MBDA MISSILE SYSTEMS

Technische Hochschule Ingolstadt

# Technische Hochschule Ingolstadt

| | |
|---|---|
| **Specialist area (faculty):** | **Computer Science** |
| **Master's course:** | **Computer Science** |
| **Specialization:** | **Safety and Security** |

## Master's Thesis

Subject:

# Integrating the Future into the Past – Approach to Seamlessly Integrate Newly-Developed Rust-Components into an Existing C++ System


Name and Surname: Philipp von Perponcher



Issued on: 23.10.2023
Submitted on: 17.01.2024


First examiner: Prof. Dr. Sebastian Apel
Second examiner: Prof. Dr. Hans-Michael Windisch

# 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.

Ingolstadt, January 17, 2024

Philipp von Perponcher

This thesis was written as the conclusion of a 1.5 year dual Master's program with Technische Hochschule Ingolstadt and MBDA Deutschland.

I would like to thank all of the people I've met and worked with during this time period, first and foremost my supervisors Rico Lieback and Thomas Britzelmeier for accompanying me throughout my time at MBDA, providing me with exciting projects and help.

I would also like to thank Mr. Prof. Dr. Sebastian Apel for accompanying this thesis over the past months and always being available and helpful whenever questions came up.

A special mention also goes out to my family who supported and helped me throughout my entire studies.

Finally, I would like like to thank my good friends Dominik Bartl, Benjamin Huber, Tycho Mertens, Sebastian Rabau, Grady Orr, Dario Köllner, and Fabian Bretz for proof-reading the thesis thoroughly, allowing me to give it the finishing touch.

# Abstract

The goal of this thesis is to provide a guide that can be used by software developers that wants to include Rust into an existing C++ application.

After a brief summary of the languages, both the basics of integrating Rust code into C++ applications and the reverse way are presented to set a common base of knowledge. Additionally, a few tools and libraries are introduced, namely *bindgen*, *cbindgen*, *cxx*, and the *cc* crate. As they are being used in the following guide, the toolsets *CMake* and *CMakeRust* are presented and explained in more detail to make it easier for developers to include and use them in their own projects.

The guide is separated into three steps which help to identify the required interfaces, implement them and eventually include everything into the C++ compilation process.

Apart from more advanced types based on pointers, the integration is fairly easy and straightforward. Using structs and their implemented methods, Rust is able to replicate a class structure that is typical for object-oriented C++. After providing the required binding files, the Rust component can easily be called from C++ code like any other function. While external Rust methods need to be called using a *function(Object)* syntax, they can be also be converted and used via the standard *Object.function()* syntax by implementing wrapper classes that translate the calls. These can then be included in more complex C++ architectures like inheritance structures.
The usage of the aforementioned advanced types, which are based on pointers, is not covered in this thesis, though the *cxx* library can be used for those.

The compilation for and usage on embedded targets is also possible with its required effort depending on the target architecture. If the necessary target is not officially supported yet, some additional things like the standard library need to be downloaded and compiled manually.

The thesis comes to the conclusion that considering C++ is not officially supported by the Rust foreign function interface, it works very well with the C ABI and apart from one floating point type, no clear limitations or incompatibilities were identified. For interfaces that only work with basic types and to get a general understanding of the topic, this thesis is a good starting point and guide for a developer who is new to the topic.

# Contents


**1. Introduction** **1**
1.1. Motivation and Task . . . 2
1.2. Structure . . . 3

**2. Related Work** **4**

**3. Introduction of Rust and C++** **6**
3.1. Introduction to Rust . . . 6
3.2. Introduction to C++ . . . 12

**4. C++ and Rust Interoperability** **16**
4.1. Basics of Integration . . . 16
4.2. Integrating Rust Code into C++ Applications . . . 17
4.3. Integrating C++ Code into Rust Applications . . . 29
4.4. Equivalents of C++ and Rust . . . 39

**5. The Way from C++ to Rust** **44**
5.1. Approaches to Integration . . . 44
5.2. Step by Step Guide . . . 46
5.3. Full Process Demonstrated at Example Project . . . 48

**6. Embedded Development** **77**
6.1. Code Preparation . . . 77
6.2. Platform Support . . . 77
6.3. Nightly Build . . . 78
6.4. Summary . . . 79

**7. Discussion** **80**

**8. Conclusion** **81**
8.1. Research Questions . . . 81
8.2. Future Work . . . 82

**List of Listings** **83**

**List of Figures** 85

**List of Tables** 86

**Acronyms** 86

**Bibliography** 87

**Appendix A. Source Files** 90

# 1. Introduction

Over the past years and decades, the increasing use of computers in the everyday world also came with a rise in demand for software. This trend leads to more research and investments towards making software better and faster which could be achieved with various approaches like hardware improvements or parallel and thus more optimized development. Another approach could be the improvement of the programming languages themselves. Having a language that prevents errors and helps developers to be more efficient, thus lowering initial production time as well as the time spent fixing bugs later on, is in the interest of the companies paying for that software. An example for that is Rust, which is a programming language that released its first compiler in 2012 and has a unique approach to handling memory to prevent runtime errors. To give a small example for the efficiency of that, according to a former Mozilla engineer, nearly 74% of the security bugs in Firefox's style component would not have been possible if it had been written in Rust. [And12; Hos19]

While new programming languages which 'do things better' are nice to have, it's not always possible to use them in a corporate environment. A prominent example for this exception is industries that require safety-critical software. Due to the very high standard in for example the aerospace or defense industry, there are usually only a few certified compilers and tools that allow for software products to be approved by the authorities. Examples for that include Ada or C++ in combination with frameworks and rulesets like AUTOSAR or DO-178C [Bar14, p. 3][AUT19].

Even though modern programming languages like Rust could help greatly with development by giving access to new approaches and methods that can help with preventing mistakes, it's not always easy to include them in existing or new projects within those industries. This is partially due to them having to follow strict standards which these technologies might not (yet) be certified for but also the long lifespan that these products are developed for, leading to companies not wanting to commit to untested technologies. To name an example, platforms like the Patriot air defense system have been in development since the 1960's and introducing modern programming languages into such systems today could require a rewrite of the existing codebase. Additionally, it can be very challenging for companies to introduce new programming languages on a large scale as developers have to be trained and environments for testing or CI/CD need to be set up.

Some of these problems can be evaded by starting to update and change small components one at a time. These can be developed and maintained by a single person or small team and, due to their limited size and complexity, are easier to design in a way that adheres to the required standards.

Using this approach has the big challenge of how to include those components developed with a new technology like a modern programming language into the existing project structure as their interfaces might not be compatible.

## 1.1. Motivation and Task

The aforementioned challenge was the basis for the topic of this thesis. It is written in cooperation with MBDA Deutschland, a company specializing in the development, production, and maintenance of missiles and defense systems. These products and especially their software have to be developed to the highest levels of safety and security which can be achieved by using error-resistant languages like Ada or by using regular programming languages in combination with strict standards like for example C++ and DO-178C. This does not mean that other programming languages are forbidden to use but having already certified tools significantly simplifies the approval process.

Over the past few years now, there have been thoughts and first approaches into possibly using the programming language Rust for safety-critical projects. This is due to its way of handling memory and the resulting concepts like the ownership of variables that help with developing very robust software. The general idea is supported by efforts from other companies and industries like the consultancy *Ferrous Systems* who have joined forces with the developers of Ada to develop a safety-qualified Rust toolchain [Adaa].

Due to the challenges that come with introducing a new programming language into a corporate environment, the idea of first using smaller components as a proof of concept and feasibility came up, resulting in the topic for this thesis.

The task set out was to compose a general guide that, together with the basics of interoperability between the two languages, covers the incorporation of Rust into a C++ project. The main artifact resulting from this thesis was supposed to be a walkthrough that can be used by a developer who is familiar with both programming languages to plan and realize such an integration. An additional task was to look at how Rust can be used in embedded software projects.

From this, the following research questions were identified:

1. What is required for successfully including Rust in C++ applications and how can such an integration be approached?
2. Are there any limitations to the interoperability of C++ and Rust that could stand in the way of an integration, such as type or interface incompatibilities?

## 1.2. Structure

This thesis will start out by introducing some already existing related work around the topic of integrating Rust into C++ and the other way around in Chapter 2.

Afterwards, Chapter 3 will give a short introduction of both Rust and C++ together with their most important features and ecosystems. It will also go over the history of both languages and some typical fields of application.

Chapter 4 will then cover the basics of interoperability between the two languages, looking at both the integration of Rust into a C++ project as well as the other way around. It is concluded by a section on the data types of C++ and their equivalents in Rust.

The actual guide is included in Chapter 5, preceded by a small introduction of various approaches of how an integration can be done. It is followed by a full walkthrough which covers the introduction of a Rust component into an example project based on a publish-subscribe pattern.

Chapter 6 dives into the world of Embedded Rust and how to compile a basic library for various targets.

The thesis is completed by a reflection of the achieved results and an outlook into the future in Chapters 7 and 8.

# 2. Related Work

Including Rust in projects primarily written in other programming languages is not a new idea and thus quite a few references can be found that deal with the problem of Rust together with other programming languages. To fit the topic of this thesis, this chapter will focus on sources working on the integration with C or C++. Hereby, a majority of the papers that can be found are used for small examples for calling Rust functions from C++ code or the other way around. These will be briefly summed up in the following two sections.

This paper serves as the continuation of a topic first introduced by a Bachelor's Thesis which was written in cooperation with MBDA in 2017 that covered the usage of Rust in an embedded context. In here, Chapter 5.1.3 describes the basics of Rust's interoperability with other languages, especially C and C++. As that section only included a small introduction to the matter, the topic of this thesis was developed in order to get a better understanding of the process as well as some difficulties and limitations that can occur when introducing Rust into an existing C++ environment.[Bor21, Ch. 5.1.3]

### Guides to Integrate Rust into C++

The first big portion of sources that can be found deal with the core problem of this thesis which is the way to call and use Rust functions from C or C++ code. The first source that should be consulted would always be the official Rust documentation which in this case is a chapter called "A little Rust with your C" that can be found in *The Embedded Rust Book* [Theb, Ch. 10.2]. This chapter was used to get a first understanding of the topic such as how to get Rust to expose functions as well as small additions like the `[no_mangle]`-flag, together with a short section on building the project.

It also covers the tool *cbindgen* for automatically creating C-compatible header files, [Cbi]. Though this tool can be used for creating these files for projects of any size, it is more useful on a bigger scale where manually writing them can be labor-intensive. Within this thesis, the few small headers will be simply composed by hand. This source is being followed closely and used as a basic line of orientation in Section 4.2.

### Guides to Integrate C++ into Rust

Even though the topic of this thesis is the integration of Rust into C++ projects, the reverse direction is important as well, as very rarely, components are only being called. Being able to interact with C++ classes, functions, and types from the Rust sections is essential to a complete integration.

Taking *The Embedded Rust Book* again, this time Chapter 10.1 called "A little C with your Rust", it covers the general way to use C and C++ functions within a Rust project, briefly describing how the interface has to be implemented for using it in Rust. The guide also includes a small chapter about building the project as well as introduces two tools: *bindgen* which is used to auto-generate Rust interfaces from C header files and the *cc* crate which helps with compiling C or C++ code and including it into a Rust application[Bin; Ccc]. [Theb, Ch. 10.1]

As the official documentation is a great source for covering the basics, Section 4.3 will use some of its examples like a simple struct or function call.

### Placement of this Thesis

As the topic of this thesis is the integration of Rust components in C++ projects, it uses and builds upon the same basics as the upon named papers and articles. It places itself as a continuation of these, giving the reader a more complete guide in contrast to the single examples that can be found online.

# 3. Introduction of Rust and C++

To get an understanding of the programming languages used in this thesis, they are briefly introduced. Together with a small section about history, the main focuses lie on the toolchains, compilation workflow, and unique characteristics that are responsible for the popularity and strong arguments for using Rust or C++ respectively.

## 3.1. Introduction to Rust

This chapter serves as a general information about the programming language Rust. It is intended to give the reader a basic understanding of the history, the ecosystem and some essential features of the language. The main sources for any Rust content in this thesis are the official Rust documentation found on `https://doc.rust-lang.org` and the book *The Rust Programming Language* [KN18].

### 3.1.1. Development and History

Rust is a fairly new programming language with its development starting in 2006 as a private project by Graydon Hoare. After some basic features like testing and core concepts were realized, Mozilla started sponsoring the language in 2009. The language continued to grow and, after being able to compile itself for the first time in 2011, eventually published its 1.0 release in 2015. [Rusa]

After a big layoff at Mozilla, the financing of Rust was taken over by the Rust Foundation in 2021. Its founding members include AWS, Huawei, Google, Microsoft and Mozilla, which continue to support the development of the language with the exception of the latter, which has been replaced by Meta. [Wil21; Mem]

In the software development community, Rust has been a very popular language over the past years which is reflected by its status of “Most loved/admired language” in the Stack Overflow Developer Survey for eight years in a row. It is also a language that a lot of people want to use or start getting into, with 17.6% people in 2022 and even 30.56% in 2023 stating so. [Staa; Stab; Stac]

Today, Rust applications can be found in a lot of aspects of programming. Examples for these are operating systems like the Linux Kernel or Windows, web browsers like Mozilla Firefox or messenger applications like Signal. [Sim22; Rusb; Ruse; Sig]

It has also been in talks as a possible language for safety-related software. This approach has become much more concrete in recent months and years with the developers of Ada, a very prominent programming language in the aviation and defense industry, joining forces with Ferrous Systems, a consultancy specializing in Rust applications in early 2022. Their ambitious goal has already reached a few crucial milestones, with their first toolchain currently undergoing ISO 26262 and IEC 61508 qualification with TÜV SÜD. [Adaa; Adab]

### 3.1.2. Toolchain - Cargo

Nearly everything that a developer wants to do with Rust can be done using *cargo*, the official package manager. This includes the creation of new projects, compilation into various targets, testing, and of course downloading packages from *crates.io*, the official crate registry.

Cargo is installed together with the official Rust installation and can be invoked using the `cargo` command. Some of the most important commands include

- `cargo new` to create a new project
- `cargo build` to build or rebuild the current project directory if there were any changes
- `cargo run` to build or rebuild the code and execute the resulting application
- `cargo test` to run tests specified in the *tests* folder

An extensive documentation about cargo can be found in *The Cargo Book* [Thea].

#### Cargo.toml

Any flags given to the cargo command can also be specified in a so-called *manifest*, a file called *Cargo.toml*. It provides a way to keep track of not one but several different compile operations and targets, dependencies, as well as the project's metadata. More information, together with all the available options, can be found in Chapter 3.2 of *The Cargo Book*. [Thea]

### Compilation of a Debug or Release Target

Rust and Cargo provide a simple way of building for different profiles with two being predefined. The first and default one is *Debug* which includes "good defaults for development" as Chapter 14.1 in *The Rust Programming Language* describes it [KN18]. The other one is *Release* which can be built by adding `--release` to the Cargo call. This profile has good default values for release versions as it includes a higher number of optimizations for the code as well as omits things like debug assertions which both improve performance but on the other hand also increase the compile-time. Talking about some concrete values as an example, a medium-sized REST server project (~1300 lines of code with additional libraries) required ~40 seconds to compile the *Debug* version compared to ~50 seconds for the *Release* configuration. On the other hand, the startup time when running the server decreased from over 13 to ~0.3 seconds. A more detailed description of the different profiles can be found in the official documentation. [KN18, Ch. 14.1] [Thea, Ch. 3.5]

### Compilation into a Library vs. a Binary Target

The two main applications that will be used in this thesis will be the compilation into a binary and a library target. The former will be needed in Section 4.3 which talks about Rust applications with added C++ components. The latter is more important in Section 4.2 and the entirety of Chapter 5.

The big difference between both approaches is the starting point for compilation. For a library target including the modules *foo* and *bar*, the project structure needs to contain a *lib.rs* file like shown in Listing 3.1. By default, it is located in the same *src/* folder as the other components and is created automatically when a new project is created using
`cargo new <<name>> --lib`.

```rust
mod foo;
mod bar;
```

Listing 3.1: Very simple *lib.rs* file for compiling a library containing the modules *foo* and *bar*

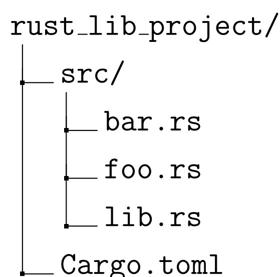


Figure 3.1.: File structure of a simple Rust library project

The *Cargo.toml* file would then contain a line declaring the `crate-type` to be a static library, as seen in line 7 in Listing 3.2.

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

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

Listing 3.2: *Cargo.toml* compiling into a static library target

Running `cargo build` in the *rust_lib_project/* folder results in the files *librust_lib_project.a* and *librust_lib_project.d* in the *rust_lib_project/target/debug/* folder with the file structure being shown in Figure 3.2.

```
rust_lib_project/
├── src/
│   ├── bar.rs
│   ├── foo.rs
│   └── lib.rs
├── Cargo.toml
└── target/
    └── debug/
        ├── librust_lib_project.a
        └── librust_lib_project.d
```

Figure 3.2.: File structure of a simple Rust library project with compiled target

A binary target on the other hand doesn't have to be explicitly specified as it's the default configuration. The starting and thus required file for the compilation is called *main.rs* and contains a *main()* function that is the entry point.

```
rust_bin_project/
├── src/
│   └── main.rs
└── Cargo.toml
```

Figure 3.3.: File structure of a simple Rust program

```rust
fn main() {
    println!("Hello World!");
}
```

Listing 3.3: Simple "Hello World!" example in *main.rs*

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

Listing 3.4: *Cargo.toml* compiling into a binary target

This project would compile into an executable named *rust_bin_project* saved under *rust_bin_project/target/debug/*, resulting in a file structure as seen in Figure 3.4.

```
rust_bin_project/
├── src/
│   └── main.rs
├── Cargo.toml
└── target/
    └── debug/
        └── rust_bin_project
```

Figure 3.4.: File structure of a simple Rust binary project with compiled target

### 3.1.3. Unique Characteristics

#### Ownership

A big reason for Rust's popularity is its way of handling memory. Each variable that is defined has some kind of owner, which can for example be a function, a loop, or, at the simplest level, a code block marked by a set of curly brackets. As soon as this owner goes out of scope, the variable's memory is freed. This approach eliminates the need for a garbage collector, helping with the real-time performance and expectability of Rust programs.

The second feature that anybody new to Rust might not be familiar with is that every variable is immutable by default. By adding the keyword `mut` to the declaration, the variable can be changed, though the compiler warns users from doing so if it's not necessary. A special restriction in this case applies to references. Any variable can have as many immutable references to it as necessary, as they're not able to change the value.

Mutable references on the other hand are a lot more restrictive as only one of them can be valid at a given time. At the same time, no other references may be in scope, even if they're immutable.

Using this mechanic, data races and common errors like null-pointer errors can be prevented at compile-time.

More information on this, together with a lot of well-explained and documented examples can be found in Chapter 4.2 of *The Rust Programming Language*. [KN18, Ch. 4.2]

### Safe and Unsafe

In case the developer wants to do something that is officially forbidden by the compiler, they can use something called 'unsafe Rust', which is "like regular Rust, but gives us extra superpowers" as [KN18] calls it. It is basically an area of Rust code where the compiler does not check for possible errors regarding Rust's memory safety guarantees. [KN18, Ch. 19.1] This can help with low-level programming or generally areas where the developer knows what they are doing while the compiler isn't able to exclude a residual risk and thus normally wouldn't allow a compilation.

An obvious example where *unsafe* Rust is required is the de-referencing of raw pointers as that could lead to a null pointer error, therefore the regular compiler forbids this.

If an unsafe operation is necessary, there are two ways of dealing with it. The first way is marking the Rust function itself as unsafe which delegates the problem of dealing with the unsafe behavior to wherever it is called (done by preceding the keyword 'unsafe' as in `unsafe fn foo() {}`). Another option would be dealing with the unsafe call directly by wrapping it in an *unsafe*{...}-block and verifying any output that results from it before continuing.

Again, just like with the references, *The Rust Programming Language* has a great chapter on unsafe Rust (Chapter 19.1) with a lot of examples and a list of 'superpowers' that are enabled when using it. [KN18, Ch. 19.1]

## Key Takeaways

This chapter briefly introduced Rust, its history and unique features. The main takeaways from this chapter that will be used in the rest of this thesis are first of all the **usage of cargo as Rust's toolchain** and the **compilation of Rust code into a static library**. Additionally, the **concept of unsafe code** that needs to be dealt with will be relied on later in the thesis.

## 3.2. Introduction to C++

This chapter serves as a general information about the programming language C++. It is meant for the reader to get a basic understanding of the history, a tool to help with compilation, and some essential features of the language.

