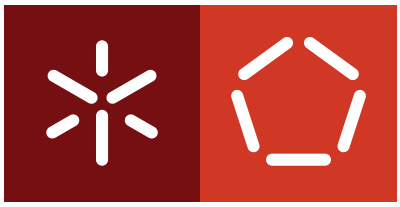
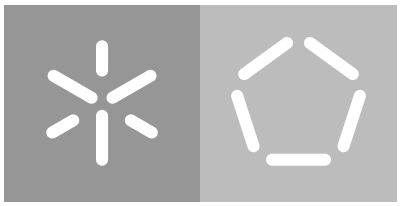


Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Brandão de Pinho

Exploring Rust for Embedded and Critical Systems

January 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Brandão de Pinho

Exploring Rust for Embedded and Critical Systems

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho realizado sob a orientação de
José Nuno Oliveira
Luís Diogo Couto

January 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

ACKNOWLEDGEMENTS

This dissertation marks the end of my journey at the University of Minho. First of all, I wish to thank UTRC Ireland for proposing the theme of this dissertation. This experience definitely helped me evolve as a person and as a master student. I also wish to thank my supervisors for all the guidance and supervision during this project. Prof. José Nuno Oliveira was the reason why I chose Formal Methods and Dr. Luís Couto always made sure I had a good time at UTRC in Cork (Ireland).

A special thanks to my girlfriend and all my friends from Braga, Oliveira and Cork. Some have been part of my life since the 7th grade and others for just the last 8 months. You all helped me during this journey and I will always remember all the work, fun and friendship we shared.

Finally, I thank my family, specially my parents and my sister, for supporting and believing in me.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RUST FOR CRITICAL SYSTEMS - ABSTRACT

The C programming language is perhaps the most widespread in the design of safety-critical systems. However, its adoption suffers from disadvantages because it lacks in safety, entailing extensive and expensive verification processes.

There are other programming languages, such as Ada and SPARK, that offer safety features that automatically comply with the current safety standards. Nevertheless, the software industry continues to use C after all and its associated verification process to build safety-critical software.

Rust is a modern programming language that promises to soften such insecurities by design, thus improving on the development of safety-critical software. Due to its ownership model, Rust can ensure memory safety at compile time. Because it is a systems programming language, it is also a promising language for embedded systems.

The main aim of the project reported in this dissertation is to understand how Rust can alleviate the certification process of safety-critical software, while evaluating its maturity for embedded systems. We analyse in which platforms Rust is available and compare embedded Rust to other languages used in the safety-critical domain. This analysis consists in comparing Rust safety features with coding guidelines commonly used in the software industry.

Some case studies are carried out, such as preemptive and cooperative scheduling (both in C and Rust to better understand the programming differences in these languages), a driver for an accelerometer and finally a program that uses the scheduler, the accelerometer and the leds of a micro-controller, where one thread reads acceleration values and another thread turns leds on/off according to such readings.

The dissertation ends with an overview of the results obtained. Not only a comparison is given with coding guidelines used in industry, but also concerning the case studies developed. It also anticipates some important work that could be added, as well as some details where Rust could be improved to become prominent in the industry of safety critical software.

Keywords: Critical Systems, Embedded Systems, Rust.

RUST PARA SISTEMAS CRÍTICOS - RESUMO

A linguagem de programação C é talvez a que tem mais representação no design de sistemas críticos. Contudo, a sua utilização tem algumas desvantagens pois falha em segurança, o que resulta em processos de verificação extensos e dispendiosos.

Existem outras linguagens de programação, como Ada e SPARK, que possuem funcionalidades de segurança que automaticamente correspondem aos standards de segurança. Apesar disso, o que se verifica é que a indústria continua a usar C e o consequente processo de verificação para desenvolver software crítico.

Rust é uma linguagem moderna que promete atenuar tais inseguranças pelo *design*, melhorando assim o desenvolvimento de software crítico. Graças ao seu modelo de *ownership*, a linguagem consegue assegurar uma utilização da memória de forma segura no momento da compilação do código. Como Rust é uma linguagem de programação de sistemas, esta é promissora para ser usada nos sistemas embebidos.

O objectivo principal desta dissertação é investigar como é que Rust pode suavizar o processo de certificação de software crítico, e avaliar a maturidade de Rust para sistemas embebidos. Assim, analisamos em que plataformas Rust está disponível e comparamos Rust embebido com outras linguagens utilizadas neste domínio. Esta análise consiste em comparar as características de segurança de Rust com as normas de codificação utilizadas na indústria.

Alguns casos de estudo foram desenvolvidos, tais como *schedulers* preemptivos e cooperativos (tanto em C como em Rust para perceber melhor quais as diferenças em programar nestas linguagens), um *driver* para usar um acelerómetro e, por fim, um programa que faz uso do *scheduler*, acelerómetro e leds presentes no microcontrolador, tendo uma *thread* a ler valores de aceleração e outra *thread* a ligar ou desligar os leds de acordo com essas leituras.

A dissertação acaba com uma visão global dos resultados obtidos. Não é feito apenas uma comparação com as normas de codificação usadas na indústria, como também uma comparação dos casos de estudo desenvolvidos. Também exploramos algum trabalho importante que pode ser desenvolvido no futuro, bem como alguns detalhes onde a linguagem pode ser melhorada para poder fazer parte da indústria crítica.

Palavras-chave: Rust, Sistemas Críticos, Sistemas Embebidos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

A

ANSI American National Standards Institute

D

DAC Digital to Analogue Converter
DAL Design Assurance Level
DoD Department of Defense

E

EASA European Aviation Safety Agency
ESC Extended Static Checking

F

FAA Federal Aviation Administration

H

HAL Hardware Abstraction layer

I

IEC International Electrical Commission
IEEE Institute of Electrical and Electronic Engineers
ISO International Standards Organization
ITM Instrumentation Trace Macrocell

J

JPL Jet Propulsion Laboratory

M

MIR Mid-level Intermediate Representation
MISRA Motor Industry Software Reliability Association

N

NASA National Aeronautics and Space Administration

P

PhD Doctor of Philosophy

R

RTOS Real Time Operating System

S

SPI Serial Peripheral Interface

T

TCB Thread Control Block

U

UTRC United Technologies Research Centre

1

INTRODUCTION

1.1 CONTEXT

Software plays a major role in human life nowadays, as it is present in areas of activity as diverse as entertainment, finance, health, military and transportation. The process of developing software varies across these fields: software produced for the entertainment industry does not require the same precautions as the software produced for the transportation or military industry. When software failures occur in health or transportation, they can lead to loss of human life and loss of large amounts of money. In order to safeguard software users from such unfortunate and severe outcomes, there must be a way to ensure that software is fault-free.

SAFETY CRITICAL SOFTWARE There are various definitions of safety-critical software. The *Institute of Electrical and Electronic Engineers (IEEE)* defines it as follows:

> *"(...) software whose use in a system can result in unacceptable risk. Safety-critical software includes software whose operation or failure to operate can lead to a hazardous state, software intended to recover from hazardous states, and software intended to mitigate the severity of an accident" [18].*

In its *Software Safety Standard* [39], the *National Aeronautics and Space Administration (NASA)* identifies safety critical software wherever one of the following criteria is met:

1. It resides in a safety-critical system (as determined by a hazard analysis) AND at least one of the following apply:
   - Causes or contributes to a hazard.
   - Provides control or mitigation for hazards.
   - Controls safety-critical functions.
   - Processes safety-critical commands or data.
   - Detects and reports, or takes corrective action, if the system reaches a specific hazardous state.

- Mitigates damage if a hazard occurs.
- Resides on the same system (processor) as safety-critical software.

2. It processes data or analyses trends that lead directly to safety decisions.
3. It provides full or partial verification or validation of safety-critical systems, including hardware or software systems.

Safety-critical software must typically undergo some sort of certification before it can be out into production. This typically requires a significant amount of testing and/or formal validation.

Testing consists in executing the software with various input data and analysing the results [38]. There are various approaches to testing, such as WhiteBox Testing where the implementation/structure of the software is known to the tester, or BlackBox Testing where testers do not know how the software is implemented, just its functionality. These approaches are applied in Unit Testing, Integration Testing and System Testing. Testing is an important part of software development, but cannot guarantee program correctness, since it can only prove the occurrence of errors (vulg. bugs), never their absence. By contrast, provable correctness can be achieved by use of so-called formal methods [6].

Formal Methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems [9]. These can be applied to any part of the development cycle. When modelling an automated system, a formal language such as e.g. Alloy [2] can be used to express and verify assumptions and desired properties, to explore alternative designs, to clarify and validate the requirements, and so on. On the other hand, tools such as e.g. Frama-C [15] perform static analysis of code. This analysis provides warnings of possible bugs, for example divisions by zero, out-of-bounds array indexing and others dangerous constructs.

INDUSTRIAL CONTEXT This dissertation is proposed by *United Technologies Research Centre (UTRC)* Ireland. It was founded in Cork in 2010, and it focuses on building energy, security and aerospace systems. UTRC Ireland, the European hub of UTRC, is the central research unit of United Technologies Corporation, the world's largest supplier of aerospace products.

The Cork office has approximately 80 employees, 63% of whom hold a *Doctor of Philosophy (PhD)* degree from more than 20 countries worldwide [20]. The staff is divided in several working groups, such as:

- Network and Embedded Systems.
- Formal Methods.
- Control and Decisions Support.

- System Modelling and Optimisation.
- Power Electronics.

1.2 MOTIVATION AND AIMS

As stated before, critical software must be fail-safe. Unfortunately, there are several obstacles to achieve this. The formal methods approach has a steep learning curve for those not having the required background in maths and logics. Moreover, there are also problems with the current state of critical software development such as the increase in lines of code [44]. The Airbus A380 is estimated to rely on around 100 million lines of code, compared to 20 million for the A330 [55]; the Boeing 787 has an eight to tenfold increase in lines of code, and the trend is for this number to keep increasing [40]. In addition, these systems are becoming more complex: while flight control surface interfaces were mostly mechanic in 2003, in 2013 fly-by-wire software dominates, improving aircraft performance and stability [40].

The majority of critical systems are of an embedded nature because of real time constraints, typical in aerospace industry. In this domain software is usually written in low-level system languages such as C and Ada, which are efficient languages that can meet the requirements of embedded systems. The C programming language [25] is the first choice for real-time embedded systems applications, because it offers flexibility and portability across a wide range of hardware. C is used for some specific reasons such as [3]:

- Available on most microprocessors.
- Good support for high-speed, low-level, input/output operations, essential in embedded systems.
- Unlike many other high-level languages, it compiles to smaller and less RAM-intensive assembly code.

Ada is frequently said to be a fail proof language but it is far from popular, being often regarded as a difficult language with a slow learning curve. Meanwhile C is one of the most used languages, being constantly on top of charts every year [33].

When C was designed, its creators intention was that C developers could do anything they wanted. However, this resulted in a language without fail-safes, which can lead to disastrous consequences. Some examples of unsafe behaviour are described below.

PROGRAMMING MISTAKES Programmers are human beings, so it is expected that they will make mistakes, such as writing beyond the boundaries of an array or failing to catch integer overflows. Unfortunately, C allows for typing mistakes that result in valid code,

like typing an assignment instead of a logical comparison or a semi-colon typed on the end of an if statement. Also, as C lacks automated memory management, the programmer must know how to use *malloc* and *free* functions correctly, otherwise memory leaks and/or corruption are likely to happen.

RUN-TIME ERRORS C does not provide a good run-time error protection, since compilers do not provide the detection of arithmetic exceptions, overflows, validity of pointers or array bound errors.

LACK OF TYPE SAFETY C is a weakly typed language. The compiler tries to ensure type correctness, but programmers can override it with explicit casts, converting a value or pointer from one type to another.

COMPILER OPTIMIZATIONS Although compiler optimizations can be turned off, they can cause issues. Compilers are not obliged to generate code for undefined behaviours, generating code with better performance, but assuming these behaviours won't occur increases the risk of software defects and vulnerabilities.

The Rust programming language [43] promises reliability and high performance. High performance concerns not only the programs created, but also the speed of writing them. It offers good reliability: a rich type system, an ownership model that guarantees memory and thread safety, and a compiler capable of detecting many kinds of bugs. There is also a rich, open documentation in the form of books, namely *Rust By Example* [41] and videos.

The option for Rust in this dissertation is motivated not only by the ambition to increase safety and productivity, speed and ergonomics, but also due to the investment of Rust in embedded systems, promising benefits such as:

- *Powerful static analysis*, preventing data races at compile time.
- *Flexible memory management*, with dynamic memory allocation being optional.
- *Fearless concurrency*, with the help of a number of abstractions and safety guarantees.
- *Interoperability*, to provide integration of Rust with C.
- *Portability*, with the help of embedded-*Hardware Abstraction layer (HAL)*.
- *Community Driven*, i.e. supported by an open source community.

Rust offers the benefits of C and Ada, without the drawbacks of such languages. As shown before, C requires extensive verification due to lack of safety of the language, and

other languages such as Ada are more difficult to use but offer more safety (due to compile-checks and the structure of the language itself). Rust on the other hand has the potential to address these issues, by having a compiler capable of checking safety features that C compilers do not, and at the same time being user friendly (when compared to Ada). In short, Rust promises various safety features while also offering most capabilities of the current trending and used programming languages.

DISSERTATION AIMS The main objective of this dissertation is to investigate the safety capabilities that Rust offers, to see if they comply with current industry standards for critical software, and to check the possibility of running Rust on embedded systems. In the end, we intend to understand if Rust is ready to be used in critical systems, and if not, to identify what is missing.

1.3 DISSERTATION STRUCTURE

The remainder of this dissertation is organised as follows:

- Chapter 2 addresses the safety standards currently followed to guarantee that software is safe, the languages used in critical systems, and provides an overview of the Rust language and its current use in embedded systems.
- Chapter 3 covers the main objectives of the dissertation: to investigate potential benefits of Rust for critical software by comparing Rust with coding guidelines used in aerospace, and assessing the maturity of embedded Rust.
- Chapter 4 presents some case studies. These include scheduling algorithms developed both in C and Rust, an accelerometer driver, and a program that makes use of a microcontroller's leds and accelerometer.
- Chapter 5 ends the dissertation by evaluating our response to the objectives proposed in the third chapter, and also identifying possible tracks for future work.

2

STATE OF THE ART

2.1 DO178-C AND OTHER SAFETY STANDARDS

Software developed for safety-critical applications must be compliant with safety standards to ensure it is indeed safe. DO178-C *Software Considerations in Airborne Systems and Equipment Certification* [36] is the primary standard for airborne systems. Published on December 13th of 2011, it is approved by authorities such as the *Federal Aviation Administration (FAA)* and *European Aviation Safety Agency (EASA)*, and was recognised by the FAA on July 19th of 2013 through the document AC 20-115-C [14]. It was preceded by DO178-B, which was replaced to add guidance for parameter data items, clarify some definitions on key concepts of requirements and other criteria. This standard provides recommendations for developing critical embedded software, describing "what" shall be done and not "how" it shall be done [56].

