuring polymorphism and code reuse [8] [48].

Below is an example of a trait. Note that the *summarize_author* method does not have a default implementation, meaning it must be implemented by any type that uses the trait. On the other hand, the summarize method has a default implementation [8].

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}

 pub struct SocialPost {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub repost: bool,
}

impl Summary for SocialPost {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

Source: The Rust Programming Language [8]

Rust's standard library provides several predefined traits, among which the following stand out [48]:

- Display: Specifies the textual output format for a given type.
- Debug: Similar to Display, but intended for debugging, allowing the internal representation of a value to be shown.
- Clone: Indicates that an object can be explicitly duplicated, requiring the use of the clone() method.
- Copy: Allows for implicit copying, suitable for simple types with lightweight copy semantics.
- Deref: Allows the use of the * operator on the type, as if it were a reference.
- Into Iterator: Allows a type to be iterated over in for loops.

Text Data Types: String and &str

In Rust, there are two main types used to represent text: String and &str, which differ significantly in terms of ownership, mutability, and memory allocation [7].

String

The data types previously discussed are stored on the stack, which means that the program knows the size at compile time. However, when dealing with types that use the heap such as the string type the situation differs [7]. The following code provides an example:

```
1. fn main(){
2.    let mut my_string = String::from("Hello");
3.    my_string.push_str(", Rust!");
4.    println!("{}", my_string); // This will print `Hello, Rust!`
5. }
```

A string in Rust is mutable and dynamically allocated [7]. Figure B-2 provides a schematic representation of this memory allocation.

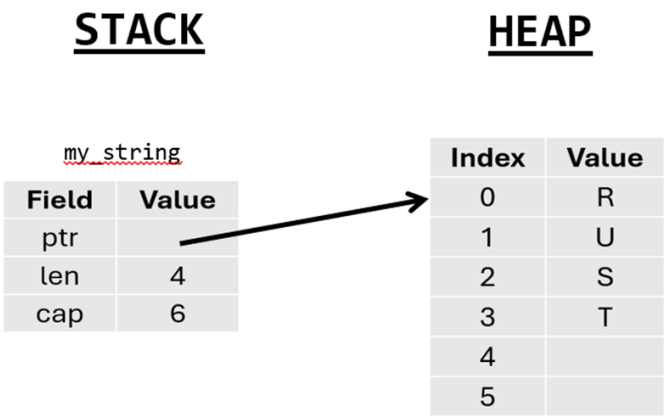


Figure B-2 - Memory allocation of the String "Rust"

String Slices (&str)

On the other hand, &str is a reference to a sequence of characters. It does not own the data it refers to, instead, it points to a slice of an existing string. As such, &str is immutable and cannot be modified. It is commonly used to inspect or access parts of a string without requiring data duplication [7]. A string slice is a reference to part of a string, and it looks like this:

```
1. let s = String::from("hello world");
2. let hello = &s[0..5];
3. let world = &s[6..11];
```

Source: The Rust Programming Language [7]

Figure B-3 illustrates a string and a corresponding slice, "World", which references a specific part of the original string.

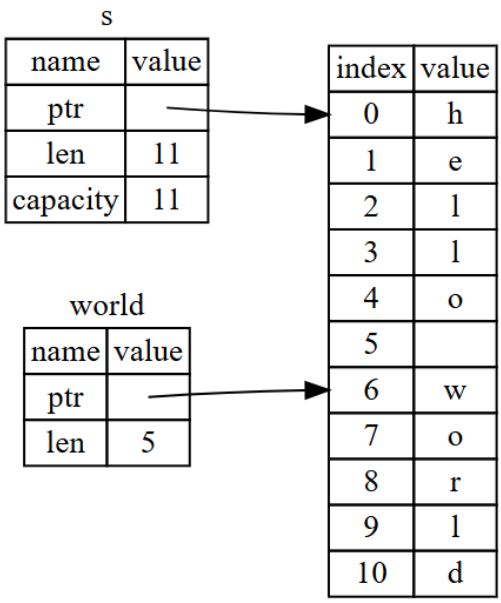


Figure B-3 - String slice "World" referring to a part of the String "Hello, World".