### 3.2.1. Development and History

The development of C++ started back in 1979 when Bjarne Stroustrup began working on “C with Classes” which eventually evolved into the first versions of C++. The original goal, as stated by him, were to “provide Simula’s facilities for program organization together with C’s efficiency and flexibility for systems programming”[Str96]. A first commercial implementation was released in October of 1985, followed by C++ 2.0 in 1989, C++98, C++03, C++11, C++14, C++17, and C++20.

Given its age of now almost 40 years, it is remarkable that C++ is still as popular as it is, especially with people learning to code, where it was named by 31.11% of all survey participants in the 2023 Stack Overflow Developer Survey [Stab]. This could be due to the clear separation that C++ programs have, as they are separated into header and source files, making it suitable for new developers to learn the language’s structure. It is also convenient for various different usages, ranging from embedded programming to UI design (using established libraries like QT), making it a popular programming language in various different industries.

The standardization of C++ is done by an ISO working group known as JTC1/SC22/WG21.

### 3.2.2. Toolchain - CMake

Contrary to Rust and Cargo, there is no official toolchain provided by the developers and maintainers of C++. Instead, a few lightweight ones like CMake or NuGet (mainly supporting things like the compilation process) or bigger ones like Qt or Boost (coming with their own range of libraries) can be used, depending on the individual use case.

As it will be used later in the thesis, CMake will be explained in more detail here. According to their website, it “is the de-facto standard for building C++ code” and “a powerful, comprehensive solution for managing the software build process” [Cmaa].

Unlike Rust’s cargo, there is no way to initiate a full project and its necessary files using CMake alone. Even though there are tools like *The missing CMake project initializer* on GitHub or integrations into integrated development environments (IDEs) like CLion has, it is not a feature by CMake itself [fri23; Cmac].

### CMakeLists

During the development process, someone working with CMake will mainly be using the file *CMakeLists.txt*. It contains everything relevant to the compilation process, from variable definitions to build and link steps. A tutorial to set up a first project can be found on the official CMake Website [Cmaa, *CMake Tutorial* → *Step 1*].

The *CMakeLists.txt* file resulting from these steps can be seen in Listing 3.5. As the tutorial is more extensive than a minimum working example, it uses a few features such as copying and setting variables or including libraries that might not be required in every project. Nevertheless it is still a good starting point to get an understanding of the basic CMake capabilities.

A minimum working example resulting in an executable named *example* can be seen in Listing 3.6.

```
# Set the minimum required version of CMake to be 3.10
cmake_minimum_required(VERSION 3.10)

# Create a project named Tutorial
# and set the project version number as 1.0
project(Tutorial VERSION 1.0)

# Set the variable CMAKE_CXX_STANDARD to 11
# and the variable CMAKE_CXX_STANDARD_REQUIRED to True
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Use configure_file to configure and copy
# TutorialConfig.h.in to TutorialConfig.h
configure_file(TutorialConfig.h.in TutorialConfig.h)

# Add an executable called Tutorial to the project
add_executable(Tutorial tutorial.cxx)

# Use target_include_directories to include ${PROJECT_BINARY_DIR}
target_include_directories(Tutorial PUBLIC "${PROJECT_BINARY_DIR}")
```

Listing 3.5: Setup of the basic *CMakeLists.txt* file developed in the CMake tutorial

```
cmake_minimum_required(VERSION 3.15)

project(ExampleProject)

add_executable(example example.cpp)
```

Listing 3.6: Minimum working *CMakeLists.txt* file

#### Compilation into a Binary vs. a Library Target

Just like in the previous chapter, a compilation into the two different targets (binary and library) might be necessary and can be done using a simple command in the *CMakeLists* file. In Listing 3.5, this is achieved by the command *add_executable* in line 20 which tells the compiler to take *tutorial.cxx* and build an executable called 'Tutorial' from it.

The counterpart of this command is *add_library* which produces a library with the correct naming according to naming conventions. The full options, together with a distinction of normal, object, interface, imported, and alias libraries can be found in the CMake documentation [Cmaa, *cmake-commands* → *add_library*].

### 3.2.3. Unique Characteristics

C++ doesn't get its popularity from nowhere but rather from its various characteristics that make it a viable language in many situations. These are, contrary to Rust, not unique to the language, it's more the combination from all of them, alongside a long history and already existing frameworks, codebases, and standards, that form C++ into a complete and well-rounded language.

#### Structure

One of the big advantages of C++ that helps with both teaching the concept of programming to new developers as well as assists experienced ones is the clear structure of C++ programs. By splitting the code into header files (.h / .hpp) for declarations and interface definitions and source files (.cpp) for any implementations, a developer can keep a good overview over the entire program. This clear structure also helps with the development of new or revisions of old components as interfaces can be identified quickly, making a C++ codebase well suited for lasting a long time.

#### Versatility and Platform Independence

Another big advantage of C++ is the flexibility that a developer has. This is relating to both the type of program that is shipped as well as the platform that is used for coding. As mentioned before, C++ has a long established history which led to a lot of frameworks and libraries being developed. Combined with the freedom and responsibility to manage memory manually, C++ can be used for all kinds of applications, whether it's a kernel module that needs a lot of low-level memory operations or a GUI that is being programmed on a higher level.

Additionally, C++ can be developed for and on a multitude of different platforms and operating systems which make it a very viable option for a lot of companies from all kinds of different industry sectors.

## Key Takeaways

This past chapter introduced C++ together with its history and reasons for why it's still as popular as it is. There are not that many things that are necessary for a better understanding of the rest of the thesis, the main one being the basic concept of **CMake** together with **the *CMakeLists.txt* file**. It's also good to have some knowledge of **the separation into header and source files** as they will be used in the following chapters.

# 4. C++ and Rust Interoperability

This section will go over some of the basics similar to the examples mentioned in Chapter 2, namely how C++ and Rust can be integrated into each other and which steps are necessary for this to work.

For this, some small steps which are described in Section 4.1 Basics of Integration are going to be applied first to a C++ application with a Rust component, then the other way around.

## 4.1. Basics of Integration

In order to get two languages to work with each other, it is necessary to know about the basics of their foreign function interfaces (FFIs). In object-oriented programming, which is one of the programming paradigms of both Rust and C++ and the main application area for this thesis, the required basics consist of the definition of **structs and classes**, the **call of functions and methods**, as well as any additionally **defined types**. Furthermore, it's important to know how to **compile and link** the programs to work together.

The following two sections will go over the general way to integrate Rust into existing C++ applications and the other way around. They will go over the three big components, namely **Structs and Entities**, **Function Calls**, and **Method Calls**, together with some additional notes like **type safety** or **compilation & linking**. Type safety in this case is important for the projects to prevent any over- or underflow errors as well as compatibility issues as the underlying definitions and value ranges of the languages' native types don't always fit each other. An overview containing the native types of both C++ and Rust and their respective equivalents will be included in the last section of this chapter. It is a collection of various sources and is meant to serve as a lookup table to quickly find the required types.

Special sections about custom type definitions are left out in the following sections as they are simply defined in both languages and don't need any additional integration work.

## 4.2. Integrating Rust Code into C++ Applications

The first question, which is also one of the key tasks of this thesis, is the approach to integrate Rust components into C++ applications. This needs to be done while keeping in mind that interoperability with C++ is not officially supported by Rust's FFI. Thus, this integration works with the C application binary interface (ABI) instead. A short example of the integration of Rust into C is covered in Chapter 10.2 of *The Embedded Rust Book* which this section is loosely based on. [Theb]

### 4.2.1. Integration into the Code

#### Setup

In order to use a Rust definition like for example a function in C++, it needs to first be exposed to the FFI by making some annotations to the Rust code. These are going to be explained in the following paragraphs.

Afterwards, equivalent C++ declarations, the so-called **bindings** need to be written down. These are placed in a file, in this thesis a separate header file, and can be included and used by the rest of the project like a normal C++ entity with the only difference being that the call is executed by Rust code.

#### Structs and Entities

Structs and other entities, as well as custom type definitions, are a bit of a special case as they don't need to be exposed to the FFI directly, they are simply defined in both languages containing the same attributes. That way, they can be constructed and/or used in both languages. Unless the project setup requires it, they don't even need to be made *public* as they're both only used by their own code. The only annotation that needs to be made to the Rust struct is the attribute `#[repr(C)]` to fit C++'s type layout [Thec, Ch. 10.3][Rusf, Ch. 2.3].

An example can be seen in the following two Listings 4.1 and 4.2 where a Rust struct definition and its corresponding C++ syntax are shown.

```rust
#[repr(C)]
struct ExampleStruct {
    pub x: i32,
    pub c: char,
}
```

Listing 4.1: Rust struct definition

```cpp
#ifndef BINDINGS_H
#define BINDINGS_H

namespace RustComponentNS {

struct ExampleStruct {
    int x;
    char c;
};

} // namespace RustComponent

#endif
```

Listing 4.2: Corresponding C++ struct definition

This approach is the same for Enumerations, Unions, as well as type definitions. Even though it is not required, it is recommended to use a namespace like in this case `RustComponentNS` to be able to distinguish between members from different modules.

Lines 1, 2, and 13 of Listing 4.2 are also just regular C++ boilerplate code. As it's not relevant for the functionality of the code, it is being omitted in the Listings from this point on to focus on the essential parts.

### Function Calls

Rust functions also need to be preceded by certain keywords, this time `pub extern "C"` and the attribute `#[no_mangle]`. The former, just like with structs, declares the exposure to the FFI while the latter prompts the compiler to not mangle up function names so they can be identified and called externally, like in this case from C++. [Theb, Ch. 10.2]

An example for a simple function can be seen in Listing 4.3.

```rust
#[no_mangle]
pub extern "C" fn say_hello() -> i32 {
    println!("Hello from Rust");
    return 42;
}
```

Listing 4.3: Rust function definition

The corresponding C++ bindings are pretty straight-forward, the function declaration just needs to be marked with `extern "C"` to tell the compiler (this time the C++ side) to not mangle up the function name. A short example can be seen in Listing 4.4.

```
namespace RustComponentNS {

    extern "C" int say_hello();

} // namespace RustComponent
```

Listing 4.4: Corresponding C++ function binding

Using a regular call within a namespace, the function can then be used by simply including the bindings file, as seen in Listing 4.5. The Rust function will be executed, prompting in a "Hello from Rust" console output.

```
#include "example_bindings.h"
#include <iostream>

int main(int argc, char* argv[]) {
    int x = RustComponentNS::say_hello();
}
```

Listing 4.5: Usage of Rust function in C++ code

### Method Calls

Another use case would be the Rust struct having an associated function in an `Impl` block, which would be the equivalent of a method in C++. As an *Object.function()* notation is not supported by the FFI, the call first needs to be transformed into a syntax including the Object as a parameter, like *function(Object)*.

Taking advantage of the ability of every Rust *Impl*-function to be called standalone with `&self` as its first parameter, the Rust side does not need to be prepared any further than it would have to be with a regular function. [Rusd]

An example for such an implemented function with the required keywords can be seen in Listing 4.6. As the struct is changed during the function call, its argument needs to be mutable, thus resulting in `&mut self`.

```rust
#[repr(C)]
struct ExampleStruct {
    pub x: i32,
    pub c: char,
}

impl ExampleStruct {
    #[no_mangle]
    pub extern "C" fn set_values(&mut self, new_x: i32, new_c: char) -> () {
        self.x = new_x;
        self.c = new_c;
    }
}
```

Listing 4.6: Rust struct and implemented function definition (*example.rs*)

The binding file which can be seen in Listing 4.7 looks very similar to the one defining the simple function call. The only difference is in lines 11-13 where the `extern "C"` declaration is written as a block instead of preceding the function directly. This can and will be used later in this thesis when there is more than one external function to sum up repetitive declarations.

```cpp
namespace RustComponentNS {

struct ExampleStruct {
    int x;
    char c;
};

extern "C" {
    void set_values(const ExampleStruct* self, int new_x, char new_c);
} // extern "C"

} // namespace RustComponent
```

Listing 4.7: C++ binding file with a simple method

Applied in a simple example shown in Listing 4.8, the method has to be called using a *Namespace::function(Object)* syntax seen in line 13.

```cpp
#include "example_bindings.h"
#include <iostream>

int main(int argc, char* argv[]) {
    RustComponentNS::ExampleStruct ex_struct = {
        x: 1,
        c: 'a',
    };

    int new_x = 2;
    char new_c = 'b';

    RustComponentNS::set_values(&ex_struct, new_x, new_c);

    std::cout << "Struct after member function: " << ex_struct.x << " | " <<
        ex_struct.c << std::endl;
}
```

Listing 4.8: Usage of Rust function in C++ code

```
./ExampleCppProjectWithRust
Struct after member function: 2 | b
```

Listing 4.9: Output from C++ program with call to Rust method

#### C++ Wrapper Classes

Using methods by taking the object as a parameter as seen in Listing 4.8 is not a very common way to call them, they're usually in an *Object.function()* syntax. This can be replicated in C++ by implementing an additional wrapper class that has the following three tasks:

1. Hold the Rust object
2. Provide an access point for method calls in an *Object.function()* syntax
3. Provide access to the struct's/class's attributes as they can't be directly accessed anymore

In this case, the wrapper is fairly simple, as seen in Listing 4.10. It is able to hold the `ExampleStruct` with a local variable, has a constructor, in this case with the struct as a parameter, two getter functions for the struct's variables, and the method `set_values(int, char)` already seen in the previous example.

```cpp
#ifndef EXAMPLE_WRAPPER_H
#define EXAMPLE_WRAPPER_H

#include "example_bindings.h"

class RustComponentWrapper {
private:
    RustComponentNS::ExampleStruct ex_struct;

public:
    RustComponentWrapper(RustComponentNS::ExampleStruct ex_struct);

    int get_x();
    char get_c();
    RustComponentNS::ExampleStruct get_struct();

    void set_values(int new_x, char new_c);
};

#endif
```

Listing 4.10: Header file of a simple wrapper implementation (*example_wrapper.h*)

The implementation of the demonstrated wrapper is also very straight-forward. While the getter functions can simply access the variables and return them, the same *Namespace::function(Object)* syntax as introduced in the previous subsection can be used for relaying method calls. An example implementation can be seen in Listing 4.11. The function `get_c()` is not shown as it's pretty much the exact same function as `get_x()`.

```cpp
#include "example_wrapper.h"

RustComponentWrapper::RustComponentWrapper(RustComponentNS::ExampleStruct
    ex_struct) {
    this->ex_struct = ex_struct;
}

int RustComponentWrapper::get_x() {
    return this->ex_struct.x;
}

RustComponentNS::ExampleStruct RustComponentWrapper::get_struct() {
    return this->ex_struct;
}

void RustComponentWrapper::set_values(int new_x, char new_c) {
    RustComponentNS::set_values(&this->ex_struct, new_x, new_c);
}
```

Listing 4.11: Implementation of a simple wrapper class with a constructor, a method, and a getter function (*example_wrapper.cpp*)

Listing 4.12 shows the usage of such a wrapper class. After initializing the struct (lines 6-9) and handing it to a wrapper object (l. 11), any method calls can then be executed using the regular *Object.function()* syntax (l. 16).

One visible downside to a wrapper is the direct access to variables that is not possible anymore using the *Object.variable* syntax, which is why the getter functions were introduced (direct getter usage visible in line 19, access via the `get_struct()` function with *Object.variable*-syntax visible in line 20).

```
1  #include "example_bindings.h"
2  #include "example_wrapper.h"
3  #include <iostream>
4
5  int main(int argc, char* argv[]) {
6      RustComponentNS::ExampleStruct ex_struct = {
7          x: 1,
8          c: 'a',
9      };
10
11     RustComponentWrapper wrapper = RustComponentWrapper(ex_struct);
12
13     int new_x = 2;
14     char new_c = 'b';
15
16     wrapper.set_values(new_x, new_c);
17
18     std::cout << "Struct after member function: "
19               << wrapper.get_x() << " | "
20               << wrapper.get_struct().c << std::endl;
21 }
```

Listing 4.12: Usage of a wrapper class in C++ code

### Type Safety

When compiling the Rust part from Listing 4.6 using `cargo build`, the console output shows a warning regarding type safety as seen in Listing 4.13.

```
[ 25%] running cargo
warning: `extern` fn uses type `char`, which is not FFI-safe
 --> src/example.rs:9:64
  |
9 |     pub extern "C" fn set_values(&mut self, new_x: i32, new_c: char) ->
   () {
  |                                                                ^^^^ not
     FFI-safe
  |
  = note: `#[warn(improper_ctypes_definitions)]` on by default
  = help: consider using `u32` or `libc::wchar_t` instead
  = note: the `char` type has no C equivalent

warning: `rust_component` (lib) generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
```

Listing 4.13: Compilation output with type safety warnings

The warnings arise due to `char` not being type-safe for Rust's FFI and the C ABI. It can be prevented by defining the struct using the `c_char` type from the `std::ffi` library.
This type "provides utilities to handle data across non-Rust interfaces, like other programming languages and the underlying operating system" as stated in *Rust Standard Library FFI* [Std].

Internally, `c_char` is being replaced by `i8` or `u8`, depending on the underlying architecture, to match C's definition. More information on this will be included and explained in more detail in Section 4.4, more specifically 4.4.1.

The necessary changes to the Rust struct are shown in Listing 4.14, now not resulting in any compiler warnings.

```rust
#[repr(C)]
#[derive(Debug)]
pub struct ExampleStruct {
    pub x: i32,
    pub c: std::ffi::c_char,
}
```

Listing 4.14: Rust struct updated with `std::ffi::c_char` type

## 4.2.2. Compilation & Linking

### Rust Compilation

As already covered in Section 3.1, it is fairly simple to compile the Rust program into a library by including a new lib target in the *Cargo.toml*-file. When executing `cargo build` using Listing 4.15, this results in a *librust_component.a* file being placed in the *target/debug/* or *target/release/* directory, depending on the build type. The name of the library depends on the name of the package, which in this case is "rust_component" (see line 2 of Listing 4.15). The Rust compiler will take <<name>> as a variable and name the resulting file lib<<name>>.a. By doing this, it can later on be linked using -l <<name>>.

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

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

Listing 4.15: *Cargo.toml* of a simple Rust project, building a static library

### g++ Compiler

The C++ part can be compiled using a standard C++ compiler like **gcc** or **clang**. For this thesis, g++ is used on version 12.2.0 as it was the compiler used for multiple projects within MBDA which this thesis is being written in cooperation with.

After everything has been compiled, the static Rust library needs to be linked into the regular compilation process.

Using g++, the flags `-L` and `-l` can be used to specify the locations of additional libraries and then include them into the compilation process. Currently, the project has the file structure shown in Figure 4.1, only showing the files relevant for the compilation process.

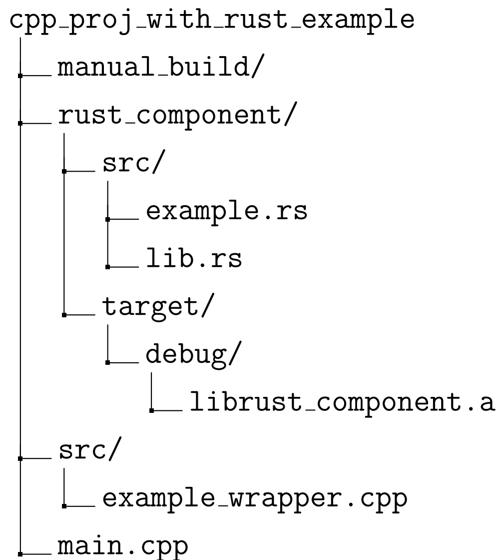


Figure 4.1.: File structure of a C++ project with a Rust component

The C++ project can now be compiled and linked using the following g++-command in *manual_build/* resulting in an executable named *ExampleCppProjectWithRust*.

```
g++ ./../main.cpp ./../src/example_wrapper.cpp
    -o ExampleCppProjectWithRust
    -L /ABSOLUTE_PROJECT_PATH/cpp_proj_with_rust_example/rust_component/
       target/debug/
    -l rust_component
```

#### CMake

The entire compilation and linking process can be done manually or automated using a tool like CMake (see Section 3.2.2 for more information). If the project is set up using CMake, the Rust library can easily be included like any other library using the `target_link_libraries()` call.

