Bachelor's Thesis

The Rust Programming Language for Embedded Software Development

Nico Borgsmüller

Faculty:	Computer Science
Field of study:	Computer Science
Issued on:	05.11.2020
Submitted on:	15.01.2021
First examiner:	Prof. Dr. Ulrich Margull
Second examiner:	Prof. Dr. rer. nat. Franz Regensburger
Company supervisor:	Tobias Mucke

Declaration

I hereby declare that this thesis is my own work, that I have not presented it elsewhere for examination purposes and that I have not used any sources or aids other than those stated. I have marked verbatim and indirect quotations as such.

Schrobenhausen, January 15 2021

Nico Borgsmüller

Abstract

Current languages employed for embedded software development usually provide unsafe memory handling often resulting in vulnerabilities. Languages that can provide memory safety at compile-time can be used instead. This thesis examines the suitability of the Rust programming language for embedded software development. The relatively new language provides such memory safety guarantees while offering high performance comparable to other common languages. The objectives of the thesis are to assess the suitability of Rust in comparison to an existing language and to explain the necessary steps when eventually switching development to Rust.

The C language is used for the comparison, as it is currently the most common language for embedded software development. To be able to assess Rust objectively, the comparison to C is carried out on the basis of pre-defined aspects that were deemed important. This comparison is explained and discussed in the main part of the thesis. Furthermore, this part contrasts safety and performance by explaining the different safety problems as well as countermeasures and subsequently comparing an example program according to the performance in both languages. To be able to examine the switching process, another set of criteria is employed. Rust is analyzed according to them and an exemplary switching process is provided.

Comparing the criteria leads to different findings. Some aspects show a similar suitability of Rust and C. These include the popularity, the compliance with safety standards as well as the performance, although the latter highly depends on the tested application. Positive aspects of C include the large amount of available compilation targets and the maturity. Rust on the other hand stands out through a greater variety of programming paradigms, a state-of-the-art and well-adopted toolchain as well as good documentation, community support and a sound ecosystem. Furthermore the safety guarantees pose a clear advantage over C and make embedded software provably safer. It is also shown that for switching to Rust, numerous resources for employee training are available and different tools for the integration into an existing code base are offered by the programming language and its ecosystem.

The results suggest that Rust is indeed suitable for embedded software development, although the choice must not be made without careful consideration of the examined criteria.

Acknowledgements

For supporting me while writing this thesis, I'd like to thank the following people:

- Tobias Mucke, my supervisor at MBDA, for providing ideas, proofreading the thesis and enabling me to write about this topic in the first place.
- Patrizia Weidinger for providing the particle filter source code and proofreading.
- Dr. Andreas Ehmanns for giving tips on embedded development and many ideas for aspects to consider.
- Michael Erskine for giving an overview on certification and safety standards.
- Philipp von Perponcher, Lukas Meitz, Niklas Merkelt for proofreading the thesis.
- Prof. Dr. Ulrich Margull for supervising the thesis, giving helpful advise and feedback.

Contents

Acronyms

ABI Application Binary Interface. 17, 59

ANSI American National Standards Institute. 5

API Application Programming Interface. 30

ASLR Address Space Layout Randomization. 45

BCPL Basic Combined Programming Language. 5

BSP Board Support Package. 15

CCM Core-Coupled Memory. 13

CCRAM Core-Coupled Random Access Memory. 13, 14

CERT Computer Emergency Response Team. 37, 44

CIA Confidentiality, Integrity, Availability. 40

CPU Central Processing Unit. 13

CSS Cascading Style Sheets. 61

CYCCNT Cycle Count. 49

DEP Data Execution Prevention. 45

DWT Data Watchpoint and Trace Unit. 49

FFI Foreign Function Interface. 47, 59, 60, 61

FPU Float Processing Unit. 15

GCC GNU Compiler Collection. 13, 25, 30, 34, 37, 38, 52, 53, 54

GDB GNU Debugger. 12, 16, 19, 26, 27, 38, xxvii, xxviii

GNU GNU's Not Unix!. 12, 26, 29

GPIO General Purpose Input/Output. 11, 15, 16, 19

HAL Hardware Abstraction Layer. 11, 15

HTML HyperText Markup Language. 25, 26

IDE Integrated Development Environment. 6, 26, 27

IDR Input Data Register. 16, 19

IoT Internet of Things. 1, 2

ISO International Organization for Standardization. 5

ITM Instrumentation Trace Macrocell. 12

JIT Just-in-time. 3, 4, 59

JNI Java Native Interface. 59

JSON JavaScript Object Notation. 8, 27, 28

JVM Java Virtual Machine. 4

LED Light-Emitting Diode. 11, 12, 16, 17, 19, 62

LTO Link Time Optimization. 53

MAC Micro-Architecture Crate. 10, 11, 18, 34

MISRA Motor Industry Software Reliability Association. 37, 44

MSVC Microsoft Visual C++. 25, 30, 37

OOP Object-Oriented Programming. 22, 23, 24

OpenOCD Open On-Chip Debugger. 12, 16, 26, 27

PAC Peripheral Access Crate. 11, 18, 27, 34

PDF Portable Document Format. 25, 26

POSIX Portable Operating System Interface. 30

PYPL PopularitY of Programming Language. 35

RAM Random Access Memory. 13, 14

RFC Request For Comment. 7, 32

RPC Remote Procedure Call. 12

RTOS Real-Time Operating System. 26

SDK Software Development Kit. 6

SEI Software Engineering Institute. 37, 44

SIL Safety Integrity Levels. 36, 37

SPI Serial Peripheral Interface. 11, 14

SQL Structured Query Language. 22

SVD System View Description. 18, 27, 34

SWD Serial Wire Debug. 12

Tcl Tool command languag. 12

UEFI Unified Extensible Firmware Interface. 30

USB Universal Serial Bus. 12, 15

List of Figures

List of Tables

Listings

1 Introduction

Motivation

Embedded software plays an important role in today's interconnected world. Mobile devices, smart infrastructure and the *Internet of Things* (IoT) require safe and reliable systems. While former embedded devices were not connected to networks, security is becoming increasingly important due to interconnectivity. Examples include the IoT for consumers, but also the so-called Industry 4.0, where production devices are connected and capable of adapting to changes due to external factors. The importance of interconnectivity is also rising for military applications through for example a Combat Cloud [84], where multiple assets of a nation's forces are communicating with each other to exchange tactical data. In such interconnected scenarios, every vulnerability on any device could potentially impair the whole system's security.

Many security vulnerabilities are caused by poor memory safety (cf. [39]). As an example, looking at vulnerabilities in the Linux kernel [91], it becomes clear that a large number of problems could be solved by better memory handling and that these issues are also a continuous problem. Embedded programming is highly focused on C, posing risks through not guaranteeing memory safety and leaving this obligation to the developer. Recent years have also shown that regardless of high investments into improvement, such problems are not solved by supporting the developers through coding guidelines, finding issues through code analysis or other measures. In the real world, dangers like botnets are already a common occurrence. To create a botnet, an attacker automatically searches for exploitable devices, especially in the IoT, and takes control through a previously known weakness, like for instance the PureMasuta botnet [3]. Such botnets were responsible for numerous denial of service attacks in the past [140].

Most higher level programming languages that ensure save memory management, come with a huge performance overhead through the use of either a garbage collector or interpreter. The Rust programming language on the other hand promises zero-cost abstractions for guaranteed memory safety. This way, vulnerabilities through unsafe memory handling would be impossible, while still allowing the program to run as fast as without the memory safety measures. Rust also emerged to be very popular among developers in recent years. To evaluate these claims, this thesis tries to fulfill some predefined objectives.

Objectives

Switching from a well-established programming language to a fairly new alternative is no simple choice. The first goal of the thesis is to find out if and how well Rust is suited to replace programming languages already in use for embedded software development. This will compare languages according to a set of predefined criteria and result in an overview of which aspects Rust can improve on and where it is still lacking functionality. The second goal focuses on a different set of aspects to consider when eventually switching to Rust. It tries to assess what needs to be taken care of when changing programming languages.

Context

The thesis was created with a focus on the defense industry, targeting safety critical applications. Basic conditions like reliability and predictability are an important factor for software development in this area. While performance is not a crucial aspect, it must still be predictable and within reasonable bounds. Embedded software development must also follow these conditions.

The MBDA Product Cyber Security Team focuses on the security of the company's products. Preliminary examinations have shown that secure software development has to rely on a programming language addressing common safety and security issues. Although Rust is a fitting candidate for these conditions, concerns were raised for its suitability on embedded systems. This motivated the goal of providing an examination of Rust for embedded programming.

Relevance

Although Rust being an alternative to C or other programming languages is a highly discussed topic on the internet, very few comprehensive comparisons exist. An exception on one hand is [132] about Rust for the IoT, which offers a comparison of many topics, but doesn't dive deeper into the details. On the other hand, [92] examines the suitability of Rust for critical systems, also referencing embedded development. Other publications are mostly limited to specific aspects, like [16] and [18] concerning performance measurements, or compare other languages to Rust, like [35] considering the Go programming language. Especially rare is an overview of embedded related topics. With Rust becoming more and more popular in open source projects, this thesis will offer a decision base allowing to make informed decisions for introduction of Rust into industry projects.

Structure

To fulfill the objectives defined above, the thesis is structured as follows. Firstly, the compared programming languages and their basic features will be introduced. They will be further presented by explaining how to program a microcontroller and how their toolchains work. The following main chapter will evaluate technical as well as non-technical aspects comparing C and Rust. This comparison will be completed by contrasting the memory safety and performance aspects. The languages' approaches to memory safety will be presented and an example program will be tested according to different performance criteria to show how these memory safety measures might influence the program's performance. Afterwards it will be discussed, how switching to Rust can be accomplished and what aspects need to be considered for this goal. The thesis will finish, by summarizing discussed topics and giving a recommendation under which circumstances Rust can be considered a suitable replacement for the C language.

2 Programming Language Basics

This first chapter will introduce the compared programming languages as well as describe their basic and most important features. Before diving into the actual contrasted languages for this thesis, an overview of different programming languages will be provided. They each have been chosen as representatives for depicting a certain important aspect for this thesis. Table 1 provides a comparison of these languages according to some basic criteria. The following paragraphs will introduce existing memory management techniques and shortly describe each of the programming languages as well as explain why they were or weren't chosen for comparison in the whole thesis.

Language	**Aspect**	**Memory management**	**Systems program.**	**Bare-metal**
Rust	Zero-cost safety	Automatic	Yes	Yes
C	Currently in use	Manual + Automatic	Yes	Yes
Ada	Safety critical apps	Manual + Automatic	Yes	Yes
Python	Interpreted	Garbage Collected	No	No
Java	JIT Compiler	Garbage Collected	No	No
Go	Modern language	Garbage Collected	Yes	Not officially
Haskell	Functional	Garbage Collected	No	No

Table 1: Overview of different possible programming languages

There are three common types of memory management. In the first one, the automatic management, memory is automatically allocated and deallocated by the compiler without detailed instructions by the developer. This is very common for stack memory, but can also be applied to the heap section. With manual memory management on the contrary, the programmer has to take care of allocating and deallocating memory at the correct times and places. This is mostly used for managing heap memory. The third type of memory management is the garbage collection. While in this case, allocation can be either automatic or manual, deallocation is performed by the garbage collector. This part of the run-time environment stops the execution of the program in certain intervals, searches for memory that is no longer in use, and deallocates these sections accordingly.

The **C** programming language is currently the major choice for low-level or systems programming. Its manual memory management offers high flexibility for the programmer, but also requires care to be taken as memory safety is not guaranteed. C was chosen as a language for comparison in this thesis due to its popularity and widespread usage in the industry. More information can be found below.

Rust is a relatively new low-level programming language. It provides compiler-verified memory management and promises memory safety with zero-cost abstractions. Its use cases include embedded or systems development as well as network services and web development. Since Rust is the main focus of this thesis, more information can be found below.

Ada, developed by the US American Department of Defense, is especially targeted at safety-critical systems in the defense industry. It offers features for safe concurrency, compile- and run-time checks for memory safety as well as tools for program verification. It is mostly used for embedded and real-time applications in aerospace or other safety-critical industries. Although being a similar alternative to C as Rust, it is often unpopular among developers due to its verbosity and outdated programming style. It was not chosen in this thesis because many comparisons already exist. [22]

The **Python** programming language is mostly perceived as a scripting language, but can also be used for application development. It offers functional, procedural as well as object-oriented paradigms. It is very well suited for data processing and widely used for machine learning, but is also becoming popular for embedded development, especially for the Raspberry Pi. Because Python is an interpreted and garbage-collected language, it is not suited for safety-critical applications that must be predictable. Thus it is not chosen as a compared language. [96]

Java is a strictly object-oriented programming language, running on the so-called *Java Virtual Machine* (JVM). It is a *Just-in-time* (JIT) compiled language and finds its use mostly in enterprise applications or server software. Java is still very common, but again not suited for predictable software applications as it is normally garbage collected and requires a virtual machine to run. Thus it is not chosen for this thesis. [58]

Go is a programming language developed by Google for everyday use. It is advertised for its simple usage as well as fast compilation times and offers utilities for easy concurrent programming. Its ecosystem and tooling are quite elaborate and similar to Rust [78]. Use cases for Go include server software like network services, databases, web development, DevOps and more [42]. Although this language is suited for systems development, the official implementation does not support bare-metal applications and is garbage collected. It is not chosen for comparison because even though it has some similarities with Rust, it is not targeted at embedded development.

Haskell is a pure functional programming language originally developed for teaching and research, but also used in the industry in general. Its regular functions don't have any side effects and it has a strong, static type system. Many of Rust's features were influenced by Haskell, including traits (type classes), pattern matching and the expression syntax. The language's use cases include backend services like web applications or data analysis tools. Although the language handles most of the memory management tasks at compile-time, a garbage collector is still employed making it unsuitable for systems or embedded development. This is also the reason why it wasn't chosen for the comparison with Rust. (cf. [48], [139] and [123])

This overview has shown on one hand that many languages are garbage collected or interpreted and thus generally not suited for embedded or safety-critical applications. On the other hand, alternatives to the very popular C language exist. Since C is the de-facto standard for systems or embedded programming, the thesis will only look at a comparison of Rust and C with occasional references to other programming languages where considered useful.

2.1 C

This introduction to C will first give an overview of the language's history before explaining the main programming concepts. After introducing different user bases, the chapter will

finish by providing an overview on how embedded software development is approached with the C programming language.

History

The C programming language was developed from 1969 to 1978 as an replacement for the existing Assembly languages used to program Unix, which itself was eventually rewritten in C. After being influenced by BCPL and B, the C language was still changing as no standard specification existed. In 1978, the book "The C Programming Language" by Brian Kernighan and Dennis Ritchie [61] was released and was considered the de-facto standard reference for many years. In 1989 and 1990, the C98 ANSI C as well as the C90 ISO standards were released, finally providing an official standard specification. Since then, the standard was regularly adapted by the C95, C99, C11 and C17 releases. Historically being mostly used for Operating Systems, C has gained popularity in many fields of software development. Considering the C++ language being an extension to C, it is even more important in today's world. [49]

Programming Concepts

The C language itself only consists of very few keywords and provides a clean programming experience. Most of the functionality is provided through the standard library (e.g. string utilities, networking, etc.). Logic is represented by functions that can call each other [61].

C is a weakly typed language, meaning that type safety is not enforced and every type can easily be reinterpreted as every other type (through so-called type casts). C's basic types include:

- Integers of different sizes: `char` (8bit), `short` (16bit), `int` (32bit) and `long long` (64bit) with each type existing in a signed as well as an unsigned variant.
  Example: `int answer = 42;`

- Floating point types in a 32bit (`float`) and 64bit (`double`) variant.
  Example: `float pi = 3.14159265f;`

- Arrays: Represented as a consecutive list of data in memory. Elements are accessed using the address of the first element and adding the required offsets. Example:
  `int[] fib = {1, 1, 2, 3, 5, 8};`

- Strings are represented as arrays of `char` (character) elements in C.

- Pointers: One of the most important data types in C are pointers. These contain an address of a memory location that can be accessed by dereferencing the pointer variable. This offers high flexibility to the developer, but also poses safety risks as there is no check if the pointed-to data is valid or the address can be accessed at all. Nevertheless, this feature is especially useful in embedded programming to access memory-mapped registers. Example: `int* REG = (int*) 0x12345678; int data = *REG;`
  Pointer arithmetic allows basic calculations with pointers (like addition or multiplication). To access an array for example, either the array access syntax can be used (e.g. `int fib3 = fib[2];` to access the third element) or the requested address can be computed by using pointer arithmetic (`int fib3 = *(fib + 2);`), which in this case increases the address by 2 times the size of the referenced data type (e.g. 32bits for an `int`).

- Structs are used for integrating multiple variables of potentially different types together into a single data type.
- Enums (Enumerations) are a type that can represent one of multiple declared constant values.

Another important aspect to consider is the memory management techniques employed. In C, automatic memory management is used for stack variables. The required stack space is already considered during compilation and automatically reserved in the stack frame when the corresponding function is called. The memory is no longer valid after the end of the respective function. For managing memory space on the heap, manual memory management is required. The developer needs to request a memory section of a certain size by using a call to the standard library (e.g. `malloc`). After the space is no longer needed, it must be deallocated using the function call `free`.

The last aspect to consider for the C language is the toolchain. To compile a C program, all .c files are to be separately compiled using a C Compiler. Subsequently, they need to be linked together into an executable or library file using a Linker. If additional libraries need to be included in the binary, they must be explicitly defined using the `-l` flag of the Linker. The final result will either by an executable file in the target's format, or a library to be linked into other software products.

User Bases

The C community consists of many different user bases. The most important are:

- Embedded developers: The capability to directly access registers and peripherals as well as the low resource usage, while at the same time allowing the use of many higher level control flow structures and abstractions, make C a well-suited choice for embedded programming.
- Operating system/driver developers: C allows to integrate assembly code where needed and offers fine-grained control of data structures and memory in general, thus making it a good choice for OS and driver development
- Performance-critical/real-time applications: The deterministic nature and high performance of the language in combination with the compiler's optimizations make C a valuable tool for this group of developers.

Embedded

As explained above, C offers many features for embedded development. Apart from them, abstraction libraries and *Software Development Kits* (SDKs) for accessing the hardware of microcontrollers are almost always available directly from the vendor. These include definitions of register addresses and abstractions for accessing peripherals. Many vendors even offer their own *Integrated Development Environments* (IDEs) to provide a simplified developer experience (e.g. the very common Arduino IDE). Although many of these features are very helpful, the unchecked and possibly unsafe handling of memory can pose a risk on bare-metal systems that are not protected by an operating system.

2.2 Rust

This chapter will explain the basics of the Rust programming language to give an overview of the differences and similarities to C.

History

According to a talk by Steve Klabnik [62], Rust's history can be divided into four epochs. The first epoch from 2006 to 2010 can be called "The personal years". Rust was a personal project of the inventor Graydon Hoare with the goal of developing a memory safe programming language. The strong type system and the concept of immutability by default were already in the language, although it still included a garbage collector, making it unsuitable for certain environments. The second phase from 2010 to 2012, called "The Graydon years", represents an epoch, where Graydon Hoare was still the primary maintainer, but the language was already developed as a Mozilla project. Steady improvements on the core concepts led up to "The typesystem years" from 2012 to 2014 where Rust's type system was majorly overhauled and improved. Many features from the standard library were moved into external libraries to follow the philosophy of small separated libraries. With Graydon Hoare stepping back, Rust became a community driven project, introducing an *Request For Comment* (RFC) process to propose language changes. The epoch from 2015 to 2016 is now called "The release year" as Rust 1.0 was released in 2015. Development focused not only on language stability, but also explicitly on the ecosystem, user friendly tooling and the ever-growing community. [63]

Since the 1.0 release, development is divided into editions. After the first edition in 2015 marks the end of the release epoch, the second edition released was the 2018 edition, summarizing changes from the years 2016 to 2018. It was focused on general improvements of the language, the compiler, the tooling and much more. The 2018 roadmap defined four major domains that should be addressed in the following years. The focus was on network services, command line applications, WebAssembly and development for embedded devices. It is already planned to release the next edition in 2021, summarizing these development efforts. [117]

Even though Rust has changed drastically over the years, the goal to provide a concurrent, safe, systems programming language always stayed the same.

Programming Concepts

The main promise of Rust is to provide memory safety guarantees at compile-time, rendering a large number of programming mistakes impossible. A default Rust program implicitly uses this safe subset of the programming language, but to have more possibilities when needed (e.g. to interact with existing C code), unsafe Rust is available, too. It must be explicitly enabled for specific parts of the code and allows the programmer more degrees of freedom that are not as strictly checked by the compiler.

Rust provides a standard library for basic functionality just like C. What's special though is that for operating system related functionality Rust utilises the C standard library available on the target system.

Logic in Rust is represented through standalone functions (subroutines) as well as implementation functions belonging to a specific data type or trait. Variables are declared using the `let` keyword and are immutable by default. E.g. `let a = 5;` defines an integer variable that cannot be modified, whereas `let mut a = 5;` would allow modifications. In Rust's type system, types are generally inferred implicitly at compile-time so that the explicit notation of data types on variable declaration is only necessary sometimes. For example: `let a : u16 = 5;` In addition to that type casts are always explicit, meaning expressions like `5.3 / 3` are not allowed because both numbers have different data

types. A correct version could look like `5.3 / 3 as f32`, contributing to the strong type safety of the Rust programming language [90]. Rust's basic types include: [120]

- Integers of different sizes, each in a signed (`i8, i16, i32, i64`) as well as unsigned (`u8, u16, u32, u64`) variant. Additionally the `usize` and `isize` types are defined with sizes corresponding to the register width of the current architecture.
- There are also two floating point sizes `f32` and `f64`.
- A boolean type `bool` with the values `true` and `false` is also a part of the language.
- Rust provides first-class string types, even with full UTF-8 support (`str` and `String`).
- As a systems programming language, Rust also supports arrays represented as consecutive elements in memory. These elements can only be accessed through a specific array syntax (e.g. `a[3]`), but not through calculating addresses manually (at least in the safe subset of Rust).
- The Rust language also includes a tuple type, so that returning multiple values is easily achieved (e.g. `(5, "hello")`).
- Just like in C, structs are available for holding different types of data, however in Rust, they are also allowed to be empty. In addition to that, they can have related implementations and also implement traits (see below).
- Rust enums are an example of algebraic data types, meaning that they can represent different types of data. Each enum item can have different types associated with it. The following example represents some arbitrary data structure (e.g. parsed JSON):

```
enum Data {
  Int(i32),
  Float(f32),
  String(String)
}

fn do_stuff() {
  let dat = [Data::Int(5), Data::Float(2.3), Data::Int(7)];
}
```

Listing 1: Algebraic data types with enums (Rust)

  The `dat` variable is now of type `[Data; 3]`, representing an array of three arbitrary `Data` objects. Every enum item can contain data of a different type, which can only be accessed when the item type is checked beforehand,
  e.g. `if let Data::Int(num) = dat[0] { return num; }`.

- Another important aspect of Rust's type system are references, that are also called borrows. These can either be mutably or immutably borrowed values.
  So for example `let mut x : u32 = 5; let b = &x;` results in `b` having the reference type `&u32`, where `b` is not allowed to be modified (immutable). Borrowing `x` like this: `let c = &mut x;` gives the type `&mut u32` and allows modifications of the value. These reference types are different to C pointers as they are always guaranteed to point to valid data (see paragraph about ownership below). It is also worth noting that to guarantee safety either multiple immutable or a single mutable borrow can exist at any given time.

Rust also includes a powerful trait system, which allows to abstract common behaviour. Types that implement the `Eq` trait for instance can be checked for equality using the `==` operator and types implementing the `Add` trait can be added together using `+`. Some traits can be implemented automatically (derived), if certain conditions are met. To derive the `Eq` trait on a struct using `#[derive(Eq)]` for example, all members must also implement this trait. To abstract their own common behaviour, programmers can also define their own traits. To enable polymorphism, the trait type can be used instead of the real type. When static dispatch is used, the types must be known at compile-time. This is represented through generic functions like for instance:

```
fn equal<T>(obj1: T, obj2: T) -> bool where T : Eq {
    obj1 == obj2
}
```

Listing 2: Generic function with trait bound (Rust)

This example only allows types that implement the `Eq` trait to be passed as arguments. When using dynamic dispatch, the actual type is only known at run time and denoted as `fn do(obj: &dyn TraitName)`.

Another special feature of Rust is that almost every statement can also be used as an expression. So for instance while `call();` is a statement, removing the semicolon results in an expression. This is especially useful as the last expression of a block is always returned [65]. For example:

```
fn max(a: u32, b: u32) -> u32 {
  if a > b {
    a
  } else {
    b
  }
}
```

Listing 3: Using expressions for the return value (Rust)

The if/else block returns either `a` or `b` and since it is the last expression of the function, the value of the if/else expression is used as the function return value. This feature is extremely useful for functional programming, where solely expressions should be used to prevent side-effects.

Rust's main approach to safety is the ownership model. Every value has a single owner denoted as a variable. Memory is only valid as long as the owner lives and automatically dropped (freed) as soon as the owner goes out of scope. To pass the value around (e.g. into a function), it can either be moved, where the new location is the new owner and the old variable can no longer be used, or borrowed as a reference, where the old owner stays valid. Every reference has an explicit (for example when put into a struct) or implicit (when the compiler can infer it) lifetime and is not allowed to live longer than its owner. This concept allows to verify at compile-time that every reference is valid and no two mutable references to the same object can exist at the same time. Another safety relevant feature are panics that represent unrecoverable errors (e.g. a division by zero or an array access that is out-of-bounds). These are handled cleanly by the runtime and can print an error message or possibly a backtrace to help with debugging. Furthermore, Rust provides a standard way to break its strict rules (e.g. working with raw pointers) by introducing unsafe sections of code that are not as rigidly checked by the compiler. This reduces the amount

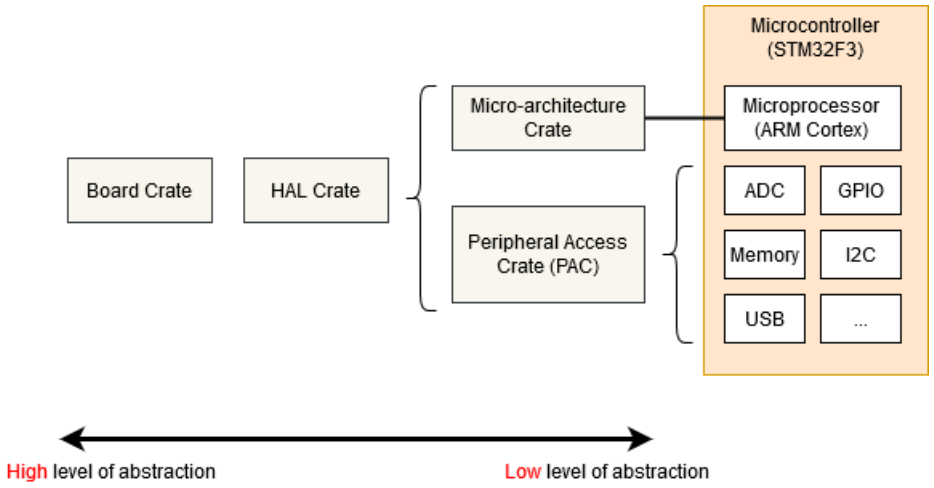

Figure 1: Hierarchy crate types for embedded development [119]

of code that needs to reviewed for safety issues to some clearly defined and auditable regions.

Rust's compilation toolchain works similarly to C, but also shows some important differences. Source files are firstly processed by the Rust compiler, although only the main file needs to be provided. Referenced source files are found automatically due to a clearly defined folder structure, where the module name equals the file name of the respective source file. The outputs of the Rust compiler are a low-level representation of the code and subsequently given to the LLVM compiler and linker for assembly code generation and linking. External libraries are linked together automatically when defining them using the Cargo build system, but they can also be defined manually through an `-l` flag similarly to the C compiler. [124]

User Bases

The Rust community consists of many former developers of different languages, which can be separated into three categories. The first category consists of former C or C++ programmers with a focus on low-level, systems or embedded programming. Another group of programmers are former users of scripting languages like Python, who switched for the high performance of Rust, while still being able to use a modern and capable language for data processing. The last category are former functional programmers now focusing on a language more capable for lower-level applications, since a functional programming style (e.g. like in Haskell) is easily possible in Rust.

Embedded

Since embedded development has been one of the focal areas of the Rust roadmap since 2018 [117], it is worth taking a look at how such development is approached within Rust. Since direct memory access is necessary, the use of unsafe Rust is required. It is advisable to quickly abstract these accesses for keeping the room for errors as small as possible. An example on how such safe interfaces are implemented can be found in [93], but is out of scope for this thesis. The Rust ecosystem promotes a de-facto standard for the classification of embedded crates (=libraries). As seen in figure 1, five different crate types are defined:

- *Micro-Architecture Crate* (MAC): Low-level crate type allowing access to the peripherals of the processor (e.g. controlling interrupts on a cortex-m processor)

- *Peripheral Access Crate* (PAC): Safe wrapper around most of the controllers peripherals, used for accessing memory-mapped registers and often generated automatically if the required data is available.
- Embedded *Hardware Abstraction Layer* (HAL): Not a library by itself (and thus not in the figure), but a collection of traits for unified interfaces to hardware functionality (e.g. a *Serial Peripheral Interface* (SPI) that can be used independently of the actual hardware).
- HAL crate: An implementation of one or multiple traits of the Embedded HAL according to specific hardware, mostly making use of PACs for reading and modifying registers.
- Board crate: To provide support for a complete board, board crates often summarize all the available hardware by including the relevant HAL, PAC and MAC crates. They for example preconfigure certain GPIO ports as LEDs or buttons according to the actual conditions.

It should be noted here that while many of these terms are also common in C programming, they are not as standardized or strictly used as in the Rust ecosystem.

The following code snippet is an example of Rust's zero-cost abstractions in the context of embedded development. Accessing a register happens through type-safe interfaces when the correct libraries are used. The following code sets the GPIO pins number 9 and 10 on port E as active, by using the `bsrr` (Bit Set/Reset Register) register of the `gpioe` register set:

```
let periph = stm32f303::Peripherals.take().unwrap();
periph.GPIOE.bsrr.write(|w| {
    w.bs9().set_bit()
     .bs10().set_bit()
});
```

Listing 4: Setting bits in memory mapped registers (Rust)

The above code allows the programmer to modify registers without having to use actual addresses, any kinds of binary operators or temporary variables. These data structures are highly abstracted, often posing a performance overhead when no optimizations are enabled. The data structure for the `bsrr` registers for instance looks as follows:

```
bsrr: stm32f303::gpioe::BSRR {
  register: vcell::VolatileCell<u32> {
    value: core::cell::UnsafeCell<u32> {
      value: 0x0
    }
  }
}
```

Listing 5: The BSRR register's data structure in Rust

When compiled with all optimizations enabled though, the above code results in a direct memory access with only a single write operation, showing a perfect use-case for Rust's zero-cost abstractions.

3 Programming a Microcontroller

This chapter will provide an example on how a microcontroller can be programmed with either language. A bare-metal device was chosen as it will most likely provide better insights into differences of the programming languages. Development supported by an operating system would show more similarities with normal application development and will not be part of this thesis. Nevertheless could a investigation on such systems give valuable insights.