Annexes

ANNEX C: RESULTS OF C COMPLEXITY ANALYSIS

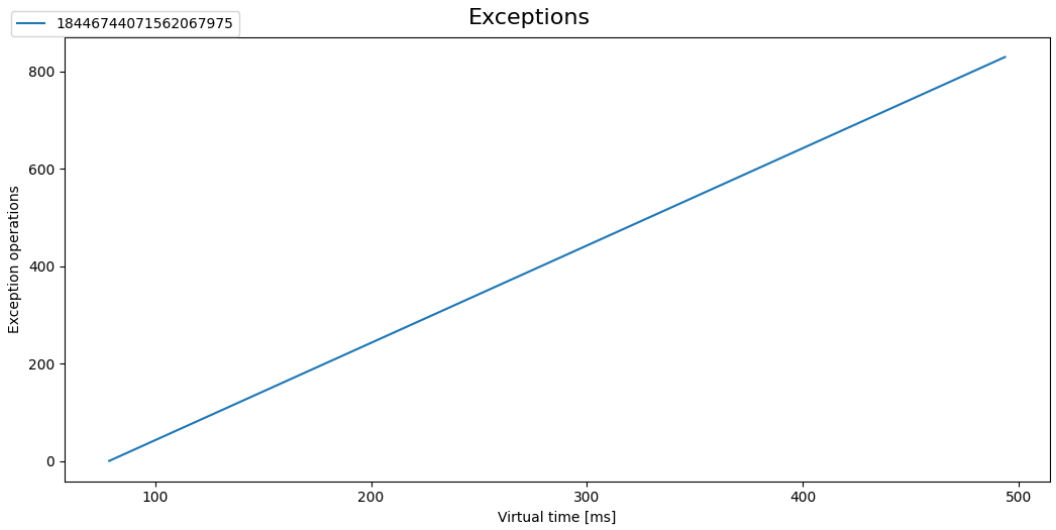
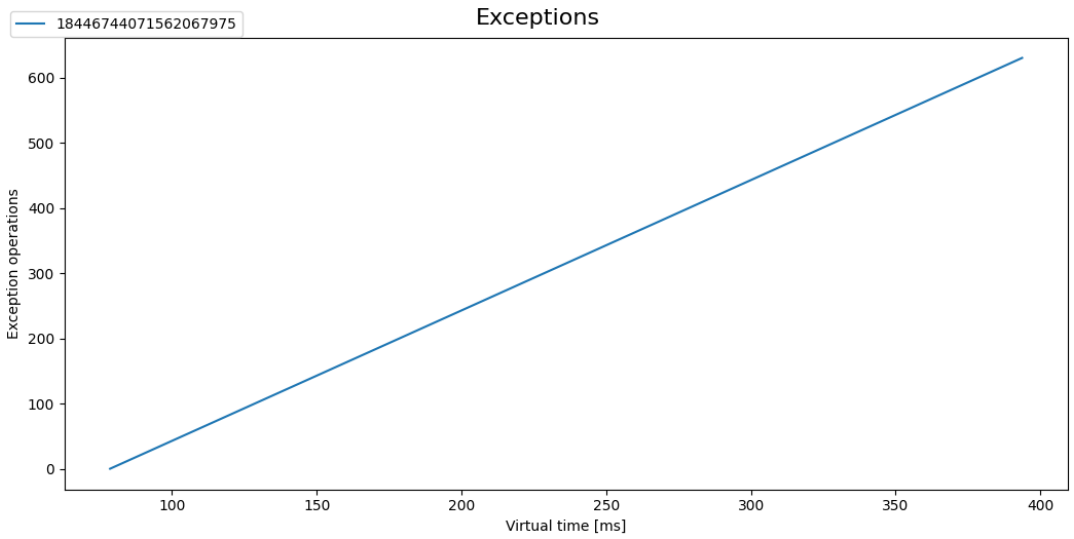
```
memory_manager/fsw/private$ lizard .
================================================
  NLOC    CCN   token  PARAM  length  location
       33      5    168      6      37 MemoryManager_CreateNvmReadWriteRequest@22-58@./memory_manager_nvm_operations.c
       33      5    165      5      37 MemoryManager_CreateNvmChecksumRequest@60-96@./memory_manager_nvm_operations.c
        7      1     72      1      11 MemoryManager_ExecuteWriteNvm@98-108@./memory_manager_nvm_operations.c
        7      1     72      1       9 MemoryManager_ExecuteReadNvm@110-118@./memory_manager_nvm_operations.c
        9      1     80      1      13 MemoryManager_ExecuteChecksumNvm@120-132@./memory_manager_nvm_operations.c
       23      3    245      3      26 MemoryManager_UpdateFATBlocks@23-48@./memory_manager_file_operations.c
       27      6    189      6      29 MemoryManager_GetAddressPointerAndLength@61-89@./memory_manager_file_operations.c
       27      4    178      3      34 MemoryManager_WriteFileBlockAllocation@97-130@./memory_manager_file_operations.c
       45      8    463      1      75 MemoryManager_ExecuteWriteFile@137-211@./memory_manager_file_operations.c
       23      3    276      1      41 MemoryManager_ExecuteReadFile@217-257@./memory_manager_file_operations.c
       27      3    240      1      36 MemoryManager_ExecuteCopyMoveFile@265-300@./memory_manager_file_operations.c
       29      5    154      0      33 MemoryManager_ScrubMemory@19-51@./memory_manager_scrubbing.c
       12      2    108      1      14 MemoryManager_ExecuteChecksumMemoryArea@23-36@./memory_manager_direct_operations.c
       13      2    117      1      15 MemoryManager_ExecuteDumpMemoryArea@47-61@./memory_manager_direct_operations.c
        8      1     60      1       8 MemoryManager_ExecuteLoadMemoryArea@68-75@./memory_manager_direct_operations.c
       30      5    183      0      55 MemoryManager_Init@86-140@./memory_manager.c
       15      4     48      0      26 MEMORY_MANAGER_Main@142-167@./memory_manager.c
       40      5    158      0      54 MemoryManager_CyclicOperationsProcess@177-230@./memory_manager.c