An example for a simple *CMakeLists.txt* file can be seen in Listing 4.16, resulting in the same *ExampleCppProjectWithRust* executable as when using the manual compilation. The file structure remains the same as seen in Figure 4.1.

```
cmake_minimum_required(VERSION 3.0)

project(ExampleCppProjectWithRust)

include_directories("src")

add_executable(ExampleCppProjectWithRust
    main.cpp
    src/example_wrapper.cpp)

target_link_libraries(ExampleCppProjectWithRust ${CMAKE_SOURCE_DIR}/rust_component/target/
    debug/librust_component.a)
```

Listing 4.16: *CMakeList.txt* of a simple C++project, including the Rust library in line 11

### Using CMakeRust

In order to not have to manually compile and then link the Rust library, a tool can be used to automate the full process like in this case **CMakeRust**, which is a collection of CMake files that can compile and link Rust Code. It can be easily included in a project by following five simple steps and adding some commands to the *CMakeLists.txt* file.

1. Add a simple *CMakeLists.txt* in the Rust project's directory (*rust_component/*) with the following content:

```
cargo_build(NAME rust_project_name)
```

2. Add the CMakeRust files to the project directory (in this case they are saved in the directory *cmakerust/*) and add CMakeRust to the CMake Module Path:

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/
    cmakerust/")
```

   This directory is called *cmake/* by default. Saving the files in a different directory (in this case *cmakerust/*) is not a problem, as long as the correct path is set in the *CMakeRust-Compiler.cmake.in* file:

```
configure_file(${CMAKE_SOURCE_DIR}/cmakerust/CMakeRustCompiler.cmake.in
    [...])
```

3. Enable Rust support and add the CMake files:

```
enable_language(Rust)
include(CMakeCargo)
```

4. Add the Rust project's directory using `add_subdirectory(<<rust_project_name>>)`

5. Link the Rust library using `target_link_libraries(<<rust_project_name>>)`

Steps 1-4 are used to compile the Rust library, whereas step five then includes the Rust library into the project, just like the previous section that used the manually compiled library. The full *CMakeLists.txt* file can be seen in Listing 4.17.

```cmake
cmake_minimum_required(VERSION 3.0)

project(ExampleCppProjectWithRust)

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmakerust/") # Step 2

enable_language(Rust) # Step 3
include(CMakeCargo) # Step 3

include_directories("include")
include_directories("src")
add_subdirectory(rust_component) # Step 4

add_executable(ExampleCppProjectWithRust
    main.cpp
    src/example_wrapper.cpp)

target_link_libraries(ExampleCppProjectWithRust rust_component) # Step 5
```

Listing 4.17: *CMakeLists.txt* using CMakeRust to compile the Rust code

The full CMakeRust files used for this thesis will be added to the appendix, they can also be found on the creator's GitHub repository under `https://github.com/Devolutions/CMakeRust`. [Dev23, Commit 9a4a7a1]

For the rest of this thesis, CMakeRust will be used to compile and include any Rust components into C++ projects.

## Key Takeaways

As this section laid the ground work for the step-by-step guide and example integration later on in this thesis, there are quite a few things that should be known after reading this thesis. Firstly, it should be known which **additions** have to be made **to Rust declarations** to expose them to the FFI and enable them to be called externally as well as how to mark C++ components to signal the compiler that they're going to be defined somewhere else. Additionally, it should be known how to **define, call and use Rust structs, functions, and methods** in C++ and how to **include the component into the compilation process** using **CMakeRust**.

Finally, the terms "**binding**", "**binding file**", and "**wrapper class**" should be known as they will only be briefly explained later in this thesis if at all.

## 4.3. Integrating C++ Code into Rust Applications

The other way around, which is the integration of C++ components into a Rust project, is not necessarily the focus of this thesis but it is still included for completion reasons. The procedure will be the same as the first, though a special focus is laid on classes which are included in the part with method calls. This chapter is again loosely based on *The Embedded Rust Book*, this time Chapter 10.1.

### 4.3.1. Integration into the Code

#### Setup

The integration of C++ into Rust applications is done by implementing a binding file in Rust that acts in the same function as the header file in Section 4.2 by providing access to all necessary entities and functions.

In C++, there is no differentiation between structs or functions when exposing them to the FFI. For any element that needs to be usable externally, their definitions have to be preceded by the keywords `extern "C"`. The Rust binding file then contains an `extern "C"`-block with the equivalent Rust function headers.

#### Structs and Entities

Just like in Section 4.2, structs and entities simply need to be declared in both languages independently. A short example can be seen below with the declaration in *example.h* (Listing 4.18) and the respective Rust implementation in *example_bindings.rs* (Listing 4.19).

```
4 struct ExampleStruct {
5     int x;
6     char c;
7 };
```

Listing 4.18: Struct declaration in C++ header file (*example.h)*

```
1 #[repr(C)]
2 pub struct ExampleStruct {
3     pub x: i32,
4     pub c: char,
5 }
```

Listing 4.19: Rust bindings for struct example (*example_bindings.rs)*

#### Function Calls

Function calls work analogously, adding `extern "C"` to the declaration, here in the header file *example.h*, seen in line 3 of Listing 4.20. It is then defined in *example.cpp* (Listing 4.21) whereas this example even includes *iostream* as an additional import to show that imported libraries get added during the compilation.

```
4 #include <iostream>
5
6 extern "C" int say_hello();
```

Listing 4.20: Function declaration in C++ header file (*example.h)*

```
1 #include "example.h"
2
3 int say_hello() {
4     std::cout << "Hello from C++" << std::endl;
5     return 42;
6 }
```

Listing 4.21: Function definition in C++ source file (*example.cpp)*

```
1 extern "C" {
2     pub fn say_hello() -> i32;
3 }
```

Listing 4.22: Rust bindings for function example (*example_bindings.rs)*

As the Rust compiler is not able to check the C++ function for memory safety, it needs to be marked as `unsafe` when used in *main.rs*, as seen in lines 5-7 of Listing 4.23.

For a little more information on Rust's *unsafe* functionality, refer to Section 3.1.3 in this thesis or to Chapter 19.1 in *The Rust Programming Language*. [KN18]

```
1 mod example_bindings;
2 use crate::example_bindings::say_hello;
3
4 fn main() {
5     unsafe {
6         println!("C++ function return: {:?}", say_hello());
7     }
8 }
```

Listing 4.23: Usage of C++ Function in Rust

The program output includes both console outputs, also showing the functionality of the *iostream*-import without explicitly including it.

```
$ ./target/debug/rust_proj_with_cpp_example
Hello from C++
C++ function return: 42
```

### Classes, Method Calls, and Wrappers

Classes and their methods in C++ are equivalent to Rust structs and implemented functions. As C++ methods don't have the ability to be called standalone like Rust did (see 4.2.1 Method Calls), they have to be implemented with a wrapper that transforms the *Object.function()* call to a *function(Object)* syntax, as visible in Listings 4.24 and 4.25.

```
4  extern "C" class ExampleClass
5  {
6  private:
7      int x;
8      char c;
9
10 public:
11     ExampleClass(int x, char c);
12     void set_values(int new_x, char new_c);
13 };
14
15 extern "C" void set_values_cpp_wrapper(ExampleClass *object, int new_x, char
       new_c);
```

Listing 4.24: Class and method declaration in C++ header file (*example.h)*

```
1 #include "example.h"
2
3 ExampleClass::ExampleClass(int x, char c) { ... }
4
5 void ExampleClass::set_values(int new_x, char new_c) { ... }
6
7 void set_values_cpp_wrapper(ExampleClass *object, int new_x, char new_c) {
8     object->set_values(new_x, new_c);
9 }
```

Listing 4.25: Class and method definition in C++ source file (*example.cpp)*

This function can then be used in Rust, as seen in the binding file in Listing 4.26 and the subsequent usage in Listing 4.27.

```
1 #[repr(C)]
2 pub struct ExampleClass {
3     pub x: i32,
4     pub c: char,
5 }
6
7 extern "C" {
8     pub fn set_values_cpp_wrapper(obj: &mut ExampleClass, new_x: i32, new_c:
          char) -> ();
9 }
```

Listing 4.26: Rust bindings for class and method example (*example_bindings.rs)*

```rust
mod example_bindings;
use crate::example_bindings::{set_values_cpp_wrapper, ExampleClass};

fn main() {
    let mut ex_object = ExampleClass { x: 1, c: 'a' };
    let new_x = 2;
    let new_c = 'b';
    unsafe {
        set_values_cpp_wrapper(&mut ex_object, new_x, new_c);
    }
}
```

Listing 4.27: Usage of C++ Class and its method in Rust, using the *function(Object)*-syntax

Additionally, a wrapper struct can be implemented in Rust that changes the call to an *Object.function()*-syntax, equivalently to 4.2.1.

One possible implementation of said wrapper is shown in lines 8-10 in Listing 4.28 where it is directly implemented in the bindings file. As the call to the external function `set_values_cpp_wrapper()` can't be verified by the Rust compiler, it has to be marked as *unsafe*. Like mentioned in Section 3.1.3, *unsafe* functions can be dealt in two different ways. The option that is used in this case is simply marking the function as such to delegate the problem to where it's called. The second option which is directly dealing with the unsafe call (by wrapping it in an *unsafe* control block) can for example be seen in lines 8-10 of Listing 4.27.

The method can then be accessed by using the dot notation on the Rust struct, as seen in line 10 of Listing 4.29.

```rust
#[repr(C)]
pub struct ExampleClass {
    pub x: i32,
    pub c: char,
}

impl ExampleClass {
    pub unsafe fn set_values_rust_wrapper(&mut self, new_x: i32, new_c: char
        ) {
        set_values_cpp_wrapper(self, new_x, new_c);
    }
}

extern "C" {
    pub fn set_values_cpp_wrapper(obj: &mut ExampleClass, new_x: i32, new_c:
        char) -> ();
}
```

Listing 4.28: Rust bindings and implemented wrapper function for C++ method (*example_bindings.rs*)

```rust
mod example_bindings;
use crate::example_bindings::ExampleClass;

fn main() {
    let mut ex_object = ExampleClass { x: 1, c: 'a' };
    let new_x = 2;
    let new_c = 'b';

    unsafe {
        ex_object.set_values_rust_wrapper(new_x, new_c);
    }
    println!(
        "Struct after example_function: {} | {}", ex_object.x, ex_object.c
    );
}
```

Listing 4.29: Usage of dot notation to access a C++ method in Rust

The output in both cases, with or without wrapper, is shown below.

```
$ ./target/debug/rust_proj_with_cpp_example
Struct after example_function: 2 | b
```

#### Type Safety

When compiling this small project, the output from cargo raises the following two warnings seen in Listing 4.30

```
warning: `extern` block uses type `char`, which is not FFI-safe
  --> src/example_bindings.rs:14:40
   |
14 |     pub fn set_values_cpp_wrapper(obj: &mut ExampleClass, new_x: i32,
   new_c: char) -> ();
   |                                        ^^^^^^^^^^^^^^^^^ not FFI-safe
   |
   = note: `#[warn(improper_ctypes)]` on by default
   = help: consider using `u32` or `libc::wchar_t` instead
   = note: the `char` type has no C equivalent
```

Listing 4.30: Compilation Output raising FFI-safety warnings

This is the same problem already introduced in Section 4.2.1 and can be solved by using the `std::ffi::c_char` type. The updated binding file can be seen in Listing 4.31, now without compiler warnings.

```
#[repr(C)]
#[derive(Debug)]
pub struct ExampleClass {
    pub x: i32,
    pub c: std::ffi::c_char,
}

impl ExampleClass {
    pub unsafe fn set_values_rust_wrapper(&mut self, new_x: i32, new_c: std
        ::ffi::c_char) {
        set_values_cpp_wrapper(self, new_x, new_c);
    }
}

extern "C" {
    pub fn set_values_cpp_wrapper(obj: &mut ExampleClass, new_x: i32, new_c:
         std::ffi::c_char) -> ();
}
```

Listing 4.31: Updated *example_bindings.rs* using the `std::ffi::c_char` type

The only thing that is left to be changed is the calls in *main.rs*, as the parameter has to be cast from a char into a `std::ffi::c_char`. To print the variable as a character, it needs to be converted as such.

Thus, the beginning of the updated `main()`-function is changed as seen in Listing 4.32.

```
fn main() {
    let mut ex_object = ExampleClass { x: 1, c: 'a' as std::ffi::c_char };
    let new_x = 2;
    let new_c = 'b' as std::ffi::c_char;
    [...]
    println!(
        "Struct before example_function: {} | {}",
        ex_object.x, ex_object.c as u8 as char
    );
}
```

Listing 4.32: Updated main function using type-safe datatypes

## 4.3.2. Compilation & Linking

### C++ Compilation

For compiling the C++ code, g++ is used again. The second step, linking, is done using the `ar` command, which is used to create archives, in this case, a *libexample.a* file.

```
g++ -c cpp_code/example.cpp -o manual_compile/example.o
ar rcs manual_compile/libexample.a manual_compile/example.o
```

Using these commands in the rust_proj_with_cpp_example/ directory results in the file structure shown in Figure 4.2.

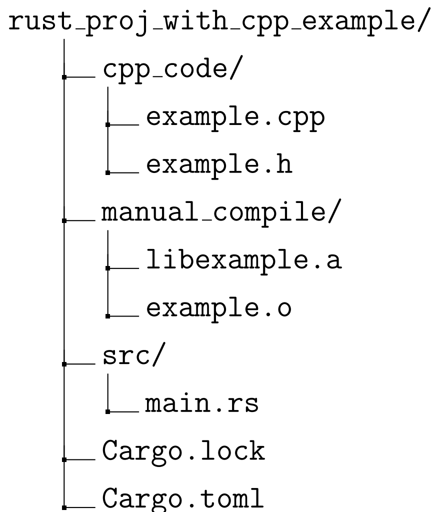


Figure 4.2.: File structure of a Rust project with a C++ component

### Cargo and rustc Compilation

A Rust program is usually compiled using the cargo-toolchain (see Subsection 3.1.2) which composes a call to the rustc-compiler with the correct flags being set. This can also be done manually but involves extensive knowledge of the compiler and its options.

For reference, the command in Listing 4.33 is being executed when calling `cargo build` on this project before any C++ code was added. Any `cargo` command can be expanded like this by adding the `--verbose` flag.

```
rustc   --crate-name rust_proj_with_cpp_example
        --edition=2021
        src/main.rs
        --error-format=json
        --json=diagnostic-rendered-ansi,artifacts,future-incompat
        --crate-type bin
        --emit=dep-info,link
        -C embed-bitcode=no
        -C debuginfo=2
        -C metadata=caf733afa82fb3b0
        -C extra-filename=-caf733afa82fb3b0
        --out-dir $PROJECT_WORKSPACE/target/debug/deps
        -C incremental=$PROJECT_WORKSPACE/target/debug/incremental
        -L dependency=$PROJECT_WORKSPACE/target/debug/deps
        --extern libc=$PROJECT_WORKSPACE/target/debug/deps/liblibc-
           b9cd94e63f265a51.rlib
```

Listing 4.33: Expanded rustc compilation command

The variable `$PROJECT_WORKSPACE` is hereby a replacement for the absolute location of the Rust project called 'rust_proj_with_cpp_example'.

#### Manual Linking

If libraries like the *libexample.a* archive need to be included, they can be linked using the `-l` and `-L` flags, just like the dependency directory in line 14 of Listing 4.33 [Thed, Ch. 2]. If one does not want to dive into the compiler flags, the libraries can also be linked using the `rustc-link-lib` and `rustc-link-search` options in the *Cargo.toml* file.

A *Cargo.toml* file for a project including the example library can be seen in Listing 4.34.

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

[target]
rustc-link-search = ["$PROJECT_WORKSPACE/rust_proj_with_cpp_example/
    manual_compile"]
rustc-link-lib = ["example"]
```

Listing 4.34: Cargo.toml file including the example library

#### Linking Using a build.rs File

In order to keep an overview over more complicated build processes, Rust gives the possibility to write a script in a file called *build.rs* in the *src/* directory of the project. Including a library in this script works by first specifying the location of the library (equivalent to the `-L` flag) by adding `cargo:rustc-link-search=[KIND=]PATH` and then only naming the library name using the `cargo:rustc-link-lib=LIB` instruction as the equivalent for the `-l` flag. [Thea, Ch. 3.8]

Listing 4.35 shows a *build.rs* file located in the *rust_proj_with_cpp_example/* directory. In there, line 2 includes the directory where the C++ archive file (*libexample.a)* is located, while line 3 includes it. To use standard library functions and types as well as native libraries, the C++ standard *stdc++* library needs to be included as well.

```
fn main() {
    println!("cargo:rustc-link-search=$PROJECT_WORKSPACE/
        rust_proj_with_cpp_example/manual_compile");
    println!("cargo:rustc-link-lib=example");
    println!("cargo:rustc-link-lib=stdc++");
}
```

Listing 4.35: build.rs file to include a *libexample.a* archive and the C++ standard library

It results in the same `rustc` command as shown in Listing 4.33, including the new linking commands, as seen in lines 7 and 8 of Listing 4.36.

```
rustc   --crate-name rust_proj_with_cpp_example
        --edition=2021
        src/main.rs
        [...]
        --extern libc=$PROJECT_WORKSPACE/target/debug/deps/liblibc-b9cd94e63f265a51.rlib
        -L $PROJECT_WORKSPACE/manual_compile
        -l example
        -l stdc++
```

Listing 4.36: Compilation command including the C++ archive and standard library

Something to be noted is the naming of the C++ library being *libexample.a* even though it is included with `rustc-link-lib=example`. This is the same convention that was used in Section 4.2.2 - naming a file `lib<<name>>.a` and including it using only `<<name>>`.

If the name is not in this specific format, it can be included using `-l:filename.a`. [Thed, Ch. 2]

The C++ files can also be compiled within the build script with g++ by using the `Command::new()` object included in `std::process::Command`. An example for that can be found in Chapter 3.8.1 in *The Cargo Book*. [Thea, Ch. 3.8.1]
The book also shows some more examples for things like conditional instructions as well as options for recompiling when specific files were changed. Additionally, examples for conditional instructions and helpful options like a recompilation when certain files were changed, are presented and walked through.

### Using the cc Crate

The compilation and linking can also be done automatically using the **cc crate** which "provide[s] the utility functions necessary to compile C code into a static archive which is then linked into a Rust crate"[Ccc].

It utilizes the *build.rs* script by including an additional command which can be seen in Listing 4.37. Hereby, the `compile()` command also automatically includes the resulting library.

```
fn main() {
    cc::Build::new()
        .file("cpp_code/example.cpp")
        .cpp(true)
        .compile("example");
}
```

Listing 4.37: build.rs using a call by the cc crate

Thanks to the `cpp(true)` call, the C++ standard library is automatically included. This can be confirmed by looking at the last three lines of the `rustc`-command that is executed internally (see Listing 4.38). They are pretty much the same as in the previous commands shown in Listings 4.33 and 4.36 with only the library location being a generated directory name.

```
rustc   --crate-name rust_proj_with_cpp_example
        --edition=2021
        src/main.rs
        ...
        --extern libc=$PROJECT_WORKSPACE/target/debug/deps/liblibc-
            b9cd94e63f265a51.rlib
        -L native=$PROJECT_WORKSPACE/target/debug/build/
            rust_proj_with_cpp_example-caeee79d6b60fc54/out
        -l static=example
        -l stdc++
```

Listing 4.38: Compilation command with cc

### Using the cmake Crate

Since many larger C++ projects are set up using the CMake infrastructure, it could be helpful to be able to directly include this process which is exactly what the **cmake crate** can be used for. It offers additional utilities to compile the CMake component and directly integrate it into the Rust project. [Cmab]

Listing 4.39 shows an example *build.rs* file that was taken from the documentation, showing the general functionality.
As it's not relevant for the rest of this thesis, a more detailed description of these crates (**cc** and **cmake**) will not be included but can be found in [Ccc] and [Cmab].

```
use cmake;

// Builds the project in the directory located in `libfoo`, installing it
// into $OUT_DIR
let dst = cmake::build("libfoo");

println!("cargo:rustc-link-search=native={}", dst.display());
println!("cargo:rustc-link-lib=static=foo");
```

Listing 4.39: build.rs using the cmake crate

### Key Takeaways

As this section described the integration of C++ components in Rust projects which is not essential to the main thesis topic itself, there are no takeaways relevant for the rest of this thesis.

## 4.4. Equivalents of C++ and Rust

In order to correctly define interfaces, it's important to know the equivalent types of C++ and Rust. This chapter is meant as a lookup table to get the corresponding types and read up some additional information that might be necessary when working with the FFI.