DO178-C has five levels of criticality, *Design Assurance Level (DAL)*, from A to E (level A being the most rigorous). These levels are based on the contribution of software to potential failure conditions, and for each level, there is a number of objectives to be accomplished.

DAL	Failure Condition	Number of Objectives
A	Catastrophic	71
B	Hazardous	69
C	Major	62
D	Minor	26
E	No Effects	0

Table 2.1.: Number of objectives for each DAL

DO178-C Annex A table provides guidelines for applying these objectives by software level. Table 2.2 contains the number of objectives per process.

Table #	Table Title	Number of Objectives
A-1	Software planning process	7
A-2	Software development processes	7
A-3	Verification of outputs of software requirements process	7
A-4	Verification of outputs of software design process	13
A-5	Verification of outputs of software coding and integration process	9
A-6	Testing of outputs of integration process	5
A-7	Verification of verification process results	9
A-8	Software configuration management process	6
A-9	Software quality assurance process	5
A-10	Certification liaison process	3

Table 2.2.: Summary of DO178-C Annex A Tables

Alongside DO178-C, there were 4 other standards published to supplement it:

- DO-330 Software Tool Qualification Considerations.
- DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A.
- DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A.
- DO-333 Formal Methods Supplement to DO-178C and DO-278A.

DO-330 provides software tool qualification guidance. Tool qualification is the process used to gain certification credit for a software tool whose output is not verified, when the tool eliminates, reduces, or automates processes required by DO-178C [40]. The other three documents contain modifications to DO178-C if Model-Based Development and Verification, Object-Oriented Technology or Formal Methods are used as part of software life cycle.

Alongside the safety standards, companies often use software coding guidelines for writing safety critical software. MISRA-C is a set of software development guidelines first published in 1998 by *Motor Industry Software Reliability Association (MISRA)*. The intention is to provide a subset of a standardised structured language to promote the safest use possible of the language. While originally written for the automotive industry, it has now been

adopted in a wide range industries and applications, including aerospace. It consists of 122 mandatory rules and 20 advisory [3], so we can understand that it implies an extensive verification process.

There are other guidelines which comply with MISRA-C, for instance the guidelines used by the *Jet Propulsion Laboratory (JPL)* [26]. These guidelines are aimed for critical flight software, focused on embedded software. As an example, rule number 5 (equivalent to rule 20.4 of MISRA-C) states "There shall be no use of dynamic memory allocation after task initialisation". Memory allocators often have unpredictable behaviour that can significantly impact performance, and errors such as forgetting to free memory or using memory after it has been freed can lead to problems. Another example rule is 6.3 "Typedefs that indicate the size and the signedness should be used in place of basic type." [1]. This rule exists because C compilers use different underlying types for basic types, resulting in different programs with the same source code. As we will see in the next chapter, Rust can provide some promising features to develop safety-critical software and potentially eliminate some of the current standard rules for safety critical software development.

2.2 Languages used in Critical Systems

There are few programming languages used in critical systems, C and Ada being the most used nowadays [40]. FORTRAN and Pascal were used in the past, but today are mostly in legacy code, or being ported to C [19]. We shall focus more on C and Ada because they represent two different styles of programming in safety-critical, one being very permissive, while the other too restrictive.

2.2.1 *C*

C, as stated before, is a programming language developed in 1972 by Dennis Ritchie. It became on of the most popular programming languages after the UNIX operating system kernel being written in C. Seeing that C is normally the most common language on critical-safe systems, and because of the unsafe properties mentioned in the previous chapter, one option is to use coding guidelines in order to meet the safety requirements of safety critical software.

Since C has poor language support for safety, there is a need of using tools to support its use in critical software, such as source code analysers, logic model extractors, metrics tools, debuggers, test support tools, and a choice of mature, stable compilers [17]. There are several tools for safety-critical C, such as Frama-C, Astrée and Polyspace, that are capable of detecting divisions by zero, out-of-bounds array indexing, erroneous pointer manipulation. In 2003, Astrée proved the absence of any runtime errors in 132000 lines of C code used

on an Airbus model [52]. Unfortunately, these tools require knowledge on how to operate them, and so the verification process can be a tedious one. Fortunately, some of these tools are already compliant with DO178-C and MISRA-C.

2.2.2 *Ada*

Another software language present in critical-safe software is Ada. It was developed at request of United States *Department of Defense (DoD)* and mandated by the DoD for several years. As there were so many languages used in DoD projects, their ambition was to standardise one language to support embedded, real-time, and mission-critical applications. It has been standardised through the *International Standards Organization (ISO)*, *American National Standards Institute (ANSI)*, and *International Electrical Commission (IEC)*. In 1997, as the number of programming languages had declined, the DoD removed its mandate to use Ada because the remaining languages were mature enough to be used in critical software [40].

Ada is acclaimed as a very safe programming language, with features like strong typing, built-in support for design-by-contract, run-time checking, exception handling, packages, scalar ranges, generic templates and high-level concurrency – features that were not used in mainstream languages when Ada appeared. Outside of the safety-critical environment, Ada is not used which caused its decline in popularity [32].

2.2.3 *SPARK*

SPARK is a formally defined software language based on Ada. It consists of a defined subset of Ada, but an integral part of SPARK is its annotation language, forming an essential part of the ”contractual” specifications of programs. Besides signatures, SPARK has additional contractual elements like core annotations that allow a fast, polynomial time analysis of SPARK source code to check for errors [8]. There are also proof annotations, which support mathematical proof of properties of the source code. Image 2.1 presents the ”relationship” of Ada and SPARK.

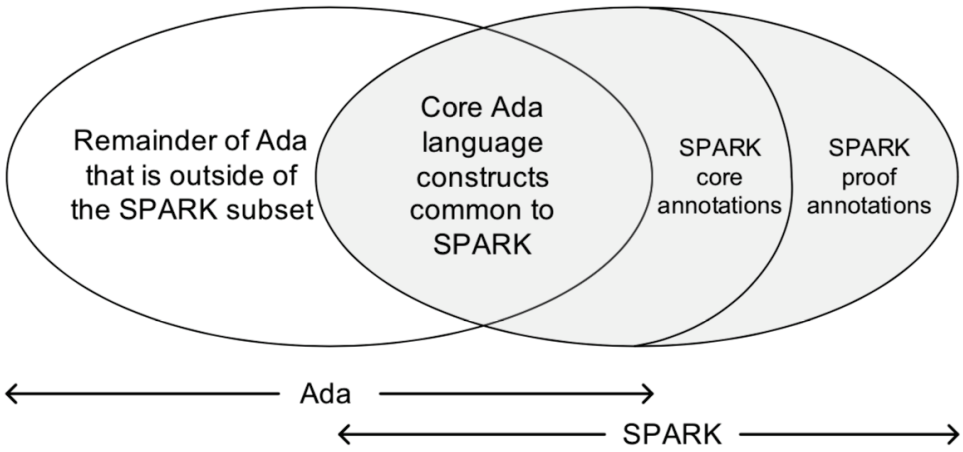


Figure 2.1.: Relationship between Ada and SPARK [8]

SPARK programs are meant to be unambiguous, and the choice of Ada compiler must not affect the behaviour of the program. SPARK also excludes some Ada constructs such as pointers, dynamic memory allocation or recursion to make static analysis viable [45]. Some major objectives on defining the language identified in its Language Reference Manual [8] were:

- Logical soundness.
- Simplicity of formal language definition.
- Expressive power.
- Security.
- Verifiability.
- Bounded space and time requirements.

To verify SPARK code, there is GNATprove, a static analyser that aims to prove subprograms by analysing each individually. It is based on Why3, a platform for deductive program verification that relies on external theorem provers like cvc4, alt-ergo or z3 [7].

Although SPARK does not allow pointers, there is a subset of SPARK called µSPARK which features pointers, records, loops and procedure calls. It uses a permission-based static alias analysis method inspired by Rust's borrow-checker [22]. SPARK has been used in various safety-critical systems, most notably on the Vermont Technical College CubeSat Laboratory [53].

While being regarded as safe, and alleviating verification processes, it is often said that Ada is an unpopular language mainly due to the difficulty of writing Ada programs, with

programmers often preferring to code in C and have an extensive verification process, than coding in Ada with a small verification process.

Therefore, on one hand, we have C which gives freedom to the programmer, even with the guidelines mentioned, but results in extensive verification processes. On the other, Ada is a lot more restrict, which leads to less errors and less verification, but it is not a popular language, neither outside or inside safety-critical software contexts.

2.3 RUST

Rust shares some objectives with DoD requirements, with emphasis on safety and targeting embedded environments, with efficiency and real-time responsiveness [35]. That is a good indicator for being used in safety-critical software. As noted before, standards and guidelines have numerous objectives for safety-critical software languages to comply with, such as memory management. The memory management of Rust, unlike C, is made with a system of ownership and is verified by the compiler at compile time. This allows for less verification, since the risk of memory errors such as freeing memory already freed (and other cases of misuse of C) are protected against.

We will proceed to explain some of key features of the language responsible for safety.

2.3.1 *Ownership*

In order to achieve memory and thread safety without garbage collection, Rust uses an ownership system that has three rules [49]:

- Every value has a variable called its owner;
- There can only be one owner at a time;
- When the owner goes out of scope, the value will be dropped.

In listing 2.1, every rule is presented while also showing invalid uses of the ownership system. When a variable is assigned to another or when a function receives a variable as a parameter, the ownership is transferred, and when the variable owner goes out of scope, memory is automatically returned. One important thing to note is that types with a known size at compile time, like integers or booleans, are copied (not moved), because they are stored on the stack.

```
{
    let s1 = String::from("hello");
    let s2 = s1; //s1 value is dropped here, s2 becomes the owner
    let s3 = String::from("world");

    println!("{}, world!", s1); //error

    func(s2); //ownership is transferred to function func

    println!("{}, world!", s2); //error again!
} //scope is over, s3 is no longer valid
```

Listing 2.1: Example of ownership

2.3.2 *Borrowing*

In order to share a variable without transferring ownership, Rust provides a mechanism which allow us to use references called borrowing.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1); // immutable borrow of s1