       48     14    156      1      58 MemoryManager_ExecuteRequest@240-297@./memory_manager.c
       41     14    173      2      52 MemoryManager_FreeRequest@303-354@./memory_manager.c
       34      4    175      7      37 MemoryManager_CreateFileWriteReadRequest@372-408@./memory_manager.c
       35      4    180      6      38 MemoryManager_CreateFileCopyMoveRequest@425-462@./memory_manager.c
       34      4    174      7      36 MemoryManager_CreateGeneralMemoryDumpRequest@480-515@./memory_manager.c
       41      5    206      7      44 MemoryManager_CreateGeneralMemoryLoadRequest@532-575@./memory_manager.c
       34      4    174      7      37 MemoryManager_CreateGeneralChecksumRequest@592-628@./memory_manager.c
       22      2     91      2      29 MemoryManager_UpdateRequestPointer@637-665@./memory_manager.c / 7 file analyzed.
================================================================
NLOC    Avg.NLOC  AvgCCN  Avg.token  function_cnt    file
----------------------------------------------------------------
      93      17.8     2.6      111.4          5     ./memory_manager_nvm_operations.c
     173      28.7     4.5      265.2          6     ./memory_manager_file_operations.c
      35      29.0     5.0      154.0          1     ./memory_manager_scrubbing.c
       0       0.0     0.0        0.0          0     ./memory_manager_config.h
      35      11.0     1.7       95.0          3     ./memory_manager_direct_operations.c
     409      34.0     5.9      156.2         11     ./memory_manager.c
      50       0.0     0.0        0.0          0     ./memory_manager.h
=================================================================================================
No thresholds exceeded (cyclomatic_complexity > 15 or length > 1000 or nloc > 1000000 or parameter_count > 100)
=================================================================================================
Total nloc   Avg.NLOC  AvgCCN  Avg.token   Fun Cnt  Warning cnt   Fun Rt   nloc Rt
-------------------------------------------------------------------------------------------------
       795      26.8     4.4      165.6       26            0      0.00    0.00
```

