CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The viability of Rust as a real-time system language

A comparative analysis between Rust and C for real-time systems

Master's thesis in Computer science and engineering

Simon Johansson

Emrik Lindahl

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

The viability of Rust as a real-time system language

A comparative analysis between Rust and C for real-time systems

Simon Johansson
Emrik Lindahl

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

The viability of Rust as a real-time system language
A comparative analysis between Rust and C for real-time systems
Simon Johansson
Emrik Lindahl

© Simon Johansson, 2024.
© Emrik Lindahl, 2024.

Supervisor: Magnus Almgren, Computer Science and Engineering
Advisor: Benhur Tessele, Satcube
Examiner: Magnus Almgren, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

The viability of Rust as a real-time system language
A comparative analysis between Rust and C for real-time systems
Simon Johansson
Emrik Lindahl
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Real-time systems are used in many crucial applications in our society, for example, airplanes and satellites. These systems need to be secure and reliable since they are often time-crucial and safety-critical. Security in this aspect means that there are no unforeseen crashes or undefined behavior of the system.

Historically, these systems have been created in C due to the performance C can achieve along with the ability to manipulate hardware at a low level. A performant system is important to ensure that the system can handle the workload put on it, however, it is not trivial to combine the aspects of performance and security. This is because security often comes at the expense of performance since additional checks are needed. A new emerging language, Rust, tries to solve this problem.

This thesis will extend the work done by Tjäder [1] on the framework RTIC by testing if Rust is viable in a real-time setting. To achieve this aim, real-time applications in both Rust and C are created, which are then run on different hardware. The purpose is to test the applications with metrics such as performance, memory usage, and reaction time for events. This will be used to evaluate the viability of Rust for real-time systems to make them more secure and reliable.

The thesis concludes that although Rust offers security in terms of memory management, Rust can compete with C in aspects such as performance. It all comes down to different tradeoffs, from what is needed in the specific system to be built.

Keywords: Rust, C, Real-time systems, FreeRTOS, RTIC.

Acknowledgements

We would like to thank our supervisor at Satcube, Benhur Tessele, for his invaluable advice during the thesis as well as Martin Löfgren at Satcube for his support. We would also like to thank Satcube for the loaned equipment that allowed us to conduct our tests. Last but not least, we would like to thank our supervisor as well as examiner at Chalmers, Magnus Almgren, for his invaluable advice and insights.

Simon Johansson, Gothenburg, 2024-08-26
Emrik Lindahl, Gothenburg, 2024-08-26

Contents

Acronyms **xiii**

List of Figures **xv**

List of Tables **xix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aim . . . 2
1.3 Limitations . . . 3
1.4 Outline . . . 3

2 Related work **5**
2.1 Memory safety . . . 5
2.2 Real-time systems . . . 6
2.3 Safety of Rust . . . 6

3 Background **9**
3.1 Real-time system concepts . . . 9
3.1.1 Tasks . . . 9
3.1.2 Scheduling algorithms . . . 10
3.1.3 Priorities . . . 10
3.2 The C programming language . . . 12
3.2.1 Memory in the C language . . . 12
3.2.2 Dynamic allocation of memory . . . 14
3.2.3 Security aspects . . . 14
3.3 The Rust programming language . . . 15
3.3.1 Mutability . . . 15
3.3.2 Ownership . . . 15
3.3.3 Borrowing . . . 17
3.3.4 Other security aspects . . . 20
3.3.5 Unsafe Rust . . . 20
3.4 The real-time systems . . . 20
3.4.1 FreeRTOS . . . 20
3.4.1.1 Scheduling tasks and timing . . . 21
3.4.1.2 Resources and synchronisation . . . 21

3.4.1.3 Interrupts 21
3.4.2 RTIC 22
3.4.2.1 Scheduling tasks and timing 22
3.4.2.2 Resources and synchronisation 22
3.4.2.3 Interrupts 23

4 Method 25
4.1 Methodology 25
4.2 Motivation of chosen programming languages 25
4.3 Motivation of chosen real-time operating systems 26
4.3.1 Requirements 26
4.3.2 C RTOS: FreeRTOS 27
4.3.3 Rust RTOS: RTIC 27
4.4 Choice of hardware 27
4.4.1 Requirements 28
4.4.2 Hardware 28
4.4.2.1 STM32F0Discovery 29
4.4.2.2 MD407 29
4.4.2.3 Raspberry Pi Pico 29
4.5 Applications 29
4.5.1 Minimal application 30
4.5.2 Reaction time 30
4.5.3 Matrix multiplication 32
4.5.4 Tone generator 33
4.5.5 Concurrency 34
4.6 Metrics 35
4.6.1 Size of binary 35
4.6.2 Compile time 36
4.6.3 Memory usage 36
4.6.4 Application specific metrics 37
4.6.4.1 Reaction time 37
4.6.4.2 Matrix multiplication 37
4.6.4.3 Tone generator 38
4.6.4.4 Concurrency 39
4.7 Conduction of tests 39
4.7.1 Size of binary 39
4.7.2 Compile time 39
4.7.3 Memory usage 39
4.7.4 Applications 40

5 Implementation 41
5.1 General tools and analysis methods 41
5.2 C tools and general implementations 42
5.2.1 Compiler and buildsystem 42
5.2.2 Libraries 42
5.2.3 Running FreeRTOS 42
5.2.4 Flashing and debugging 43

5.2.5 Analysis . . . 44
5.3 Rust tools and general implementations . . . 44
5.3.1 Compiler and build system . . . 44
5.3.2 Libraries . . . 45
5.3.3 Running RTIC . . . 45
5.3.4 Flashing and debugging . . . 46
5.3.5 Analysis . . . 46
5.3.6 Creating tasks . . . 46
5.4 Applications . . . 47
5.4.1 Reaction time . . . 47
5.4.2 Matrix multiplication . . . 48
5.4.3 Concurrency . . . 48
5.4.4 Tone generator . . . 49

6 Results **51**
6.1 Size of binary . . . 51
6.2 Compile time . . . 53
6.3 Memory usage . . . 53
6.4 Application specific metrics . . . 54
6.4.1 Concurrency . . . 55
6.4.2 Tone generator . . . 55

7 Discussion **63**
7.1 Interpretation of the results . . . 63
7.1.1 Binary sizes . . . 63
7.1.2 Memory usage . . . 64
7.1.3 Compilation time . . . 64
7.1.4 Application specific metrics . . . 64
7.2 Anecdotes from development . . . 65
7.3 The implications of a restrictive language . . . 66
7.4 Other RTOS alternatives in Rust . . . 67
7.4.1 Embassy . . . 67
7.4.2 Tock . . . 68
7.4.3 Wrappers for C RTOSs . . . 68
7.5 The future of Rust in real-time systems . . . 68
7.6 Ethical and sustainability consideration . . . 69
7.7 Future work . . . 70

8 Conclusion **71**

Bibliography **73**

A Appendix FFT FreeRTOS **I**

B Appendix FFT RTIC **IX**

Glossary

CPU Central Processing Unit.

CVE Common vulnerability and exposures.

DAC Digital to Analog Converter.

EXTI External interrupts.

FFT Fast Fourier Transform.

FPU Floating Point Unit.

GPIO General Purpose Input Output.

HAL Hardware Abstraction Layer.

LED Light-emitting diode.

LIFO Last In, First Out.

NVIC Nested Vector Interrupt Control.

PAC Peripheral Access Crate.

PCP Priority Ceiling Protocol.

RTOS Real-time operating system.

SDK Software Development Kit.

SRP Stack Resource Policy.

USART Universal Synchronous/Asynchronous Receiver/Transmitter.

VTOR Vector Table Offset Register.

WCET Worst-Case Execution Time.

List of Figures

3.1 A state diagram of the different task states. A task in the *running* state can either become *ready* or *blocked*. A *blocked* task can only become *ready*, and a task in the *ready* state can only become *running*. 10
3.2 The priority inversion example. The y-axis represents the different tasks. If a task is running, then the line connected to each task is raised. The x-axis represents different time instances. The `L` task starts executing, and then gets preempted by M_1 and M_2. Then, when `L` has finished its execution, `H` can execute. As can be observed, the `H` task never runs until the end even though it has the highest priority. 11
3.3 The flow of execution with priority inheritance. Neither M_1 nor M_2 are allowed to preempt `L`, since `L` has the priority of `H`. When `L`'s execution has finished, `H` will execute fully. Then, M_1 and M_2 are allowed to execute. 12
3.4 A typical memory layout of a process. The stack and heap use the same memory area. All other data, for example, the text segment, lives after the heap. 13
3.5 Ambiguous execution. It is not apparent whether the `++` operation is executed before or after the evaluation of the boolean expression in the `if` statement. 14
3.6 A mutable variable in Rust. 15
3.7 An example of code that will panic due to the ownership rules of Rust. 16
3.8 Representation of a variable `s1` in memory. It has a `ptr` field that points to the data of the string, the characters themselves. The `len` field holds the actual length of the string, while the `capacity` field holds the capacity of the data type, i.e. the maximum number of characters allowed in the string. This picture is copied from the Rust book [31]. 17
3.9 Different methods of handling the situation in which a variable `s1` is assigned another variable `s2`. 18
3.10 This code will not panic, because of the cloning of `s1`. This means that it is allowed to use `s1` after it has been cloned to `s2`. 19
3.11 This code is an example of transferring ownership. The ownership of `transfer` is transferred from `main` to `another_function`, causing the function to panic. 19
3.12 This code is an example of borrowing. The function `another_function` borrows the variable `transfer` from `main`. 19

4.1 A flowchart describing the execution of the *Reaction time* application. 31
4.2 A flowchart describing the interrupt handler used in the *Reaction time* application. 31
4.3 A flowchart describing the execution of a background task used in the *Reaction time* application. 31
4.4 A flowchart describing the execution of the *Matrix multiplication* application. 33
4.5 A flowchart describing the execution of the tasks created by the *Matrix multiplication* application. 33
4.6 A flowchart over the *Tone generator* application. 34
4.7 A flowchart over the background task used in the *Tone generator* application. 34
4.8 A flowchart describing the execution of the task that writes to the DAC and creates the tone. 34
4.9 Flowcharts describing the execution of the tasks used for the *Concurrency* applications. 35
4.10 The function for incrementing the shared variable for the *Concurrency* application. 36
4.11 An abstraction of jitter between an interrupt and the execution of a task. 37
4.12 Flowchart over a context switch from process P_0 to P_1 and back. 38

6.1 FFT of the output signal when using 50 background tasks in FreeRTOS. 56
6.2 FFT of the output signal when using 190 background tasks in FreeRTOS. 60
6.3 FFT of the output signal when using 50 background tasks in RTIC. . 61
6.4 FFT of the output signal when using 190 background tasks in RTIC. 61

A.1 FFT of the output signal when using no background tasks in FreeRTOS. I
A.2 FFT of the output signal when using 10 background tasks in FreeRTOS. I
A.3 FFT of the output signal when using 20 background tasks in FreeRTOS. II
A.4 FFT of the output signal when using 30 background tasks in FreeRTOS. II
A.5 FFT of the output signal when using 40 background tasks in FreeRTOS. II
A.6 FFT of the output signal when using 50 background tasks in FreeRTOS. III
A.7 FFT of the output signal when using 60 background tasks in FreeRTOS. III
A.8 FFT of the output signal when using 70 background tasks in FreeRTOS. III
A.9 FFT of the output signal when using 80 background tasks in FreeRTOS. IV
A.10 FFT of the output signal when using 90 background tasks in FreeRTOS. IV
A.11 FFT of the output signal when using 100 background tasks in FreeRTOS. IV
A.12 FFT of the output signal when using 110 background tasks in FreeRTOS. V
A.13 FFT of the output signal when using 120 background tasks in FreeRTOS. V
A.14 FFT of the output signal when using 130 background tasks in FreeRTOS. V
A.15 FFT of the output signal when using 140 background tasks in FreeRTOS. VI
A.16 FFT of the output signal when using 150 background tasks in FreeRTOS. VI
A.17 FFT of the output signal when using 160 background tasks in FreeRTOS. VI
A.18 FFT of the output signal when using 170 background tasks in FreeRTOS. VII
A.19 FFT of the output signal when using 180 background tasks in FreeRTOS. VII
A.20 FFT of the output signal when using 190 background tasks in FreeRTOS. VII

B.1 FFT of the output signal when using no background tasks in RTIC. . IX
B.2 FFT of the output signal when using 10 background tasks in RTIC. . IX
B.3 FFT of the output signal when using 20 background tasks in RTIC. . X
B.4 FFT of the output signal when using 30 background tasks in RTIC. . X
B.5 FFT of the output signal when using 40 background tasks in RTIC. . X
B.6 FFT of the output signal when using 50 background tasks in RTIC. . XI
B.7 FFT of the output signal when using 60 background tasks in RTIC. . XI
B.8 FFT of the output signal when using 70 background tasks in RTIC. . XI
B.9 FFT of the output signal when using 80 background tasks in RTIC. . XII
B.10 FFT of the output signal when using 90 background tasks in RTIC. . XII
B.11 FFT of the output signal when using 100 background tasks in RTIC. XII
B.12 FFT of the output signal when using 110 background tasks in RTIC. XIII
B.13 FFT of the output signal when using 120 background tasks in RTIC. XIII
B.14 FFT of the output signal when using 130 background tasks in RTIC. XIII
B.15 FFT of the output signal when using 140 background tasks in RTIC. XIV
B.16 FFT of the output signal when using 150 background tasks in RTIC. XIV
B.17 FFT of the output signal when using 160 background tasks in RTIC. XIV
B.18 FFT of the output signal when using 170 background tasks in RTIC. XV
B.19 FFT of the output signal when using 180 background tasks in RTIC. XV
B.20 FFT of the output signal when using 190 background tasks in RTIC. XV
B.21 FFT of the output signal when using 200 background tasks in RTIC. XVI

List of Tables

4.1 The hardware chosen for the thesis. 29

6.1 Binary sizes for *Matrix multiplication* on MD407. The values are given in bytes. 51
6.2 Binary sizes for *Matrix multiplication* on Discovery. The values are given in bytes. 51
6.3 Binary sizes for *Matrix multiplication* on Pico. The values are given in bytes. 52
6.4 Binary sizes for *Concurrency* on MD407. The values are given in bytes. 52
6.5 Binary sizes for *Concurrency* on Discovery. The values are given in bytes. 52
6.6 Binary sizes for *Concurrency* on Pico. The values are given in bytes. . 52
6.7 Binary sizes for *Reaction time* on MD407. The values are given in bytes. 53
6.8 Binary sizes for *Reaction time* on Discovery. The values are given in bytes. 53
6.9 Binary sizes for *Reaction time* on Pico. The values are given in bytes. 53
6.10 The compile times for the applications in C. The values are in seconds. 53
6.11 The compile times for the applications in Rust. All the values are in seconds. The *v2* rows are the ones where RTIC version 2 was used instead of RTIC version 1. 54
6.12 Memory usage in the *Matrix multiplication* application on the Discovery microcontroller. The values are in bytes. 54
6.13 Memory usage in the *Matrix multiplication* application on the MD407 microcontroller using floats. The values are in bytes. 54
6.14 Memory usage in the *Matrix multiplication* application on the MD407 microcontroller using doubles. The values are in bytes. 55
6.15 Memory usage in the *Matrix multiplication* application on the Pico microcontroller. The values are in bytes. 55
6.16 Memory usage in the *Concurrency* application on the Discovery microcontroller. The values are in bytes. 55
6.17 Memory usage in the *Concurrency* application on the MD407 microcontroller. The values are in bytes. 56
6.18 Memory usage in the *Concurrency* application on the Pico microcontroller. The values are in bytes. 56
6.19 Memory usage in the *Reaction time* application on the Discovery microcontroller. The values are in bytes. 57

6.20 Memory usage in the *Reaction time* application on the Pico microcontroller. The values are in bytes. 57
6.21 Memory usage in the *Reaction time* application on the MD407 microcontroller. The values are in bytes. 57
6.22 The average measured time for the *Reaction time* application on the MD407 microcontroller. The values are in microseconds. 57
6.23 The average measured time for the *Reaction time* application on the Discovery microcontroller. The values are in microseconds. 57
6.24 The average measured time for the *Reaction time* application on the Pico microcontroller. The values are in microseconds. 58
6.25 The average measured run time for the *Matrix multiplication* application on the Discovery. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 58
6.26 The average measured run time for the *Matrix multiplication* application on the MD407 using doubles. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 58
6.27 The average measured run time for the *Matrix multiplication* application on the MD407 using floats. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 58
6.28 The average measured run time for the *Matrix multiplication* application on the Pico with RTICv1. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 59
6.29 The average measured run time for the *Matrix multiplication* application on the Pico with RTICv1. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 59
6.30 The averaged measured time for the *Concurrency* application on the Discovery. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 59
6.31 The averaged measured time for the *Concurrency* application on the MD407. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 59
6.32 The averaged measured time for the *Concurrency* application on the Pico with RTICv1 compared to FreeRTOS. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 60
6.33 The averaged measured time for the *Concurrency* application on the Pico with RTICv2 compared to FreeRTOS. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit. 60

1

Introduction

Real-time systems are widely used in our society [2], [3]. Everything from cars to the microwave in a kitchen is running some real-time system, that ensures the device works to certain expectations. These real-time systems are realized using tasks that may have deadlines for when they should be completed. For example, activating the brakes in a car could be a task and one would want the brakes to activate rather quickly once one presses down the brake pedal. But there might be other tasks doing other computations that take processing power at the same time, which might make it unfeasible to have the brakes activated immediately. By introducing time constraints on such tasks, one introduces a mechanism to try to guarantee that a task finishes its execution within a reasonable time. It is also crucial that there are no software bugs that could prevent the activation of the brakes. This is where a new language called Rust could aid, which promises memory security at compile time.

This thesis will research whether Rust can be a viable real-time systems language, and in which circumstances Rust may be good. This chapter will begin with the motivation of our thesis. Following the motivation, the aim will be described in more detail as well as the limitations of the thesis. Then, an outline of the thesis will be presented.

1.1 Motivation

C is a widely used language when it comes to embedded development and real-time systems [3], [4]. But in recent years, a new language called Rust has been emerging. Rust has a similar performance to C [5], whilst offering security in terms of implicit memory management. Thus it is useful to research if Rust is a viable real-time systems language, since if Rust is at the least on par with C in terms of performance it could lead to fewer security problems stemming from manual memory management.

As real-time systems, especially safety-critical real-time systems, grow more complex and more critical infrastructure depend on these systems, the need to make them reliable and safe is becoming more important. Different ways of increasing reliability and safety in safety-critical systems have been proposed. Hilburn and Zalewski [6] analyze safety-critical real-time systems, how they are described in literature, and what the state of the art is in the industry. They show four domains in which the literature and industry applications collide. These are the following four points:

- Specification and design methodologies for real-time safety-critical systems.
- Programming languages with real-time and safety-oriented constructs.
- Operating system kernels with real-time and safety-oriented features.
- Real-time safety-related hardware architectures.

As can be seen from the domains, the second one is directly involved with the programming language used, and the first one is affected by the programming language used. With how connected programming languages used for real-time applications are to the functionality, reliability, and safety of the overall system, the need to use reliable and safe languages, with the right functionality, increases.

Microsoft has stated that 70% of their vulnerabilities are caused by inadvertently creating memory safety issues in C and C++ [7]. To mitigate the vast number of memory vulnerabilities, they propose the use of memory-safe languages such as Rust [7]. With the wide use cases of real-time systems from important civilian applications to military applications, the need to use memory-safe languages grows.

With the importance of programming languages for safety-critical systems in the aspects of reliability, safety, functionality, and security, the introduction of memory-safe languages to the real-time system domain makes sense. Therefore an interesting point to look at would be how a safety-oriented programming language such as Rust compares to C for different metrics when used in real-time systems. With Rust's goal of being type and memory-safe, it could be a good addition in increasing reliability, safety, and security during the development of and use in safety-critical applications.

This thesis will look at these attributes of the Rust language in real-time applications and examine how they compare to similar applications made using C. This could show how a type and memory-safe language works in the real-time system domain and if it can be used efficiently and functionally, both when it comes to resource usage but also how the language supports the needed constructs for real-time systems.

1.2 Aim

This thesis aims to design, implement, and evaluate a real-time system in a Rust framework, called RTIC. This is achieved by doing an analysis and comparison between RTIC and FreeRTOS (which is a real-time framework written in C).

To achieve this aim, multiple applications in RTIC and FreeRTOS are developed to make a comparative analysis between the frameworks, and in doing so also make a comparative analysis between the languages themselves given the setting. The analysis focuses on how the two languages work in a specific system and on a specific set of hardware. In this thesis, there will be different microcontrollers acting as the hardware.

1.3 Limitations

There are some limitations to this project. Firstly, we only investigate the C and Rust languages. The point of the thesis is to investigate if Rust is a viable real-time language, not to figure out which language is the best real-time language. It thus makes sense to only compare Rust to one of the most commonly used languages in real-time systems, which is C according to a survey done amongst embedded developers [8].

Another limitation is that microcontrollers are the only hardware used for the benchmarks. This is because large systems are expensive, and microcontrollers used for real-time systems are typically not advanced. A limitation is that all of the microcontrollers used for evaluation are using the ARM architecture. However, ARM is one of the more well-known architectures in the industry [8].

The final limitation is that the test applications are relatively small. Even though the applications are small, are still big enough to get a satisfying answer since the metrics that are measured are still relevant even for larger applications.

1.4 Outline

This thesis will begin with an introduction, related works, and background chapters. The introduction chapter gives an introduction to the thesis by explaining our motivation and goals. Some limitations in the thesis are also outlined in this chapter. The related works chapter presents some other work that has been conducted in related fields such as real-time systems and security. The background chapter will describe real-time systems and the programming languages used in this thesis. There will also be some background about the real-time operating systems chosen for the thesis in this chapter.

Following the background, the method for the thesis is introduced. The requirements for the hardware used are laid out, as well as the metrics that the applications should test for. Furthermore, there is a description of the applications that are developed.

Then, the implementations are described. In that chapter, a detailed description of how the applications are implemented is given as well as how the measurements of the metrics are done in practice. There is also some discussion about problems that were encountered, and how they were solved.

Afterwards, there are the results and discussion chapters. In these chapters, the results from the tests are presented, and a discussion regarding them. There is also a more general discussion about the state of Rust as an embedded real-time system language as well.

At the end of the thesis, a conclusion is made regarding the viability of the Rust language in a real-time system setting. There is also a comparative analysis between the two languages in a real-time system setting.

2

Related work

In this chapter an overview of related work is presented with earlier work done on Rust as a secure programming language, but also examples of Rust used in embedded programming. There is also some work presented related to real-time systems.

2.1 Memory safety

Oorschot [9] evaluates different memory safety problems, using C as a case study. Oorschot states that 65-70% of all security-related issues are memory errors. Furthermore, Oorschot also states that C/C++ has a majority of these problems. Many of the problems come from the language expectation of the programmer handling issues, such as bounds checking of arrays. Human error is however a big problem.

Another problem described by Oorschot [9] is C's handling of raw pointers. The programmer can create a raw pointer that points to some arbitrary memory location and then cast the value in that memory location as any type. This means that a programmer can cast, for example, a string into an integer without any checks to see if this operation makes sense, which could cause unintentional bugs.

A solution to memory safety is to not use manual memory management and instead use garbage collectors. However, garbage collectors could hamper the performance of the application. Hertz et al. [10] investigate the programming language Java's garbage collector in terms of performance. They find that the garbage collector could decrease the performance by an order of magnitude compared to explicit memory management if physical memory is sparse. Oracle, the developer of Java, has stated that the garbage collector is a source of unpredictability for real-time systems [11]. This means that programming languages that rely on garbage collectors to ensure memory safety might not be suitable for real-time systems.

Another solution to memory safety is to design and implement software for searching through code and fixing memory errors. One such method is the approach developed by Gao et al. [12]. They developed a tool named *LeakFix*, that fixes multiple memory issues when tested on example programs. However, it is hard to create a tool that fixes these problems without having the tool alter the flow of the program. Thus it is difficult to develop a tool that fixes all memory-related issues without changing the flow of the program substantially.

This thesis will explore if Rust could be a solution to these problems. Rust ensures that

some mistakes that can be made in C cannot happen in Rust. This is accomplished using strong restrictions on types and proven memory safety, which are described further in Chapter 3. Rust also does not have a garbage collector, which could add overhead. Since Rust offers these features, the methods presented in this section would become less relevant and Rust might offer a solution at minimal cost and affect the working of the systems.

2.2 Real-time systems

In a case study by Kopec and Tamang [13], it is revealed that there have been numerous critical failures involving different systems, where some stem from application bugs. In multiple cases, failure in computer systems has led to death. Their recommendations to prevent this are more rigorous testing and better company structures, to catch these types of errors earlier.