The hardware used is the STM32F3DISCOVERY board by STMicroelectronics (datasheet [31]), containing the ARM Cortex M4 controller STM32F303VC (datasheet [114]). The board also consists of several LEDs, two buttons, an accelerometer, a gyroscope and a magnetometer. It also contains a second microcontroller for debugging (an ST-LINK with *Serial Wire Debug* (SWD) interface), and two USB ports. One for the debugger and a user USB port that can be accessed by programs running on the microcontroller. This first example will introduce embedded development with both languages by using a simple program that allows someone to turn on a LED by pressing the button.

The following paragraphs will now explain the common steps that had to be taken before being able to program the development board. All development was performed on Linux, so the first step was to configure the udev (Linux' device manager) rules, so that every user on the system is allowed to access the device. This can be done by creating a new rule file in the `/etc/udev/rules.d/` directory. The following content must be filled into it to set the permissions of the device file when the device with the given vendor and product id is connected:

```
# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", MODE:="0666"
```

Listing 6: udev rule to be able to access the microcontroller on Linux

The microcontroller can now be connected via the ST-Link USB port. To connect with the debugger, the *Open On-Chip Debugger* (OpenOCD) utility [89] can be used like this: `openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg`.
This software needs to stay open to be able to flash and debug programs. OpenOCD provides a telnet interface on port 4444 to execute manual commands using the scripting language Tcl, a *GNU Debugger* (GDB) server on port 3333 for the GNU debugger to connect to and an RPC server on port 6666 for automated Tcl scripting. Another important aspect for debugging embedded applications, is the ability to log text messages from the program. The following methods exist:

- The user USB port could be used to send serial messages.
- Many ARM microcontrollers provide the so-called *Instrumentation Trace Macrocell* (ITM), which is a debugging tool for logging messages.
- The third approach is semihosting. This is a technique to forward IO calls to the host computer of the debugging session. This approach was chosen because it seemed fairly easy. The program on the microcontroller uses the ARM `bkpt` (breakpoint) instruction to communicate with the host computer. The debugger now translates these calls to open a file or to write to a handle. To log messages, the stdout descriptor is opened and the returned file handle is written to.

The following chapters will now explain the approach to program a bare-metal ARM microcontroller with both programming languages.

3.1 ...in C

Installation

To get started, a cross compiler for the ARM platform needs to be installed. In this case, the *GNU Compiler Collection* (GCC) toolchain will be used, requiring the `arm-none-eabi-gcc` executable.

Configuration

The first step to configure the project is to create a Makefile for automation of the build process. The Firstly, the Makefile firstly contains definitions to configure the compiler and linker to use correct CPU target as well as standard library settings and secondly describes the steps that must be taken to generate the final binary file. The following compiler flags have to be provided:

- `-mcpu=cortex-m4`: Set the CPU target
- `-mthumb`: Generate Thumb assembly instructions instead of ARM instructions (for smaller binary sizes)
- `-mfloat-abi=hard`: Use hardware support for floating point calculations
- `-g`: Add debug information (doesn't increase flash usage as it is only used on the host)
- `-ffreestanding`: Assume that no standard library is present when optimizing the code
- `-Wall`: Log all warnings

The linker must be provided with the same `-m...` arguments as above, but also with some additional flags:

- `-nostdlib`: Do not include a standard library
- `-specs=nosys.specs`: Link against standard library stubs
- `-Tstm32.ld`: Use the linker script `stm32.ld` for section definitions

The complete file and all other relevant files can be found in appendix B.

The next step is to create the linker script, which configures the linker to move all the given information, like code and static variables into the defined sections. These sections are configured with memory addresses, describing where they will be placed on the actual target. The first section must define the memory layout according to the datasheet [114, pp. 51 - 54]. The following peripherals of the STM32F303VC are provided for storing data:

- FLASH (Starts at: 0x08000000. Size: 256KB): Read-only memory containing the program and read-only static variables.
- CCRAM (Starts at: 0x10000000. Size: 8KB): This small and fast, so-called *Core-Coupled Memory* (CCM) can contain data or the program stack.
- RAM (Starts at: 0x20000000. Size: 40KB): The main memory of the controller contains most of the data and can also contain the stack.

Afterwards, the sections of the binary file and their memory placement have to be defined:

- `vector-table`: The vector table must be placed at the origin of FLASH memory. It contains an address to the beginning of the stack and the address of the program's entry point (The Reset function). This address is extracted from the program code by using the `vector_table.reset_vector` section (see below).
- `text`: After the `vector-table`, the program's code is placed in the FLASH.
- `rodata`: Read-only data follows in the FLASH.
- `bss`: This section will be placed in RAM and contains all static variables that are zero-initialized. This must also be done during startup.
- `data`: This RAM-section will be filled with pre-defined data from FLASH.
- `stack`: The stack is not explicitly defined in the linker script, but starts at the end of RAM and grows towards lower addresses. If 8KB are enough for the stack, it can also be very convenient to place it in the CCRAM memory to separate the stack from the data section.

Startup Code

The next step is to create the startup code that is invoked when the microcontroller is reset (started) and sets up the controller before jumping to the main function. It will be placed in `startup.c` and contains a function called `startup()`. This function will be referenced by a static reset vector variable:

```
unsigned int * startup_vec __attribute__
((section(".vector_table.reset_vector"))) = (unsigned int *) startup;

void startup() {}
```

Listing 7: Creating a pointer to the startup code (C) (startup.c in Appendix: p. xxx)

The section attribute declares the section where this variable should reside. The linker script includes this section at the beginning of flash memory, after the stack address. This will allow the microcontroller to read the address of the first instruction when starting.

The startup code must now perform the following steps:

1. Initialize the `bss` section with zeros.
2. Copy the pre-defined data from FLASH (`rodata` section) to RAM (data section)
3. Call the main function `void main() {}`. It can then contain arbitrary code for the actual program.

`startup.c` should also contain handler methods for hardware interrupts or exceptions, although they were left out for simplicity here. Exceptions are problems in the controllers execution flow, including for example access to an invalid address or the use of an undefined instruction. Interrupts can be fired by many different events, and can mostly be configured by the user. Examples include timers that have run out or data that was received (e.g. on the SPI bus). When an interrupt or an exception happens, normal execution flow is interrupted and execution jumps into the handler routine. After the routine has returned, normal execution flow resumes. Since the interruption can happen at any point of the program, the rules of safe concurrency need to be applied. A list of possible interrupts and exceptions can be found in the datasheet [114, pp. 285 - 288].

Using Libraries

After explaining some of the basics without using any foreign code, the following paragraphs will now use a library to simplify development. The STM32CubeF3 library is provided directly by STMicroelectronics [113] and contains modules for accessing the ARM core, definitions for peripheral addresses, hardware abstraction layers (HALs) for the peripherals, board support packages (*Board Support Packages* (BSPs)) for different boards including the STM32F3DISCOVERY as well as high level middlewares e.g. for the USB port and many example projects. No self-written startup code or linker script is needed anymore. In addition to that, the provided code now contains full interrupt and exception handlers as well as additional initialization code for setting up the system clock and the *Float Processing Unit* (FPU). The main function is also automatically called by the library's startup code. The library also simplifies register access by providing preprocessor constants. An example on how to configure a GPIO port in output mode looks as follows:

```
GPIO_TypeDef* gpioe = (GPIO_TypeDef) GPIOE_BASE;
uint32_t temp = gpioe->MODER; // Get current mode value
temp &= ~(GPIO_MODER_MODER8_Msk); // Reset mode for port 8
// Set mode to push/pull for port 8:
temp |= GPIO_MODE_OUTPUT_PP << GPIO_MODER_MODER8_Pos;
gpioe->MODER = temp; // Set the new mode value
```

Listing 8: Modifying a register (C)

The constants and structs are provided by the library like this (only a short excerpt is shown):

```
// ...
#define GPIOE_BASE            (AHB2PERIPH_BASE + 0x00001000UL)
#define GPIO_MODER_MODER8_Pos (16U)
#define GPIO_MODER_MODER8_Msk (0x3UL << GPIO_MODER_MODER8_Pos)
// ...
#define GPIO_MODE             (0x00000003U)
#define GPIO_MODE_OUTPUT_PP   (0x00000001U) /* Mode Output Push-Pull */
// ...
typedef struct
{
  volatile uint32_t MODER;  /* GPIO mode register, Offset: 0x00 */
  volatile uint32_t OTYPER; /* GPIO output type register, Offset: 0x04 */
  // ...
} GPIO_TypeDef;
// ...
```

Listing 9: Register definitions (C)

This manual way of access is often abstracted through macros:

```
MODIFY_REG(GPIOE->MODER, GPIO_MODER_MODER8_Msk,
           GPIO_MODE_OUTPUT_PP << GPIO_MODER_MODER8_Pos);
```

Listing 10: Using macros for modifying a register (C)

Even though this way of controlling registers is often used, helper functions from the hardware abstraction layer provide a safer alternative without the need to directly access memory addresses. To finish the integration of this library, the C and Assembly sources required for compilation as well as the header file include path must be added to the Makefile.

Example Program

After all these prerequisites, writing the actual example program is fairly easy. The goal is to turn on an LED when the button is pressed and to turn it off when it is released. The steps to achieve this are as follows:

1. Turn on the clock for the GPIO ports. This includes GPIO port A for the button and GPIO port E for the LED:
```
MODIFY_REG(RCC->AHBENR, RCC_AHBENR_GPIOAEN_Msk,
 1 << RCC_AHBENR_GPIOAEN_Pos);
MODIFY_REG(RCC->AHBENR, RCC_AHBENR_GPIOEEN_Msk,
 1 << RCC_AHBENR_GPIOEEN_Pos);
```

2. Set the mode for the button GPIO pin to input (pin 0):
```
MODIFY_REG(GPIOA->MODER, GPIOA_MODER_MODE0_Msk,
 GPIO_MODE_INPUT << GPIO_MODER_MODE0_Pos);
```

3. Set the mode for the LED GPIO pin to output (LEDs are pins 8 to 15):
```
MODIFY_REG(GPIOE->MODER, GPIOA_MODER_MODE15_Msk,
 GPIO_MODE_OUTPUT_PP << GPIO_MODER_MODE15_Pos);
```

4. Construct an infinite `while` loop around the following steps because a program running on a microcontroller should never end

5. Wait until the button is pressed by reading the *Input Data Register* (IDR) register:
```
while((GPIOA->IDR & GPIO_IDR_0) == 0) {}
```

6. Turn on the chosen LED:
```
SET_BIT(GPIOE->BSRR, GPIO_BSRR_BS_15);
```

7. Wait until the button is no longer pressed:
```
while((GPIOA->IDR & GPIO_IDR_0) == 1) {}
```

8. Turn off the chosen LED:
```
SET_BIT(GPIOE->BSRR, GPIO_BSRR_BR_15);
```

The `c/lib/main.c` code can be found in the appendix on page xxxi.

Testing the Program

To test the above program, it needs to be flashed onto the board and can then be debugged. First of all, use `make` to run the Makefile which will compile all input files and link the resulting object files together into a single binary file with the correct layout. Start GDB with the binary file as an argument and connect to the OpenOCD GDB server by using the `target remote :3333` command. Now flash the binary onto the microcontroller with the `load` command and reset the controller to set the instruction pointer to the beginning of the program by using `monitor reset halt`. To simplify debugging, a breakpoint can be set to the main function (`break main`). Use the `continue` command to let the program run until it reaches main. Since they are needed very regularly, these commands can be put into a GDB script (e.g. `openocd.gdb`) and specified on the GDB invocation by using the `-x` option. The debugger can now be used to step through the program running remotely and inspect its execution state. When pressing the button, the LED turns on. When the button is released, the LED turns off.

3.2 ...in Rust

The following paragraphs will explain the necessary steps to reach the same goal of turning and LED on and off in Rust.

Installation

To install the development toolchain, Rust can either be installed through the system's package manager, or preferably with the rustup utility [107]. The latter approach allows to install different toolchains and compilation targets. This chapter will use the default toolchain with `rustup toolchain install stable` and the following compile target to build code for an ARMv7 controller with hardware supported floating point (hf) and the ARM Thumb instruction set: `rustup target add thumbv7em-none-eabihf`

Configuration

To create a new project, Rust's package manager Cargo can be used:
`cargo new --edition 2018 --bin app`. This initializes a default folder structure and creates the main file `main.rs`. To tell the compiler that no standard library should be included, because no operating system is available, the main file contains the `#![no_std]` annotation. Because the main interface will be defined differently, the `#![no_main]` annotation also needs to be added. To set the target for this project and define that a custom linker script is required, the following content needs to be written into the `.cargo/config` file:

```
[build]
target = "thumbv7em-none-eabihf"

[target.thumbv7em-none-eabihf]
rustflags = ["-C", "link-arg=-Tstm32.ld"]
```

Listing 11: Cargo configuration file for the ARM target (in Appendix: p. li)

The target must be one of those listed in [125]. Further information on targets can be found in section 4.2.3 Compilation Targets. The linker script is mostly the same as for linking the C program, although the entry point must be explicitly listed by using `ENTRY(Reset);`. Otherwise, the optimizer would get rid of the function as it is not called anywhere.

Startup Code

Just like in C, the reset function has to be declared and its address must be written to the vector table:

```
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR : unsafe extern "C" fn() -> ! = Reset;

#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {...}
```

Listing 12: Creating a pointer to the startup code (Rust) (in Appendix: p. li)

The `Reset` function has several modifiers: Firstly, `extern "C"` switches the *Application Binary Interface* (ABI) to a C compatible version, allowing the microcontroller to jump into the function without having to consider Rust specifics. The `unsafe` keyword

declares this function as unsafe, which means that some strong safety checks are not performed because memory addresses need to be dereferenced directly. The return type `!` denotes a divergent function, meaning that the function is not allowed to finish at any time. The `#[no_mangle]` attribute prevents mangling of the function name, which results in a function that is exactly called "Reset" in the final binary. The `link_section` attribute places the `RESET_VECTOR` in the `vector_table.reset_vector` section so it can be moved to the beginning of FLASH memory by the linker script.

The reset function handles initialization of `bss` and data sections just like in C, although helper functions are available to simplify this process. Start and end addresses of the sections are available in `_sbss` and `_ebss` as well as `_sdata` and `_edata`. `_sidata` denotes the start of the `rodata` section:

```
let ct = &_ebss as *const u8 as usize - &_sbss as *const u8 as usize;
ptr::write_bytes(&mut _sbss as *mut u8, 0, ct);
let ct = &_edata as *const u8 as usize - &_sdata as *const u8 as usize;
ptr::copy_nonoverlapping(&_sidata as *const u8, &mut _sdata as *mut u8, ct);
```

Listing 13: Initializing the data and bss sections (Rust) (in Appendix: p. li)

Normally, there would be the implementation of exception and interrupt handlers needed, but this will not be covered here. A feature that's special to Rust is the panic handler, which is executed on unrecoverable errors like for example a division by zero. It is declared as follows:

```
#[panic_handler]
fn panic(panic: &PanicInfo<'_>) -> ! {...}
```

Listing 14: Defining a panic handler (Rust) (in Appendix: p. li)

Using Libraries

To use a higher level of abstraction, crates (Rust libraries) are now included. The low-level crates use small sections of unsafe code to directly access the memory addresses (e.g. memory-mapped registers) and provide a safe interface on top. Since Rust follows a philosophy of providing many small, self-contained libraries, multiple crates must be included (see `Cargo.toml` in appendix on page lii). The following types of crates are relevant:

- MAC: Crates `cortex-m` [25] for peripheral access to the ARM Cortex-M processor as well as `cortex-m-rt` for the runtime (startup code etc.) can be included.
- PAC: Include the `stm32f3` crate [112]. Memory and type safe register access with zero-cost abstractions is provided. This means that writes to read-only registers are prevented at compile-time. This crate, like many others, was automatically generated from an *System View Description* (SVD) file provided by the vendor.
- Panic handler: The panic behaviour can be chosen through many existing crates. Using the `panic-halt` crate halts the processor when a panic occurs.

  The `panic-semihosting` crate on the other hand, prints the encountered error messages over the semihosting debugger.

Since the above-mentioned libraries support more than a single microcontroller, the exact memory layout still needs to be configured in a file called `memory.x`, which is part of the linker scripts. The section layout is already covered by the libraries though and does not need to be provided here.

Example Program

The following steps need to be taken for the example program to perform the same actions as the C example:

1. Get a handle of the stm32f303 peripherals:
   ```
   let periph = stm32f303::Peripherals::take().unwrap();
   ```
2. Turn on the clock for the GPIO ports A and E:
   ```
   periph.RCC.ahbenr.modify(|_, w| {
     w.iopaen().set_bit().iopeen().set_bit();
   });
   ```
3. Set the mode for the button GPIO pin to input (pin 0):
   ```
   periph.GPIOA.moder.modify(|_, w| w.moder0().input());
   ```
4. Set the mode for the LED GPIO pin to output:
   ```
   periph.GPIOE.moder.modify(|_, w| w.moder15().output());
   ```
5. Construct an infinite loop around the following steps so the program never ends.
6. Wait until the button is pressed by reading the IDR register:
   ```
   while periph.GPIOA.idr.read().idr0().bit_is_clear() {}
   ```
7. Turn on the LED:
   ```
   periph.GPIOE.bsrr.write(|w| w.bs15().set_bit());
   ```
8. Wait until the button is released:
   ```
   while periph.GPIOA.idr.read().idr0().bit_is_set() {}
   ```
9. Turn off the LED:
   ```
   periph.GPIOE.bsrr.write(|w| w.br15().set_bit());
   ```

The `main.rs` can be found in the appendix on page lii The main difference to C is that every register access is already type checked at compile-time and no magic values are used, making many accidental issues impossible.

Testing the Program

To build the Rust program, `cargo build` is used. GDB takes again care of flashing and debugging the program. The same GDB script `openocd.gdb` as with C is used here. To further improve the development workflow, the GDB command line call can be added to `.cargo/config`, so that building and flashing the program can be done with `cargo run`. It can be seen that when pressing the button the LED turns on and when releasing the button, it turns off again.

4 Comparison of Languages

After having introduced the C and Rust programming languages and described how to use them exemplarly, this section will now compare the two programming languages for their suitability for embedded software development. At first, the methodology of this section will be explained. Afterwards, the actual criteria, separated into technical and non-technical aspects, will be examined. The following chapter will now analyze the safety measures of both languages and test how they may influence the application's performance.

4.1 Methodology

Through developing an example program, the previous chapter has provided an overview of both languages. To be able to compare Rust and C more objectively, a defined set of criteria must be considered. The following criteria are a mixture of personal experiences and the results of surveying colleagues as well as some other sources like [11], [134], [132] and [34]. The following list provides an overview of all considered aspects and denotes the reasons why a specific criterion was chosen for further examination or why it was not. Technical aspects are shown first, followed by the non-technical aspects. The aspects safety and performance are placed into a separate chapter due to their importance for the thesis.

- **Safety**
  Being the main reason why the Rust language is considered, this aspect will compare both languages according to their susceptibility to different types of safety issues.

- **Performance**
  To be a suitable alternative to C, Rust must offer similar performance in spite of the additional safety measures. This aspect will consider multiple dimensions, such as time spent, memory used and more while testing an example program.

- **Programming Paradigms**
  Programming Languages offer different possibilities which can be classified using Programming Paradigms. This aspect will compare the availability of paradigms in the two programming languages.

- **Tooling**
  Considers how well a developer is supported by suitable tooling through investigating the availability of language-specific tools in different categories.

- **Targets**
  The compilation target availability may largely influence the suitability when specific hardware is required, being especially important in embedded development.

- **Functionality**
  This aspect would compare the available functionality in the ecosystem by defining categories of required features and finding suitable libraries for both languages. Since virtually every feature is available in both languages, this aspect was not covered in the thesis. No valuable conclusions could be drawn from this criterion.

- **Real-time Requirements**
  Although an important aspect for many embedded applications, it was not chosen due to its complexity and the limited scope of this thesis. A suitable example program would need to be written and tested using appropriate tooling.

- **Ease of Use & Productivity**
  This first non-technical aspect will compare how easy it is to work with both languages especially referring to documentation and the development workflow. It will also consider how these properties influence productivity.

- **Maturity**
  To be suitable for the industry in safety-critical applications, the language, tooling and ecosystem must be sufficiently mature.

- **Popularity**
  Popularity in the industry and among developers can be an important indication for language suitability.

- **Standards & Certification**
  Certifications can be important for certain customers or industries, especially for safety-critical products. This criterion will reference many of the previous aspects to give an indication of standards conformity.

- **Cost**
  Even though cost is an important factor for technological decisions, this aspect was not chosen because the most important compilers, libraries and other tools are available free of charge. There also exists almost no proprietary software in the Rust ecosystem making a comparison to C very difficult.

As one might notice, many of these aspects are generic and applicable to many types of software. Since embedded development not only includes bare-metal hardware, but is also used with operating systems like Linux, an embedded programming language needs to be examined from a broader perspective. This means that the examination in the following chapters will contain generic as well as embedded-specific aspects.

To be able to assess the criteria, a broad research was performed. Due to Rust being a quite recent language, very few scientific resources could be found. For this reason, many blogs and other websites were used for information gathering. Additionally, the experiences gained during the practical parts of this thesis were incorporated into all aspects.

4.2 Technical Aspects

Starting with the comparison, technical aspects are considered first. This includes an investigation of different programming paradigms the two languages support, available tooling as well as the support of compilation targets.

4.2.1 Programming Paradigms

Programming paradigms are used to differentiate specific programming styles. Every programming language can most likely be used to implement multiple paradigms. Paradigms define for instance how the goal of the program is described (e.g. through instructions or a declarative syntax), how the programs are structured (e.g. in functions) or how interaction with data takes place. This chapter will first introduce different relevant programming paradigms, before comparing C and Rust on how well they support each of them. [135]

Paradigms

- Structured Programming
  Being the most common programming paradigm, programs adhering to it make use of a program flow that consists of sequences of statements. The flow can be altered through conditional statements and loops. Structured programming is compromised when instructions are used that break out of the control flow.

- Imperative Programming
  In Imperative Programming, statements express step by step how the program should operate. It can be seen in contrast to Declarative Programming, where only the outcome is formulated, while the actual execution steps are determined by the run-time environment (e.g. SQL).

- Procedural Programming
  Being a type of Imperative Programming, programs following the Procedural Programming paradigm separate their statements into procedures. These procedures can be called from any point in the control flow of other procedures and thus make code reuse and modularity easily possible.

- Object-Oriented Programming (OOP)
  This type of Imperative Programming combines data and related logic into so-called objects. Instead of procedures, statements are now grouped into methods that have direct access to the object's data.

- Functional Programming
  Functional Programming is highly influenced by mathematical functions. Programs adhering to it consist of procedures with no side effects (called pure functions). Programs are build as a tree structure of function calls. This is mostly a Declarative Programming paradigm, although it exists in many different variations. Functions are first-class citizens of the language and can be passed around like variables.

- Concurrent Programming
  To allow for multiple execution contexts, data sharing and synchronization utilities must be available.

C

The following list will provide an overview on important programming paradigms that can be achieved by using the C language:

- Structured Programming
  C's flow can be altered through conditional statements like `if` or `switch` and loops like `while` or `for`. Keywords like `goto` and early function returns are possible. Thus C can mostly be considered a structured programming language, although it is not enforced.

- Imperative Programming
  C is an imperative programming language as statements are executed one after another.

- Procedural Programming
  Procedures are represented by functions.
  (e.g. `void doSteps() { step1(); step2(); }`).

- Object-Oriented Programming
  C does not offer support for this paradigm, as it is not possible to assign functionality to its data structures. It should be noted though, that the C++ language is mainly built around this paradigm by introducing classes and inheritance.

- Functional Programming
  Functional Programming is possible in C when being strict, but not all typical features like for example closures (anonymous functions) are available.

- Concurrent Programming
  Support for concurrency is not an explicit language feature, but very common by using operating system utilities. Data sharing between concurrent execution flows mostly happens through shared mutable memory, where synchronization is optional and has to be performed manually.

Rust

The following list examines how much Rust adheres to the programming paradigms explained above:

- Structured Programming
  In respect to this paradigm, Rust is very similar to C with programs made of sequences of statements being controlled by conditions and loops. It doesn't provide as many features to break this paradigm (e.g. no `goto` keyword), but is still not a perfectly structured programming language (e.g. early return statements are possible).

- Imperative Programming
  This paradigm is completely fulfilled and at the same level as C.

- Procedural Programming
  The existence of functions that can call each other also fulfills this paradigm just like C.

- Object-Oriented Programming [32][121]
  Rust is mostly an object-oriented programming language as it provides structs that can have associated functions called implementations. These structs can also implement multiple traits which are similar to interfaces. Although implementation inheritance is intentionally not part of the language, Rust's support for polymorphism as well as composition and thus code reuse still fulfills the object-oriented programming paradigm.

- Functional Programming [86]
  Functions are first-class citizens in the Rust language as they can be passed around like a variable. Furthermore, Rust provides anonymous functions in the form of closures. Rust encourages the use of pure functions (functions with no side-effects) through its expression syntax (see below), immutability by default and the absence of global mutable variables. An example are iterators for the processing of data items:

  ```
  // Sum square of numbers from 1 to 100
  (1..101).map(|x| x * x).fold(0, |x, y| x + y)
  ```

  This iterator chain is created by a range notation to iterate over the numbers 0 to 100. Every number is now mapped to its square and finally folded into a single output value by summing up all squares.

- Concurrent Programming [130]
  Rust concurrency is often described as "fearless concurrency" due to its compile-time safety guarantees. Rust provides three safe data sharing possibilities, completely prohibiting unsynchronized access in safe Rust. Immutable static memory can always be accessed from multiple execution flows. Access to mutable shared memory is only possible with synchronization, which is compiler checked and encoded in the type, through for instance a mutex that owns the shared data. The third possibility is message passing by moving data through for example channels that are part of the standard library.

Conclusion
As seen above, both languages are structured, imperative and procedural languages, although Rust is more strict with the Structured Programming paradigm. Object-Oriented Programming is possible in Rust but not in C. Functional programming can theoretically be done in both languages, Rust however provides additional language structures to make it easier to use. Both languages are well suited for concurrent programming, although Rust enforces synchronized memory handling during compilation. In conclusion one can say that Rust offers a wider variety of programming paradigms by explicitly supporting functional as well as object-oriented programming.

4.2.2 Tooling

With software applications getting bigger and more complex, more elaborate tooling is required. This includes tools that are independent from the programming language like code versioning or issue management, but also language specific tools like package managers or the compilers themselves. Even though this aspect is not directly related to the programming languages, it largely affects the productivity when working with them and thus needs to be considered when assessing the suitability of Rust and C.

The chapter will start by introducing and explaining different categories of required or convenient tools. Furthermore, it will present and compare existing solutions for C and Rust before demonstrating a selection of tools with an example.

Categories of Tools
The following categories of tools were deemed relevant for this aspect, however the list doesn't aim to cover all possible tools:

- **Compiler**: The compiler is the software used to translate the actual source code files into machine code. Common compiler toolchains often include a linker, which is used to combine different files of machine code into one executable. The compiler toolchain is expected to produce well-optimized code while still ensuring correctness and do so quickly to allow for fast development cycles.

- **Test Framework**: To be able to assert correctness of certain parts of a program, unit tests should be written by the developer. A test framework is then needed to support writing and executing these tests. It may offer the ability to write stubs (small replacements of complex functions), mock objects (replacements for data objects) and to provide statistics on the tested code.
- **Documentation Generator**: Code documentation can be exported in a more readable format to for example provide other developers with an interface specification without handing out the complete code base. Such generators often create HTML pages, but can also generate PDF files or other formats.
- **Build System**: A build system makes sure that according to a configuration, multiple source files, libraries and possibly other resources are combined into a single binary (which can be an executable or a library). Build systems frequently provide support for other tools, like e.g. executing a test framework, running a formatter, performing static code analysis, etc.
- **Package Manager**: As applications are getting more complex, they often depend on a large amount of software dependencies. A package manager takes care of downloading those, keeping them up to date and maintaining a list of currently used package versions. The package source can either be a global, public repository on the Internet, or a private one hosted in a company's network for instance.
- **Embedded Tooling**: This category includes tools for flashing programs onto embedded devices, debugging them and more.

Tools in C

- **Compiler**: A large amount of different compilers for the C language exist, but the following list only contains the most common as well as a choice of certified compilers [41]:
  - GCC: Existing since 1987, this is one of the oldest existing C compilers. It is open source and very commonly used on Linux platforms. It supports C, C++, Ada, Go and many more languages, targeting a large variety of platforms (see section 4.2.3 Compilation Targets). To allow for cross-compilation, multiple toolchains can either be installed manually or through the package manager of the OS. Example: Using the command `arm-none-eabi-gcc` instead of `gcc`, targeting ARM processors with no operating system and the embedded application binary interface.
  - Clang: This open source compiler, existing since the year 2000, uses LLVM as its compiler backend just like Rust. It offers faster compilation times than GCC, but benchmarks show that GCC's optimizations are generally better [41]. Additionally Clang only supports C, C++ and Objective-C, even though more languages are using LLVM as their compiler backend (e.g. Rust, Julia or Swift). The compiler provides more readable compile errors and user friendly diagnostic information. It can directly be used for cross-compilation by providing so-called target triples. Example: `armv7em-none-eabihf` being more specific than GCC, targeting ARM v7 with hardware floating point (hf) support.
  - *Microsoft Visual C++* (MSVC): Microsoft's C compiler exists since 1983 and is proprietary software. It can compile code only to Windows targets and offers no real cross-compilation (even though compilation on target machines can be integrated into Visual Studio).
  - CompCert: This compiler is verified according to the C standard, providing proven standards conformity [24].

  - VxWorks Cert Edition: To compile code for the VxWorks *Real-Time Operating System* (RTOS), Windriver provides a compiler certified according to multiple safety standards, allowing to use their product in regulated industries [137].

- **Test Framework**: There are many different options, all with varying levels of build system integration. Widely used examples include the Googletest (very common) [43] and Unity (targeted at embedded environments) [131] frameworks.

- **Documentation Generator**: The most popular tool, which can be seen as a de-facto standard, is Doxygen. It provides the ability to generate HTML, PDF and various other formats for numerous different programming languages [33].

- **Build System**: Again, many different systems exist. Some of the most popular are: [60]
  - Make: A recipe called Makefile needs to be provided which defines dependencies between files and their compilation order (see section 3 Programming a Microcontroller). Many of the other systems generate Makefiles at the end, but provide a higher layer of abstraction.
  - GNU Autotools: This is GNU's set of programming tools to create portable source code, but it is known for its high complexity.
  - CMake: LLVM's build system provides platform-independent configuration and fast incremental builds.
  - MSBuild: Microsoft's build system is well-integrated into the Visual Studio IDE and can compile applications for the .NET framework and Microsoft Visual C++.

- **Package Manager**: There is no package manager with wide adoption, but different options with rising popularity exist. The first example is conan, containing around 640 packages (as of November 2020) [99] and being well integrated with multiple build systems like CMake, MSBuild or the GNU Autotools. Another example is vcpkg (Visual C++ Pkg), containing around 1400 packages (as of November 2020) [99], supporting MSBuild and CMake.
  Although these solutions exist, they are not widely accepted. According to [80], only 20% of developers in the C/C++ community use package managers. The reasons given include the use of closed software as well as complicated configuration, insufficient documentation and low adoption.

- **Embedded Tooling**: No well-integrated solutions exist. Either a standalone IDE like the Arduino or STM32Cube IDE should be used, or a combination of separate programs like OpenOCD and GDB must be utilized.

As this overview shows, the C ecosystem offers many high quality tools for software development, but due to their low adoption and high fragmentation, the daily use becomes impractical. Rust on the other hand tries to provide a highly integrated, well adopted and easy to use toolchain to overcome these obstacles.