### 4.4.1. Fundamental Types

#### Integer Types

Following the definitions from *ISO 14882:2020 Programming languages - C++* and *The Rust Programming Language*, the following table can be derived, listing the respective types with the same sizes and boundaries [ISO20, Ch. §6.8.2] [KN18, Ch. 3.2]:

| C++Type | Minimum Bit Width | Sign | Rust Type |
|---|---|---|---|
| char | 8 | signed<br>unsigned | i8<br>u8 |
| short | 16 | signed<br>unsigned | i16<br>u16 |
| int | 16 | signed<br>unsigned | i16<br>u16 |
| long | 32 | signed<br>unsigned | i32<br>u32 |
| long long | 64 | signed<br>unsigned | i64<br>u64 |

Table 4.1.: Integer type comparison between Rust and C++

Within Rust, the types are very straightforward, as both the sign as well as the size can be directly derived from the type name, for example `u16` being an unsigned 16-bit Integer. A problem that can arise with C++ is that the actual size of the types depend on the target system, the bit widths in Table 4.1 only state the minimum guaranteed size per [ISO20, Ch. §6.8.2].

In order to circumvent the uncertainty, the C++ standard defines **fixed width integer** types which always have the same size regardless of the underlying system or architecture [Cpp, Fixed width integer types]. It is highly advisable to use these in systems with high safety requirements to prevent unintended behavior, with some standards like AUTOSAR even making their usage mandatory [AUT19]. Table 4.2 shows these fixed width integer types together with their Rust counterparts.

| Bit Width | Sign | cpp Type | Rust Type |
|---|---|---|---|
| 8 | signed<br>unsigned | int8_t<br>uint8_t | i8<br>u8 |
| 16 | signed<br>unsigned | int16_t<br>uint16_t | i16<br>u16 |
| 32 | signed<br>unsigned | int32_t<br>uint32_t | i32<br>u32 |
| 64 | signed<br>unsigned | int64_t<br>uint64_t | i64<br>u64 |

Table 4.2.: Fixed Width Integer Type comparison between Rust and C++

The `char` type in C++ (see row 1 in Table 4.1) can be misunderstood as only holding and storing what is commonly known as a 'character' while being listed as an Integer type. According to the *ISO 14882:2020 Programming languages - C++*, it is used for "represent[ing] distinct codes for all members of the implementation's basic character set" and can be used to simply store an Integer. [ISO20, Ch. §6.8.2 par. 7] The representation of characters is covered at a later point in this section.

### Floating Point Types

Just like with integers, a table for floating point values can be formed, even though it's a bit simpler due to them always being signed.

| C++Type | Minimum Bit Width | Precision | Rust Type |
|---|---|---|---|
| float | 32 | single | f32 |
| double | 64 | double | f64 |
| long double | 128 | extended | |

Table 4.3.: Floating Point Type comparison between Rust and C++

According to the minimum and maximum values and the given precision, the types `f32` and `f64` can be derived as Rust's `float` and `double` equivalents. Only C++'s `long double` does not have a type within Rust that could be used for storing these values. [F32; F64; Cpp]

Even though these values may not be necessary in every program, this is a fairly clear limitation of a value or variable that can't be represented in Rust.

### Character

Characters in Rust are represented by the type `char` "which is any 'Unicode code point' other than a surrogate code point" [Cha].

In C++ on the other hand, a `char` is simply an integer that can be interpreted as a character. Its size is dependent on the architecture of the underlying operation system, sometimes translating to `i8` or `u8`.

To prevent any problems using the foreign function interface and C++'s variable sized types, Rust has the `std::ffi` library which offers some support, like an `std::ffi::c_char` type [Std]. Depending on the architecture, it represents the character as either an `i8` or a `u8`, taking out the possibility of over- or underflows happening when sending characters using the FFI.

#### Boolean

Boolean values, which can only take on the values `true` and `false`, are the same in both C++ and Rust, being called `bool`.

### 4.4.2. Compound Types

#### Enumerations and Unions

Enumerations as well as Unions exist in both Rust and C++. They can be used equivalently after being defined correctly.

#### Structs and Classes

Structs exist in both languages and can also be treated as equivalents. A special note has to be made when talking about C++ classes as officially, Rust doesn't have them. On the other hand, a Rust struct can be equipped with methods that can then be called using the usual dot notation that is popular in object-oriented programming. These methods are regular functions implemented in a code section opened by
`impl <StructName> { ... }`.

An example on the usage of C++ methods and implemented Rust functions via the FFI can be found in Sections 4.2.1 and 4.3.1 respectively.

### 4.4.3. Custom types

As long as custom types are declared in both languages equally, they can be used over the FFI. They have to be declared using the correct counterpart though, to prevent any over- or underflows when using these types.

#### More Complex Types - Using the cxx Library

Any other types, especially those requiring a pointer like for example Arrays, Vectors, or Strings are a lot more complicated to transfer via the FFI due to their varying sizes and additionally implemented functions that might not always be compatible with the FFI. To still be able to handle these types, a library called **cxx** can be used. It is able to generate C bindings that serve as a bridge between the C++ and Rust bindings and support more complex types with the help of internal implementations. [Cxxa; Cxxb]

As using this crate is not within the scope of this thesis, it will not be introduced in more detail or used in the example project later on. Thus, the project will work with integers, structs, and custom defined types and not include byte arrays that could serve as a potential payload in messages.

### 4.4.4. Full Table

Table 4.4 contains a compact overview over the simple types and structures and their equivalents in C++ and Rust.

| | **C++ type** | **Rust type** | **Comment** |
|---|---|---|---|
| Integer | signed char | i8 | |
| | unsigned char | u8 | |
| | signed short | i16 | |
| | unsigned short | u16 | |
| | signed int | i16 | |
| | unsigned int | u16 | |
| | signed long | i32 | |
| | unsigned long | u32 | |
| | signed long long | i64 | |
| | unsigned long long | u64 | |
| Fixed Width Integer | int8_t | i8 | |
| | uint8_t | u8 | |
| | int16_t | i16 | |
| | uint16_t | u16 | |
| | int32_t | i32 | |
| | uint32_t | u32 | |
| | int64_t | i64 | |
| | uint64_t | u64 | |
| Float | float | f32 | |
| | double | f64 | |
| | long double | | No Rust equivalent |
| Other | char | char | |
| | char | std::ffi:c_char | FFI-safe type |
| | bool | bool | |
| | struct | struct | |
| | class with methods | struct with impl | |

Table 4.4.: Table with equivalent datatypes of C++ and Rust

# 5. The Way from C++ to Rust

Since a full integration from scratch and only with the theoretical background set in the previous chapters can be overwhelming for a developer who is new to this topic, this next chapter will first list a few approaches to how such an integration could be done.
Afterwards, the process is going to be described in more detail, resulting in a guide that can be followed by a developer who wants to integrate a Rust component into their C++ application. As guides are more often than not accompanied by examples which are important to understanding the actual practical process behind it, it is followed by a small project where exactly that integration is going to be done following the guide.

## 5.1. Approaches to Integration

This chapter will look at a first approach to integrating a Rust component into an existing C++ project. It will contain two levels of integration:

1. Only the Rust component is implemented and the C++-code has to be adapted accordingly to fit function calls etc.
2. The Rust component is implemented together with a C++ wrapper/translator file that has the same endpoints as the original component and passes any calls on to the Rust component. This will ensure that the existing C++ code will only have minimal changes in case a component is being replaced. The downsides include additional work at the moment of implementation as well as potentially more effort when introducing more Rust components.

#### Adding Only the Rust Component

The first option, which consists of only adding in a new component and then implementing any future function calls to fit the Rust notation (see Section 4.2.1 Function Calls) is more suitable for smaller projects. A downside to using this approach is the constant switching between calling conventions and a possibly difficult integration into an existing architecture. Thus, for bigger projects, the effort of writing C++ wrapper classes can be worth it to ensure a better fit into the current project.

### Using a C++ Wrapper Class

The second level consists of implementing a wrapper in C++ which enables all calls to this component to be made in a standard *Object.function()* syntax. The basics of this additional class were laid in Section 4.2.1, more specifically the subsections "Method Calls" and "C++ Wrapper Classes". Together with any necessary changes to types and casts that can also be made in these wrapper functions, this leaves the C++-code as unchanged as possible when a component is being replaced. For completely new components, it's often easier to fit C++ classes into an existing architecture rather than FFI calls all over the place.

While this approach may require a bit more work in the first place, it will simplify quite a few things in the later usage, like calls in general or any further integration, for example in an inheritance hierarchy. Using naming conventions, like `PreviousClassNameWrapper`, this approach can make it clear where the interfaces are located and which C++-equivalent they are supposed to represent.

Due to these reasons, the wrapper approach is going to be used in this chapter.

If the C++-code is too extensive and changing class names for example from `ClassName` to `ClassNameWrapper` is too much work, the wrapper class could of course be named like the original class was. The big advantage to this is that the C++ calls to the new component don't have to be changed at all, it seems to the rest of the project as if the old code was still active. A disadvantage to this could be the now unclear difference between the old component and the wrapper calling Rust. In a project with strict naming conventions, this could become a problem and should be well thought out.

Another disadvantage with the wrapper could be the direct access of variables using the *Object.variable* notation. As the object isn't directly accessed via the FFI, it's difficult holding the correct variable values at all times. This problem could easily be avoided by using setter and getter functions which is already the standard of most coding practices. This would mean a bit more work changing any occurrences of *Object.variable* into instances of *Object.getVariable()* but these could then easily be used via the FFI.

## 5.2. Step by Step Guide

Taking the previous chapters into account, this section describes the steps required to introduce a Rust component, in this case the equivalent of a C++ class, into an existing C++ application. It is meant to serve as a complete guide and will be used in the next section to introduce a Rust component into a project using a simple publish-subscribe pattern. The step describing and explaining the actual implementation of the Rust component is hereby left out, as this is not the focus of this thesis though the code will be included.

1. **Identify**

    a) The Rust component's tasks (clear cut definition of what it's supposed to be doing)

    b) Any required integration into the current structure (for example inheritance) and the resulting structure requirements for the C++ wrapper classes

    c) Any custom types that both sides require

    d) The Rust component's interface (which member and standalone functions are required by / need to be available to the rest of the project?)

    e) The C++ wrapper classes' attributes and functions

2. **Implement**

    a) The C++ header files containing the required C++ bindings for the Rust interface (as identified in step 1d)
    *Refer to Section 4.2.1 and Listing 4.2 for more information.*

    b) The C++ wrapper to fit the current projects' structure (as identified in steps 1b, 1d, and 1e)
    Note: Any typecasts necessary to fit Rust's FFI should be implemented in this wrapper class
    *Refer to Section 4.2.1 for more information on the wrapper implementation and to Listings 4.10 and 4.11 for an example.*
    *Refer to Section 4.4 for more information on necessary type conversions.*

    c) Any custom type declarations (both in Rust and in C++, identified in step 1c)

    d) The Rust component

3. **Include**

    a) The new component in the current project
    Hereby, there are two options, depending on whether the component is ...

    - ... the replacement of an old component → Change the already existing calls
    - ... a totally new component → Use the wrapper class like a normal C++ class

    b) The new component in the existing compilation process.
    *Refer to Section 4.2.2 for more information on the different ways on how this can be done.*

## 5.3. Full Process Demonstrated at Example Project

This chapter will demonstrate the full process of a component replacement in a C++ project. Compared to the previous chapters which dealt with single parts, this will accompany the entire process and is a 'walkthrough' of the guide in Section 5.2 that is the artifact of this thesis. It is applying the previous chapters, especially Sections 4.2 and 5.1 to the example project.

The example project is an implementation of a publish-subscribe pattern. It is supposed to simulate a simple communication component that can be used in a project to send messages. As already described in the end of Section 4.4, using more complex data types is out of the scope for this thesis. Thus, the *Message*-struct for this project has a simple ID and a length, without an actual payload which could be realized using a byte array for example.

The current project structure is shown in Figure 5.1.

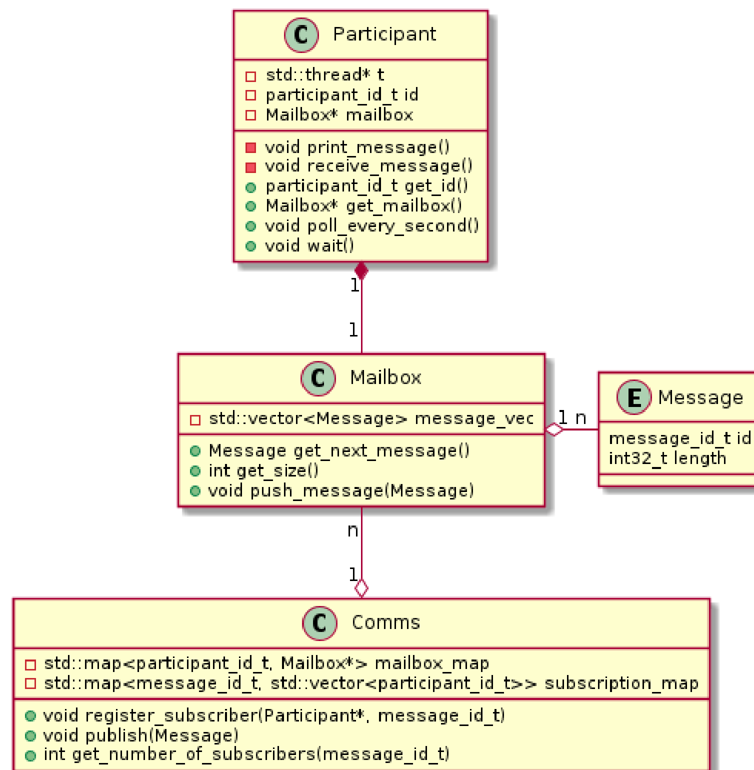


Figure 5.1.: Class diagram of the starting situation

Any custom types are stored in a file called *types.h* which can be included and used by all the other files. The file structure can be seen in the directory tree shown in Figure 5.3.

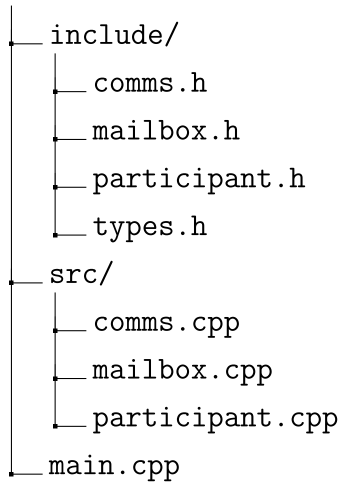


Figure 5.2.: Directory tree showing the initial project setup

The task is now to include a Rust component that acts like one of the current C++ subscribers, namely the *Participant*. It was also planned to generalize the entire structure in a way that the two components are not seen separately but that both derive from an abstract class that can be used in any further development.

Originally, it was implemented that only the *Participant* was going to get a Rust equivalent. Due to the structure and for demonstration purposes that multiple Rust parts can fully communicate with each other without additional interfaces, *Mailbox* was also given a Rust version and thus an abstract class. Therefore, in preparation to implementing the Rust component, the C++ application is adapted to use *BaseMailbox* and *BaseParticipant* as base classes, resulting in the class diagram seen in Figure 5.3.

The resulting file structure can be seen in Figure 5.4 with the inclusion of the abstract classes.

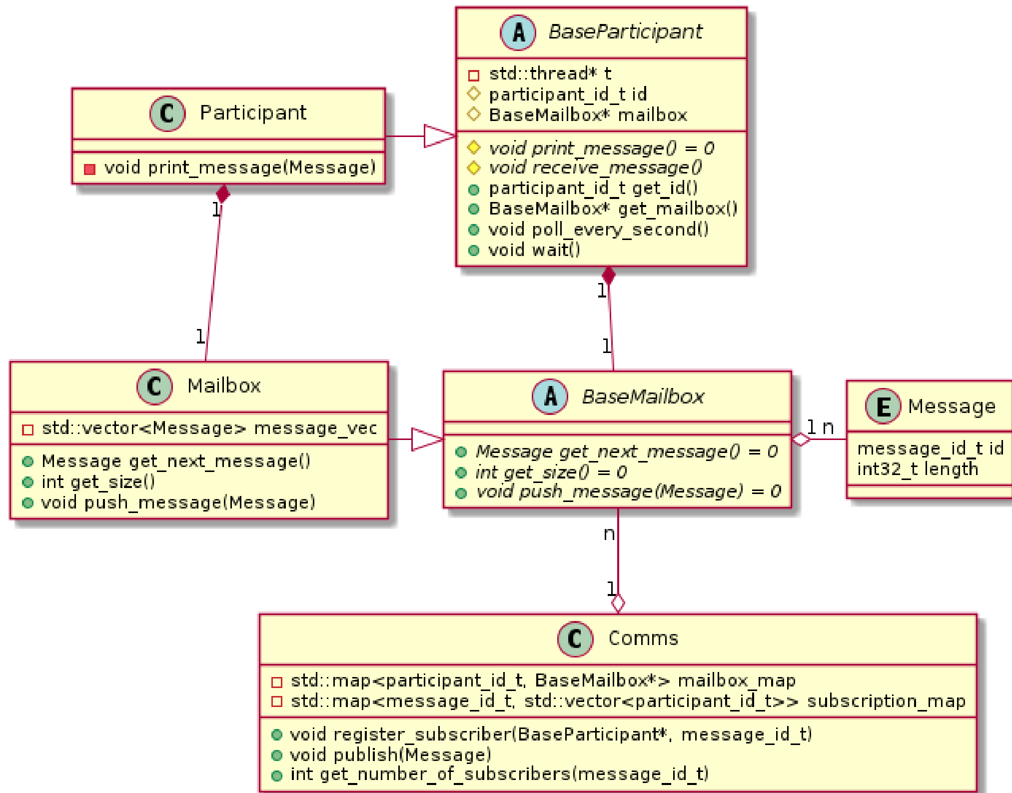


Figure 5.3.: Class diagram with abstract classes

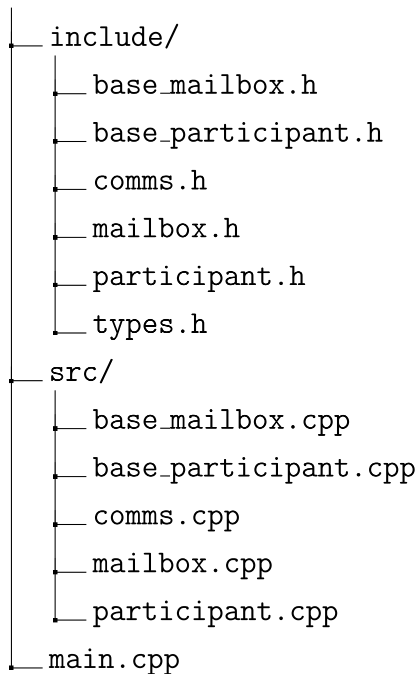


Figure 5.4.: Directory tree showing the project setup with abstract classes

Based on this situation, the steps introduced in the guide in Section 5.2 can now be applied.

### 5.3.1. Identify

#### a) The Component's Tasks

This first point of the guide is more of a help to get some form of definition into the new component. If the component is completely new, this can help with the other four identification tasks, namely the structure integration, types, interface, and wrappers.

In the case of the communication example on the other hand, this is not a lot of work as the functionality is supposed to mirror the functionality of the current *Participant* and *Mailbox* modules.

Based on the original task and the existing code for *Participant* and *Mailbox* (the source files can be found towards the end of this section, namely on pages 72 and 73), the following tasks can be identified:

- **Participant**
  - Get registered at Comms module
  - Have a Mailbox to hold messages
  - Retrieve messages from Mailbox
- **Mailbox**
  - Receive, store, and return messages

The first point ('Get registered at Comms module') is written down as a more broad task to hint to it having to be included in the inheritance structure. Any functionality coming from that requirement is being implemented by it being a *BaseParticipant* which can get registered at the Comms module.

#### b) The Integration into the Current Structure

At this point, it needs to be defined how the component should fit into the current project's structure. As described before, the new component is supposed to fit within the abstract structure of the project. This is done with the goal of other modules like *Comms* being able to simply use *BaseParticipant* without differentiating between the Rust or the C++ version, leading to less repetitive code.

To be able to include the component without much troubles, it is very helpful to implement a C++ wrapper class. Analog to the existing C++ modules *Participant* and *Mailbox*, the wrappers can then be included in the abstract structure, resulting in the structure shown in Figure 5.5.