There are multiple works related to designing formal methods for the verification of timing constraints in real-time systems. Using formal logic called Real-Time logic, Jahanian, and Mok [14] formalise the verification of systems to make them more secure. These formal methods are however complicated, and might not always be applicable to each use case.

Rust is promising as it offers robust error handling if used correctly, and prevents certain memory-related issues from occurring, as is explained in Chapter 3. That is why the usage of Rust in real-time systems is interesting, to see if these features can help in making more secure systems while still having the same capabilities as before. Rust could also aid in not needing as formal methods to make systems more secure.

2.3 Safety of Rust

Evans et al. [15] investigate whether Rust is used safely by developers. Even though Rust offers safety throughout the language, none of that matters if those features are not being utilized by the developers themselves. Evans et al. formulated multiple research questions to aid them in their research. For example, the questions "How much do developers use Unsafe Rust?"[1] and "How much of the Rust code is Safe Rust?" are asked. They can see that most Rust libraries use the *unsafe* keyword very sparsely, but they are still not considered "safe" because dependencies can use *unsafe*. Thus, they conclude that although *unsafe* is sparsely used, it might still be used in most projects due to dependencies of which the developer may be unaware. This might be a pitfall in real-time development since one often needs to use *unsafe* to interact with the hardware.

Pinho et al. [16] research Rust in the context of critical systems. They evaluate certain guidelines for how to write safe C code and explain how some of these guidelines do not apply to Rust. This is because Rust does not allow certain programming

[1] We will explain unsafe Rust in Section 3.3.

techniques that could potentially lead to security problems[2]. They found that Rust has not been used extensively in embedded systems yet, but they conclude by stating that Rust is a language worth keeping an eye on.

Hedgren in his master thesis [17] researches, among other subjects, the use of Rust in microcontrollers. He concludes that languages with automatic memory management using garbage collectors do not live up to certain expectations, but Rust enables the developer to be both productive and secure. The usage of Rust also leads to fewer memory-related security issues, compared to C/C++. Surprisingly, Rust also offered a better performance when generating RSA keys compared to C/C++. This thesis has a similar goal as Hedgren, however, the field of real-time systems is explored instead of cryptography and more focus will be placed on comparing the languages.

Xu et al. [18] also discuss Rust and the security features it has. In this work, they investigate different common vulnerability and exposures (CVEs) in the Rust language. One point of interest is that all memory-related bugs came from code that was in Unsafe Rust. This means that if a large part of the program needs to rely on the use of *unsafe* Rust, Rust has a higher risk of memory safety issues. The difference to C/C++ is that for Rust only the *unsafe* parts of the program need to be checked for errors and not the whole program. This means if a programmer is conservative with the use of the *unsafe* keyword, only small parts of the program need to be checked manually for security errors.

[2]These techniques will be presented in Section 3.3.

3
Background

In this chapter, some needed background is provided. Firstly, some real-time system concepts are explained such as tasks, scheduling algorithms, and priorities. Then, an overview of the C programming language is given followed by an overview of the Rust language and concepts such as ownership and borrowing. Lastly, the real-time operating systems (RTOSs) used in this thesis are presented.[1]

3.1 Real-time system concepts

A real-time system (also called a real-time operating system, or in short RTOS) is a system that consists of tasks that need to be run, and a scheduler that manages the execution of the tasks. In this section, concepts regarding real-time systems are explained.

3.1.1 Tasks

A task is a unit of work that does some sort of computation. It could be anything between reading from an I/O device to handling a software fault in the application. A real-time system is defined by its tasks, and the scheduler of a real-time system tries to ensure that tasks are being executed according to certain timing constraints. Tasks can either run periodically or be dependent on an event, such as a hardware interrupt. Some tasks may also have deadlines, which is a point in time when the task should have finished its execution. Another concept is the worst-case execution time (WCET), which is the time it takes to execute a task in the worst case. Tasks can also have a priority assigned to them, which symbolizes the urgency of a task. The higher priority a task has, the more urgent it is for that task to be executed.

Typically, there are three states a task can be in: *running*, *waiting*, or *blocked* [2]. Running tasks are tasks that are being executed at the current time instance. Waiting tasks are tasks that are ready to run and are just waiting to be dispatched by the scheduler. Blocked tasks are tasks that are unable to be dispatched for some reason, for example, because they are waiting for an I/O operation to conclude. The different states are visualized as a state diagram in Figure 3.1. Tasks may also have a priority

[1]Parts of this chapter are based on earlier work done by the authors of this thesis in a project for the course Language-based Security at Chalmers Technical University [19].

assigned to them, which represents the urgency of a task. This concept will be explored further in the upcoming sections.

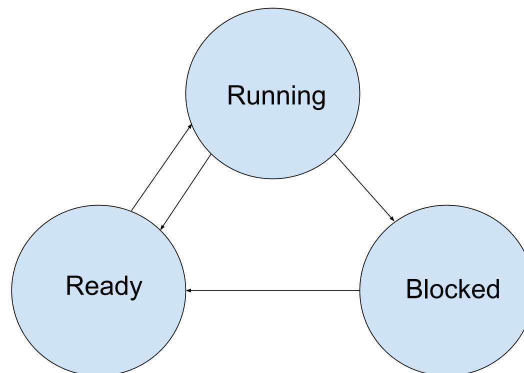


Figure 3.1: A state diagram of the different task states. A task in the *running* state can either become *ready* or *blocked*. A *blocked* task can only become *ready*, and a task in the *ready* state can only become *running*.

3.1.2 Scheduling algorithms

Scheduling algorithms are used to decide the order of task execution. A scheduling algorithm looks at tasks in the ready state and then decides which task should run. Scheduling algorithms are also a way to set priorities for tasks. An example of a scheduling algorithm is the *deadline monotonic algorithm* [20], which schedules tasks based on their *absolute* deadline. To clarify what this means, imagine two tasks: A and B. Task A has a deadline three seconds after task A has become ready, while task B has a deadline five seconds after task B becomes ready. In this scenario, task A will always have a higher priority than task B. This is also true even if task B may be closer in time to its deadline than task A when task A becomes ready. This is what is meant by an absolute deadline. So in short, tasks that have a shorter absolute deadline are assigned a higher priority and executed before tasks with lower absolute deadlines.

Scheduling algorithms can also either be preemptive or not. If a scheduler is preemptive, it can *preempt*, i.e., move a task in the running state to the waiting state if a task with higher priority becomes ready. The logical state (i.e., stack and heap allocations) of that task is then saved, and then loaded again once the scheduler dispatches that same task. Moreover, scheduling algorithms can also be either local or global. Local schedulers assign a task to a specific processor, and then that task always runs on that particular processor if it is dispatched again. With global schedulers, any task may run on any processor at any given time.

3.1.3 Priorities

As previously mentioned in Sections 3.1.1 and 3.1.2, each task may have a priority assigned to it, which symbolizes the task's urgency. The higher priority a task has, the more important that task is compared to other tasks with lower priorities. This

means that it should execute before the ones with lower priorities. Priority can be assigned either statically or dynamically to a task. Static priority is usually assigned to a task before the program begins its execution, and the priority is persistent throughout the runtime. Dynamical priorities for a task can change during the runtime, depending on different factors relevant to the execution.

There is also the concept of priority inheritance [2]. Priority inheritance is a mechanism to eliminate so-called *priority inversion*, which is a state where a task with low priority gets precedence over a task with high priority, effectively inverting the priorities. This can occur if a low-priority task, let us say `L`, is holding a resource `R` that is requested by a high-priority task, `H`. In this case, `H` has to wait for `L` to stop holding that resource. But then imagine a third task with medium priority, M_1 becomes ready and this task does not rely on `R`. Then `L` might be preempted in favour of M_1 (if the scheduler allows preemption) which effectively grants M_1 a higher priority than `H`. Then another task with medium priority, M_2, becomes ready and blocks `L` as well and in turn also blocks `H`. This case is called an unbounded priority inversion since multiple tasks with medium priority can become available and block `H`, potentially forever. In Figure 3.2, the example is visualized.

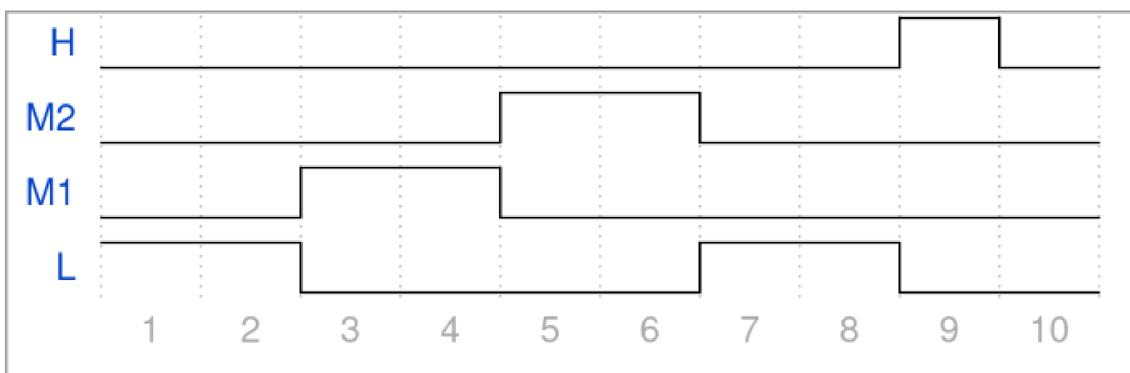


Figure 3.2: The priority inversion example. The y-axis represents the different tasks. If a task is running, then the line connected to each task is raised. The x-axis represents different time instances. The `L` task starts executing, and then gets preempted by M_1 and M_2. Then, when `L` has finished its execution, `H` can execute. As can be observed, the `H` task never runs until the end even though it has the highest priority.

Priority inheritance addresses this problem by temporarily assigning a task that holds a resource the highest priority among the other tasks that might at some point want that same resource. Let us imagine the previous example again, as given in Figure 3.2. If priority inheritance was implemented, then task `L` would temporarily get the same priority as task `H`. Thus, task M_1 would not be allowed to preempt task `L` since `L` has higher priority (the same priority as `H`). Then after `L` has finished its execution `H` would run and then lastly M_1 and M_2, avoiding priority inversions. This is visualized in Figure 3.3.

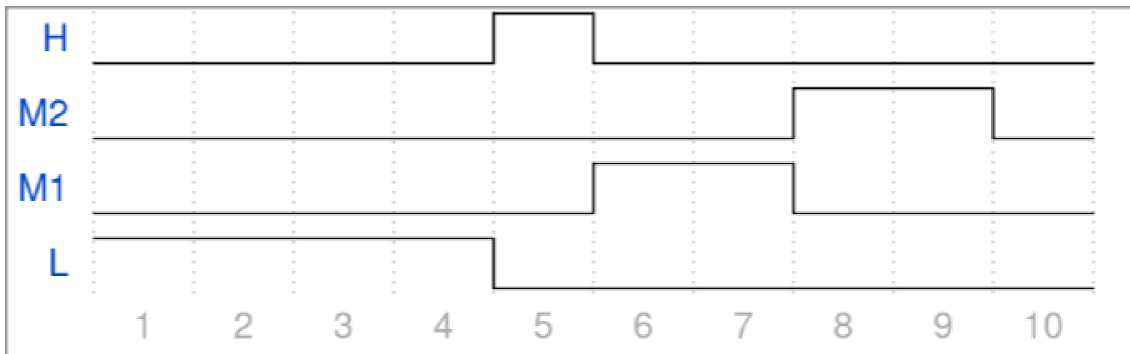


Figure 3.3: The flow of execution with priority inheritance. Neither M_1 nor M_2 are allowed to preempt L, since L has the priority of H. When L's execution has finished, H will execute fully. Then, M_1 and M_2 are allowed to execute.

3.2 The C programming language

In this section, some aspects of the C programming language are explained. In particular, the memory management and security aspects of the language will be explained. Before that, however, a general introduction to the language is given.

C is a high-performing low-level language and is one of the most influential languages of all time [21]. It is used in various environments, one of them being real-time systems. C has a high performance [22] since the language adds less overhead to its functions unlike other languages, such as bounds checking. These checks are instead left to the programmer to handle. There is also no built-in garbage collector in C which could add unnecessary and unpredictable overhead. Instead, memory management is mostly manually managed and the compiler places few restrictions on the programmer to ensure that the memory is handled safely. This means that if safe memory accesses are to be done in C, the abstractions need to be implemented by the programmer.

Because C has existed since 1972, the ecosystem is rich, and combined with the fact that C can be compiled into most architectures there are very few things C is not capable of doing. This has made C a good candidate for real-time development.

3.2.1 Memory in the C language

The memory layout in C programs uses the same typical layout as most other programming languages, as is visualized in Figure 3.4. The memory consists of a text segment, an initialized data segment, an uninitialized data segment, a stack, and a heap. The text segment contains the program instructions to be executed and is usually placed below the heap. The initialized data segment consists of global variables, static variables, and constant variables, initialized by the programmer. The initialized data can be changed at run time and the segment can be split into a read-only and a read-write area. The uninitialized data segment instead contains the global and static variables that have been declared without a specified value. Local uninitialized data live on the stack.

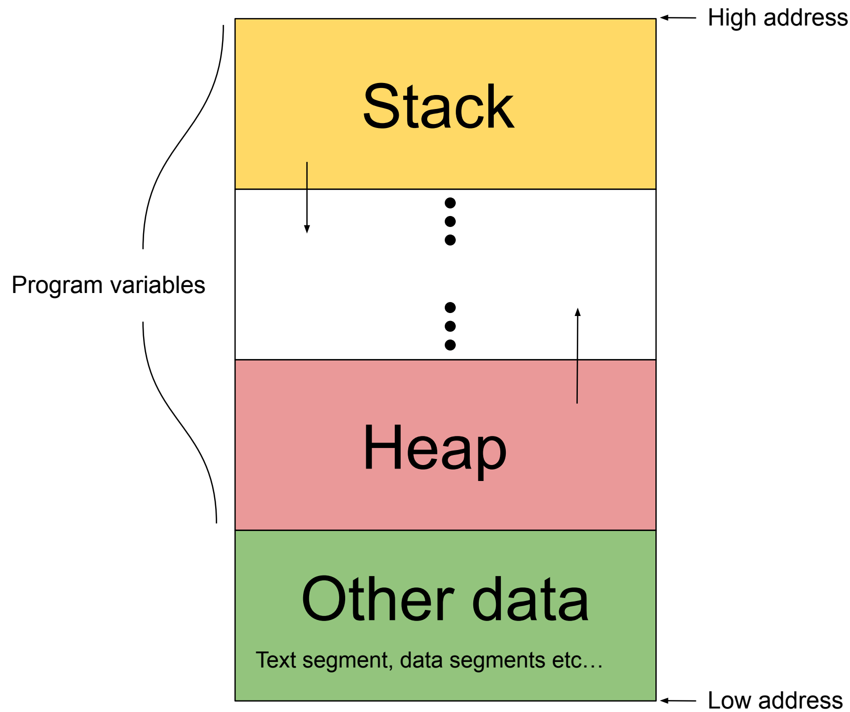


Figure 3.4: A typical memory layout of a process. The stack and heap use the same memory area. All other data, for example, the text segment, lives after the heap.

The stack and heap are parts of the memory layout which can grow and change in size. The stack of the memory, as discussed by Aleph One [23], comes from the abstract data type with the same name and works like a last in, first out (LIFO) queue. So-called stack frames are pushed and stored in this queue, whenever a function is called. The stack frame contains the parameters of the function, the local variables, and the data needed to recover the previous stack frame (like the instruction address). To keep track of the stack as it changes, a pointer called the *stack pointer* is used. The *stack pointer* points to the end of the stack, while the beginning of the stack is a fixed point in memory. The stack usually grows downwards (from a high address to a lower one).

Since the size of the data to be stored is not always known, there needs to be a way to allocate memory at run time. And this is where the heap comes in. How to dynamically allocate space in memory on the heap is described further in the next section.

3.2.2 Dynamic allocation of memory

Memory allocations in C are mostly handled manually by the programmer. In other programming languages, this might instead be handled automatically by the compiler. Examples of such languages are Java, Golang, and Python. The approach that C uses gives a lot of freedom when writing programs in terms of memory management. But it also means that it is up to the programmer to validate that all allocated memory is freed after it has been used, as otherwise, the machine will run out of memory and most likely crash.

In C, the dynamic allocation of memory in the heap is most commonly handled with the methods from the standard library such as `malloc`, `calloc`, and `realloc`. The function `free` frees up the allocated memory, as the name suggests. The implementation of the allocation methods may vary and there may exist multiple different implementations, depending on which library is used. Programmers need to take extra care to validate that situations like double-free [24] do not occur. Double-free is a situation in which the same memory location is freed twice, which might lead to undefined behavior.

3.2.3 Security aspects

As mentioned before C is a language that emphasizes freedom and has few restrictions on what is allowed to do in the language. This freedom makes C exceptionally performant, as it allows the programmer to find the best possible solution. The price for this is however in terms of security. For example, the compiler makes no bounds checking by default. This makes the language susceptible to security issues such as buffer overflow attacks. However, most C compilers warn the programmer if the programmer tries to use a known dangerous function. For example, if one tries to use the dangerous function `gets`, which may be vulnerable to a buffer overflow attack [25], the compiler will throw a warning saying that the function is dangerous and should not be used.

There may also be problems regarding ambiguity. Consider the code snippet given in Figure 3.5. In C, the expression in the `if` statement would be evaluated as `true` since the `++` operator would be calculated first. However, this might not be obvious for someone who might be less familiar with the language. It is ambiguous from just looking at the code, and it is also considered a bad practice to do this in C for that reason.

```
void main() {
    unsigned int number = 4;
    if (number++ == 5) {
        //do something
    }
}
```

Figure 3.5: Ambiguous execution. It is not apparent whether the `++` operation is executed before or after the evaluation of the boolean expression in the `if` statement.

3.3 The Rust programming language

Rust is a new high-performing multi-purpose language that has gotten some traction recently [26]. In this section, an overview detailing some aspects of the language such as its memory management will be given. There will also be some explanations of how Rust tries to achieve memory security.

Rust, like C, can be used in different environments [27], [28], and with the emergence of real-time operating systems written in Rust, it is now possible to use Rust in real-time settings without having to create a real-time operating system from scratch. Rust also has similar performance to C [5], but unlike C the programmer is normally not allowed to manually manage the memory. Memory management is instead done at compile time, and the programmer is also forced to follow strict rules to ensure that the memory is managed in a safe manner using concepts such as *ownership* and *borrowing*. Thus many security vulnerabilities that would be easy to make in C are substantially harder in Rust [29].

The flip side of this is, however, that many restrictions are placed on the programmer. The programmer is for example not allowed to interact with memory pointers directly, which is crucial for embedded development. There may also be costs in execution performance when developing in Rust compared to C, as more checks are being done natively at runtime to ensure that the application is executing correctly.

3.3.1 Mutability

In Rust, variables are not mutable by default. This means that once a value is instantiated, it is not possible to modify it. To make a variable mutable, one needs to add the `mut` keyword before the variable name, as seen in Figure 3.6

```
fn main() {
    let mut number = 3;
    number += 4;
}
```

Figure 3.6: A mutable variable in Rust.

The reason for this is to minimize the potential for bugs. If one part of the code expects a variable to stay constant during runtime, and then another part of the same code modifies the variable, undefined behavior might be introduced. This also shows the mindset of Rust, that you need to be very explicit in what you as a programmer want to do.

3.3.2 Ownership

One of the core fundamental rules of the Rust language is ownership. According to the Rust book [30], there are three rules to ownership:

- Each value must have an owner.

- There can only be one owner at a time.
- The value is dropped once the owner goes out of scope.

To explain this in simpler terms, each value is associated with an owner, where an owner is typically a variable or a function. Once that owner goes out of scope, the value will be dropped and automatically freed from memory. These rules are only applied to dynamic data types that live on the heap. Static data types, like integers, do not follow these rules as they instead live on the stack. Let's consider the code example seen in Figure 3.7 that uses both a static data type and a dynamic data type.

```
1  fn main() {
2      let s1 = String::new("hello");
3      let s2 = s1;
4      println!(s1);
5
6      let i1 = 3;
7      let i2 = i1;
8      println!("{}", i1);
9  }
```

Figure 3.7: An example of code that will panic due to the ownership rules of Rust.

The code in Figure 3.7 will panic (crash) at line 4, but not at line 8. The reason for this behavior is that when the string is created and assigned to `s1` on line 2, the variable `s1` becomes the owner of the value "hello". Once `s1` is assigned to `s2`, the ownership of the string is transferred from `s1` to `s2`, freeing `s1` in the process. Thus it is not allowed to use `s1` after line 3, which we try to do in line 4. For data types with static size, like the integer at line 6, this is not a problem. The number 3 will simply just be cloned to the variable `i2` and both variables will be valid.

The reason for the behavior in Figure 3.7 can be explained using pictures. In Figure 3.8, one can observe how `s1` is represented in memory.

When line `3` is executed, there are three options:

a) Create the variable `s2` and have it point to the same data, as seen in Figure 3.9a.

b) Create a clone of the value and let `s2` point to that clone, as seen in Figure 3.9b.

c) Create the variable `s2` and have it point to the data and dropping `s1` in the process, as seen in Figure 3.9c (current Rust implementation).

Option *a* is problematic because of the classic memory problem double-free [24]. If Rust attempts to free both variables at two different times, it could lead to memory leaks as one instance might already have freed the memory from the heap. It creates inconsistency, so this option is not viable. Option *b* is doable but this cloning operation could be expensive if the data type is big enough. When it comes to values with static sizes, such as integers, this is doable as they are often small.

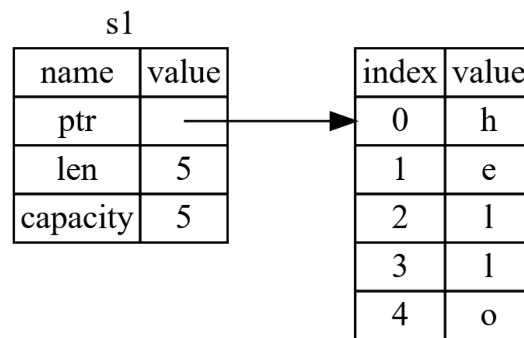


Figure 3.8: Representation of a variable `s1` in memory. It has a `ptr` field that points to the data of the string, the characters themselves. The `len` field holds the actual length of the string, while the `capacity` field holds the capacity of the data type, i.e. the maximum number of characters allowed in the string. This picture is copied from the Rust book [31].

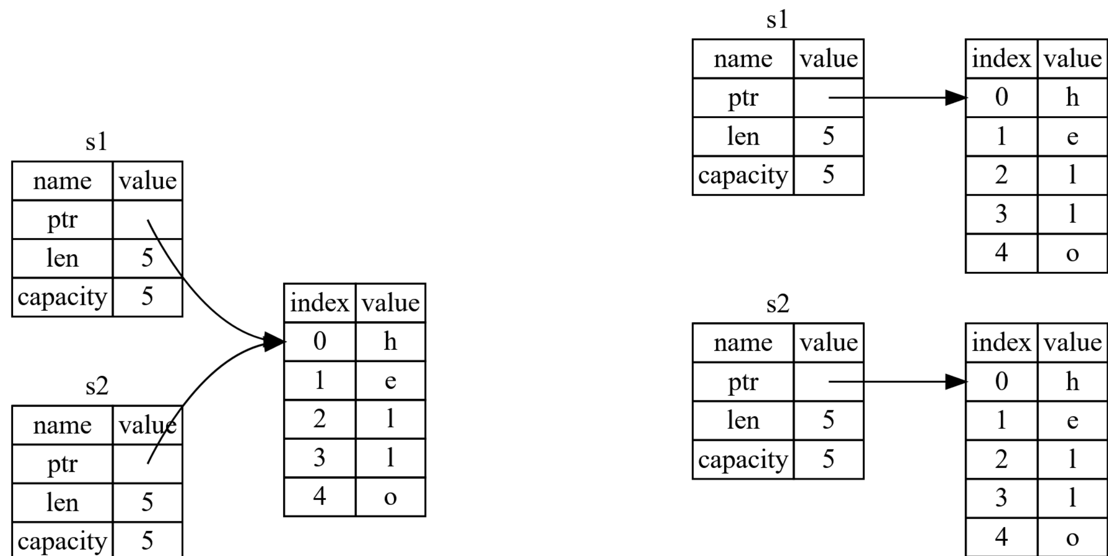
This leaves option *c*, which is also the way Rust implements memory management. `s1` is dropped and the value is *moved* to `s2`, which becomes its new owner. This is the most memory-secure way and leaves no inconsistency. The compiler will ensure that the value `s1` can not be used again, and there will be a compile error if the programmer tries to use `s1` again.