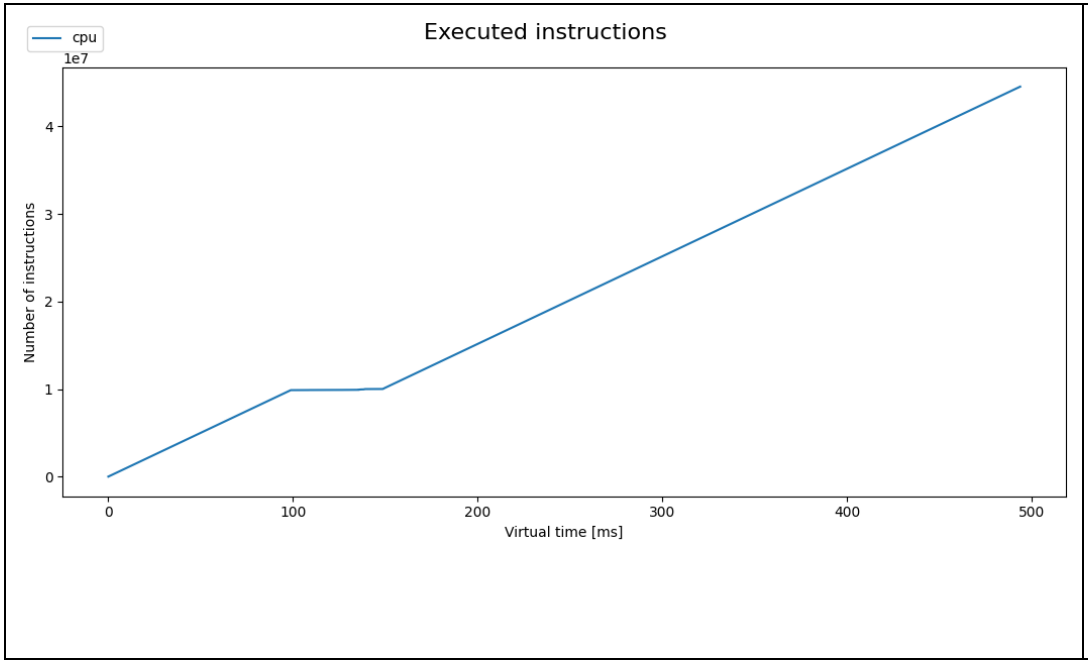
ANNEX D: RESULTS OF RUST COMPLEXITY ANALYSIS