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
  // it refers to, nothing happens.
```

Listing 2.2: Example of immutable borrow

There are two types of borrows, immutable and mutable. The previous example shows an immutable borrow, meaning it would be illegal to change the string `s1`. Mutable borrows can be defined as seen on listing 2.3.

We can have an unlimited number of immutable references or a single mutable reference, and it is only valid to have one of the borrow types at the same time. This is how Rust can prevent data races at compile time.

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s); //now it is allowed to change the string
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Listing 2.3: Example of mutable borrow

2.3.3 *Lifetimes*

Every reference has a lifetime, that is, the scope for which that reference is valid, applying both to owned and borrowed variables. Its purpose is to prevent dangling references.

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Listing 2.4: Example of lifetimes with annotations

In this example, the code will not compile because the value `r` is referring to (`x`) has gone out of scope before being used. The compiler compares the size of the two lifetimes (*'a* and *'b*), and because *'b* is shorter than *'a*, r cannot refer to memory with a shorter lifetime.

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Listing 2.5: Example without dangling reference

In listing 2.5, `r` can reference `x` because its lifetime is shorter than the lifetime of `x`. Most times the compiler infers the lifetime, but with some data types (like structs) lifetimes must be explicitly annotated by the programmer.

2.3.4 *Rust Verification*

Although Rust is a safe language, sometimes the developer needs to perform some unsafe operations. Rust compiler is conservative, because if it cannot determine whether the code upholds its guarantees, then the code is rejected. Because sometimes the programmer needs to bypass this conservative analysis of the compiler, *unsafe Rust* exists. When the unsafe keyword is used, the code inside a unsafe block is not verified by the compiler, making it prone to memory safety errors such as null pointer dereferencing. These unsafe features are needed in order to do low-level systems programming, like directly interacting with the operating system. There are libraries that contain unsafe Rust code, which means that programmers must take responsibility to uphold the guarantees normally guaranteed by the compiler.

While it is expected for bugs occurring just when unsafe code is used, there have been studies [57] showing the presence of bugs, like deadlocks and data races, on concurrent programs with ”safe” code. This means the developers should always be cautious, because the compiler isn't 100% safe proof.

As previously stated, the presence of unsafe code means there is no protection from the compiler on those code blocks. Furthermore, there is no tool provided by Rust capable of features like model checking. As with other software programming languages, Rust has already some software verification tools to address these issues.

SMACK SMACK is a software verification toolchain and a self-contained software verifier that can be used to verify assertions in programs. SMACK translates LLVM IR code into Boogie intermediate verification language, which is in turn verified using back-end Boogie verifiers such as Corral [5]. Since *rustc* can produce LLVM IR code, small efforts were needed for SMACK to support the verification of Rust programs, because LLVM IR code patterns produced by the compiler of Rust and Clang compiler can be different. For example, Rust compiler emits instructions to verify if integer operations perform overflow through the use of LLVM arithmetic. By using SMACK, it is possible to check for integer overflow. Image 2.2 shows the tool flow of SMACK.

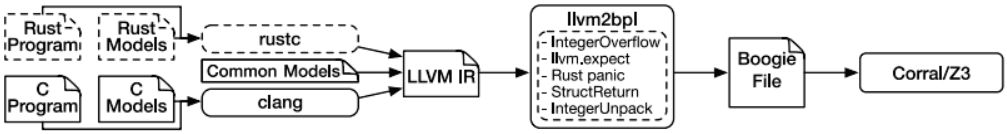


Figure 2.2.: SMACK tool flow [5]

There is still some work to do regarding this tool, namely modelling more standard libraries, checking of unsafe pointers and concurrent programming.

RUSTBELT Since none of the safety features that Rust claims were formally proven, and because some Rust standard libraries make use of unsafe blocks, the RustBelt project was proposed. In this project, the first formal (and machine-checked) tools are presented, in order to verify safe encapsulation of unsafe code [23]. The people behind this project developed λ_{Rust}, the formal version of Rust which is very close to the *Mid-level Intermediate Representation (MIR)* of Rust, and RustBelt, a semantic model of λ_{Rust} types in Iris.

The main result of RustBelt is the authors showing that for any inference rule of the type system the resulting Iris theorem holds, which means that safe and unsafe code can be used together (it is semantically well-typed), and a semantically well-typed program is memory and thread safe, meaning it will never perform any invalid memory access and will not have data races.

For future work, the authors intend to bring λ_{Rust} closer to MIR, which should allow them to reduce the number of primitive instructions.

K-RUST K-Rust is the first formal *executable* semantics for Rust [24] [54]. It uses K, an executable semantic framework which has been applied to programming languages such as C and Java. This semantics covers all safe constructors and the totality of the Rust type system, leaving only the unsafe constructors.

The workflow is to first type check a surface-rust program with the type system of K-Rust, and if the program is correct, then it is translated to a core-language level to test its operational semantics. These were executed and compared with normal Rust executions, and inconsistencies were found between the compiler and the ownership rules, namely allowing a mutable and immutable borrow simultaneously, but applying a freezing mechanism to prevent any data-race.

At time of publication, the authors aimed for providing automatic translation between the surface-language and core-language, semantics for non-safe Rust constructors and to prove refinement between K-Rust ownership system and existing abstract principles of the ownership system [24].

OTHER PROJECTS There were being developed projects for more Rust verification tools, but as of today, there have not been updates to keep up with the updates of the language itself.

CRUST was a bounded verifier, designed for detecting memory safety errors and aliasing invariant violations in unsafe Rust code. CRUST translates Rust to C code, and then uses CBMC, a well known C bounded model checker, which uses an internal SAT solver to determine what properties hold. The goal of CRUST was to find memory-safety bugs in a library, if any exists [51].

Klee-Rust was another project that utilises Rust compiler feature of producing LLVM IR code, because KLEE is a tool for symbolic execution operating on the LLVM IR of an input program. Its aim was to statically ensure memory safety and panic-free execution of code, to avoid downgrading the performance of the program and panicking [30].

2.3.5 *Embedded Rust*

Rust is a promising language to be used in embedded systems, because of the safety features without sacrificing what current languages offer. Moreover, its linear type system allows capabilities which cannot be implemented efficiently on the current languages, and improve security and reliability of system software [4]. The most stable embedded architectures Rust targets at the moment are the ARM Cortex-M, MIPS and RISC-V [48].

Since it is a recent language, Rust is not present in embedded systems like C or Ada are, but there are already some projects using Rust. Tock [50] is an embedded operating system designed for running multiple concurrent, mutually distrustful applications on Cortex-M based embedded platforms. Tock isolates software flaws, provides memory protection, and efficiently manages memory for dynamic application workloads written in any language [28]. Furthermore, the authors claim that using a type-safe language without garbage collector prevents what would be some of the largest sections of trusted code base [29], meaning a language like Rust can reduce its amount. A smaller trusted code base results in reduced susceptibility for the system to be compromised.

Sensirion, Airborne Engineering, 49 Nord and Terminal Technologies are companies already using Rust in embedded systems, in a variety of embedded products such as an embedded demonstrator for a particulate matter sensor, an Ethernet bootloader or contactless payment terminals [11].

2.4 OTHER POSSIBLE LANGUAGES

There are other programming languages rising up in popularity, and with safety features by design, which can also be a good option for future critical software.

D The D programming language is a multi-paradigm language designed to attempt to combine performance and safety of compiled languages with the expressiveness of dynamic languages. It has a garbage collector to manage memory, but it can be controlled by the programmer. Moreover, it possess a safe subset named SafeD, which guarantees no writes to memory that were not allocated.

GO Golang is a language designed at Google, syntactically close to C, but with memory safety, garbage collection, strong static typing and rich support for concurrency. One of its strong points is the GCC front end. Although it has garbage collection, it could be used in an embedded environment where that would not be a restriction.

HASKELL Haskell is a purely functional programming language with lazy evaluation. It has inspired several programming languages, including Rust. While recursive functions are forbidden by coding guidelines for critical software, the functional features of Haskell alongside its type system allow you to write safe software.

2.5 SUMMARY

There are safety standards that safety-critical software must comply with, like the DO178-C in aerospace industry, and several coding guidelines, like the MISRA-C, help software developers in writing code compliant with the safety standards.

The most used language on this environment is mostly C, with some code still being developed in Ada. C is the most popular language, with more documentation and users, but it is very unsafe so it must be written in accordance with safety guidelines. Moreover, it requires an extensive verification process to ensure it is written without errors. Ada is a much more verbose and restrictive language than C, resulting in programmers disliking it, but it does guarantee more reliability and eliminates some of the process of verification which is required by using C on safety-critical software.

Rust is a promising language, since it offers the best of both worlds: popularity and writing similar to C, and safety of Ada. The concepts of Ownership, Borrowing and Lifetimes give some potential safety features guaranteed by the design of the programming language. The ability of the compiler to detect bugs can help programming safer software in these industries. Rust is already being used in embedded systems, but it is still a very recent language compared with the established ones in the aerospace industry, where its presence has not been observed.

3

POTENTIAL FOR RUST IN CRITICAL EMBEDDED

The main research question addressed by this dissertation is whether Rust — a recent, promising programming language that is much loved by programmers [46] — can replace C as the primary language in critical embedded systems. Secondly, we wish to identify what is necessary to change, either in the language or in the embedded systems environment, for it to be effectively used in this context.

The strategy envisioned to provide answers to such questions is two-fold:

- **Objective 1** – Investigate in what measure Rust ensures software verification and meets the demands of current standards used in the safety-critical software industry.
- **Objective 2** – Assess the maturity of Rust for embedded systems (namely with respect to which processors it is available for) and compare the language with others currently used in embedded systems.

In this chapter we shall analyse Rust features and how they relate to current software standards, compare Rust with Ada/SPARK in terms of safety features and with C in terms of performance, and finally evaluate the Rust maturity for embedded systems.

3.1 RUST AND SAFETY

As mentioned in chapter 2, the design of the Rust language is essential to build reliable code. Rust features such as the strong type system, ownership and lifetimes prevents data races and dangling references, ensuring memory safety. Also, the verification tools allow to verify user assertions in programs and use Rust libraries formally verified. Nonetheless, safety-critical software must comply with safety standards.

3.1.1 *Current Software Standards*

There is an extensive verification of critical-safe software in order to achieve code compliant with, for example, MISRA-C. The objective of these coding guidelines is to eliminate any

undefined behaviour on software. Rust has the potential to ease up this process, because of its safety features that prevent undefined behaviour. We investigated some rules of MISRA-C, first some used by JPL, since they are aimed at the development of mission critical flight software, and then other rules that we understand that can be handled in some way by Rust. In the end, we identified why the rules exist and how Rust obviates them.

RULE 1.5 This rule states "Floating-point implementations should comply with a defined floating-point standard". There have been several incidents related with floating-point arithmetic, and some of these problems are solved by using an appropriate standard, such as the IEEE-754.

The floating-point types (32 and 64 bits) used in Rust are represented according to the IEEE-754.

RULE 2.3 This rule states "The character sequence /* shall not be used within a comment". C language itself does not support nested comments because of the syntax.

```
int i = 3, j = 6;
/* printf("debug: i=%d\n", i); /* check the value of i */
printf("debug: j=%d\n", j); // check the value of j */
```

Listing 3.1: C example of nested comments

As the example shows, the second *printf* would actually be executed. Rust language supports nested block comments, meaning an equivalent program would only do the variable assignments.

RULE 9.1 This rule states "All automatic variables shall have been assigned a value before being used". Although in ISO C, uninitialised variables are automatically initialised to zero by default, many embedded environments do not implement this behaviour, which can lead to undefined behaviour.

In Rust, it is impossible to use a variable not initialised, as the compiler forbids this behaviour.

RULE 12.2 This rule states "The value of an expression shall be the same under any order of evaluation that the standard permits". This means code such as the one presented on listing 3.2 is forbidden.

These operations are forbidden because of the potential of side effects, since the order in which the expressions may be evaluated can vary. In Rust, every assignment returns *()*, also called as Unit Type, which means this type of behaviour is not allowed by design of

```
x = b[i] + i++;
/* Code with potential side effects. Should be written as:
x = b[i] + i;
i++; */

a = func( i++, i ); /*Another use of increment operators with potential side
    effects*/
```

Listing 3.2: C examples of increment operators

the language. Moreover, operators such as ++ do not exist in Rust for the same reason, as they often lead to bugs and undefined behaviour.

RULE 12.9 This rule states "The unary minus operator shall not be applied to an expression whose underlying type is unsigned". In C it is possible to apply the minus operator to a unsigned variable. For example:

```
int main()
{
    unsigned int x = 2;

    printf("%u", -x); //Prints UINT_MAX - 1 = 4294967294

    return 0;
}
```

Listing 3.3: C example of assigning minus operator to unsigned integer

As we can see, this is a software practice which can lead to unexpected behaviours since a developer may expect the value of -x to be -2 (e.g. forgetting the variable is unsigned).

In Rust, this type of behaviour is forbidden by the compiler.

RULE 13.1 This rule states "Assignment operators shall not be used in expressions that yield a Boolean value". Due to being easy to type '=' instead of '==', sometimes the code is syntactically valid, but the software behaviour is different than expected. Unfortunately, mistakes such as this one can be very common, since the C language allows for this types of assignments.

As previously mentioned, assignments in Rust return a Unit type, meaning the compiler does not allow this type of behaviour, even suggesting to change the assignment to a comparison as shown on listing 3.4.

```
error[E0308]: mismatched types
 --> src\main.rs:4:8
  |     let x = 1;
  |
4 |     if x = 2 {
  |        ^^^^^
  |        |
  |        expected bool, found ()
  |        help: try comparing for equality: `x == 2`
```

Listing 3.4: Compiler error on if assignment

RULE 14.1 This rule states "There shall be no unreachable code". The example given on MISRA-C is the following:

```
switch (event)
{
 case E_wakeup:
 do_wakeup();
 break; /* unconditional control transfer */
 do_more(); /* Not compliant - unreachable code */
 /* ... */
 default:
 /* ... */
 break;
}
```

Listing 3.5: C example of unreachable code

Unreachable code is undesired because it leads to unnecessary memory overheads and caching cycles, as well as code that can be unnecessarily maintained as it is never executed. Although the Rust compiler cannot catch all types of unreachable code, it does warn for cases such as the one present on listing 3.5.

Moreover, there is a utility macro for unreachable code on the core library. It is intended to be used whenever the compiler cannot determine if some code is unreachable, like in match arms with guard conditions, or loops/iterators that dynamically terminate. In the case that the unreachable code is actually reachable, the unreachable macro invokes a panic, with the behaviour defined by the developers in the `panic_handler` function.

RULE 14.4 This rule states "The *goto* statement shall not be used". In 1962, Edsger Dijkstra explained why *goto* was consider harmful, as it lead to unstructured code, and adivsed for the statement to be abolished from all "high level" programming languages[12]. Rust, like other modern languages, does not support this type of statements.

RULE 14.8 This rule states "The statement forming the body of a *switch*, *while*, *do ... while* or *for* statement shall be a compound statement". The reason for this is due to fact that the code can be misleading and lead to unexpected behaviour, as seen in the following example.

```
for (i = 0; i < N_ELEMENTS; ++i)
{
 buffer[i] = 0; /* Even a single statement must be in braces */
}
while ( new_data_available )
 process_data (); /* Incorrectly not enclosed in braces */
 service_watchdog (); /* Added later but, despite the appearance
           (from the indent) it is actually not part of
           the body of the while statement, and is
           executed only after the loop has terminated */
```

Listing 3.6: C example of layout for compound statements

In Rust it is obligatory for the code to be enclosed within braces in loop constructs, therefore avoiding this type of mistakes.

RULE 14.9 This rule states "An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement". This rule uses the same reasoning as the previous rule, preventing unexpected behaviour. Moreover, in Rust it is obligatory for the code to be enclosed within braces in if constructs.

RULES 15 This set of rules specifies how switch cases shall be used, even having a specific MISRA-C syntax. The issue with the switch statement is that it allows complex and unstructured behaviour. We think that the Rust Pattern syntax can eliminate part of this set of rules, like:

- An unconditional *break* statement shall terminate every non-empty switch clause;
- The final clause of a switch statement shall be the *default* clause;
- Every switch statement shall have at least one case clause;

Regarding the *break* statement, there is no need of its use in Pattern Match of Rust, as every match arm is separated by a comma. In terms of the *default* clause, while it is defensive programming, Rust compiler will warn for non-exhaustive patterns, and if a match arm is left without any code, the compiler will issue an error, expecting expression. Coding the C example on MISRA-C guidelines to Rust, we confirmed these compiler errors:

```
match i {
       let x; /*expected pattern, not variable assignment*/
       0 => a=b,
       1 => , /*expected expression*/
       2 => {
           a = c;
           if a == b {
               3 => println!("Error");      /*The compiler doesn't allow the => to
                   be used here*/
           };
       },
       4 => a=b,
       5 => a=c,
       _ => errorflag = 1,
   };
```

Listing 3.7: Rust version of switch example on MISRA-C guidelines

There is also another rule stating the switch expression shall not represent a boolean value. Rust Match pattern evaluates the type of the value, so in case it is coded a boolean value, the only accepted patterns are *True* and *False*.

RULE 16.10 This rule states "If a function returns error information, then that error information shall be tested". This is because C does not enforce using the return value of a function, and the return value may provide information on the occurrence of an error. The Enum Result is a more robust and simple mechanism to handle the occurrence of errors, opposed to C where an error must be coded with a special flag. The Result type contains two variants, `Ok(T)` represents success and contains a value, `Err(E)` represents an error and contains an error value. This rule is still required in Rust but if the Enum Result is used, Rust compiler issues a warning when a Result value is ignored.