The second option could also be done manually if one wishes, utilizing the `clone` function for Strings as seen in the code snippet in Figure 3.10. Cloning data types thus becomes a conscious action, where one should take into consideration the possible performance implications.

Ownership can also be transferred to other functions. Consider the code snippet in Figure 3.11. This code will panic at line 8 since it is disallowed to use the variable `transfer` after we have transferred the ownership to the function `another_function`. Once the function `another_function` goes out of scope, the value of *transfer* will be dropped and no longer exist in memory. Although if `transfer` was a data type with a known size at compile time, this code would not panic as the variable `transfer` would just be cloned instead. If the ownership was not transferred, it would create ambiguity about when the value should be dropped. It could either be at the end of the current function's scope (`main`'s scope) or at the end of `another_function`'s scope. By transferring the ownership the ambiguity is removed and the value is dropped at the end of `another_function`'s scope.

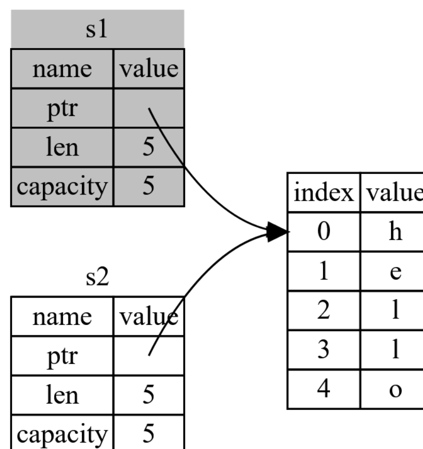
3.3.3 Borrowing

There exist situations where one might want to use an object in another function without passing the ownership of the object to the other function. This can be accomplished with borrowing, which essentially creates a reference to an object. Variables can also be *mutually borrowed*, where one is allowed to change the value of the borrowed variable. The concept of borrowing is similar to pointers in other languages, but two rules need to be followed:



(a) `s2` simply creates a new pointer to the same data, while `s1` keeps its pointer. This picture is copied from the Rust book [31].

(b) The value is cloned, and `s2` starts pointing towards that new data. This picture is copied from the Rust book [31].



(c) The variable `s2` starts pointing towards the data and `s1` is dropped. This is the current Rust implementation. This picture is copied from the Rust book [31].

Figure 3.9: Different methods of handling the situation in which a variable `s1` is assigned another variable `s2`.

```
1  fn main() {
2      let s1 = String::new("hello");
3      let s2 = s1.clone();
4      println!(s1);
5  }
```

Figure 3.10: This code will not panic, because of the cloning of `s1`. This means that it is allowed to use `s1` after it has been cloned to `s2`.

```
1  fn another_function(transfer: String) {
2      println!("Received transfer {}", transfer);
3  }
4
5  fn main() {
6      let transfer = String::new("I will be transferred!");
7      another_function(transfer);
8      println!("{}", transfer);
9  }
```

Figure 3.11: This code is an example of transferring ownership. The ownership of `transfer` is transferred from `main` to `another_function`, causing the function to panic.

- At any given time, you may only have one mutable reference or multiple immutable references to the same value.
- References must always be valid.

A valid reference is a reference that points to an object in memory. An invalid reference points to either random data or nothing (a null pointer). Both rules are enforced by the compiler, which makes it impossible to break them.

Using the same example as earlier in Figure 3.11, we can rewrite it with borrowing to make it work. A code snippet of this can be found in Figure 3.12.

```
1  fn another_function(transfer: &String) {
2      println!("Received transfer {}", transfer);
3  }
4
5  fn main() {
6      let transfer = String::new("I will be transferred!");
7      another_function(&transfer);
8      println!("{}", transfer);
9  }
```

Figure 3.12: This code is an example of borrowing. The function `another_function` borrows the variable `transfer` from `main`.

The `&` symbol creates a reference to the object after the `&` symbol, in this case `transfer`. Thus a reference to `transfer` is sent into the function instead of transfer-

ring the ownership of `transfer` to the function. To make this work, the argument for `another_function` needs to be changed to allow references.

3.3.4 Other security aspects

The Rust language also has other features to make the language more secure. For example, Rust offers runtime checks against buffer and integer overflows. If a buffer or integer is overflowed the application will simply crash, preventing buffer overflow attacks [25]. However, when compiling Rust for production purposes integer overflow detection is not enabled by default since it increases the execution time and binary sizes. Rust also offers integer overflow checks at compile time, but these can only recognize obvious buffer overflows such as trying to assign the value 256 to an 8-bit integer.

Another feature is that it is not possible to increment a variable using the `++` operator, as seen earlier in C in Chapter 3.2.3. This is to prevent ambiguous execution, as also illustrated in Chapter 3.2.3. For the same reason, it is not allowed to assign to a variable in an `if` statement.

3.3.5 Unsafe Rust

For Rust to qualify as a systems programming language, it is necessary to be able to interact with the memory directly sometimes. Unsafe Rust enables the programmer to do this [32]. Unsafe Rust enables some extra features, such as being able to dereference a raw pointer. It is however important to note that unsafe Rust does not turn off any of the other security features. It merely enables more features for the programmer, but the rules of ownership and borrowing must still be followed. The implications of Unsafe Rust for the language itself will be further discussed in Chapter 7.

3.4 The real-time systems

There are two real-time systems written in respective languages that are popular: FreeRTOS for C and RTIC for Rust. They are very similar in most aspects, but there are also some dissimilarities between them that will be explored in the following sections.

3.4.1 FreeRTOS

FreeRTOS is a real-time operating system written in C [33]. It is an open-source project, as anyone is free to contribute to the project or review the code. Anyone is free to create their own sub-licenses and own versions of FreeRTOS to distribute. The project license (MIT or Apache) has the benefit that the framework can be used commercially in the industry. The operating system also supports many different devices [34].

3.4.1.1 Scheduling tasks and timing

To schedule its tasks, FreeRTOS uses a fixed-priority preemptive scheduling policy [35]. This means that each task is assigned a priority at compilation time. Tasks may dynamically get higher priorities due to priority inheritance but otherwise remain the same throughout the entire runtime. Being preemptive the operating system is capable of stopping already running tasks in case a task with a higher priority becomes available. The scheduler also uses round-robin as a time-slice, which means that tasks that share the same priority take turns being executed. It also means that tasks are executed until they are finished unless they are preempted by some other event. The scheduler in FreeRTOS is a local scheduler, but there exist versions of FreeRTOS where the scheduler is global.

There exist two types of tasks in FreeRTOS: a normal task and a co-routine [35]. A task supports priority and preemption, but each task manages its own data stack which leads to higher memory usage. A co-routine on the other hand uses a shared stack with all other co-routines, which results in lower memory usage. However, tasks always have a higher priority than co-routines and co-routines are more complex to handle because of the shared stack. Co-routines are more suitable to be used in microcontrollers where memory is limited; otherwise, tasks are preferred.

The tasks themselves have no way of scheduling themselves after they are completed. If one wants a task to execute after a certain period, one could either use an infinite for loop that sleeps for a certain period or utilize so-called ”*software timers*” [36]. *Software timers* is a way to schedule the execution of a function in the future. They can either be assigned a period to execute on or just be executed once. The *software timers* are not tasks per se, and the functions are executed inside the *timer service task*.

3.4.1.2 Resources and synchronisation

FreeRTOS has no built-in functions for resource handling. Resource handling is done in the classic C way, with either global variables for sharing between multiple threads or resources defined locally in each task.

For synchronization between tasks, FreeRTOS supports the classic synchronization methods such as *mutexes*, *message queues* and *semaphores* [35]. The synchronization methods can be both blocking and non-blocking, depending on the use case.

3.4.1.3 Interrupts

To run FreeRTOS on a Cortex core, three interrupt lines need to be connected to the kernel. These are the SysTick, PendSV, and SVCCall interrupts. The SysTick interrupt is used for controlling timings, such as dispatching a new task at a specific interval. The PendSV interrupt is used for context switching, and the SVCCall interrupt is used to access certain device drivers.

FreeRTOS does not handle hardware interrupts directly. However, it is possible to set up a task that is triggered upon an interrupt occurring. This is called a ”deferred

interrupt", and is useful if one for some reason needs to be able to use the full FreeRTOS kernel during an interrupt.

3.4.2 RTIC

RTIC is a real-time operating system written in Rust [37]. It is similar to FreeRTOS in the sense that it is also an open-source project with the same license as FreeRTOS (MIT or Apache). This makes RTIC suitable for industry usage as well as for smaller hobby projects. RTIC has a long history and is a continuation of the real-time for the masses library released in 2013 [38], but the current framework originated as a master thesis at Luleå University by Tjäder [1] which is built on the work by Rivera [39]. RTIC has grown substantially throughout the years, with half a million total downloads from the Rust package repository [40] in 2023. The aim of the original master thesis by Tjäder was to create a framework that has similar performance as a C framework but also gives the security that comes with Rust.

3.4.2.1 Scheduling tasks and timing

RTIC uses a fixed-priority preemptive cooperative scheduler to schedule its tasks. In addition to this, RTIC also uses the Stack Resource Policy (SRP) that is described by Baker [41]. The goal of this policy is to improve schedulability under fixed-priority preemptive scheduling, by using another protocol called the Priority Ceiling Protocol (PCP) and extending it with more features. SRP guarantees, among other things, race-free scheduling and improves resource efficiency.

RTIC has two types of tasks: software tasks and hardware tasks. Software tasks are triggered by the operating system, while hardware tasks are triggered by hardware interrupts. These tasks however use a shared stack, compared to the FreeRTOS tasks that use separate stacks. Due to a restriction of SRP, the tasks are restricted to running on only a single core (local scheduling). Another reason for the lack of multiple core support in RTIC as mentioned by Tjäder [1] is the complexity that stems from handling tasks running on different cores, and the edge cases that come from it.

Due to RTIC using a cooperative scheduler, it is not possible to create multiple instances of the same task. In RTIC v1 however, there exists a "capacity" attribute one can assign a task. This makes it so that it is possible to schedule multiple tasks up to the capacity limit, but due to the cooperative scheduler, they will never interleave. This means that in practice, there only exists one instance of any task at any time. In RTIC v2, the capacity attribute does not exist. It is however possible to simulate the capacity attribute using message queues.

3.4.2.2 Resources and synchronisation

RTIC has a special way of allocating resources. Both resources local to a task and resources shared between tasks must be declared in two objects, and then those two objects need to be passed to the RTIC initialization function. The tasks can then access resources provided by a function decorator. Local resources can be accessed

directly without any means of synchronization, but a unique local resource can only be assigned to one task. This means that if multiple tasks try to use the same local resource, the program will not compile. Shared resources can be used for multiple tasks, but they must be locked to be accessed. It is possible to declare a shared variable ”lock-free”, but for this to be valid the tasks that access the resource must have the same priority. This is fine, as RTIC will never preempt a task in favor of running another task with the same priority.

Another way to synchronize is to use *message passing* over channels. Channels in RTIC are “multiple producer single consumer” channels, meaning that multiple tasks can write to the same channel but only one task is allowed to listen.

3.4.2.3 Interrupts

RTIC is built around hardware interrupts. Tasks are dispatched by hardware interrupts, which is accomplished by having each priority level assigned a unique interrupt line. When the scheduler wants to dispatch a task of a certain priority level, the relevant interrupt line is triggered. This also means that the number of priority levels possible is bound by the number of hardware interrupt lines available. It is also possible to have tasks directly connected to a hardware interrupt. This means that whenever a specific interrupt is triggered, the associated task is dispatched as well.

It is also possible to wrap a code section in a critical section. This is provided by an external library called “critical_section”. No interrupts, either external or internal, can be executed while the program is inside a critical section.

4 Method

This chapter presents the methodology by which the languages C and Rust and the RTOSs will be tested. A motivation for why C was chosen to be used as a comparison to Rust is given as well as a motivation for the choices of RTIC and FreeRTOS. After that the hardware that will be used is described and a justification for that hardware is provided. The applications used to evaluate and examine the languages and RTOSs are presented along with the metrics they will be tested for.

4.1 Methodology

The aim of this thesis, as explained in Section 1.2, is to evaluate real-time systems written in Rust using RTIC and compare them to systems written in C using FreeRTOS. This will be done by creating multiple real-time applications for different hardware and analyzing them based on different metrics. The applications are created to test specific aspects that are important to real-time systems. The different versions of Rust and RTIC as well as C and FreeRTOS will be made as similar as possible to give a representative view of Rust and RTICs performance and function compared to C and FreeRTOS.

4.2 Motivation of chosen programming languages

When choosing the languages to compare with Rust in real-time systems there are some different choices. Among these are C, real-time Java, and Ada. In this thesis, C was chosen as the language to compare Rust to as described in Chapter 1.

Our choice of comparing Rust with C is partially based on the wide use of C as a real-time systems programming language [3]. Since the aim of this thesis, as described in Section 1.2, is to evaluate and compare the language of Rust and a framework written in Rust there also needs to exist a similarly featured framework in C. The C framework chosen for this thesis was FreeRTOS, given in Section 1.2, and a motivation for this choice is provided in Section 4.3.

Ada could be a good choice to compare with Rust. Ada has strong typing and support for tasks. It was created for embedded programming and there are RTOSs written in Ada [42]. With these similarities and that Ada uses a similar memory management system as C, adding Ada to the analysis can be interesting. Unfortunately, the

RTOSs that are written in Ada do not have the needed support for different hardware that is required, as will be further discussed in Section 4.3, which makes it difficult to include Ada in the analysis.

Real-time Java is also an option, however, few frameworks exist written in Java which makes it difficult to use in this thesis. There exist frameworks written in Java such as JARTOS or javolution. JARTOS was developed by Lu [43] in 2007 and two more works by McKerrow [44] [45] were also done in 2007. Beyond this the use of JARTOS is limited and there is not enough documentation and hardware support to use in this thesis. For javolution, there exists documentation on how to use the library, but the project seems abandoned which goes against the aim of investigating if Rust is a viable real-time language today and in the future. Since to see if Rust is viable, it should be compared to a language and framework that is used in the industry at the moment.

C makes a good choice to compare Rust with since it has similarities but also differences with Rust. Both languages are imperative but have different typing disciplines. This allows us to focus on the features of Rust that are described in Section 3.3. With the aim being to evaluate Rust, an important aspect is how these features can affect a real-time system and C gives a more similar comparison than what the other languages would. There also exists more documentation for C both in terms of the language itself but also for hardware. With the advantages mentioned above and the addition that the authors have the most personal experience with C, it was the language chosen for this thesis.

4.3 Motivation of chosen real-time operating systems

In this section, a motivation for the chosen RTOS is given. The requirements that the real-time operating systems need to adhere to are presented along with the chosen RTOSs.

4.3.1 Requirements

The requirements that the RTOSs need to adhere to are given in the list below. A description of why the requirements are made can also be found below.

1. Support for a variety of hardware.
2. Similar feature set between the RTOS for Rust and C.
3. Used in industry. (Only applicable to C version)

Requirement 1 is to ensure that the requirements for the hardware, given in Section 4.4.1, can be satisfied. The need for the RTOSs to be able to support a variety of hardware is important to allow for flexibility in what hardware is available. As stated in Section 1.2, different microcontrollers will be used for the thesis. Requirement 2 is that both RTOSs need to have similar features. This is important as the applications

that are going to be built need to be as similar as possible. With a similar feature set, the applications become easier to implement and there will not be a need to change the workings of the operating systems to make the applications work similarly. It is important to emphasize that this does not mean that the RTOSs need to work in the same way but instead, they need to be able to do the same things. As the analysis is supposed to explore the viability of Rust as a real-time system language to be used in actual applications, the RTOS chosen for the Rust applications need to be built with actual use in mind. This need for a RTOS made with actual use in mind is why the third requirement is added. If we want to test Rust for use in real applications it should be compared against an industry-proven RTOS, this means that the RTOS chosen for the C versions need to be an established RTOS used in industry.

4.3.2 C RTOS: FreeRTOS

To give a good view of the viability of Rust as a real-time systems language the RTOS used in the C applications needed to be robust and already used for industry applications. To satisfy this and the other requirements given in Section 4.3.1 FreeRTOS was chosen. A description of FreeRTOS and how it works is given in Section 3.4.1. As described there the open source nature of FreeRTOS as well as the emphasis on reliability and ease of use makes it a good choice for industry use. This can be seen from the partners that support it and thus Freertos satisfies the third requirement.

4.3.3 Rust RTOS: RTIC

For the RTOS used for the Rust applications, there were some other potential choices aside from RTIC. Among these are Embassy and Drone. Both make use of Rust features to optimize the RTOS. RTIC was chosen because it is the most downloaded of the three. RTIC has over half a million downloads [40] while Embassy has a little over 88 thousand downloads [46] and Drone has just over 42 thousand downloads [47].

RTIC also exists in two major versions: version 1 and version 2. Version 1 is the most commonly used version [40], and during some smaller initial tests, the conclusion that version 1 was faster than version 2 could be drawn. For this reason, it was decided to develop the applications in version 1. This is because we want to test a RTOS as close to the state-of-the-art in the Rust world as possible, and since version 2 is relatively young (released in 2023) it felt more appropriate to make the applications in version 1. However, all applications will also be developed in version 2 for one of our hardware to make a more thorough comparison between version 1 and version 2, as our initial tests could be wrong. The results for version 2 will be presented along with the results for version 1.

4.4 Choice of hardware

In this section, an explanation of and justification for the hardware choices is given. The hardware choices are based on the requirements that are given in Section 4.4.1

as well as other criteria needed to evaluate different aspects of the languages and RTOSs. Due to the wide use of real-time systems, they can be used in powerful systems with an ample amount of resources to low-powered or resource-constrained systems. To better evaluate the languages and RTOS chosen in Section 4.2 and 4.3 respectively, the hardware choices should reflect this range of hardware that real-time systems can be used on.

4.4.1 Requirements

The requirements given below are to ensure that the planned applications given in Section 4.5 can be developed and implemented.

1. The hardware architecture needs to be supported by C and Rust.
2. The hardware needs to be supported by the two RTOSs.
3. The hardware need to have light-emitting diode (LED) capabilities.
4. One hardware needs to support a Digital to Analog Converter (DAC).

The first two requirements state that it is possible to develop our applications on the hardware. Both C and Rust have compilation targets for most existing architectures (except for certain niche ones), so the first requirement is easy to fulfill. The second one states that the RTOSs must be able to interact with the hardware. This is usually done using something called a Hardware Abstraction Layer (HAL). A HAL is similar to a library, where functions are provided that make it possible to interact with the hardware. For FreeRTOS, there also needs to exist port files for the architecture. These port files allow FreeRTOS to install its interrupt handlers in the correct places and are provided by the FreeRTOS library. For RTIC, only the HAL is needed.

The third requirement is important to ease the development process. With an LED it is possible to quickly develop a simple application that blinks the LED. This gives a good starting ground for the rest of the applications, and it is also easier to develop these LED applications since they give immediate visual feedback if they behave correctly or not. For example, the timings of the RTOSs can be tested to see if they behave correctly and accurately. The last requirement is the support for a DAC. This is needed for the *Tone generator* application which is explained in Section 4.5.4.

Apart from these strict requirements, it is also preferred to have different levels of resources for each microcontroller. Since real-time systems are used in a wide range of different hardware, it would be desirable to have different levels of resources to capture this aspect.

4.4.2 Hardware

In Table 4.1, one can observe the hardware chosen for this thesis. Each microcontroller will be explained more thoroughly in the following sections.

Hardware	Processor type	Clock Frequency	RAM	Flash
Discovery	Cortex-M0	48Mhz	8KB	64KB
MD407	Cortex-M4	168MHz	192KB	1MB
PICO	Cortex-M0+	133MHz	256KB	2MB

Table 4.1: The hardware chosen for the thesis.

4.4.2.1 STM32F0Discovery

The STM32F0Discovery (hereby referred to as ”Discovery”) is an STM32F051R8 microcontroller that uses an ARM Cortex-M0 CPU with a clock frequency of up to 48MHz with 64KB Flash and 8KB SRAM [48]. With the low amount of resources, both in terms of memory size and clock frequency, the Discovery satisfies the need for a microcontroller with low resources. For the other requirements, as stated in Section 4.4.1, the board has LED capabilities with the two built-in LEDs and there exist compilers for both C and Rust. Support from the RTOSs is made with finished portables in FreeRTOS and RTIC support is handled by supplying an appropriate HAL. This is also how it should be done for all other hardware.

4.4.2.2 MD407

The MD407 is based on the specification for the STM32F407 microcontroller and uses the STM32F407VG [49] specification. This means it has 192KB of SRAM and 1MB of Flash memory and makes use of the Cortex-M4 running at 168 MHz [50]. The MD407 satisfies the requirements stated in Section 4.4.1. It has two built-in LEDs and there exist compilers for both languages. When it comes to support from the RTOSs the finished portables in FreeRTOS can be modified to work with the MD407, this is described further in Section 5.2.3. With the STM32F407 specification of the MD407, it is a good candidate as a next step in resources compared to the Discovery. With the included DAC the fourth requirement is also satisfied.

4.4.2.3 Raspberry Pi Pico

The Raspberry Pi Pico (hereby referred to as ”Pico”) consists of an RP2040 microcontroller, using dual ARM Cortex-M0+ with a flexible frequency of up to 133MHz. For memory, it has 256KB of SRAM and 2MB of flash memory [51]. With the dual-core layout and the increase in memory, the Pico is used to show a system with ample resources. As with the other hardware there exist compilers for both languages and the Pico has a built-in LED. The Pico is supported in FreeRTOS with a third-party portable which allows for the use of both cores (not available in the portable from FreeRTOS) and can be used with the Pico SDK [52].

4.5 Applications

In this section, the applications that will be designed and implemented are explained. The applications are listed below and further descriptions of the applications are

given in Section 4.5.1 to 4.5.5 respectively. The applications are created to test one or more metrics. Which metric or metrics each application is used for is described in Section 4.6.4.

- Minimal
- Reaction time
- Matrix multiplication
- Tone generator
- Concurrency

4.5.1 Minimal application

The first immediate goal for the applications in both C and Rust is to create a minimal application. This minimal application will serve as a base for the other applications, as well as a testing ground for the different tools and libraries used. For all of the hardware, this minimal application will be a simple ”blinky” program whose only purpose is to make an LED blink on a regular interval. This blinking would be induced by the RTOSs, meaning that tasks should be in charge of the LEDs. This approach will allow us to test the feasibility of FreeRTOS and RTIC on all of the microcontrollers, but also give us a good starting point for all the other applications.

4.5.2 Reaction time

One important aspect of real-time systems is the ”reaction time” for important tasks. It might be desirable that an important task is run immediately upon some sort of event. For example, when the brake pedal in a car is pressed the car should engage the brakes immediately and not have a long delay. This means that real-time systems need to be able to handle an abrupt change in the executing task quickly and without much delay.

To test how big the delay is the *Reaction time* application will be built to measure the time delay created between an event and when the system has started the task associated with the event. In the case of the *Reaction time* application, the event is an interrupt, and further explanations of the metric tested by the *Reaction time* application is given in Section 4.6.4.1.

An overview of the *Reaction time* application is shown in Figure 4.1. To start, the application spawns background tasks, which will run on the microcontroller and simulate a system with other work being done. These background tasks will be simple and take up Central Processing Unit (CPU) time by blocking for a randomized time between 1 and 10 milliseconds. They will then sleep for a randomized number of milliseconds before simulating work again. This is shown in Figure 4.3 where a random work time and sleep time are calculated, then the task loops for the work time simulating work, followed by the task sleeping and lastly having it loop back. The number of background tasks running can vary to give a good insight into how

an increase in workload might affect response time. Details, such as, how many background tasks will be used and how the tests will be conducted, can be found in Section 4.7.4. Then the application will then spawn a task that creates an interrupt at an interval of 2 seconds, this is the interrupt spawner in Figure 4.1. It records the time when the interrupt is triggered which is then used by the interrupt task, shown in Figure 4.2, to measure the time delay between when the interrupt occurs and when the called task by the interrupt starts execution. The interrupt task then clears the interrupt and loops back to wait for the next interrupt.

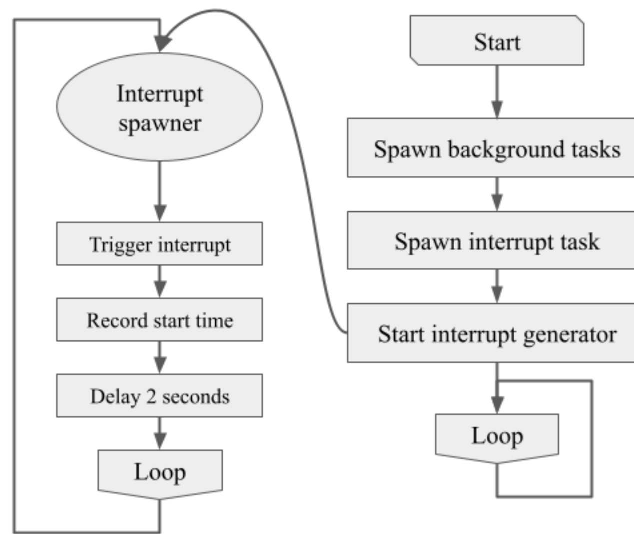


Figure 4.1: A flowchart describing the execution of the *Reaction time* application.