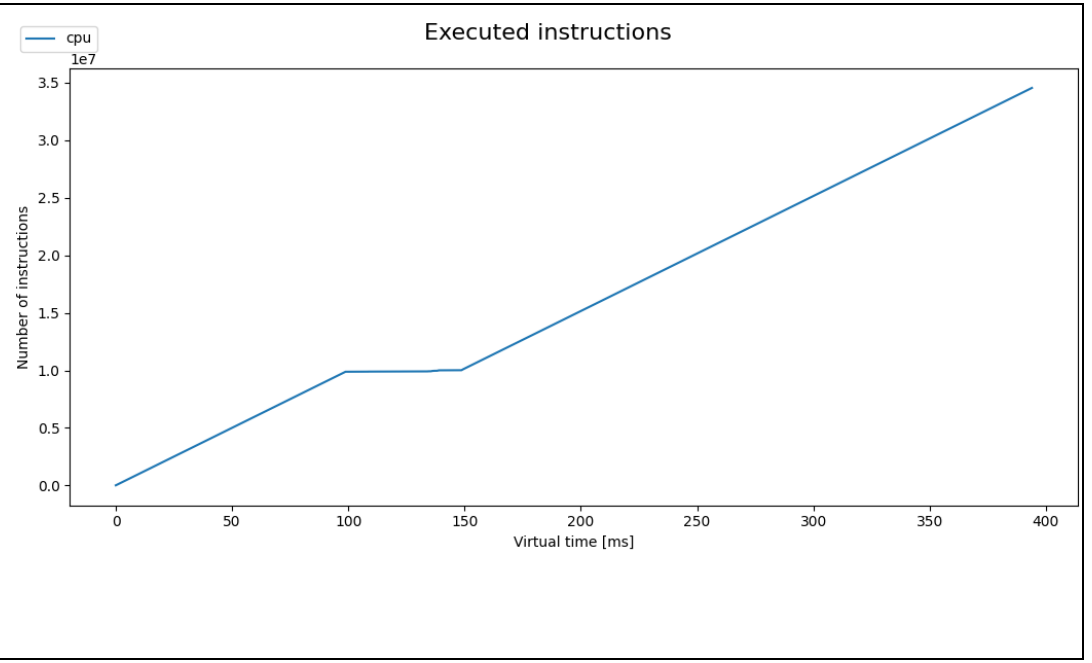
```
memory_manager_rust/src$ lizard .
================================================
  NLOC    CCN   token  PARAM  length  location
------------------------------------------------
      36      2    216      1      46 MemoryManager_ExecuteChecksumMemoryArea
      31      2    216      1      43 MemoryManager_ExecuteDumpMemoryArea
      17      1    109      1      23 MemoryManager_ExecuteLoadMemoryArea
      49      3    259      6      59 MemoryManager_CreateNvmReadWriteRequest
      51      4    269      5      61 MemoryManager_CreateNvmChecksumRequest
      32      1    173      1      39 MemoryManager_ExecuteWriteNvm
      32      1    172      1      39 MemoryManager_ExecuteReadNvm
      34      1    180      1      41 MemoryManager_ExecuteChecksumNvm
      39      6    240      0      54 MemoryManager_ScrubMemory
      66      4    396      1      85 MemoryManager_UpdateFATBlocks
      51      8    298      5      57 MemoryManager_GetAddressPointerAndLength
      44      4    269      3      51 MemoryManager_WriteFileBlockAllocation
      75      5    458      1      98 MemoryManager_ExecuteReadFile
     127     11    734      1     149 MemoryManager_ExecuteWriteFile
      66      3    403      1      77 MemoryManager_ExecuteCopyMoveFile
      23      2    127      3      33 MemoryManager_UpdateRequestPointer
      58      6    311      7      77 MemoryManager_CreateGeneralMemoryLoadRequest
      47      3    238      7      58 MemoryManager_CreateGeneralMemoryDumpRequest
      49      3    238      7      61 MemoryManager_CreateGeneralChecksumRequest
      49      3    240      7      62 MemoryManager_CreateFileWriteReadRequest
      48      3    244      6      61 MemoryManager_CreateFileCopyMoveRequest
       1      1      6      0       1 rust_eh_personality
      17      1     47      6      17 create_mm_operation
      37      2     66      0      60 get_instance
       3      1     17      1       3 clear_operation_queue
      15      3     56      1      16 show_operations
      14      1     81      1      15 increments_Index
       6      2     34      1       7 increments_Index
       3      1     14      1       3 get_Index
     126      1     25      0     126 get_instance
       6      2     34      1       7 increments_Index
       3      1     14      1       3 get_Index
      62      1     25      0      62 get_instance
       6      2     34      1       7 increments_Index
       3      1     14      1       3 get_Index
      60      1     25      0      60 get_instance
       6      2     34      1       7 increments_Index
       3      1     14      1       3 get_Index
      40      1     25      0      40 get_instance
      33      2    157      0      45 MemoryManager_Init
      52     14    230      1      75 MemoryManager_ExecuteRequest
      49     13    312      1      70 MemoryManager_FreeRequest
      30      5    136      0      41 MemoryManager_CyclicOperationsProcess
      17      2     50      0      30 Memory_Manager_Main
7 file analyzed.
==============================================================
NLOC    Avg.NLOC  AvgCCN  Avg.token  function_cnt    file
--------------------------------------------------------------
     87      28.0     1.7      180.3         3      ./memory_manager_direct_operations.rs
    215      33.5     1.8      179.7         6    ./memory_manager_nvm_operations.rs
     40      39.0     6.0      240.0         1    ./memory_manager_scrubbing.rs
    439      71.5     5.8      426.3         6    ./memory_manager_file_operations.rs
    284      45.7     3.3      233.0         6    ./memory_manager_create_operations.rs
    124       0.0     0.0        0.0         0    ./memory_manager_interface.rs
    767      25.7     2.7       63.0        23    ./lib.rs


===============================================================================================
===============
No thresholds exceeded (cyclomatic_complexity > 15 or length > 1000 or nloc > 1000000 or
parameter_count > 100)
=========================================================================================
Total nloc   Avg.NLOC  AvgCCN  Avg.token   Fun Cnt  Warning cnt   Fun Rt    nloc Rt
-----------------------------------------------------------------------------------------
       1956       36.0      3.1      161.4        45            0      0.00    0.00
```