Tools in Rust

- **Compiler**: The Rust ecosystem provides only a single compiler with full support called rustc. This could be caused by the language's low stability (see section 4.3.2 Maturity), as it would be hard to keep up with the development of multiple compilers. While different C compilers comply to a standard, Rust only provides a reference manual and thus doesn't allow any statement on standards conformity. Since the Rust compiler performs optimizations over the whole crate (e.g. inlining across object file borders) and the high complexity of the borrow checker, compilation times tend to be relatively long in comparison to C (see section 4.4.3 Performance).

- **Test Framework**: The Rust compiler includes a test mode for running unit tests provided in a test module. Libraries like µTest provide utilities to simplify testing on bare-metal targets [6].

- **Documentation Generator**: Rustdoc, being integrated into the toolchain, generates navigable documentation web pages for the current crate and all its dependencies. Another interesting feature is the placement of code in documentation comments, which will be compiled and tested, ensuring that all code examples in the documentation are always up to date. [127]

- **Build System**: The major build system of Rust is called Cargo. It includes packages, manages their compilation, can run tests and benchmarks, generate documentation and even install further subcommands from a remote repository.

- **Package Manager**: Cargo downloads its crates from `http://crates.io`, which currently contains about 50.000 crates (November 2020). When deploying crates to the repository, documentation is also deployed to `http://docs.rs`. The integrated nature with the build system and other tools makes it very easy to build applications including numerous dependencies. As the compiler needs to build all dependencies with rustc and the general philosophy is to create many small, self-contained crates, a clean build can sometimes take a very long time (around one to five minutes for even small applications).

- **Embedded Tooling**: The Rust ecosystem provides no standard or de-facto standard way to work with embedded devices, but extensive documentation and tutorials provide easy to follow explanations on how to set up such development environments using OpenOCD and GDB. Even though they are currently not very common, there are still tools like probe-rs that combine the process of flashing and debugging into a Cargo integration [86].
  Since Rust doesn't have as many PACs as C and they are most likely not provided by the manufacturer, it may be necessary to create them manually. The tool svd2rust can simplify this process by generating register access libraries from SVD files [115].

The Rust toolchain also includes a linter (Clippy), a formatter (rustfmt) and an IDE integration (Rust Language server). This means that Rust provides a highly integrated and de-facto standard toolchain for the most common development tasks.

Example
The following example shows how including dependencies and building a project can be done with some of the above-mentioned tools. The goal is to include a library for serializing a struct to JSON in C and deserializing in Rust as well as building both projects using a build system. The complete examples can be found in Appendix B (`c/tools/` (p. xlix) and `rust/tools` (p. lxvi))
For the C example, the cmake build system and the conan package manager will be used. For serializing to JSON, the json-c library will be used. Steps:

1. Create a project folder.

2. Create `toolDemo.c` with code for the data structure and include `json-c/json.h` file.

3. Create `CMakeLists.txt` with the following content:

```
project(toolDemo)
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)
add_executable(toolDemo toolDemo.c)
target_link_libraries(toolDemo CONAN_PKG::json-c)
```

Listing 15: CMake configuration for including the json-c library

4. Create conanfile.txt with the following lines:

```
[requires]
json-c/0.13.1
[generators]
cmake
```

Listing 16: Conan configuration for including the json-c library

5. Create a folder called `build` and run the following commands to build the project: `conan install .. --build=json-c`, `cmake ..` and `cmake --build .`

6. The output file can now be found in `build/bin` and executed using `./toolDemo`.

Rust will use Cargo both as the build system and package manager. For JSON, the serde and serde-json libraries will be included. Steps:

1. Create a project folder using `cargo new tool_demo --bin`

2. Edit `main.rs` with code for the data structures and include `serde::Deserialize`

3. Edit `Cargo.toml` for adding `serde_json = "1.0"` and, to be able to automatically generate the serialization helpers `serde` as dependencies:
`serde = { version = "1.0", features = ["derive"]}`

4. Run `cargo build` to compile the project and its dependencies. It can be executed using `cargo run`. The binary can then be found at `target/debug/tool_demo`.

The Rust configuration with only two lines that must be added manually is much shorter and more intuitive than the two files for C with nine lines in total. It also posed a challenge to combine conan and cmake, as the documentation did work as expected on the first try.

Conclusion
While the C community has produced numerous tools during the long time that C exists, there were no tools designed together with the language during its beginnings. Even today, C developers refrain from using tools like package managers that are common with most modern programming languages. Even when such tools are used, the communities are fragmented and no de-facto standard exists. The Rust developers however made it part of their design philosophy to provide a well-integrated and easy-to-use toolchain closely linked to the programming language that is indeed popular among their community members.

4.2.3 Compilation Targets

After having discussed the different types of existing tools in the last chapter, this chapter will now take a closer look at the compilers and which targets they are able to create code for. This firstly includes the processor architecture of the target system and secondly the used standard library. In addition, the chapter will consider the portability of code between different targets for both C and Rust.

Architectures
For languages that are translated to machine code, compilers always need to target a specific processor architecture to create the correct assembly code. The target can be the current host machine itself or the architecture of a different device to perform so-called cross-compiling. The most common architectures are x86 for AMD or Intel processors and the ARM architectures used by many manufacturers especially for embedded or lower-end devices. To be able to use a language for a specific target device, the processor's target architecture must be supported by an appropriate compiler. While C supports many targets through a large number of different compilers or compiler versions, Rust uses the same compiler for all supported targets. The support is separated into three tiers, where Tier 1 means "guaranteed to work", Tier 2 "guaranteed to build" (but tests aren't run), with Tier 3 only having very basic support without any guarantees. [125]

Standard libraries
Other important aspects when targeting a specific system are the operating system (OS) and the standard library. Standard libraries provide implementations of common algorithms and data structures as well as abstractions of operating system interfaces like file systems or networking. For the C language, numerous implementations of the standard library exist, including the GNU libc (glibc), the Microsoft Visual C (MSVC) run-time library, the musl libc, the Android C standard library Bionic and standard libraries focused on embedded devices like uClibc, Newlib or Nanolib. Rust on the other hand uses its own standard library which depends on the C standard lib of the current operating system for memory allocation, networking and more. On Linux, Rust only supports the GNU libc and musl standard libraries, making it harder to work on embedded Linux platforms, where e.g. uClibc is installed. On Windows, the Microsoft Visual C and MinGW standard libraries are supported. On bare-metal embedded devices, no standard library is available since there is no operating system that would support multithreading, file systems or other utilities. C solves this problem by bundling required libraries with the executable, where most commonly Newlib or Nanolib are in use. Rust takes a similar approach by bundling the so-called Core library with the executable. Core is a subset of the Rust standard library not including any features related to operating systems or memory allocation.

Supported Targets
What follows is a list of supported architectures of C and Rust with occasional references to standard libs where deemed informative: [125][111] As the table shows, it can generally be assumed that a C compiler exists for every possible computer architecture. A new architecture can barely be seen as finished as long as no C compiler exists. Rust support, however, is still very sparse as only the most common operating systems and standard libraries are well-supported. Nevertheless can adding support for targets with only the Rust Core (called `no_std` targets) be an easy task if the target is already supported by LLVM. Integrating the full standard library can be a harder task though. In conclusion this means that when using Rust, devices must be chosen according to the compiler support for the given architecture while this restriction does not exist for C programming.

Target	C	Rust
x86 (32bit) + x86_64	Y	Tier 1: Linux (glibc), Windows (MinGW, MSVC), 64bit MacOS Tier 2: musl standard lib; iOS, Android and most desktop OSes Tier 3: e.g. UEFI, VxWorks and 32bit MacOS
ARM 64bit (aarch64)	Y	Tier 1: only Linux with glibc Tier 2: Most other OSes and std libs Tier 3: e.g. FreeBSD
ARM 32bit (+Thumb instruction set)	Y	Tier 2: mostly, e.g. bare-metal targets Tier 3: e.g. FreeBSD
Mips, PowerPC, RISC-V, Sparc	Y	Tier 2: Only the most common variations Tier 3: Others
nvptx (NVIDIA CUDA)	Y	Tier 2
WebAssembly	Y[1]	Tier 2
AVR	Y	Unofficial fork: Rust-AVR [102]
Blackfin, PA-RISC, PDP-11 and many more	Y	No official support

Table 2: List of supported targets for Rust and C

Portability
Even though a stable language definition should ensure that code works in every compiler and standard libraries try to abstract utilities for every system, reality has shown that portability is indeed a problem. The topic can be separated into portability between different compilers (on the same architecture and operating system) and portability between different architectures or operating systems (using the same compiler).

Different C compilers offer specific extensions that are often not compatible with each other. Most standard features work the same, but ambiguities in the standard definition may be implemented differently. Providing attributes to functions or variables, for example to configure a function as exported or set the memory alignment of a variable, is one of many differences. In MSVC, attributes are declared using `__declspec(...)`, while GCC and Clang use the `__attribute__(...)` modifier. Although it should be noted that GCC and Clang nowadays support both options for portability reasons. Since the Rust ecosystem only offers a single compiler, there are no such portability issues present.

Different architectures can have varying pointer sizes, differ between little and big endian number interpretation and much more. Operating systems adhere to different standards and calling conventions, most notably Windows and Linux. They especially show large differences related to features going beyond what's defined in the C language standard. An example is the API to create and manage threads. While on Windows, a new thread is created using the `CreateThread(...)` function from the `processthreadsapi.h` header, Linux uses the POSIX variant `pthread_create(...)` from the `pthread.h` header. Portable software often bypasses these differences through the use of preprocessor conditionals. Even though Rust tries to abstract more methods using the standard library, system specifics (especially in the embedded domain) still need to be considered (e.g. different memory-mapped registers). Thread creation in Rust however has a single interface `thread::spawn(...)` for all operating systems.

Conclusion
Due to the long history as well as the enduring prevalence of C in the modern programming world, almost every possible target will have a suitable C compiler available. Rust, being a very recent language, only provides a handful of targets with full support and some more that are only partially supported. Even though Rust improves the portability as only a single implementation of the language exists, not all target differences can be represented through abstract interfaces of the language. The large amount of available targets for C poses a clear advantage over Rust, especially when specific hardware is required for a project.

4.3 Non-technical Aspects

The previous chapters only considered hard technical facts. To get an overall impression on the suitability of both programming languages, a wider picture is required. The following criteria consist of non-technical aspects and will investigate how easy it is to use the languages, how mature and popular they are and how well they fulfill certain requirements for certification purposes.

4.3.1 Ease of Use & Productivity

This first non-technical aspect will analyze challenges for the daily development process. It will cover the availability of information and support, the development workflow as well as the ecosystem and the embedded experience for both programming languages.

Availability of information
The first step to become a developer is to learn the specific language. The C community offers numerous different tutorials, many of which offer high quality content, although no central location for providing an overview is available. Such an overview for the most important tutorials and reference manuals on Rust on the other hand can be found at `https://www.rust-lang.org/learn`.

Once the first learning steps are taken, developers will likely need to find further information in reference manuals. On C, the only official source is the C standard, which is not available for free. Nevertheless, many unofficial reference manuals exist (like for example [13]). Rust provides official and extensive documentation, including the Rust book as well as the Rust embedded book for teaching the basic concepts, reference manuals for looking up the details and high quality code documentation. These code comments often not only document specific functions, but provide overview pages explaining the overall structure of a library and providing examples. There is even a book on how good documentation can be written [126]. A different aspect is to find documentation on industry-related topics and proprietary hard- and software. With C being the main language for many industries, professional support is available for almost all use cases. In Rust, although having a very helpful community, they often might not know about proprietary hard- and software. While for instance assistance for C libraries is almost always provided by hardware vendors, official Rust support is rarely available. Generally one can say that documentation for Rust is by far more visible and centrally promoted than it is the case for C, although proprietary vendors are still mostly focusing on C support and documentation.

Community support
When problems occur, most developers rely on community support. While for C there is no central platform, the community on the Q&A site StackOverflow is very active [98]. Rust however provides a central forum and a Github organization, where the language and its tools are developed. Rust's StackOverflow community is also quite active. An aspect that can be address in this context is the project governance. The focus of the C working group [14] lies only on the actual language and decisions are made by the committee's members [56]. The Rust project though manages different teams and working groups like for example the Compiler Team or the Embedded Devices Working Group [45] focusing not only on the language, but also its environment and ecosystem. Decisions about the language are handled through an RFC process which is completely open to the public [100].

Development workflow
While the availability of information and support are two important aspects on the usability, the actual development workflow also needs to be considered. As section 4.2.2 Tooling has shown, development tools providing a well-integrated programming experience have higher adoption in the Rust community with C developers relying on a larger set of different tools. Another example that influences the developers productivity is the project structure. In C, many different approaches for organizing a project exist. Function declarations are provided in header files, definitions in the source file. The developer must take care of including a header file only once and providing all source files to the compiler separately. In Rust, the project structure is standardized. All source files or modules are automatically provided to the compiler when they are included somewhere in the include hierarchy starting at the main file.

Another aspect that should be mentioned are compilation times. The strict Rust compiler leads to a focus on the develop/compile workflow which is largely affected by the long compilation times, whereas C offers very fast compilation speeds. This is partially caused by Rust performing optimizations over the whole crate, while C compilers mostly only optimize across a single source file. After the basic development of a project is done, bug fixing is often a time consuming activity. Because in C, almost every syntactically correct program compiles, most errors happen at run time. Most prominently, the segmentation fault (or a HardFault on embedded) caused by unsafe memory access, is well-known. Such issues need to be investigated by a debugger or other techniques, while one can generally assume that bug fixing with C needs the same amount of time as writing the program in the first place. Rust on the other hand provides its very strict compiler and borrow checker. Although developing the program in the first place and satisfying these tools is often called "fighting the borrow checker", a program that passes these checks oftentimes just works. In addition to that, Rust offers helpful run time crashes called panics including error messages with the originating source file and line (e.g. when accessing an array out of bounds). These differences are summarized by [19]. While using the C programming language improves the time until the first version can be shipped, Rust repays its higher early development efforts in the later stages of development. Because shipped Rust code is way more proven at compile-time, while C often causes unexpected issues like crashes when used by the customer. One can conclude that prototyping is faster in C because not all cases must be considered. Rust however is better suited for reaching production quality fast.

Ecosystem
Another important factor for the ease of use is the ecosystem of accessible libraries. C offers a large number of libraries for almost every possible use case. Nevertheless it is often difficult to find these libraries in the first place as there's no central platform or package manager with wide adoption. Libraries are also often published without proper documentation or guidance on how to integrate and build them. Contrastingly, Rust offers a highly integrated ecosystem with most packages being automatically deployed to `http://crates.io`. This package library additionally provides metadata like the popularity and dependent crates. It is generally accepted that the development of a library also includes documentation and examples, which can be seen by reading through the documentation of libraries like for instance actix [138] or serde [109].

Embedded experience To move the focus to embedded development, this paragraph will discuss the user friendliness when working with a microcontroller. The example in chapter 3 has shown that the Rust tooling makes it easier as only very few configuration options are needed and the write/compile/run cycle works almost out-of-the-box even for embedded.

Tweaking the GCC compiler until the program compiles on the other hand took quite a long time. The Makefile needed for C consists of almost 100 lines, while the Rust configs only add up to about ten lines of code. Certain tasks that need to be manually performed with the GCC toolchain like managing include paths or linking the correct files and libraries together were all handled by Cargo in the Rust example. This picture looks different when examining more industry-related hardware. Not many support crates (e.g. MACs or PACs) for proprietary hardware are available. Although writing board support libraries by hand is possible, it may be too much of an effort for peripheral access crates. Many vendors though offer so-called SVD files containing a machine-readable description of the memory layout and available peripherals. The Rust tool svd2rust can be used to generate the corresponding PACs from these files.

Conclusion
Concluding this chapter, Rust generally provides better information and support, but looses to C when considering proprietary applications. A Rust developer is highly supported by the ecosystem and development workflow, producing high quality results. Nevertheless may Rusts longer compilation and development complexity make C better suited for prototyping. Considering embedded development, Rust also provides a solid foundation for support and libraries, but the ecosystem is by far not on the same level as C, especially with industry-related technologies. Generally, Rust is better suited for productivity and easier to use, but still lacking in the proprietary or industry sector.

4.3.2 Maturity

The maturity of a programming language and its tools can be an important factor when deciding on the languages' suitability. Using a tool with a high volatility can be difficult to use in certain products, especially for safety-critical applications. This chapter will examine the rate of change of the C and Rust languages, compilers as well as their ecosystems.

Language Standard
The first aspect to be considered is the stability of the standard. The C standard specification is only changing very rarely with seven years in between the two most recent releases C11 and C18 [12]. This specification is developed independently from the compiler, requiring them to implement the changes strictly according to the standard. Rust on the other hand has no language standard. Although language changes can possibly happen on every compiler release, Rust provides strong stability guarantees. Breaking changes are only introduced into the language with the release of a so-called edition. These editions are released every three years, including the 2015 edition which marked the 1.0 release, the 2018 edition and the upcoming 2021 edition. [118]

Compiler
Another factor is the maturity of the compilers. While for C, the GCC and Clang compilers only have one and two major releases per year respectively [40][73], the Rust compiler team publishes a release every six weeks summing up to around nine major releases per year [105]. Nevertheless is backwards compatibility an important feature of Rust. All code, using stable functionality, since version 1.0 is guaranteed to work. Even mixing dependencies written for different editions is easily possible [118].

Ecosystem
Lastly, the stability of the ecosystem should be considered. As described in section 4.3.1 Ease of Use & Productivity, C has a long history including a very stable ecosystem, especially in the industry. Rust however is quite new, but already has the majority of important

libraries available. Many libraries still have version numbers below 1.0 implying that they have not yet matured. Rust's ecosystem is also not well-developed in the industry, as currently most Rust libraries are only created as hobby projects.

Conclusion
It can be concluded that C, with many decades of development history, offers a high level of stability in the language standard as well as in available tooling and libraries. The high volatility of the Rust ecosystem and its language changes could make it more immature, although Rust provides stability guarantees through the regularly released editions and backwards compatibility for all previous editions.

4.3.3 Popularity

After analyzing many aspects related to the language itself, the developers' perspective should be taken into account. A language cannot be used in practice if there are no developers available on the market or willing to learn the language. To be able to assess the general popularity, exemplary projects and programming language rankings will be examined. For evaluation of professional development, the popularity among companies is also considered. In addition to that, the adoption of embedded programming is assessed.

General Popularity
The general popularity of C is well-known. It is used widely among many different communities and industries. Although many modern applications use the C++ derivative instead of C, it is still very similar and should also be taken in account when examining the popularity. Applications include operating systems like Linux or Windows, browsers like Firefox and Chrome, language interpreters like CPython and much more. Rust's popularity is generally not so clear. It is still mostly used in open source projects and seems to show no widespread use in larger applications, although its popularity is rising. Examples include the Wire app using Rust for its protocol implementation [95] or Discord relying on Rust's high performance for some of their critical services [52].

According to the Stack Overflow developer survey 2020 [1] Rust is the most loved programming language of people developing in the language and wanting to continue to do so with 86%. It is followed by TypeScript with 67%, while C only resides on 21st place with only 33%. Among the most wanted languages by people currently not developing in the language Rust is positioned on 5th place with 15% and C on 16th place with only 5% of the respondents. However when looking at the actually used languages, Rust is placed on 19th place with only 5% of the respondents developing in this languages, while C resides on 11th place with 22%. Looking at previous years of the survey [29][28], Rust could improve all its scores, while C stayed the same or even became more unpopular. Another interesting statistic is the *PopularitY of Programming Language* (PYPL) ranking from January 2021 [17], which ranks programming languages according to how often a programming tutorial is searched on Google. Their table shows C/C++ in 5th rank with a 6.33% share and Rust only on 16th place with a 1.06% share. Considering the past, C/C++ popularity is almost stable, while Rust's popularity is rising since the release year 2015. RedMonk [88] computes its rankings by considering the languages' popularity on GitHub as well as on StackOverflow. Their ranking denotes Rust as having entered the top 20 in the year 2020 for the first time. They position C on 10th place, closely reflecting the StackOverflow results.

Professional Adoption
Another aspect to examine is the adoption among companies. While C is a popular language, it is often used only for very specific use cases, where a higher level language is

not suitable, e.g. where precise control over memory or very high performance is required. When these requirements are not needed, C++ or other languages are used regularly. Rust on the other hand finds adoption in many different fields. A case study by NPM [30] has identified Rust as a very suitable language for their use in performance-critical applications. Other real-world examples using Rust include: Amazon for their MicroVM supervisor Firecracker [47], Mozilla for parts of their browser Firefox [23], Atlassian, Cloudflare, Dropbox and many more [94].

Embedded Adoption
The languages' popularity in embedded development needs to be separated into the adoption by open source developers and by professionals. In a professional environment C is the de-facto standard for bare-metal embedded development and still common in other embedded applications, although C++ is often used instead as it is equally suited [57]. Among hobbyists the C language is very common through the Arduino platform. Rust also has a very active embedded community, which becomes clear when looking at the "awesome embedded rust" library list [9] or the Weekly Driver Initiative [5]. In professional embedded development, Rust has not gained popularity yet. Although commercial interest is rising, which can be seen by for example WindRiver including Rust support into their real-time operating system VxWorks [141], no examples of actual productive usage could be found.

Conclusion
In conclusion, this chapter has shown that although Rust is extremely popular among those already working with it, the languages market share is still very low. C however has a more stable and higher market share including a larger adoption in commercial products. Rust has gained traction in the open source community, but is entering industry products more slowly.

4.3.4 Standards & Certification

Many industries, especially those developing critical systems like the aerospace or energy sector, need to verify their development efforts for their customers. For this reason, standards containing guidelines and best practices were developed that a company can and sometimes must adhere to. The development process needs to be documented according to those standards and can then be checked for conformity. To prove the standards conformity, a certification can be performed. Since the programming language plays a vital role in development, it must be part of the certification process including its tools.

IEC 61508
This chapter will exemplarily analyze the C and Rust programming languages according to the IEC 61508 [53] standard. This standard deals with functional safety for safety-related systems and defines four *Safety Integrity Levelss* (SILs) SIL1 to SIL4 with SIL1 being the lowest and SIL4 the highest level. They each define a set of criteria which need to be fulfilled in order to be compliant to the respective SIL. Part seven (IEC 61508-7 [54]) of the standard lists measures that should be taken by a developer to reduce safety risks. The following requirements were deemed relevant for this thesis (each denoted with their respective section number from the standard):

C.4.1 A strongly typed programming language should be used.

C.4.3 Only certified tools and compilers should be used.

C.4.4 C.4.3 can be weakened: Only *proven* tools and compilers should be used, which means according to C.2.10.1, that the following aspects must apply:

1. The specification should be stable.
2. The tool is used in different applications.
3. The tool has at least one year on record without safety relevant failures.
4. All relevant problems are documented.

C.4.5 A suitable programming language should be used. The most important criteria are:

1. A block structure
2. Useful compiler checks
3. Run-time checks (especially for types and array bounds)
4. Support for self-contained software modules
5. The possibility to define validity ranges for variables
6. Language structures for error containment
7. Supported by a suitable compiler, libraries, a debugger and other relevant tools
8. Predictable execution

To prove standards conformity, validation suites can be employed. To certify a compiler, such a suite can prove the compliance with the programming language's standard specification [83]. Other test suites can check for functional safety and certify the compiler or the programs themselves for specific requirements. Even when the tests report problems or deviations, the certification can be considered complete. All the discovered problems need to be fully documented and considered when working with the certified software (e.g. not using the problematic compiler features).

The following paragraphs will now apply the above-mentioned criteria from the IEC 61508-7 standard to the C and Rust programming languages.

C

C.4.1 C is only weakly typed (see section 4.4.1 Safety)

C.4.3 Validation suites and multiple certified compilers like for instance the VxWorks Cert Edition (SIL3) [137] are available. Paper [83] shows amongst other things how a compiler can be certified for SIL4.

C.4.4 There are some very common and proven compilers (e.g. GCC, Clang, MSVC) (see section 4.3.2 Maturity)

1. The specification is very stable. Even though the compilers differ slightly, they each have their own stable standards (see part about portability in section 4.2.3 Compilation Targets).
2. They are widely used in many open source and industry applications.
3. Known issues are always fixed quickly.
4. Problems are documented for example in the GCC docs [66] or in the LLVM bug tracker [72] for Clang.

C.4.5 IEC 61508-7 table C.1 denotes the complete C language as not suited for SIL3 and SIL4. To be well-suited for all SILs, only a subset of C should be used. As shown in C.2.6.2, no recursion (C.2.6.7), no implicit type casting and no use of pointers (C.2.6.6) is allowed, coding guidelines like the MISRA C [81] or SEI CERT C Coding standards [20] should be applied and static code analysis tools should be employed. These are the explicitly checked criteria from the list above:

1. C is a block structured language.
2. Some compile-time checks (e.g. warnings on uninitialized values) are offered, but many problematic programs are allowed (e.g. missing checks for memory safety, see section 4.4.1 Safety).
3. There are almost no run-time checks provided by the language itself (no type or bounds checking)
4. C provides modularity, even though the configuration must happen through an external build system and is not part of the actual language (see section 4.2.2 Tooling).
5. For range definitions, only certain machine-driven steps like 16, 32 and 64 bit integer variables are available.
6. The language also doesn't support error handling by itself.
7. C doesn't provide a standard set of tools, but certain solutions are well-integrated (e.g. GCC compiler and linker + GDB debugger + make build system).
8. The execution is predictable if the external influences are predictable.

To solve the deficiencies, a safer language like Rust can be employed. Nevertheless, this language needs to be checked according to the same criteria.

Rust

C.4.1 Rust is a strongly typed language.

C.4.3 The single existing Rust compiler is not certified.

C.4.4 The official Rust compiler is not yet as well proven as many C compilers:

1. There is no clear language standard yet. Stability is provided through regularly released editions, but apart from them the language is still rapidly changing (see section 4.3.2 Maturity).
2. It is used in many open source projects, but not really proven in the industry.
3. Safety relevant problems are quickly fixed.
4. Problems are documented in Rust's issue tracker [103].

C.4.5 Table C.1 of the IEC 61508-7 standard does not mention Rust explicitly due to the language being quite new, but notes that other than the listed languages are allowed as long as they are assessed according to the same set of criteria which follows below:

1. Rust is a block structured language.
2. Numerous compile-time checks for memory and concurrent safety are provided exceeding the C compiler checks by far.
3. There are also some run-time checks included: Array bounds are always verified. Types are not checked at run-time because they are strongly proven at compile-time.
4. Rust provides a module system that is well-integrated into the compiler and package manager.
5. For range definitions, only certain machine-driven steps like 16, 32 and 64 bit integer variables are available.
6. Rust provides some language supported error handling. Verifying results that could fail before accessing a value must happen explicitly and is checked by the compiler.

7. A large set of well-integrated tools exists (see section 4.2.2 Tooling).
8. The execution is predictable, if external factors are predictable.

Conclusion
After comparing the two languages according to the given criteria, it becomes clear that they are both only partially suited for use in safety critical applications according to IEC 61508. C is highly verified and proven in the real world while Rust does not offer a certified compiler and is still changing rapidly. On the contrary Rust fulfills many of the technical criteria very well, while for C only a smaller subset should be used for standards conformity. This also means that Rust can possibly comply to the standard once the compiler is proven or certified, while the C language can never be compliant without restrictions.

4.4 Safety vs Performance

This section will examine how safety is handled in C and Rust and investigate how well both languages perform. Between both chapters, an example program will be introduced that will be used to carry out the measurements.

4.4.1 Safety

One of the most important aspects and the main reason why the Rust programming language could be a better alternative to C is the improved memory safety. Unsafe memory management often leads to undefined behaviour of the application, which can additionally affect one or more security goals.

This chapter introduces different types of safety from a programming perspective and explains the problems that currently exist in the C language which can lead to unsafe behaviour of the application. Afterwards, some real world implications and an overview of current countermeasures are shown. The chapter finishes by looking at how Rust handles these issues and which problems the new language is able to solve.

Safety basics
Before diving into the types of safety, the term *undefined behaviour* needs to be explained. Undefined behaviour means that in a certain situation, the compiler can freely decide what to do because the situation is not valid according to the language standard. Dereferencing a null pointer in C is an example of undefined behaviour. This means that a compiler is allowed to assume that every pointer that is dereferenced is indeed valid and can apply optimizations accordingly. [68]

The safety domain can be divided into different categories. The lack of a specific type of safety can also lead to an unfulfilled security goal. For each category, the implications for security are mentioned in the description. As most categories affect all security goals according to the CIA triad (Confidentiality, Integrity, Availability) the goals will not always be mentioned explicitly in the following list. When only a specific goal is affected, it will be denoted in the description. The categories are as follows:

- Type safety:
  Type safety prevents type errors like type confusion: Reinterpreting a memory location as a different type (e.g. through pointer casting) can lead to wrong semantics. When casting a small pointer type to a larger one and reading its referenced value, uninitialized memory or different variables will be accessed.

- Memory safety (see [116] and [51]):
  Issues regarding the access of invalid memory locations. While they could lead to a crash of the application, they can often impair security by leaking data or enabling further attacks.
  - Spatial safety: Memory issues caused by accessing memory at improper locations.
    * Buffer overflow or underflow
      This is realized by accessing buffer indices that are larger than the size of the buffer (overflow) or smaller than zero (underflow). This can lead to arbitrary memory reads and writes on the stack, heap or data segment of the program.
    * Null pointer dereferencing
      In addition to be seen as undefined behaviour, dereferencing a null pointer can have different implications. Running as a user space program in an operating system,

it will lead to a segmentation fault and thus to a program crash. This is of course not the intended behaviour, but only affects the availability of the application. Running in kernel space or bare-metal embedded environments, access to the null address is indeed possible and could allow an attacker to read or store data at this location.

  - Temporal safety: Memory issues caused by accessing memory at an invalid time.
    * Uninitialized use
      In this case, a variable is accessed before it is initialized. Because memory is not cleared (e.g. set to zero) before introducing variables most of the time, foreign or invalid data can be accessed.
    * Dangling pointer to stack
      This issue is caused by returning a pointer to a stack variable from a function. Because the stack frame is no longer valid after returning from the function and will even be overwritten by new function calls, the memory address points again to foreign or invalid data, which can thus be accessed.
    * Use-after-free
      Using a pointer to a previously allocated heap section after it is freed causes the same problems as the dangling pointer to stack issue, just with heap sections instead. A common special case of using such a pointer is freeing the memory again (called double-free), which can possibly corrupt the memory manager leading to unexpected behaviour in totally different parts of the program.

- Memory leakage:
  In this case, allocated heap space is not freed, leading to an increased memory usage. This issue again only affects the availability by provoking a Denial of Service.

- Safe concurrency:
  Running concurrent tasks for example through multiple operating system threads or interrupt handlers on embedded systems can cause a different set of problems.
  - Race conditions:
    Multiple tasks accessing the same data at the same time without any synchronization measures can lead to unintended results (e.g. skipping security logic). If the first task for example only executes code if a variable is non-null and the second task regularly replaces this variable, a situation can occur where the check for non-null was successful, then the other task sets the variable to null and only then the code of the first task is executed, now leading to a null pointer dereference.
  - Deadlocks:
    A program can block itself from executing if two tasks prevent their advance mutually. If for example task 1 waits for lock A having exclusively acquired lock B and task 2 waits for lock B having exclusively acquired lock A, no more progress is possible. This is once again only a problem for availability by provoking a Denial of Service.