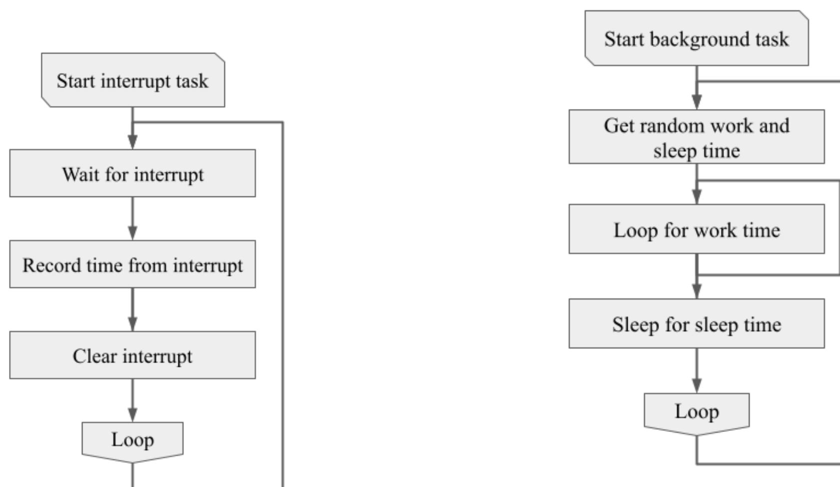


Figure 4.2: A flowchart describing the interrupt handler used in the *Reaction time* application.

Figure 4.3: A flowchart describing the execution of a background task used in the *Reaction time* application.

4.5.3 Matrix multiplication

In a real-time system, computation should be done in a relatively quick manner. This allows for more tasks to run, and in turn more room for unexpected critical tasks.

With the *Matrix multiplication* application we want to test how efficient each language and RTOS is at utilizing the performance of the microprocessor. Since matrix multiplication is computationally strenuous but also easily parallelizable the application can be built utilizing the ability of real-time systems to create and handle multiple concurrent tasks. The use of concurrent tasks allows us insights into how the RTOSs handle multiple tasks and the cost of changing between two tasks which is called context switching. More information on this is given in Section 4.6.4.2.

The matrices used in the application will vary in size and complexity. To investigate the difference between having fewer larger tasks and having more smaller tasks. This is done by having one matrix that has few rows and many columns, since many columns in the first matrix and many rows in the second results in a matrix with many rows, and that in turn means many tasks. The other way with few columns in the first matrix and few rows in the second will result in fewer tasks. This shows the performance with a few heavy tasks instead of many small ones. Finally, a quadratic matrix is used to see what the middle ground between the first two cases looks like. With this application the impact of switching between tasks can be seen since the RTOSs might handle the switching differently and due to language architecture more checks need to be done to ensure correct behavior in this switching.

As shown in Figure 4.4 the parallelization of the matrix multiplication is made by creating multiple tasks before starting the scheduler. With this approach, the creation of tasks does not add extra time to the applications' runtime since the tasks are created before the start time is recorded.

These tasks, as described in Figure 4.5, are supposed to calculate the result of one row in the resulting matrix `C` when multiplying a matrix `A` with a matrix `B`.

$$A \times B = C$$

This means that the work done by each task can be described as:

$$C_i = \{c_{ij} : c_{ij} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + ... + a_{in} * b_{nj}\} = \{c_{ij} : c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}\}$$

Where `i` is the row number in `C` assigned to a specific task and is static for each task, for example, `Task 1` would have $i = 1$. Furthermore `n` is the number of columns in matrix `A` and rows in matrix `B` and and `p` is the number of columns in `B`. This means that for $j = 1, ..., p$ the expression will produce a set that is equivalent to the `i`:th row in the matrix `C`.

A task also records the start time if it is the first task and, if it is the last task, it records the end time. This shows the run time to execute the matrix multiplication. To show how much time the use of concurrency with the use of RTOSs might save or add, a reference application is used. The reference application is similar to the *Matrix multiplication* application but instead of utilizing multiple tasks, only one task is used to calculate the entire resulting matrix `C`. This difference between the time it takes for the reference application and the *Matrix multiplication* application is the time the scheduler either adds or saves. The difference between the implementation in RTIC and FreeRTOS shows if one language handles the scheduling better.

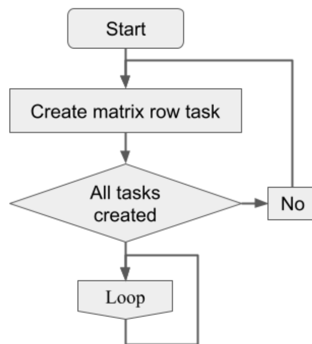


Figure 4.4: A flowchart describing the execution of the *Matrix multiplication* application.

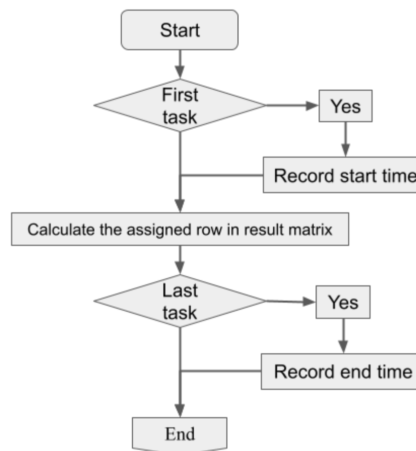


Figure 4.5: A flowchart describing the execution of the tasks created by the *Matrix multiplication* application.

4.5.4 Tone generator

The need for real-time systems to be precise and run tasks at the moment they are needed is one of the major reasons why RTOSs are made. One case where the precision of this execution is crucial is when creating sound. The *Tone generator* application is made to test this use case of RTOSs and to see how it is handled by FreeRTOS and RTIC respectively.

As seen in Figure 4.6 the application alternates between writing either 0 or *volume* to a register connected to a DAC. *Volume* in this case is an integer greater than 0 and represents the magnitude of the signal produced. The writing is done at a specific interval to create a tone at a specified frequency, as seen in Figure 4.8. With no disturbance (in the form of background tasks that will use a blocking sleep to take up CPU time) the desired frequency should be the only one produced. The background tasks, as described in Figure 4.7, will be simple tasks that take up CPU time by blocking for a millisecond and then doing a non-blocking sleep for some time.

To see how well the different RTOSs handle the disturbance from background tasks and still be able to produce a tone, the number of background tasks is varied. The

result can then be analyzed by computing a Fast Fourier Transform (FFT) over the output signal. The *Tone generator* application is further described in Section 4.6.4.3.

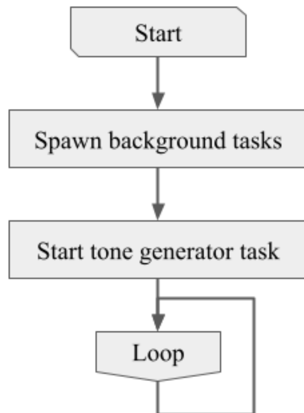


Figure 4.6: A flowchart over the *Tone generator* application.

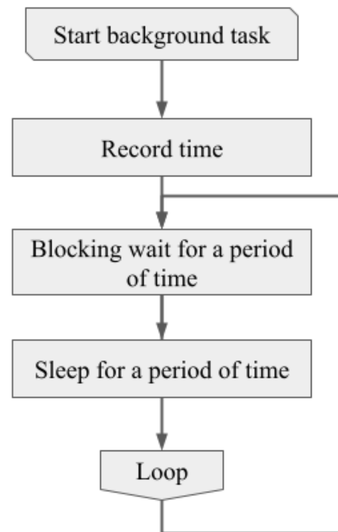


Figure 4.7: A flowchart over the background task used in the *Tone generator* application.

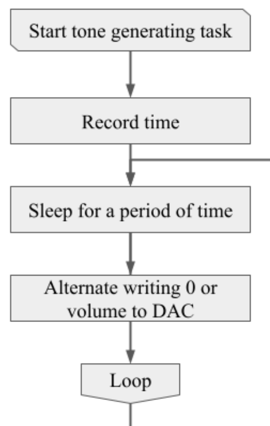


Figure 4.8: A flowchart describing the execution of the task that writes to the DAC and creates the tone.

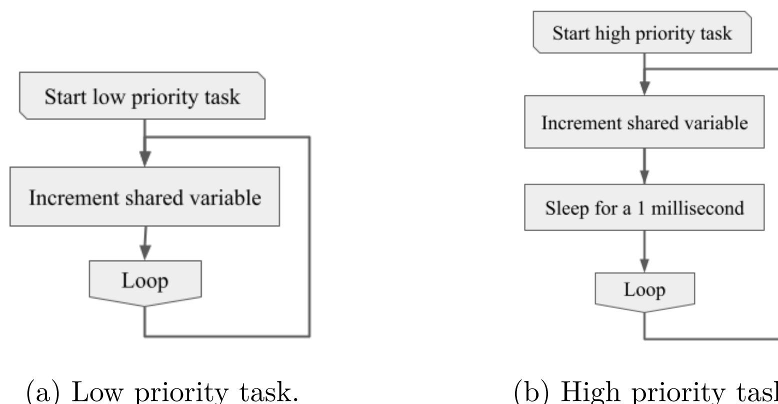
4.5.5 Concurrency

The *Concurrency* application focuses on how the synchronization mechanisms work in FreeRTOS and RTIC, but also how they impact the performance of the RTOSs. To do this, two different types of tasks are used. They are referred to as the *low-priority* and the *high-priority* tasks and are illustrated in Figure 4.9. When the application is executed there exists one instance of each task. These tasks have different levels of priority with the high-priority task having a greater priority than the low-priority

task. The low-priority task is implemented to continuously write to a variable shared between the two tasks, while the high-priority task writes to the shared variable at an interval. When the shared variable has gone from 0 to a specified value the total run time is recorded.

To increment the shared variable the same function is used between the tasks. This is done for the implementation in both Rust and C. This function is described in Figure 4.10. The function starts by locking the shared variable so that the other task can not access it. The function then increments the shared variable and checks if it has reached the specified value, which in this case is 100,000. If so, the time is recorded and the resource is released and the application ends. Otherwise, the resource is released and the application schedules the next task to run.

A reference task is used to record how much time the synchronization mechanisms adds. The reference application is similar in structure but instead of two tasks with different priorities, there is only one task that continuously increments a variable. By recording the run time for this reference application the difference between the reference application and the *Concurrency* application can be shown and subsequently, the added time lost from synchronizing tasks.



(a) Low priority task. (b) High priority task.

Figure 4.9: Flowcharts describing the execution of the tasks used for the *Concurrency* applications.

4.6 Metrics

In this section, the metrics used for comparing Rust/RTIC with C/FreeRTOS are listed, as well as an explanation for why the metric is chosen. In addition to the metrics, a short description of the testing methodology is given, but a more extensive explanation of the implementation of the tests is given in Section 5.

4.6.1 Size of binary

As explained in Section 4.4, real-time systems may need to operate in a resource-constrained environment. To be able to run applications on microcontrollers with minimal resources the program files need to be as small as possible to allow for

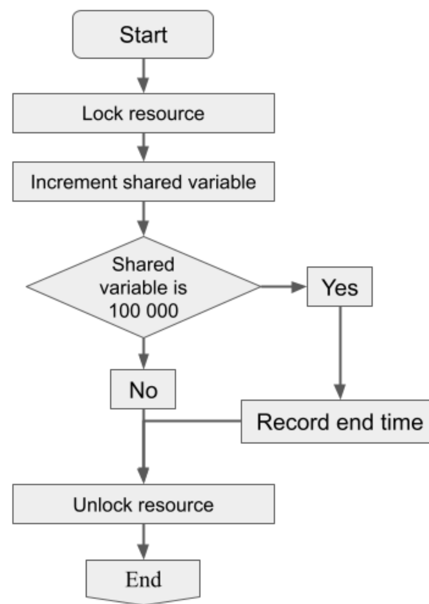


Figure 4.10: The function for incrementing the shared variable for the *Concurrency* application.

resources to be allocated to the task at hand but also to fit in the memory on small microcontrollers.

To test this, the binaries that are compiled for both languages on all applications are compared in terms of memory size. The compilers used for Rust and C both have an optimization level parameter. This means that the level of optimization can be changed. With this parameter, the compiler can change the binary's structure to be optimized not only for speed and execution time but also for size and the combination of the two.

4.6.2 Compile time

Compile time is interesting since it will give some insight into what development is like with the different languages. A long compile time slows down the development since it takes longer to test code when changed. So it is interesting to see how the security mechanisms in Rust affect the compile time and the development process compared to C. This metric does not affect the use in real-time systems but it is interesting in the realm of developing software and to give some statistics for the discussion.

4.6.3 Memory usage

One of the advantages of Rust is the memory-safety features as discussed in Section 3.3 as well as the strong typing. However, these features might come with disadvantages such as increased memory usage. Therefore an interesting metric is to compare applications memory usage between the languages. Since no heap memory is used in any of the applications memory comparisons can be made by only analyzing the

stack.

4.6.4 Application specific metrics

This section describes the metrics that will be retrieved from the developed applications.

4.6.4.1 Reaction time

The *Reaction time* application measures the jitter of a system. Jitter in the domain of real-time systems refers to the time between when an action is triggered to when it happens. This is exemplified in Figure 4.11. Here an interrupt occurs and then there is a time delay until the task that the interrupt wanted to trigger starts executing. This time delay is what is called jitter and is what the *Reaction time* application is supposed to measure.

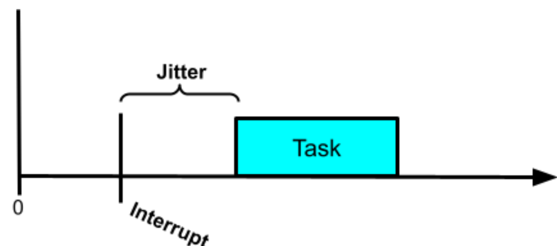


Figure 4.11: An abstraction of jitter between an interrupt and the execution of a task.

This time delay between interrupts and the execution of tasks is desirable to be as small as possible to have a responsive system. A responsive system can be important for safety-critical real-time systems both in terms of reacting to critical situations and also to the predictable workings of the system. For example, the brakes on a car must engage as soon as the brake pedal is pressed to ensure that the car brakes as quickly as possible, but also so that the driver of the car knows that the brakes work right away and can act accordingly.

4.6.4.2 Matrix multiplication

The metric gained from the Matrix multiplication is performance and how context switching is handled. As described in Section 4.5.3, this is done by calculating matrix multiplication on matrices with large floats. Using the reference task described earlier, the additional overhead created by using concurrency by utilizing multiple tasks can be recorded.

Quick context switching and good performance are important to allow more CPU time for the execution of tasks. A short context switch means that less time is spent idle. As can be seen in Figure 4.12, the time used for context switching is time not spent on executing tasks and therefore results in a loss of CPU time. The same goes for the performance. If a programming language can execute more instructions with

fewer CPU cycles, CPU time is freed up to allow for more tasks to run or to ensure that the system functions correctly.

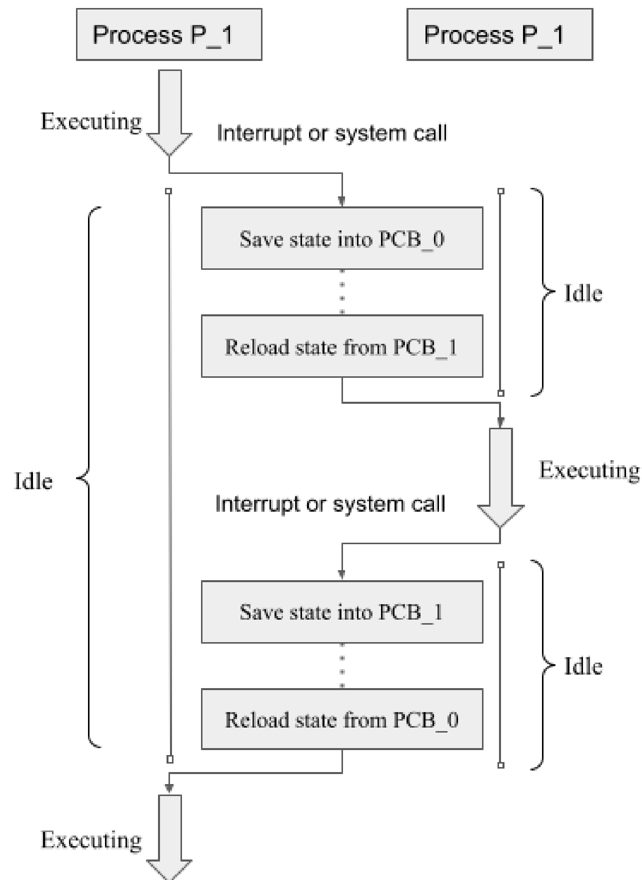


Figure 4.12: Flowchart over a context switch from process P_0 to P_1 and back.

4.6.4.3 Tone generator

The *Tone generator* application will, as with the *Reaction time* application, measure the jitter of the system. The difference from the *Reaction time* application is that, as described in Section 4.5.4, the *Tone generator* application investigates how the systems handle jitter in the context of a continuous signal. For a signal of this frequency, it is acceptable if the frequency deviates a bit since this would barely be noticeable. This application will however measure how many tasks it takes until it becomes a noticeable problem. As explained earlier, the *Tone generator* application creates a tone using a DAC and then generates disturbance by having several background tasks taking up CPU time and disturbing the normal execution. Real-time systems need to be reliable and an important part of that is to be able to schedule a large number of tasks.

To measure how the systems are affected by the background tasks, FFTs are used. An FFT can transform a discrete signal in the time domain to the frequency domain. This means that the FFT shows which frequencies are created by the DAC. So if the

background tasks do not affect the execution of the *Tone generator* task there should be only one spike at the value of the produced tone. When enough background tasks are running alongside the task writing to the DAC, changes can occur in the FFT. If there are multiple spikes in the FFT, it shows that multiple frequencies are produced. If the spike is shifted it means that the tone-generating task is delayed and cannot be executed at the interval to produce the desired frequency.

4.6.4.4 Concurrency

The *Concurrency* application measures the performance of the synchronization mechanisms in the languages and RTOSs. Similarly to the *Matrix multiplication*, a reference application is used to observe the difference between using synchronization mechanisms and not. This is explained in Section 4.5.5. Synchronization is needed to ensure the correct execution of the system and to avoid synchronization problems such as deadlocks or mutual exclusion. Because of this, they are required in systems that support concurrent tasks. Since synchronization mechanisms are needed in real-time systems to ensure the correct execution of systems using concurrency it is favorable for those mechanisms to be fast and efficient.

4.7 Conduction of tests

This section gives an overview of how the application and metric tests will be conducted.

4.7.1 Size of binary

The binary size metric will be investigated by compiling a binary and then retrieving the size using the command line tool *Disk usage* on Linux.

4.7.2 Compile time

The compile time will be tested on all applications for each hardware and the compilation optimisation level will be set to Os. The tests will be run 30 times for each application and then the average value, the median value, the highest value, and the lowest value will be calculated. The compilation tests will all run on the same computer, a ThinkPad T14s Gen 2 laptop with an Intel Core processor (i7-1165G7).

4.7.3 Memory usage

The memory usage for the applications will be polled at different times during the application execution and the largest value extracted will be taken as the memory usage. The method for extracting the memory used by the applications is done differently for the Rust and C versions. The methods used are described in Section 5.3.5 and Section 5.2.5 respectively.

4.7.4 Applications

The tests for the *Reaction time* application will consist of running the application with 10, 30, 50, 70, and 100 tasks on the Pico and MD407. Since the Discovery can not handle that many background tasks, it is run with 1, 3, 5, 7, and 10 background tasks instead. The application is run for 20 interrupts which results in 20 separate values for the jitter or time delay between the interrupt and the task starting to execute. These tests are then done 3 times for each level of background tasks, so there are 60 values for 10 background tasks and so on. Then the average of all the results is calculated.

Three different pairs of matrices will be used to test the *Matrix multiplication* application. These can be found on Github [53]. The first pair, called `A`, consists of a 40x8 matrix multiplied by an 8x8 matrix which results in a 40x8 matrix. This will, as described in Section 4.5.3, test the application with many but smaller tasks. Then there is a pair, called `C`, with an 8x8 matrix and an 8x40 matrix which results in an 8x40 matrix. This test has a small number of tasks but instead, the tasks have more work that they each need to do. The last pair, called `B`, are two 15x15 matrices which result in a 15x15 matrix. These represent an in-between test for the first two pairs since there are not as many tasks as the first and not as much work as the second test. These pairs of matrices are used on the Pico and MD407, but for the Discovery other smaller matrix pairs are used since the memory on the Discovery is too small. The pairs of matrices for the Discovery are 5x2 multiplied with a 2x2, two 3x3, and 2x2 multiplied with a 2x5. All three of the tests are repeated 50 times and then the average is used as the result for each test.

To test the *Tone generator* application it will be run for 30 seconds, during which the tone is captured. This is repeated 3 times. This data is then used to create FFTs, as explained earlier in Sections 4.5.4 and 4.6.4.3. The mean of the three created FFTs is then used to generate a final FFT. This is done to prevent random disturbances from affecting the result. A more in-depth explanation of how the tone will be sampled can be found in Section 5.1.

The *Concurrency* application will be tested with different levels of compilation optimization The levels used are `Oz`, `Os`, `O0`, `O1`, `O2`, and `O3`. Both `Oz`and `Os` optimize the application in terms of binary size, i.e. smaller binaries are created. `Oz` is more aggressive than `O0`. The other optimization levels optimize for speed in an orderly fashion, where `O3` is the fastest and `O0` is the slowest. Both the reference and the *Concurrency* applications will be run 50 times on all optimization levels for both languages. Then the average of those 50 runs is calculated.

5

Implementation

In this chapter, the development of the different applications is described. First, an overview of the tools used are given, followed by a deeper look into language-specific implementations and the final implementation of the applications.

5.1 General tools and analysis methods

We are using a debug probe named the Raspberry Pi Debug Probe to aid us in the development process. This probe is made specifically for the Pico, but can also be utilized for the other hardware. It is possible to read the Universal Synchronous/Asynchronous Receiver/Transmitter (USART) pins on the hardware using the probe to capture any output from these pins. The probe can then be connected to a computer, which makes it possible to read the output of the hardware in real time. This allows us to capture information such as execution time, stack usage, and so on from our hardware in a simple manner.

For the Pico, the debug probe is also used to flash the applications to the microcontroller. Without the probe, one would need to flash the application manually by connecting the Pico to a USB port on a computer, and then moving the application from the computer to a newly established mount point for the Pico in the laptop's file system. This is tedious when running multiple applications many times, so the debug probe was a great aid in this regard.

Another tool that is used is the Saleae Logic Pro analyzer. This analyzer could be used to act as an oscilloscope, which could then in turn be used to measure the frequency of the *Tone generator* application. According to signal processing theory, the sampling rate must not be lower than the Nyquist Rate, which is at least two times faster than the bandwidth of the signal. Otherwise, it is not possible to recreate the original signal from the sampling alone.

The Saleae Logic Pro analyzer can be scripted using Python tools, which makes it possible to automate these tests. However, there are restrictions in the Python tools that enable only a predefined set of sampling rates that could be used. This meant that the sample rate had to be quite high compared to the frequency and thus resulted in more data that would take longer time to process. This is not a large problem, since the data is still small enough to be able to calculate an FFT in a relatively short time.

The development is done using Git [54]. Git is a tool for version control, meaning that a history is maintained of the code produced. This also means that if a bug is introduced somewhere that breaks an application, it is easy to roll back to a version where the application worked.

5.2 C tools and general implementations

In this section, the toolchains and libraries used in the C development will be described along with justifications for them, and then some more technical explanations as to how to run the RTOS as well as how to capture our metrics.

5.2.1 Compiler and buildsystem

There exist multiple C compilers. The C compiler that is used in this thesis is the GCC compiler, as it is capable of compiling to the architecture we use (ARM) as well as to the CPUs and is the most commonly used compiler [4]. The choice stood between GCC and Clang, as both compilers are equal in terms of what is needed from a compiler. The Raspberry Pi Foundation recommended GCC for Pico development, which made it the better choice in the end [55].