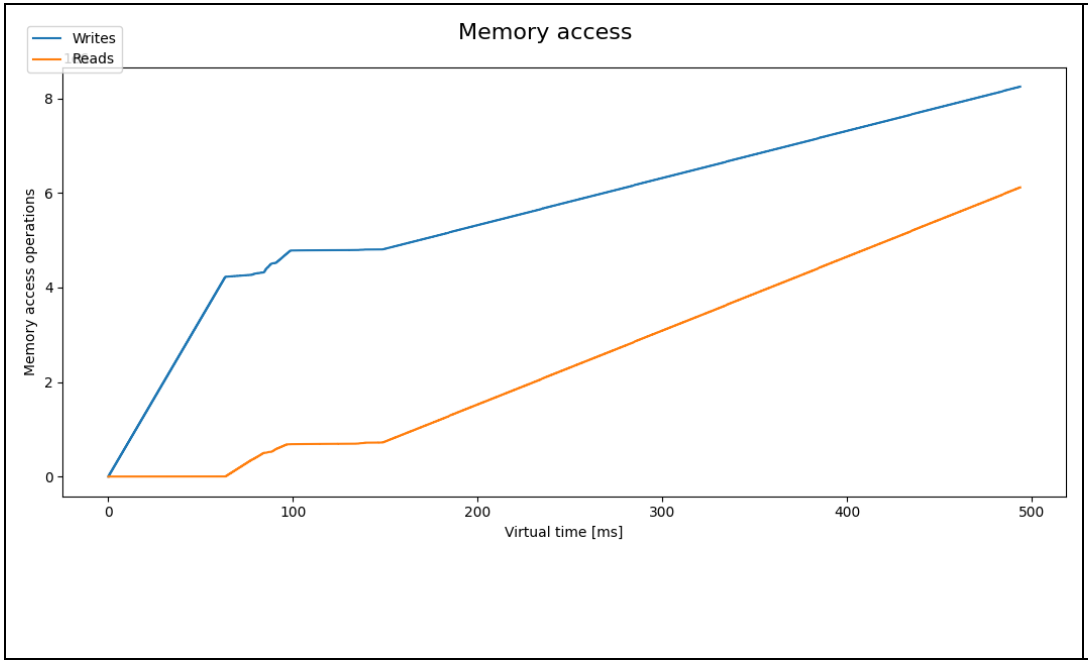
ANNEX E: RENODE PROFILING VISUALIZATION

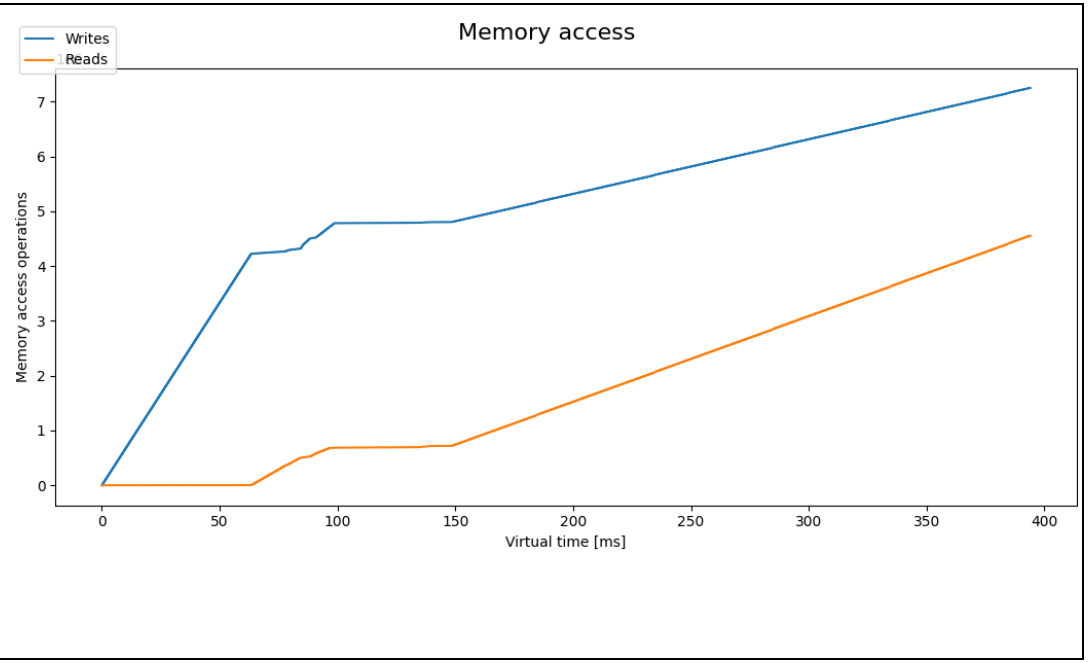
Service 6 and subservice 10

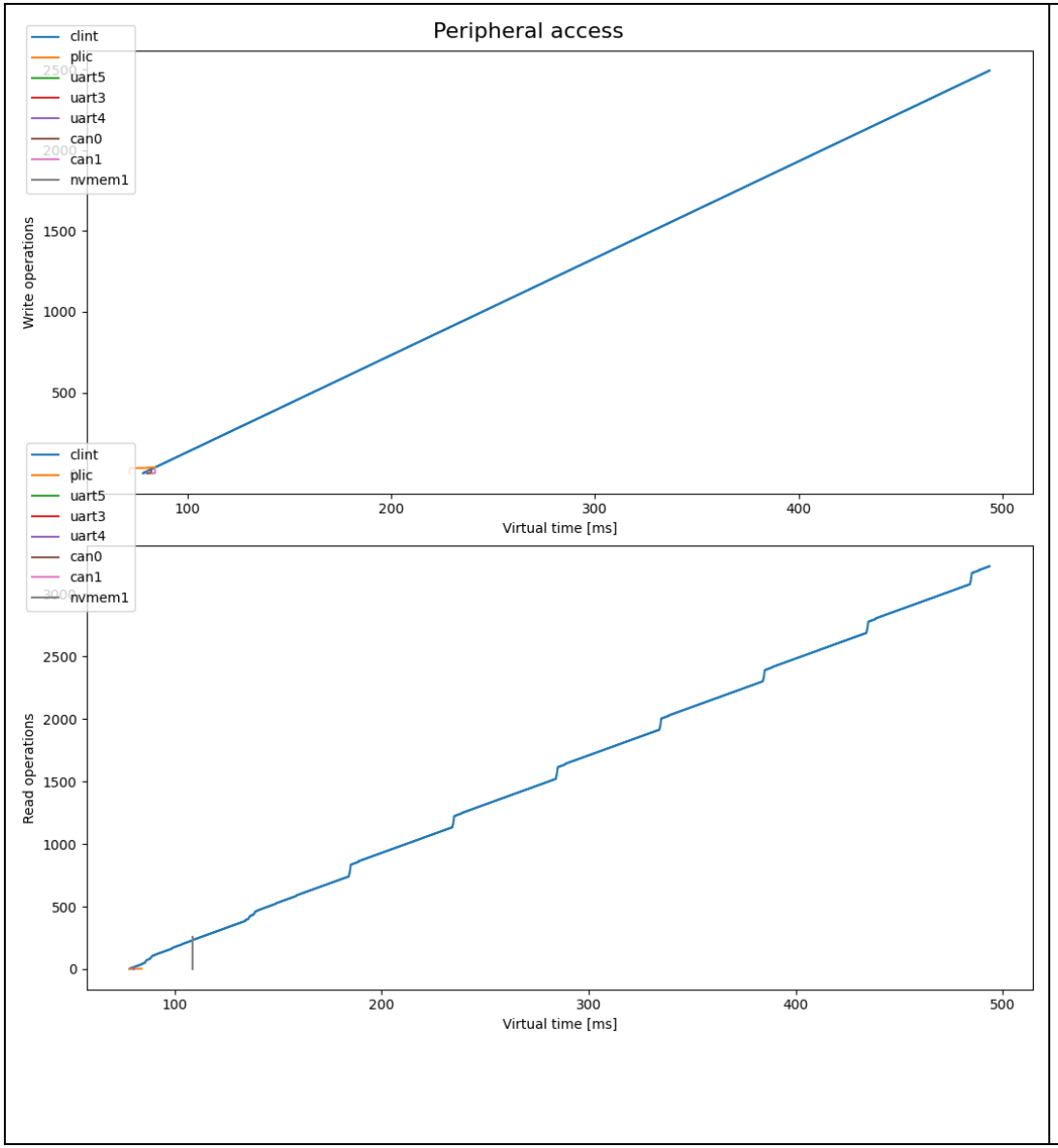
Result in C	Result in Rust
	

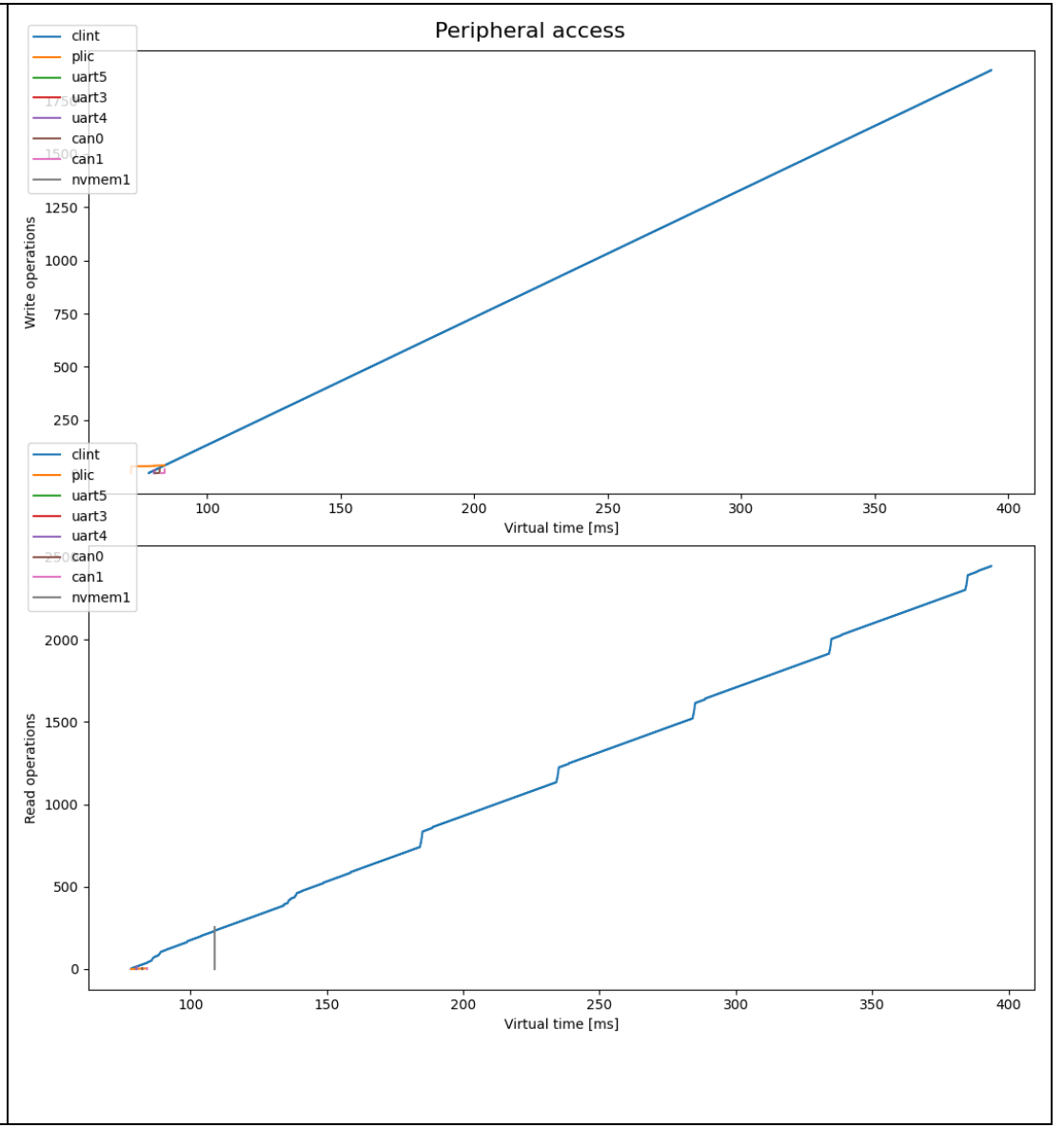

Executed instructions
cpu
1e7
Number of instructions
Virtual time [ms]
0
1
2
3
4
0
100
200
300
400
500


Executed instructions
cpu
1e7
Number of instructions
Virtual time [ms]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
50
100
150
200
250
300
350
400


Memory access
Writes
Reads
Memory access operations
Virtual time [ms]
0
2
4
6
8
0
100
200
300
400
500


Memory access
Writes
Reads
Memory access operations
Virtual time [ms]
0
1
2
3
4
5
6
7
0
50
100
150
200
250
300
350
400


Peripheral access
clint
plic
uart5
uart3
uart4
can0
can1
nvmem1
Write operations
Virtual time [ms]
Read operations
Virtual time [ms]


Peripheral access
clint
plic
uart5
uart3
uart4
can0
can1
nvmem1
Write operations
Virtual time [ms]
Read operations
Virtual time [ms]

Instituto Superior de Engenharia
Politécnico de Coimbra