```
warning: unused `std::result::Result` that must be used
 --> src\main.rs:8:1
  |
8 | file.write_all(b"important message");
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(unused_must_use)] on by default
  = note: this `Result` may be an `Err` variant, which should be handled
```

Listing 3.8: Compiler warning if Result value is ignored

RULE 21.1 This rule states "Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults". While there are few verification tools for Rust, this rule also gives some guidance on dynamic checks that Rust can handle:

- *Pointer Dereferencing*: Due to Rust ownership system, all dereferences are always valid at compile time. Another important feature is the absence of NULL in Rust, having instead the Enum Option that encodes the concept of a value being present or absent.
- *Arithmetic Errors*: While compiling in debug mode, Rust includes checks for integer overflow, and if it occurs, the program will panic (exit with an error);
- *Divisions by zero*: Once again, the Option Enum of Rust can be an answer to this problem, returning values when the division is possible and returning None when the quotient is zero. Nonetheless, verification for division by zero is still required;
- *Array bound Errors*: In Rust, when trying to access a position outside the bounds of the array, the program will panic;

We summarise this rules in table 3.1:

Misra Rule	Satisfied	Partially Satisfied
1.5	✓	
2.3	✓	
9.1	✓	
12.2	✓	
12.9	✓	
13.1	✓	
14.1		✓
14.4	✓	
14.8	✓	
14.9	✓	
15		✓
16.10		✓
21.1		✓
Total Rules	9	4

Table 3.1.: Summary of evaluation of Rust on MISRA-C Rules

It is important to note that although the features of Rust eliminate some of the rules on software standards, the certification process usually involves using coding guidelines in order to be compliant with DO178-C, and if it is not possible to completely eliminate these guidelines, then there must exist alternative guidelines for Rust or, for example, using

formal methods. Another important information is that automatic analysis of Rust code must be validated by the certification authority, that is, must be target of a Tool Qualification process.

3.1.2 *Comparison with other safety-critical languages*

SPARK Comparing Rust with SPARK regarding safety features, there are some present in both languages [21] :

- *No-aliasing*: Both languages prevent harmful aliasing in the source code;
- *Lifetime check*: Rust compiler borrow-checker ensures lifetimes, whereas it is a separate analysis of GNAT compiler;
- *Automatic reclamation*: In Rust this is handled by the borrow checker, and in Ada it is handled with *pools*;
- *Initialisation checking*: Initialisations are checked by the borrow-checker, and SPARK does it with flow analysis, catching some uninitialised variable usage;
- *Nullity checking*: There are no NULL pointers in Rust (excluding unsafe code), while in SPARK they must be checked by static analysis tools.

C After noticing how Rust safety features exceed those of C by analysing the MISRA guidelines, we also need to compare Rust with C, but in terms of performance, one of the main objectives for being used in critical systems. In two projects, one where Rust is used to build network drivers(Ixy) [13] and other where it is used on developing a kernel [31], there were tests comparing the performance of Rust and C.

In Ixy, the developers compared the Rust implementation with a C implementation. In terms of throughput, the benchmark showed the relative difference between these implementations varying from the C version being one-eight faster than the Rust version, to a negligible performance difference. In the safe kernel project, no losses of performance were registered, with similar results being obtained. But it is important to note that if a different device driver were to be used, that could result in different performance results.

A common point of both of these investigations is that Rust provides memory safety, and even if some unsafe code is used, it is always safer than the full unsafe C code.

3.2 EMBEDDED RUST MATURITY

One important aspect is to assess in what platforms Rust is available in the present. Currently, the Rust embedded working group [42] is divided in teams, developing and maintaining the core crates (libraries), tools and resources in the embedded ecosystem.

Rust is currently available for the ARM Cortex-M, Cortex-A and Cortex-R, with the last two being in a early state. The Cortex-A crate currently only provides safe wrappers around assembly instructions, and the Cortex-R crate is being worked on, aiming for a safe API to emit architecture specific instructions and to manipulate system registers. The Cortex-M architecture is the most stable, with the crate providing access to core peripherals (e.g. *NVIC*, *SysTick*), access to core registers (e.g. *MSP*, *PSR*), interrupt manipulation mechanisms and safe wrappers around Cortex-M specific instructions (e.g. *bkpt*). Rust is also available on RISC-V, with its crate providing access to core registers such as *mstatus* or *mcause*, interrupt manipulation mechanisms and wrappers around assembly instructions like *WFI*. Lastly, it is available in the MIPS CPU architecture, with its crate providing abstractions for virtual and physical addresses, MIPS specific instructions, control the interrupt processing and paging virtual address management.

Alongside crates for the architectures referred previously, there are also crates for specific boards, with the more popular ones being the *STM32F3* Discovery, the Stellaris Launchpad and the Micro Bit (regarding downloads on crates.io). There are several crates for peripheral access API for a number of boards, like for most *STM32* microcontrollers, Texas Instruments *TM4C123x* and *TM4C129x* microcontrollers, and various others. Most of these crates are generated using *svd2rust*, which generates Rust register maps from SVD files. There is also the embedded-HAL, which aims to build abstractions (traits) for all the embedded I/O functionality commonly found on microcontrollers. Some of the defined traits in the HAL are the input and output pins (*GPIO*), serial communication and timers/countdowns.

Several driver crates, agnostic to platform, have been released with documentation. These use the *embedded-hal* interface to support all the devices which implement the traits. There are currently over 20 drivers with released status, and more than 60 with work in progress status.

There are also *no-std* crates, designed to run on resource constrained devices by not using the Rust standard library. Currently there are over 800 embedded crates on *crates.io*. Table 3.2 summarises the information of embedded Rust.

Supported Architectures	ARM Cortex-M, Cortex-A and Cortex-R RISC-V MIPS
Supported Boards	Several from STMicroeletronics, Texas Instruments, Nordic, Adafruit
Number of Hardware Drivers	20+
Number of Work in Progress Drivers	60+
Number of *no-std* crates	800+

Table 3.2.: Aspects of Embedded Rust Maturity

Another important aspect of the Rust language is the crate system. In C there is no standard package manager, and when projects are big, complex, with a significant number of files, inappropriately handling *includes* results in:

- Bigger compilation times.
- Increase in complexity.
- Increased difficulty in restructuring programs.
- Often importing unnecessary functions.
- Difficulty in managing versions of libraries.

In Rust, there is a tool named Cargo, capable of downloading dependencies of a package, compiling them, updating (with option to just update a desired package), and also invoking the compiler or other build tool. The website *crates.io* is the crate registry from the community, and cargo is configured to use it by default, although allowing to use crates from other websites (e.g. *GitHub*) is possible. Every project has a manifest file where the user can specify which dependencies the project will use, as well as which version. Comparing with C, this means dependencies are much easier to deal with. It also uses conventions for file placement in order to better structure the project.

Another feature available on Rust is testing. In case of testing for embedded, there are different approaches available:

- Testing the drivers without accessing real hardware (using crates like *embedded-hal-mocking*)

- User and System testing inside an emulator (*QEMU* for example)
- Testing on bare metal with the help of crates such as *µtest*

Unfortunately, there is not a big support for testing in Rust as there is in C, where there exist several testing tools specific for critical industries like aerospace, which can guarantee code compliant with DO-178C. Moreover, testing inside an emulator such as QEMU is only available for Cortex-M processors without FPU.

3.3 SUMMARY

There exist coding guidelines for C like MISRA-C and its objective is to avoid writing code that leads to undefined behaviour. Since Rust provides safety in its type system and ownership model, along with the compiler's borrow checker, we consider some guidelines can be eliminated because Rust simply does not allow those coding practices. By analysing MISRA-C rules, we identified 9 rules that can be eliminated by using Rust. It is also important to refer that unsafe code does allow some undefined behaviours, meaning that guidelines for unsafe Rust will probably be necessary.

Comparing Rust with the other languages used in aerospace critical systems, we found some similarities that Rust shares with these languages. Safety wise, Rust and Spark share features such as initialisation checking or nullity checking. Comparing with C in terms of performance, we found out that similar results were achieved for both implementations.

In terms of maturity on the embedded environment, Rust is still at a fairly early state, but has a community focused on implementing several HAL, drivers, and board support crates.

4

CASE STUDIES

After a better understanding of how Rust can be influential in the critical software industry and assessing its maturity for embedded systems, some case studies on Rust embedded code were carried out to further continue this evaluation of programming languages for critical systems.

The chosen platform is the *STM32F407G-DISC1* micro-controller with *ARM Cortex-M4* 32 bit core with floating point unit, 1 Mbyte of flash memory and 192 Kbyte of RAM. It includes an *ST-LINK/V2-A* embedded debug tool, four user leds, push buttons, accelerometer and a *Digital to Analogue Converter (DAC)*.

The implementations consist of different schedulers of a *Real Time Operating System (RTOS)*, specifically preemptive and cooperative. A RTOS differs from a regular operating system because a regular operating system usually provides non-deterministic responses, which means there are no guarantees when each task is going to be completed. RTOS are used in critical embedded systems because of its deterministic execution pattern, allowing to meet real time requirements.

To better understand Rust capabilities in embedded software, we implemented first the schedulers in C and then in Rust. This allows us to verify differences in the software development process. Moreover, we implemented a driver for the accelerometer, and a program in Rust which makes use of the accelerometer, user leds on the *STM32F407G-DISC1*, and the preemptive scheduler previously developed.

4.1 PREEMPTIVE SCHEDULER

A preemptive scheduler is a type of scheduler where the operating system chooses when to interrupt a thread from running and start a different one running. This may happen because the thread has used all of the time given by the CPU, or when a thread with higher priority has entered a ready state. Preemption is often needed to guarantee fairness, and helps guaranteeing that deadlines are met, a characteristic of any (hard) real time software system. Some algorithms which are based on preemptive scheduling are the Round Robin and the Shortest Remaining Time.

Figure 4.1 illustrates the Round Robin algorithm.

Thread	Arrival Time	Burst Time	Turnaround Time
A	0	5	14
B	1	3	5
C	2	8	20
D	3	6	17

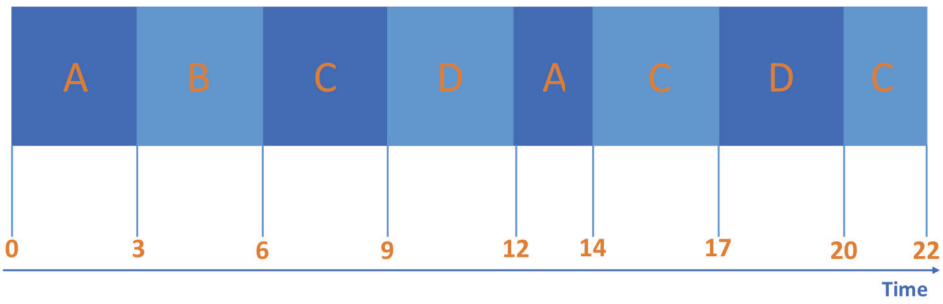


Figure 4.1.: Example of preemptive scheduling

In this example, every thread has the same priority and 3 units of time to execute (quanta). When the time slice runs out, the operating system puts the thread in waiting state and gives the CPU to the next thread.

4.1.1 *C implementation*

The first implementation is a preemptive scheduler in C. We first start by defining a *Thread Control Block (TCB)* structure with two elements: a pointer to the thread stack and a pointer to the next element on the list. The last element has a pointer to the first element, thus making it a circular linked list. There is also a variable indicating which current thread is running and an array that represents the stack of each thread.

The kernel initialisation starts by defining the value of millis prescaler with the bus frequency value divided by 1000 in order to have the quanta time in milliseconds. Then occurs the addition of threads, that consists on defining which TCB is the next, initialising the stack of each thread and assigning a value to the current thread variable. Finally, the kernel is launched when the *SysTick* is initialised. Launching the scheduler consists in determining which thread is the first and assigning the stack pointer to the first thread stack pointer.

The Systick is triggered at a fixed rate, the quanta. When an interrupt occurs, the Systick Handler will perform the preemptive thread switch by saving the content of registers and then switching the stack pointer to the next threads stack pointer.

```
#define NUM_OF_THREADS 3
#define STACKSIZE 100

struct tcb{
   int32_t *stackPt;
   struct tcb *nextPt;
};

typedef struct tcb tcbType;

tcbType tcbs[NUM_OF_THREADS];

const tcbType *currentPt;

int32_t TCB_STACK[NUM_OF_THREADS][STACKSIZE];
```

Listing 4.1: Structure of the Thread Control Block

It is important to note that in Cortex M4 processors, the registers R0 to R3, R12, LR, PC and PSR are automatically saved, which means we are saving the registers R4 to R11 manually in the Systick Handler.

```
SysTick_Handler
   CPSID   I
   PUSH    {R4-R11}
   LDR        R0, =currentPt ; r0 points to currentPt
   LDR        R1, [R0] ; r1 = currentPt
   STR        SP, [R1]
   LDR        R1, [R1,#4] ; r1 = currentPt->next
   STR        R1, [R0] ; currentPt = r1
   LDR        SP, [R1] ; SP = currentPt->stackPt
   POP        {R4-R11}
   CPSIE   I
   BX      LR
```

Listing 4.2: SysTick Handler perfoming the context switching

The *main* file is a simple program with 3 functions representing the desired threads, where each of these threads increments one global variable.

Running the scheduler and inspecting the counter variables values, we observe the expected behaviour: the variables are incremented at a similar rate.

```
uint32_t count0,count1,count2;

void Task0(void){

   while(1){
      count0+= 1;
   }

}

void Task1(void){
   while(1){
      count1+= 1;
   }
}

void Task2(void){
   while(1){
      count2+= 1;
   }
}

int main (void){
   osKernelInit();
   osKernelAddThreads(&Task0,&Task1,&Task2);
   osKernelLaunch(QUANTA);
}
```

Listing 4.3: Preemptive scheduler main file

4.1.2 *Rust Implementation*

The Rust implementation starts by identifying in which aspects this version would differ from the C one. For starters, circular linked lists in Rust are known for being problematic to implement because of the ownership system rules (if each variable can only have one owner, in a linked list every node owns the next one). As most implementations make use of pointers, and heap allocation in critical systems are forbidden, we implemented the TCB as a struct that has the same behaviour as the linked list.

This struct contains an array of Tasks (instead of a linked list) and a variable which indicates the current index of the task running, *current_task*. When we switch threads, we compare the number of the current task with *current_task*, and if it is equal to the total number of tasks, we update it to the first index of the array, thus obtaining the circular behaviour without breaking the ownership rules.

```
pub struct Task {
  stack_pointer : *mut usize,
  states: [usize; 8], //Number of registers
}

pub struct TaskCB {
    current_task : usize,
    tasks : [Task; TASK_NUM],// TASK_NUM = 3
}

fn update_to_next_task(&mut self) {
        self.current_task += 1;

        if self.current_task == TASK_NUM {
            self.current_task = 0;
        }

    }
```

Listing 4.4: Task and TCB structures

The function *switch_to_task* is responsible for saving the registers R4 to R11, what was also done in the C implementation. This function also triggers a service call interrupt. Then, the SVC handler function performs the switch, and the execution returns to *switch_to_task*, restoring the registers.

The *asm!* macro is how we use inline assembly in Rust. The first parameter is a literal string containing the instructions. The following parameters are output operands, input

```
pub unsafe extern "C" fn switch_to_task(
    mut user_stack: *mut usize,
    process_regs: &mut [usize; 8],
) -> *mut usize {
    asm!("
    msr psp, $0
    ldmia $2, {r4-r11}
    svc 0xff
    stmia $2, {r4-r11}
    mrs $0, PSP
    "
    : "={r0}"(user_stack)
    : "{r0}"(user_stack), "{r1}"(process_regs)
    : "r4","r5","r6","r7","r8","r9","r10","r11" : "volatile" );
    user_stack
}
```

Listing 4.5: Function that executes the context switching

operands, clobbers and options, all of them optional. This macro is only present in the nightly compiler and not yet stabilised.

Similarly to the C implementation, the actual program is 3 functions representing threads, each one incrementing a global variable. Note that in Rust, global mutable variables require *unsafe* blocks to perform actions. In our case, we are sure that these variables are only being mutated in each thread, but because the ownership rules ensures absence of data races and having multiple threads accessing these variables would likely result in data races, we have to use *unsafe* in our code.

Running the scheduler, the values printed are the ones expected, replicating the behaviour of the C preemptive scheduler.

DIFFICULTIES ENCOUNTERED One objective of these implementations was to try to write similar programs in C and Rust, with code and behaviour as close as possible.

As mentioned before, the circular linked list became an obstacle but quickly overtaken with the solution found. Also, if we tried to write Rust as C, we would end up with a lot of *unsafe* code, ignoring one of the strongest points in using Rust. Although we use *static mut* as global variables and enforced the use of *unsafe*, we could have instead used *Mutex*, a synchronisation primitive which ensures exclusive access to a variable.