5.2.2 Libraries

To interact with the hardware, the standard peripheral libraries as supplied by STMElectronics were used [56]. STMElectronics are suppliers of microcontrollers using Cortex processors, which all of the hardware uses. These gave us the ability to interact with the peripherals such as General Purpose Input Output (GPIO), Nested Vector Interrupt Control (NVIC), and External interrupt (EXTI), without having to manipulate the memory directly or implement abstractions on our own.

The Pico technically uses a Cortex-M0 processor, which would in theory make it possible to use the same libraries as for the other hardware. However, for the Pico there exists a Software Development Kit (SDK) that is provided by the Raspberry Pi foundation. This also gave us access to peripherals, but also other functions such as the function *time_us_64*. This function provided the time elapsed since the system started in microseconds, which meant that we would not need to implement any other timer functions to extract the time.

5.2.3 Running FreeRTOS

FreeRTOS requires a config file to work correctly. In this file, one specifies what kind of functionality is requested from FreeRTOS. For example, the FreeRTOS functions used in the application need to be specified. In this config, one might also need to configure the hardware interrupts used by FreeRTOS: SVC, SysTick, and PendSV. For FreeRTOS to schedule its tasks correctly, FreeRTOS needs to place its functions in these interrupt handlers. This could be done using what is called ”weak handles” (which are not the same thing as the handlers themselves) in the linker script. Weak

handles sort of export the name of the interrupt handler from the linker script to the application, so that it can be used in code. If you define a function with the same name as the weak handle, then the function will be placed in the interrupt vector for that hardware interrupt by the linker script.

The approach of using weak handles worked without any larger problems for all of the microcontrollers, except for the MD407. The MD407 has a relocated vector table, which means that the vector table is moved to another location which in turn means that FreeRTOS has trouble finding where to put its interrupt handlers. To fix this, the Vector Table Offset Register (VTOR) needs to be modified to match the actual location of the vector table. Then, the FreeRTOS functions are manually loaded into their respective memory address, which then enables FreeRTOS to run without any errors.

Another requirement to get FreeRTOS running is to use the correct "port file". The port file describes the hardware for FreeRTOS and enables FreeRTOS to interact with the hardware correctly. It also lets FreeRTOS know which hardware functionality is available to use.

5.2.4 Flashing and debugging

For flashing, different techniques are used for each microcontroller. For the MD407, the K1 port is utilized. This port is connected to a chip on the microcontroller that can simulate a serial port on another computer that is connected to it using a USB cable. Using this serial port, it is then possible to send instructions to the microcontroller. For example, one can send the command "load" along with the data of the compiled binary. This will load the application on the MD407. To execute the application, one can then simply issue the "go" command. To reset the chip to load a new program, one simply presses the reset button on the microcontroller which restarts the microcontroller and clears its memory. It is also possible to step line-by-line and inspect memory addresses using this built-in chip.

To flash the Discovery card, the tools STLink and OpenOCD are used. STLink is a debugger for STM32 microcontrollers, and OpenOCD (Open On-Chip Debugger) is an open-source debug solution. The Discovery card has an embedded STLink debugger, which can be communicated with using OpenOCD. With this setup, it is not only possible to flash the applications to the Discovery card but also to examine memory locations as well as step through single instructions.

Another technique for debugging is writing to a terminal from the microcontroller with relevant information. For all of the microcontrollers, this can be done using USART communication. The K1 port on the MD407 supports USART communication, but for the Discovery card and the Pico, the debug probe is used to read and write from USART.

5.2.5 Analysis

To measure timings, the built-in hardware timers are used. A first approach was to use RTOS tasks to count the time on a regular interval. However, this interfered with other tasks by using unnecessary processing power. The hardware timers use a resolution of 1 microsecond and are initialized before the FreeRTOS scheduler is started. The value of these timers can then be read throughout the entire execution of the FreeRTOS application. The timer peripheral that is being used is the one with the highest counter register, which is 64 bits for the Pico and 32 bits for the others. This means that the timers will not overflow until around 584,000 years for the 64-bit timer, or a little more than an hour for the 32-bit timers which is more than well enough for the chosen tests.

The applications will as stated in Section 4.6.3 only use static memory allocations. This aids us in having more control over memory usage and thus allows us to analyze memory usage more easily. With GCC it is possible to generate files that describe the stack usage for each function in any given *.c* file. Alongside this, there also exists a tool called *cflow* that generates a sort of flowchart of all the function calls. At a first attempt, it was attempted to combine these two methods to extract the stack usage of the applications. However, it was hard to get a correct output since recursion made the method unreliable.

Instead, the method of using the built-in FreeRTOS function `uxTaskGetStackHighWaterMark` was chosen. This function made it possible to extract the max stack usage for each task in the RTOS. However, this has the disadvantage of not knowing the stack usage of the system as a whole (including the scheduler). But this was the most accurate method we could find, and during our development, we concluded that tasks were the main source of stack usage.

5.3 Rust tools and general implementations

In this chapter, the tools and general implementations of the Rust applications will be described similarly to Chapter 5.2.

5.3.1 Compiler and build system

The standard compiler used in Rust is *rustc*. *rustc* has three different release channels: *nightly*, *beta* and *stable*. To compile the Rust code, the *nightly* channel is used. The *nightly* channel is regarded as unstable since it receives updates “nightly” as the name suggests with features that are not tested properly. The reason for using the nightly channel is to be able to use some extra features that aid the development, which would not be possible otherwise.

In addition to this compiler, it is needed to specify a target architecture. This can be configured with the usage of *rustup*, which installs the desired compiler targets from official channels. For the Cortex-M0 cores, the *thumbv6m-none-eabi* target is used, while for the Cortex-M4 the *thumbv7em-none-eabihf* target is used.

The compile times for Rust, compared to C, are generally slower. This is because the aforementioned borrowing and ownership rules are complicated to verify. However, *rustc* also has incremental compilation. This means that *rustc* saves information when compiling the program that can be reused at a later compilation time, thus reducing compile times significantly for incremental updates.

5.3.2 Libraries

The libraries (or crates, as they are called in Rust) used are kept at a minimum. Crates are provided by the centralized crate repository called ”crates.io”, where crate dependencies can easily be added by either adding the required crate to the Cargo configuration file for the project or via the command line. Crates also has a feature called *features*, that enables certain features to be added or removed from crates depending on what functionality is needed from the crate. This removes unnecessary code, as it is possible to only enable the features that are needed.

To interact with the hardware, a HAL can be used. These HALs are supplied by crates and provide abstractions to be able to interact with the hardware without having to access the hardware using memory addresses directly. There exists a HAL crate for each microcontroller that is used. The STM HALs are more generic and aim towards all STM32F4 and STM32F0 microprocessors, but using features it is possible to only use the relevant microprocessor (STM32F407 and STM32F051). The crate *cortex-m* grants more access to the hardware, which is necessary for using, as an example, critical sections. Additionally, the crate *cortex-m-rtic* provides an implementation of the RTIC framework for cortex cores, which all of the cores on all hardware are.

There are also crates needed to get the monotonic timer working correctly in RTIC. For the MD407 and the Discovery, there is the *systick-monotonic* crate that provides a means of using the systick timer as a monotonic timer. For the Pico, the *rp2040-monotonic* crate exists that converts the only timer the Pico has to a monotonic timer.

Apart from these libraries, some hardware-specific crates are used. For example, the MD407 needs the *rand_core* crate to utilize the built-in random number generator efficiently. There is also a crate called *defmt*, that was useful during development to easily print logs to the terminal without using USART. For the final applications, it was not used since USART was more reliable.

5.3.3 Running RTIC

To get RTIC running on the microcontrollers, at least one interrupt line needs to be provided to the RTOS. As described in Section 3.4.2.3, RTIC uses interrupt lines to schedule its tasks. The choice of interrupt lines was chosen arbitrarily among the lines not used by the applications themselves, as the type of hardware interrupt has no impact on performance or behavior. One also needs to provide a Peripheral Access Crate (PAC) that describes the hardware. The PAC for each microcontroller

could be found in each HAL crate used for the microcontrollers. This was not a problem for any of the microcontrollers used.

To enable the functionality of spawning tasks after some time, one needs to provide a timer to the RTOS as stated in Section 3.4.2.1. This is a problem when it comes to the Pico, as the Pico only has one peripheral timer. To enable the functionality to schedule tasks at a specific time, the timer needs to be provided to RTIC. But because of Rust's ownership and borrowing rules, in doing so one cannot also use the timer for other purposes, such as measuring time manually. To circumvent this, the addresses of the timer registers needed to be saved and cloned using raw pointers and then accessed using unsafe raw pointer dereferencing. This allows the timer to be used for both scheduling RTIC tasks periodically and also to measure timings in the RTOS for the tests.

5.3.4 Flashing and debugging

To flash the Rust applications on the microcontrollers, there exists a tool called *probe-rs*. *probe-rs* is both able to flash Rust programs and handle logging to the terminal. This did not however work with the MD407, but for the MD407 it was possible to use the same approach as for C, described in Section 5.2.4.

There are also problems with using *probe-rs* for the Pico. There was trouble with having the logging (using the aforementioned create *defmt*) and flashing functionalities work at the same time, for some unknown reasons. The tool was inconsistent, with it working sometimes but at other times not at all. Because of this, we opted to use *probe-rs* for the flashing and then rely on the same technique described in Section 5.2.4 for logging purposes, using USART.

5.3.5 Analysis

To capture timings, the same method as for the C implementations is used, where the hardware timers are configured to tick each microsecond. To analyze the memory usage, the stack pointer is used. The start of the stack could be determined using the linker script. Then, at various times during the execution of an application, the stack pointer value is checked. Since the stack grows downwards in memory, the lowest stack pointer value captured during the execution is stored in a variable. Then, at the end of the execution, this variable is read to determine the maximum stack usage reached during execution using the stack pointer start value. This method was possible to use for the Rust programs but not the C programs. This is because FreeRTOS utilizes two different stack pointers (the MSP and the PSP) and it was hard to extract the meaning of the stack pointer values as well as the origins of these.

5.3.6 Creating tasks

As mentioned in Section 3.4.2.1, it is not possible in RTIC v2 to schedule multiple instances of the same task. Since most of the applications require multiple instances of the same task, message queues are utilized to achieve the same goal.

The reaction time application will be used as an example to illustrate this. In this application, a beforehand specified number of background tasks needs to be created. To create these tasks, a task that will act as a ”task creator” is created. This task has the send-end of a message queue, while the background task has the receive-end. The background task waits on the channel until a message is received, while the ”task creator” task sends messages to simulate a task creation.

5.4 Applications

In this section, the applications and their implementations in the two different languages are described.

5.4.1 Reaction time

The reaction time application's purpose, as can be recalled, is to periodically issue an interrupt while simultaneously having background tasks running, and recording the time it took to execute. The interval that is used for the interrupt is two seconds, which means that an interrupt is triggered exactly once every other second.

C

In FreeRTOS, hardware interrupts are not handled by the RTOS. Instead, hardware interrupts are handled by normal C functions that are placed in the correct hardware interrupt handler. If we were to benchmark the time it took for these hardware interrupt handlers to trigger, we would just be measuring C's performance and not the RTOS's performance. Additionally, when the execution takes place within a hardware interrupt handler, certain FreeRTOS features are locked and cannot be used (or a special version of the feature needs to be used).

We opted for the solution of using a FreeRTOS task that is awoken by the interrupt handler to circumvent this. The FreeRTOS task is waiting on a binary semaphore, which is lowered by the interrupt handler, and then the time it takes for that to happen is measured. Using this method means that two different metrics are extracted: the time it takes for the C interrupt to occur, and the time it takes for the FreeRTOS task to be awoken.

Rust

In RTIC it is possible to have a task handle a hardware interrupt, contrary to FreeRTOS where this is not possible. This means that we only need to have two tasks: one task which periodically sends a hardware interrupt, and one task that is awoken by that hardware interrupt. Thus, it is quite easy to implement this application.

5.4.2 Matrix multiplication

C

The implementation of the matrix multiplication application is fairly straightforward. Each task is assigned a row to calculate in the result matrix and tasks are all created before the scheduler starts. When the matrix is calculated, the finish time is recorded. However, when running the first version of the application we noted that it executed way too quickly. We realized that the C compiler optimized away the result matrix, since it was never read, meaning that the computations never took place. To fix this, the result matrix needed to be attributed to the `static` keyword. Simplified, this keyword forces the variable to not be optimized away.

When it came to the MD407, it had the advantage of having a Floating Point Unit (FPU). This means that the floating point operations could be calculated using the hardware. This FPU however can only handle normal floats of 32 bits, not 64-bit floats (also known as doubles). To capture the performance of the FPU, the applications are run with both floats and doubles to see the performance difference.

Rust

The first version of the matrix multiplication application has similar problems to the one in FreeRTOS, where the entire calculation is optimized away by the compiler since the result is never used. To mitigate this, Rust has a function called `black_box`. Simplified, this function takes a variable of any type as input and then tries to prohibit any optimizations to the passed variable. If a pointer to our result matrix is passed into `black_box`, then the result matrix is not optimized away and the calculations are done.

5.4.3 Concurrency

Since the purpose of the application is to increment a shared variable to a certain value, that value needs to be the same for all applications. The limit chosen for the shared variable was 100,000, as explained in Section 4.5.5.

C

The concurrency application utilizes the FreeRTOS mutexes, which are binary semaphores with a priority inheritance mechanism to eliminate priority inversion. This ensures that the shared variable can be accessed without any data races.

Similarly to the first version of the matrix multiplication application, this shared variable is found to be optimized away by the compiler since it is also never read. The fix for this is the same as for the matrix multiplication application, adding the `static` keyword to the shared variable.

Rust

To use shared resources in RTIC, one must lock them. Thus it is impossible to create this application without using locks and synchronizing the reads and writes. The implementation of this application does not deviate from the specification given in the previous chapter. This application does also not suffer from the optimization problem that the version in FreeRTOS did. The shared variable for some unknown reason is never optimized away in this version, and as such no measures are needed to prevent that.

5.4.4 Tone generator

The frequency that is used in tests for this application is 247Hz, which has a period of 4048 microseconds. The reason for this period being chosen is partially arbitrary, but also because it produces a pleasant sound for the ear. Another reason is that it is not a very round number, since it has that extra 48 microseconds from becoming 2 milliseconds. This adds an increased granularity, which might lead to some extra strain on the RTOSs. This also means that the sampling frequency that is used needed to be at least two times 247Hz, or about at least 500Hz which results in a minimum sample rate of 2 milliseconds.

C

The implementation of the tone generator application is not trivial on the FreeRTOS side. The frequency that is generated uses a period of 4048μs, as stated before. The granularity of the FreeRTOS scheduler is 1ms, which means that it is not possible to have a task be awoken every 2024μs. An additional hardware timer needs to be used to circumvent this. This hardware timer can trigger an interrupt each 2024μs, and then wake up the associated FreeRTOS task that writes to the DAC. To awaken the task, the task is waiting on a ”notification”. A notification is a type of synchronization method, where one task may wait on a notification and another one can send it. The task waiting is blocked until a notification is received. So when the task receives a notification, the task is awoken and can continue executing by writing either a one or a zero to the DAC and generating the signal.

Rust

RTIC has sufficient granularity for its scheduler, which means that it is possible to generate the desired tone using only the RTIC tasks. It is however not easy to configure the DAC to work correctly, compared to C. There are no abstractions in the HAL that are used to write to the DAC register that is desired, so this has to be done manually using *unsafe* Rust.

6

Results

This chapter presents the results from running the applications given in Section 4.5. The results that are presented are the metrics that the applications would be tested for. These are described in Section 4.6.

First, the results for the binary size are presented in Section 6.1. Then compile time is presented in Section 6.2 and memory usage is given in Section 6.3. Lastly, the results for the application-specific metrics are presented.

6.1 Size of binary

In this section, the binary sizes for each application are given. They are presented for each hardware and the result for FreeRTOS and RTIC version 1 (and for the Pico version 2 as well) is given. The binary sizes are given in bytes because the difference between binaries is small enough that they might be rounded to the same number.

The results related to the binary sizes of the *Matrix multiplication* application are given in Tables 6.1, 6.2 and 6.3. The test A, B, and C represent the 3 different pairs of matrices used as discussed in Section 4.7.4.

From the tables mentioned in the last paragraph, it can be seen that the Rust and RTIC binaries are larger than those for C and FreeRTOS. The difference varies between RTIC being 1.8 to 4.25 times greater in size compared to FreeRTOS. The largest difference can be seen in Table 6.3 where the *Matrix multiplication* application is run on the Pico. The difference is around 537 600 bytes for both RTIC versions 1 and 2. It can also be seen from the tables that the binaries are larger for both FreeRTOS and RTIC compared to the ones on the MD407 and Discovery.

MD407	**FreeRTOS**	**RTICv1**
Test A	59,444B	109,092B
Test B	60,980B	111,196B
Test C	59,396B	110,180B

Table 6.1: Binary sizes for *Matrix multiplication* on MD407. The values are given in bytes.

Discovery	**FreeRTOS**	**RTICv1**
Test A	23,612B	82,640B
Test B	23,620B	82,384B
Test C	23,612B	82,356B

Table 6.2: Binary sizes for *Matrix multiplication* on Discovery. The values are given in bytes.

Pico	FreeRTOS	RTICv1	RTICv2
Test A	164,680B	702,608B	704,320B
Test B	165,192B	702,672B	702,528B
Test C	164,680B	702,172B	703,916B

Table 6.3: Binary sizes for *Matrix multiplication* on Pico. The values are given in bytes.

MD407	FreeRTOS	RTICv1
Os	54,228B	62,872B
Oz	54,228B	67,116B
O3	72,260B	38,176B
O2	64,116B	39,012B
O1	63,252B	89,204B
O0	203,988B	124,640B

Table 6.4: Binary sizes for *Concurrency* on MD407. The values are given in bytes.

Discovery	FreeRTOS	RTICv1
Os	24,516B	41,008B
Oz	24,516B	40,648B
O3	30,076B	41,488B
O2	30,396B	42,160B
O1	27,940B	89,204B
O0	Not relevant	Broken

Table 6.5: Binary sizes for *Concurrency* on Discovery. The values are given in bytes.

The results related to the binary sizes of the *Concurrency* application are given in Tables 6.4, 6.5, and 6.6. The first column gives the optimization level used.

In most of the tests, the FreeRTOS binaries are smaller than the RTICv1 counterpart with the ones on for the Pico having the largest difference. For the Discovery the difference is smaller but the FreeRTOS binaries are always smaller. In contrast to the other binaries the RTICv1 binaries for the MD407 are smaller than the binaries for FreeRTOS for the optimization levels `O3`, `O2` and `O0`. On the other optimization levels, FreeRTOS still has the smaller binaries.

The results related to the binary sizes of the *Reaction time* application are given in tables 6.7, 6.8 and 6.9. The binaries for FreeRTOS are smaller than RTICv1 and RTICv2 for all the tests and are almost always the same. A large difference can be seen between the 1-task test on the Discovery and the 10-task test for the Pico.

Pico	FreeRTOS	RTICv1	RTICv2
Os	161,624B	683,748B	685,304B
Oz	161,624B	689,808B	690,148B
O3	171,540B	681,420B	682,844B
O2	163,480B	683,404B	683,580B
O1	164,576B	687,692B	687,748B
O0	202,292B	801,572B	813,920B

Table 6.6: Binary sizes for *Concurrency* on Pico. The values are given in bytes.

MD407	FreeRTOS	RTICv1
10 tasks	56,612B	63,828B
30 tasks	56,612B	63,348B
50 tasks	56,612B	63,828B
70 tasks	56,612B	64,308B
100 tasks	56,612B	65,012B

Table 6.7: Binary sizes for *Reaction time* on MD407. The values are given in bytes.

Discovery	FreeRTOS	RTICv1
1 tasks	26,336B	42,892B
3 tasks	26,360B	43,020B
5 tasks	26,360B	43,148B
7 tasks	26,360B	43,116B
10 tasks	26,360B	43,276B

Table 6.8: Binary sizes for *Reaction time* on Discovery. The values are given in bytes.

Pico	FreeRTOS	RTICv1	RTICv2
10 tasks	163,896B	681,384B	686,276B
30 tasks	163,788B	681,864B	686,340B
50 tasks	163,788B	682,344B	686,340B
70 tasks	163,788B	682,824B	686,340B
100 tasks	163,788B	683,560B	686,344B

Table 6.9: Binary sizes for *Reaction time* on Pico. The values are given in bytes.

6.2 Compile time

In Table 6.10 and Table 6.11, the compile times for the applications written in C and Rust respectively are presented. The compile times for the Rust applications are longer than those for the C applications.

C	Average	Median	Low	High
MD407 Concurrency	1.46s	1.46s	1.44s	1.49s
MD407 Reaction	1.51s	1.51s	1.49s	1.55s
MD407 Matrix	1.48s	1.48s	1.47s	1.51s
MD407 Tone generator	0.45s	0.45s	0.44s	0.50s
Discovery Concurrency	0.76s	0.76s	0.75s	0.77s
Discovery Reaction	0.79s	0.79s	0.78s	0.82s
Discovery Matrix	0.73s	0.73s	0.72s	0.73s
Pico Concurrency	2.09s	2.09s	2.04s	2.16s
Pico Reaction	5.22s	5.21s	5.14s	5.34s
Pico Matrix	5.23s	5.23s	5.15s	5.30s

Table 6.10: The compile times for the applications in C. The values are in seconds.

6.3 Memory usage

In tables 6.12, 6.13, 6.14 and 6.15 the memory usage for the *Matrix multiplication* applications can be observed. The memory usage is generally larger on the FreeRTOS applications, and lower on the RTIC applications.

Rust	Average	Median	Low	High
MD407 Concurrency	14.62s	14.62s	14.57s	14.68s
MD407 Reaction	14.61s	14.61s	14.53s	14.69s
MD407 Matrix	14.74s	14.74s	14.60s	14.94s
MD407 Tone generator	15.33s	15.33s	15.08s	15.67s
Discovery Concurrency	9.67s	9.66s	9.62s	9.77s
Discovery Reaction	9.63s	9.62s	9.56s	9.71s
Discovery Matrix	9.56s	9.55s	9.49s	9.69s
Pico Concurrency	8.89s	8.88s	8.81s	9.09s
Pico Reaction	8.89s	8.88s	8.84s	9.06s
Pico Matrix	8.42s	8.96s	6.79s	9.31s
Pico v2 Concurrency	9.44s	9.44s	9.35s	9.53s
Pico v2 Reaction	9.48s	9.49s	9.36s	9.57s
Pico v2 Matrix	9.61s	9.60s	9.55s	9.76s

Table 6.11: The compile times for the applications in Rust. All the values are in seconds. The *v2* rows are the ones where RTIC version 2 was used instead of RTIC version 1.

Discovery	FreeRTOS	RTIC v1
A	600B	408B
B	336B	272B
C	216B	224B

Table 6.12: Memory usage in the *Matrix multiplication* application on the Discovery microcontroller. The values are in bytes.

MD407 Floats	FreeRTOS	RTIC v1
A	20,640B	192B
B	7,740B	192B
C	4,128B	200B

Table 6.13: Memory usage in the *Matrix multiplication* application on the MD407 microcontroller using floats. The values are in bytes.

In tables 6.16, 6.17 and 6.18 the memory usage for the *Concurrency* applications can be observed. The memory usage is similar in all of the tests, except for on the Pico where FreeRTOS has an advantage at all optimization levels.

In tables 6.19, 6.21 and 6.20 the memory usage for the *Reaction time* applications can be observed. The memory usage in this application is again generally lower on the RTIC applications compared to the FreeRTOS applications.

6.4 Application specific metrics

The findings for the metrics explained in Section 4.6.4 are given here.

In tables 6.22, 6.23, and 6.24, the results from the reaction time in the *Reaction time* application can be seen. RTIC and FreeRTOS have generally similar reaction times. What is important to note, however, is that a time of 0.00μs does not mean that the interrupt fired immediately. Since the timers have a granularity of 1μs, it is not possible to measure the time between 0 to 1 microseconds. As such, a reaction time of 0.00μs simply means that all the measurements of the reaction time were between

MD407 Doubles	FreeRTOS	RTIC v1
A	20,960B	256B
B	8,400B	176B
C	4,320B	184B

Table 6.14: Memory usage in the *Matrix multiplication* application on the MD407 microcontroller using doubles. The values are in bytes.

Pico	FreeRTOS	RTIC v1	RTIC v2
A	16,800B	10,944B	10,144B
B	6,300B	9,696B	8,752B
C	3,360B	10,888B	9,896B

Table 6.15: Memory usage in the *Matrix multiplication* application on the Pico microcontroller. The values are in bytes.