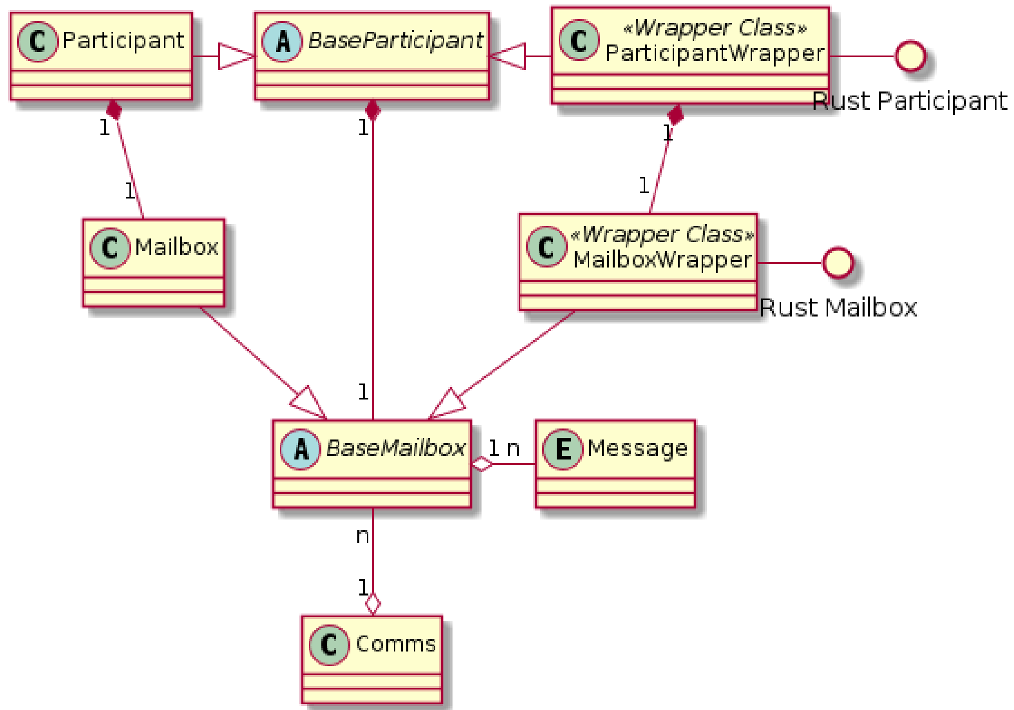


Figure 5.5.: Bare class diagram with Rust components

### c) Custom Types

When looking at the structure and the definitions in *types.h*, one struct and two types that can be identified to be required in both components.

```
4  #include <cstdint>
5
6  typedef int32_t participant_id_t;
7  typedef int32_t message_id_t;
8
9  struct Message
10 {
11     message_id_t id;
12     int32_t length;
13 };
```

Listing 5.1: Custom types and structs defined in *types.h*

Firstly, the *Message* struct that is used by the *BaseMailbox* which contains an ID as well as a length. The other two custom types introduced are the *participant_id_t* and the *message_id_t*.

#### d) The Component's Interface

As the Rust components in this example are counterparts to the already existing C++ classes which are defined by the abstract structure, it is less work to define the necessary interface in this case compared to a replacement or a completely new component.

Thus, to identify the interface in this example project, it is good to look at the interface that the C++ classes currently have, especially which functions and methods are being implemented by the abstract class and which of these have to be defined by the derived class that is supposed to be an equivalent of the already existing component.

As it is a little simpler, the *Mailbox* will be looked at first in Listings 5.2 and 5.3.

```
4  #include "types.h"
5
6  class BaseMailbox
7  {
8  public:
9      BaseMailbox(){};
10
11     virtual Message get_next_message() = 0;
12     virtual int get_size() = 0;
13     virtual void push_message(Message msg) = 0;
14 };
```

Listing 5.2: Header file with declarations for the abstract class BaseMailbox (*base_mailbox.h*)

```
4  #include <vector>
5
6  #include "base_mailbox.h"
7
8  class Mailbox : public BaseMailbox
9  {
10 private:
11     std::vector<Message> message_vec;
12
13 public:
14     Mailbox() : BaseMailbox(){};
15
16     Message get_next_message() override;
17     int get_size() override;
18     void push_message(Message msg) override;
19 };
```

Listing 5.3: Header file with declarations for the derived class Mailbox (*mailbox.h*)

As the Rust implementation of the *Mailbox* needs to hold values, in this case multiple *Messages* like the *Mailbox* does with the `std::vector<Message>` (line 11 in Listing 5.3), it needs to have some kind of data structure. The most suitable option for a class equivalent would be a Rust struct with a variable, which in this case will also be a Vector with *Messages*.

As structs in Rust don't have a constructor but have to be created manually, a function that creates it is required, thus resulting in the following requirements:

- a **struct** called *MailboxRust* that has a **Vector with *Messages***
- a **function** called *create_mailbox_rust()* that creates and **returns that struct**

The names *MailboxRust* and *create_mailbox_rust()* are hereby chosen completely freely, there is no naming convention that has to be followed. As a C++ wrapper class is going to be used for the integration into the inheritance structure, the names of the Rust functions could also be chosen freely. To keep some kind of order, the Rust functions are getting the same name as their C++ counterparts with a `_rust` appendix to tell them apart.

The only thing missing now are the three pure virtual functions that are required by the *BaseMailbox* component (lines 11-13 in Listing 5.2), also implemented by *Mailbox* (lines 16-18 in Listing 5.3):

- a **function** called *get_next_message_rust()* with **no parameters** that **returns a *Message***
- a **function** called *get_size_rust()* with **no parameters** that **returns an Integer**
- a **function** called *push_message_rust()* with a ***Message*** **parameter** that **returns a void value**

That is everything that is required for the *Mailbox* interface.
The *Participant* works just the same with the header files seen in Listings 5.4 and 5.5.

```
4  #include <unistd.h>
5  #include <thread>
6  #include <iostream>
7
8  #include "types.h"
9  #include "base_mailbox.h"
10
11 class BaseParticipant
12 {
13 private:
14     std::thread *t;
15
16 protected:
17     participant_id_t id;
18     BaseMailbox *mailbox;
19
20     virtual void print_message(Message msg) = 0;
21     virtual void receive_message();
22
23 public:
24     BaseParticipant(participant_id_t id = 0);
25
26     participant_id_t get_id();
27     BaseMailbox *get_mailbox();
28
29     void poll_every_second();
30     void wait();
31 };
```

Listing 5.4: Header file with declarations for the abstract class BaseParticipant (*base_participant.h*)

```
4  #include "mailbox.h"
5  #include "base_participant.h"
6
7  class Participant : public BaseParticipant
8  {
9  private:
10     void print_message(Message msg) override;
11
12 public:
13     Participant(participant_id_t id);
14 };
```

Listing 5.5: Header file with declarations for the derived class Participant (*participant.h*)

Just like the *Mailbox*, the *Participant* also needs to hold data, thus requiring a struct and its creation function. The data is a bit different though, as it only needs the members that are actually required within the Rust component. An example for that is the private member `std::thread *t` that is only used in the *wait()* function (as seen in *participant.cpp* in the end of this section, found in Listing 5.27 on page 73) which is not virtual. Thus, it is not going to be re-implemented in Rust, leading to it not being required in the Rust component as well

as the interface. The other two members, the *participant_id_t* and *BaseMailbox* have to be implemented. While the ID is simply being passed as a parameter, as visible in line 22 of Listing 5.4, the *BaseMailbox* will be initiated in the creation function.

It is notable that the Mailbox component used in this variable (line 17 of Listing 5.4) isn't a regular C++ type but also has a counterpart written in Rust. This enables the *ParticipantRust* to simply use the Rust equivalent, namely the *MailboxRust*-struct.

Thus, the first part of the interface looks as follows, very similar to the *Mailbox*:

- a **struct** called *ParticipantRust* that has a **participant_id_t** and a ***MailboxRust***
- a **function** called *create_participant_rust()* with a **parameter of type participant_id_t** that creates and **returns that struct**

Again, the naming of *ParticipantRust* and the functions can be chosen freely, just like with the *Mailbox*.

When looking at the virtual functions in Listing 5.4, only *print_message()* has to be re-implemented in the derived class as it is a pure virtual function (line 20, marked by '`= 0`').

The other three functions are defined in *BaseParticipant* and don't need to be overridden. This can be seen in *Participant* (see line 10 in Listing 5.5) where only *print_message()* is included in the header file.

To use at least some Rust program logic and to demonstrate a regular override, it was decided that the function *receive_message()* was also supposed to be implemented in Rust. This is not necessary for the functionality of the program, it works fine with either the abstract implementation of *BaseParticipant* or the Rust version.

From this, the second part of the interface can be completed with the following two functions:

- a **function** called *print_message_rust()* with a ***Message*** **parameter** that **returns a void value**
- a **function** called *receive_message_rust()* with **no parameters** that **returns a void value**

#### e) The C++ Wrappers

The C++ wrappers are there to serve two main purposes which will be talked through similar to the previous paragraphs. The purposes are as follows:

1. Fit into the current structure, in this case the inheritance
2. Translate function calls with correct typecasts

Starting with the *Mailbox* again, the wrapper is supposed to become a derived class from *BaseMailbox* meaning that on one hand, it needs to have the necessary functions that the *BaseMailbox* requires like *get_next_message()*. Just like with the interface, these are easy to identify by looking at the virtual functions and their derived declarations in Listings 5.2 and 5.3 as well as the functions that the Rust interface offers. On the other hand, the wrapper is responsible for setting up and communicating with the Rust component. For this, it needs to have the Rust struct *MailboxRust* as a local variable, which is the last missing thing in the wrapper class.

Thus, the wrapper needs to be a **class** inheriting from ***BaseMailbox*** and requires

- a **variable** of the type ***MailboxRust***
- a **constructor** to create the above mentioned variable
- a **function** called *get_next_message()* with **no parameters** that **returns a *Message***
- a **function** called *get_size()* with **no parameters** that **returns an Integer**
- a **function** called *push_message()* with a ***Message*** **parameter** that **returns a void value**

The *Participant* wrapper works exactly the same way:
The wrapper needs to be a **class** inheriting from ***BaseParticipant*** and requires

- a **variable** of the type ***ParticipantRust***
- a **constructor** with a **parameter of type participant_id_t** to create the above mentioned variable
- a **function** called *print_message()* with a ***Message*** **parameter** that **returns a void value**
- a **function** called *receive_message()* with **no parameters** that **returns a void value**

### Summary

The class diagram in Figure 5.6 shows the full project together with the wrappers.

This type of diagram, though not required, can help greatly with the implementation and understanding of the interface.

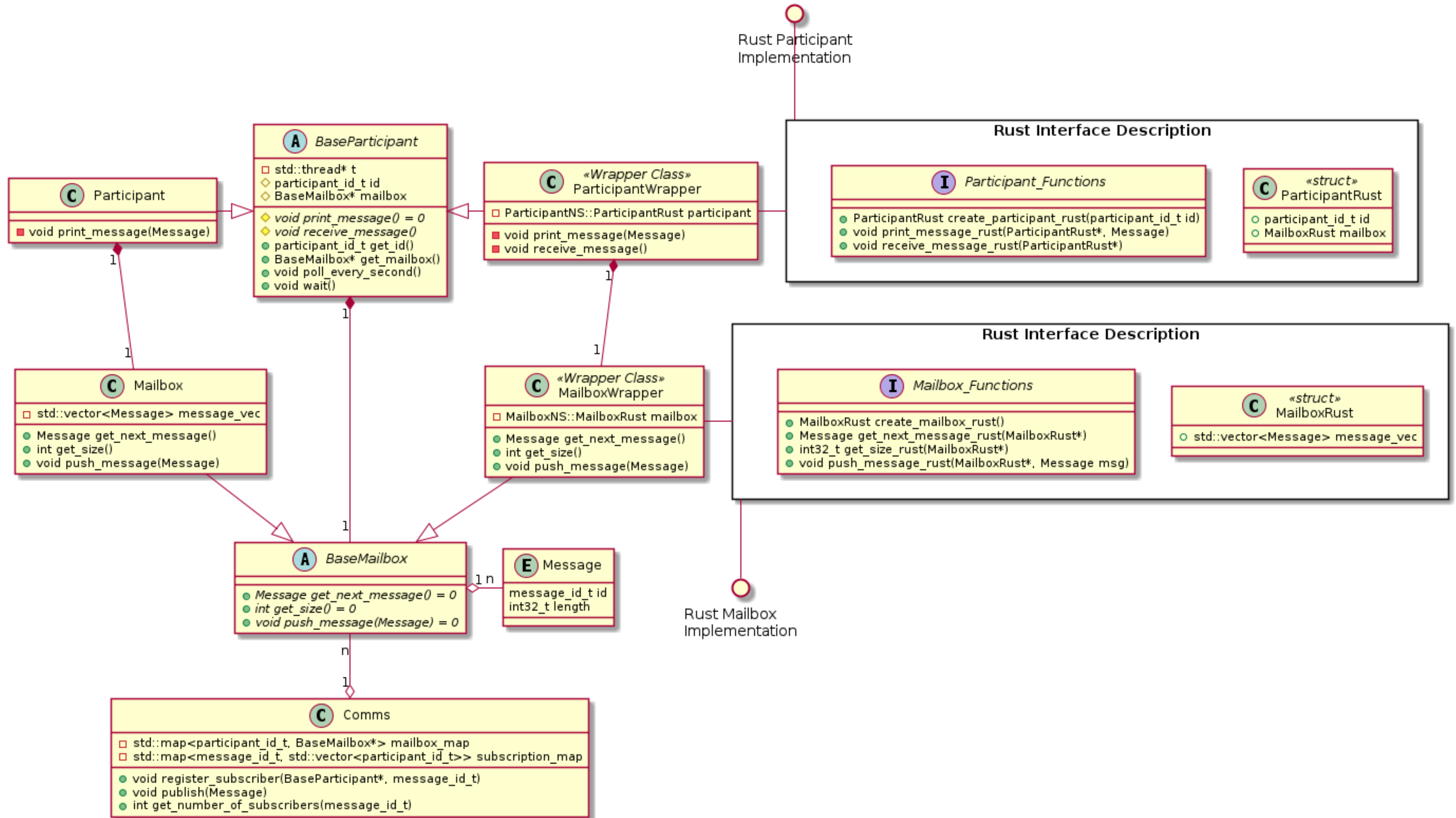


Figure 5.6.: Class diagram with detailed interface description

### 5.3.2. Implement

This short section will go over the implementation of the necessary files with the identified functionalities. It is important to know that not every variable and function has to be found out during the identification process, some additional requirements may also come up during the implementation.

#### a) The Header Files Containing the Required C++ Bindings for the Rust Interface

The first step to implementing the binding file, again starting with the *Mailbox*, is taking the requirements and writing them down in C++ (see step d) in Subsection 5.3.1 for the identification of the requirements). Any member functions will be called standalone with the *function(RustObject)* notation, taking `&self` as their first argument (refer to Subsection 4.2.1 for more information), resulting in a `MailboxRust *` parameter getting added to the call.

Thus, the following requirements taken from page 54

- a **struct** called *MailboxRust* that has a **Vector with *Messages***
- a **function** called *create_mailbox_rust()* that creates and **returns that struct**
- a **function** called *get_next_message_rust()* with **no parameters** that **returns a *Message***
- a **function** called *get_size_rust()* with **no parameters** that **returns an Integer**
- a **function** called *push_message_rust()* with a ***Message* parameter** that **returns a void value**

can be translated to C++ code seen in Listing 5.6.

```
struct MailboxRust
{
    std::vector<Message> message_vec;
};

extern "C"
{
    MailboxRust create_mailbox_rust();

    Message get_next_message_rust(MailboxRust *);
    int32_t get_size_rust(MailboxRust *);
    void push_message_rust(MailboxRust *, Message msg);

} // extern "C"
```

Listing 5.6: First approach to implementing the MailboxRust component

This is already all the productive code necessary for the functionality of the file. As mentioned in Subsection 4.2.1, it is helpful to introduce a namespace which in this case will be called `MailboxNS`. Together with the namespace, some required imports, and standard boilerplate code, this results in the full file *mailbox_rust.h*.

```
4  #include <cstdint>
5  #include <vector>
6
7  #include "types.h"
8
9  namespace MailboxNS
10 {
11     struct MailboxRust
12     {
13         std::vector<Message> message_vec;
14     };
15
16     extern "C"
17     {
18         MailboxRust create_mailbox_rust();
19
20         Message get_next_message_rust(MailboxRust *);
21         int32_t get_size_rust(MailboxRust *);
22         void push_message_rust(MailboxRust *, Message msg);
23     } // extern "C"
24 } // namespace MailboxNS
```

Listing 5.7: Implementation of the C++ bindings for the Mailbox interface (*mailbox_rust.h*)

The implementation of the *Participant* interface works exactly the same:

Requirements (found on page 56):

- a **struct** called *ParticipantRust* that has a **participant_id_t** and a ***MailboxRust***
- a **function** called *create_participant_rust()* with a **parameter of type participant_id_t** that creates and **returns that struct**
- a **function** called *print_message_rust()* with a ***Message*** **parameter** that **returns a void value**
- a **function** called *receive_message_rust()* with **no parameters** that **returns a void value**

Code:

```
4  #include "types.h"
5  #include "mailbox_rust.h"
6
7  namespace ParticipantNS
8  {
9      struct ParticipantRust
10     {
11         participant_id_t id;
12         MailboxNS::MailboxRust mailbox;
13     };
14
15     extern "C"
16     {
17         ParticipantRust create_participant_rust(participant_id_t id);
18
19         void print_message_rust(ParticipantRust *, Message);
20         void receive_message_rust(ParticipantRust *);
21     } // extern "C"
22 } // namespace ParticipantNS
```

Listing 5.8: Implementation of the C++ bindings for the Participant interface (*participant_rust.h*)

### b) The C++ Wrapper

Just like the implementation of the interfaces, the code for the wrappers can be taken from the requirements that were identified in the first part of the process.

To recall (see page 57), the wrapper needs to be a **class** inheriting from ***BaseMailbox*** and requires

- a **variable** of the type ***MailboxRust***
- a **constructor** to create the above mentioned variable
- a **function** called *get_next_message()* with **no parameters** that **returns a *Message***
- a **function** called *get_size()* with **no parameters** that **returns an Integer**
- a **function** called *push_message()* with a ***Message*** **parameter** that **returns a void value**

Listing 5.9 shows a first C++ implementation of these bare components.

```
class MailboxWrapper : public BaseMailbox
{
private:
    MailboxNS::MailboxRust *mailbox;

public:
    MailboxWrapper();

    Message get_next_message() override;
    int get_size() override;
    void push_message(Message msg) override;
};
```

Listing 5.9: A base implementation of MailboxWrapper

Together with including any necessary libraries, this results in the header file seen in Listing 5.10.

```
#include "types.h"
#include "base_mailbox.h"
#include "mailbox_rust.h"

class MailboxWrapper : public BaseMailbox
{
private:
    MailboxNS::MailboxRust *mailbox;

public:
    MailboxWrapper();

    Message get_next_message() override;
    int get_size() override;
    void push_message(Message msg) override;
};
```

Listing 5.10: Header file for the C++ wrapper class representing the Mailbox (*mailbox_rust_wrapper.h*)

The respective implementation in the source file *mailbox_rust_wrapper.cpp* seen in Listing 5.11 is pretty straight-forward, simply calling functions and passing on their return types.

```
#include "mailbox_rust_wrapper.h"

MailboxWrapper::MailboxWrapper() : BaseMailbox()
{
    MailboxNS::MailboxRust m = MailboxNS::create_mailbox_rust();
    mailbox = &m;
};

Message MailboxWrapper::get_next_message()
{
    return MailboxNS::get_next_message_rust(mailbox);
}

int MailboxWrapper::get_size()
{
    return MailboxNS::get_size_rust(mailbox);
}

void MailboxWrapper::push_message(Message msg)
{
    MailboxNS::push_message_rust(mailbox, msg);
}
```

Listing 5.11: Implementation of the C++ wrapper class representing the Mailbox (*mailbox_rust_wrapper.cpp*)

The same goes for the *Participant* wrapper, first the requirements (identified on page 57) and the resulting code in Listing 5.12.