- Side-effects:
  This generally means that the program state can be modified through other means than the program code. One example are volatile addresses. If the memory at an address can change from outside of the current execution context (e.g. through memory-mapped registers on embedded, which are changed due to hardware conditions), the compiler must consider these effects when optimizing the code. Otherwise, the values may be reused instead of re-read, thus leading to incorrect behaviour.

Memory Safety in C

C offers a high degree of freedom for the developer, often allowing the programmer to write

code that leads to one of the above-mentioned issues. The first aspect is the weak **type safety** of C. A reinterpretation of memory is easily possible through casting as seen below:

```
typedef struct {
  int passenger;
} Bike;

typedef struct {
  int passengers[5];
} Car;

void main() {
  Bike bike;
  bike.passenger = 5;
  Car* car = (Car*) &bike;
  int *p = car->passengers;
  int pass = p[4]; // Will access foreign/uninitialized memory
}
```

Listing 17: Accessing a pointer of the wrong data type (C)

Spatial memory safety is not always guaranteed. Array indices for instance are not checked. Consider an array x with the length of five. `x[5]` accesses an element outside of the array's memory (**buffer overflow**). A **buffer underflow** (`x[-1]`) is also easily possible. C's pointer arithmetic, which can be very useful in many cases, often makes such issues hard to find. Considering **null pointer dereferences**, a null pointer is often used as a magic value for many different interpretations (e.g. an error occurred, a variable is not yet initialized, there was no value, etc.). Since these interpretations are not enforced by the compiler, they need to be clearly documented by the developers or otherwise lead to unexpected consequences like undefined behaviour. Example:

```
const char* getData() {
  if (global_success) {
      return &global_data;
  } else {
      // The first developer uses NULL to signal an issue.
      return NULL;
  }
}

void process() {
  const char* data = getData();
  // A different developer assumes that data will never be NULL.
  char first = *data; // May access memory at address 0.
}
```

Listing 18: Potential NULL pointer dereference (C)

Temporal memory safety is another large challenge of the C language. **Uninitialized use** of memory is not strictly checked, though many compilers will produce a warning.

Example:

```
int color;
Color colors[2];
switch (input) {
  case COLOR_BLUE:
    color = 0;
    break;
  case COLOR_RED:
    color = 1;
    break;
}

// When input is neither BLUE nor RED, access to unknown memory
Color colorRgb = colors[color];
```

Listing 19: Potential uninitialized use of a variable (C)

The following example shows a **dangling pointer to the stack**:

```
int* getState() {
  int state = 5;
  return &state;
}

void doStuff() {
  char data[10];
  //...
}

void main() {
  int* state = getState();
  doStuff();
  if (*state == 1) {
    //...
  }
}
```

Listing 20: Dangling pointer to stack (C)

Here, `getState()` returns an address to memory inside its stack frame. Calling `doStuff()` will now most likely overwrite this memory with its own stack frame because of the much larger data variable. Accessing the now overwritten memory may lead to unexpected results. Even though issues considering the heap are not relevant for low-end or safety critical embedded programming, they may become increasingly important when memory sizes get larger and are thus still considered here. Since the heap space needs to be managed manually using `malloc()` to allocate and `free()` to free memory, **use-after-free** issues are possible. The pointer used to free the memory will still be valid from a compiler perspective afterwards and can thus be used to access invalid memory. Sharing pointers to the same value may lead to a **double-free** issue when those pointers are freed at different positions in code, which may be hard to debug. Manually managing memory can also lead to **memory leakage** when new memory is allocated regularly, but not freed if no longer needed.

When running multiple threads or interrupt handlers, direct write access to all static variables is possible with no synchronization required, possibly leading to **race conditions**. In addition to that, mutable references to stack or heap variables can be shared and therefore used simultaneously, too. There is also no simple way to prevent **deadlocks** apart from carefully placing synchronization code.
Side effects are not always considered, since the compiler can reorder memory accesses. This can be solved by marking the required variables with the `volatile` keyword (e.g. `volatile int* a`), which will treat all operations on these variables as volatile and thus perform all memory accesses exactly as described in the program. This is of course error-prone because the choice of which variables to mark is highly dependent on the environment of the application.

Real World Implications
The following paragraphs will discuss the real world implications of the mentioned memory issues and present countermeasures which are currently used. According to Microsoft [79] and the Chromium team [77] about 70% of discovered vulnerabilities in recent years are caused by unsafe handling of memory. Their research has shown that this rate stayed the same over the past 20 years despite the efforts taken to reduce the issues. The major problems include heap corruption, use-after-free, type confusion, uninitialized use and others. Due to the severity of issues caused by unintended use of the null pointer, Tony Hoare, the "creator" of NULL called its invention once "the billion dollar mistake" (cf. [50, at 27:40]), hinting at the damage such simple mistakes can cause. In addition to that, the MITRE corporation lists at least five memory safety vulnerabilities in their top 25 software weakness list (in addition to some others that can be caused by them) (cf. [82]).

Countermeasures
Due to the severity and the large consequences of mistakes with memory handling, many countermeasures have been developed. They can be separated into two different categories:

The first category tries to prevent issues during development and focuses on writing better software. Firstly, coding standards and guidelines can improve code quality and prevent common mistakes. Examples include the C secure coding guidelines [55], the MISRA C standard [81], the SEI CERT C Coding standard [20] and many more. Nevertheless the application of these standards is optional or not enforced in a project. An additional approach are static code analysis tools which scan the code for possible vulnerabilities and guideline conformity without executing it. Cppcheck [27] or SonarQube [110] are popular examples for the C language. As the study [37] concludes, none of the tested tools were perfect in detecting all crafted vulnerabilities, thus requiring additional approaches. Very similar to static code analysis is the so-called Fuzzing, which tests the program dynamically by inputting random or unexpected data with the goal to produce undesired results. As one can expect, this approach will not detect every issue because most of time it can't reproduce every possible situation. Testing the code at run-time is an approach also pursued by dynamic examination tools like Valgrind [133] running the program in some kind of virtual machine. They can be used for memory debugging, memory leak detection and more, but are not viable in embedded environments. Many of the safety issues could also be solved by external libraries or even the C++ standard library (e.g. smart pointers in C++11 handling unique ownership and other concepts), but since they are not enforced by the compiler, a misuse or deviation from best practices is always possible. (cf. [38, p. 12])

The second category, which is again not applicable for embedded environments, doesn't try to solve the actual memory problems, but to contain the damage during execution.

The first approach in this category includes different operating system tools (cf. [136]) like ASLR, which makes it harder to guess memory locations, or DEP, which prevents execution of user-introduced code. Analyzing at real-world vulnerabilities shows that bypassing these measures is still very common. The second approach is sandboxing. Applications are separated inside smaller units, making access to the outer world harder and thus preventing the attacker from gaining access to the whole system if one application is compromised. This includes technologies like virtualization, containerization or process sandboxing.

Looking at Microsoft's statistics [79], the same classes of vulnerabilities have prevailed over the past 20 years, meaning that even though the countermeasures may be useful, they don't prevent the issues in the first place. It may therefore be better to consider employing a safer language like Rust.

Memory Safety in Rust

The Rust programming language reduces the freedom available to the developer with the goal to still offer everything needed in a safe way [38]. The following paragraphs explain which memory safety problems introduced at the beginning of this chapter are solved by Rust and which are not.

Rust offers strong **type safety**. This means that for example type casts are only allowed on primitive types and always need to be expressed explicitly:

```
let a : u32 = 5;
let b : f32 = 7.5;
let c = a as f32 / b; // Explicit type cast
```

Listing 21: Explicit primitive type cast (Rust)

Other type conversions must be explicitly supported by the types through implementing the `From` or `Into` traits. In the following example, a custom `Number` type is introduced that can be created from an `i32` value:

```
struct Number {
    value: i32
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
      Number { value: item }
    }
}

fn main() {
    let val = 5;
    let num: Number = val.into();
}
```

Listing 22: Explicit type conversion (Rust)

Spatial memory safety is enforced by different mechanisms. Array access with indices is always checked according to the array's length making buffer overflows or underflows impossible [90, pp. 11-15]. Rust uses so-called fat pointers containing not only the address of the first element like in C, but also the size of the array. It must be noted that these checks may introduce a run-time overhead. To solve this, Rust provides faster alternatives for common programming patterns, for example Iterators for looping over an

array (see section 5.1.2 Portability). Furthermore, safe Rust has no pointers and thus no null references. Alternatives for most use cases exist: To represent optional or uninitialized values, the `Option<T>` type can be used. To return a value that may be an error, the `Result<T, E>` type should be used.

For **temporal memory safety**, uninitialized variables are not possible, variables always need to be initialized before they can be used. Rust also makes it easier to live with these restrictions due to its expression syntax:

```
let color = match input {
  ColorBlue => 1,
  ColorRed => 2,
  _ => 0 // Default value is always required
};
// .. color can be used here
```

Listing 23: No uninitialized use and utilizing the expression syntax (Rust)

Both the **dangling pointer to stack** as well as the **use-after-free** issues are impossible in safe Rust (cf. [21]) due to its ownership model which was explained in section 2.2 Rust. The basic approach can be described as Aliasing XOR Mutability, which means that either multiple read-only references to the same value exist or the value can be modified, but not both at the same time. In addition to that, the compiler always manages lifetimes of objects and borrows (references), dropping (freeing) the object implicitly after the owner goes out of scope. This is all guaranteed during compilation and therefore poses no run-time overhead. Most of the time, using the automatic dropping of variables is the preferred way and **memory leakage** is impossible. Though it is also safe in Rust to call the `mem::forget(data)` function which prevents cleaning up the referenced object when `data` goes out of scope (while still holding all safety guarantees). This can be useful to share data with libraries written in other programming languages. But it also means in this case, that memory leakage is possible in Rust.

The checking of references also applies to concurrent tasks. **Race conditions** are impossible and checked by the compiler (cf. [130]). Only immutable references can be shared over different execution contexts (threads, interrupt handlers, etc.). As explained in section 2.2 Rust, mutably shared memory is only possible with explicit synchronization. The following is an incorrect example, that would work fine in C, but is rejected by the compiler in Rust [90, p. 21]:

```
let mut data = vec![1, 2, 3];
let mutex = Mutex::new(data); // Mutex owns data
data.push(4); // Compilation error: data was moved
let mut d = mutex.lock().unwrap();
d.push(5); // Correct way
```

Listing 24: Synchronized data sharing with a Mutex (Rust)

Although unsafe use is impossible, the provided tools can still be used incorrectly to create **deadlocks**, which are not prevented by the language just like in C. Another advantage of the ownership system is that even when concurrency was not considered during development (e.g. in a used library), safety is still guaranteed.

Considering **side effects**, the behaviour is similar to C. But instead of whole data structures, only the required operations need to be marked as volatile, offering finer control: `ptr::read_volatile(x)`

All the mentioned Rust language semantics can make it hard for beginners as development may feel like working against the borrow checker. If the compilation succeeds though, the program is proven to be free of a large amount of memory or concurrency issues.

As one might notice, many actions especially in embedded development, would not be possible with the above-mentioned strict rules. This is the reason a wider set of the language called "unsafe Rust" exists. In this set, the code is not verified by the borrow checker and thus provides the ability to perform actions like directly accessing memory addresses or working with raw pointers (which is required for memory-mapped registers). Such parts of the code must be explicitly labeled with the unsafe keyword so they can easily be audited and reviewed. The programmer is expected to write memory safe code in these small sections, which can be easier in such separated spaces in comparison to a whole C program, but still poses a risk for memory safety. It is common practice to keep these sections as small as possible and abstract them through a higher, safe interface. The paper [97] finds that reasons for using unsafe Rust include the reuse of existing code (for instance with a Foreign Function Interface (FFI) to C), performance improvements (e.g. for unchecked array access) or bypassing too strict compiler checks. The paper also finds that unsafe code can only be replaced by safe Rust in some situations, but is indeed required to perform the required actions in most cases. On the other hand it might be alarming that the survey [36] finds that in June 2019, about 30% of all crates published on crates.io used some kind of unsafe code. There are also some real-world problems. The memory safety chapter of [97] found null pointer dereferences, buffer overflows and more in unsafe code. The Rustsec Advisory Database, which collects security issues of Rust crates, also contains numerous memory safety vulnerabilities inside unsafe Rust code [46]. This means that Rust is also not completely free from these issues, but they can significantly be decreased by reducing unsafe Rust code and carefully examining the clearly separated remains.

Conclusion
After summarizing different memory safety issues in C, looking at the real world and currently used countermeasures one can conclude that to solve memory safety problems, the issues need to be eliminated at compile-time. Seeing how Rust tries to respond to these problems shows that safer low-level programming is indeed possible, still careful examination is required when developing safety or security critical code.

4.4.2 Example program

To be able to assess certain aspects, a more elaborate programming example is needed. An existing implementation of a particle filter qualified as such an example. The purpose of this algorithm lies in sensor data fusion and it can be seen as an alternative to a Kalman filter. In the defence context, it is used to merge tracking data of multiple radar sensors. It is computationally more intensive than the Kalman filter, but allows to work with non-Gaussian probabilities. At the beginning, a defined number of particles, each with the same weight, is distributed in the coordinate space. After a measurement is received, the particle weights are adapted according to the distance to the measured point. To get rid of very inaccurate particles, the particles with the lowest weight are discarded and new ones are created close to the measurement in a resampling step. To predict the movement of the measured object, the particles are moved according to a proposal function. A predicted position can now be determined through the weighted distribution of the particles. To simplify the problem, a

pendulum was used as an application. The input of the filter is now the position of the object as detected by the sensor at each time step. In this simplified version, the movement of a pendulum is generated by a differential equation at the beginning of the program and the position at each simulation step is provided to the filter. The output would result in a prediction of the object's flight path, but to be able to compare the results clearly, a single number is printed in this simplified case. The products of the particle weight and their position are summed up to provide a deterministic and comparable output value. The C++ code of the program was already available and provided by a colleague. It was adapted to C and afterwards rewritten in Rust for the comparison.

The flow of the program looks as follows. The identifier given in parentheses is later used in the performance statistics:

1. The measurements are generated (`generate`).
2. The system is initialized, by placing the particles uniformly along the pendulums path (`initialize`).
3. A loop iterates over all measurements, simulating the different time steps (`filter`).
   a) For each measurement, the particles are updated (`sir_filter`).
      i. Each particle is moved according to the proposal function (`proposal`).
      ii. Each particle's weight is changed respectively to its distance to the measurement (`importance`).
   b) The particles are resampled, so that particles with low weight are replaced by particles closer to the measurement (`resample`).

After all measurements are processed, the output metric is generated and printed over the semihosting channel.

For both the Rust and the C program, two variants of data storage are used. Because of the immutability of static variables in Rust, it is common to place data structures on the stack, while in C static variables regularly utilize the data segment. To be able to compare this aspect, two programs for each of the languages are tested. One with most of the data on the stack (stack example) and one that uses the data segment (data example). In the data example for C, structures like `cl_particle particles[N];` exist as static variables. For giving elements to a function, indices are utilized. In the stack example, these data structures are mostly initialized in the main function, and pointers are passed to function calls for referencing a certain element. In the idiomatic Rust variant (stack example), the arrays are placed in structs which reside on the stack. E.g.

```
pub struct ParticleFilter { particles: [Particle; N] }
```

Here, elements are passed as references and indices are almost never used. Using static mutable variables for the data example is a greater challenge in Rust. Two conditions must be fulfilled for the compiler to prove safety: Firstly, the data must be synchronized over multiple execution contexts (e.g. main function and interrupt handlers). This is handled by a `Mutex` object. Since there is no operating system to provide synchronization utilities, this mutex can only be locked in an interrupt-free section. Such a section is introduced by the `interrupt::free` utility method. The second condition is the interior mutability. Since it is impossible for the compiler to prove that only a single mutable borrow of a static variable exists at any time, borrowing is performed at run-time. A `RefCell` object is employed for that. The definition of the variable now looks as follows:

```
static particles : Mutex<RefCell<[Particle; N]>> = //...
```

For accessing the data mutably, three steps need to be performed. Interrupts must be disabled, the `Mutex` must be locked and the `RefCell` value must be borrowed. A typical access now looks like this:

```
interrupt::free(|critical_section| {
  let p = particles.borrow(critical_section).borrow_mut()[0];
});
```

Listing 25: Correct mutable access to a static variable (Rust)

So in the end, four programs, two for each programming language are tested. All code files can be found in Appendix B in `c/perf` (p. xxxiii) and `rust/perf` (p. liii).

4.4.3 Performance

To evaluate how well Rust delivers on its promises on zero-cost abstractions, performance measurements were performed. The additional safety checks could impair the speed of the application or increase its footprint both of the executable and in memory at run-time. To be able to assess these implications, the particle filter example program was tested in both C and Rust according to defined aspects. These criteria include the time spent per method, the memory usage, the size of the executable and the compilation time. The following paragraphs will first explain how the measurements were performed, then present the numbers and finally discuss the results.

Measurements
To be able to measure the time spent per function, the processor cycles are counted. Many ARM controllers, including the one used in this thesis, contain a so-called *Data Watchpoint and Trace Unit* (DWT) [7][76]. This unit provides a 32bit *Cycle Count* (CYCCNT) register that can be read to return the number of processor cycles since startup. To enable cycle counting, the `CYCCNTENA` (CYCCNT Enable) bit in the DWT CTRL (Control) register has to be set [26]. By reading and storing the cycle count at the beginning of a function and subtracting it from the register's value at the end of the function, the cycles spent during method execution can be computed. This value is logged over the semihosting channel (see chapter 3 Programming a Microcontroller). To allow for nested measurements, a global variable stores the cycles spent during the timing calculations, which will be subtracted from all measurements. For C, the timing calculations are contained in `time.c`, while the Rust code is in `timing.rs`. The C code can be used as follows, which will print the cycles spent in between both function calls:

```
unsigned int tval = TimerStart();
// ... function body
TimerEnd(tval, "functionName");
```

Listing 26: Timing measurement invocation (C)

To be able to use the global mutable variable for storing the time spent during measurements in Rust, explicit synchronization would be required. Because this would influence the timings and direct access is indeed safe due to the variable being only accessed from one execution flow (not from any interrupt handlers), it is read and written directly in an `unsafe` block. The Rust code uses the `Drop` trait, which works like a destructor. As soon as the object goes out of scope, which is at the end of the current block, the `drop` method is called. This method will then calculate the passed time. It can be used like follows:

```
{
  let _t = timing.time("functionName");
  // ... function body
}
```

Listing 27: Timing measurement invocation (Rust)

The measurements are performed at the seven different spots of the program flow as defined in section 4.4.2 Example program.

For the memory usage, the maximum stack size is assessed. For measuring this metric, a technique called stack painting is employed. Before the program starts, the whole stack is filled with a predetermined pattern. After completion of the program, the stack is searched from the end of the stack segment. The first element that is not equal to the pattern corresponds to the maximum stack address. Then the size can easily be computed by subtracting this value from the address of the bottom of the stack (highest address) [75]. In both Rust and C, painting begins at the current stack pointer so that the already existing stack frames for the startup code are not overwritten. The stack size is reported through the semihosting channel at the end. The relevant code can be found in `stackcheck.c` for C and `stackcheck.rs` for Rust.

Another important factor for embedded development is the size of the executable which must be loaded into flash memory. This memory section contains code as well as static data and can be very limited on some microcontrollers. To measure this aspect, the section sizes are retrieved from the binary file using the `arm-none-eabi-objdump -h` command, resulting in a list of sections including their size, target address (VMA), address in the flash (LMA), file offset, alignment (Algn) and flags. For example the text section looks as follows:

```
Idx Name  Size      VMA       LMA       File off  Algn
  1 .text 00007c34  080001c0  080001c0  000101c0  2**6
          CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Listing 28: Object dump example

The sizes of the `text` (program code), `rodata` (read-only data), `data` (mutable data) and `bss` (zero-initialized data) sections are considered for comparison.

The last aspect is the time spent during compilation of the program. Although this does not influence the actual execution time, it is still important for software development and can possibly impair productivity. For measurement, the Linux `time` command was employed, resulting in an output containing the real time that has elapsed as well as the total execution time from multiple processor cores added together.

All these parameters are measured for different programs and settings. Firstly, the C and Rust programs are tested with different optimization levels, including no optimization (-O0), the best optimization for speed (-O3) and the optimization for size (-Os). Both variants, the stack and the data example, are tested for the two programming languages.

All measurements were performed separately. Firstly, the program was compiled without any measurement code in place and the compilation time was measured. This binary was also used to output the section sizes. After that, the program was compiled and run with the timing instrumentation in place. Afterwards, the stack painting methods were build into the binary and the program was run again. These steps can be found in the respective

`run_tests.sh` scripts in every programming language's folder. Since the per-function timing and stack data was received over the semihosting channel and written to a file, the results must be parsed. The `summarize.py` script adds up all the invocations of the same functions and calculates the minimum, maximum and average cycle counts for each of them. The source code for all the measurement utilities can be found in Appendix B in `c/perf` (p. xxxiii) and `rust/perf` (p. liii).

Results
Figure 2 (page xxiii) visualizes the number of total processor cycles spent during program execution. For both variants, with data structures in the data segment as well as on the stack, the measurement was executed for the three optimization levels 0, 3 and s. The results for the two variants are very similar, although differences can be seen between C and Rust. Table 5 (page xxii) provides an overview of the cycles spent per method. The rows correspond to the function hierarchy explained in section 4.4.2 Example program. Ten particles as well as ten measurements were used, resulting in ten executions for the `sir_filter` and `resample` functions and 100 executions for `importance` and `proposal`. For each function, the sum of all invocations, the average execution time and the minimum as well as the maximum execution times are provided. Functions with a "-" mark at the per-invocation fields were only executed a single time per program run. All variants were executed multiple times and turned out to be completely deterministic.

The resulting section sizes can be seen in figure 4 (page xxiv). The differences between the data and the stack variants were minimal. Since in both cases the data structures will only be initialized after starting the program, they don't occupy any flash space. The `data` section is also not drawn to this diagram, because it was almost empty for all test cases. To be able to compare the sizes of uninitialized data, which can be an indicator of memory usage, the `bss` section could be used. Although for C this section's size is filled with useful data, Rust doesn't seem to utilize it. This is most likely due to the fact that the Rust data structures are wrapped with an additional `Mutex` and `RefCell`, complicating the initialization, so that the zero-initialized `bss` section is not used. These are the reasons why only the `text` and `rodata` segments are drawn in this diagram, depicting the total flash memory usage. These numbers were again deterministic.

Figure 6 (page xxv) now shows the stack usage for different optimization levels and variants. In this case, large differences between the two data storage variants as well as the two languages can be observed. All measurements were again deterministic.

The last measurement in figure 8 (page xxvi), depicts the average compilation time in seconds for both programming languages and the three optimization levels. Those values were not deterministic, so this test had to be run multiple times and the average was calculated for each test. In addition to cleanly building the complete project, the compilation time was also measured when only a single file was changed.

Discussion
This section will try to explain the resulting statistics and investigate certain anomalies. Problems that are found will be fixed by optimizing the code, which will result in different measurements. The improved results will then be presented in the next chapter.

The first aspect to be discussed is the execution time measured in processor cycles as can be seen in figure 2 (page xxiii). C has very similar execution times on all optimization levels, but level `3` is still the best just like in Rust. For optimization level `0` however, Rust

has a much higher execution time, due to its additional abstractions. Stepping through the Rust program with a debugger reveals that all call structures remained intact without optimizations. Large call stacks were often caused by iterators or formatting functions. The optimized code however contained almost no function calls and iterators were shrunken down to only a few assembly instructions. In C, there were no such complicated functions in the first place as on one hand the formatting was implemented very simply and on the other hand basic `for`-loops were used for iterations. This only allowed the optimizer to improve the execution time of the C program by around 100.000 cycles (9%) from level `0` to `3`, while the Rust compiler reduced the number of cycles by around 450.000 (30%). The `s` optimization level showed almost the same number as `3` by only slightly increasing execution time. All in all, the cycle counts of Rust and C on optimization levels `3` and `s` are close to each other, although C being slightly ahead. Rust requires 104% of C's execution time for the same goal.

Table 5 (page xxii) takes a deeper look at the function timings. Firstly, it can be seen that the `generate` and `initialize` functions only make up small parts of the total program. While `generate` is faster in C, Rust is quicker when executing `initialize`. The `generate` improvement most likely depends on the compiler-provided implementations of the sine and cosine functions, as only mathematical formulas are executed. The reason why `initialize` is faster in Rust could be the use of iterators as they might allow the compiler to perform more targeted optimizations than when using basic `for`-loops. Examining the difference between stack- and data-located arrays, C shows almost no difference for `generate`. Rust however shows an improvement when the measurements are placed in the data section. Even though the additional synchronization measures were expected to impair performance, `generate`'s return data does not need to be copied as it would be the case for stack-located data.

The next function, `importance` only takes very few cycles for execution. Interestingly, the Rust function is much faster than the C variant, although both functions contain almost the same code. As investigations of the C code revealed, the `double` function `fabs` for getting the absolute value was used instead of the more suited `float` function `fabsf`. This caused type conversions that were responsible for the longer execution time. It was also found that floating point literals in C did not include the trailing `f` and thus were interpreted as `double` values. This caused additional overhead, since `double` calculations are not supported natively by the processor and had to be provided by the compiler. Comparing the stack and data variants only very small differences, probably relating to the controller's memory management, can be seen.

Further examinations showed that the `double` functions `sin` and `cos` used in other parts of the code were also falsely applied. Replacing them with their `float` counterparts improved the execution time of the C program drastically. Looking at the Rust assembly, although the correct `float` functions for sine and cosine were used, they internally relied on `double` calculations, leading to the same problems as with C. Since Rust does not include any math support in the core, but only in the standard library, the external crate `libm` was used. This crate does not provide any special implementations for specific targets like GCC does and thus relies on `f64` (`double` in Rust) implementations for sine and cosine that are very slow on `float`-only targets. It is already discussed in the Rust issue tracker to provide target-specific math functions in core, but was not yet implemented [4]. Other solutions to this problem would include using the `micromath` crate for inaccurate approximations of math functions or depending on a faster C library by linking it into the binary. Since internally, `f64` arithmetic was still used, the Rust program could not be optimized as

well as the C program as the final performance measurements, explained below, have shown.

A more complex part of the program is the `proposal` function. Although containing no conditions or loops, execution times differ largely between the minimum and maximum. The sine and cosine functions used could be the root cause for this behaviour, since they will most likely provide shortcuts for special input values, e.g. one or zero. While on average, C is faster, Rust shows faster min and max values. Rust being slower could again be caused by the implementations of the sine and cosine functions. Comparing stack and data variants does not show large differences. These functions are both called in the `sir_filter` function. The additional time taken by this function is smaller in Rust than it is in C, probably again caused by the use of iterators.

Finally looking at the `resample` function, Rust takes significantly less time for execution. This result is relatively unexpected due to the use of a range-style for loop. Rust needs to check the array bounds before accessing them multiple times, which would in theory cause an increase of execution time. The time spent by C was largely decreased by the `float` optimizations. As the investigations have shown, the overhead was caused by the `(1.0/N)` divisions, that required conversions from `float` to `double` and back.

Taking a look at the flash size of optimization level `s` in figure 4 (page xxiv), Rust binaries are much larger than those of C. This is not only the case with the `text`, but also the `rodata` segment. While C generates a `text` segment of around 25KB and a `rodata` segment of only around 500 bytes, Rust requires significantly more flash memory with around 40KB of `text` and 7KB of `rodata`. To improve these numbers, *Link Time Optimization* (LTO) (`lto = true` in `Cargo.toml`) was applied, which allows the linker to perform additional optimizations. This slightly improved the size, but only by about 3KB. Further investigation with `llvm-nm -print-size`, which prints all symbols (functions) and their sizes in the binary, revealed that the float formatting functions for outputting the result were taking up large amounts of space. In C, these were already implemented manually because the standard library did not offer them. Reimplementing them manually in Rust, reduced the flash size to around 22KB (18KB + 4KB). This also reduced stack sizes, which will be described in the following paragraph. Rust now seemed to be on the same level as C, but enabling LTO for GCC revealed also largely improved code sizes. The total size was shrunken to more than half (to around 10KB) of the previous value, keeping the distance between C and Rust.
Another investigation showed that the remaining large data section is caused by UTF-8 data used for Rust's string functionality. Since strings were only used for outputting the result over the semihosting channel and this is not part of the actual program, it was left out for further tests. To make sure that the program still runs correctly, the result was written to a predefined memory location that could be examined using GDB.

Before applying the above-mentioned optimizations, the stack usage looked like depicted in figure 6 (page xxv). Rust's maximum stack sizes were much higher than C's, especially in data example. The optimizations could largely improve on that, most likely caused by a large call stack of the float to string conversion methods.

The compilation times shown in figure 8 (page xxvi) display an expected behaviour. While C, especially the GCC compiler is known for its fast compilation times, a Rust program takes significantly more time to build. Although the GCC times differ between the optimization levels, they are always below three seconds. Rust on the other hand differs only slightly between the levels, but with a constantly high compilation time of around 40 seconds. Note

that this time applies to the whole project being compiled including libraries. When only a single file is changed, Rust doesn't recompile the dependencies and C even recompiles only the specific file. This results in the compilation times on the right hand side of the diagram. Rust's high compilation time are partially caused by the numerous libraries that are included. In the tested setup, 53 libraries were compiled by Rust, while most likely only a very small amount of code was actually part of the resulting binary. To be fair one must add, that while Rust considers all input files for compilation, in C only the necessary files were included in the Makefile and not the whole library. Rust's higher compilation time can also be explained by the additional checks introduced due to the borrow checker and the strong type system. Additionally, the very mature and grown optimizations of GCC may be hard to reach.

Results after Optimizations
After the above-mentioned optimizations were applied, new measurements were performed. All diagrams were drawn again to allow for comparison. As can be seen in figure 3 (page xxiii), the optimized C version is around ten times faster as before, while Rust could only improve its timings slightly. Execution times are hardly comparable now since only the C code was optimized and the Rust issues could not be solved easily. To be able to compare the languages in a fair way, the non-optimized graph in figure 2 (page xxiii) should still give a good view on the relative speeds of C and Rust, since both languages relied on the unoptimized `double` or `f64` implementations at this point.

The section sizes were largely decreased due to the removed functions (see figure 5) (page xxiv). Interestingly, this is only true for optimization levels `3` or `s`. The size of the unoptimized Rust version was even increased by around ten kilobytes. Rust's flash size was now reduced to about 12KB and 9KB for `3` and `s` respectively, while C used around 6KB for both variants. The larger Rust size is most likely caused by the remaining `f64` functions. The tables 3 and 4 (page xxi) show the largest functions and static variables from the output of `llvm-nm -print-size`. It can be seen that the sine and cosine functions as well as their utilities take up the largest space. Only looking at the particle filter algorithm, the `start` and `main` functions in C and the `__cortex_m_rt_main` function in Rust must be considered. It can be seen that the Rust function (1272 bytes) is only slightly bigger than the C functions (1116 bytes). Another interesting aspect is the `particles` array which takes up 200 bytes in C and 204 bytes Rust, most likely containing additional management data for the `Mutex` and `RefCell`. The additional functions for double calculations in Rust also take up much space, again making the results hardly comparable.

The stack size in figure 7 (page xxv) now gives a more diverse picture. While the `0` variant was only slightly decreased, the other optimizations levels show larger improvements. C's stack usage was reduced by 62% to around 100 bytes for the data variant, while Rust only reached 250 to 350 bytes through a 40% - 50% decrease. While for the stack variant C uses slightly less memory than Rust with `s` optimizations, it was just slightly reduced to 670KB by 70KB for level `3`. In this special case, Rust can even beat C by only requiring 650KB of stack space.

The compilation time was not measured again as it did not change in any meaningful way.

Other Performance Measurements
A quick look at other published performance measurements should be taken. Looking at [8] and [16], they both measured Rust's performance in comparison to C for a specific, non-embedded use case. Both blog posts conclude that when using the correct features of Rust

for the specific problems, the language can be on par with C performance or even outperform it. The blog post "Speed of Rust vs C" [67] provides a list of Rust features that are beneficial or obstructive for performance. Positive examples include highly optimizable iterator chains or strings with sizes known at run time, so that no `strlen` with a complexity of $O(n)$ is needed. Negative examples however are the forced synchronization of static data even for single-threaded applications, the additional bounds-checks for arrays or other run-time checks that may be needed when the strict compile-time borrowing rules cannot be satisfied.

Conclusion

This chapter has shown how measurements for spent time, used memory and flash as well as compilation time can be performed for C and Rust. It also presented and discussed the results of these measurements applied to the particle filter example program. Investigations revealed different problems with the implementations, but also the languages themselves. Although the issues found during the performance measurements are very specific, general conclusions can still be drawn. It becomes clear that C requires much more attention for tasks that are covered by Rust's type system. This was shown very well by the `float` / `f64` vs `double` / `f32` problem. In Rust, it would be impossible to call a `f64` function with `f32` values without explicit conversion due to its strong type system. Although, when investing time into the optimization of C code, the performance can be greatly improved and fine tuned. Rust on the other hand already shows pretty good performance with the first compiling program, but is harder to optimize for example caused by the use of libraries. Although the `f32` variants of the sine/cosine functions were used, they internally still relied on the `f64` type, making Rust's type system advantage meaningless. This shows that Rust's ecosystem is still lacking certain utilities, especially for embedded devices, although it is only a very specific example. In such environments, C is much more optimized and mature. In conclusion one can say that while the additional safety measures of Rust don't impair the language's performance and allow similar performance to C, other factors like the ecosystem can limit the ability for further optimization.

5 Switching to Rust

After the previous section has shown that Rust could be a viable alternative, this chapter will now look at important aspects to consider when actually switching programming languages and provide examples on how such a process could be approached. Firstly, three criteria will be examined before secondly, the results are summarized by providing examples from the industry and presenting an exemplary switching process.

5.1 Aspects

Three criteria, one non-technical and two technical, are considered. The first will investigate how a company could build up know-how for Rust and how employees can be trained. The other two aspects deal with the introduction of Rust into an existing code base. They will look at how legacy code can be ported to the new language as well as how new Rust code can interoperate with the established software projects.

5.1.1 Learning Rust

Before diving deeper into the technical aspects of switching to Rust, this chapter will look at how the goal of trained employees can be reached by examining three different approaches.

Firstly, already existing skills of the employees can be considered. Since Rust is already a quite popular language for open source projects, suitable programming skills could already be available in-house. Encouraging the use of a popular language at work may even boost the employees' motivation.

The second approach is of course trying to find Rust programmers on the market. Although it is a popular programming language, professional Rust developers are still quite rare (see section 4.3.3 Popularity). This may change in the future though, as according to the StackOverflow survey [1], the Rust developer is one of the best-paying programmer jobs available: A median salary of $74,000 a year puts Rust in fourth place globally. Even though this may be large investment, hiring Rust programmers can also be a plus for employer marketing in general, promoting the use of modern and popular programming languages.

An additional approach that will almost certainly be applied is to train the current employees. Feedback from the current development has to be considered. Everyone has favorite languages and technologies and may react differently to a change thereof. Rust also poses some challenges for learning and is known for its steep learning curve especially at the beginning. The developers need to get familiar with the ownership concept including lifetimes and borrowing and will have to get used to new tools and conventions. To perform the actual training, different methods are available. The employees can be given time to teach themselves, as everyone will have different learning speeds. As shown in section 4.3.1 Ease of Use & Productivity, elaborate official resources are available on the Internet. Examples include "The Rust Programming Language" [64], "Rust By Example" [104] or "Rustlings" [106], which provides small exercises for beginners. As the study [2] has found, examples are the most important resources when learning the Rust programming language, directly followed by Q&A sites. Employees can also be trained by their coworkers. If knowledge of Rust is already available in-house, the already qualified programmers can use free training concepts, like for instance the RustBridge workshop [70] to train their team members. Another but also generally more expensive method is to employ external training courses or coaches. Examples include online courses like the "Rust Fundamentals" on PluralSight [85], but also Rust consulting companies like Ferrous Systems [69], which offer training programs

for Rust beginners and even courses on Rust embedded development.

The best approach highly depends on the specific situation of the company, including the budget and the industry where it is situated. After qualified developers are now available, Rust's integration into actual software projects can be addressed.

5.1.2 Portability

Switching programming languages often includes rewriting existing code. This chapter will examine different aspects that have to be considered when porting C code to Rust. Firstly, some general considerations are explained and two possible approaches are presented. Afterwards, an exemplary procedure is denoted and some real-world examples are described.

When porting code, the results must be somehow verifiable. It should be tested that the new code still works as expected and produces the same results. This should happen through fine-grained unit tests on the one hand and higher level system tests on the other hand. Either way, a high test coverage must be ensured. Another aspect to consider before porting existing code is the availability of required libraries. Not everything might be available for the relatively recent Rust programming language and may need additional development efforts for the dependencies or the use of interoperability features (see section 5.1.3 Interoperability).

The rewrite of existing code should happen on a per-function basis. This allows to divide the whole task into smaller, easily approachable sub-tasks. Rewriting a function can either happen by starting completely new with the goal to reach the same functionality or by porting every line of code to Rust. In the first case, the function signature is copied and translated to fit Rust's syntax. Afterwards, the function is implemented anew according to the documentation or requirements. In the second case, the whole function is copied to Rust and modified so that it compiles by only changing the syntax. This might prove difficult when the previous C code contained unsafe approaches in regards to memory handling, which will be uncovered by Rust's additional safety checks (see section 4.4.1 Safety). At the beginning, the new code will most likely perform worse than the C code due to these checks, meaning that the Rust program will need to be refactored to employ more idiomatic Rust features (see examples below). Regardless of the approach used, tests need to be carried out to check if the results are correct or fixes need to be performed. Performance measurements can also play a vital role for optimizing the new code and reaching the same or even better execution speeds than before. [87]

The following list will now provide a exemplary procedure for moving C code to Rust:

- Signature: Move the return type to the end of the line (after an `->` operator). Invert the type definitions (e.g. `flag: bool` instead of `bool flag`), change basic types accordingly (e.g. `u32` and `i16` instead of `unsigned int` and `short`) and finally add the `fn` keyword at the beginning.
- Variables: Remove types from variable definitions where possible and add the `let` keyword instead. When variables need to be modified, also add the `mut` keyword for mutability.
- Remove parentheses from control structures: e.g. `if flag` instead of `if (flag)`
- Change for loops to use Rust's range syntax:
  e.g. `for i in 0..N` instead of `for (int i = 0; i < N; i++)`.

- Change the last return statement in a function from e.g. `return sum;` to an expression like `sum`.

The use of custom types may require additional steps to be taken, including the introduction of structs or enums. At this point, functions will most likely need the unsafe keyword because raw pointers might still be used. To make the code more idiomatic, the following guidelines should be considered:

- Change pointers to references and make sure that the borrow checker is satisfied. This might require increased efforts when for instance multiple mutable references to the same object exist, but will definitely be a benefit for the overall safety of the program.
- Introduce a more functional syntax like iterators and closures when working with arrays or other data structures (see examples below).
- Change function and variable names to match Rust's conventions (normally camel_case): e.g. `particle_filter` instead of `ParticleFilter`
- Further aspects and details can be found in the Rust Porting book [74].

The following example will compare how the iteration over an array can be rewritten in Rust. The following C code should be ported:

```
particle particles[N] = ...;

int weight_sum = 0;
for (int i = 0; i < N; i++) {
    weight_sum += particles[i].weight;
}
```

Listing 29: For loop over an array to be ported to Rust (C)

Directly rewriting the for loop in Rust would perform a size check on the array on each iteration. Thus it is better to use so-called Iterators, which only perform a single size check:

```
let particles : [Particle; N] = ...;

let mut weight_sum = 0;
for particle in particles.iter() {
    weight_sum += particle.weight;
}
```

Listing 30: Iterator over an array ported from C (Rust)

Using a more functional syntax with closures, this loop can be compacted even more, relying on Rust to take care of the actual iteration process:

```
let particles : [Particle; N] = ...;
let sum : u32 = particles.iter().map(|particle| particle.weight).sum();
```

Listing 31: Functional iterator over an array (Rust)

A different approach than rewriting everything manually is to use source code converters. The most common example is `http://c2rust.com` [15], although it only performs the first step of porting C code to Rust by producing a compilable, but unsafe and non-idiomatic version of the code.

Rewriting software in a new programming language can improve the code, reveal issues,

increase safety and performance, but can also easily introduce new bugs and take a lot of time. Instead of porting the software, code can also be reused, which will be explained in the following chapter.

5.1.3 Interoperability

Switching to a new programming language often raises the fear of having to rewrite entire code bases. An alternative is to reuse certain software modules written in C or other languages. While for example critical code like parsing user provided data should be implemented in a memory safe programming language like Rust, proven and unexposed algorithms can still be kept in an external C library. Rust provides the ability to interact with C code through a so-called *Foreign Function Interface* (FFI).

An FFI is a standardized way of calling functions from a different programming language. In the best case, calls and function signatures even look idiomatic in both languages. The interface needs to be realized through the compiler, by providing compatible Application Binary Interfaces (ABIs). These ABIs define calling conventions for function calls. Examples include the `cdecl` (for x86 C applications on Linux) or `fastcall` (for system calls on Linux) calling conventions. When using `cdecl`, function parameters are pushed to the stack in reverse order before the function is called. After returning, the stack needs to be cleaned up and the result can be read from the eax register. `Fastcall` on the other hand requires the parameters to be loaded into registers before the function is called. In the special case of Linux system calls on x86, `eax` contains the system call id and `ebx`, `ecx` and `edx` contain the parameters. To be inter-operable with C code, other programming languages' compilers need to adhere to these standards. It is very common for many languages to provide an FFI to the C language. Examples include Ada, where FFIs are called language bindings or C++ by using the `extern "C"` declaration. Even JIT-compiled languages like Python or Java (called *Java Native Interface* (JNI)), provide this feature.

Due to Rust using its own non-standardized ABI and thus calling convention, C function calls must be explicitly indicated. To be able to call C functions from Rust, they need to be declared in an `extern` block and include a semicolon after the signature. To inform the linker about the library to link to, the `#[link(name = "libname")]` attribute must be provided on the `extern` block. Calls to these functions must be wrapped in an unsafe block, as the compiler can make no safety guarantees on external code. The programmer has to make sure that the invoked C code is actually safe. To provide safe abstractions, these unsafe blocks are generally encapsulated. [128]
This includes code for converting inputs and outputs, including for example:

- Convert a Rust array into a pointer and length with `arr.ptr()` and `arr.len()`.
- Convert a Rust string into a null-terminated byte array.
- Convert a return value that could be NULL into a Result or an Option type.

To be able to call Rust functions from C code however, defined functions can be made ABI compatible using the `extern` specifier. Multiple ABIs are available [122]. Examples include `extern "C"` for the default C ABI of the current target as well as `extern "cdecl"` and `extern "fastcall"` for `cdecl` and `fastcall` calling conventions respectively. Normally, Rust functions are mangled, which means that the function name given in the program is different from the resulting name in the object file. So for example `_RNvNtCs1234_7mycrate3foo3bar` is the internal name for `mycrate::foo::bar` [142]. This can be turned off using the `#[no_mangle]`

attribute. In general, these FFI methods can be safe, but still require unsafe blocks for certain actions, e.g. when working with raw pointers that were received from C. For both languages to have a shared understanding of basic types, the Rust crate libc provides type definitions for C types containing for instance the `size_t` or `c_int` types. More complex data structures require more consideration, e.g. Rust `structs` are allowed to be freely reordered by the compiler. To force binary compatibility with C, they need to be annotated with `#[repr(C)]`.

To be able to use these interoperability tools in production code, some more aspects must be considered. As an interesting example, the Chromium team [101] has defined some features that are required for productive use of the FFI. The first issue they have identified is to reduce boilerplate code, since redefining all functions in Rust manually might require too much effort. This problem is not completely solved yet, but the definitions can at least be generated automatically using a tool called bindgen [10]. For example typing `bindgen include/api.h -o src/bindings.rs` automatically generates Rust bindings for the api.h header. Reducing the amount of unsafe keywords required for basic operations like passing data is another challenge identified by the Chromium team. This has sparked some discussion about whether or not unsafe blocks are required in all cases. From a Rust developer's point of view, these are mandatory because the safety of interfaces to C cannot be proven [59]. The Chromium developers noted that around 60% of their C and C++ code can be called from Rust with already available tools like the cxx library [129] and up to 92% are possible with some more tweaks.

Additional safety recommendations for FFIs can be found in [108], with the most important being:

- Type use must be consistent in both languages: E.g. don't mix integers of different sizes.
- Validity checks should happen on the Rust side for improved safety.
- FFI functions should use raw pointers instead of Rust references, as their validity cannot be guaranteed by the C side.
- A single language must be responsible for allocation as well as deallocation of data.
- Rust code called from the FFI must not panic, or else it would be undefined behaviour.

The Rust Omnibus page [44] contains many examples of programs making use of the Rust FFI from several different languages and can be a useful reference when dealing with interoperability.

An FFI allows high flexibility when writing software in different languages, making code reuse easily possible. Rust offers many tools to enable and simplify this process, although there are still some challenges present that may hinder productive use. This feature may also allow testing the suitability of Rust in large or complex applications with low effort by being able to include small parts of Rust code into the existing code base.

5.2 Example Process

This final chapter will exemplarly describe how switching to Rust can happen in a real world project. It will firstly discuss some successful examples from the software industry and finish by presenting an example process.

As a first example, Microsoft is looking for ways to improve memory safety in their products.

They classified their projects in three different categories and provide a recommendation for each case: [38, p. 21]

- New software: In this case a safer language like Rust should be employed. To start a project, only trained employees are needed, allowing this choice to be taken without too much of a risk.
- Software actively developed: A safer language should be used where possible, otherwise safe development practices must be applied. This requires some of the above-mentioned portability or interoperability approaches to be applied.
- Software in maintenance: Since switching to a safer language is too expensive, adopting safer practices must be enough in this case.

The most interesting and common case is the second one, where it needs to be decided what code is ported to Rust and which parts remain in the original language and only communicate with Rust through for example the FFI explained above. It must be evaluated, how much rewriting is advisable and when integrating code is the better solution [71].

A different approach that is also very common in the industry is starting out with small projects instead of rewriting entire systems in order to get familiar with the language. Examples include components of larger systems (e.g. the CSS Engine Stylo for Firefox, [23]) on one hand, but also separate systems that are completely implemented in Rust (e.g. Amazon's MicroVM host Firecracker [47]) on the other hand.

An example process for switching development to Rust could look as follows: Firstly, the current employees must be trained and/or professional Rust developers should be recruited (see section 5.1.1 Learning Rust). Then, to build up know-how, a complete project should be realized in Rust. To stay on the safe side, this can be a small, non-business-critical project. After analyzing the results, it should be decided if Rust can be incorporated in more important software projects or if the use of the Rust language will no longer be pursued. If the results are promising and a baseline of knowledge is available, Rust can be integrated into business-critical software. The first step is now to decide if components should be rewritten in Rust or if only new parts of the project should use this language. According to [87], a rewrite, which was described in section 5.1.2 Portability is most sensible for performance and safety reasons as well as lower maintenance costs, since Rust promises fewer runtime problems than other languages. Furthermore, new components should always be developed in Rust, especially if they are safety or security critical. All components written in Rust must now be integrated into the existing system. This can happen through the FFIs explained in section 5.1.3 Interoperability. At all times, it must be ensured that introducing Rust into the code base does not produce any regressions or performance losses by performing automated tests and comparing the Rust performance to the old program in the case of a rewrite.

Conclusion
In summary, the decision of switching to a new programming language is influenced by many factors and needs to be adapted to the individual business needs. The steps explained above can be a guideline on how to approach this process. Taking small steps can be a low-risk approach for introducing this new language into the company, like many examples have successfully shown.

6 Conclusion

To finish the thesis, a summary will be provided. Furthermore, aspects remaining for future research will be identified and motivated. Lastly, the objectives defined in the introduction are fulfilled by giving recommendations on the suitability of Rust for embedded software development.

Summary

To introduce the Rust and C programming languages, their history, programming concepts, user bases as well as their embedded functionality was explained. After that, a comparison of microcontroller programming followed. It showed that both languages are equally suited for the presented task. While the approach differed in the utilized tooling, libraries and code to modify registers, the general approach was the same, including the section configuration and startup logic. The required effort to read the input value of a button and toggle an LED was very similar.

The following comparison evaluated the two programming languages according to a previously defined and motivated set of criteria. On some aspects, the suitability of Rust was on par with C, while on others, it differed significantly. As it turned out, the popularity of both languages is very balanced. While Rust is gaining adoption and is more popular than C among programmers using the language, C is in use by a larger number of developers and industries. Another diverse topic is the performance. While execution speed and memory usage can be very similar to C, it was shown that with the reliance on the ecosystem Rust does not always allow the same optimizations easily. Rust code also generally results in larger binary sizes. Another positive aspect for C is the large amount of available targets. While Rust only supports a small number of architectures and standard libraries, it can generally be assumed that every hardware target available has a suitable C compiler. Due to C's long history, the language and its ecosystem are more mature and stable compared to Rust. While C offers a stable standard definition, Rust mostly depends on the implementations of a single compiler. However, there are also several aspects where Rust offers improvements upon C. The new language offers a greater variety of programming paradigms including object-oriented as well as functional programming. Additionally, Rust offers state-of-the-art tooling for many development tasks which increases productivity. While C has numerous tools available too, they are not in widespread use and often hard to integrate. Furthermore, very extensive documentation, community support and a sound ecosystem make Rust easy to use.
Coming to the main motivation to use Rust, memory safety guarantees at compile-time are offered by the language. C however requires the developer to take care of these potential issues. Besides safe memory handling, Rust code is also free from race conditions and offers strong type safety. A criterion that references many of the other aspects is the certification. The reviewed guidelines of the IEC 61508 standard showed that the Rust language is very well suited for safety critical applications from a technical perspective. C however should only be allowed with certain limitations in place. On the other hand it also showed that C's maturity is equally important for such applications, which may pose a problem for the relatively new and potentially immature Rust programming language.

The final chapter presented different aspects to be considered when eventually switching to Rust and provided an exemplary process. It was shown that numerous resources are available for employee training, but also that Rust programmers are not very common yet on the market. The language and its ecosystem also offer different tools for integration into an existing code base, including the Foreign Function Interface as well as portability

utilities. For a company to switch to the Rust programming language, it is advisable to take smaller steps by firstly introducing the language into non-business-critical projects.

Suggested Future Research

To further examine the suitability of Rust, further investigations can be carried out. An important field could be to continue with the performance improvements, for instance implementing different approaches for the sine and cosine functions and testing them. Further performance tests could also be carried out including peripheral interaction or interrupt handling. To get a clearer picture on the performance of certain aspects of the languages, smaller tests could be performed for looking at different safety measures separately. Measuring real-time behaviour can also be the goal of future work. Many of the experiences in this thesis were gathered by working with a bare-metal microcontroller. Since embedded development also includes more powerful devices with an operating system, such targets can also be an object of further examinations. Another goal for future research could be the focus on other programming languages like Ada for comparison to Rust.

Recommendations

Corresponding to the objectives defined in the introduction, the following recommendations can be given:

1. When choosing a programming language, one has to make a decision based on the above-mentioned criteria. It should become clear that while Rust can improve on many problems of C, it still comes with some drawbacks. Rust can thus be a suitable replacement for existing programming languages, although the decision should not be made without careful consideration of the examined criteria.

2. It was shown, that when switching to Rust, a holistic approach has to be taken to overcome obstacles from the beginning and make the transition a success. Employees must be trained, the new code must be integrated and a process tailored to the company's situation must be applied.

Outlook

The above-mentioned results and recommendations provide a foundation for informed decisions on the usage of Rust. In the MBDA Product Cyber Security team, the use of Rust is considered for safety critical components. According to this thesis' findings, the programming language will be further examined for a potential use in the company's products.

References

[1] *2020 Developer Survey*. StackOverflow. 2020. URL: https://insights.stackoverflow.com/survey/2020 (visited on 12/30/2020).

[2] Parastoo Abtahi and Griffin Dietz. "Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language". In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383069.

[3] Ankit Anubhav. *Masuta : Satori Creators' Second Botnet Weaponizes A New Router Exploit*. NewSky Security. Jan. 23, 2018. URL: https://blog.newskysecurity.com/masuta-satori-creators-second-botnet-weaponizes-a-new-router-exploit-2ddc51cc52a7 (visited on 01/05/2021).

[4] Jorge Aparicio. *Math support in core*. July 23, 2018. URL: https://github.com/rust-lang/rfcs/issues/2505 (visited on 11/15/2020).

[5] Jorge Aparicio. *The weekly driver initiative*. Jan. 18, 2018. URL: https://github.com/rust-embedded/wg/issues/39 (visited on 12/05/2020).

[6] Jorge Aparicio. *utest*. URL: https://github.com/japaric/utest (visited on 12/28/2020).

[7] *Arm Cortex-M4 Processor Technical Reference Manual Revision r0p1*. ARM. URL: https://developer.arm.com/documentation/100166/0001/Data-Watchpoint-and-Trace-Unit (visited on 11/15/2020).

[8] Reid Atcheson. *Rust And C++ On Floating-Point Intensive Code*. Oct. 19, 2019. URL: https://www.reidatcheson.com/hpc/architecture/performance/rust/c++/2019/10/19/measure-cache.html (visited on 01/06/2021).

[9] *Awesome Embedded Rust*. Rust Embedded Working Group. URL: https://github.com/rust-embedded/awesome-embedded-rust (visited on 12/05/2020).

[10] *bindgen*. Rust. URL: https://github.com/rust-lang/rust-bindgen (visited on 12/16/2020).

[11] Benjamin M. Brosgol. "A Comparison of Ada and Java as a Foundation Teaching Language". In: *Ada Lett.* XVIII.5 (Sept. 1998), pp. 12–38. ISSN: 1094-3641. DOI: 10.1145/291712.291752.

[12] *C - Project status and milestones*. JTC1/SC22/WG14. URL: http://www.open-std.org/jtc1/sc22/wg14/www/projects (visited on 12/20/2020).

[13] *C language*. cppreference.com. URL: https://en.cppreference.com/w/c/language (visited on 12/10/2020).

[14] *C working group*. JTC1/SC22/WG14. URL: http://www.open-std.org/jtc1/sc22/wg14/ (visited on 12/10/2020).

[15] *C2Rust Demonstration*. Galois and Immunant. URL: https://c2rust.com/ (visited on 12/12/2020).

[16] Bryan Cantrill. *The relative performance of C and Rust*. Sept. 28, 2018. URL: http://dtrace.org/blogs/bmc/2018/09/28/the-relative-performance-of-c-and-rust/ (visited on 12/05/2020).

[17] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. Jan. 2021. URL: https://pypl.github.io/PYPL.html (visited on 01/05/2021).

[18] Cosmin Cartas. “Rust - The Programming Language for Every Industry”. In: *Economy informatics* 19.1 (2019), pp. 45–51. ISSN: 1582-7941. DOI: 10.12948/ei2019.01.05.

[19] Nick Carter. *Moving from C to Rust*. Nov. 27, 2019. URL: https://www.flocknetworks.com/moving-from-c-to-rust/ (visited on 01/11/2021).

[20] CERT. *SEI CERT C Coding Standard. Rules for Developing Safe, Reliable, and Secure Systems*. Pittsburgh, 2018. URL: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[21] Yong Wen Chua. *Appreciating Rust's Memory Safety Guarantees*. Government Digital Services, Singapore. July 14, 2017. URL: https://blog.gds-gov.tech/appreciating-rust-memory-safety-438301fee097 (visited on 12/09/2020).

[22] D. C. S. Clair. “Ada: A new programming language: The Department of Defense developed an incredible new programming language and named it in honor of Ada Lovelace, the world's first programmer”. In: *IEEE Potentials* 4.3 (1985), pp. 26–29. DOI: 10.1109/MP.1985.6500263.

[23] Lin Clark. *Inside a super fast CSS engine: Quantum CSS (aka Stylo)*. Mozilla. Aug. 22, 2017. URL: https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/ (visited on 12/31/2020).

[24] *COMPCERT*. INRIA. URL: https://compcert.org/ (visited on 12/28/2020).

[25] *cortex-m*. Rust Embedded Working Group. URL: https://github.com/rust-embedded/cortex-m (visited on 12/17/2020).

[26] *Cortex-M3 Technical Reference Manual*. ARM. URL: https://developer.arm.com/documentation/ddi0337/e/system-debug/dwt/summary-and-description-of-the-dwt-registers?lang=en (visited on 11/15/2020).

[27] *Cppcheck*. Cppcheck team. URL: http://cppcheck.sourceforge.net/ (visited on 12/09/2020).

[28] *Developer Survey Results 2018*. StackOverflow. 2018. URL: https://insights.stackoverflow.com/survey/2018 (visited on 12/30/2020).

[29] *Developer Survey Results 2019*. StackOverflow. 2019. URL: https://insights.stackoverflow.com/survey/2019 (visited on 12/30/2020).