Discovery	FreeRTOS	RTIC v1
Os	284B	64B
Oz	284B	288B
O3	340B	224B
O2	308B	200B
O1	324B	280B
O0	Broken	Broken

Table 6.16: Memory usage in the *Concurrency* application on the Discovery microcontroller. The values are in bytes.

0 and 1 microseconds for those parameters and the average thus became 0.00μs.

In tables 6.25, 6.26, 6.27, 6.28, and 6.29 the execution time of the *Matrix multiplication* application can be observed. The numbers in the cells represent the time it took to complete the matrix multiplication in microseconds. The “Ref” columns represent the time it took for the reference task to finish its execution, while the “Real” columns represent the time it took for the concurrent version to finish. The ratio column is the RTIC execution time divided by the FreeRTOS execution time. This means that if the ratio is lower than one, RTIC is faster. Otherwise, FreeRTOS is faster.

6.4.1 Concurrency

In the tables 6.30, 6.31, 6.32 and 6.33 the execution times for the *Concurrency* application can be seen. The numbers in the cells represent the time it took to count to 100,000 in microseconds. The “Ref” columns represent the time it took for the reference task to finish its execution, while the “Real” columns represent the time it took for the concurrent version to finish. The ratio column here works the same way as the ratio column in the previous section.

6.4.2 Tone generator

In this section, the results for the *Tone generator* application will be presented. In tables 6.1 and 6.2, the FFT of the output signal with 50 and 190 background tasks respectively in FreeRTOS can be observed. In tables 6.3 and 6.4, the same data is presented but for the RTIC applications.

In an FFT, the Amplitude (on the Y-axis) depicts the occurrence of a frequency. If the amplitude is high, the associated frequency (as seen on the x-axis) occurs

MD407	FreeRTOS	RTIC v1
Os	252B	192B
Oz	252B	352B
O3	228B	272B
O2	228B	312B
O1	316B	336B
O0	388B	1,408B

Table 6.17: Memory usage in the *Concurrency* application on the MD407 microcontroller. The values are in bytes.

Pico	FreeRTOS	RTIC v1	RTIC v2
Os	272B	576B	624B
Oz	272B	712B	632B
O3	280B	768B	704B
O2	264B	728B	696B
O1	264B	648B	696B
O0	344B	2,536B	4,136B

Table 6.18: Memory usage in the *Concurrency* application on the Pico microcontroller. The values are in bytes.

frequently. If the amplitude is low, that specific frequency does not occur frequently.

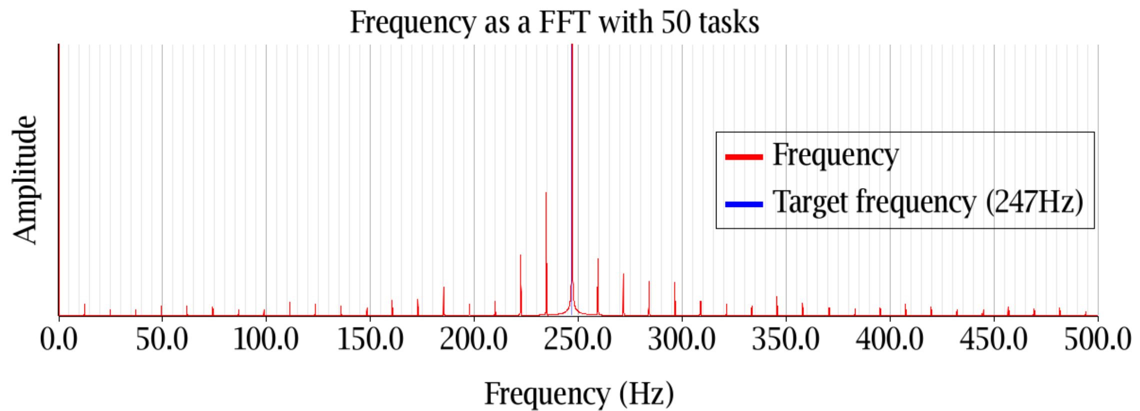


Figure 6.1: FFT of the output signal when using 50 background tasks in FreeRTOS.

Discovery	FreeRTOS	RTIC v1
1 task	476B	200B
3 tasks	660B	312B
5 tasks	844B	328B
7 tasks	1,028B	408B
10 tasks	1,340B	504B

Table 6.19: Memory usage in the *Reaction time* application on the Discovery microcontroller. The values are in bytes.

Pico	FreeRTOS	RTIC v1	RTIC v2
10 task	3,072B	1,816B	776B
30 tasks	8,032B	2,336B	832B
50 tasks	12,992B	2,856B	896B
70 tasks	17,952B	3,376B	3,376B
100 tasks	25,392B	4,168B	4,168B

Table 6.20: Memory usage in the *Reaction time* application on the Pico microcontroller. The values are in bytes.

MD407	FreeRTOS	RTIC v1
10 task	4,596B	304B
30 tasks	12,036B	304B
50 tasks	19,476B	304B
70 tasks	26,916B	304B
100 tasks	38,076B	304B

Table 6.21: Memory usage in the *Reaction time* application on the MD407 microcontroller. The values are in bytes.

MD407	C_{irq}	C_{task}	RTICv1
10 tasks	0.00μs	4.00μs	1.00μs
30 tasks	0.00μs	4.00μs	1.00μs
50 tasks	0.33μs	4.3μs	1.00μs
70 tasks	0.00μs	5.00μs	0.58μs
100 tasks	0.00μs	4.00μs	1.00μs

Table 6.22: The average measured time for the *Reaction time* application on the MD407 microcontroller. The values are in microseconds.

Discovery	C_{irq}	C_{task}	RTICv1
1 tasks	1.77μs	33.87μs	7.87μs
3 tasks	1.68μs	33.80μs	8.00μs
5 tasks	1.63μs	33.82μs	7.98μs
7 tasks	1.75μs	33.83μs	8.15μs
10 tasks	1.77μs	33.85μs	8.33μs

Table 6.23: The average measured time for the *Reaction time* application on the Discovery microcontroller. The values are in microseconds.

Pico	C_{irq}	C_{task}	RTICv1	RTICv2
10 tasks	10.77μs	85.15s	0.85s	7.6667
30 tasks	9.48s	167.02s	0.90s	7.65s
50 tasks	9.70s	282.78s	0.88s	7.60s
70 tasks	9.43s	385.78s	0.85s	7.62s
100 tasks	9.43s	568.18s	1.07s	7.60s

Table 6.24: The average measured time for the *Reaction time* application on the Pico microcontroller. The values are in microseconds.

Discovery	FreeRTOS Ref	FreeRTOS Real	RTICv1 Ref	RTICv1 Real	Ratio Ref	Ratio Real
Test A	229.00μs	270.40μs	414.00μs	425.00μs	1.2	1.3
Test B	158.00μs	347.00μs	567.00μs	573.00μs	1.13	1.28
Test C	229.00μs	241.00μs	423.00μs	423.00μs	1.14	1.34

Table 6.25: The average measured run time for the *Matrix multiplication* application on the Discovery. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

MD407 Double	FreeRTOS Ref	FreeRTOS Real	RTICv1 Ref	RTICv1 Real	Ratio Ref	Ratio Real
Test A	2,085.00μs	2149.00μs	6,551.00μs	6,659.00μs	3.14	3.09
Test B	2,765.00μs	2,788.00μs	8,912.00μs	8,859.00μs	3.22	3.18
Test C	2,046.00μs	2,099.00μs	6,538.00μs	6,653.00μs	3.19	3.17

Table 6.26: The average measured run time for the *Matrix multiplication* application on the MD407 using doubles. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

MD407 Float	FreeRTOS Ref	FreeRTOS Real	RTICv1 Ref	RTICv1 Real	Ratio Ref	Ratio Real
Test A	144.00μs	206.00μs	248.00μs	407.00μs	1.72	1.97
Test B	175.00μs	196.00μs	304.00μs	484.00μs	1.74	2.47
Test C	142.00μs	151.00μs	272.00μs	382.00μs	1.91	2.53

Table 6.27: The average measured run time for the *Matrix multiplication* application on the MD407 using floats. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

Pico	**FreeRTOS Ref**	**FreeRTOS Real**	**RTICv1 Ref**	**RTICv1 Real**	**Ratio Ref**	**Ratio Real**
Test A	6,212.14μs	6,344.30μs	19,230.90μs	19,381.16μs	3.09	3.05
Test B	8,105.02μs	8,169.62μs	25,592.68μs	25,751.02μs	3.16	3.15
Test C	6,228.72μs	6,270.02μs	19,242.36μs	19,371.26μs	3.09	3.09

Table 6.28: The average measured run time for the *Matrix multiplication* application on the Pico with RTICv1. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

Pico	**FreeRTOS Ref**	**FreeRTOS Real**	**RTICv2 Ref**	**RTICv2 Real**	**Ratio Ref**	**Ratio Real**
Test A	6,212.14μs	6,344.30μs	19,311.12μs	19,618.60μs	3.11	3.09
Test B	8,105.02μs	8,169.62μs	20,983.34μs	25,821.68μs	2.59	3.16
Test C	6,228.72μs	6,270.02μs	19,061.82μs	15,101.60μs	3.06	2.41

Table 6.29: The average measured run time for the *Matrix multiplication* application on the Pico with RTICv1. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

Discovery	**FreeRTOS Ref**	**FreeRTOS Real**	**RTICv1 Ref**	**RTICv1 Real**	**Ratio Ref**	**Ratio Real**
Os	68,395.00μs	880,490.00μs	58,335.00μs	291,929.00μs	0.85	0.33
Oz	72,665.00μs	901,667.00μs	74,077.00μs	1,494,500.00μs	1.02	1.66
O3	25,713.00μs	747,957.00μs	72,288.00μs	148,769.00μs	2.81	0.2
O2	38,527.00μs	783,161.00μs	61,538.00μs	746,770.00μs	1.6	0.95
O1	51,350.00μs	1,078,860.00μs	68,968.00μs	1,578,138.00μs	1.34	1.46
O0	N/R	N/R	Broken	Broken	Broken	N/R

Table 6.30: The averaged measured time for the *Concurrency* application on the Discovery. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

MD407	**FreeRTOS Ref**	**FreeRTOS Real**	**RTICv1 Ref**	**RTICv1 Real**	**Ratio Ref**	**Ratio Real**
Os	7,218.00μs	133,895.00μs	11,311.00μs	114,299.00μs	1.57	0.85
Oz	7,218.00μs	133,895.00μs	12,502.00μs	185,738.00μs	1.73	1.39
O3	3,647.00μs	102,774.00μs	4,168.00μs	24,409.00μs	1.14	0.24
O2	4,834.00μs	110,602.00μs	11,312.00μs	127,990.00μs	2.34	1.16
O1	5,430.00μs	141,366.00μs	8,929.00μs	147,636.00μs	1.64	1.04
O0	8,421.00μs	246,791.00μs	21,443.00μs	468,118.00μs	2.55	1.9

Table 6.31: The averaged measured time for the *Concurrency* application on the MD407. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

Pico	FreeRTOS Ref	FreeRTOS Real	RTICv1 Ref	RTICv1 Real	Ratio Ref	Ratio Real
Os	17,000.54μs	235,670.27μs	12,271.62μs	117,122.10μs	0.72	0.5
Oz	17,000.40μs	235,668.14μs	12,621.70μs	209,622.68μs	0.74	0.89
O3	8,194.00μs	221,246.70μs	10,413.14μs	20,279.82μs	1.27	0.09
O2	11,411.32μs	247,092.62μs	12,827.02μs	136,339.44μs	1.12	0.55
O1	11,408.74μs	258,611.02μs	13,927.22μs	171,956.52μs	1.22	0.66
O0	20,352.10μs	384,135.24μs	33,724.96μs	1,443,140.88μs	1.66	3.76

Table 6.32: The averaged measured time for the *Concurrency* application on the Pico with RTICv1 compared to FreeRTOS. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

Pico	FreeRTOS Ref	FreeRTOS Real	RTICv2 Ref	RTICv2 Real	Ratio Ref	Ratio Real
Os	17,000.54μs	235,670.27μs	11,235.81μs	77,467.04μs	0.66	0.33
Oz	17,000.40μs	235,668.14μs	11,238.88μs	131,836.10μs	0.66	0.56
O3	8,194.00μs	221,246.70μs	12,036.06μs	53,801.28μs	1.47	0.24
O2	11,411.32μs	247,092.62μs	12,035.38μs	53,810.42μs	1.05	0.22
O1	11,408.74μs	258,611.02μs	12,034.92μs	52,205.48μs	1.05	0.2
O0	20,352.10μs	384,135.24μs	32,251.64μs	7,201,079.50μs	1.58	18.75

Table 6.33: The averaged measured time for the *Concurrency* application on the Pico with RTICv2 compared to FreeRTOS. The values are in microseconds, except for the ratios which are the RTIC performance divided by the FreeRTOS performance and thus have no unit.

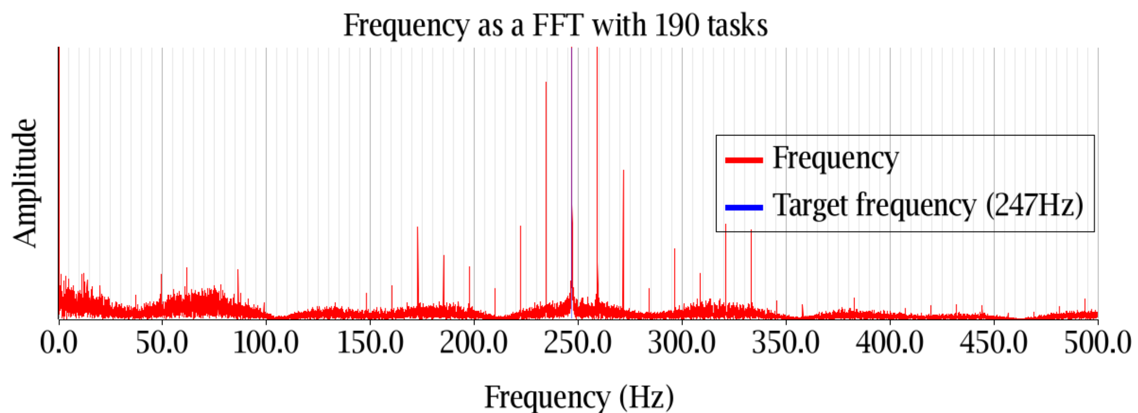


Figure 6.2: FFT of the output signal when using 190 background tasks in FreeRTOS.

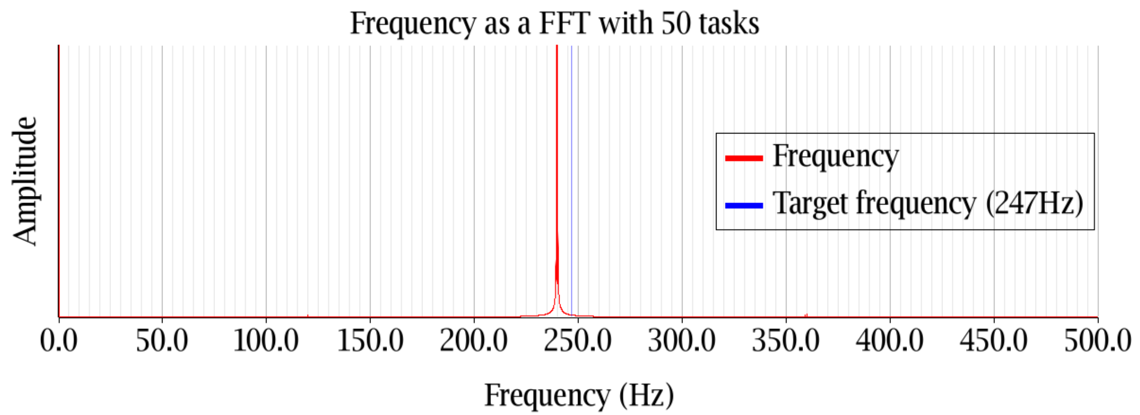


Figure 6.3: FFT of the output signal when using 50 background tasks in RTIC.

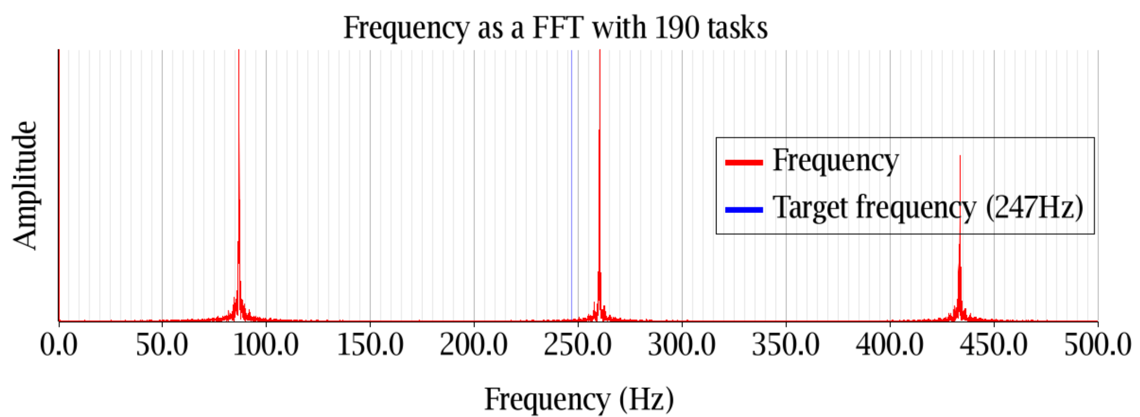


Figure 6.4: FFT of the output signal when using 190 background tasks in RTIC.

7

Discussion

In this chapter, the results are interpreted and analyzed. Furthermore, some anecdotes from the development is discussed along with the place of Rust in a real-time environment.

7.1 Interpretation of the results

In this section, the results from the previous chapter are discussed. The implications of the results are explored, as well as what might be affecting each result.

7.1.1 Binary sizes

The binary sizes are generally larger on the Rust side compared to C. This is no surprise, as Rust binaries have earlier been observed to be larger than C binaries as Rust prefers speed over binary size. This has been researched in the paper done by Ayers et al. [57]. They explore different ways to make binary sizes smaller on embedded Rust applications. In this thesis, however, all of these techniques have not been applied, but we still believe that the binary sizes are acceptable. Some of the methods proposed were unusual and relied on new programming techniques, and we wanted our tests to more accurately reflect regular programming principles. It has been possible to run all of our applications on the desired hardware, and the Rust binary sizes were never a problem. There were a few times in which we had the wrong compilation settings, causing enormous binaries. These binaries were still however capable of being flashed to the microcontrollers.

There are more problems regarding binary sizes when it comes to the C applications, which is surprising. The problems lies in the way in which FreeRTOS handles static tasks. Each static task needs to be assigned a buffer with a pre-defined size. This takes up space in the binary. It is also hard to get an accurate buffer size since it is not possible to determine the buffer space the static task needed beforehand. This also means that it is necessary to try and determine the buffer size through trial and error. This is a problem in the tone generator application. The reason for the max number of background tasks being 190 is that the data part of the C application does not fit in the data part of the memory of the microcontroller if there are more tasks. In RTIC however, there is only one instance of a task at all times. This means that different instances do not need separate buffers, meaning smaller data sizes.

When running our initial tests, the Rust binaries are compiled with run-time overflow checks. This caused the binaries to almost double in size in some cases. Since the default setting of the `release` compilation settings is to turn off the run-time checks, they are also disabled for our tests to better reflect real circumstances.

An anomaly in these results is in the binary sizes for the *Concurrency* application, as seen in Table 6.4. For some reason, the binary size is smaller when using `O3` and `O2` in Rust. In theory, `Os` and `Oz` should produce the smallest binaries. Why this is not the case here is hard to conclude. It is important to note that the compilation levels are best-effort, and when using for example compilation level `Os`, the compilation flags which *usually* reduce the binary size are added. But this may not always be the case depending on the code, which can be observed here.

7.1.2 Memory usage

Memory usage depends heavily on the type of application, and this is not a surprise either. In applications where there are multiple instances of the same tasks, one can see that the C applications use more memory. This is mainly because of how the RTOSs differ in how they handle tasks. As mentioned earlier, RTIC does not allow multiple instances of the same task to exist at any given time, while FreeRTOS allows an unbounded amount of instances of the same task (assuming there is memory available to give to these tasks). This means that, for example in the matrix multiplication application, there will be in the worst case 40 more tasks in FreeRTOS than in RTIC. Since the tasks' memory usage is being measured in FreeRTOS, FreeRTOS will show significantly higher memory usage compared to RTIC in these types of applications.

For the other applications, one can observe that the memory usage is fairly similar with FreeRTOS having an advantage the majority of the time. However, RTIC is still able to use a limited amount of resources, which is ideal in an embedded environment.

7.1.3 Compilation time

The Rust compilation times are longer than their C counterparts. This is not a surprise, as Rust does more checks during compile time to make sure that the Rust code is more safe and correct. This might however be a hindrance in development since one might want to recompile projects from scratch multiple times. As mentioned in Section 5.3.1, Rust has incremental compilation meaning that recompilation of the same Rust application is quicker. This is however very hard to measure since multiple factors can affect the speed, such as the number of code lines changed.

7.1.4 Application specific metrics

We can observe that when it comes to performance, C is performing slightly better. This is no surprise, as C undoubtedly is quicker than Rust in most situations. However, when comparing the reference tests with the real tests on both the concurrency

applications as well as on the matrix applications we can see something interesting. On the concurrency applications, we can see that most of the time RTIC is slower on the reference tests, but on the real tests, RTIC is faster. This can easily be seen in the right-most columns in the tables, as a value lower than 1 in the ratio indicates that RTIC is quicker than FreeRTOS.

On the matrix applications results, one can observe a similar behavior. However, the behavior is harder to observe compared to previously. The difference, in percentage, between the reference test and the real test on the RTIC runs is lower than the difference on FreeRTOS. This could mean that there is less overhead added by RTIC when running multiple tasks, as compared to FreeRTOS. The huge performance difference observed between RTIC and FreeRTOS on these tests could stem from C having better libraries for handling floating point operations since only the MD407 microcontroller has an FPU. However, the tests on the MD407 speak against this hypothesis. When running our applications using 32-bit floats, which enables the usage of the FPU, it seems that FreeRTOS has a lower overhead when context switching compared to RTIC. This is very interesting, and it is hard to say why this behavior is only observed on the MD407. One theory could be that RTIC just has slower context switches since the absolute time difference between the reference and real tests is larger on the RTIC side compared to the FreeRTOS tests on all hardware. Then, the huge performance difference is only because of the matrix calculation and nothing else.

7.2 Anecdotes from development

In this section, we present anecdotes about our experience from developing the applications. For example, we will discuss how much easier it was with package management in Rust as well as how surprisingly easy it was to interact with the hardware using Rust. Furthermore, we also discuss the immaturity of the RTIC real-time system as well as the immaturity of the HAL packages in Rust.

When the Rust applications were developed, there were fewer problems and the development was more smooth. On the C side, there were problems with how to organize the compilation so that every necessary part was compiled and also found by the compiler in the correct order. In Rust, this was organized by *Cargo* and it was quicker to start developing applications for Rust. Another aspect that was easier in Rust was finding bugs in the code. The compiler disallowed obvious type errors, while the C compiler was not always able to recognize these. This is partly due to programmer issues, such as not enabling all warnings and understanding them correctly, but it also shows that for inexperienced developers such as ourselves, this becomes a significant problem. We also have more previous experience with the Rust compiler, which aided us greatly in getting the Rust application working quickly.

It is also surprisingly easy to get RTIC running on different hardware, compared to FreeRTOS. RTIC is basically “plug-and-play”, which FreeRTOS also is in most cases except for on the MD407 microcontroller. This microcontroller has a relocated vector table, which means that FreeRTOS is not able to find the correct memory

address for the placement of necessary interrupt handlers. The reason for this is most likely not because of FreeRTOS itself, but because of the linker script not being set up correctly. On the RTIC side, however, this is not a problem. But if it is a problem for FreeRTOS, it should also be a problem for RTIC. Most likely the reason for the RTIC applications working correctly is that the template that is used for the linker script for RTIC was correct from the beginning, as compared to the FreeRTOS applications' linker scripts.

Another observation is that during the development of the applications is that the Rust libraries (especially the HALs) are immature compared to their C counterparts. This means that it is needed to implement some functionality from scratch, such as configuring the hardware timer correctly on the Discovery microcontroller. The abstractions that exist for the hardware timer did not fit our purpose, where we want the timer to count from 0 and up indefinitely.

Another anecdote from the development is that Rusts borrow checker often worked against us. The Pico microcontroller only has one hardware timer, which we want to use as both a measurement tool for our applications as well as for RTIC to schedule its tasks correctly. However, to achieve this two instances of the same timer object are needed. Rust forbid this, as it is not possible to clone the object. This could however be fixed by relying on `unsafe` operations. The timer memory addresses are saved in a variable and then are accessed directly using `unsafe`. This is however safe to do since we only ever read the value of the timer and never modified it, but it could potentially introduce undefined behavior.