```
static mut COUNT0 : usize = 0;
static mut COUNT1 : usize = 0;
static mut COUNT2 : usize = 0;

fn main() -> ! { // The exclamation point represents that the function never returns
    let mut cp = Peripherals::take().unwrap();

    let mut tcb = TaskCB::new();

    let mut process_task0 = unsafe {
        Task::new(TASK_STACKS[0].last_mut().unwrap() as *mut usize, task0)
    };
    (...)
    tcb.add_task(process_task2,2);

    systick::systick_start(&mut cp.SYST);

    loop {
        unsafe {
            hprintln!("c0: {} c1: {} c2: {}",COUNT0,COUNT1,COUNT2).unwrap();
        }
        tcb.scheduling();
    }
}

fn task0() -> ! {
    loop{
        unsafe {
            COUNT0+=1;
        }
    }
}
(...)
```

Listing 4.6: Excerpt from the preemptive scheduler main file

4.2 COOPERATIVE SCHEDULER

Cooperative scheduling is a style in which the operative system does not initiate the context switching from one thread to another, instead letting the threads yield control of the processor, giving it to the next thread. Some algorithms that are based on cooperative scheduling are the First Come First Serve and the Shortest Job Next.

Figure 4.2 illustrates the First Come First Serve algorithm.

Thread	Arrival Time	Burst Time	Turnaround Time
A	0	5	5
B	1	3	8
C	2	8	16
D	3	6	22

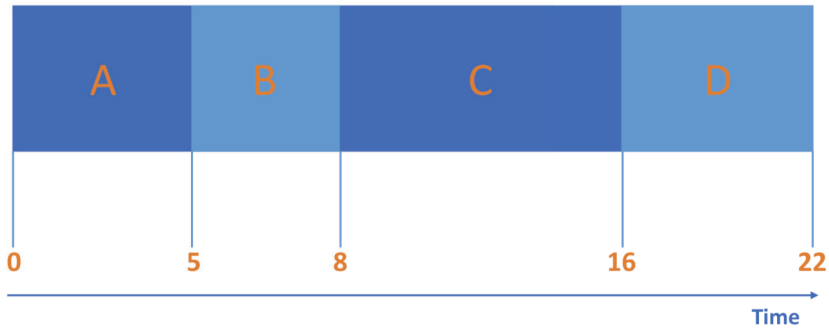


Figure 4.2.: Example of cooperative scheduling

In this example, the threads execute on the order of arrival, independent of the time. The operating system will always wait until the thread completes its execution, then allowing the next thread to execute.

4.2.1 *C implementation*

To build a cooperative scheduler we add a function called *osThreadYield*. This function writes directly a value into the interrupt control register, triggering the Systick and when an interrupt request is made to Systick, the scheduler chooses the next thread. With it, every thread can simply run this function after their task is done and giving the cpu to the following thread.

Every other aspect of the kernel is maintained (initialisation of kernel, threads and launching the scheduler), so the preemptive scheduler turns into a cooperative when every thread calls the yielding function.

```
#define INTCTRL  (*((volatile uint32_t *)0xE000ED04))

void osThreadYield(void) {
   INTCTRL = 0x04000000;
}
```

Listing 4.7: Function responsible for yielding the control of the CPU

```
void Task0(void){
   while(1){
      count0+= 1;
      osThreadYield();
   }
}

void Task1(void){
   while(1){
      count1+= 1;
      osThreadYield();
   }
}
```

Listing 4.8: Excerpt of cooperative scheduler main file

Running this scheduler and inspecting the counter variables values, we observe the expected behaviour: the variables are incremented at the same rate, only performing one variable increment before yielding the processor.

4.2.2 *Rust Implementation*

As with the C implementation, in Rust we also build a function that writes directly into the interrupt control register, while using the rest of the preemptive scheduler code.

Because we only know the address of the register, and it is not mapped in any peripheral access crate, we use the *write_volatile* function to directly write into the register.

```
unsafe fn os_thread_yield() {
    ptr::write_volatile(0xE000ED04 as *mut usize, 0x04000000);
}
```

Listing 4.9: Function responsible for yielding the control of the CPU

Likewise in C implementation, in Rust the only alteration to the code is adding the function on every thread.

```
fn task0() -> ! {
    loop{
        unsafe {
            COUNT0+=1;
            os_thread_yield();
        }
    }
}
```

Listing 4.10: Excerpt of cooperative scheduler main file

Running this scheduler, the values printed are the ones expected, replicating the behaviour of the C cooperative scheduler.

4.3 ACCELEROMETER DRIVER

We built a platform agnostic driver to interface with the LIS3DSH, the accelerometer available in the STM32F4 Discovery. The LIS3DSH is an ultra-low power high-performance 3-axis linear accelerometer, capable of measuring accelerations with output data rates from 3.125Hz to 1.6 kHz and allows for user-selectable full scales of ±2g/±4g/±6g/±8g/±16g. It provides both I^2C and *Serial Peripheral Interface (SPI)*, although it only uses SPI to communicate with our platform. The implementation uses the *embedded-hal* interface to support all the devices that implement *embedded-hal* traits. The accelerometer measures proper acceleration, that is, the physical acceleration experienced by an object.

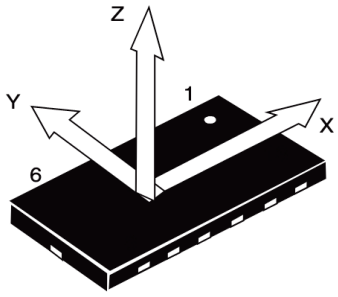


Figure 4.3.: Direction of detectable accelerations

The SPI allows half/full-duplex, synchronous serial communication with external devices interface. It uses the master-slave architecture. In our case study, the micro-controller is the master, with the accelerometer being the slave.

Several functions are written to interact with the accelerometer, with the core ones being the functions responsible for writing and reading registers, and the function responsible for returning the acceleration values.

The *write_register* function receives as parameters the desired Register and the byte of information to write. The process starts by setting the slave select to low in order to activate the slave device. Then, the information is configured and written into the register. Finally, the slave pin is set as high, deactivating the slave device. As everything went as expected, an *Ok()* is returned.

```
fn write_register(&mut self, reg : Register, byte: u8) -> Result<(), E> {
        self.cs.set_low();

        let buffer = [reg.addr() | SINGLE | WRITE , byte];

        self.spi.write(&buffer)?;

        self.cs.set_high();

        Ok(())
    }
```

Listing 4.11: Function to write bytes on a register

The *read_register* receives as parameter the desired Register to read. The behaviour is similar to the function responsible for writing on the register, but in this case, the *transfer* method performs a write and read on the register. Since we only intend to read from the register, no bytes are written on it. The value is then returned on the second position of the array, returned inside the Result type.

```
fn read_register(&mut self, reg : Register) -> Result<u8, E> {
        self.cs.set_low();

        let mut buffer = [reg.addr() | SINGLE | READ, 0];

        self.spi.transfer(&mut buffer)?;

        self.cs.set_high();

        Ok(buffer[1])
    }
```

Listing 4.12: Function to read bytes from a register

The function *accel* is the one responsible with reading the *OUT_X_H*, *OUT_X_L*, *OUT_Y_H*, *OUT_Y_L*, *OUT_Z_H* and *OUT_Z_L* registers. These contain, respectively, the most significant part and least significant part of the acceleration signals acting on the X,Y and Z axes. The values are then stored in a structure as 16 bit integers.

```
pub fn accel(&mut self) -> Result<I16x3, E> {
       let x_l = self.read_register(Register::OUT_X_L)?;
       let x_h = self.read_register(Register::OUT_X_H)?;
       (...)
       let z_h = self.read_register(Register::OUT_Z_H)?;

       Ok(I16x3 {
           x : (x_l as u16 | (x_h as u16) << 8) as i16,
           y : (y_l as u16 | (y_h as u16) << 8) as i16,
           z : (z_l as u16 | (z_h as u16) << 8) as i16,
       })
   }
```

Listing 4.13: Function responsible for obtaining the acceleration values

With this library we are able to use the accelerometer and read the acceleration values. The type system of Rust, more specifically *Result* allow us to be more confident on the robustness of the code written. Also, the initialisation requires a specific pin configuration, and thanks to the type system, a pin configuration that does not comply with the types will result in a compiler error.

4.4 PUTTING IT ALL TOGETHER

To have a better use case of the scheduler built in Rust, we implemented a program that measures the acceleration of the micro-controller and turns on the corresponding Led. The structure is the following:

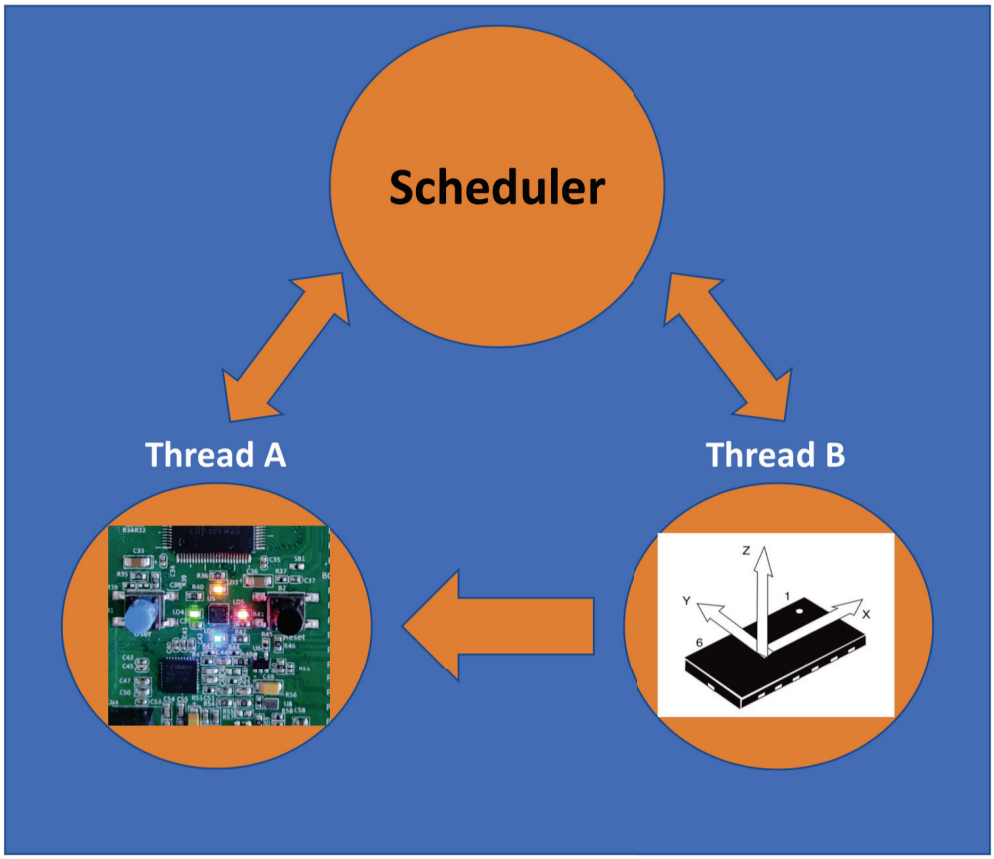


Figure 4.4.: Structure of the program

The scheduler used for this case study is the preemptive scheduler, as most RTOS implement a preemptive scheduling algorithm [16] [10]. We want to note that we also implemented the cooperative scheduler and the program did not suffer significant changes.

The initialisation of this program starts by creating the stacks for each of the threads, the TCB and initialising the systick. Because we are using user leds and the accelerometer, we need to make use of more peripherals. The crate built for the *STM32F407G-DISC1* provides us several methods to safely interact with the peripherals, like the method *take* that returns all the peripherals, modelled as singletons to ensure there is only one instance of any one of them at any given point in time, inside an *Option*. Subsequent calls to this method will result in a *None* value.

```
fn main() -> ! {
    let mut cp = Peripherals::take().unwrap();
    let dp = stm32::Peripherals::take().unwrap();
```

Listing 4.14: Example of safely obtaining the peripherals

As previously explained on 4.3, the accelerometer requires a specific pin configuration to be initialised. After checking the required pin configuration and the data-sheet of the micro-controller, we initialise the GPIOA and GPIOE in the desired configuration.

To manipulate the user leds, we once again use an abstraction available on the crate of the micro-controller. The abstraction safely initialises the leds, and provides an array containing the 4 user leds, with 3 methods to interact with them: *on*, *off* and *toggle*.

To safely use both the leds and accelerometer in the threads, we use Mutex. With it, we safely guarantee exclusive access to both of them. Moreover, another reason to use Mutex is because of the initialisation taking place in the main thread. To use the leds or accelerometer inside the thread, we had to store them in a global variable and again, and in Rust that implies several lines of unsafe code. With Mutex, not a single line of unsafe code is written.

```
static MY_LEDS: Mutex<RefCell<Option<Leds>>> =
    Mutex::new(RefCell::new(None));
(...)
let leds = Leds::new(gpiod);
interrupt::free(|cs| MY_LEDS.borrow(cs).replace(Some(leds)));
(...)
interrupt::free(|cs| {
           if let Some(ref mut leds) = MY_LEDS.borrow(cs).borrow_mut().deref_mut() {
               leds[0].on();
           }
       });
```

Listing 4.15: Leds being used inside the critical section

With *RefCell*, we can safely mutate data even when there are immutable references to that data. *RefCell* uses a runtime check to ensure only one reference to a peripheral is given out at a time. Furthermore, we use the *Option* to initialise *MY_LEDS* with a *None* value, being later replaced when the leds are initialised.

After we have both the accelerometer and leds working in each thread, we use another global variable to communicate between them using again a Mutex, but this time with a *Cell* inside because we are using a simple type, like *i16*, that can be copied. The communication is then performed by writing the values from the accelerometer in the global variable, and the other thread reads those values, turning on the leds accordingly.

Finally, the leds are turned on and off accordingly to the values read. First, we compare the absolute values of X and Y, when the first is greater than the second it means the micro-controller is either being tilted to the left of right, otherwise top or bottom. Then we compare the coordinate value with a threshold value to define which led shall be turned on. The leds are then turned off, waiting for new values read from the accelerometer.