[30] The Rust Project Developers. “Rust Case Study: Community makes Rust an easy choice for npm”. In: (Feb. 25, 2019). URL: https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[31] *Discovery kit with STM32F303VC MCU*. UM1570. Rev. 6. STMicroelectronics. Aug. 2020. URL: https://www.st.com/resource/en/user_manual/dm00063382-discovery-kit-with-stm32f303vc-mcu-stmicroelectronics.pdf.

[32] Steve Donovan. *A Gentle Introduction To Rust - Object-Orientation in Rust*. URL: https://stevedonovan.github.io/rust-gentle-intro/object-orientation.html (visited on 12/22/2020).

[33] *Doxygen*. URL: https://www.doxygen.nl/index.html (visited on 12/28/2020).

[34] Emily. *Is Rust ready for embedded software?* Sept. 5, 2019. URL: https://www.bluefruit.co.uk/quality/is-rust-ready-for-embedded-software/ (visited on 12/17/2020).

[35] Matthias Endler. *Go vs Rust? Choose Go*. Sept. 15, 2017. URL: https://endler.dev/2017/go-vs-rust/ (visited on 12/27/2020).

[36] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. "Is Rust Used Safely by Software Developers?" In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 246–257. ISBN: 9781450371216. DOI: 10.1145/3377811.3380413.

[37] A. Fatima, S. Bibi, and R. Hanif. "Comparative study on static code analysis tools for C/C++". In: *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. 2018, pp. 465–469. DOI: 10.1109/IBCAST.2018.8312265.

[38] Sebastian Fernandez. *The Quest to Memory Safety*. Microsoft Security Response Center. Sept. 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_09_Ekoparty/EKO19_Quest_Memory_Safety_PL.pdf (visited on 12/09/2020).

[39] Alex Gaynor. *Introduction to Memory Unsafety for VPs of Engineering*. Aug. 12, 2019. URL: https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/ (visited on 09/21/2020).

[40] *GCC Releases*. GNU. URL: https://gcc.gnu.org/releases.html (visited on 12/20/2020).

[41] *GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers*. Alibaba Tech. Aug. 29, 2019. URL: https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378 (visited on 12/28/2020).

[42] *Go - Use Cases*. Google. URL: https://go.dev/solutions#use-cases (visited on 12/27/2020).

[43] *Googletest - Google Testing and Mocking Framework*. Google. URL: https://github.com/google/googletest (visited on 01/12/2021).

[44] Jake Goulding. *The Rust FFI Omnibus*. URL: http://jakegoulding.com/rust-ffi-omnibus/ (visited on 12/16/2020).

[45] *Governance*. Rust. URL: https://www.rust-lang.org/governance (visited on 12/10/2020).

[46] Rust Secure Code Working Group. *The Rust Security Advisory Database*. URL: https://rustsec.org/advisories/ (visited on 12/09/2020).

[47] Arun Gupta and Linda Lian. *Announcing the Firecracker Open Source Technology: Secure and Fast microVM for Serverless Computing*. Amazon. Nov. 27, 2018. URL: https://aws.amazon.com/de/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/ (visited on 12/31/2020).

[48] *Haskell*. Haskell.org. URL: https://www.haskell.org/ (visited on 12/27/2020).

[49] *History of C*. cppreference.com. URL: https://en.cppreference.com/w/c/language/history (visited on 12/27/2020).

[50] Sir Charles Antony Richard Hoare. *Null References: The Billion Dollar Mistake*. [Video]. Aug. 25, 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 12/09/2020).

[51] Diane Hosfelt. *Fearless Security: Memory Safety*. Mozilla. Jan. 23, 2019. URL: https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/ (visited on 12/09/2020).

[52] Jesse Howarth. *Why Discord is switching from Go to Rust*. Discord. Feb. 4, 2020. URL: https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f (visited on 12/30/2020).

[53] *IEC-61508. Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, 2010.

[54] *IEC-61508-7. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 7: Overview of techniques and measures*. International Electrotechnical Commission, 2010.

[55] *Information Technology — Programming languages, their environments and system software interfaces — C Secure Coding Rules*. JTC1/SC22/WG14, May 30, 2013. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1717.pdf.

[56] Javier Luis Cánovas Izquierdo and Jordi Cabot. "Analysis and Modeling of the Governance in General Programming Languages". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 179–183. ISBN: 9781450369817. DOI: 10.1145/3357766.3359533.

[57] j4x. *C vs C++ in embedded Linux*. Mar. 1, 2011. URL: https://stackoverflow.com/a/5158223 (visited on 01/12/2021).

[58] *Java*. Oracle. URL: https://go.java/ (visited on 01/12/2021).

[59] Russell Johnston. *The problem of safe FFI bindings in Rust*. Aug. 22, 2020. URL: https://www.abubalay.com/blog/2020/08/22/safe-bindings-in-rust (visited on 12/16/2020).

[60] Julien Jorge. *An overview of build systems (mostly for C++ projects)*. July 17, 2018. URL: https://medium.com/@julienjorge/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444 (visited on 12/28/2020).

[61] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Upper Saddle River, NJ, USA: Prentice Hall, 1978. ISBN: 9780131101630.

[62] Steve Klabnik. "The History of Rust". In: *Applicative 2016*. Applicative 2016. New York, NY, USA: Association for Computing Machinery, 2016. ISBN: 9781450344647. DOI: 10.1145/2959689.2960081.

[63] Steve Klabnik. *The Story of Rust*. URL: http://steveklabnik.github.io/history-of-rust/ (visited on 01/05/2021).

[64] Steve Klabnik and Carol Nichols. *The Rust programming language*. San Francisco, CA, USA: No Starch Press, 2018. ISBN: 978-1-59327-828-1.

[65] Aleksey Kladov. *Why is Rust the Most Loved Programming Language?* Feb. 14, 2020. URL: https://matklad.github.io/2020/02/14/why-rust-is-loved.html (visited on 01/05/2021).

[66] *Known Causes of Trouble with GCC*. URL: https://gcc.gnu.org/onlinedocs/gcc/Trouble.html (visited on 12/09/2020).

[67] Kornel. *Speed of Rust vs C*. URL: https://kornel.ski/rust-c-speed (visited on 01/06/2021).

[68] Chris Lattner. *What Every C Programmer Should Know About Undefined Behavior*. LLVM. May 13, 2011. URL: https://blog.llvm.org/posts/2011-05-13-what-every-c-programmer-should-know/ (visited on 12/09/2020).

[69] *Learn Rust from world-class trainers*. Ferrous Systems. URL: https://ferrous-systems.com/training/ (visited on 01/11/2021).

[70] *Learn with RustBridge*. URL: https://rustbridge.com/learn/ (visited on 01/11/2021).

[71] Ryan Levick and Sebastian Fernandez. *R-Evolution - A Story of Rust Adoption*. Microsoft Security Response Center. Nov. 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_11_RustFest/RustFestEU19_REvolution_Keynote.pdf (visited on 12/09/2020).

[72] *LLVM bug tracker*. URL: https://bugs.llvm.org/ (visited on 12/09/2020).

[73] *LLVM Download Page*. LLVM. URL: https://releases.llvm.org/ (visited on 12/20/2020).

[74] locka99. *A Guide to Porting C/C++ to Rust*. URL: https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/ (visited on 12/12/2020).

[75] Terry Louwers. *Stack Painting*. URL: https://github.com/tlouwers/embedded/tree/master/StackPainting (visited on 11/15/2020).

[76] *Measuring the cycle count of the Cortex-M3 and Cortex-M4 processor's own activity*. ARM. URL: https://developer.arm.com/documentation/ka001406/1-0 (visited on 11/15/2020).

[77] *Memory safety*. The Chromium project. URL: https://www.chromium.org/Home/chromium-security/memory-safety (visited on 12/09/2020).

[78] J. Meyerson. "The Go Programming Language". In: *IEEE Software* 31.5 (2014), pp. 104–104. DOI: 10.1109/MS.2014.127.

[79] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. Microsoft Security Response Center. Feb. 7, 2019. URL: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf (visited on 12/09/2020).

[80] André Miranda and João Pimentel. "On the Use of Package Managers by the C++ Open-Source Community". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, 2018, pp. 1483–1491. ISBN: 9781450351911. DOI: 10.1145/3167132.3167290.

[81] MISRA. *MISRA C:2012. Guidelines for the use of the C language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK, Mar. 2013.

[82] MITRE. *2020 CWE Top 25 Most Dangerous Software Weaknesses*. The MITRE Corporation. 2020. URL: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[83] O. Morgan. "Certified Testing of C Compilers for Embedded Systems". In: *2007 3rd Institution of Engineering and Technology Conference on Automotive Electronics*. 2007, pp. 1–5.

[84] *Multi-Domain Combat Cloud*. Airbus. URL: https://www.airbus.com/defence/Multi-Domain-Combat-Cloud.html (visited on 01/05/2021).

[85] Dmitri Nesteruk. *Rust Fundamentals*. May 27, 2016. URL: https://www.pluralsight.com/courses/rust-fundamentals (visited on 01/11/2021).

[86] Raphael Nestler and Noah Hüsser. *Embedded Rust*. June 18, 2020. URL: https://github.com/rust-zurichsee/meetups/blob/master/2020-06-18_embedded-update-probe.rs/slides.pdf (visited on 12/22/2020).

[87] Carol Nichols. *Rust out your C*. URL: https://github.com/carols10cents/rust-out-your-c-talk (visited on 01/11/2021).

[88] Stephen O'Grady. *The RedMonk Programming Language Rankings: June 2020*. RedMonk. July 27, 2020. URL: https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/ (visited on 12/30/2020).

[89] *OpenOCD User's Guide*. The OpenOCD Project. URL: http://openocd.org/doc/html/index.html (visited on 12/17/2020).

[90] Philipp Oppermann. *The Rust Way of OS Development*. May 30, 2018. URL: https://phil-opp.github.io/talk-konstanz-may-2018/ (visited on 12/09/2020).

[91] Serkan Özkan. *Linux : Vulnerability Statistics*. URL: https://www.cvedetails.com/vendor/33/Linux.html (visited on 01/10/2021).

[92] A. Pinho, L. Couto, and J. Oliveira. "Towards Rust for Critical Systems". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036.

[93] Dan Pittman. *C vs. Rust: Which to choose for programming hardware abstractions*. Jan. 17, 2020. URL: https://opensource.com/article/20/1/c-vs-rust-abstractions (visited on 01/05/2021).

[94] *Production users*. Rust. URL: https://www.rust-lang.org/production/users (visited on 12/31/2020).

[95] *Proteus*. Wire. URL: https://github.com/wireapp/proteus (visited on 12/30/2020).

[96] *Python*. Python Software Foundation. URL: https://www.python.org/about/ (visited on 01/12/2021).

[97] Boqin Qin et al. "Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 763–779. ISBN: 9781450376136. DOI: 10.1145/3385412.3386036.

[98] *Questions tagged [rust]*. Stackoverflow. URL: https://stackoverflow.com/questions/tagged/rust (visited on 01/12/2021).

[99] *Repository statistics*. Repology.org. URL: https://repology.org/repositories/statistics (visited on 11/26/2020).

[100] *RFCs*. Rust. URL: https://github.com/rust-lang/rfcs (visited on 12/10/2020).

[101] *Rust and C++ interoperability*. Chromium. URL: https://www.chromium.org/Home/chromium-security/memory-safety/rust-and-c-interoperability (visited on 12/15/2020).

[102] *Rust AVR*. The AVR-Rust project. URL: https://github.com/avr-rust (visited on 01/11/2021).

[103] *Rust bug tracker*. URL: https://github.com/rust-lang/rust/issues (visited on 12/09/2020).

[104] *Rust by Example*. Rust. URL: https://doc.rust-lang.org/stable/rust-by-example/ (visited on 01/11/2021).

[105] *Rust Releases*. Rust. URL: https://github.com/rust-lang/rust/blob/master/RELEASES.md (visited on 12/20/2020).

[106] *Rustlings*. Rust. URL: https://github.com/rust-lang/rustlings (visited on 01/11/2021).

[107] *rustup*. Rust. URL: https://rustup.rs/ (visited on 12/17/2020).

[108] *Secure Rust Guidelines - Foreign Function Interface*. Agence nationale de la sécurité des systèmes d'information. URL: https://anssi-fr.github.io/rust-guide/07_ffi.html (visited on 12/16/2020).

[109] *Serde*. Serde. URL: https://serde.rs/ (visited on 01/12/2021).

[110] *SonarQube*. SonarSource S.A. URL: https://www.sonarqube.org/ (visited on 12/09/2020).

[111] *Status of Supported Architectures from Maintainers' Point of View*. GNU. URL: https://gcc.gnu.org/backends.html (visited on 01/11/2021).

[112] *stm32-rs*. URL: https://github.com/stm32-rs/stm32-rs (visited on 12/17/2020).

[113] *STM32CubeF3*. STMicroelectronics. URL: https://github.com/STMicroelectronics/STM32CubeF3 (visited on 12/17/2020).

[114] *STM32F303xB/C/D/E, STM32F303x6/8, STM32F328x8, STM32F358xC, STM32F398xE advanced ARM®-based MCUs*. RM0316. Rev. 8. STMicroelectronics. Jan. 2017. URL: https://www.st.com/resource/en/reference_manual/dm00043574-stm32f303xb-c-d-e-stm32f303x6-8-stm32f328x8-stm32f358xc-stm32f398xe-advanced-arm-based-mcus-stmicroelectronics.pdf.

[115] *System View Description*. ARM. URL: https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html (visited on 12/28/2020).

[116] L. Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[117] The Rust Core Team. *Rust's 2018 roadmap*. Dec. 3, 2018. URL: https://blog.rust-lang.org/2018/03/12/roadmap.html (visited on 01/05/2021).

[118] *The Edition Guide - What are Editions?* Rust. URL: https://doc.rust-lang.org/edition-guide/editions/index.html (visited on 12/20/2020).

[119] *The Embedded Rust Book - Memory Mapped Registers*. Rust. URL: https://rust-embedded.github.io/book/start/registers.html (visited on 01/05/2021).

[120] *The Rust Programming Language - Data Types*. Rust. URL: https://doc.rust-lang.org/book/title-page.html (visited on 01/05/2021).

[121] *The Rust Programming Language - Object Oriented Programming Features of Rust*. Rust. URL: https://doc.rust-lang.org/book/ch17-00-oop.html (visited on 12/22/2020).

[122] *The Rust References - External blocks*. Rust. URL: https://doc.rust-lang.org/reference/items/external-blocks.html (visited on 12/15/2020).

[123] *The Rust References - Influences*. Rust. URL: https://doc.rust-lang.org/reference/influences.html (visited on 12/27/2020).

[124] *The rustc book - Command-line arguments*. Rust. URL: https://doc.rust-lang.org/rustc/command-line-arguments.html (visited on 01/05/2021).

[125] *The rustc book - Platform Support*. Rust. URL: https://doc.rust-lang.org/nightly/rustc/platform-support.html (visited on 12/17/2020).

[126] *The rustdoc book*. Rust. URL: https://doc.rust-lang.org/rustdoc (visited on 01/11/2021).

[127] *The rustdoc book - Documentation tests*. Rust. URL: https://doc.rust-lang.org/rustdoc/documentation-tests.html (visited on 12/28/2020).

[128] *The Rustonomicon - Foreign Function Interface*. Rust. URL: https://doc.rust-lang.org/nomicon/ffi.html (visited on 12/15/2020).

[129] David Tolnay. *CXX — safe FFI between Rust and C++*. URL: https://github.com/dtolnay/cxx (visited on 12/16/2020).

[130] Aaron Turon. *Fearless Concurrency with Rust*. Rust. Apr. 10, 2015. URL: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html (visited on 12/09/2020).

[131] *Unity*. ThrowTheSwitch. URL: http://www.throwtheswitch.org/unity (visited on 01/12/2021).

[132] T. Uzlu and E. Şaykol. "On utilizing rust programming language for Internet of Things". In: *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2017, pp. 93–96. DOI: 10.1109/CICN.2017.8319363.

[133] *Valgrind*. URL: https://valgrind.org/ (visited on 12/09/2020).

[134] Ivan Valkov, Natalia Chechina, and Phil Trinder. "Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, 2018, pp. 218–225. ISBN: 9781450351911. DOI: 10.1145/3167132.3167144.

[135] Peter Van Roy. "Programming Paradigms for Dummies: What Every Programmer Should Know". In: (Apr. 2012).

[136] V. van der Veen. et al. "Memory Errors: The Past, the Present, and the Future". In: *Research in Attacks, Intrusions, and Defenses. RAID 2012*. Ed. by Cova M. Balzarotti D. Stolfo S.J. Berlin, Heidelberg: Springer, 2012, pp. 86–106. DOI: 10.1007/978-3-642-33338-5_5.

[137] *VXWORKS CERT EDITION*. WindRiver. URL: https://www.windriver.com/products/vxworks/certification-profiles/#vxworks_cert (visited on 12/28/2020).

[138] *Welcome to Actix*. The Actix team. URL: https://actix.rs/docs/ (visited on 01/12/2021).

[139] *What exactly do companies use Haskell for?* Aug. 23, 2016. URL: https://www.reddit.com/r/haskell/comments/4z4svh/what_exactly_do_companies_use_haskell_for/ (visited on 12/27/2020).

[140] *What is a DDoS Botnet?* Cloudflare. URL: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-botnet/ (visited on 12/27/2020).

[141] *Wind River Redefines Embedded Software Development with New VxWorks Release*. WindRiver. Oct. 8, 2019. URL: https://www.windriver.com/news/press/pr.html?ID=22444 (visited on 12/05/2020).

[142] Michael Woerister. *RFC 2603 Rust Symbol Mangling*. Nov. 27, 2018. URL: https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html (visited on 01/12/2021).

Appendix A - Performance Measurements

The raw data of all measurements can be found in the attachment in the `performance` folder.

Name	Size [Byte]	F/V	Description
`__kernel_rem_pio2f`	1664	F	Required part of the sine and cosine functions
`two_over_pi`	792	V	Precomputed $\frac{2}{\Pi}$ value for use in sine and cosine functions
`start`	760	F	The whole particle filter algorithm (All function calls were inlined)
`__ieee754_rem_pio2f`	668	F	Wrapper around `__kernel_rem_pio2f`
`main`	356	F	The program's main function that calls `start`
`__kernel_cosf`	260	F	Internal cosine function
`scalbnf`	228	F	Used for sine and cosine. Multiplies a number with two to the power of another number
`particles`	200	V	Particles array to be placed in the `bss` section

Table 3: Largest functions (F) and static variables (V) in the C binary after optimizations (with -Os and static data)

Name (Demangled)	Size [Byte]	F/V	Description
`rem_pio2f`	2696	F	Required part of the sine and cosine functions
`__cortex_m_rt_main`	1272	F	The main function including the whole particle filter algorithm
`__adddf3`	976	F	Double addition for the sine/cosine functions
`__muldf3`	768	F	Double multiplication for the sine/-cosine functions
`sinf`	448	F	The sine function
`__truncdfsf2`	392	F	Double utility used for sine/cosine functions
(...)	(...)	(...)	(...)
`particles`	204	V	Particles array to be placed in the `bss` section

Table 4: Largest functions (F) and static variables (V) in the Rust binary after optimizations (with -Os and static data)

	C with Stack				Rust with Stack				C with Data				Rust with Data			
Function	Total	Avg	Min	Max	Total	Avg	Min	Max	Total	Avg	Min	Max	Total	Avg	Min	Max
generate	84343	-	-	-	89592	-	-	-	84344	-	-	-	89339	-	-	-
initialize	42716	-	-	-	41206	-	-	-	42675	-	-	-	41272	-	-	-
importance	14400	144	144	144	9000	90	90	90	14500	145	145	145	9200	92	92	92
proposal	822356	8224	1283	12272	885066	8851	455	9661	822856	8229	1288	12277	884666	8847	451	9657
sir_filter	844496	84450	64204	97014	898236	89824	84932	92074	844796	84480	64234	97044	897716	89772	84880	92022
resample	25942	2594	2544	2600	7081	708	688	713	25526	2553	2513	2557	7547	755	724	759
filter	871299	-	-	-	905645	-	-	-	871183	-	-	-	905631	-	-	-
Total	998358	-	-	-	1036443	-	-	-	998202	-	-	-	1036242	-	-	-

Table 5: Cycle counts in Rust and C with -O3

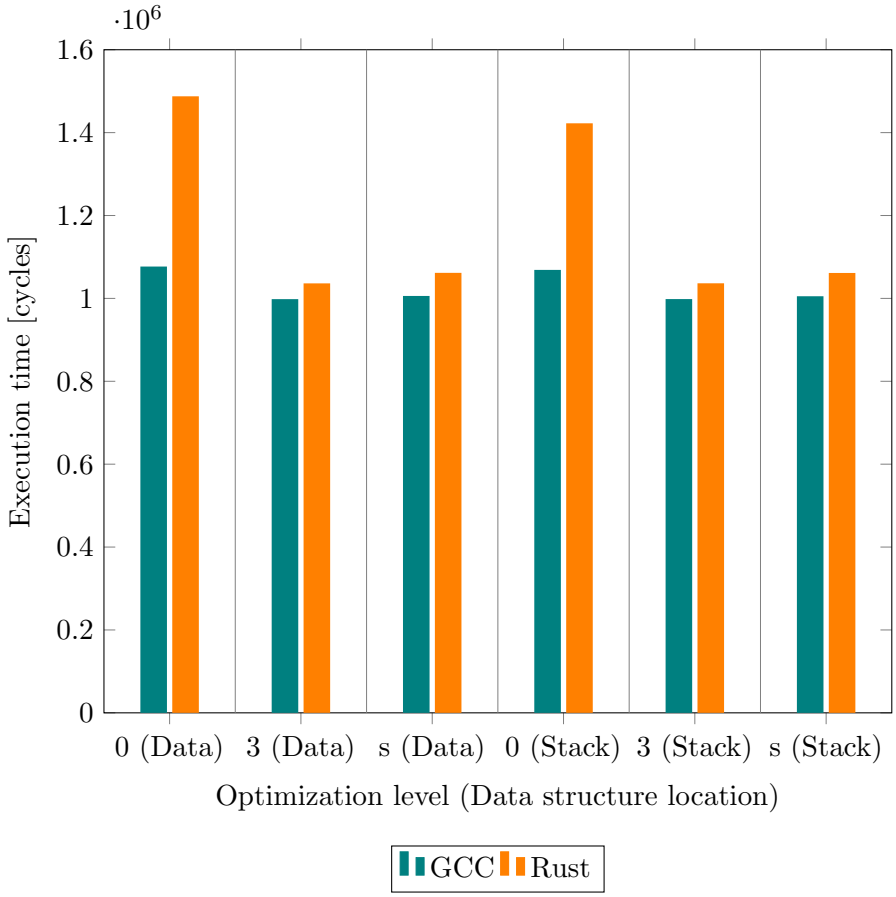

Figure 2: Rust (Stack) and C (Data) times

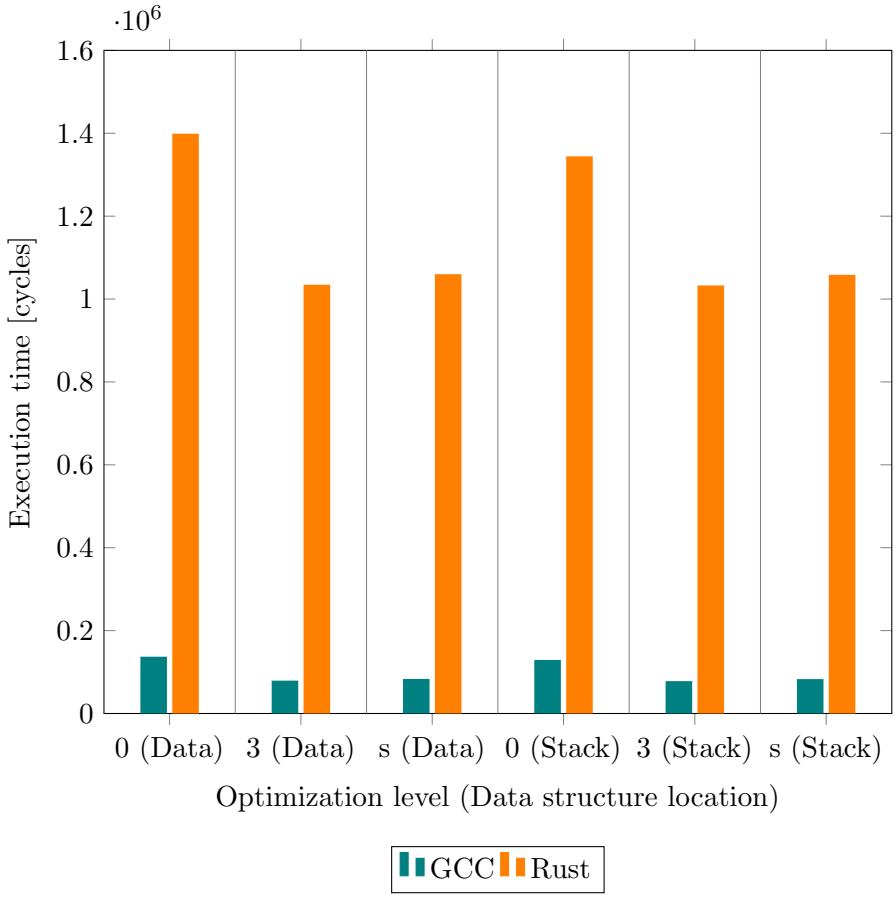

Figure 3: Rust (Stack) and C (Data) times with optimizations

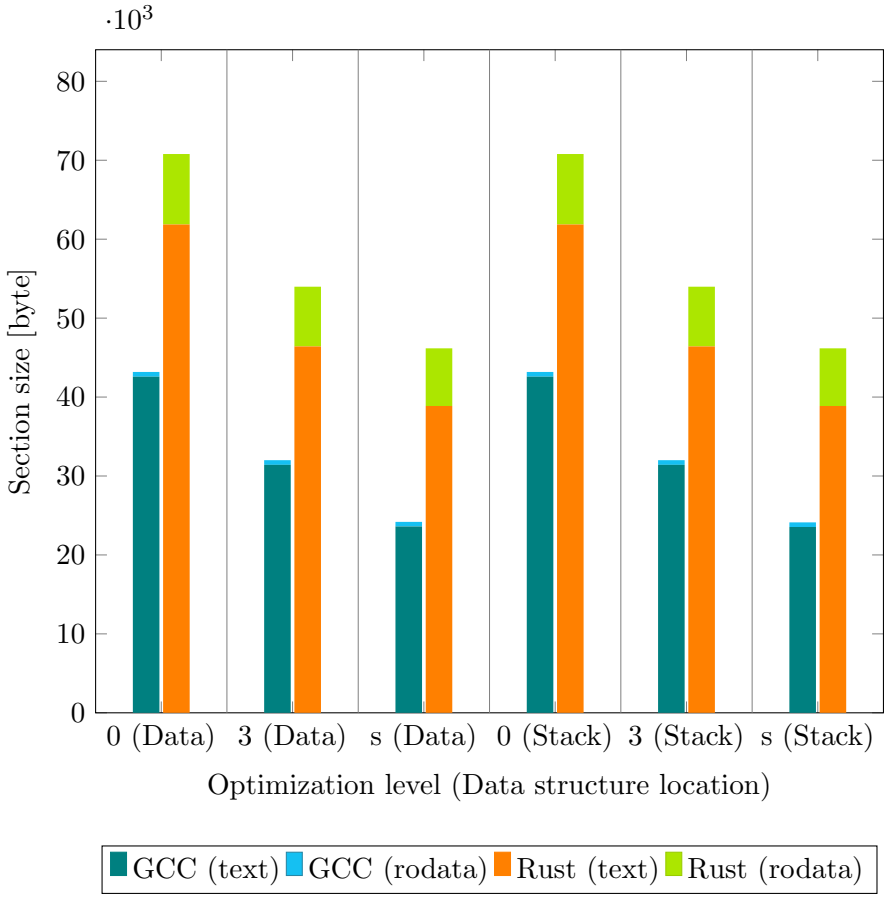

Figure 4: Rust and C flash size

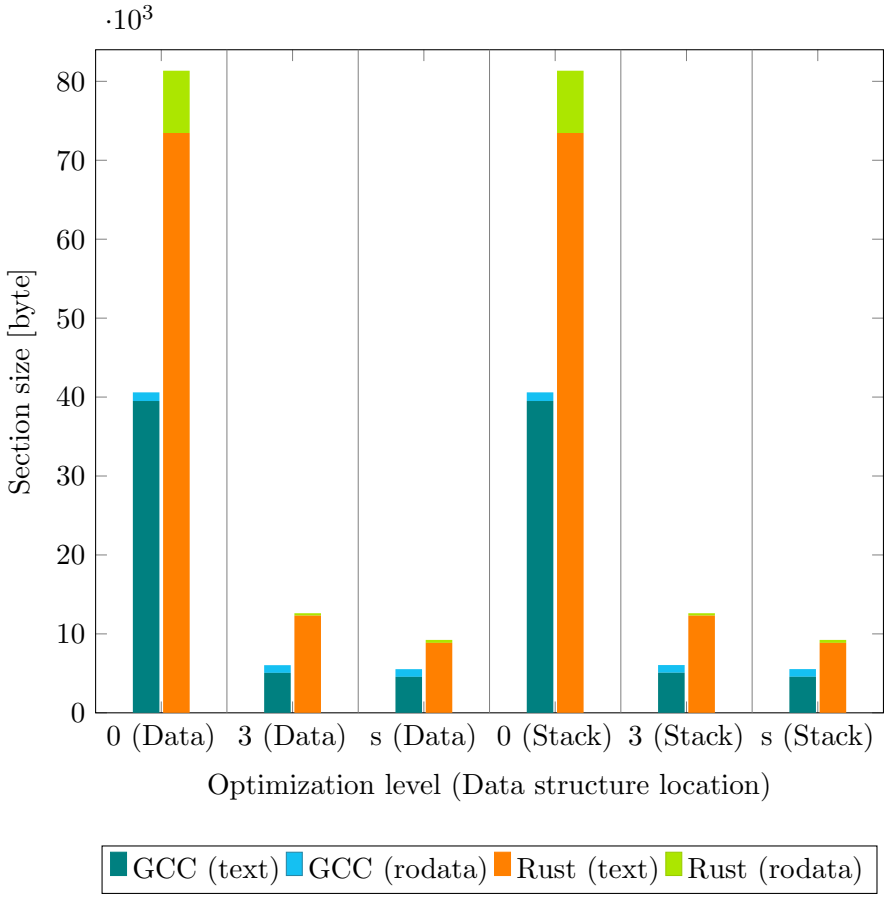

Figure 5: Rust and C flash size with optimizations

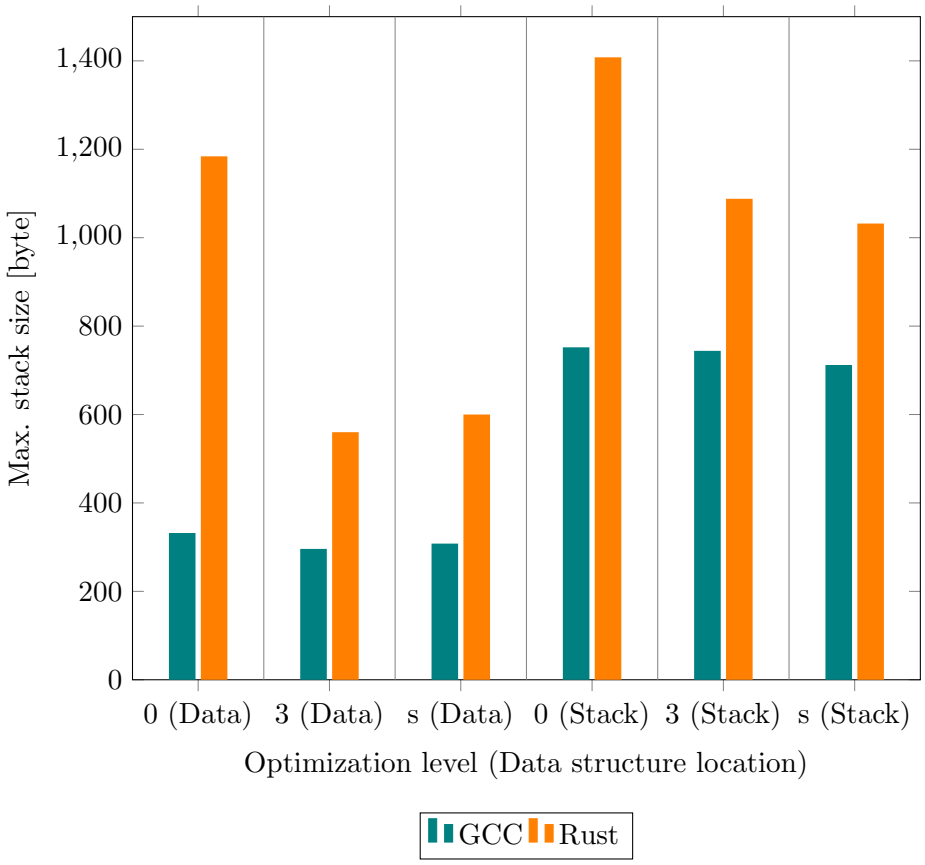

Figure 6: Rust and C memory usage

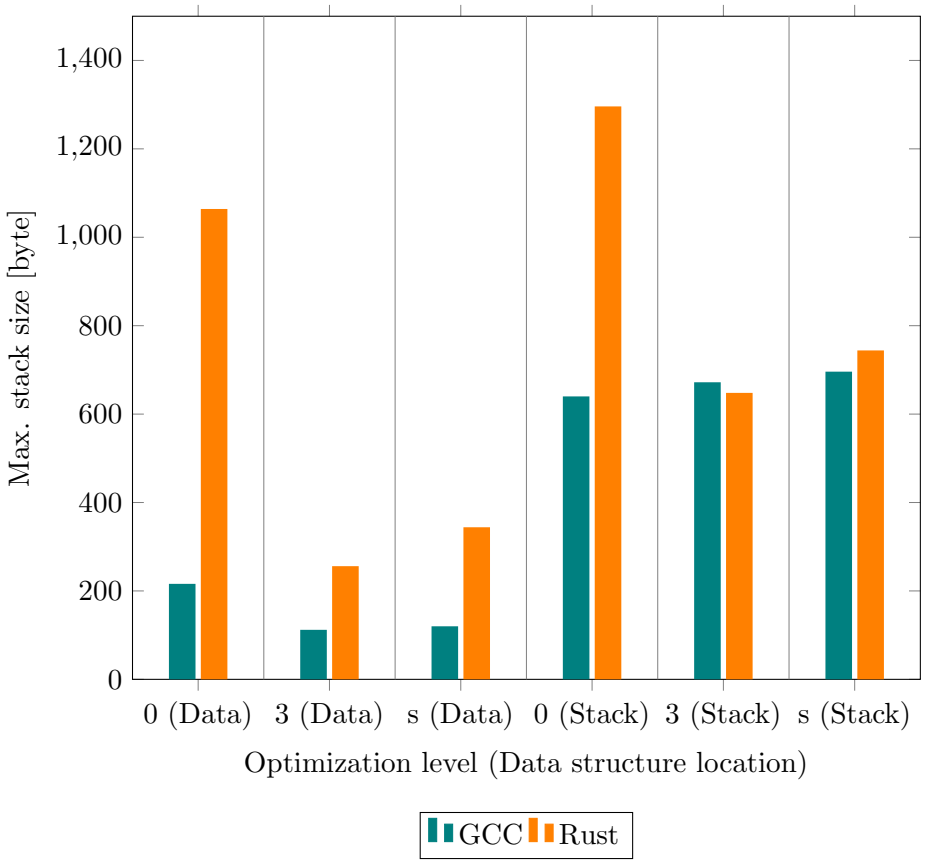

Figure 7: Rust and C memory usage with optimizations

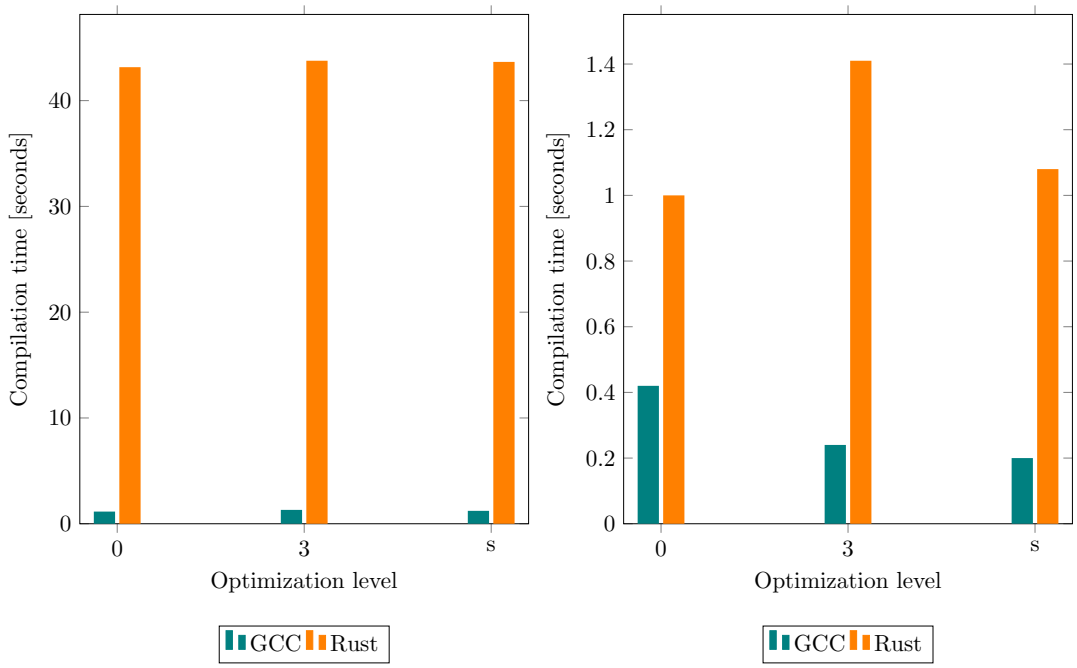

Figure 8: Rust and C compilation times (using Stack data structures)
Left: Clean build. Right: Only single file changed

Appendix B - Source Code

Overview

The source code of all files that were referenced in this thesis can be found on the following pages. All other relevant files (e.g. configuration files etc.) are only in the attached zip file in the `source` folder. The following list contains all files. Descriptions are only provided if not self-explanatory.

File path	Printed below?	Description
`/c/bare/`	-	The C bare metal example
`main.c`	No	
`Makefile`	Yes	
`openocd.gdb`	No	The GDB script for flashing
`run.sh`	No	
`startup.c`	Yes	
`stm32.ld`	Yes	Linker script
`/c/lib/`	-	The C library example
`main.c`	Yes	
`Makefile`	No	
`openocd.gdb`	No	
`run.sh`	No	
`startup_stm32f303xc.s`	No	Library provided startup code
`stm32.ld`	No	Library provided linker script
`stm32f3xx_hal_conf.h`	No	Library configuration file
`STM32CubeF3/`	No	STM32 library taken from `https://github.com/STMicroelectronics/STM32CubeF3`
`/c/perf/`	-	C Particle filter example
`filter.c`	Yes	Main file
`Makefile`	No	
`manual.gdb`	No	Script for manual flashing
`openocd.gdb`	No	Script for automated flashing
`openocd.sh`	Yes	Script to start openocd
`run.sh`	No	
`run_tests.sh`	Yes	Script to run the tests
`semi.c`	Yes	Semihosting implementation
`semi.h`	Yes	
`stackcheck.c`	Yes	Stack painting implementation
`stackcheck.h`	Yes	
`startup_stm32f303xc.`	No	
`stm32.ld`	No	Library provided linker script
`stm32f3xx_hal_conf.h`	No	Library configuration file
`summarize.py`	Yes	Script to summarize timings
`time.c`	Yes	Cycle count implementation
`time.h`	Yes	
`STM32CubeF3/`	No	STM32 library
`tracking/`	-	Tracking sub-folder

cl_particle.h	Yes	
cl_particle_filter.c	Yes	Particle filter stack example
cl_particle_filter.h	Yes	
cl_particle_filter_data.c	Yes	Particle filter data example
cl_particle_filter_data.h	Yes	
cl_pendulum.c	Yes	
cl_pendulum.h	Yes	
/c/tools/	-	JSON example on Tooling
CMakeLists.txt	Yes	
conanfile.txt	Yes	
toolDemo.c	Yes	
/rust/bare/	-	The Rust bare metal example
Cargo.toml	No	
openocd.gdb	No	The GDB script for flashing
.cargo/config	Yes	
rt/	-	Runtime sub-crate
build.rs	No	
Cargo.toml	No	
link.x	No	Linker script
.cargo/config	Yes	
src/lib.rs	Yes	Startup code
src/main.rs	No	Main file
/rust/crates/	-	The Rust example with libraries
Cargo.toml	No	
memory.x	No	The memory configuration
openocd.gdb	No	
.cargo/config	No	
src/main.rs	Yes	
/rust/perf/	-	Rust Particle filter example
Cargo.toml	No	
memory.x	No	
openocd.gdb	No	
openocd.sh	No	Script to start OpenOCD
run_tests.sh	Yes	Script to run the tests
summarize.py	No	Script to summarize timings
.cargo/config	No	
src/	-	Source folder
main.rs	Yes	
measurement.rs	Yes	Measurement struct
particle_filter.rs	Yes	Particle filter stack example
particle_filter_data.rs	Yes	Particle filter data example
pendulum.rs	Yes	Pendulum generator (stack example)
pendulum_data.rs	Yes	Pendulum generator (data example)
stackcheck.rs	Yes	Stack painting implementation
timing.rs	Yes	Cycle count implementation

`/rust/tools/`	-	JSON example on Tooling
`Cargo.toml`	Yes	
`family.json`	Yes	JSON input file
`src/main.rs`	Yes	

Table 6: Overview of the file structure

The sources

source/c/bare/Makefile