7.3 The implications of a restrictive language

Rust offers good memory security because of its type system as well as borrowing/ownership rules, but it also introduces more overhead in terms of compilation time and run time. C on the other hand is very fast in terms of both compilation and run time, but also allows the programmer to make crucial mistakes. In this section, this dynamic will be explored further.

Rust, as described in Section 3.3, offers a lot of built-in memory security. This does not come for free, as can be seen in the results section. We can see an increase in binary size, compilation time, and general execution time when compared to C. The rules in Rust are also more complicated compared to C. In a survey by Fulton et al. [58], they asked multiple programmers about their experiences with the Rust language. Many interviewees felt confident that they produced less bug-prone code when programming in Rust, but many also thought that the ownership and borrowing rules were hard to understand.

Another aspect to consider is the library management of the two languages. There could be severe vulnerabilities in external libraries, or even simply malicious libraries whose aim is to cause damage. In C, to install a library is to download a package from any source and then place it into the project. In Rust, one must use the built-in library management tool called *Cargo* to use any external libraries. On the surface, one can assume that Rust must be safer since it uses a centralized library distribution

(*crates.io*). However, this distributor does not review any crates before publishing them and anyone can publish their crates. Crates.io does however remove crates that are found to be malicious, but it could be too late once the crate already is installed in an application. So in a way, this approach is safer than downloading a library from anywhere on the web. But precautions should still be taken to not introduce any malicious crates to an application and to not place blind trust in a crate.

The Rust language also emphasizes correctness and predictability, which can be seen in its features such as mutability. Rust requires that variables that are going to be modified at some point in execution need to be marked as mutable, to make the expected behavior explicitly clear. This is also a reason why Rust does not allow the "++" operator, as compared to C which allows this practice, since it could be ambiguous as explained in Section 3.2.3. This, we think, clearly shows the mindset of the Rust language. The programmer should be explicit in what they want to accomplish, otherwise the language does not allow it. This could also be a potential hindrance for programmers less experienced with Rust, as they could see it as the language working against them.

Another important aspect of the language is Unsafe Rust, which is needed because of the restrictive nature of Rust. Unsafe Rust could introduce critical memory securities since one is allowed to interact directly with the memory with Unsafe Rust. However, it is also important to note that languages such as C are always "unsafe", and Rust at the very least encapsulates the problem. If there is some sort of memory issue in a Rust program, it can be narrowed down to the Unsafe parts. In contrast to C, one would need to scrutinize the entire application. Unsafe code could however also be hidden in crates used, which would make it harder to spot and fix. But then again, C is always "unsafe" so in this regard Rust has the upper hand. This is also discussed in the paper introduced in Chapter 2 by Evans et al. [15]. They conclude that it is a problem with hidden Unsafe Rust in libraries and that programmers need to be careful.

7.4 Other RTOS alternatives in Rust

There exists other RTOSs written in Rust as well, that might be of interest. In this section, some of them will be presented and discussed.

7.4.1 Embassy

Embassy is built around the `async/await` functionality in Rust, to enable easy concurrency. Furthermore, Embassy offers its own HALs for multiple different microcontrollers which would make integration with most other hardware easier.

Embassy is similar to RTIC in multiple ways, such as using asynchronous tasks and a similar scheduling algorithm. There is a difference in how shared resources are used, however. In Embassy, shared resources are declared globally as static variables. To make this thread-safe, the variable could be wrapped in a `Mutex` type which requires locking to access.

There have been some benchmarks done in comparing FreeRTOS and Embassy, which do show promise [59]. However, the benchmarks are limited and would require more testing to be more assertive. But Embassy does show indications of being a viable RTOS.

7.4.2 Tock

Tock is another RTOS written in the Rust language [60]. Tock emphasizes security, by utilizing memory protection units on the controller to isolate applications from one another and the kernel itself. The point is to not allow potential malicious applications from interacting with other applications or the kernel. The Tock RTOS also allows the user to define a scheduling algorithm, making the RTOS more ergonomic. The concept of priorities does not exist in Tock.

Tock is an interesting choice if one heavily emphasizes security in their embedded application. The performance of the RTOS would however depend on one's implementation of the scheduler, although there exists pre-made schedulers to use. While the concept of priorities does not exist natively in Tock, priorities could be implemented with a custom scheduler.

7.4.3 Wrappers for C RTOSs

There exist multiple wrappers for other RTOSs written in C, which means that it is possible to run for example FreeRTOS using Rust. This is done by creating bindings between Rust and C, where Rust code can call C code. The advantage of doing this is that you get a fully-fledged RTOS without having to rewrite the same RTOS in Rust. Another advantage is that memory-critical tasks can be made more secure in Rust.

The drawback of doing this is however that most of the system still relies on C code, since the RTOS itself is executing C code for its core functionalities. If a project requires strict security standards, this approach might not be better than using an RTOS written fully in Rust. But, again, at least some parts of the system can be written in secure Rust.

7.5 The future of Rust in real-time systems

We think that Rust has a future in the role of real-time systems. In a world where security is becoming increasingly important in all sorts of devices, Rust could be a good candidate for achieving memory security while still obtaining high performance. The results shown in the previous chapter show indications that Rust is useful in embedded systems. There exist multiple projects, some described in the earlier section, that are making progress in developing Rust in real-time systems.

The current problems with Rust in real-time systems as it stands now are, as previously mentioned, the immaturity of the libraries available and the difficulties of the language itself. These are real problems that need to be addressed within the

Rust community. However, progress is made each day in improving the libraries and the language itself to make it more ergonomic and easier to use, which might in the future result in wider usage within the real-time system world.

We have spoken to some companies within the real-time industry while writing this thesis. These companies show an interest in the language but have yet not made any steps towards implementing the language into their real-time core. This is largely because most systems are already written in C, and it is easier to rely on pre-existing tools rather than creating entirely new ones. The customers are also reluctant to switch over to a new language, since the systems they use already work 'good enough' and thus see no reason in investing in the Rust language. But interest in the language does however exist, and only time will tell whether the Rust language will gain any major traction.

7.6 Ethical and sustainability consideration

The cases for real-time systems are wide and they can be part of crucial systems from household appliances to a car or an airplane. This means the need to be able to make secure, safe, and reliable real-time systems is important and this is where the memory safety in Rust and the philosophy behind it can help. As talked about earlier, a large number of security issues can be traced back to memory and to make it easier to prevent those issues using a language like Rust that still suits the real-time domain since it does not rely on a garbage collector could be a good step in the right direction.

One important aspect of this potential shift that this thesis has not tested is power consumption. The electrical system in, for example, a car can be vast with a large number of sensors that continuously transmit data to the computers they are connected to. To handle all that data, to ensure the correct and predicted working of the car, can require a considerable amount of energy which means higher emissions. The real-time systems running on the computers can affect the energy needed by being efficient and reducing the power needed to run them which helps reduce the emissions of the vehicle. The programming language used to create the real-time system might have a significant effect on the power needed to run the real-time system which might mean that a language like Rust with more checks to ensure memory safety will lead to higher power consumption and in turn more emissions. This means that the construction of electrical systems, such as those in cars and airplanes, that want to minimize the power required to run them might need to balance the positive sides of secure languages with potentially higher power consumption. As power consumption has not been tested in this thesis this aspect should be considered alongside the aspect brought up in this thesis

Another potential ethical issue is the usage of RTOSs in military systems such as missiles. This thesis could inadvertently contribute to making such systems more reliable, in contributing to the usage of Rust in RTOSs. Missiles might become more accurate, and thus become better at causing damage to civilians. This however is out of our power, as almost all research done could be used for military purposes in one

way or another. It is not a good enough reason to stop research in real-time systems because it could also be used in missiles. Real-time systems are an integral part of society and contribute positively to society in multiple ways, which we would argue is more important than the negative sides of the military usage of real-time systems.

7.7 Future work

In this thesis, we focused mostly on RTIC version 1, but it would be interesting to see more extensive tests regarding RTIC version 2 since it shows some promising results.

Another area of future work could be exploring some of the other RTOSs that are written in Rust, such as Embassy and Tock which both look promising. It could be worth comparing Rust to other RTOSs written in C, such as AUTOSAR.

The area of power consumption could also be interesting to explore. As mentioned in Section 7.6, usage of real-time systems in cars and airplanes contributes to the emissions released by those vehicles. This means that if the indirect power consumption of the running code can be reduced so can the emissions.

8 Conclusion

Rust could be a tool in reducing security issues stemming from incorrect memory management, and could also, in turn, lead to more reliable real-time systems. This thesis has shown that Rust could outperform another high-performing language, C, in some aspects, while Rust still lacks in some aspects such as floating point operations.

However, Rust is not to be taken as a silver bullet. There is still room in the language for an unsuspecting programmer to make crucial mistakes with the usage of Unsafe Rust. It is also still possible to introduce logical bugs that are not caught by the memory safety mechanisms of Rust.

In conclusion, it can be stated that Rust is a viable language when it comes to real-time programming. Rust offers security, performance, and reliability which are all important aspects of a real-time system. Some libraries available for Rust are immature and need further work, but we find Rust to be a viable real-time language and that it will continue to improve in the future.

Bibliography

[1] H. Tjäder, "Rtic - a zero-cost abstraction for memory safe concurrency," M.S. thesis, Luleå tekniska universitet, Institutionen för system- och rymdteknik, 2021.

[2] J. Wang, *Real-Time Embedded Systems*. Newark : John Wiley & Sons, Incorporated, 2017., 2017.

[3] K. Erciyes, *Distributed Real-Time Systems*. Springer eBooks, 2019.

[4] "The state of developer ecosystem 2023." Accessed: 2024-03-20. (), [Online]. Available: `https://www.jetbrains.com/lp/devecosystem-2023/`.

[5] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," *2nd International Graduate Studies Congress*, vol. 32, no. 5, pp. 713–744, 2022. arXiv: `2206.05503 [cs.PL]`.

[6] T. Hilburn and J. Zalewski, "Real-time safety-critical systems: An overview," *IFAC Proceedings Volumes*, vol. 28, no. 25, pp. 127–138, 1995, 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, USA, 1-3 November, ISSN: 1474-6670. DOI: `https://doi.org/10.1016/S1474-6670(17)44835-X`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S147466701744835X`.

[7] "We need a safer systems programming language." Accessed: 2024-04-17. (), [Online]. Available: `https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/`.

[8] *Embedded survey 2023*, `https://www.embedded.com/embedded-survey-2023-more-ip-reuse-as-workloads-surge/`, Accessed: 2024-01-23.

[9] P. C. van Oorschot, "Memory errors and memory safety: C as a case study," *IEEE Security and Privacy*, vol. 21, no. 2, pp. 70–76, 2023. DOI: `10.1109/MSEC.2023.3236542`.

[10] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," *SIGPLAN Not.*, vol. 40, no. 10, pp. 313–326, 2005, ISSN: 0362-1340. DOI: `10.1145/1103845.1094836`. [Online]. Available: `https://doi.org/10.1145/1103845.1094836`.

[11] "An introduction to real-time java technology: Part 1, the real-time specification for java (jsr 1)." Accessed: 2024-07-11. (), [Online]. Available: `https://www.oracle.com/technical-resources/articles/javase/jsr-1.html`.

[12] Q. Gao, Y. Xiong, Y. Mi, *et al.*, "Safe memory-leak fixing for c programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 459–470. DOI: `10.1109/ICSE.2015.64`.

[13] D. Kopec and S. Tamang, "Failures in complex systems: Case studies, causes, and possible remedies," *SIGCSE Bull.*, vol. 39, no. 2, pp. 180–184, Apr. 2007, ISSN: 0097-8418. DOI: 10.1145/1272848.1272905. [Online]. Available: https://doi.org/10.1145/1272848.1272905.

[14] F. Jahanian and A. K.-L. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 890–904, 1986. DOI: 10.1109/TSE.1986.6313045.

[15] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 246–257, ISBN: 9781450371216. DOI: 10.1145/3377811.3380413. [Online]. Available: https://doi.org/10.1145/3377811.3380413.

[16] A. Pinho, L. Couto, and J. Oliveira, "Towards rust for critical systems," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036.

[17] D. Hedgren, "Designing secure applications for the internet of vehicles," M.S. thesis, Chalmers University of Technology, 2023.

[18] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu, *Memory-safety challenge considered solved? an in-depth study with all rust cves*, 2021. arXiv: 2003.03296 [cs.PL].

[19] *Project report: Language security of rust*, https://github.com/emrikll/master-thesis/blob/main/LBS___Project.pdf, Fetched: 2024-06-19.

[20] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," *IFAC Proceedings Volumes*, vol. 24, no. 2, pp. 127–132, 1991, IFAC/IFIP Workshop on Real Time Programming, Atlanta, GA, USA, 15-17 May 1991, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)51283-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017512835.

[21] D. M. Ritchie, "The development of the c language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.

[22] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000. DOI: 10.1109/2.876288.

[23] A. One. "Smashing the stack for fun and profit." Fetched 2024-04-04. (), [Online]. Available: https://insecure.org/stf/smashstack.html.

[24] "Doubly freeing memory." Accessed: 2024-03-01. (), [Online]. Available: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory.

[25] "Buffer overflow attack." Accessed: 2024-03-04. (), [Online]. Available: https://owasp.org/www-community/attacks/Buffer_overflow_attack.

[26] *Popularity of languages according to google trends*, https://pypl.github.io/PYPL.html, Accessed: 2024-01-19.

[27] T. Uzlu and E. Şaykol, "On utilizing rust programming language for internet of things," in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, IEEE, 2017, pp. 93–96.

[28] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "Gpu programming in rust: Implementing high-level abstractions in a systems-level

language," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, 2013, pp. 315–324.

[29] W. Bugden and A. Alahmar, "The safety and performance of prominent programming languages," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 05, pp. 713–744, 2022.

[30] S. Klabnik and C. Nichols. "What is ownership?" Accessed: 2024-03-01. (), [Online]. Available: https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html.

[31] *The rust book pictures*, https://github.com/rust-lang/book/tree/main/src/img, Fetched: 2024-05-13.

[32] S. Klabnik and C. Nichols. "Unsafe rust." Accessed: 2024-03-01. (), [Online]. Available: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[33] *Freertos*, https://www.freertos.org/, Fetched: 2024-05-13.

[34] *Freertos supported devices*, https://www.freertos.org/RTOS_ports.html, Accessed: 2024-01-19.

[35] *Freertos documentation*, https://www.freertos.org/features.html, Accessed: 2024-01-19.

[36] "Freertos software timers." Accessed: 2024-03-04. (), [Online]. Available: https://www.freertos.org/RTOS-software-timer.html.

[37] *Rtic documentation*, https://rtic.rs/2/book/en/preface.html, Accessed: 2024-01-19.

[38] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming api and static priority srp kernel primitives," in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013, pp. 110–113. DOI: 10.1109/SIES.2013.6601482.

[39] J. A. Rivera, "Real time rust on multi-core microcontrollers," M.S. thesis, Luleå University of Technology, 2020.

[40] *Cortex rtic crate*, https://crates.io/crates/cortex-m-rtic, Accessed: 2024-01-19.

[41] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991, ISSN: 1573-1383. DOI: 10.1007/BF00365393. [Online]. Available: https://doi.org/10.1007/BF00365393.

[42] J. G. P. Barnes, "An overview of ada," *Software: Practice and Experience*, vol. 10, no. 11, pp. 851–887, 1980. DOI: https://doi.org/10.1002/spe.4380101102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380101102. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380101102.

[43] Q. Lu, "Real-time operating system in java," MA thesis, University of Wollongong, 2007.

[44] P. J. McKerrow, "Developing real-time systems in java on macintosh," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:55906757.

[45] P. J. McKerrow, "Software development of embedded systems on macintosh," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:55515374.

[46] *Embassy-executor crate*, https://crates.io/crates/embassy-executor, Accessed: 2024-04-08.

[47] *Drone core crate*, https://crates.io/crates/drone-core, Accessed: 2024-04-08.
[48] *Stm32f0discovery*, https://www.st.com/en/evaluation-tools/stm32f0discovery.html, Accessed: 2024-01-19.
[49] F. Östman, "Md407 – en arm-baserad laborationsdator för utbildning," M.S. thesis, Chalmers University of Technology, Department of Computer Science and Engineering, 2016.
[50] *Stm32f407vg*, https://www.st.com/en/microcontrollers-microprocessors/stm32f407-417.html, Accessed: 2024-01-19.
[51] *Raspberry pi pico*, https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html, Accessed: 2024-01-19.
[52] *Freertos-kernel rp2040*, https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/main/portable/ThirdParty/GCC/RP2040, Accessed: 2024-04-08, 2024.
[53] *Project report: Language security of rust*, https://github.com/emrikll/master-thesis, Fetched: 2024-06-19.
[54] "Git." Accessed: 2024-07-31. (), [Online]. Available: https://git-scm.com/.
[55] *Getting started with raspberry pi pico*, https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf, Fetched: 2024-06-26.
[56] "Stm32 standard peripheral libraries." Accessed: 2024-07-13. (), [Online]. Available: https://www.st.com/en/embedded-software/stm32-standard-peripheral-libraries.html.
[57] H. Ayers, E. Laufer, P. Mure, *et al.*, "Tighten rust's belt: Shrinking embedded rust binaries," ser. LCTES 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, ISBN: 9781450392662. DOI: 10.1145/3519941.3535075. [Online]. Available: https://doi.org/10.1145/3519941.3535075.
[58] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Symposium on Usable Privacy and Security*, 2021.
[59] *Async rust vs rtos showdown!* https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown, Fetched: 2024-07-02.
[60] *The tock book*, https://book.tockos.org/doc/overview, Fetched: 2024-07-02.

A

Appendix FFT FreeRTOS

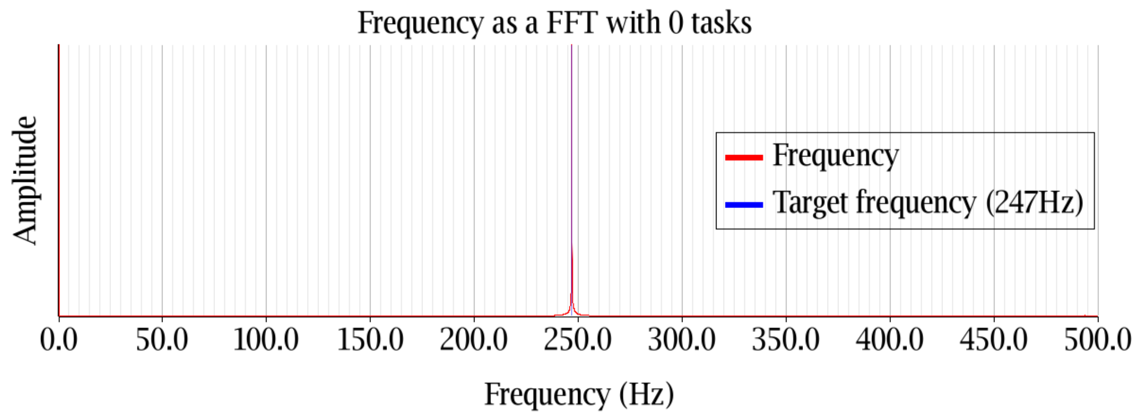


Figure A.1: FFT of the output signal when using no background tasks in FreeRTOS.

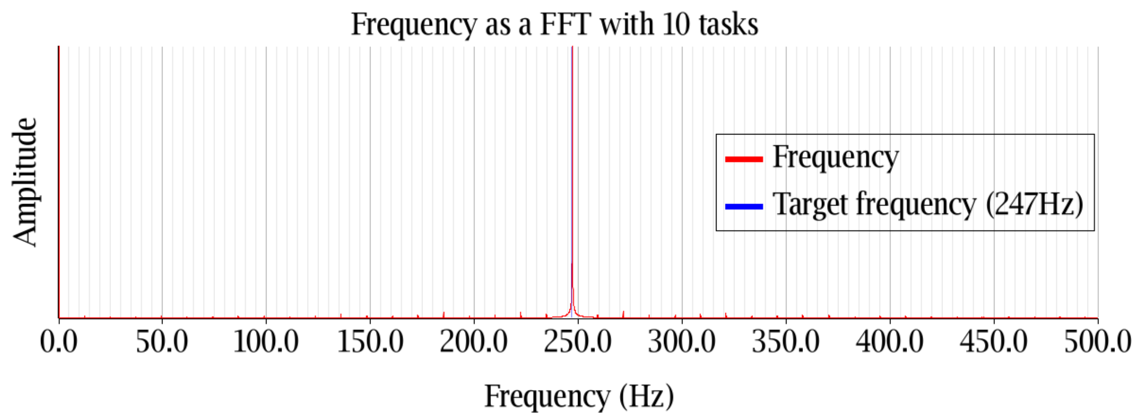


Figure A.2: FFT of the output signal when using 10 background tasks in FreeRTOS.

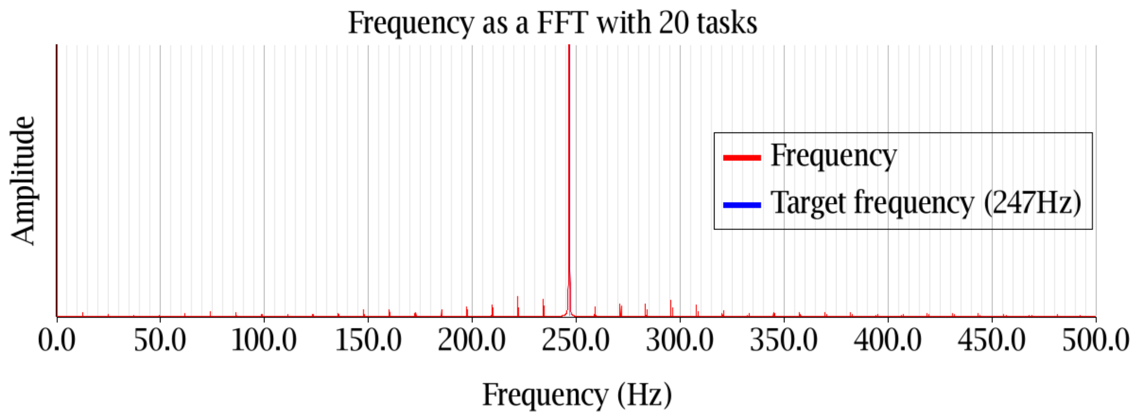


Figure A.3: FFT of the output signal when using 20 background tasks in FreeRTOS.

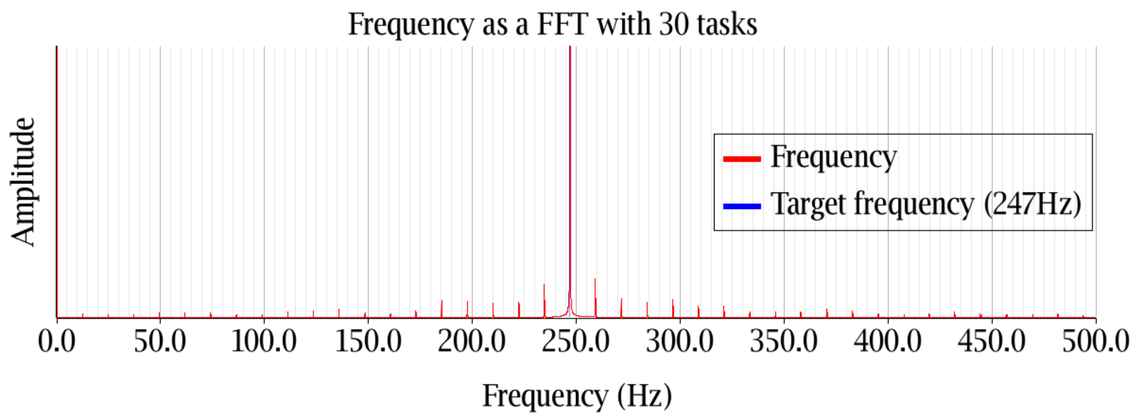


Figure A.4: FFT of the output signal when using 30 background tasks in FreeRTOS.