```
static COORDINATES: Mutex<Cell<(i16,i16)>> = Mutex::new(Cell::new((0,0)));

//Thread 1
{
    interrupt::free(|cs| {
        if let Some(ref mut accelerometer) =
            ACCELEROMETER.borrow(cs).borrow_mut().deref_mut() {
            let I16x3 {x,y,z} = accelerometer.accel().unwrap();
                interrupt::free(|cs| COORDINATES.borrow(cs).set((x,y)));
            }
    });
}
//Thread 2
{
    interrupt::free(|cs| {
                if let Some(ref mut leds) =
                    MY_LEDS.borrow(cs).borrow_mut().deref_mut() {
                    let (x,y) = COORDINATES.borrow(cs).get();
(...)
}
```

Listing 4.16: Communication between the threads

```
let (x,y) = COORDINATES.borrow(cs).get();
    if x.abs() > y.abs() {
        if x > THRESHOLD_HIGH {
            leds[2].on();
        } else if x < THRESHOLD_LOW {
            leds[0].on();
        }
    } else {
        if y < THRESHOLD_LOW {
            leds[3].on();
        } else if y > THRESHOLD_HIGH {
            leds[1].on();
        }
    }

    leds[0].off();
    leds[1].off();
(...)
```

Listing 4.17: Algorithm to turn on the leds according to acceleration values

4.5 LESSONS LEARNED

Thanks to these case studies, we were able to draw some conclusions concerning using Rust in embedded software development instead of C.

First, the way Rust is designed encourages programmers to write safe code. The ownership system means memory safety, but it also means we have to think twice about the code we are writing, since code perfectly acceptable in C will not compile in Rust. It can take time to get used to the restrictions imposed by the language, but once they are understood the reasoning behind the code changes.

Programming with types is also another example of Rust's safety-by-design. Using types means catching errors at compile-time. For instance (and opposed to C), in Rust one cannot just cast a variable to another type in order to bypass the compiler. The lack of a NULL type is a big plus (Tony Hoare called NULL references "The Billion Dollar Mistake" [34]), with Rust using instead the Option and Result types which allow us to have a better handling of the code.

However, throughout this implementation phase we assessed some limitations of the language. Compilation time gets longer as projects get bigger, the debugging tools gave us a few setbacks and we had a few problems on using *Instrumentation Trace Macrocell (ITM)* to log messages from the micro-controller. Note however that some of these problems could have been caused by the Windows 10 operating system, since some debugger errors were found in this operating system and not in others (e.g. MacOS). Moreover, while developing the case studies, we found an issue with integer overflowing that caused the program to panic. In safety-critical software, this type of issues are detected by using tools that perform *Extended Static Checking (ESC)*, and in our case, could be solved by using SMACK.

Another aspect we wanted to address is the use of assembly code in Rust. Currently, there are three options to use assembly code.

- Inline assembly with the *asm!* macro is only available in nightly versions of the compiler, without estimate for stabilisation.
- Free form assembly with the *global_asm!* macro suffers from the same problem of inline assembly.
- Writing assembly code in an external file and then using an external crate (*cc*) to assemble the file into an archive file. Although the compiler version can be stable, the downside of using this approach is that the crate requires some assembler program depending on the target architecture.

In this project we used inline assembly for its simplicity, and a nightly version of the compiler still allowed us to develop the case studies.

4.6 SUMMARY

The case studies developed in this dissertation consisted in two schedulers, preemptive and cooperative, a driver for the accelerometer present on a micro-controller and finally a program that makes use of the schedulers already built, by having one thread reading acceleration values and other thread toggling leds according to the values read. We developed case studies both in Rust and C, namely the schedulers, to understand the differences between these languages. Also, the ones developed in Rust allowed us to evaluate Rust capabilities in embedded environments.

With these implementations, we were able to catch some behaviours that are normal in C but cannot be replicated exactly in Rust, like e.g. the circular linked list example, which in Rust was sorted out with a normal array replicating its behaviour. We also took advantage of safety features present in Rust. Due to global mutable variables being considered unsafe we can use the synchronisation primitive Mutex that allows us to write concurrent code without the use of unsafe blocks.

Nevertheless, throughout this work we identified some limitations of developing Rust embedded code, such as long compiling times and the use of assembly code in Rust as opposed to C.

5

CONCLUSION

The main aim of this final chapter is to discuss what was achieved in this dissertation with respect to the objectives proposed in chapter 3, taking into account the case studies carried out. We finish this by proposing a few ideas for future work, concerning the project reported in this thesis and the language itself.

5.1 ACHIEVEMENTS AND CONTRIBUTIONS

In this dissertation we proposed to answer the following research questions:

- **Objective 1** – Investigate in what measure Rust ensures software verification and meets the demands of current standards used in the safety-critical software industry.
- **Objective 2** – Assess the maturity of Rust for embedded systems (namely with respect to which processors it is available for) and compare the language with others currently used in embedded systems.

Starting with the Objective 1, we investigated how the Rust language works and what safety features it offers, and presented some verification tools that can be used alongside the compiler and borrow checker. Thereafter, we proceeded to identify on MISRA-C guidelines (which are used in the development of critical software for aerospace and other industries) rules that would be eliminated and/or alleviated by Rust's safety features, having identified 13 rules.

As previously noted, these guidelines aim at eliminating any undefined behaviour from safety-critical software. However, with *unsafe* Rust there could be guidelines identifying some common mistakes and helping developers using *unsafe* Rust, safely. Looking at the results obtained, we believe using Rust in the safety critical domain can help the verification process by eliminating part of it, but we also acknowledge that in order for Rust to be used in the aerospace industry, it has to be compliant with DO178-C, and the automatic analysis of Rust code must be target of a Tool Qualification process.

Regarding Objective 2, we examined how the Rust embedded team is organised, how mature the ecosystem is by analysing supported architectures, boards, drivers and crates.

Currently, Rust is mostly available for Cortex-M processors, but with developments on other architectures. Also, we evaluated the Rust crate system and testing features, which are non existent in C, and how it can help in the development of critical software. We believe that, for Rust to be used in the aerospace industry, its availability has to increase as well as the maturity of the whole embedded ecosystem.

Also for Objective 2, we developed some case studies both in Rust and in C on a STM micro-controller. We developed two types of schedulers, preemptive and cooperative, a driver for the accelerometer present on the micro-controller used and a bigger project that makes use of the schedulers and accelerometer driver. With these case studies we identified some differences between Rust and C code, such as structures that do not meet the ownership rules and type safety that Rust provides.

The experience of developing code in Rust was satisfying. Rust is close to other low level programming languages (e.g. C and C++) in terms of syntax, which means a short adaption period on how the language works. In the beginning, comparing with other programming languages, we spent more time analysing compiler errors due to the borrow checker not allowing some incorrect behaviours (that is, not respecting ownership and borrowing rules). After adjusting to it, the process of developing became easier and the code consequently safer. Features already mentioned several times in this dissertation (e.g. Ownership model, types specific for handling errors (Result type), absence of NULL instead using the Option type, immutability of variables by default) changed our way of thinking about software development. Furthermore, the embedded crates used (crate of the board used and embedded-HAL crate) definitely help in understanding which functions should be used and how to use them.

The work presented in this dissertation has originated a scientific paper entitled "Towards Rust for Critical Systems" [37] to appear in the 30th International Symposium on Software Reliability Engineering.

5.2 Prospect for future work

In this dissertation we were able to answer the questions proposed, and developed some case studies to use Rust on bare metal. Still, we identify the need for more work that could be done as follow up.

Future work on this project For starters, the micro-controller used has the capability of reproducing sound and a microphone. We could have implemented in our case study the micro-controller reproducing a sound depending on the movement of it. This would better test our schedulers, and the safety guarantees Rust provides concerning concurrency. Since the micro-controller used contains an audio DAC, we could have also used

it to output sounds according to the acceleration values, or even a program that would measure sound intensity by turning on the leds accordingly.

Furthermore, there is still the possibility to add more complexity to the schedulers, like assigning priority to the threads. In this case, we could use the scheduler in a more complex system and better evaluate how the RTOS behaves with performance and time measurements.

SUGGESTIONS FOR THE RUST COMMUNITY In order for Rust to become usable for critical embedded systems there is a need to certify Rust compilers and associated tool chain for use in such industry. The languages currently used in safety critical systems are stable, mature and their compilers are fairly optimised. Rust is a fairly recent language, reaching its first stable release only in 2015. To this date there is only Rust compiler (rustc). Finally, only nightly compiler allows for compilation of programs with unstable features, like inline assembly. So, we think the language needs to be more mature to be used in safety-critical. Sealed Rust [47] is a plan of Ferrous Systems to bring Rust to the safety critical domain by qualifying the language and compiler. They propose a subset of Rust with a different level of stability to be used in safety critical domains. This different level of stability would allow for specifications and validation of new releases.

Another important aspect is concerned with the use of software coding guidelines. Rust can dispense with some of the guidelines that are concerned with eliminating undefined behaviour in safety-critical software. However, there are situations where unsafe Rust has to be used, and in these cases it certainly would be beneficial to have coding guidelines for this situations.

We also reviewed some verification tools for Rust, and although it was not an objective of this dissertation, we know that testing is a important part of critical software cycle. Software verification tools for Rust are still in an early state, in particular software testing tools. C has numerous verification tools used in the industry that make sure code meet safety standards (e.g. DO-178C). Furthermore, it is common to use tools that perform ESC, which means to identify errors such as integer overflow, division-by-zero, array out of bounds or null-dereferencing. SMACK is already capable of detecting integer overflow, and although null-dereferencing is not possible in safe Rust, there is no such verification tool that allow us to identify errors as the ones mentioned. One possible solution would be to implement a tool that would use Dafny [27] (by translating Rust code to Dafny) in order to verify for divisions-by-zero and array out of bounds errors. To conclude, we believe more verification tools for Rust would be very important for its adoption in safety-critical industries.

Finally, we believe an IDE specific for embedded Rust (comparable to μVision for C) would be very beneficial for developers writing and debugging embedded software.

BIBLIOGRAPHY

[1] K. C. Addagarrala and P. Kinnicutt. Safety critical software ground rules. *International Journal of Engineering & Technology*, 7(2.28):344–350, 2018. ISSN 2227-524X. doi: 10.14419/ijet.v7i2.28.13209. URL https://www.sciencepubco.com/index.php/ijet/article/view/13209.

[2] Alloy, 2018. URL http://alloytools.org/.

[3] M. I. S. R. Association et al. *MISRA-C: 2004: Guidelines for the Use of the C Language in Critical Systems*. MIRA, 2008.

[4] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk. System programming in rust: Beyond safety. *ACM SIGOPS Operating Systems Review*, 51(1):94–99, 2017.

[5] M. Baranowski, S. He, and Z. Rakamarić. Verifying Rust programs with smack. In *International Symposium on Automated Technology for Verification and Analysis*, pages 528–535. Springer, 2018.

[6] M. Batra. Formal methods: Benefits, challenges and future direction. *Journal of Global Research in Computer Science*, 4(5):21–25, 2013.

[7] M. Becker, E. Regnath, and S. Chakraborty. Development and verification of a flight stack for a high-altitude glider in ada/spark 2014. In *International Conference on Computer Safety, Reliability, and Security*, pages 105–116. Springer, 2017.

[8] J.-L. Boulanger. *Industrial use of formal methods: formal verification*. John Wiley & Sons, 2013.

[9] R. W. Butler, J. L. Caldwell, V. A. Carreno, C. M. Holloway, P. S. Miner, and B. L. Di Vito. Nasa langley's research and technology-transfer program in formal methods. In *COMPASS'95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*, pages 135–149. IEEE, 1995.

[10] ChibiOS. Rtos concepts: Scheduling, states and priorities, 2019. URL http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts.

[11] Companies using Rust Embedded, 2019. URL https://www.rust-lang.org/what/embedded.

[12] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3): 147–148, 1968.

[13] S. Ellman. *Writing Network Drivers in Rust*. PhD thesis, B. Sc. Thesis. Technical University of Munich, 2018.

[14] FAA. Ac 20-115c. *Airborne Software Assurance*, 2013.

[15] Frama-C, 2018. URL https://frama-c.com/.

[16] R. Ghattas and A. G. Dean. Preemption threshold scheduling: Stack optimality, enhancements and analysis. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 147–157, April 2007. doi: 10.1109/RTAS.2007.27.

[17] G. J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, June 2006. doi: 10.1109/MC.2006.212.

[18] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064.

[19] Interstellar 8-Track: How Voyager's Vintage Tech Keeps Running, 2013. URL https://www.wired.com/2013/09/vintage-voyager-probes/.

[20] U. T. R. C. Ireland. Utrc - ireland fact sheet, 2019. URL http://www.utrc.utc.com/media/factsheets/FactSheet_UTRC_IrelandItaly.pdf.

[21] G.-A. Jaloyan. Safe pointers in spark 2014. *arXiv preprint arXiv:1710.07047*, 2017.

[22] G.-A. Jaloyan, Y. Moy, and A. Paskevich. Borrowing safe pointers from rust in spark. *arXiv preprint arXiv:1805.05576*, 2018.

[23] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, Dec. 2017. ISSN 2475-1421. doi: 10.1145/3158154. URL http://doi.acm.org/10.1145/3158154.

[24] S. Kan, D. Sanán, S. Lin, and Y. Liu. K-rust: An executable formal semantics for Rust. *CoRR*, abs/1804.07608, 2018. URL http://arxiv.org/abs/1804.07608.

[25] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL https://books.google.pt/books?id=161QAAAAMAAJ.

[26] B. Kernighan et al. Jpl institutional coding standard for the c programming language. *California Insititute of Technology*, 2009.

[27] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[28] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251. ACM, 2017.

[29] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 1. ACM, 2017.

[30] M. Lindner, J. Aparicius, and P. Lindgren. No panic! verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 108–114. IEEE, 2018.

[31] J. Lundberg. Safe kernel programming with Rust. Master's thesis, KTH, Software and Computer systems, SCS, 2018.

[32] K. Magel. Revisiting the impact of the ada programming language. *computer*, 50(9): 10–11, 2017.

[33] Most Popular and Influential Programming Languages of 2018, 2017. URL https://stackify.com/popular-programming-languages-2018/.

[34] Null References: The Billion Dollar Mistake, 2009. URL https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[35] Q. Ochem. Rust and spark: Software reliability for everyone. *Electronic Design*, 65: 32–37, 07 2017.

[36] S. C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[37] A. Pinho, L. D. Couto, and J. N. Oliveira. Towards Rust for critical systems. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 19–24. IEEE, 2019. doi: 10.1109/ISSREW.2019.00036.

[38] M. Rainer-Harbach. Methods and tools for the formal verification of software. *Technische Universität Wien*, 2011.

[39] C. M. Ramsay. Nasa's software safety standard. In *ESA Special Publication*, volume 599, page 507, 2005.

[40] L. Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013.

[41] Rust By Example, 2018. URL https://doc.rust-lang.org/rust-by-example/index.html.

[42] Rust Embedded Working Group, 2019. URL https://github.com/rust-embedded/wg.

[43] Rust programming language, 2018. URL https://www.rust-lang.org/.

[44] Software Complexity, Lines of Code and Digital Derby, 2009. URL http://blog.klocwork.com/software-complexity/software-complexity-lines-of-code-and-digital-derby/.

[45] SPARK Ada programming language, 2014. URL https://jj09.net/spark-ada-programming-language/.

[46] Stack Overflow Developer Survey, 2018. URL https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-languages.

[47] F. Systems. Sealed Rust announcement, 2019. URL https://ferrous-systems.com/blog/sealed-rust-the-pitch/. Accessed: 2019-07-09.

[48] R. E. R. Team. Embedded Rust github repository, 2019. URL https://github.com/rust-embedded/awesome-embedded-rust#architecture-support-crates.

[49] The Rust Programming Language, 2019. URL https://doc.rust-lang.org/stable/book/.

[50] Tock Embedded Operating System, 2018. URL https://www.tockos.org.

[51] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for Rust (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 75–80. IEEE, 2015.

[52] Use of Astrée on aviation, 2019. URL https://www.absint.com/astree/index.htm.

[53] Vermont Technical College CubeSat Laboratory, 2013. URL http://www.cubesatlab.org/.

[54] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. Krust: A formal executable semantics of Rust. *arXiv preprint arXiv:1804.10806*, 2018.

[55] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu. Formal verification of critical aerospace software. *AerospaceLab*, 2012.

[56] S. M. H. Yelisetty, J. Marques, and P. M. Tasinaffo. A set of metrics to assess and monitor compliance with RTCA DO-178C. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D2–1. IEEE, 2015.

[57] Z. Yu, L. Song, and Y. Zhang. Fearless concurrency? understanding concurrent programming safety in real-world Rust software. *arXiv preprint arXiv:1902.01906*, 2019.

A

PREEMPTIVE SCHEDULER IN C

A.1 MAIN FILE