```
TARGET = main

# Set compilation sources
C_SRC = ./main.c ./startup.c

# Set linker script, opt level and target processor
OPT_LEVEL = 0
LD_SCRIPT = stm32.ld
MCU_SPEC = cortex-m4

# Configure toolchain
TOOLCHAIN = /usr
CC = $(TOOLCHAIN)/bin/arm-none-eabi-gcc
OC = $(TOOLCHAIN)/bin/arm-none-eabi-objcopy

# Set flags for compiler and linker
CFLAGS += -mcpu=$(MCU_SPEC)
CFLAGS += -mthumb
CFLAGS += -Wall
CFLAGS += -g
CFLAGS += -mfloat-abi=hard
CFLAGS += -O$(OPT_LEVEL)
CFLAGS += -ffreestanding

LSCRIPT = ./$(LD_SCRIPT)
LFLAGS += -mcpu=$(MCU_SPEC)
LFLAGS += -mthumb
LFLAGS += -Wall
LFLAGS += -specs=nosys.specs
LFLAGS += -nostdlib
LFLAGS += -T$(LSCRIPT)
LFLAGS += -mfloat-abi=hard
LFLAGS += -O$(OPT_LEVEL)

OBJS += $(C_SRC:.c=.o)

.PHONY: all
all: $(TARGET).bin

%.o: %.c
    $(CC) -c $(CFLAGS) $(INCLUDE) $< -o $@

$(TARGET).elf: $(OBJS)
```

```
    $(CC) $^ $(LFLAGS) -o $@

$(TARGET).bin: $(TARGET).elf
    $(OC) -S -O binary $< $@

.PHONY: clean
clean:
    rm -f $(OBJS)
    rm -f $(TARGET).elf
```

source/c/bare/startup.c

```
// Linker defined section variables
extern unsigned int _sidata;
extern unsigned int _sbss;
extern unsigned int _ebss;
extern unsigned int _sdata;
extern unsigned int _edata;

void startup();
void main();

// Put address of startup function (reset vector) into separate section
unsigned int * startup_vec __attribute__ ((section(".vector_table.reset_vector")))
    = (unsigned int *) startup;

void startup() {
    unsigned int * bss_start_p = &_sbss;
    unsigned int * bss_end_p = &_ebss;

    // Initialize BSS section to 0
    while (bss_start_p != bss_end_p) {
        *bss_start_p = 0;
        bss_start_p++;
    }

    unsigned int * data_rom_start_p = &_sidata;
    unsigned int * data_ram_start_p = &_sdata;
    unsigned int * data_ram_end_p = &_edata;

    // Copy DATA section from FLASH to RAM
    while (data_ram_start_p < data_ram_end_p) {
        *data_ram_start_p = *data_rom_start_p;
        data_ram_start_p++;
        data_rom_start_p++;
    }

    // Call main
    main();
}
```

source/c/bare/stm32.ld

```
/* Define memory regions */
MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  CCMRAM : ORIGIN = 0x10000000, LENGTH = 8K
  RAM : ORIGIN = 0x20000000, LENGTH = 40K
```

```
}

SECTIONS
{
    /* Vector table for stack address and reset vector at the beginning of FLASH */
    .vector_table ORIGIN(FLASH) :
    {
        /* First entry: initial Stack Pointer value */
        LONG(ORIGIN(CCMRAM) + LENGTH(CCMRAM));

        /* Second entry: reset vector */
        KEEP(*(.vector_table.reset_vector));
    } > FLASH

    /* text section for the program code in FLASH */
    .text :
    {
        *(.text .text.*);
    } > FLASH

    /* rodata section for read-only data in FLASH */
    .rodata :
    {
        *(.rodata .rodata.*);
    } > FLASH

    /* Beginning of data in FLASH to be copied into data section in RAM */
    _sidata = .;

    /* Zero-initialized data section to be situated in RAM */
    .bss :
    {
        /* Define start of bss for startup code */
        _sbss = .;
        *(.bss .bss.*);
        /* Define end of bss for startup code */
        _ebss = .;
    } > RAM

    /* Data section to be situated in RAM, copied from FLASH at _sidata */
    .data : AT(_sidata)
    {
        /* Define start of data for startup code */
        _sdata = .;
        *(.data .data.*);
        /* Define end of data for startup code */
        _edata = .;
    } > RAM

    /* Discard additional sections */
    /DISCARD/ :
    {
        *(.ARM.exidx .ARM.exidx.*);
    }
}
```

source/c/lib/main.c

```
#define STM32F303xC
```

```c
#include "stm32f3xx_hal.h"

static GPIO_InitTypeDef  GPIO_InitStruct;

/*
 * Main function, can either execute the PA or HAL main function.
 * Both functions implement a program listening for button presses.
 * When the button is pressed, an LED lights up, if it is released, it turns off.
 */
void main() {
  main_pa();
  //main_hal();
}

/*
 * This PA (Peripheral Access) main function
 * Directly accesses the register values by using preprocessor definitions
 */
void main_pa() {

  // Start peripheral clock for GPIOA and GPIOE ports
  MODIFY_REG(RCC->AHBENR, RCC_AHBENR_GPIOAEN_Msk, 1 << RCC_AHBENR_GPIOAEN_Pos);
  MODIFY_REG(RCC->AHBENR, RCC_AHBENR_GPIOEEN_Msk, 1 << RCC_AHBENR_GPIOEEN_Pos);

  // Configure button
  MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0_Msk, 0b00 << GPIO_MODER_MODER0_Pos);

  // Configure LED
  MODIFY_REG(GPIOE->MODER, GPIO_MODER_MODER15_Msk, 0b01 << GPIO_MODER_MODER15_Pos);

  // Loop forever
  while (1) {

    // Wait until button is pressed
    while ((GPIOA->IDR & GPIO_IDR_0) == 0b0) {};

    // Turn on LED by setting the set bit
    SET_BIT(GPIOE->BSRR, GPIO_BSRR_BS_15);

    // Wait until button is released
    while ((GPIOA->IDR & GPIO_IDR_0) == 0b1) {};

    // Turn off LED by setting the reset bit
    SET_BIT(GPIOE->BSRR, GPIO_BSRR_BR_15);

  }
}

/*
 * HAL (Hardware abstraction layer) main function.
 * This function uses a higher level API to work with the GPIO pins.
 * It is only provided as reference.
 */
void main_hal() {
  // Enable GPIOE and GPIOA Clock
  __HAL_RCC_GPIOE_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
```

```
  // Configure LED
  GPIO_InitStruct.Pin = GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;

  HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

  // Configure button
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  // Loop forever
  while (1) {

    // Wait until button is pressed
    while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET);

    // Turn on LED by setting the set bit
    HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_15);

    // Wait until button is released
    while (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_SET);

    // Turn off LED by setting the reset bit
    HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_15);;
  }
}
```

source/c/perf/filter.c

```
#define STM32F303xC

#include "stm32f3xx_hal.h"
#include "semi.h"
#include "time.h"
#include "stackcheck.h"
#include "tracking/cl_particle_filter.h"

void main() {
  // Semihosting channel is needed if measurement data must be printed
  #if defined MEASURE_TIME || defined STACK_PAINT
  openSemihosting();
  #endif

  // Paint the stack
  stack_paint();

  // Initialize the cycle counter
  TimerInit();

  // Run the particle filter algorithm
  float res = start();

  // Save result to pre-defined memory address
```

```
  volatile float *keep = (float*) 0x20001000;
  *keep = res;

  // Print stack usage
  stack_print();

  while(1);
}

void HardFault_Handler() {
    while(1);
}
```

source/c/perf/openocd.sh

```
#!/bin/bash

openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg > bench/times_stack.log
```

source/c/perf/run_tests.sh

```
#!/bin/bash

# Measure compilation time
make clean
{ time make 2>/dev/null; } 2> bench/time.log

# Extract section sizes
arm-none-eabi-objdump -h filter.elf > bench/sizes.log

# Measure timings
make clean
export FEATURES=-DMEASURE_TIME
make
arm-none-eabi-gdb -q -x openocd.gdb filter.elf

# Measure stack sizes
make clean
export FEATURES=-DSTACK_PAINT
make
arm-none-eabi-gdb -q -x openocd.gdb filter.elf

# Summarize timing measurements
python summarize.py bench/times_stack.log > bench/times_stack.json
rm -f bench/times_stack.log
```

source/c/perf/semi.c

```
#include <string.h>

// OPEN system call id
int SYS_OPEN = 0x01;
// WRITE system call id
int SYS_WRITE = 0x05;
// Truncate open mode
int W_TRUNC = 0x4;
// Handle of the semihosting target file
int SEMI_HANDLE;
```

```
/*
 * Perform the actual system call
 * nr: System call id
 * argPointer: Array with three elements
 */
int _syscall(int nr, int argPointer[3]) {
  register int nr_reg asm("r0") = nr;
  register int* arg_reg asm("r1") = argPointer;
  register int ret asm("r0");
  // GCC ASM: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
  // ARM Semihosting:
  // https://www.keil.com/support/man/docs/armcc/armcc_pge1358787048379.htm
  __asm__ volatile (
    "bkpt␣0xab\n\t"     // Call semihosting breakpoint (when enabled on host)
    : "=r" (ret)        // Bind ret as output register variable
    : "r" (nr_reg),     // Bind nr_reg as input register variable
    "r" (arg_reg),      // Bind arg_reg as input register variable
    "m" (argPointer[0]),
    "m" (argPointer[1]),
    "m" (argPointer[2])
  );
  return ret;
}

/*
 * Perform a semihosting system call.
 * nr: System call id
 * arg1-arg3: System call arguments
 */
int syscall(int nr, int arg1, int arg2, int arg3) {
  int args[] = {arg1, arg2, arg3};
  return _syscall(nr, args);
}

/*
 * Perform the open system call.
 * name: Name of the file
 * mode: Open mode
 * size: Size of the name
 */
int open(char* name, int mode, int size) {
  return syscall(SYS_OPEN, (int) name, mode, size);
}

/*
 * Write a string over the semihosting channel.
 * Can only be called when the target was opened with openSemihosting.
 * data: The string
 * size: The length of the string
 */
int debug(char* data, int size) {
  int written = 0;
  while (written < size){
    int res = syscall(SYS_WRITE, SEMI_HANDLE,
    (int) &data[written], size - written);
    if (res == 0){
      break;
    } else if (res <= size - written){
```

```
      written = (size - written) - res;
    } else if (res > 0xfffffff0){
      break;
    } else {
      return -1;
    }
  }
  return size;
}

/*
 * Formats an integer value and writes it over the semihosting channel.
 * Note that prefixLen + length of number + suffixLen must be < 30
 * val: The value to print
 * base: The base of the number (2-16)
 * prefix: A string to prepend
 * suffix: A string to append
 */
void debugInt(int val, int base, char* prefix,
              char* suffix) {
  if (base > 16)
      return;

  char const digit[] = "0123456789ABCDEF";
  char b[30];
  char* p = b;
  int shifter = val;
  int size = 0;
  int i;

  // Add prefix
  for (i = 0; i < strlen(prefix); i++) {
    *p++ = prefix[i];
    size++;
  }

  // Compute number length
  do {
    ++p;
    shifter = shifter / base;
    size++;
  } while(shifter);

  // Add suffix
  for (i = 0; i < strlen(suffix); i++) {
    *(p+i) = suffix[i];
    size++;
  }
  *(p+i) = '\0';

  // Write number
  do {
      *--p = digit[val % base];
      val = val / base;
  } while(val);

  debug(b, size);
}
```

```
/*
 * Formats a float value and writes it over the semihosting channel.
 * fVal: The value to print
 */
void debugFloat(float fVal)
{
    char result[21];
    int dVal, dec, i, j;

    char sign = '+';

    if (fVal < 0) {
      fVal = -fVal;
      sign = '-';
    }

    fVal += 0.005;

    dVal = fVal;
    dec = (int)(fVal * 100.f) % 100;

    result[0] = '\0';
    result[1] = '\n';
    result[2] = ' ';
    result[3] = (dec % 10) + '0';
    result[4] = (dec / 10) + '0';
    result[5] = '.';

    i = 6;
    while (dVal > 0)
    {
        result[i] = (dVal % 10) + '0';
        dVal /= 10;
        i++;
    }

    result[i] = sign;
    i++;

    char str[21];

    for (i=i-1, j=0; i>=0; i--, j++)
        str[j] = result[i];

    debug(str, j - 1);
}

/*
 * Opens the semihosting channel so that
 * writes can be performed.
 */
void openSemihosting(){
  SEMI_HANDLE = open(":tt", W_TRUNC, 3);
}
```

source/c/perf/semi.h

```
#ifndef SEMI_H
```

```c
#define SEMI_H

void openSemihosting();
void debugInt(int val, int base, char* prefix, char* suffix);
void debugFloat(float fVal);

#endif // SEMI_H
```

source/c/perf/stackcheck.c

```c
#include "semi.h"

extern unsigned int _estack;
extern unsigned int _sstack;

// Only implement functions when stack painting is enabled
#ifdef STACK_PAINT

// Paint the stack beginning at current stack pointer
void stack_paint() {
    unsigned int* stack_top = &_sstack;
    unsigned int stack_now;

    // Read current stack pointer
    asm (
        "mov␣%0,␣sp"
        : "=r" (stack_now)
        ::
    );

    unsigned int size = (stack_now - (unsigned int) stack_top) / 4;

    // Fill with pattern
    for (unsigned int i = 0; i < size; i++) {
        *stack_top = 0xBADEAFFE;
        stack_top++;
    }
}

// Get the maximum stack size by analyzing where the
// pattern still stands
unsigned int stack_get() {
    unsigned int* stack_bottom = &_estack - 1;

    unsigned int following_patterns = 0;
    unsigned int stack_size = 0;

    while (stack_bottom > &_sstack) {
        if (*stack_bottom == 0xBADEAFFE) {
            following_patterns++;
        } else {
            following_patterns = 0;
        }

        if (following_patterns == 1) {
            stack_size = (unsigned int) (&_estack - (stack_bottom + 1)) * 4;
        }

        stack_bottom--;
```

```
    }

    return stack_size;
}

// Print the maximum stack size over the semihosting channel
void stack_print() {
    unsigned int size = stack_get();
    debugInt(size, 16, "0x", "␣bytes\n");
}

#else
void stack_paint() {}
unsigned int stack_get() { return 0; }
void stack_print() {}
#endif
```

source/c/perf/stackcheck.h

```
#ifndef STACKCHECK_H
#define STACKCHECK_H

void stack_paint();
unsigned int stack_get();
void stack_print();

#endif // STACKCHECK_H
```

source/c/perf/summarize.py

```
import sys

functions = {}

if len(sys.argv) < 2:
  print("Provide␣input␣file");
  exit();

with open(sys.argv[1]) as f:
  for line in f.readlines():
    if line.startswith("-"):
      continue
    elif line.startswith("0x"):
      functions["stack"] = line.split("␣")[0]
      continue

    parts = line.split("␣")
    name = parts[0]
    val = int(parts[1])
    if name in functions:
      old = functions[name]
      functions[name] = {'time': old['time'] + val, 'num': old['num'] + 1, \
                  'max': max(old["max"], val), 'min': min(old["min"], val)}
    else:
      functions[name] = {'time': val, 'num': 1, 'max': val, 'min': val}

# Compute average
for f in functions:
  function = functions[f]
```

```
  if type(function) is dict:
    function['avg'] = function["time"] / function["num"]

print(functions)
```

source/c/perf/time.c

```
#include "stm32f3xx_hal.h"
#include "semi.h"

TIM_HandleTypeDef  TimHandle;

int subtractCount = 0;

// Only implement functions when time measurements are enabled
#ifdef MEASURE_TIME

// Start the timer by returning the current cycle count
unsigned int TimerStart() {
  return DWT->CYCCNT - subtractCount;
}

// Stop the timer and print the amount of elapsed cycles since the given start value
void TimerEnd(unsigned int start_val, char* name){
  uint32_t now = DWT->CYCCNT;
  uint32_t endTime = now - subtractCount;
  uint32_t diff;
  if (endTime >= start_val){
    diff = endTime - start_val;
  } else {
    diff = (UINT32_MAX - start_val) + endTime;
  }
  debugInt(diff, 10, name, "␣cycles\n");
  uint32_t spent = DWT->CYCCNT - now;
  subtractCount += spent;
}


// Initialize the timer by enabling cycle counting
void TimerInit() {
  // Enable DWT cycle counter register
  DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

#else
unsigned int TimerStart() {return 0;}
void TimerEnd(unsigned int start_val, char* name) {}
void TimerInit() {}
#endif
```

source/c/perf/time.h

```
#ifndef TIME_H
#define TIME_H

unsigned int TimerStart();
void TimerEnd(unsigned int s, char* n);
void TimerInit();
```

```
#endif // TIME_H
```

source/c/perf/tracking/cl_particle.h

```
#ifndef CL_PARTICLE_H
#define CL_PARTICLE_H

typedef struct
{
    float x;
    float y;
    float w;
    float phi;
    float omega;
} cl_particle;

#endif // CL_PARTICLE_H
```

source/c/perf/tracking/cl_particle_filter.c

```
#include "cl_particle_filter.h"
#include "cl_particle.h"
#include "cl_pendulum.h"
#include "../time.h"

#include <math.h>
#define PI 3.14159265f

//Number of samples
#define N 10

// Number of inputs
#define M 10

//Duration of one timeslot
float dt;

/*
 * Resample particle space.
 * Get rid of particles with low weights and place new
 * particles at places with other high-weight particles.
 * Also resets all weights to 1/N
 */
void resample(cl_particle* particles)
{
    unsigned int tval = TimerStart();

    float c[N];
    float U[N];

    //Initialize the CSW(cumulative sum of weights)
    c[0] = particles[0].w;

    for(int i=1; i<N; i++) {
        c[i]= c[i-1] + particles[i].w;
    }

    int i=0;
```

```
    // Draw a starting point
    // Normally would use a random number,
    // but to be able to compare Rust and C,
    // 0.54 is random enough!
    U[0] = 0.54f * (1.0f / N);

    for(int j=0; j<N; j++) {
        U[j] = U[0] + (1.0f/N) * j;

        while(U[j] > c[i]) {
            i++;
        }

        particles[j].x = particles[i].x;
        particles[j].y = particles[i].y;
        particles[j].phi = particles[i].phi;
        particles[j].omega = particles[i].omega;

        particles[j].w = 1.0f/N;
    }

    TimerEnd(tval, "resample␣");
}

/* This is basically the first step of filtering: Prediction
 * Move particles according to the motion model.
 * In this case: A differential equation for typical pendulum movement
 */
void calculate_proposal_pendulum(cl_particle* particle)
{
    unsigned int tval = TimerStart();

    //Known Parameters:
    float l = 1.0;    //Length of the pendulum in m
    float x_a = 0;  //Attachment point x
    float y_a = 0;  //Attachment point y
    float r = 0.2;    //Friction in 1/s
    float deltat = dt; //Time interval in s
    float g = 9.81; //Acceleration due to gravity
    float gdl = g/l;

    //Unknown parameters:
    float phi;      //phi at time 0 in radians
    float omega;    //angular velocity
    float omegazw;  //Derivative of Omega
    float phizw;    //Derivative of phi
    float omeganeu;

    phi = particle->phi;
    omega = particle->omega;

    omegazw = omega + (-gdl * sinf(phi) - r * omega) * deltat;
    phizw = phi + omega * deltat;
    omeganeu = omega + (-gdl * sinf(0.5f * (phi + phizw)) - r * 0.5f
             * (omega + omegazw)) * deltat;
    phi = phi + 0.5f * (omega + omeganeu) * deltat;
    omega = omeganeu;
```

```
    // Update particle for the new time step
    particle->x = -sinf(phi)+ x_a;
    particle->y = -cosf(phi)+ y_a;
    particle->phi = phi;
    particle->omega = omega;

    TimerEnd(tval, "proposal␣");
}

/*
 * Changes the weight of a particle according to its distance to the measurement.
 */
void calculate_importance_density(cl_particle* measurement,
                                  cl_particle* particle, float* erg)
{
    unsigned int tval = TimerStart();

    float dx = fabsf(particle->x - measurement->x);
    float dy = fabsf(particle->y - measurement->y);

    *erg = 1.0f/(dx * dy) + particle->w;

    TimerEnd(tval, "importance␣");
}

/*
 * This is the actual filtering process, this includes every step but
 * resampling and initialization.
 */
void SIR_filter(cl_particle* particles, cl_particle *measurement)
{
    unsigned int tval = TimerStart();
    // Tmp variable for the weight before normalizing
    float erg[N];

    for(int i=0; i<N; i++) {
        //calculate the proposal to predict the next position of the particles
        calculate_proposal_pendulum(&particles[i]);

        //Evaluate the importance weights
        calculate_importance_density(measurement, &particles[i], &erg[i]);

    }

    //Calculate total weight
    float t = erg[0];
    for(int i = 1; i<N; i++) {
        t = t + erg[i];
    }

    // Normalize weights
    for(int i = 0; i<N; i++) {
        particles[i].w = erg[i] * (1.0f/t);
    }

    TimerEnd(tval, "sir_filter␣");
}
```

```
/*
 * Run complete particle filter algorithm.
 */
float start()
{
    cl_particle measurements[M];
    cl_particle particles[N];

    // Generate pendulum data
    unsigned int tval = TimerStart();
    generate_pendulum_data(measurements, M, &dt);
    TimerEnd(tval, "generate␣");

    // Generate N samples with weigth N/1 that are equally distributed
    tval = TimerStart();
    for(int i = 0; i<N; i++){

        particles[i].w = 1.0f/N;
        particles[i].phi = i * ((2.0f*PI)/N);
        particles[i].x = -sinf(particles[i].phi);
        particles[i].y = -cosf(particles[i].phi);
        particles[i].omega = 0.0f;
    }
    TimerEnd(tval, "initialize␣");

    tval = TimerStart();
    // Perform filter and resample steps on every measurement
    for (int k = 0; k < M; k++) {
        SIR_filter(particles, &measurements[k]);
        resample(particles);
    }
    TimerEnd(tval, "filter␣");

    // Compute result value
    float res = 0;
    for (int i = 0; i < N; i++) {
        res += particles[i].phi * particles[i].w;
    }

    return res;
}
```

source/c/perf/tracking/cl_particle_filter.h

```
#ifndef CL_PARTICLE_FILTER_H
#define CL_PARTICLE_FILTER_H

#include "cl_particle.h"

void resample(cl_particle* particles);
void calculate_proposal_pendulum(cl_particle* particle);
void calculate_importance_density(cl_particle* measurement, cl_particle* particle,
                                  float* erg);
void SIR_filter(cl_particle* particles, cl_particle *measurement);
float start();

#endif // CL_PARTICLE_FILTER_H
```

source/c/perf/tracking/cl_particle_filter_data.c

```c
#include "cl_particle_filter.h"
#include "cl_particle.h"
#include "cl_pendulum.h"
#include "../time.h"

#include <math.h>

//Number of samples
#define N 10
#define PI 3.14159265f

// Number of inputs
#define M 10

//Duration of one timeslot
float dt;

cl_particle particles[N];
cl_particle measurements[M];

float erg[N];

float c[N];
float U[N];

/*
 * Resample particle space.
 * Get rid of particles with low weights and place new
 * particles at places with other high-weight particles.
 * Also resets all weights to 1/N
 */
void resample()
{
    unsigned int tval = TimerStart();

    //Initialize the CSW(cumulative sum of weights)
    c[0] = particles[0].w;

    for(int i=1; i<N; i++) {
        c[i]= c[i-1] + particles[i].w;
    }

    int i=0;

    // Draw a starting point
    // Normally would use a random number,
    // but to be able to compare Rust and C,
    // 0.54 is random enough!
    U[0] = 0.54f * (1.0f / N);

    for(int j=0; j<N; j++) {
        U[j] = U[0] + (1.0f/N) * j;

        while(U[j] > c[i]) {
            i++;
        }
```

```
        particles[j].x = particles[i].x;
        particles[j].y = particles[i].y;
        particles[j].phi = particles[i].phi;
        particles[j].omega = particles[i].omega;

        particles[j].w = 1.0f/N;
    }

    TimerEnd(tval, "resample␣");
}

/* This is basically the first step of filtering: Prediction
 * Move particles according to the motion model.
 * In this case: A differential equation for typical pendulum movement
 */
void calculate_proposal_pendulum(int i)
{
    unsigned int tval = TimerStart();

    //Known Parameters:
    float l = 1.0;    //Length of the pendulum in m
    float x_a = 0;  //Attachment point x
    float y_a = 0;  //Attachment point y
    float r = 0.2;    //Friction in 1/s
    float deltat = dt; //Time interval in s
    float g = 9.81; //Acceleration due to gravity
    float gdl = g/l;

    //Unknown parameters:
    float phi;      //phi at time 0 in radians
    float omega;    //angular velocity
    float omegazw;  //Derivative of Omega
    float phizw;    //Derivative of phi
    float omeganeu;

    phi = particles[i].phi;
    omega = particles[i].omega;

    omegazw = omega + (-gdl * sinf(phi) - r * omega) * deltat;
    phizw = phi + omega * deltat;
    omeganeu = omega + (-gdl * sinf(0.5f * (phi + phizw)) - r * 0.5f
             * (omega + omegazw)) * deltat;
    phi = phi + 0.5f * (omega + omeganeu) * deltat;
    omega = omeganeu;

    // Update particle for the new time step
    particles[i].x = -sinf(phi)+ x_a;
    particles[i].y = -cosf(phi)+ y_a;
    particles[i].phi = phi;
    particles[i].omega = omega;

    TimerEnd(tval, "proposal␣");
}

/*
 * Changes the weight of a particle according to its distance to the measurement.
 */
void calculate_importance_density(int k, int i)
```

```
{
    unsigned int tval = TimerStart();

    float dx = fabsf(particles[i].x - measurements[k].x);
    float dy = fabsf(particles[i].y - measurements[k].y);

    erg[i] = 1.0f/(dx * dy) + particles[i].w;

    TimerEnd(tval, "importance␣");
}

/*
 * This is the actual filtering process, this includes every step but
 * resampling and initialization.
 */
void SIR_filter(int k)
{
    unsigned int tval = TimerStart();

    for(int i=0; i<N; i++) {
        //calculate the proposal to predict the next position of the particles
        calculate_proposal_pendulum(i);

        //Evaluate the importance weights
        calculate_importance_density(k, i);

    }

    //Calculate total weight
    float t = erg[0];
    for(int i = 1; i<N; i++) {
        t = t + erg[i];
    }

    // Normalize weights
    for(int i = 0; i<N; i++) {
        particles[i].w = erg[i] * (1.0f/t);
    }

    TimerEnd(tval, "sir_filter␣");
}

/*
 * Run complete particle filter algorithm.
 */
float start()
{
    // Generate pendulum data
    unsigned int tval = TimerStart();
    generate_pendulum_data(measurements, M, &dt);
    TimerEnd(tval, "generate␣");