The wrapper needs to be a **class** inheriting from ***BaseParticipant*** and requires

- a **variable** of the type ***ParticipantRust***
- a **constructor** with a **parameter of type participant_id_t** to create the above mentioned variable
- a **function** called *print_message()* with a ***Message*** **parameter** that **returns a void value**
- a **function** called *receive_message()* with **no parameters** that **returns a void value**

```
class ParticipantWrapper : public BaseParticipant
{
private:
    ParticipantNS::ParticipantRust participant;

    void print_message(Message msg);
    void receive_message() override;

public:
    ParticipantWrapper(participant_id_t id);
};
```

Listing 5.12: First base implementation of ParticipantWrapper

With the necessary includes and boilerplate code, the following header file is the result:

```
#include "types.h"
#include "base_participant.h"
#include "participant_rust.h"

class ParticipantWrapper : public BaseParticipant
{
private:
    ParticipantNS::ParticipantRust participant;

    void print_message(Message msg);
    void receive_message() override;

public:
    ParticipantWrapper(participant_id_t id);
};
```

Listing 5.13: Header file for the C++ wrapper class representing the Participant (*participant_rust_wrapper.h*)

Analog to the *Mailbox*, the source file can now be put together, resulting in *participant_rust_wrapper.cpp* seen in Listing 5.14.

```
#include "participant_rust_wrapper.h"

ParticipantWrapper::ParticipantWrapper(participant_id_t id) :
    BaseParticipant(id)
{
    participant = ParticipantNS::create_participant_rust(id);
}

void ParticipantWrapper::print_message(Message msg)
{
    ParticipantNS::print_message_rust(&participant, msg);
}

void ParticipantWrapper::receive_message()
{
    ParticipantNS::receive_message_rust(&participant);
}
```

Listing 5.14: Implementation of the C++ wrapper class representing the Participant (*participant_rust_wrapper.cpp*)

In theory, the wrapper implementation could be done at this point. In reality, when the user tries to run this code, they will get a *Segmentation fault (core dumped)* due to the attributes that *BaseParticipant* requires but aren't defined yet. These attributes consist of the `participant_id_t id` and the `BaseMailbox *mailbox` (see lines 17 and 18 in Listing 5.4).

Thus, these required arguments need to be defined in the constructor of the *ParticipantWrapper*. The *id* is fairly simple, as it is already passed as a parameter like in the normal C++ *Participant* (see Listing 5.5), it can just be assigned. The *Mailbox* on the other hand, in this case the *MailboxRust*, has to be created somewhere and assigned to the *ParticipantRust*-struct accordingly. This can either happen by creating and assigning it in the wrapper or by the *Participant* or *RustParticipant* initializing their own *Mailbox* which can then be accessed with a getter-function.

The second option was chosen for this project, resulting in a few necessary changes. To start with the wrappers, the current version of *MailboxWrapper* creates its own *MailboxRust* in its constructor (see line 5 in Listing 5.11). If that is created externally, a new constructor is required with a *MailboxRust* parameter that can then simply be assigned to the variable.

Apart from the actual implementation in the Rust component of course, the following steps and updates are required to add the derived arguments:

1. Add a *get_mailbox()*-function to the Rust component (shown in Listing 5.22 on page 69) and the *RustParticipant* interface (line 21 in Listing 5.15)

```
4  #include "types.h"
5  #include "mailbox_rust.h"
6
7  namespace ParticipantNS
8  {
9      struct ParticipantRust
10     {
11         participant_id_t id;
12         MailboxNS::MailboxRust mailbox;
13     };
14
15     extern "C"
16     {
17         ParticipantRust create_participant_rust(participant_id_t id);
18
19         void print_message_rust(ParticipantRust *, Message);
20         void receive_message_rust(ParticipantRust *);
21         MailboxNS::MailboxRust *get_mailbox_rust(ParticipantRust *);
22     } // extern "C"
23 } // namespace ParticipantNS
```

Listing 5.15: *participant_rust.h* with added getter function in line 21

2. Add a constructor for *MailboxWrapper* with a *MailboxRust* parameter (line 15 in Listing 5.16 and lines 5-8 in Listing 5.17)

```
4  #include "types.h"
5  #include "base_mailbox.h"
6  #include "mailbox_rust.h"
7
8  class MailboxWrapper : public BaseMailbox
9  {
10 private:
11     MailboxNS::MailboxRust *mailbox;
12
13 public:
14     MailboxWrapper();
15     MailboxWrapper(MailboxNS::MailboxRust *mb);
16
17     Message get_next_message() override;
18     int get_size() override;
19     void push_message(Message msg) override;
20 };
```

Listing 5.16: *mailbox_rust_wrapper.h* with additional constructor in line 15

```cpp
#include "mailbox_rust_wrapper.h"

MailboxWrapper::MailboxWrapper() : BaseMailbox() { ... };

MailboxWrapper::MailboxWrapper(MailboxNS::MailboxRust *mb) : BaseMailbox()
{
    mailbox = mb;
};

Message MailboxWrapper::get_next_message() { ... }

int MailboxWrapper::get_size() { ... }

void MailboxWrapper::push_message(Message msg) { ... }
```

Listing 5.17: *mailbox_rust_wrapper.cpp* with additional constructor in lines 9-12

3. Initialize the *participant_id_t* and *MailboxWrapper* variables in the *ParticipantWrapper* constructor (lines 6 and 7 in Listing 5.18)

```cpp
#include "participant_rust_wrapper.h"

ParticipantWrapper::ParticipantWrapper(participant_id_t id) :
    BaseParticipant(id)
{
    this->id = id;
    participant = ParticipantNS::create_participant_rust(id);
    mailbox = new MailboxWrapper(ParticipantNS::get_mailbox_rust(&
        participant));
}

void ParticipantWrapper::print_message(Message msg) { ... }

void ParticipantWrapper::receive_message() { ... }
```

Listing 5.18: *participant_rust_wrapper.cpp* with updated constructor (lines 5 and 7)

#### c) Custom Types

The custom types that were identified and that are required were the struct *Message* and the two types *participant_id_t* and *message_id_t*, all saved in the file `types.h` (Listing 5.19).

```cpp
#include <cstdint>

typedef int32_t participant_id_t;
typedef int32_t message_id_t;

struct Message
{
    message_id_t id;
    int32_t length;
};
```

Listing 5.19: *types.h* with the necessary type definitions and bindings

Converted to Rust, they are saved in a file called *types.rs* seen in Listing 5.20. The types are defined just the same, with the only difference being the naming due to the Rust conventions which state that types are supposed to be named using UpperCamelCase [Rusc, Ch. 1].

```rust
pub type ParticipantIdT = i32;
pub type MessageIdT = i32;

#[repr(C)]
pub struct Message {
    pub id: MessageIdT,
    pub length: i32,
}
```

Listing 5.20: *types.rs* with the corresponding type and struct definitions

#### d) The Rust Component

Even though the implementation is not part of this thesis, the following two listings show the code of both *ParticipantRust* and *MailboxRust*:

```
use crate::types::Message;

#[repr(C)]
pub struct MailboxRust {
    message_vec: Vec<Message>,
}

impl MailboxRust {
    #[no_mangle]
    pub extern "C" fn create_mailbox_rust() -> Self {
        MailboxRust {
            message_vec: Vec::new(),
        }
    }

    #[no_mangle]
    pub extern "C" fn get_next_message_rust(&mut self) -> Option<Message> {
        self.message_vec.pop()
    }

    #[no_mangle]
    pub extern "C" fn get_size_rust(&self) -> i32 {
        return self.message_vec.len() as i32;
    }

    #[no_mangle]
    pub extern "C" fn push_message_rust(&mut self, msg: Message) -> () {
        self.message_vec.push(msg);
    }
}
```

Listing 5.21: Rust implementation of MailboxRust (*mailbox_rust.rs*)

```
use crate::mailbox_rust::MailboxRust;
use crate::types::{Message, ParticipantIdT};

#[repr(C)]
pub struct ParticipantRust {
    id: ParticipantIdT,
    mailbox: MailboxRust,
}

impl ParticipantRust {
    #[no_mangle]
    pub extern "C" fn create_participant_rust(id: ParticipantIdT) -> Self {
        ParticipantRust {
            id,
            mailbox: MailboxRust::create_mailbox_rust(),
        }
    }
```

```
18
19      #[no_mangle]
20      pub extern "C" fn print_message_rust(&self, msg: Message) -> () { ... }
21
22      #[no_mangle]
23      pub extern "C" fn receive_message_rust(&mut self) -> () {
24          if self.mailbox.get_size_rust() >= 1 {
25              let msg = self.mailbox.get_next_message_rust().unwrap();
26              self.print_message_rust(msg);
27          }
28      }
29
30      #[no_mangle]
31      pub extern "C" fn get_mailbox_rust(&self) -> &MailboxRust { ... }
32  }
```

Listing 5.22: Rust implementation of ParticipantRust (*participant_rust.rs*)

The only thing missing to be able to compile this project and jump to the final step in the guide are the additions in *lib.rs* and *Cargo.toml*.

```
1  pub mod types;
2
3  mod mailbox_rust;
4  mod participant_rust;
```

Listing 5.23: Simple *lib.rs* including the modules MailboxRust and ParticipantRust

```
1  [package]
2  name = "component_rust"
3  version = "0.1.0"
4  edition = "2021"
5
6  [lib]
7  crate-type = ["staticlib"]
```

Listing 5.24: Barebone *Cargo.toml* to compile the Rust component into a static library

#### Summary

The class diagram on the following page (Figure 5.7) shows the updated project. Even though the only difference to Figure 5.6 is the additional `get_mailbox_rust(ParticipantRust*)` function in the Rust Interface Description, this diagram is included for completion reasons.

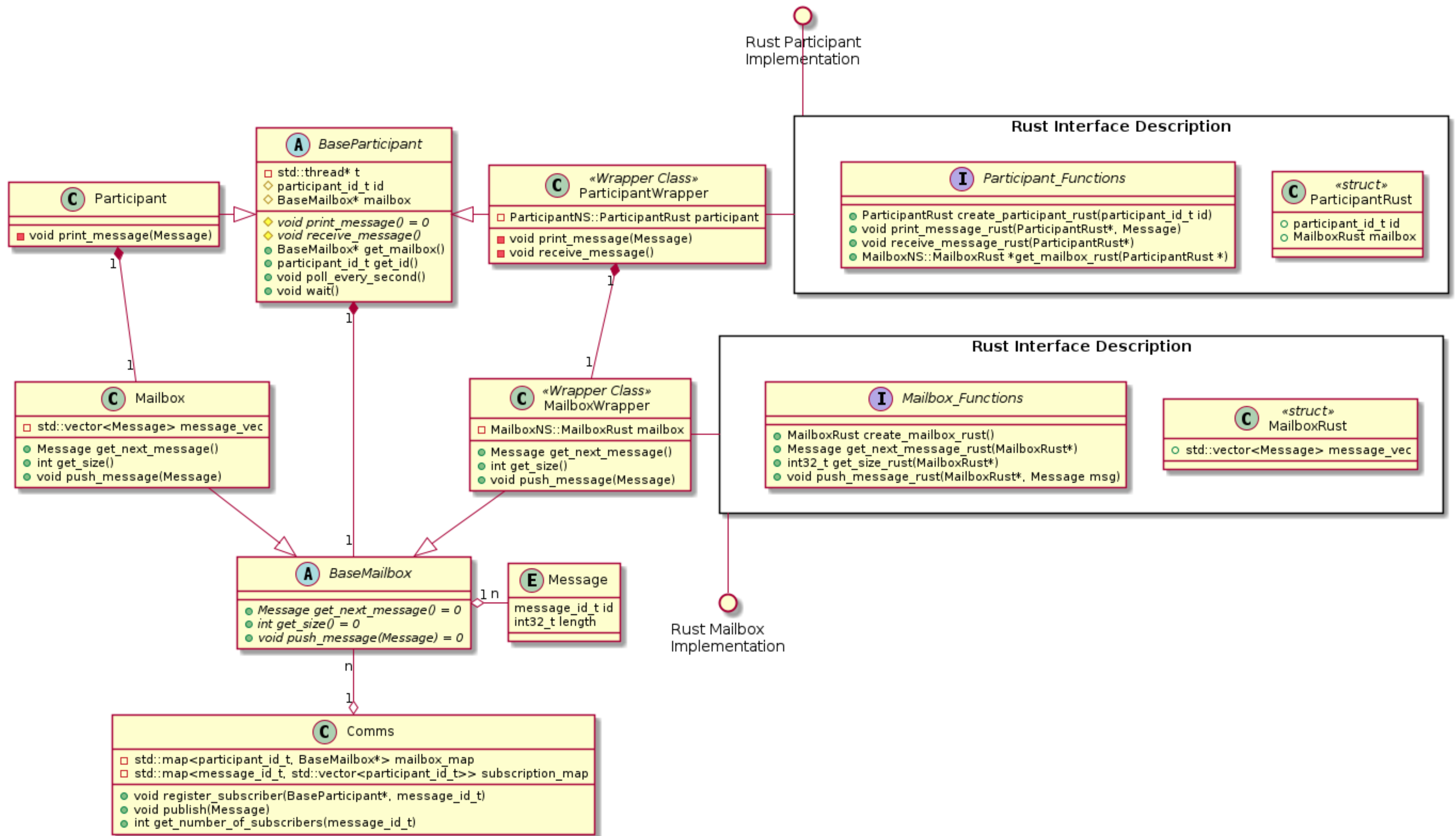


Figure 5.7.: Updated class diagram with detailed interface description

Additionally, the implementations of the rest of the files, namely *Participant* and *Mailbox* together with *BaseParticipant*, are included on the following pages. As all members in *BaseMailbox* are virtual, there is no implementation of *base_mailbox.cpp*.

```
#include "mailbox.h"

Message Mailbox::get_next_message()
{
    Message front;
    if (this->message_vec.size() > 0)
    {
        front = this->message_vec.front();
        this->message_vec.erase(this->message_vec.begin());
    }
    return front;
}

int Mailbox::get_size() { ... }

void Mailbox::push_message(Message msg)
{
    this->message_vec.push_back(msg);
}
```

Listing 5.25: C++ implementation of Mailbox (*mailbox.cpp*)

```
#include "base_participant.h"

BaseParticipant::BaseParticipant(participant_id_t id) { this->id = id; }

participant_id_t BaseParticipant::get_id() { return id; }
BaseMailbox *BaseParticipant::get_mailbox() { return mailbox; }

void BaseParticipant::receive_message()
{
    if (this->mailbox->get_size() >= 1)
    {
        Message msg = this->mailbox->get_next_message();
        print_message(msg);
    }
}

void BaseParticipant::poll_every_second() { ... }

void BaseParticipant::wait() { ... }
```

Listing 5.26: Shortened C++ implementation of the abstract class BaseParticipant (*base_participant.cpp*)

The full file (*base_participant.cpp*) without placeholders can be found in the appendix in Listing A.8.

```cpp
#include "participant.h"

Participant::Participant(participant_id_t id) : BaseParticipant(id)
{
    mailbox = new Mailbox();
}

void Participant::print_message(Message msg)
{
    std::cout << "Participant " << this->id << ": Message with id " << msg.
        id << " and length " << msg.length << " was received" << std::endl;
}
```

Listing 5.27: C++ implementation of Participant (*participant.cpp*)

### 5.3.3. Include

The guide's last section deals with the inclusion of the new code into the already existing code. As the Rust part isn't going to replace any existing module in this example, the following paragraphs will deal with the calls to the wrappers and whether there's a difference between the currently existing and new component.

#### a) The New Component in the Current Project

As the Rust component in this project was implemented using a C++ wrapper in the inheritance structure, it can easily be called and treated like a C++ participant from *main.cpp*, as seen in lines 15 and 16 of Listing 5.28 where the participants are being created and lines 22-25 which is how they get subscribed to their necessary messages.

```cpp
#include <iostream>
#include <thread>

#include "comms.h"
#include "participant.h"
#include "participant_rust_wrapper.h"

using namespace std;

int main(int argc, char *argv[])
{
    Comms comms = Comms();

    BaseParticipant *part_1 = new Participant(1);
    BaseParticipant *part_2 = new Participant(2);
    BaseParticipant *rust_part_3 = new ParticipantWrapper(3);

    part_1->poll_every_second();
    part_2->poll_every_second();
    rust_part_3->poll_every_second();

    comms.register_subscriber(part_1, 1);
    comms.register_subscriber(rust_part_3, 1);
    comms.register_subscriber(rust_part_3, 2);
    comms.register_subscriber(rust_part_3, 3);

    std::cout << "All Subscribers registered and listening" << std::endl;

    std::this_thread::sleep_for(2000ms);

    Message msg;
    msg.id = 1;
    msg.length = 5;

    comms.publish(msg);

    std::this_thread::sleep_for(2000ms);

    msg.id = 3;
    msg.length = 10;

    comms.publish(msg);

    part_1->wait();
    part_2->wait();
    rust_part_3->wait();

    return 0;
}
```

Listing 5.28: An example main function showing the usage of both the C++ as well as the Rust components

### b) The New Component in the Existing Compilation Process

The final step consists of including the Rust component into the current process. At the moment, the project is compiled and linked using the following *CMakeLists.txt* script.

```
cmake_minimum_required(VERSION 3.0)
project(CommmunicationExample)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wpedantic -Werror")
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")

include_directories("include")
include_directories("src")

add_executable(communication
    main.cpp
    src/base_participant.cpp
    src/comms.cpp
    src/mailbox_rust_wrapper.cpp
    src/mailbox.cpp
    src/participant_rust_wrapper.cpp
    src/participant.cpp
)
```

Listing 5.29: *CMakeLists.txt* file only containing the C++ files

In order to include the Rust project *component_rust*, a way to compile the source files has to be added to the process. Using the tool *CMakeRust*, the necessary CMake files are downloaded and placed in a subdirectory named *cmake/*. [Dev23]

The only thing left that is required to be able to use *CMakeRust* is a *CMakeLists.txt* file in *component_rust/* with the following line:

```
cargo_build(NAME component_rust)
```

Together with an additional *build/* directory, this results in the file structure seen in Figure 5.8, showing only the relevant files.

```
├── build/
├── cmake/
├── component_rust/
│   └── CMakeLists.txt
├── include/
├── src/
├── CMakeLists.txt
└── main.cpp
```

Figure 5.8.: File structure with first additions

The Rust component can then simply be included in the compilation process by adding its subdirectory and linking it, which can be seen in lines 12 and 24 in Listing 5.30.

```
cmake_minimum_required(VERSION 3.0)
project(CommmunicationExample)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wpedantic -Werror")
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")

enable_language(Rust)
include(CMakeCargo)

include_directories("include")
include_directories("src")
add_subdirectory(component_rust)

add_executable(communication
    main.cpp
    src/base_participant.cpp
    src/comms.cpp
    src/mailbox_rust_wrapper.cpp
    src/mailbox.cpp
    src/participant_rust_wrapper.cpp
    src/participant.cpp
)

target_link_libraries(communication component_rust)
```

Listing 5.30: Updated *CMakeLists.txt* with CMakeRust call

This is everything that's required in order to include the Rust component. Calling `cmake ..` and `make` in the *build/* directory creates an executable file called *communication* which has the output seen in Listing 5.31 after execution.

```
All Subscribers registered and listening
Participant 1: Message with id 1 and length 5 was received
Rust Participant 3: Message with id 1 and length 5 was received
Rust Participant 3: Message with id 3 and length 10 was received
```

Listing 5.31: Final output of the communication example in Listing 5.28

The output shows that both the C++ and Rust component work equally.

# 6. Embedded Development

An additional task was the integration of a Rust library into an embedded C++ application running on an ARMv7-A processor with hard-float numbers. For that, a small example application was written and then included into and called by an already existing software.

## 6.1. Code Preparation

As this was only a small feasibility analysis, the small default library that is created when executing `cargo new name --lib` was used, although slight changes to the data types were done. Additionally, as it was planned to use this library in a `no_std` environment, this attribute together with a panic handler had to be added. The latter's implementation is taken from the chapter "The smallest #![no_std] program" from *The Embedonomicon* [Emb, Ch. 1]. The full code of the file *lib.rs* can be seen in Listing 6.1.

```
1  #![no_std]
2
3  #[no_mangle]
4  pub extern "C" fn add(left: usize, right: usize) -> usize {
5      left + right
6  }
7
8  #[panic_handler]
9  fn panic(_info: &core::panic::PanicInfo) -> ! {
10     loop {}
11 }
```

Listing 6.1: Example of Rust library code for an embedded system

## 6.2. Platform Support

Rust's support for embedded platforms is split into three different tiers that, together with the availability of host tools for them, can be found in Chapter 6 of *The rustc Book*[Thed]. The difference with these tiers is within the set of guarantees that can be made for each target.

Any target found within Tier 1 or 2 can simply be compiled by first adding it via
rustup target add <<target_name>> and then executing
cargo build --target <<target_name>>.

The combination of ARMv/-A together with the hard-float option results in the target for this example being *armv7a-none-eabihf* which can be found in Tier 3 as of October 31st, 2023. [Thed]

For all the targets found in Tier 3, no official build of the standard library is available thus it needs to be built manually by adding `-Z build-std`. Hereby, the `build-std` argument is an unstable feature which can only be used by adding the `-Z` flag. [Thea, Ch. 3.18] If only a certain part, which in the case of the small example here is `core`, is required, it can be specified to only build this part by using `build-std=core` for example. If nothing is specified, `core`, `std`, `alloc`, and `proc_macro` are all being built.