```
#define QUANTA 10

uint32_t count0,count1,count2;

void Task0(void){

   while(1){
      count0+= 1;
   }

}

void Task1(void){
   while(1){
      count1+= 1;
   }
}

void Task2(void){
   while(1){
      count2+= 1;
   }
}

int main (void){
   osKernelInit();
   osKernelAddThreads(&Task0,&Task1,&Task2);
   osKernelLaunch(QUANTA);
}
```

A.2 KERNEL FUNCTIONS

```
#define SYSPRI3 (*((volatile uint32_t *)0xE000ED20))

void osSchedulerLaunch(void);

#define NUM_OF_THREADS 3
#define STACKSIZE 100

#define BUS_FREQ 16000000
uint32_t MILLIS_PRESCALER;

struct tcb{
   int32_t *stackPt;
   struct tcb *nextPt;
};

typedef struct tcb tcbType;

tcbType tcbs[NUM_OF_THREADS];

const tcbType *currentPt;

int32_t TCB_STACK[NUM_OF_THREADS][STACKSIZE];

void osKernelStackinit(int i) {
   tcbs[i].stackPt = &TCB_STACK[i][STACKSIZE-16];
   TCB_STACK[i][STACKSIZE-1] = 0x01000000;
}


uint8_t osKernelAddThreads(void (*task0)(void), void (*task1)(void), void
    (*task2)(void)) {
   __disable_irq();
   tcbs[0].nextPt = &tcbs[1];
   tcbs[1].nextPt = &tcbs[2];
   tcbs[2].nextPt = &tcbs[0];
   osKernelStackinit(0);
   TCB_STACK[0][STACKSIZE-2] = (int32_t)(task0);//location of pc on the stack

   osKernelStackinit(1);
   TCB_STACK[1][STACKSIZE-2] = (int32_t)(task1);//location of pc on the stack
```

```
   osKernelStackinit(2);
   TCB_STACK[2][STACKSIZE-2] = (int32_t)(task2);//location of pc on the stack

   currentPt = &tcbs[0];

   __enable_irq();
   return 1;
}

void osKernelInit(void){
   __disable_irq();
   MILLIS_PRESCALER = BUS_FREQ / 1000;
}

void osKernelLaunch(uint32_t quanta) {
   SysTick->CTRL = 0;
   SysTick->VAL = 0;
   SYSPRI3 = (SYSPRI3&0x00FFFFFF) | 0XE0000000; //Priority 7
   SysTick->LOAD = (quanta*MILLIS_PRESCALER)-1;
   SysTick->CTRL = 0x00000007;
   osSchedulerLaunch();
}
```

A.3 ASSEMBLY CODE

```
AREA | .text|,CODE,READONLY,ALIGN=2
   THUMB
   EXTERN currentPt
   EXPORT SysTick_Handler
   EXPORT osSchedulerLaunch

SysTick_Handler    ; Salva automaticamente r0,r1,r2,r3,r12,lr,pc,psr
   CPSID   I
   PUSH    {R4-R11} ; Save r4,r5,r6,r7,r8,r9,r10,r11
   LDR        R0, =currentPt ; r0 points to currentPt
   LDR        R1, [R0] ; r1 = currentPt
   STR        SP, [R1]
   LDR        R1, [R1,#4] ; r1 = currentPt->next
   STR        R1, [R0] ; currentPt = r1
   LDR        SP, [R1] ; SP = currentPt->stackPt
   POP        {R4-R11}
```

```
    CPSIE   I
    BX      LR


osSchedulerLaunch
    LDR         R0,=currentPt
    LDR         R2, [R0] ; R2 = currentPt
    LDR         SP, [R2] ; SP =currentPt->stackPt
    POP         {R4-R11}
    POP         {R0-R3}
    POP         {R12}
    ADD         SP,SP,#4
    POP         {LR}
    ADD         SP,SP,#4
    CPSIE   I
    BX      LR

    ALIGN
    END
```

B

PREEMPTIVE SCHEDULER IN RUST

B.1 MAIN FILE

```
#![no_std]
#![no_main]

entry!(main);

pub const TASK_NUM: usize = 3;
const TASK_STACK_SIZE: usize = 100;
static mut TASK_STACKS: [[usize; TASK_STACK_SIZE]; TASK_NUM] = [[0;
    TASK_STACK_SIZE]; TASK_NUM];

static mut COUNT0 : usize = 0;
static mut COUNT1 : usize = 0;
static mut COUNT2 : usize = 0;

fn main() -> ! {
    let mut cp = Peripherals::take().unwrap();
    let dp = stm32::Peripherals::take().unwrap();

    let mut tcb = TaskCB::new();

    let mut process_task0 = unsafe {
        Task::new(TASK_STACKS[0].last_mut().unwrap() as *mut usize, task0)
    };


    let mut process_task1 = unsafe {
        Task::new(TASK_STACKS[1].last_mut().unwrap() as *mut usize, task1)
    };
```

```
    let mut process_task2 = unsafe {
        Task::new(TASK_STACKS[2].last_mut().unwrap() as *mut usize, task2)
    };

    tcb.add_task(process_task0,0);
    tcb.add_task(process_task1,1);
    tcb.add_task(process_task2,2);

    let mut rcc = dp.RCC.constrain();

    let clocks = rcc.cfgr.sysclk(168.mhz()).freeze();

    systick::systick_start(&mut cp.SYST);

    let gpiod = dp.GPIOD.split();

    let mut leds = Leds::new(gpiod);

    leds[0].on();
    leds[1].on();
    leds[2].on();
    leds[3].on();

    hprintln!("Kernel Initiated").unwrap();

    loop {
        unsafe {
            hprintln!("c0: {} c1: {} c2: {}",COUNT0,COUNT1,COUNT2).unwrap();
        }
        tcb.scheduling();
    }
}

#[no_mangle]
fn task0() -> ! {
    loop{
        unsafe {
            COUNT0+=1;
        }

    }
}
```

```
#[no_mangle]
fn task1() -> ! {
    loop{
        unsafe {
            COUNT1+=1;
        }

    }
}
#[no_mangle]
fn task2() -> ! {
    loop {
        unsafe {
            COUNT2 += 1;
        }
    }
}
```

B.2 KERNEL FUNCTIONS

```
pub fn systick_start(syst: &mut SYST) {
    syst.set_clock_source(SystClkSource::External);

    syst.set_reload(159999); //rvr

    syst.clear_current(); //cvr.write(0)
    syst.enable_interrupt(); //csr
    syst.enable_counter(); //csr

}

pub unsafe extern "C" fn switch_to_task(
    mut user_stack: *mut usize,
    process_regs: &mut [usize; 8],
) -> *mut usize {
    asm!("
    msr psp, $0
    ldmia $2, {r4-r11}
    svc 0xff
    stmia $2, {r4-r11}
```

```
    mrs $0, PSP
    "
    : "={r0}"(user_stack)
    : "{r0}"(user_stack), "{r1}"(process_regs)
    : "r4","r5","r6","r7","r8","r9","r10","r11" : "volatile" );

    user_stack
}

#[no_mangle]
pub unsafe extern "C" fn SVCall() {
    asm!("
    cmp lr, #0xfffffff9
    bne to_kernel
    movw lr, #0xfffd
    movt lr, #0xffff
    bx lr
    to_kernel:
    movw lr, #0xfff9
    movt lr, #0xffff
    bx lr"
    :::: "volatile" );
}


#[no_mangle]
pub unsafe extern "C" fn SysTick() {
    asm!("
    movw lr, #0xfff9
    movt lr, #0xffff
    bx lr
    "
    :::: "volatile" );
}
```

B.3 TASKS FUNCTIONS

```
pub struct Task {
  stack_pointer : *mut usize,
  states: [usize; 8],
}
```

```
impl Task {
    pub unsafe fn new(stack_pointer: *mut usize, callback: fn() -> !) -> Self {
        Self {
            stack_pointer: unsafe { push_function_call(stack_pointer, callback) },
            states: [0; 8],
        }
    }
}

pub struct TaskCB {
    current_task : usize,
    tasks : [Task; TASK_NUM],
}

impl TaskCB {

    pub fn new() -> TaskCB {
        TaskCB {
            current_task : 0,
            tasks : [Task { stack_pointer : 0 as *mut usize , states : [0; 8]};
                TASK_NUM],
        }
    }

    pub fn add_task(&mut self, t: Task , index : usize) {
        self.tasks[index] = t;
    }

    fn update_to_next_task(&mut self) {
        self.current_task += 1;

        if self.current_task == TASK_NUM {
            self.current_task = 0;
        }

    }

    fn get_current_task (&mut self) -> &mut Task {
        &mut self.tasks[self.current_task]
    }

    fn get_next_task (&mut self) -> &mut Task {
```

```
        let i = TASK_NUM - 1;

        match self.current_task {
            i => {
                &mut self.tasks[0]
            }
            _ => {
                &mut self.tasks[self.current_task+1]
            }
        }
    }

    pub fn scheduling (&mut self) {
        for i in 0..TASK_NUM{
          let t = self.get_current_task();
          unsafe {
              t.stack_pointer = switch_to_task(t.stack_pointer, &mut t.states);
          }
          self.update_to_next_task();
        }
    }
}


pub unsafe fn push_function_call(user_stack: *mut usize, callback: fn() -> !) ->
    *mut usize {
    let stack_bottom = user_stack.offset(-8);
    write_volatile(stack_bottom.offset(7), 0x01000000);
    write_volatile(stack_bottom.offset(6), callback as usize | 1);
    write_volatile(stack_bottom.offset(5), 0 | 0x1);
    write_volatile(stack_bottom.offset(3), 0);
    write_volatile(stack_bottom.offset(2), 0);
    write_volatile(stack_bottom.offset(1), 0);
    write_volatile(stack_bottom.offset(0), 0);
    stack_bottom
}
```

C

COOPERATIVE SCHEDULER IN C

C.1 MAIN FILE

```
#define QUANTA 10

uint32_t count0,count1,count2;

void Task0(void){

   while(1){
      count0+= 1;
      osThreadYield();
   }

}

void Task1(void){
   while(1){
      count1+= 1;
      osThreadYield();
   }
}

void Task2(void){
   while(1){
      count2+= 1;
      osThreadYield();
   }
}

int main (void){
   osKernelInit();
```

```
   osKernelAddThreads(&Task0,&Task1,&Task2);
   osKernelLaunch(QUANTA);
}
```

C.2 KERNEL FUNCTIONS

```
#define SYSPRI3 (*((volatile uint32_t *)0xE000ED20))
#define INTCTRL  (*((volatile uint32_t *)0xE000ED04))

void osSchedulerLaunch(void);

#define NUM_OF_THREADS 3
#define STACKSIZE 100

#define BUS_FREQ 16000000
uint32_t MILLIS_PRESCALER;

struct tcb{
   int32_t *stackPt;
   struct tcb *nextPt;
};

typedef struct tcb tcbType;

tcbType tcbs[NUM_OF_THREADS];

tcbType *currentPt;

int32_t TCB_STACK[NUM_OF_THREADS][STACKSIZE];

void osKernelStackinit(int i) {
   tcbs[i].stackPt = &TCB_STACK[i][STACKSIZE-16];
   TCB_STACK[i][STACKSIZE-1] = 0x01000000;
}


uint8_t osKernelAddThreads(void (*task0)(void), void (*task1)(void), void
    (*task2)(void)) {
   __disable_irq();
   tcbs[0].nextPt = &tcbs[1];
   tcbs[1].nextPt = &tcbs[2];
```

```
    tcbs[2].nextPt = &tcbs[0];
    osKernelStackinit(0);
    TCB_STACK[0][STACKSIZE-2] = (int32_t)(task0);//location of pc on the stack

    osKernelStackinit(1);
    TCB_STACK[1][STACKSIZE-2] = (int32_t)(task1);//location of pc on the stack

    osKernelStackinit(2);
    TCB_STACK[2][STACKSIZE-2] = (int32_t)(task2);//location of pc on the stack

    currentPt = &tcbs[0];

    __enable_irq();
    return 1;
}

void osKernelInit(void){
    __disable_irq();
    MILLIS_PRESCALER = BUS_FREQ / 1000;
}

void osKernelLaunch(uint32_t quanta) {
    SysTick->CTRL = 0;
    SysTick->VAL = 0;
    SYSPRI3 = (SYSPRI3&0x00FFFFFF) | 0XE0000000; //Priority 7
    SysTick->LOAD = (quanta*MILLIS_PRESCALER)-1;
    SysTick->CTRL = 0x00000007;
    osSchedulerLaunch();
}

void osThreadYield(void) {
    INTCTRL = 0x04000000; // trigger SysTick
}
```

D

COOPERATIVE SCHEDULER IN RUST

D.1 MAIN FILE