    //Generate N samples with weigth N/1 that are equally distributed

    tval = TimerStart();
    for(int i = 0; i<N; i++){

        particles[i].w = 1.0f/N;
```

```
        particles[i].phi = i * ((2.0f*PI)/N);
        particles[i].x = -sinf(particles[i].phi);
        particles[i].y = -cosf(particles[i].phi);
        particles[i].omega = 0.0;
    }
    TimerEnd(tval, "initialize␣");

    tval = TimerStart();
    // Perform filter and resample steps on every measurement
    for (int k = 0; k < M; k++) {
        SIR_filter(k);
        resample();
    }
    TimerEnd(tval, "filter␣");

    // Compute result value
    float res = 0;
    for (int i = 0; i < N; i++) {
        res += particles[i].phi * particles[i].w;
    }

    return res;
}
```

source/c/perf/tracking/cl_particle_filter_data.h

```
#ifndef CL_PARTICLE_FILTER_H
#define CL_PARTICLE_FILTER_H

void resample();
void calculate_proposal_pendulum();
void calculate_importance_density(int k, int i);
void SIR_filter(int k);
float start();

#endif // CL_PARTICLE_FILTER_H
```

source/c/perf/tracking/cl_pendulum.c

```
#include "cl_pendulum.h"
#include <math.h>

void DGlStep (float *phi, float *omega, float dt, float gdl, float r);

void generate_pendulum_data(cl_particle *measurements, int maxMeasures,
                            float *deltat_ret) {
    // Parameters
    float phi = 2.0;
    float omega = 0.0;
    float time = 0.0;
    float deltat = 0.1;
    float l = 1.0;
    float g = 9.81;
    float r = 0.2;
    float x_a = 0;
    float y_a = 0;
    float accuracy = 0.15;
    int measureId = 0;
```

```
    *deltat_ret = deltat;

    // Simulation loop
    while (time < maxMeasures * deltat)
    {
        float x = -sinf(phi) + x_a;
        float y = -cosf(phi)+ y_a;

        measurements[measureId].x = x;
        measurements[measureId].y = y;
        measurements[measureId].w = accuracy;
        measureId++;

        // Compute simulation step
        DGlStep (&phi, &omega, deltat, g / l, r);
        time += deltat;

    }
}

void DGlStep(float *phi, float *omega, float dt, float gdl, float r)
{
    float omegazw = *omega + (-gdl * sinf(*phi) - r * *omega) * dt;
    float phizw = *phi + *omega * dt;
    float omeganeu = *omega + (-gdl * sinf(0.5f * (*phi + phizw)) - r * 0.5f
                    * (*omega + omegazw)) * dt;
    *phi = *phi + 0.5f * (*omega + omeganeu) * dt;
    *omega = omeganeu;
}
```

source/c/perf/tracking/cl_pendulum.h

```
#ifndef CL_PENDULUM_H
#define CL_PENDULUM_H

#include "cl_particle.h"

void generate_pendulum_data(cl_particle *measurements, int maxMeasures,
                            float *deltat_ret);

#endif // CL_PENDULUM_H
```

source/c/tools/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(toolDemo)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)

#include_directories(${CONAN_INCLUDE_DIRS_JSON-C})

add_executable(toolDemo toolDemo.c)

target_link_libraries(toolDemo CONAN_PKG::json-c)
```

source/c/tools/conanfile.txt

```
[requires]
json-c/0.13.1

[generators]
cmake
```

source/c/tools/toolDemo.c

```c
#include <stdio.h>
#include <json-c/json.h>

typedef struct {
  char* firstName;
  char* lastName;
} Human;

typedef struct {
  Human parents[2];
  Human child;
} Family;

struct json_object* serialize_human(Human* human) {
  struct json_object* target = json_object_new_object();
  json_object_object_add(target, "first_name",
                         json_object_new_string(human->firstName));
  json_object_object_add(target, "last_name",
                         json_object_new_string(human->lastName));
  return target;
}

struct json_object* serialize_parents(Human parents[2]) {
  struct json_object* parents_res = json_object_new_array();
  for (int i = 0; i < 2; i++) {
    json_object_array_add(parents_res, serialize_human(&parents[i]));
  }
  return parents_res;
}

struct json_object* serialize_family(Family* family) {
  struct json_object* family_res = json_object_new_object();
  json_object_object_add(family_res, "parents",serialize_parents(family->parents));
  json_object_object_add(family_res, "child", serialize_human(&family->child));
  return family_res;
}

void main() {
  // Create data
  Family flintstones = {
          {
            {"Fred", "Flintstone" },
            {"Wilma", "Flintstone"}
          },
          {"Pebbles", "Flintstone"},
  };

  struct json_object* flintstones_json = serialize_family(&flintstones);
  printf("The␣family:␣\n%s\n", json_object_to_json_string(flintstones_json));
}
```

source/rust/bare/.cargo/config

```
[target.thumbv7em-none-eabihf]
# Set compiler flag to include the link script
rustflags = ["-C", "link-arg=-Tlink.x"]
# Command to run when execution cargo run
runner = "arm-none-eabi-gdb -q -x openocd.gdb"

# Set default compiler target triple
[build]
target = "thumbv7em-none-eabihf"
```

source/rust/bare/rt/.cargo/config

```
# Set compiler flag to include the link script
[target.thumbv7em-none-eabihf]
rustflags = ["-C", "link-arg=-Tlink.x"]

# Set default compiler target triple
[build]
target = "thumbv7em-none-eabihf"
```

source/rust/bare/rt/src/lib.rs

```
// Crate without standard lib
#![no_std]

use core::panic::PanicInfo;
use core::ptr;

// Put address of startup function (reset vector) into separate section
#[link_section = ".vector_table.reset_vector"]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;

// Define panic handler for unrecoverable runtime errors
#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}

#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    extern "C" {
        static mut _sbss: u8;
        static mut _ebss: u8;
        static mut _sdata: u8;
        static mut _edata: u8;
        static _sidata: u8;
    }

    // Initialize BSS section to 0
    let count = &_ebss as *const u8 as usize - &_sbss as *const u8 as usize;
    ptr::write_bytes(&mut _sbss as *mut u8, 0, count);

    // Copy DATA section from FLASH to RAM
    let count = &_edata as *const u8 as usize - &_sdata as *const u8 as usize;
    ptr::copy_nonoverlapping(&_sidata as *const u8, &mut _sdata as *mut u8, count);

    // Call main
```

```
    extern "Rust" {
        fn main() -> !;
    }

    main()
}

// Define "entry" macro for easier definition of main
#[macro_export]
macro_rules! entry {
    ($path:path) => {
        #[export_name = "main"]
        pub unsafe fn __main() -> ! {
            let f: fn() -> ! = $path;

            f()
        }
    };
}
```

source/rust/crates/Cargo.toml

```
[package]
name = "embed-demo-crates"
version = "0.1.0"
authors = ["Nico Borgsmueller <nico.borgsmueller@mbda-systems.de>"]
edition = "2018"

[dependencies]
cortex-m = "0.6.3"
cortex-m-rt = "0.6.13"
panic-semihosting = "0.5.3"
cortex-m-semihosting = "0.3.5"

[dependencies.stm32f3]
features = ["stm32f303", "rt"]
version = "0.12.1"

[profile.release]
debug = true
opt-level = 3
```

source/rust/crates/src/main.rs

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

// Use semihosting strategy for panics (by including a crate)
use panic_semihosting as _;

use cortex_m_rt::entry;
use stm32f3::stm32f303;

#[entry]
fn main() -> ! {
    let peripherals = stm32f303::Peripherals::take().unwrap();

    // Enable peripheral clock for IO ports A and E
    peripherals.RCC.ahbenr.modify(|_, w| {
```

```
        w
            .iopeen().set_bit()
            .iopaen().set_bit()
    });

    // Set mode of button to input
    peripherals.GPIOA.moder.modify(|_, w| {
        w.moder0().input()
    });

    // Set mode of LED 9 to output
    peripherals.GPIOE.moder.modify(|_, w| {
        w.moder9().output()
    });

    loop {
        while peripherals.GPIOA.idr.read().idr0().bit_is_clear() {}

        // Write to GPIO bit set/reset register
        peripherals.GPIOE.bsrr.write(|w| {
            // Turn LED 9 on
            w.bs9().set_bit()
        });

        while peripherals.GPIOA.idr.read().idr0().bit_is_set() {}

        // Write to GPIO bit set/reset register
        peripherals.GPIOE.bsrr.write(|w| {
            // Turn LED 9 off
            w.br9().set_bit()
        });
    }
}
```

source/rust/perf/Cargo.toml

```
[package]
name = "embed-perf"
version = "0.1.0"
authors = ["Nico Borgsmueller <nico.borgsmueller@mbda-systems.de>"]
edition = "2018"

[features]
paint_stack = []
measure_time = []
use_data = []
print = []

[dependencies]
cortex-m = "0.6.3"
cortex-m-rt = "0.6.13"
panic-semihosting = "0.5.3"
cortex-m-semihosting = "0.3.5"
libm = "0.2.1"

[dependencies.arrayvec]
version = "0.5.2"
default-features = false
```

```
[dependencies.stm32f3]
features = ["stm32f303", "rt"]
version = "0.12.1"

[profile.release]
opt-level = 's'
lto = true
```

source/rust/perf/run_tests.sh

```
#!/bin/bash
cargo clean
{ time cargo +nightly build --release -q 2>/dev/null; } 2> bench/time.log

arm-none-eabi-objdump -h target/thumbv7em-none-eabihf/release/embed-perf>sizes.log

cargo +nightly run --release --features measure_time,use_data
#cargo +nightly run --release --features measure_time
cargo +nightly run --release --features paint_stack,use_data
#cargo +nightly run --release --features paint_stack

python summarize.py bench/times_stack.log > bench/times_stack.json
rm -f bench/times_stack.log
```

source/rust/perf/src/main.rs

```
#![no_std]
#![no_main]
#![feature(asm)]
#![feature(core_intrinsics)]

// Conditional imports for the different features:

mod timing;
mod measurement;

#[cfg(feature="paint_stack")]
mod stackcheck;

#[cfg(not(feature="use_data"))]
mod particle_filter;
#[cfg(not(feature="use_data"))]
mod pendulum;

#[cfg(feature="use_data")]
mod particle_filter_data;
#[cfg(feature="use_data")]
mod pendulum_data;

#[cfg(debug_assertions)]
use panic_semihosting as _;

#[cfg(not(debug_assertions))]
use core::intrinsics;
#[cfg(not(debug_assertions))]
use core::panic::PanicInfo;

#[cfg(not(debug_assertions))]
#[panic_handler]
```

```
fn panic(_info: &PanicInfo) -> ! {
    intrinsics::abort()
}

use cortex_m_rt::entry;

#[cfg(feature="paint_stack")]
use crate::stackcheck::StackChecker;
#[cfg(any(feature="paint_stack",feature="print"))]
use cortex_m_semihosting::hio;
#[cfg(any(feature="paint_stack",feature="print"))]
use core::fmt::Write;

#[cfg(not(feature="use_data"))]
use crate::particle_filter::ParticleFilter;
#[cfg(feature="use_data")]
use crate::particle_filter_data::ParticleFilter;

#[cfg(feature="print")]
use core::str;

use core::ptr;

#[allow(unused_imports)] // Required to link the interrupt handlers
use stm32f3::stm32f303::{interrupt, Interrupt, NVIC};

#[entry]
fn main() -> ! {

    #[cfg(feature="paint_stack")]
    StackChecker::paint();

    #[cfg(any(feature="paint_stack",feature="print"))]
    let mut h_std_out = hio::hstdout().unwrap();

    // Run the particle filter
    let mut filter = ParticleFilter::init();
    let res = filter.start();

    // To be able to check the result in GDB with x/fw 0x20001000
    unsafe {
        ptr::write_volatile(0x20001000 as *mut f32, res);
    }

    // If the print feature is enabled, log the result over the semihosting channel
    #[cfg(feature="print")]
    {
        let res_str = to_str(res);
        let res_str = str::from_utf8(&res_str).unwrap();

        writeln!(h_std_out, "Result:␣{}", res_str).ok();
    }

    // If the paint_stack feature is enabled, print the stack size over semihosting
    #[cfg(feature="paint_stack")]
    {
        let stack_size = StackChecker::get();
        writeln!(h_std_out, "{:#x}␣bytes", stack_size).ok();
```

```
    }

    loop {}
}

// Function for converting the resulting float into a string
#[cfg(feature="print")]
fn to_str(mut f_val: f32) -> [u8; 10]
{
    let sign = if f_val < 0.0 {
        f_val = -f_val;
        '-'
    } else {
        '+'
    };

    f_val += 0.005;

    let mut d_val = f_val as i32;
    let dec = (f_val * 100.0) as i32 % 100;

    let mut bytes = [0_u8; 10];
    bytes[0] = (dec % 10) as u8 + '0' as u8;
    bytes[1] = (dec / 10) as u8 + '0' as u8;
    bytes[2] = '.' as u8;

    let mut i = 3;
    while d_val > 0 {
        bytes[i] = (d_val % 10) as u8 + '0' as u8;
        d_val /= 10;
        i+=1;
    }

    bytes[i] = sign as u8;

    bytes.reverse();
    bytes
}
```

source/rust/perf/src/measurement.rs

```
#[derive(Clone, Copy)]
pub struct Measurement {
    pub x: f32,
    pub y: f32,
    pub accuracy: f32
}
```

source/rust/perf/src/particle_filter.rs

```
use crate::timing::TimingManager;
use crate::pendulum::Pendulum;
use crate::measurement::Measurement;

use arrayvec::ArrayVec;
use libm::{sinf, cosf};

// Number of particles
const N : usize = 10;
```

```rust
// Trait to get the absolute value
trait Abs {
    fn abs(self) -> Self;
}

impl Abs for f32 {
    fn abs(self) -> Self {
        if self >= 0_f32 { self } else { -self }
    }
}

#[derive(Clone, Copy)]
struct Particle {
    x: f32,
    y: f32,
    w: f32,
    phi: f32,
    omega: f32
}

pub struct ParticleFilter {
    particles: [Particle; N]
}

impl ParticleFilter {

    pub fn init() -> ParticleFilter {
        ParticleFilter { particles: [Particle {x: 0.0, y: 0.0, w: 0.0, phi: 0.0,
                                               omega: 0.0 }; N] }
    }

    pub fn start(&mut self) -> f32 {
        let cortex_peripherals = cortex_m::Peripherals::take().unwrap();
        let timing = TimingManager::init(&cortex_peripherals.DWT);

        let measurements = {
            let _t = timing.time("generate");
            Pendulum::generate()
        };

        {
            let _t = timing.time("initialize");
            for (i, particle) in self.particles.iter_mut().enumerate() {
                particle.w = 1.0/N as f32;
                particle.phi= i as f32 * ((2_f32 * core::f32::consts::PI)/N as f32);
                particle.x = -sinf(particle.phi);
                particle.y = -cosf(particle.phi);
            }
        }

        {
            let _t = timing.time("filter");
            for measurement in measurements.0.iter() {
                self.sir_filter(measurement, measurements.1, &timing);
                self.resample(&timing);
            }
        }
```

```
        let res = self.particles.iter().map(|p| p.phi * p.w).sum::<f32>();

        res
    }

    fn sir_filter(&mut self, measurement: &Measurement, delta_t: f32,
                  timing: &TimingManager) {
        let _t = timing.time("sir_filter");
        for particle in self.particles.iter_mut() {
            particle.calculate_proposal_pendulum(delta_t, timing);
            particle.calculate_importance_density(measurement, timing);
        }

        let total = self.particles.iter().map(|p| p.w).sum::<f32>();
        for particle in self.particles.iter_mut() {
            particle.w = particle.w * (1_f32 / total);
        }
    }

    fn resample(&mut self, timing: &TimingManager) {
        let _t = timing.time("resample");
        let csw : ArrayVec<[_; N]> = self.particles.iter()
                                        .scan(0_f32, |state, particle| {
            *state = *state + particle.w;
            Some(*state)
        }).collect();

        let csw = csw.into_inner().expect("CSW array was not completely filled!");

        let mut u = [0_f32; N];
        let init = 0.54 * (1_f32 / N as f32);
        let mut i = 0;

        for j in 0..N {
            u[j] = init + (1_f32 / N as f32) * j as f32;

            while u[j] > csw[i] {
              i += 1;
            }

            self.particles[j].x = self.particles[i].x;
            self.particles[j].y = self.particles[i].y;
            self.particles[j].phi = self.particles[i].phi;
            self.particles[j].omega = self.particles[i].omega;

            self.particles[j].w = 1_f32 / N as f32;
        }
    }
}

impl Particle {
    fn calculate_proposal_pendulum(&mut self, delta_t: f32,
                                   timing: &TimingManager) {
        let _t = timing.time("proposal");
        let l: f32 = 1.0;
        let x_a: f32 = 0.0;
        let y_a: f32 = 0.0;
```

```
        let r: f32 = 0.2;
        let g: f32 = 9.81;
        let gdl = g/l;

        let phi = self.phi;
        let omega = self.omega;

        let omegazw = omega + (-gdl * sinf(phi) - r * omega) * delta_t;
        let phizw = phi + omega * delta_t;
        let omeganeu = omega + (-gdl * sinf(0.5 * (phi + phizw)) - r * 0.5
                    * (omega + omegazw)) * delta_t;
        let phi = phi + 0.5 * (omega + omeganeu) * delta_t;
        let omega = omeganeu;

        self.x = -sinf(phi) + x_a;
        self.y = -cosf(phi) + y_a;
        self.phi = phi;
        self.omega = omega;
    }

    fn calculate_importance_density(&mut self, measurement: &Measurement,
                                    timing: &TimingManager) {
        let _t = timing.time("importance");
        let dx = (self.x - measurement.x).abs();
        let dy = (self.y - measurement.y).abs();

        self.w = 1_f32 / (dx * dy) + self.w;
    }
}
```

source/rust/perf/src/particle_filter_data.rs

```
use crate::timing::TimingManager;
use crate::pendulum_data::Pendulum;
use crate::measurement::Measurement;

use arrayvec::ArrayVec;
use libm::{sinf, cosf};
use core::cell::RefCell;
use cortex_m::interrupt;
use cortex_m::interrupt::Mutex;
use cortex_m::interrupt::CriticalSection;

const N : usize = 10;
const M : usize = 10;
const ACCURACY : f32 = 0.15;

trait Abs {
    fn abs(self) -> Self;
}

impl Abs for f32 {
    fn abs(self) -> Self {
        if self >= 0_f32 { self } else { -self }
    }
}

#[derive(Clone, Copy)]
struct Particle {
```

```
    x: f32,
    y: f32,
    w: f32,
    phi: f32,
    omega: f32
}

pub struct ParticleFilter {}

static particles : Mutex<RefCell<[Particle; N]>>
  = Mutex::new(RefCell::new([Particle { x: 0.0, y: 0.0, w: 0.0,
                                        phi: 0.0, omega: 0.0 }; N]));
static measurements : Mutex<RefCell<[Measurement; M]>>
  = Mutex::new(RefCell::new([Measurement { x: 0.0, y: 0.0,
                                           accuracy: ACCURACY }; M]));

impl ParticleFilter {

    pub fn init() -> ParticleFilter {
        ParticleFilter { }
    }

    pub fn start(&mut self) -> f32 {
        let cortex_peripherals = cortex_m::Peripherals::take().unwrap();
        let timing = TimingManager::init(&cortex_peripherals.DWT);

        let delta_t = {
            let _t = timing.time("generate");
            interrupt::free(|cs| {
                Pendulum::generate(measurements.borrow(cs).borrow_mut())
            })
        };

        {
            let _t = timing.time("initialize");
            interrupt::free(|cs| {
                for (i, particle) in particles.borrow(cs).borrow_mut().iter_mut()
                                                          .enumerate() {
                    particle.w = 1.0/N as f32;
                    particle.phi = i as f32 * ((2_f32
                                * core::f32::consts::PI)/N as f32);
                    particle.x = -sinf(particle.phi);
                    particle.y = -cosf(particle.phi);
                }
            });
        }

        {
            let _t = timing.time("filter");
            interrupt::free(|cs| {
                for measurement in measurements.borrow(cs).borrow().iter() {
                    self.sir_filter(measurement, delta_t, &timing, cs);
                    self.resample(&timing, cs);
                }
            });
        }
        let res = interrupt::free(|cs| {
            particles.borrow(cs).borrow().iter().map(|p| p.phi * p.w).sum::<f32>()
```

```
        });

        res
    }

    fn sir_filter(&mut self, measurement: &Measurement, delta_t: f32,
                  timing: &TimingManager, cs: &CriticalSection) {
        let _t = timing.time("sir_filter");
        for particle in particles.borrow(cs).borrow_mut().iter_mut() {
            particle.calculate_proposal_pendulum(delta_t, timing);
            particle.calculate_importance_density(measurement, timing);
        }

        let total = particles.borrow(cs).borrow().iter().map(|p| p.w).sum::<f32>();
        for particle in particles.borrow(cs).borrow_mut().iter_mut() {
            particle.w = particle.w * (1_f32 / total);
        }
    }

    fn resample(&mut self, timing: &TimingManager, cs: &CriticalSection) {
        let _t = timing.time("resample");
        let csw : ArrayVec<[_; N]> = particles.borrow(cs).borrow().iter()
                                      .scan(0_f32, |state, particle| {
            *state = *state + particle.w;
            Some(*state)
        }).collect();

        let csw = csw.into_inner().expect("CSW array was not completely filled!");

        let mut u = [0_f32; N];
        let init = 0.54 * (1_f32 / N as f32);
        let mut i = 0;

        let mut parts = particles.borrow(cs).borrow_mut();

        for j in 0..N {
            u[j] = init + (1_f32 / N as f32) * j as f32;

            while u[j] > csw[i] {
              i += 1;
            }

            parts[j].x = parts[i].x;
            parts[j].y = parts[i].y;
            parts[j].phi = parts[i].phi;
            parts[j].omega = parts[i].omega;

            parts[j].w = 1_f32 / N as f32;
        }
    }
}

impl Particle {
    fn calculate_proposal_pendulum(&mut self, delta_t: f32,
                                   timing: &TimingManager) {
        let _t = timing.time("proposal");
        let l: f32 = 1.0;
        let x_a: f32 = 0.0;
```

```
        let y_a: f32 = 0.0;
        let r: f32 = 0.2;
        let g: f32 = 9.81;
        let gdl = g/l;

        let phi = self.phi;
        let omega = self.omega;

        let omegazw = omega + (-gdl * sinf(phi) - r * omega) * delta_t;
        let phizw = phi + omega * delta_t;
        let omeganeu = omega + (-gdl * sinf(0.5 * (phi + phizw)) - r * 0.5
                    * (omega + omegazw)) * delta_t;
        let phi = phi + 0.5 * (omega + omeganeu) * delta_t;
        let omega = omeganeu;

        self.x = -sinf(phi) + x_a;
        self.y = -cosf(phi) + y_a;
        self.phi = phi;
        self.omega = omega;
    }

    fn calculate_importance_density(&mut self, measurement: &Measurement,
                                    timing: &TimingManager) {
        let _t = timing.time("importance");
        let dx = (self.x - measurement.x).abs();
        let dy = (self.y - measurement.y).abs();

        self.w = 1_f32 / (dx * dy) + self.w;
    }
}
```

source/rust/perf/src/pendulum.rs

```
use crate::measurement::Measurement;

use libm::{sinf, cosf};

pub struct Pendulum;

const M : usize = 10;

impl Pendulum {
    pub fn generate() -> ([Measurement; M], f32) {
        let mut phi : f32 = 2.0;
        let mut omega : f32 = 0.0;
        let delta_t : f32 = 0.1;
        let l : f32 = 1.0;
        let g : f32 = 9.81;
        let r : f32 = 0.2;
        let x_a : f32 = 0.0;
        let y_a : f32 = 0.0;
        let accuracy : f32 = 0.15;

        let mut measurements = [Measurement {
            x: 0.0,
            y: 0.0,
            accuracy
        }; M];
```

```
        for measurement in measurements.iter_mut() {
            let x = -sinf(phi) + x_a;
            let y = -cosf(phi) + y_a;

            measurement.x = x as f32;
            measurement.y = y as f32;
            let (phi2, omega2) = Pendulum::dgl_step(phi, omega, delta_t, g / l, r);
            phi = phi2;
            omega = omega2;
        }

        (measurements, delta_t)
    }

    fn dgl_step(phi: f32, omega: f32, dt: f32, gdl: f32, r: f32) -> (f32, f32) {
        let omegazw = omega + (-gdl * sinf(phi) as f32 - r * omega) * dt;
        let phizw = phi + omega * dt;
        let omeganeu = omega + (-gdl * sinf(0.5 * (phi + phizw)) as f32 - r
                    * 0.5 * (omega + omegazw)) * dt;
        let phineu = phi + 0.5 * (omega + omeganeu) * dt;
        (phineu, omeganeu)
    }
}
```

source/rust/perf/src/pendulum_data.rs

```
use crate::measurement::Measurement;

use libm::{sinf, cosf};
use core::cell::RefMut;

pub struct Pendulum;

const M : usize = 10;

impl Pendulum {
    pub fn generate(mut measurements: RefMut<[Measurement; M]>) -> f32 {
        let mut phi : f32 = 2.0;
        let mut omega : f32 = 0.0;
        let delta_t : f32 = 0.1;
        let l : f32 = 1.0;
        let g : f32 = 9.81;
        let r : f32 = 0.2;
        let x_a : f32 = 0.0;
        let y_a : f32 = 0.0;

        for measurement in measurements.iter_mut() {
            let x = -sinf(phi) + x_a;
            let y = -cosf(phi) + y_a;

            measurement.x = x as f32;
            measurement.y = y as f32;
            let (phi2, omega2) = Pendulum::dgl_step(phi, omega, delta_t, g / l, r);
            phi = phi2;
            omega = omega2;
        }

        delta_t
    }
```

```
    fn dgl_step(phi: f32, omega: f32, dt: f32, gdl: f32, r: f32) -> (f32, f32) {
        let omegazw = omega + (-gdl * sinf(phi) as f32 - r * omega) * dt;
        let phizw = phi + omega * dt;
        let omeganeu = omega + (-gdl * sinf(0.5 * (phi + phizw)) as f32 - r
                    * 0.5 * (omega + omegazw)) * dt;
        let phineu = phi + 0.5 * (omega + omeganeu) * dt;
        (phineu, omeganeu)
    }
}
```

source/rust/perf/src/stackcheck.rs

```
use core::ptr;

pub struct StackChecker;

impl StackChecker {

    pub fn paint() {
        extern "C" {
            static mut _stack_start: u32;
            static mut _stack_end: u32;
        }

        unsafe {
            let mut stack_start : *mut u32;
            // Get current stack pointer
            asm!("MOV␣{},␣SP", out(reg_thumb) stack_start);

            // Add buffer for the following call stack
            stack_start = stack_start.offset(-0xc);
            let count = stack_start as usize - &_stack_end as *const u32 as usize;
            ptr::write_bytes(&mut _stack_end as *mut u32 as *mut u8, 0xAB, count);
        }
    }

    pub fn get() -> usize {
        extern "C" {
            static mut _stack_start: u32;
            static mut _stack_end: u32;
        }

        let mut stack_size = 0;
        let mut following_patterns = 0;

        unsafe {
            let mut cur_ptr = &_stack_start as *const u32 as *mut u32;
            cur_ptr = cur_ptr.offset(-0x1);

            while cur_ptr > &_stack_end as *const u32 as *mut u32 {
                if *cur_ptr == 0xABABABAB {
                    following_patterns += 1;
                } else {
                    following_patterns = 0;
                }

                if following_patterns == 1 {
                    let stack_start_addr = &_stack_start as *const u32 as usize;
```

```
                    stack_size = stack_start_addr - cur_ptr.offset(0x1) as usize;
                }

                cur_ptr = cur_ptr.offset(-0x1);
            }
        }


        stack_size
    }
}
```

source/rust/perf/src/timing.rs

```
#[cfg(feature="measure_time")]
use cortex_m_semihosting::hio::HStdout;
#[cfg(feature="measure_time")]
use core::fmt::Write;
#[cfg(feature="measure_time")]
use cortex_m_semihosting::hio;


use cortex_m::peripheral::DWT;

#[cfg(feature="measure_time")]
const DWT_CTRL_ENABLE_CYCCNT : u32 = 0x00000001;

// Static mutable is ok here because no concurrency or borrowing is performed
// It is required, so that timings are not influenced by this timing mechanism
#[cfg(feature="measure_time")]
static mut SUBTRACT_COUNT : u32 = 0;

#[cfg(feature="measure_time")]
pub struct Timing<'a> {
    start: u32,
    dwt: &'a DWT,
    h_std_out : HStdout,
    name: &'static str
}

#[cfg(feature="measure_time")]
impl Timing<'_> {

    pub fn init<'t>(dwt: &'t DWT, name: &'static str) -> Timing<'t> {
        let now = unsafe { dwt.cyccnt.read() - SUBTRACT_COUNT };
        let h_std_out = hio::hstdout().unwrap();
        Timing { start: now, dwt, h_std_out, name }
    }
}

#[cfg(feature="measure_time")]
impl Drop for Timing<'_> {

    fn drop(&mut self) {
        let now = self.dwt.cyccnt.read();
        let end = unsafe { now - SUBTRACT_COUNT };
        let diff = if end >= self.start {
            end - self.start
        } else {
```

```
            (u32::max_value() - self.start) + end
        };

        writeln!(self.h_std_out, "{}: {} cycles", self.name, diff).ok();
        let spent = self.dwt.cyccnt.read() - now;
        unsafe {
            SUBTRACT_COUNT += spent;
        }
    }
}

#[cfg(not(feature="measure_time"))]
pub struct Timing;

#[cfg(not(feature="measure_time"))]
impl Timing {
    fn init(_: &DWT, _: &'static str) -> Timing { Timing {} }
}

pub struct TimingManager<'a> {
    dwt: &'a DWT
}

impl TimingManager<'_> {

    #[cfg(feature="measure_time")]
    pub fn init(dwt: &DWT) -> TimingManager {
        // Enable DWT cycle count register
        unsafe {
            dwt.ctrl.modify(|mut val| {
                val |= DWT_CTRL_ENABLE_CYCCNT;
                val
            });
        }

        TimingManager { dwt }
    }

    #[cfg(not(feature="measure_time"))]
    pub fn init(dwt: &DWT) -> TimingManager {
        TimingManager { dwt }
    }

    pub fn time(&self, name: &'static str) -> Timing {
        Timing::init(self.dwt, name)
    }
}
```

source/rust/tools/Cargo.toml

```
[package]
name = "tool_demo"
version = "0.1.0"
authors = ["Nico Borgsmueller <nico.borgsmueller@mbda-systems.de>"]
edition = "2018"

[dependencies]
serde = { version = "1.0", features = ["derive"]}
serde_json = "1.0"
```

source/rust/tools/family.json

```
{ "parents": [ { "first_name": "Fred", "last_name": "Flintstone" },
{ "first_name":"Wilma", "last_name": "Flintstone" } ],
"child": { "first_name": "Pebbles", "last_name": "Flintstone" } }
```

source/rust/tools/src/main.rs

```
use serde::Deserialize;

use std::error::Error;
use std::fs::File;
use std::io::BufReader;

#[derive(Deserialize, Debug)]
struct Human {
  first_name: String,
  last_name: String
}

#[derive(Deserialize, Debug)]
struct Family {
  parents: [Human; 2],
  child: Human
}

fn main() -> Result<(), Box<dyn Error>> {
  let file = File::open("family.json")?;
  let reader = BufReader::new(file);

  let family : Family = serde_json::from_reader(reader)?;

  println!("Deserialized␣family:␣{:#?}", family);

  Ok(())
}
```