## 6.3. Nightly Build

As `-Z` is an unstable feature, the regular Rust compiler won't be able to execute the command and compile the library. Thus, the *nightly* branch of the compiler needs to be used which contains experimental features that have not made it into the full release (yet). [KN18, Ch. 21.7]

Switching the compiler to use a nightly version is simply done by executing `rustup default nightly`. This downloads the required packages and finishes by telling the developer the version that it has switched to. The console output of the switch in this example can be seen in Listing 6.2

```
> rustup default nightly
info: syncing channel updates for 'nightly-x86_64-unknown-linux-gnu'
info: latest update on 2023-10-31, rust version 1.75.0-nightly (31bc7e2c4
   2023-10-30)
info: downloading component 'cargo'
[...]
info: installing component 'rustfmt'
info: default toolchain set to 'nightly-x86_64-unknown-linux-gnu'

  nightly-x86_64-unknown-linux-gnu installed - rustc 1.75.0-nightly (31
     bc7e2c4 2023-10-30)
```

Listing 6.2: Console output after switching the default Rust compiler to nightly

Afterwards, the source code for the standard library needs to be downloaded before it can be compiled. This can be done by executing `rustup component add rust-src`. If that's not done, the compiler will not be able to find the source code and exit with an error.

After all these steps have taken place, the small example project can be compiled using `cargo build -Z build-std=core --target armv7a-none-eabihf`.

To switch back to the stable toolchain, the command `rustup default stable` can be executed.

## 6.4. Summary

The following list sums up the steps necessary to compile a program for a target in the respective Tier:

### Tier 1 & 2

1. Add the target

```
> rustup target add <<target_name>>
```

2. Compile the project

```
> cargo build --target <<target_name>>
```

### Tier 3

1. Switch to a nightly branch

```
> rustup default nightly
```

2. Download the Rust source code

```
> rustup component add rust-src
```

3. Compile the project
   If for example only `core` and `std` need to be compiled, this can be done by changing `build-std` to `build-std=core,std`

```
> cargo build -Z build-std --target <<target_name>>
```

# 7. Discussion

When looking at the results, it was surprising to see how well C++ and Rust are able to work together considering it's not officially supported but is rather done using the ABI that Rust provides for working with C code. Additionally, the lack of an official reference by the Rust team on application binary interfaces was a bit surprising, considering their normally very high quality level of guides and documentation. The two small guides introduced in Chapter 2 only gave some basic examples and introduced the reader to a few more points of contact for additional research, like the *bindgen* and *cbindgen* tools, but did not really explore the basics of why this works as it does.

As mentioned in Subsection 4.4.3, this thesis did not cover more advanced and pointer-based types like Strings, Arrays, or Vectors. As these are commonly used within software development, this is a limitation that anyone using this guide and planning to do a full integration in their own project might encounter. In hindsight, it might have been a better choice to put the focus of this thesis on the cxx tool more extensively for the support of more complex types rather than the basics of including C++ in Rust applications. Not doing this places this thesis more in the "general integration/interoperability of C++ and Rust" section rather than the full integration guide that was originally planned.

That being said, as long as the exclusion of non-basic types and structures is being kept in mind, the guide composed in Section 5.2 is a very solid starting point for anyone looking to integrate Rust into their C++ applications.

Additionally, it is not being evaluated whether the usage of the FFI somehow affects the performance of C++ or Rust code. Though it is improbable that the Rust compiler acts and optimizes the code differently just because it is exposed to the FFI, any additional checks for memory safety and blocks of unsafe code could slow down the performance of the entire program.

# 8. Conclusion

The main takeaway from this thesis should be a basic understanding of the interaction between C++ and Rust components which was introduced in Sections 4.2 and 4.3. Using the guide (Section 5.2), it should also be possible to include newly developed Rust code into existing C++ applications.

As getting into Embedded Rust is another whole thesis on its own, its chapter (Chapter 6) was kept rather short, only giving a brief introduction into the compilation for certain target architectures in case the integration was planned for an embedded target.

## 8.1. Research Questions

In the beginning of this thesis (Section 1.1), the two following research questions were set out to be answered:

1. What is required for successfully including Rust in C++ applications and how can such an integration be approached?
2. Are there any limitations to the interoperability of C++ and Rust that could stand in the way of an integration, such as type or interface incompatibilities?

The first question was answered in Chapter 4 on a more theoretical level and with small working examples. Sections 5.1 and 5.2 then introduced a more concrete guide of working with bigger projects, eventually resulting in the big example in Section 5.3.

The second question can only be answered partially. While only covering basic types, there were no major limitations to the interoperability, with the only incompatible type being C++'s `long double` which does not have a counterpart in Rust. That being said, there might be things standing in the way of reliably being able to use Rust in C++ applications when dealing with more complex types that require pointers. As these were not covered in this thesis, no conclusive and complete statement can be made on this question.

## 8.2. Future Work

As already stated in the previous paragraph and in the **cxx**-part of Section 4.4.3, this thesis is not a complete guide for every C++ program as some data types like arrays or vectors were not fully covered. To complete this guide, a walkthrough on how to work with these types (for example by using the cxx Crate) would be necessary. A limitation here could be the dependency on the developers of this crate though there might also be a way to do it independently from any external crates. This could of course become a lot easier if Rust at some point switches to actually supporting C++'s ABI.

Additionally, it would be interesting how other frameworks like for example Qt with its GUI possibilities deal with Rust interoperability and whether its custom types are compatible with being included in Rust.

Another topic that could be expanded upon is the embedded development of Rust, maybe in connection with C's or C++'s low-level programming abilities.

# Listings


3.1. Very simple *lib.rs* file for compiling a library containing the modules *foo* and *bar* 8
3.2. *Cargo.toml* compiling into a static library target . . . . . . . . . . . . . . . . . . 9
3.3. Simple "Hello World!" example in *main.rs* . . . . . . . . . . . . . . . . . . . . 9
3.4. *Cargo.toml* compiling into a binary target . . . . . . . . . . . . . . . . . . . . 10
3.5. Setup of the basic *CMakeLists.txt* file developed in the CMake tutorial . . . . . 13
3.6. Minimum working *CMakeLists.txt* file . . . . . . . . . . . . . . . . . . . . . . 13

4.1. Rust struct definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2. Corresponding C++ struct definition . . . . . . . . . . . . . . . . . . . . . . . 18
4.3. Rust function definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.4. Corresponding C++ function binding . . . . . . . . . . . . . . . . . . . . . . . 19
4.5. Usage of Rust function in C++ code . . . . . . . . . . . . . . . . . . . . . . . 19
4.6. Rust struct and implemented function definition (*example.rs*) . . . . . . . . . . 20
4.7. C++ binding file with a simple method . . . . . . . . . . . . . . . . . . . . . . 20
4.8. Usage of Rust function in C++ code . . . . . . . . . . . . . . . . . . . . . . . 21
4.9. Output from C++ program with call to Rust method . . . . . . . . . . . . . . . 21
4.10. Header file of a simple wrapper implementation (*example_wrapper.h*) . . . . . . . 22
4.11. Implementation of a simple wrapper class with a constructor, a method, and a getter function (*example_wrapper.cpp*) . . . . . . . . . . . . . . . . . . . . . . . 22
4.12. Usage of a wrapper class in C++ code . . . . . . . . . . . . . . . . . . . . . . 23
4.13. Compilation output with type safety warnings . . . . . . . . . . . . . . . . . . 24
4.14. Rust struct updated with `std::ffi::c_char` type . . . . . . . . . . . . . . . . . 24
4.15. *Cargo.toml* of a simple Rust project, building a static library . . . . . . . . . . 25
4.16. *CMakeList.txt* of a simple C++project, including the Rust library in line 11 . . . 27
4.17. *CMakeLists.txt* using CMakeRust to compile the Rust code . . . . . . . . . . . 28
4.18. Struct declaration in C++ header file (*example.h)* . . . . . . . . . . . . . . . . 29
4.19. Rust bindings for struct example (*example_bindings.rs)* . . . . . . . . . . . . . 29
4.20. Function declaration in C++ header file (*example.h)* . . . . . . . . . . . . . . . 30
4.21. Function definition in C++ source file (*example.cpp)* . . . . . . . . . . . . . . . 30
4.22. Rust bindings for function example (*example_bindings.rs)* . . . . . . . . . . . . 30
4.23. Usage of C++ Function in Rust . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.24. Class and method declaration in C++ header file (*example.h)* . . . . . . . . . . 31

4.25. Class and method definition in C++ source file (*example.cpp)* . . . . . . . . . . . 31
4.26. Rust bindings for class and method example (*example_bindings.rs)* . . . . . . . . 31
4.27. Usage of C++ Class and its method in Rust, using the *function(Object)*-syntax . 32
4.28. Rust bindings and implemented wrapper function for C++ method (*example_bindings.rs*) 32
4.29. Usage of dot notation to access a C++ method in Rust . . . . . . . . . . . . . . 33
4.30. Compilation Output raising FFI-safety warnings . . . . . . . . . . . . . . . . . . 33
4.31. Updated *example_bindings.rs* using the `std::ffi::c_char` type . . . . . . . . . . 34
4.32. Updated main function using type-safe datatypes . . . . . . . . . . . . . . . . . . 34
4.33. Expanded rustc compilation command . . . . . . . . . . . . . . . . . . . . . . . . 35
4.34. Cargo.toml file including the example library . . . . . . . . . . . . . . . . . . . . 36
4.35. build.rs file to include a *libexample.a* archive and the C++ standard library . . . 36
4.36. Compilation command including the C++ archive and standard library . . . . . . 37
4.37. build.rs using a call by the cc crate . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.38. Compilation command with cc . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.39. build.rs using the cmake crate . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

5.1. Custom types and structs defined in *types.h* . . . . . . . . . . . . . . . . . . . . . 52
5.2. Header file with declarations for the abstract class BaseMailbox (*base_mailbox.h*) 53
5.3. Header file with declarations for the derived class Mailbox (*mailbox.h*) . . . . . . 53
5.4. Header file with declarations for the abstract class BaseParticipant (*base_participant.h*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.5. Header file with declarations for the derived class Participant (*participant.h*) . . 55
5.6. First approach to implementing the MailboxRust component . . . . . . . . . . . 59
5.7. Implementation of the C++ bindings for the Mailbox interface (*mailbox_rust.h*) . 60
5.8. Implementation of the C++ bindings for the Participant interface (*participant_rust.h*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.9. A base implementation of MailboxWrapper . . . . . . . . . . . . . . . . . . . . . 62
5.10. Header file for the C++ wrapper class representing the Mailbox (*mailbox_rust_wrapper.h*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.11. Implementation of the C++ wrapper class representing the Mailbox (*mailbox_rust_wrapper.cpp*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5.12. First base implementation of ParticipantWrapper . . . . . . . . . . . . . . . . . 64
5.13. Header file for the C++ wrapper class representing the Participant (*participant_rust_wrapper.h*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.14. Implementation of the C++ wrapper class representing the Participant (*participant_rust_wrapper.cpp*) . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.15. *participant_rust.h* with added getter function in line 21 . . . . . . . . . . . . . . 66
5.16. *mailbox_rust_wrapper.h* with additional constructor in line 15 . . . . . . . . . . . 66
5.17. *mailbox_rust_wrapper.cpp* with additional constructor in lines 9-12 . . . . . . . . 67
5.18. *participant_rust_wrapper.cpp* with updated constructor (lines 5 and 7) . . . . . . 67

5.19. *types.h* with the necessary type definitions and bindings . . . . . . . . . . . . . . 68
5.20. *types.rs* with the corresponding type and struct definitions . . . . . . . . . . . . 68
5.21. Rust implementation of MailboxRust (*mailbox_rust.rs*) . . . . . . . . . . . . . . 69
5.22. Rust implementation of ParticipantRust (*participant_rust.rs*) . . . . . . . . . . . 69
5.23. Simple *lib.rs* including the modules MailboxRust and ParticipantRust . . . . . . 70
5.24. Barebone *Cargo.toml* to compile the Rust component into a static library . . . . 70
5.25. C++ implementation of Mailbox (*mailbox.cpp*) . . . . . . . . . . . . . . . . . . 72
5.26. Shortened C++ implementation of the abstract class BaseParticipant (*base_participant.cpp*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.27. C++ implementation of Participant (*participant.cpp*) . . . . . . . . . . . . . . . . 73
5.28. An example main function showing the usage of both the C++ as well as the Rust components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.29. *CMakeLists.txt* file only containing the C++ files . . . . . . . . . . . . . . . . . 75
5.30. Updated *CMakeLists.txt* with CMakeRust call . . . . . . . . . . . . . . . . . . . 76
5.31. Final output of the communication example in Listing 5.28 . . . . . . . . . . . . 76

6.1. Example of Rust library code for an embedded system . . . . . . . . . . . . . . . 77
6.2. Console output after switching the default Rust compiler to nightly . . . . . . . 78

A.1. *CargoLink.cmake* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
A.2. *CMakeCargo.cmake* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
A.3. *CMakeDetermineRustCompiler.cmake* . . . . . . . . . . . . . . . . . . . . . . . 93
A.4. *CMakeRustCompiler.cmake.in* . . . . . . . . . . . . . . . . . . . . . . . . . . 94
A.5. *CMakeRustInformation.cmake* . . . . . . . . . . . . . . . . . . . . . . . . . . 95
A.6. *CMakeTestRustCompiler.cmake* . . . . . . . . . . . . . . . . . . . . . . . . . 97
A.7. *FindRust.cmake* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
A.8. Full C++ implementation of the abstract class BaseParticipant (*base_participant.cpp*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

# List of Figures

3.1. File structure of a simple Rust library project . . . . . . . . . . . . . . . . . . . 8
3.2. File structure of a simple Rust library project with compiled target . . . . . . . 9
3.3. File structure of a simple Rust program . . . . . . . . . . . . . . . . . . . . . . 9
3.4. File structure of a simple Rust binary project with compiled target . . . . . . . 10

4.1. File structure of a C++ project with a Rust component . . . . . . . . . . . . . . . 26
4.2. File structure of a Rust project with a C++ component . . . . . . . . . . . . . . . 35

5.1. Class diagram of the starting situation . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2. Directory tree showing the initial project setup . . . . . . . . . . . . . . . . . . . . . 49
5.3. Class diagram with abstract classes . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.4. Directory tree showing the project setup with abstract classes . . . . . . . . . . . 50
5.5. Bare class diagram with Rust components . . . . . . . . . . . . . . . . . . . . . . . 52
5.6. Class diagram with detailed interface description . . . . . . . . . . . . . . . . . . . 58
5.7. Updated class diagram with detailed interface description . . . . . . . . . . . . . . 71
5.8. File structure with first additions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

# List of Tables

4.1. Integer type comparison between Rust and C++ . . . . . . . . . . . . . . . . . . . . 40
4.2. Fixed Width Integer Type comparison between Rust and C++ . . . . . . . . . . . 40
4.3. Floating Point Type comparison between Rust and C++ . . . . . . . . . . . . . . . 41
4.4. Table with equivalent datatypes of C++ and Rust . . . . . . . . . . . . . . . . . . . 43

# Acronyms

**ABI** application binary interface

**FFI** foreign function interface

**IDE** integrated development environment

# Bibliography

## Rust


[And12] Brian Anderson. *The Rust compiler 0.1 is unleashed.* Jan. 2012. URL: https://mail.mozilla.org/pipermail/rust-dev/2012-January/001256.html (visited on 01/20/2022).

[Bin] *bindgen Rust Crate.* URL: https://docs.rs/bindgen/latest/bindgen/ (visited on 11/09/2023).

[Bor21] Nico Borgsmüller. "The Rust programming language for embedded software development". en. Ingolstadt, 2021, pp. 63, lxvii. URL: http://nbn-resolving.de/urn:nbn:de:bvb:573-7869.

[Cbi] *cbindgen Rust Crate.* URL: https://docs.rs/cbindgen/latest/cbindgen/ (visited on 11/09/2023).

[Ccc] *cc Rust Crate.* URL: https://docs.rs/cc/latest/cc/ (visited on 06/30/2023).

[Cha] *Primitive Type char 1.0.0.* URL: https://doc.rust-lang.org/std/primitive.char.html (visited on 05/15/2023).

[Cmab] *cmake Rust Crate.* URL: https://docs.rs/cmake/latest/cmake/ (visited on 08/08/2023).

[Cxxa] *cxx Rust Crate.* URL: https://docs.rs/cxx/latest/cxx/ (visited on 10/12/2023).

[Cxxb] *CXX — safe interop between Rust and C++.* URL: https://cxx.rs/ (visited on 12/12/2023).

[Dev23] Devolutions. *CMakeRust.* 2023. URL: https://github.com/Devolutions/CMakeRust (visited on 07/23/2023).

[Emb] *The Embedonomicon.* URL: https://docs.rust-embedded.org/embedonomicon/index.html (visited on 11/02/2023).

[F32] *Primitive Type f32 1.0.0.* URL: https://doc.rust-lang.org/std/primitive.f32.html (visited on 04/27/2023).

[F64] *Primitive Type f64 1.0.0.* URL: https://doc.rust-lang.org/std/primitive.f64.html (visited on 04/27/2023).

[Hos19] Diane Hosfelt. *Implications of Rewriting a Browser Component in Rust*. Feb. 2019. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/ (visited on 12/13/2023).

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 978-1-59327-828-1.

[Mem] *Rust Foundation - Membership*. URL: https://foundation.rust-lang.org/members/ (visited on 10/03/2023).

[Rusa] *Frequently Asked Questions*. URL: https://prev.rust-lang.org/en-US/faq.html (visited on 10/03/2023).

[Rusc] *Rust API Guidelines*. URL: https://rust-lang.github.io/api-guidelines/index.html (visited on 09/01/2023).

[Rusd] *Rust Impl Definition*. URL: https://doc.rust-lang.org/std/keyword.impl.html (visited on 07/12/2023).

[Rusf] *The Rustonomicon*. URL: https://docs.rust-embedded.org/rustonomicon/index.html (visited on 12/07/2023).

[Std] *Rust Standard Library FFI*. URL: https://doc.rust-lang.org/std/ffi/index.html (visited on 07/28/2023).

[Thea] *The Cargo Book*. URL: https://doc.rust-lang.org/cargo/index.html (visited on 08/08/2023).

[Theb] *The Embedded Rust Book*. URL: https://docs.rust-embedded.org/book/ (visited on 04/25/2023).

[Thec] *The Rust Reference*. URL: https://doc.rust-lang.org/reference/introduction.html (visited on 10/01/2023).

[Thed] *The rustc Book*. URL: https://doc.rust-lang.org/rustc/ (visited on 06/30/2023).

[Wil21] Ashley Williams. *Hello World!* Feb. 2021. URL: https://foundation.rust-lang.org/posts/2021-02-08-hello-world/ (visited on 10/03/2023).

## C++

[AUT19] AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*. 2019.

[Cmaa] *CMake*. URL: https://cmake.org (visited on 11/07/2023).

[Cmac] *Quick CMake tutorial*. URL: https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html (visited on 11/07/2023).

[Cpp] *C++ reference*. Apr. 2023. URL: https://en.cppreference.com/w/cpp (visited on 04/27/2023).

[fri23] friendlyanon. *cmake-init - The missing CMake project initializer*. 2023. URL: https://github.com/friendlyanon/cmake-init (visited on 11/07/2023).

[ISO20] ISO/IEC. *ISO 14882:2020 Programming languages - C++*. Tech. rep. International Organization for Standardization, Dec. 2020.

[Str96] Bjarne Stroustrup. "A History of C++: 1979–1991". In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, 699–769. ISBN: 0201895021. URL: https://doi.org/10.1145/234286.1057836 (visited on 10/25/2023).

# Other Sources

[Adaa] *AdaCore and Ferrous Systems Joining Forces to Support Rust*. Feb. 2, 2022. URL: https://blog.adacore.com/adacore-and-ferrous-systems-joining-forces-to-support-rust (visited on 10/09/2023).

[Adab] *Announcements around Rust*. July 19, 2023. URL: https://blog.adacore.com/adacore-has-two-announcements-about-rust (visited on 10/09/2023).

[Bar14] John Barnes. *Programming in ADA 2012*. 6th printing 2017. Cambridge University Press, 2014. ISBN: 978-1-107-42481-4.

[Rusb] *Microsoft is busy rewriting core Windows code in memory-safe Rust*. Apr. 27, 2023. URL: https://www.theregister.com/2023/04/27/microsoft_windows_rust/ (visited on 10/09/2023).

[Ruse] *Shipping Rust in Firefox*. July 12, 2016. URL: https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox (visited on 10/09/2023).

[Sig] *GitHub - signalapp/libsignal*. URL: \url{https://github.com/signalapp/libsignal} (visited on 10/09/2023).

[Sim22] Sergio De Simone. *Linux 6.1 Officially Adds Support for Rust in the Kernel*. Oct. 20, 2022. URL: https://www.infoq.com/news/2022/12/linux-6-1-rust/ (visited on 10/09/2023).