```
#![no_std]
#![no_main]

entry!(main);

pub const TASK_NUM: usize = 3;
const TASK_STACK_SIZE: usize = 100;
static mut TASK_STACKS: [[usize; TASK_STACK_SIZE]; TASK_NUM] = [[0;
    TASK_STACK_SIZE]; TASK_NUM];

static mut COUNT0 : usize = 0;
static mut COUNT1 : usize = 0;
static mut COUNT2 : usize = 0;

fn main() -> ! {
    let mut cp = Peripherals::take().unwrap();
    let dp = stm32::Peripherals::take().unwrap();

    let mut tcb = TaskCB::new();

    let mut process_task0 = unsafe {
        Task::new(TASK_STACKS[0].last_mut().unwrap() as *mut usize, task0)
    };


    let mut process_task1 = unsafe {
        Task::new(TASK_STACKS[1].last_mut().unwrap() as *mut usize, task1)
    };
```

```
    let mut process_task2 = unsafe {
        Task::new(TASK_STACKS[2].last_mut().unwrap() as *mut usize, task2)
    };

    tcb.add_task(process_task0,0);
    tcb.add_task(process_task1,1);
    tcb.add_task(process_task2,2);

    let mut rcc = dp.RCC.constrain();

    let clocks = rcc.cfgr.sysclk(168.mhz()).freeze();

    systick::systick_start(&mut cp.SYST);

    let gpiod = dp.GPIOD.split();

    let mut leds = Leds::new(gpiod);

    leds[0].on();
    leds[1].on();
    leds[2].on();
    leds[3].on();

    hprintln!("Kernel Initiated").unwrap();

    loop {
        unsafe {
            hprintln!("c0: {} c1: {} c2: {}",COUNT0,COUNT1,COUNT2).unwrap();
        }
        tcb.scheduling();
    }
}

#[no_mangle]
fn task0() -> ! {
    loop{
        unsafe {
            COUNT0+=1;
            os_thread_yield();
        }
    }
}
```

```
#[no_mangle]
fn task1() -> ! {
    loop{
        unsafe {
            COUNT1+=1;
            os_thread_yield();
        }
    }
}
#[no_mangle]
fn task2() -> ! {
    loop {
        unsafe {
            COUNT2 += 1;
            os_thread_yield();
       }
    }
}


unsafe fn os_thread_yield() {
    ptr::write_volatile(0xE000ED04 as *mut usize, 0x04000000);
}
```

E

LIS3DSH DRIVER

```
#![no_std]


pub const MODE: Mode = Mode {
    phase: Phase::CaptureOnFirstTransition,
    polarity: Polarity::IdleLow,
};

pub struct LIS3DSH<SPI, CS> {
    spi: SPI,
    cs: CS,
}

impl<SPI,CS, E> LIS3DSH<SPI,CS>
where
    SPI: Transfer<u8, Error = E> + Write<u8, Error = E>,
    CS: OutputPin,
{
    pub fn new(spi:SPI, cs : CS) -> Result<Self, E> {

        let mut lis3dsh = LIS3DSH {spi, cs};

        lis3dsh.write_register(Register::CTRL_REG4, 0b01100111)?;

        lis3dsh.write_register(Register::CTRL_REG5, 0b00000000)?;

        Ok(lis3dsh)

    }

    fn write_register(&mut self, reg : Register, byte: u8) -> Result<(), E> {
        self.cs.set_low();
```

```
    let buffer = [reg.addr(), byte];

    self.spi.write(&buffer)?;

    self.cs.set_high();

    Ok(())
}

fn read_register(&mut self, reg : Register) -> Result<u8, E> {
    self.cs.set_low();

    let mut buffer = [reg.addr() | SINGLE | READ, 0];

    self.spi.transfer(&mut buffer)?;

    self.cs.set_high();

    Ok(buffer[1])
}

fn read_registers<N>(&mut self, reg: Register) -> Result<GenericArray<u8, N>, E>
where
    N: ArrayLength<u8>,
{
    self.cs.set_low();

    let mut buffer: GenericArray<u8,N> = unsafe {mem::uninitialized()};
    {
        let slice: &mut [u8] = &mut buffer;
        slice[0] = reg.addr() | MULTI | READ;
        self.spi.transfer(slice)?;
    }

    self.cs.set_high();

    Ok(buffer)

}

pub fn accel(&mut self) -> Result<I16x3, E> {
    let x_l = self.read_register(Register::OUT_X_L)?;
```

```
        let x_h = self.read_register(Register::OUT_X_H)?;
        let y_l = self.read_register(Register::OUT_Y_L)?;
        let y_h = self.read_register(Register::OUT_Y_H)?;
        let z_l = self.read_register(Register::OUT_Z_L)?;
        let z_h = self.read_register(Register::OUT_Z_H)?;

        Ok(I16x3 {
            x : (x_l as u16 | (x_h as u16) << 8) as i16,
            y : (y_l as u16 | (y_h as u16) << 8) as i16,
            z : (z_l as u16 | (z_h as u16) << 8) as i16,
        })
    }

    pub fn who_am_i(&mut self) -> Result<u8, E> {
        self.read_register(Register::WHO_AM_I)
    }

    pub fn status(&mut self) -> Result<u8, E> {
        self.read_register(Register::STATUS)
    }

    pub fn set_sensitivity(&mut self, sensitivity : Sensitivity) -> Result<(), E> {

        self.modify_sensitivity(Register::CTRL_REG4, |r| {
            r & !(0b111 << 3) | (sensitivity.value() << 3)
        })

    }

    fn modify_sensitivity<F>(&mut self, reg: Register, f : F) -> Result<(), E>
    where
        F: FnOnce(u8) -> u8,
    {
        let r = self.read_register(reg)?;
        self.write_register(reg, f(r))?;
        Ok(())
    }
}


/// XYZ triple
#[derive(Debug)]
pub struct I16x3 {
```

```
    /// X component
    pub x: i16,
    /// Y component
    pub y: i16,
    /// Z component
    pub z: i16,
}


const READ: u8 = 1 << 7;
const WRITE: u8 = 1 << 7;
const MULTI: u8 = 1 << 6;
const SINGLE: u8 = 0 << 6;

#[allow(dead_code)]
#[allow(non_camel_case_types)]
#[derive(Clone, Copy)]
enum Register {
    OUT_T = 0x0C,
    INFO1 = 0x0D,
    INFO2 = 0x0E,
    WHO_AM_I = 0x0F,
    OFF_X = 0x10,
    OFF_Y = 0x11,
    OFF_Z = 0x12,
    CS_X = 0x13,
    CS_Y = 0x14,
    CS_Z = 0x15,
    LC_L = 0x16,
    LC_H = 0x17,
    STAT = 0x18,
    PEAK1 = 0x19,
    PEAK2 = 0x1A,
    VCF_1 = 0x1B,
    VCF_2 = 0x1C,
    VCF_3 = 0x1D,
    VCF_4 = 0x1E,
    THRS3 = 0x1F,
    CTRL_REG4 = 0x20,
    CTRL_REG1 = 0x21,
    CTRL_REG2 = 0x22,
    CTRL_REG3 = 0x23,
    CTRL_REG5 = 0x24,
```

```
CTRL_REG6 = 0x25,
STATUS = 0x27,
OUT_X_L = 0x28,
OUT_X_H = 0x29,
OUT_Y_L = 0x2A,
OUT_Y_H = 0x2B,
OUT_Z_L = 0x2C,
OUT_Z_H = 0x2D,
FIFO_CTRL = 0x2E,
FIFO_SRC = 0x2F,
ST1_1 = 0x40,
ST1_2 = 0x41,
ST1_3 = 0x42,
ST1_4 = 0x43,
ST1_5 = 0x44,
ST1_6 = 0x45,
ST1_7 = 0x46,
ST1_8 = 0x47,
ST1_9 = 0x48,
ST1_10 = 0x49,
ST1_11 = 0x4A,
ST1_12 = 0x4B,
ST1_13 = 0x4C,
ST1_14 = 0x4D,
ST1_15 = 0x4E,
ST1_16 = 0x4F,
TIM4_1 = 0x50,
TIM3_1 = 0x51,
TIM2_1_L = 0x52,
TIM2_1_H = 0x53,
TIM1_1_L = 0x54,
TIM1_1_H = 0x55,
THRS2_1 = 0x56,
THRS1_1 = 0x57,
MASK1_B = 0x59,
MASK1_A = 0x5A,
SETT1 = 0x5B,
PR1 = 0x5C,
TC1_L = 0x5D,
TC1_H = 0x5E,
OUTS1 = 0x5F,
ST2_1 = 0x60,
ST2_2 = 0x61,
```

```
    ST2_3 = 0x62,
    ST2_4 = 0x63,
    ST2_5 = 0x64,
    ST2_6 = 0x65,
    ST2_7 = 0x66,
    ST2_8 = 0x67,
    ST2_9 = 0x68,
    ST2_10 = 0x69,
    ST2_11 = 0x6A,
    ST2_12 = 0x6B,
    ST2_13 = 0x6C,
    ST2_14 = 0x6D,
    ST2_15 = 0x6E,
    ST2_16 = 0x6F,
    TIM4_2 = 0x70,
    TIM3_2 = 0x71,
    TIM2_2_L = 0x72,
    TIM2_2_H = 0x73,
    TIM1_2_L = 0x74,
    TIM1_2_H = 0x75,
    THRS2_2 = 0x76,
    THRS1_2 = 0x77,
    DES2 = 0x78,
    MASK2_B = 0x79,
    MASK2_A = 0x7A,
    SETT2 = 0x7B,
    PR2 = 0x7C,
    TC2_L = 0x7D,
    TC2_H = 0x7E,
    OUTS2 = 0x7F,
}

impl Register {
    fn addr(self) -> u8 {
        self as u8
    }
}

///Output Data Rate
#[derive(Debug, Clone, Copy)]
pub enum Odr {

    Powerdown = 0x00,
```

```
    Hz3_125 = 0x10,

    Hz6_25 = 0x20,

    Hz12_5 = 0x30,

    Hz25 = 0x40,

    Hz50 = 0x50,

    Hz100 = 0x60,

    Hz400 = 0x70,

    Hz800 = 0x80,

    Hz1600 = 0x90,
}

pub enum Sensitivity {
    G2 = 0b000,

    G4 = 0b001,

    G6 = 0b010,

    G8 = 0b011,

    G16 = 0b100,
}

impl Sensitivity {
    fn value(self) -> u8 {
        self as u8
    }
}
```

F

SCHEDULER + ACCELEROMETER + LEDS

```
#![no_std]
#![no_main]

entry!(main);

pub const QUANTA: usize = 10;

pub const TASK_NUM: usize = 2;
const TASK_STACK_SIZE: usize = 10000;
static mut TASK_STACKS: [[usize; TASK_STACK_SIZE]; TASK_NUM] = [[0;
    TASK_STACK_SIZE]; TASK_NUM];

const THRESHOLD_HIGH : i16 = 650;
const THRESHOLD_LOW : i16 = -650;

pub type Lis3dsh = driver::LIS3DSH<Spi<SPI1, (PA5<Alternate<AF5>>,
    PA6<Alternate<AF5>>, PA7<Alternate<AF5>>)>, PE3<Output<PushPull>>>;

static MY_LEDS: Mutex<RefCell<Option<Leds>>> =
    Mutex::new(RefCell::new(None));

static ACCELEROMETER: Mutex<RefCell<Option<Lis3dsh>>> =
    Mutex::new(RefCell::new(None));

static COORDINATES: Mutex<Cell<(i16,i16)>> = Mutex::new(Cell::new((0,0)));

fn main() -> ! {
    let mut cp = Peripherals::take().unwrap();
    let dp = stm32::Peripherals::take().unwrap();

    let mut tcb = TaskCB::new();
```

```
// Create Tasks, creating stacks and functions
let process_task0 = unsafe {
    Task::new(TASK_STACKS[0].last_mut().unwrap() as *mut usize, task0)
};


let process_task1 = unsafe {
    Task::new(TASK_STACKS[1].last_mut().unwrap() as *mut usize, task1)
};

// Add Tasks to the Thread Control Block
tcb.add_task(process_task0,0);
tcb.add_task(process_task1,1);

// Initialization of various Peripherals
let rcc = dp.RCC.constrain();

let clocks = rcc.cfgr.sysclk(168.mhz()).freeze();

// Initialize Systick
systick::systick_start(&mut cp.SYST);

// Initialize Leds and store them in Mutex
let gpiod = dp.GPIOD.split();

let leds = Leds::new(gpiod);

interrupt::free(|cs| MY_LEDS.borrow(cs).replace(Some(leds)));

// Initialize Accelerometer
let gpioe = dp.GPIOE.split();

let mut nss = gpioe.pe3.into_push_pull_output();
nss.set_high();

let gpioa = dp.GPIOA.split();

let sck = gpioa.pa5.into_alternate_af5();
let miso = gpioa.pa6.into_alternate_af5();
let mosi = gpioa.pa7.into_alternate_af5();

let spi = Spi::spi1(
    dp.SPI1,
```

```
        (sck, miso, mosi),
        MODE,
        25.mhz().into(),
        clocks,
    );

    let mut lis3dsh = driver::LIS3DSH::new(spi, nss).unwrap();

    lis3dsh.set_sensitivity(Sensitivity::G16).unwrap();
    // Put accelerometer in Mutex
    interrupt::free(|cs| ACCELEROMETER.borrow(cs).replace(Some(lis3dsh)));

    // Assert Who Am I
    interrupt::free(|cs| {
        if let Some(ref mut accelerometer) =
            ACCELEROMETER.borrow(cs).borrow_mut().deref_mut() {
                let x = accelerometer.who_am_i().unwrap();
                assert_eq!(x, 0x3F);
            }
    });

    loop {
        // Schedule Tasks
        tcb.scheduling();
    }
}


/*
** Task responsible for turning on the leds according to the values read from the
    variable containing the readings from the accelerometer
*/
#[no_mangle]
fn task0() -> ! {
    loop{
        interrupt::free(|cs| {
            if let Some(ref mut leds) = MY_LEDS.borrow(cs).borrow_mut().deref_mut() {
                let (x,y) = COORDINATES.borrow(cs).get();

                if x.abs() > y.abs() {
                    if x > THRESHOLD_HIGH {
                        leds[2].on();
                    } else if x < THRESHOLD_LOW {
```

```
                    leds[0].on();
                }
            } else {
                if y < THRESHOLD_LOW {
                    leds[3].on();
                } else if y > THRESHOLD_HIGH {
                    leds[1].on();
                }
            }

            leds[0].off();
            leds[1].off();
            leds[2].off();
            leds[3].off();
        }
    });
  }
}

/*
** Task responsible for reading values from the accelerometer and inserting them in
    a variable to be read by the other Task
*/
#[no_mangle]
fn task1() -> ! {

    loop {
        interrupt::free(|cs| {
            if let Some(ref mut accelerometer) =
                ACCELEROMETER.borrow(cs).borrow_mut().deref_mut() {
                let I16x3 {x,y,z} = accelerometer.accel().unwrap();
                interrupt::free(|cs| COORDINATES.borrow(cs).set((x,y)));
            }
        });
    }
}
```