[Staa] *Stack Overflow Developer Survey 2022*. May 1, 2022. URL: https://survey.stackoverflow.co/2022/ (visited on 10/09/2023).

[Stab] *Stack Overflow Developer Survey 2023*. May 1, 2023. URL: https://survey.stackoverflow.co/2023/ (visited on 10/09/2023).

[Stac] *Stack Overflow Developer Surveys*. URL: https://insights.stackoverflow.com/survey (visited on 10/09/2023).

# A. Source Files

```
1
2  function(cargo_print)
3          execute_process(COMMAND ${CMAKE_COMMAND} -E echo "${ARGN}")
4  endfunction()
5
6  function(cargo_link)
7          cmake_parse_arguments(CARGO_LINK "" "NAME" "TARGETS;EXCLUDE" ${ARGN})
8
9          file(WRITE "${CMAKE_CURRENT_BINARY_DIR}/cargo-link.c" "void cargo_link() {}")
10         add_library(${CARGO_LINK_NAME} "${CMAKE_CURRENT_BINARY_DIR}/cargo-link.c")
11         target_link_libraries(${CARGO_LINK_NAME} ${CARGO_LINK_TARGETS})
12
13         get_target_property(LINK_LIBRARIES ${CARGO_LINK_NAME} LINK_LIBRARIES)
14
15         foreach(LINK_LIBRARY ${LINK_LIBRARIES})
16                 get_target_property(_INTERFACE_LINK_LIBRARIES ${LINK_LIBRARY}
       INTERFACE_LINK_LIBRARIES)
17                 list(APPEND LINK_LIBRARIES ${_INTERFACE_LINK_LIBRARIES})
18         endforeach()
19         list(REMOVE_DUPLICATES LINK_LIBRARIES)
20
21         if(CARGO_LINK_EXCLUDE)
22                 list(REMOVE_ITEM LINK_LIBRARIES ${CARGO_LINK_EXCLUDE})
23         endif()
24
25         set(LINK_DIRECTORIES "")
26         foreach(LINK_LIBRARY ${LINK_LIBRARIES})
27                 if(TARGET ${LINK_LIBRARY})
28                         get_target_property(_IMPORTED_CONFIGURATIONS ${LINK_LIBRARY}
       IMPORTED_CONFIGURATIONS)
29                         list(FIND _IMPORTED_CONFIGURATIONS "RELEASE"
       _IMPORTED_CONFIGURATION_INDEX)
30                         if (NOT (${_IMPORTED_CONFIGURATION_INDEX} GREATER -1))
31                                 set(_IMPORTED_CONFIGURATION_INDEX 0)
32                         endif()
33                         list(GET _IMPORTED_CONFIGURATIONS ${_IMPORTED_CONFIGURATION_INDEX}
       _IMPORTED_CONFIGURATION)
34                         get_target_property(_IMPORTED_LOCATION ${LINK_LIBRARY} "
       IMPORTED_LOCATION_${_IMPORTED_CONFIGURATION}")
35                         get_filename_component(_IMPORTED_DIR ${_IMPORTED_LOCATION} DIRECTORY
       )
36                         get_filename_component(_IMPORTED_NAME ${_IMPORTED_LOCATION} NAME_WE)
37                         if(NOT WIN32)
38                                 string(REGEX REPLACE "^lib" "" _IMPORTED_NAME ${
       _IMPORTED_NAME})
39                         endif()
```

```
40                         list(APPEND LINK_DIRECTORIES ${_IMPORTED_DIR})
41                         cargo_print("cargo:rustc-link-lib=static=${_IMPORTED_NAME}")
42                 else()
43                         if("${LINK_LIBRARY}" MATCHES "^.*/(.+)\\.framework$")
44                                 set(FRAMEWORK_NAME ${CMAKE_MATCH_1})
45                                 cargo_print("cargo:rustc-link-lib=framework=${FRAMEWORK_NAME
    }")
46                         elseif("${LINK_LIBRARY}" MATCHES "^(.*)/(.+)\\.so$")
47                                 set(LIBRARY_DIR ${CMAKE_MATCH_1})
48                                 set(LIBRARY_NAME ${CMAKE_MATCH_2})
49                                 if(NOT WIN32)
50                                         string(REGEX REPLACE "^lib" "" LIBRARY_NAME ${
    LIBRARY_NAME})
51                                 endif()
52                                 list(APPEND LINK_DIRECTORIES ${LIBRARY_DIR})
53                                 cargo_print("cargo:rustc-link-lib=${LIBRARY_NAME}")
54                         else()
55                                 cargo_print("cargo:rustc-link-lib=${LINK_LIBRARY}")
56                         endif()
57                 endif()
58         endforeach()
59         list(REMOVE_DUPLICATES LINK_DIRECTORIES)
60 
61         foreach(LINK_DIRECTORY ${LINK_DIRECTORIES})
62                 cargo_print("cargo:rustc-link-search=native=${LINK_DIRECTORY}")
63         endforeach()
64 endfunction()
```

Listing A.1: *CargoLink.cmake*

```
function(cargo_build)
    cmake_parse_arguments(CARGO "" "NAME" "" ${ARGN})
    string(REPLACE "-" "_" LIB_NAME ${CARGO_NAME})

    set(CARGO_TARGET_DIR ${CMAKE_CURRENT_BINARY_DIR})

    if(WIN32)
        if(CMAKE_SIZEOF_VOID_P EQUAL 8)
            set(LIB_TARGET "x86_64-pc-windows-msvc")
        else()
            set(LIB_TARGET "i686-pc-windows-msvc")
        endif()
        elseif(ANDROID)
        if(ANDROID_SYSROOT_ABI STREQUAL "x86")
            set(LIB_TARGET "i686-linux-android")
        elseif(ANDROID_SYSROOT_ABI STREQUAL "x86_64")
            set(LIB_TARGET "x86_64-linux-android")
        elseif(ANDROID_SYSROOT_ABI STREQUAL "arm")
            set(LIB_TARGET "arm-linux-androideabi")
        elseif(ANDROID_SYSROOT_ABI STREQUAL "arm64")
            set(LIB_TARGET "aarch64-linux-android")
        endif()
    elseif(IOS)
                set(LIB_TARGET "universal")
    elseif(CMAKE_SYSTEM_NAME STREQUAL Darwin)
        set(LIB_TARGET "x86_64-apple-darwin")
        else()
        if(CMAKE_SIZEOF_VOID_P EQUAL 8)
            set(LIB_TARGET "x86_64-unknown-linux-gnu")
        else()
            set(LIB_TARGET "i686-unknown-linux-gnu")
        endif()
    endif()

    if(NOT CMAKE_BUILD_TYPE)
        set(LIB_BUILD_TYPE "debug")
    elseif(${CMAKE_BUILD_TYPE} STREQUAL "Release")
        set(LIB_BUILD_TYPE "release")
    else()
        set(LIB_BUILD_TYPE "debug")
    endif()

    set(LIB_FILE "${CARGO_TARGET_DIR}/${LIB_TARGET}/${LIB_BUILD_TYPE}/${
    CMAKE_STATIC_LIBRARY_PREFIX}${LIB_NAME}${CMAKE_STATIC_LIBRARY_SUFFIX}")

        if(IOS)
                set(CARGO_ARGS "lipo")
        else()
        set(CARGO_ARGS "build")
                list(APPEND CARGO_ARGS "--target" ${LIB_TARGET})
        endif()

    if(${LIB_BUILD_TYPE} STREQUAL "release")
        list(APPEND CARGO_ARGS "--release")
    endif()

```

```
56     file(GLOB_RECURSE LIB_SOURCES "*.rs")
57
58     set(CARGO_ENV_COMMAND ${CMAKE_COMMAND} -E env "CARGO_TARGET_DIR=${CARGO_TARGET_DIR}")
59
60     add_custom_command(
61         OUTPUT ${LIB_FILE}
62         COMMAND ${CARGO_ENV_COMMAND} ${CARGO_EXECUTABLE} ARGS ${CARGO_ARGS}
63         WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
64         DEPENDS ${LIB_SOURCES}
65         COMMENT "running cargo")
66     add_custom_target(${CARGO_NAME}_target ALL DEPENDS ${LIB_FILE})
67     add_library(${CARGO_NAME} STATIC IMPORTED GLOBAL)
68     add_dependencies(${CARGO_NAME} ${CARGO_NAME}_target)
69     set_target_properties(${CARGO_NAME} PROPERTIES IMPORTED_LOCATION ${LIB_FILE})
70 endfunction()
```

Listing A.2: *CMakeCargo.cmake*

```
1
2  if(NOT CMAKE_Rust_COMPILER)
3          find_package(Rust)
4          if(RUST_FOUND)
5                  set(CMAKE_Rust_COMPILER "${RUSTC_EXECUTABLE}")
6                  set(CMAKE_Rust_COMPILER_ID "Rust")
7                  set(CMAKE_Rust_COMPILER_VERSION "${RUST_VERSION}")
8                  set(CMAKE_Rust_PLATFORM_ID "Rust")
9          endif()
10 endif()
11
12 message(STATUS "Cargo Home: ${CARGO_HOME}")
13 message(STATUS "Rust Compiler Version: ${RUSTC_VERSION}")
14
15 mark_as_advanced(CMAKE_Rust_COMPILER)
16
17 if(CMAKE_Rust_COMPILER)
18         set(CMAKE_Rust_COMPILER_LOADED 1)
19 endif(CMAKE_Rust_COMPILER)
20
21 configure_file(${CMAKE_CURRENT_LIST_DIR}/CMakeRustCompiler.cmake.in
22         ${CMAKE_BINARY_DIR}${CMAKE_FILES_DIRECTORY}/${CMAKE_VERSION}/CMakeRustCompiler.cmake
       IMMEDIATE @ONLY)
23
24 set(CMAKE_Rust_COMPILER_ENV_VAR "RUSTC")
```

Listing A.3: *CMakeDetermineRustCompiler.cmake*

```
set(CMAKE_Rust_COMPILER "@CMAKE_Rust_COMPILER@")
set(CMAKE_Rust_COMPILER_ID "@CMAKE_Rust_COMPILER_ID@")
set(CMAKE_Rust_COMPILER_VERSION "@CMAKE_Rust_COMPILER_VERSION@")
set(CMAKE_Rust_COMPILER_LOADED @CMAKE_Rust_COMPILER_LOADED@)
set(CMAKE_Rust_PLATFORM_ID "@CMAKE_Rust_PLATFORM_ID@")

SET(CMAKE_Rust_SOURCE_FILE_EXTENSIONS rs)
SET(CMAKE_Rust_LINKER_PREFERENCE 40)
#SET(CMAKE_Rust_OUTPUT_EXTENSION_REPLACE 1)
SET(CMAKE_STATIC_LIBRARY_PREFIX_Rust "")
SET(CMAKE_STATIC_LIBRARY_SUFFIX_Rust .a)

set(CMAKE_Rust_COMPILER_ENV_VAR "RUSTC")
```

Listing A.4: *CMakeRustCompiler.cmake.in*

```
#
# Usage: rustc [OPTIONS] INPUT
#
# Options:
#     -h --help           Display this message
#     --cfg SPEC          Configure the compilation environment
#     -L [KIND=]PATH      Add a directory to the library search path. The
#                         optional KIND can be one of dependency, crate, native,
#                         framework or all (the default).
#     -l [KIND=]NAME      Link the generated crate(s) to the specified native
#                         library NAME. The optional KIND can be one of static,
#                         dylib, or framework. If omitted, dylib is assumed.
#     --crate-type [bin|lib|rlib|dylib|cdylib|staticlib|metadata]
#                         Comma separated list of types of crates for the
#                         compiler to emit
#     --crate-name NAME   Specify the name of the crate being built
#     --emit [asm|llvm-bc|llvm-ir|obj|link|dep-info]
#                         Comma separated list of types of output for the
#                         compiler to emit
#     --print [crate-name|file-names|sysroot|cfg|target-list|target-cpus|target-features|
    relocation-models|code-models]
#                         Comma separated list of compiler information to print
#                         on stdout
#     -g                  Equivalent to -C debuginfo=2
#     -O                  Equivalent to -C opt-level=2
#     -o FILENAME         Write output to <filename>
#     --out-dir DIR       Write output to compiler-chosen filename in <dir>
#     --explain OPT       Provide a detailed explanation of an error message
#     --test              Build a test harness
#     --target TARGET     Target triple for which the code is compiled
#     -W --warn OPT       Set lint warnings
#     -A --allow OPT      Set lint allowed
#     -D --deny OPT       Set lint denied
#     -F --forbid OPT     Set lint forbidden
#     --cap-lints LEVEL   Set the most restrictive lint level. More restrictive
#                         lints are capped at this level
#     -C --codegen OPT[=VALUE]
#                         Set a codegen option
#     -V --version        Print version info and exit
#     -v --verbose        Use verbose output
#
# Additional help:
#     -C help             Print codegen options
#     -W help             Print 'lint' options and default settings
#     -Z help             Print internal options for debugging rustc
#     --help -v           Print the full set of options rustc accepts
#

# <TARGET> <TARGET_BASE> <OBJECT> <OBJECTS> <LINK_LIBRARIES> <FLAGS> <LINK_FLAGS> <SOURCE> <
    SOURCES>

include(CMakeLanguageInformation)

if(UNIX)
        set(CMAKE_Rust_OUTPUT_EXTENSION .o)
```

```
55 else()
56         set(CMAKE_Rust_OUTPUT_EXTENSION .obj)
57 endif()
58 
59 set(CMAKE_Rust_ECHO_ALL "echo \"TARGET: <TARGET> TARGET_BASE: <TARGET_BASE> ")
60 set(CMAKE_Rust_ECHO_ALL "${CMAKE_Rust_ECHO_ALL} OBJECT: <OBJECT> OBJECTS: <OBJECTS>
    OBJECT_DIR: <OBJECT_DIR> SOURCE: <SOURCE> SOURCES: <SOURCES> ")
61 set(CMAKE_Rust_ECHO_ALL "${CMAKE_Rust_ECHO_ALL} LINK_LIBRARIES: <LINK_LIBRARIES> FLAGS: <
    FLAGS> LINK_FLAGS: <LINK_FLAGS> \"")
62 
63 if(NOT CMAKE_Rust_CREATE_SHARED_LIBRARY)
64         set(CMAKE_Rust_CREATE_SHARED_LIBRARY
65                 "echo \"CMAKE_Rust_CREATE_SHARED_LIBRARY\""
66                 "${CMAKE_Rust_ECHO_ALL}"
67                 )
68 endif()
69 
70 if(NOT CMAKE_Rust_CREATE_SHARED_MODULE)
71         set(CMAKE_Rust_CREATE_SHARED_MODULE
72                 "echo \"CMAKE_Rust_CREATE_SHARED_MODULE\""
73                 "${CMAKE_Rust_ECHO_ALL}"
74                 )
75 endif()
76 
77 if(NOT CMAKE_Rust_CREATE_STATIC_LIBRARY)
78         set(CMAKE_Rust_CREATE_STATIC_LIBRARY
79                 "echo \"CMAKE_Rust_CREATE_STATIC_LIBRARY\""
80                 "${CMAKE_Rust_ECHO_ALL}"
81                 )
82 endif()
83 
84 if(NOT CMAKE_Rust_COMPILE_OBJECT)
85         set(CMAKE_Rust_COMPILE_OBJECT
86                 "echo \"CMAKE_Rust_COMPILE_OBJECT\""
87                 "${CMAKE_Rust_ECHO_ALL}"
88                 "${CMAKE_Rust_COMPILER} --emit obj <SOURCE> -o <OBJECT>")
89 endif()
90 
91 if(NOT CMAKE_Rust_LINK_EXECUTABLE)
92         set(CMAKE_Rust_LINK_EXECUTABLE
93                 "echo \"CMAKE_Rust_LINK_EXECUTABLE\""
94                 "${CMAKE_Rust_ECHO_ALL}"
95                 )
96 endif()
97 
98 mark_as_advanced(
99         CMAKE_Rust_FLAGS
100        CMAKE_Rust_FLAGS_DEBUG
101        CMAKE_Rust_FLAGS_MINSIZEREL
102        CMAKE_Rust_FLAGS_RELEASE
103        CMAKE_Rust_FLAGS_RELWITHDEBINFO)
104 
105 set(CMAKE_Rust_INFORMATION_LOADED 1)
```

Listing A.5: *CMakeRustInformation.cmake*

```

set(CMAKE_Rust_COMPILER_WORKS 1 CACHE INTERNAL "")
```

Listing A.6: *CMakeTestRustCompiler.cmake*

```

set(_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM ${CMAKE_FIND_ROOT_PATH_MODE_PROGRAM})
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM BOTH)
set(_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ${CMAKE_FIND_ROOT_PATH_MODE_INCLUDE})
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE BOTH)

if(CMAKE_HOST_WIN32)
        set(USER_HOME "$ENV{USERPROFILE}")
else()
        set(USER_HOME "$ENV{HOME}")
endif()

if(NOT DEFINED CARGO_HOME)
        if("$ENV{CARGO_HOME}" STREQUAL "")
                set(CARGO_HOME "${USER_HOME}/.cargo")
        else()
                set(CARGO_HOME "$ENV{CARGO_HOME}")
        endif()
endif()

# Find cargo executable
find_program(CARGO_EXECUTABLE cargo
        HINTS "${CARGO_HOME}"
        PATH_SUFFIXES "bin")
mark_as_advanced(CARGO_EXECUTABLE)

# Find rustc executable
find_program(RUSTC_EXECUTABLE rustc
        HINTS "${CARGO_HOME}"
        PATH_SUFFIXES "bin")
mark_as_advanced(RUSTC_EXECUTABLE)

# Find rustdoc executable
find_program(RUSTDOC_EXECUTABLE rustdoc
        HINTS "${CARGO_HOME}"
        PATH_SUFFIXES "bin")
mark_as_advanced(RUSTDOC_EXECUTABLE)

# Find rust-gdb executable
find_program(RUST_GDB_EXECUTABLE rust-gdb
        HINTS "${CARGO_HOME}"
        PATH_SUFFIXES "bin")
mark_as_advanced(RUST_GDB_EXECUTABLE)

# Find rust-lldb executable
find_program(RUST_LLDB_EXECUTABLE rust-lldb
        HINTS "${CARGO_HOME}"
        PATH_SUFFIXES "bin")
mark_as_advanced(RUST_LLDB_EXECUTABLE)

```

```
51 # Find rustup executable
52 find_program(RUSTUP_EXECUTABLE rustup
53         HINTS "${CARGO_HOME}"
54         PATH_SUFFIXES "bin")
55 mark_as_advanced(RUSTUP_EXECUTABLE)
56
57 set(RUST_FOUND FALSE CACHE INTERNAL "")
58
59 if(CARGO_EXECUTABLE AND RUSTC_EXECUTABLE AND RUSTDOC_EXECUTABLE)
60         set(RUST_FOUND TRUE CACHE INTERNAL "")
61
62         set(CARGO_HOME "${CARGO_HOME}" CACHE PATH "Rust Cargo Home")
63
64         execute_process(COMMAND ${RUSTC_EXECUTABLE} --version OUTPUT_VARIABLE RUSTC_VERSION
       OUTPUT_STRIP_TRAILING_WHITESPACE)
65         string(REGEX REPLACE "rustc ([^ ]+) .*" "\\1" RUSTC_VERSION "${RUSTC_VERSION}")
66 endif()
67
68 if(NOT RUST_FOUND)
69         message(FATAL_ERROR "Could not find Rust!")
70 endif()
71
72 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM ${_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM})
73 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ${_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE})
```

Listing A.7: *FindRust.cmake*

```
1 #include "base_participant.h"
2
3 BaseParticipant::BaseParticipant(participant_id_t id)
4 {
5     this->id = id;
6 }
7
8 BaseMailbox *BaseParticipant::get_mailbox() { return mailbox; }
9 participant_id_t BaseParticipant::get_id() { return id; }
10
11 void BaseParticipant::receive_message()
12 {
13     if (this->mailbox->get_size() >= 1)
14     {
15         Message msg = this->mailbox->get_next_message();
16         print_message(msg);
17     }
18 }
19
20 void BaseParticipant::poll_every_second()
21 {
22     t = new std::thread([&]()
23     {
24         int counter = 0;
25
26         while (counter < 10)
27         {
28             this->receive_message();
```

```cpp
                counter += 1;
                usleep(1000000);
            }
        });
}

void BaseParticipant::wait()
{
    if (t != nullptr)
    {
        t->join();
    }
}
```

Listing A.8: Full C++ implementation of the abstract class BaseParticipant (*base_participant.cpp*)