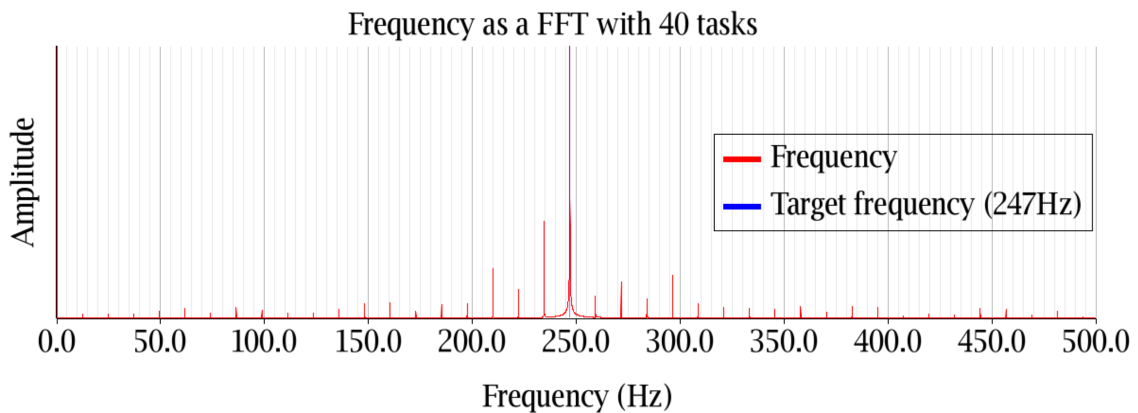


Figure A.5: FFT of the output signal when using 40 background tasks in FreeRTOS.

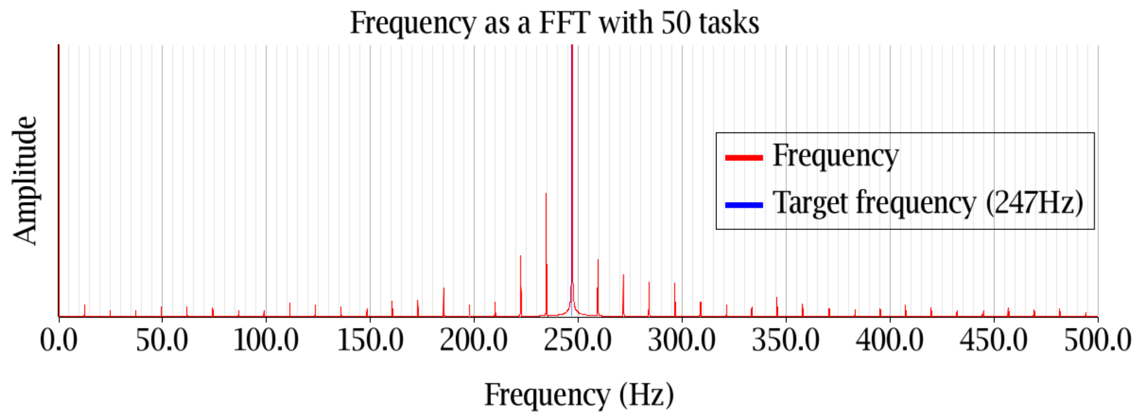


Figure A.6: FFT of the output signal when using 50 background tasks in FreeRTOS.

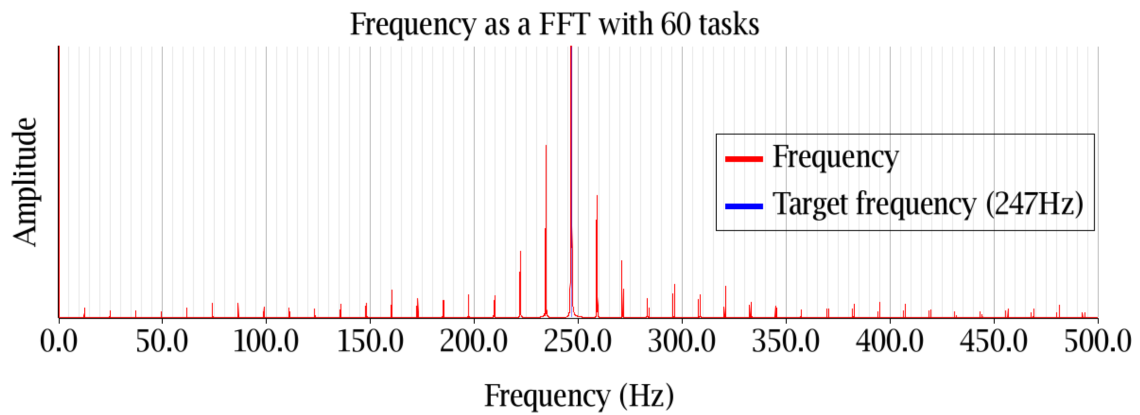


Figure A.7: FFT of the output signal when using 60 background tasks in FreeRTOS.

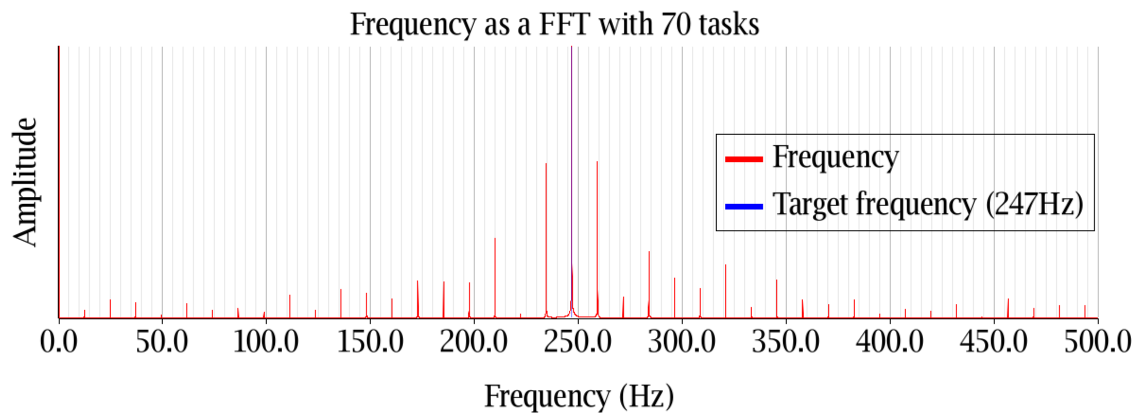


Figure A.8: FFT of the output signal when using 70 background tasks in FreeRTOS.

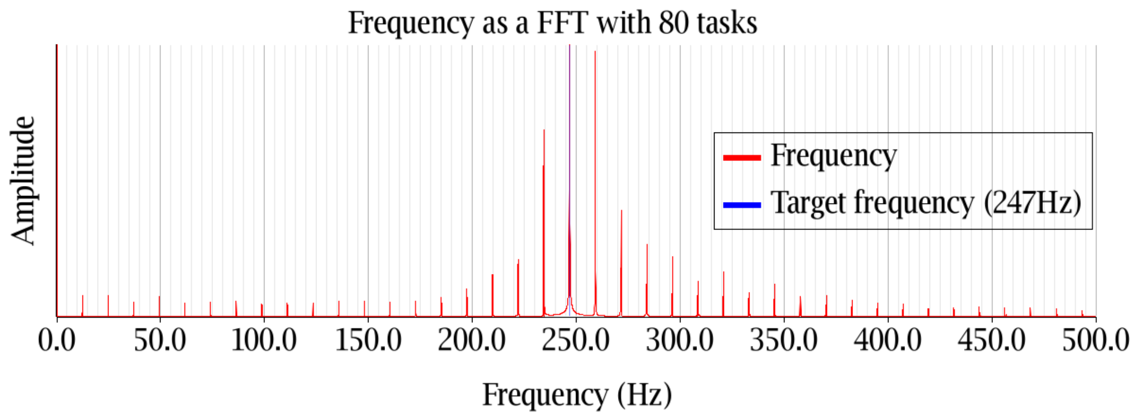


Figure A.9: FFT of the output signal when using 80 background tasks in FreeRTOS.

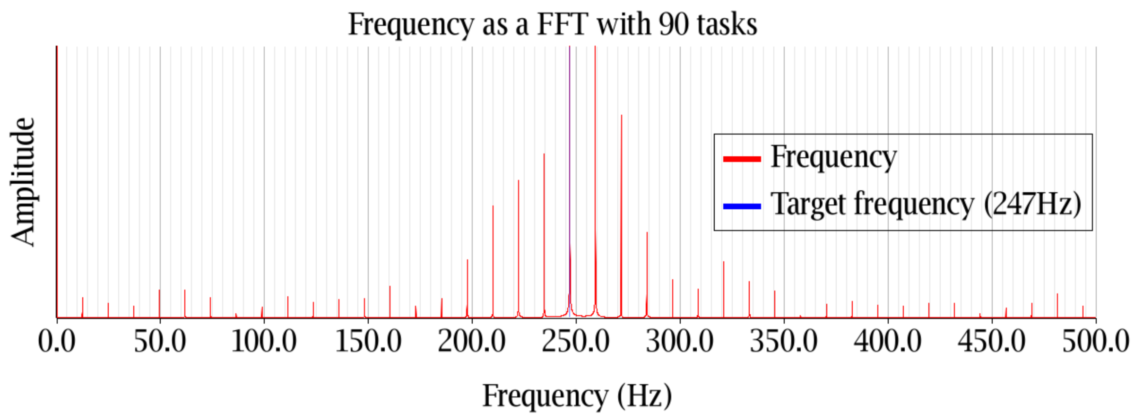


Figure A.10: FFT of the output signal when using 90 background tasks in FreeRTOS.

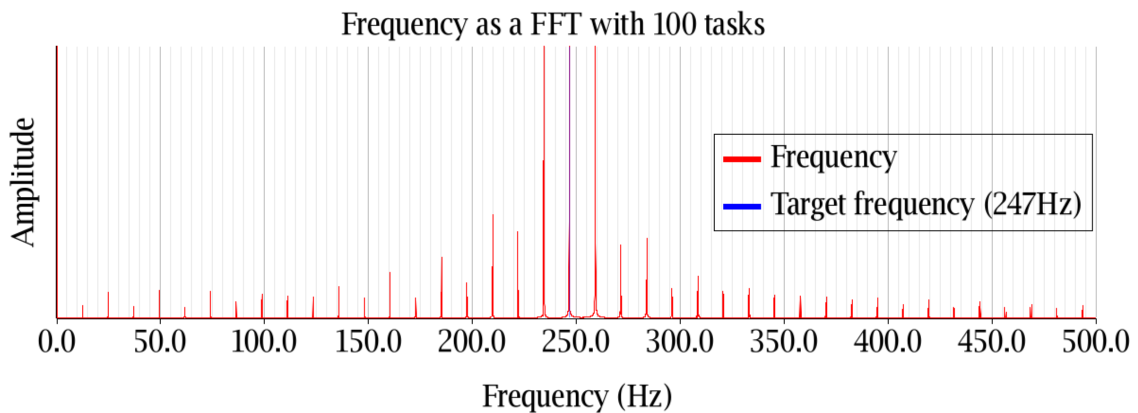


Figure A.11: FFT of the output signal when using 100 background tasks in FreeRTOS.

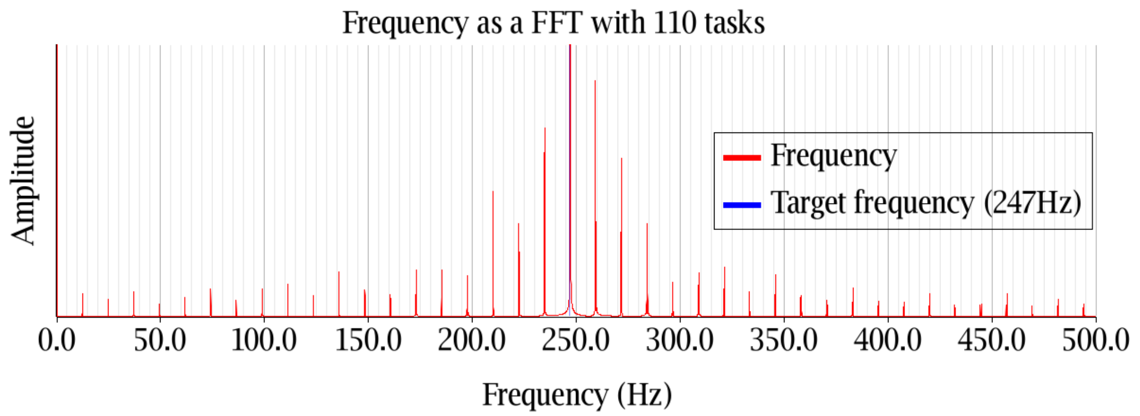


Figure A.12: FFT of the output signal when using 110 background tasks in FreeRTOS.

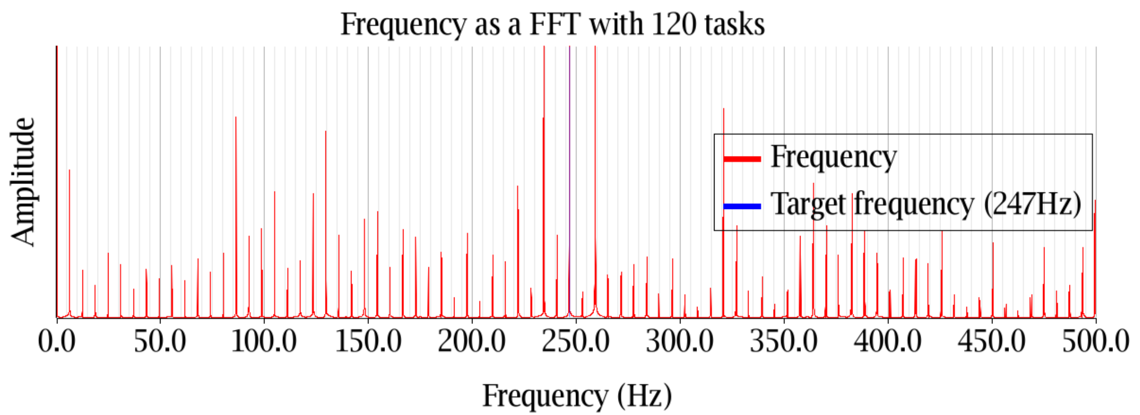


Figure A.13: FFT of the output signal when using 120 background tasks in FreeRTOS.

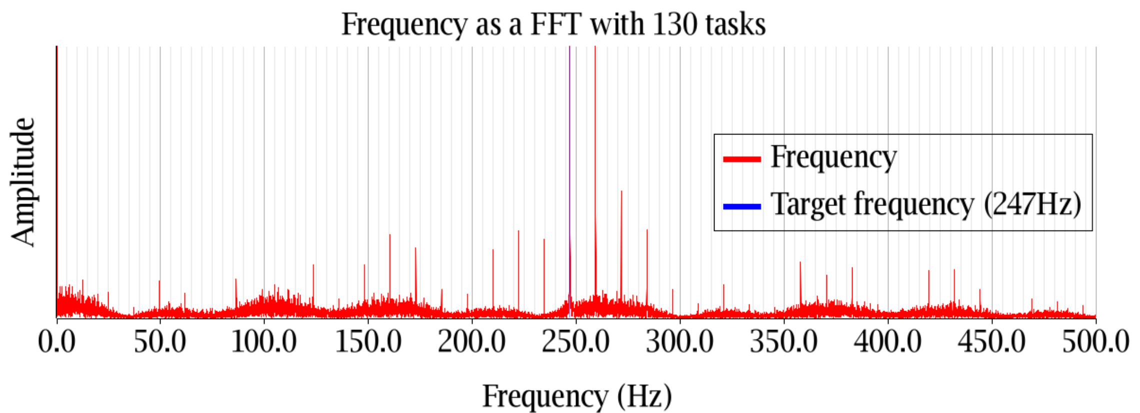


Figure A.14: FFT of the output signal when using 130 background tasks in FreeRTOS.

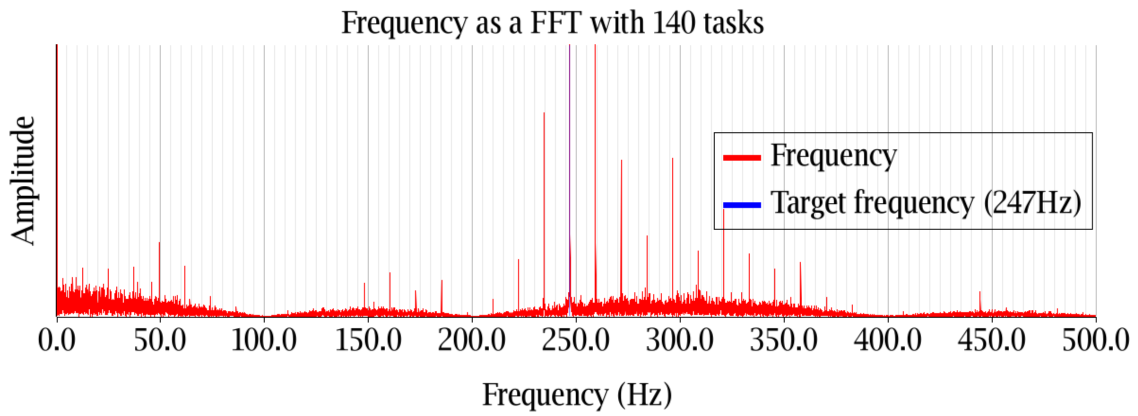


Figure A.15: FFT of the output signal when using 140 background tasks in FreeRTOS.

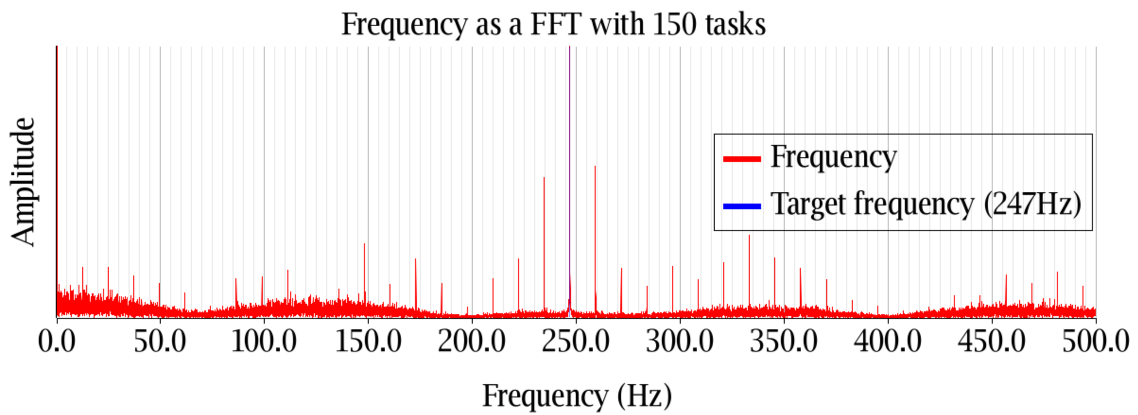


Figure A.16: FFT of the output signal when using 150 background tasks in FreeRTOS.

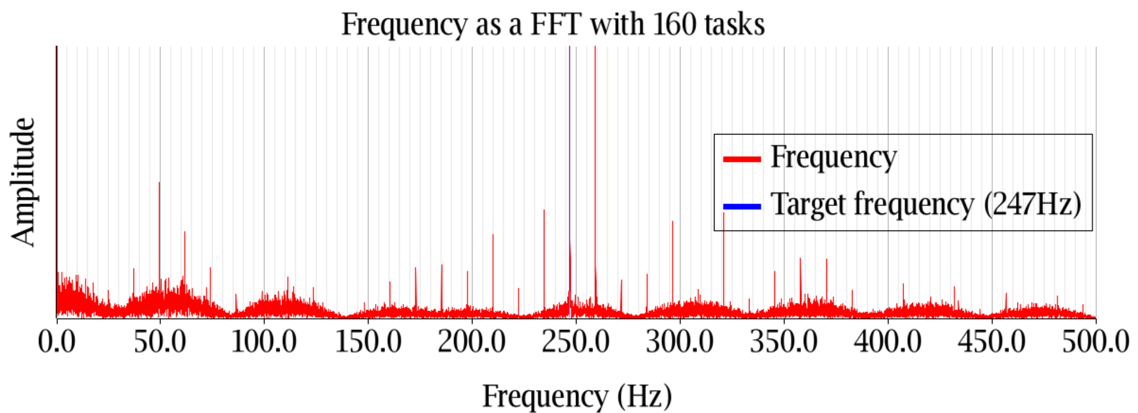


Figure A.17: FFT of the output signal when using 160 background tasks in FreeRTOS.

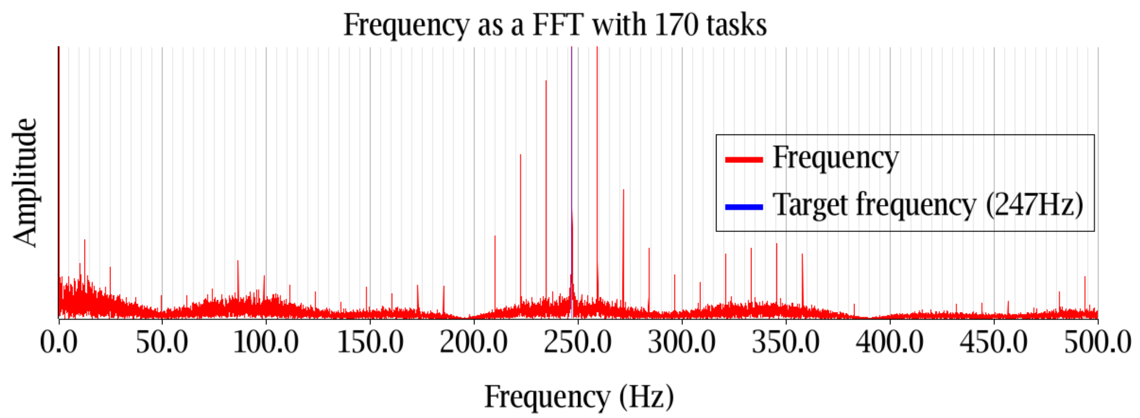


Figure A.18: FFT of the output signal when using 170 background tasks in FreeRTOS.

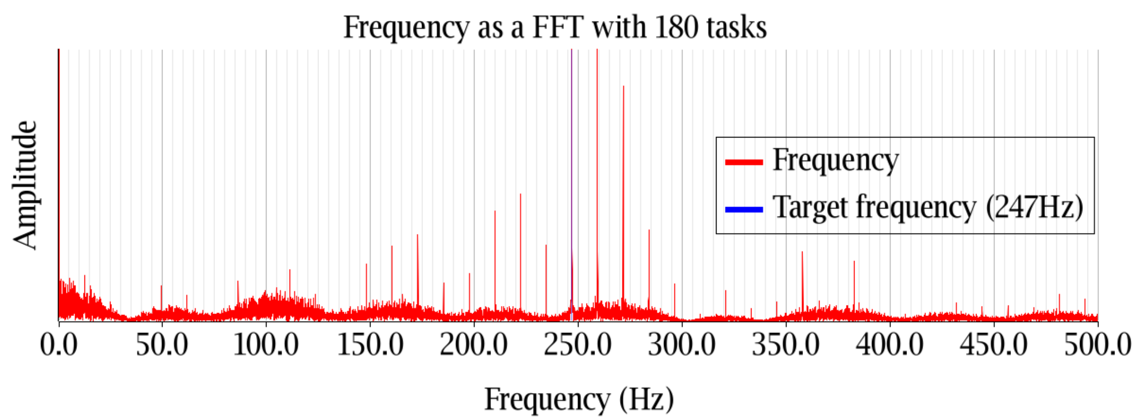


Figure A.19: FFT of the output signal when using 180 background tasks in FreeRTOS.

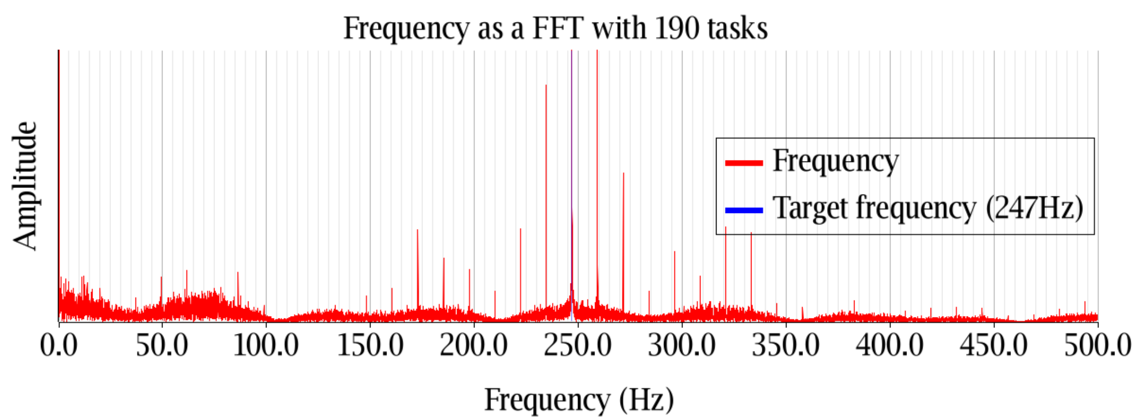


Figure A.20: FFT of the output signal when using 190 background tasks in FreeRTOS.

B

Appendix FFT RTIC

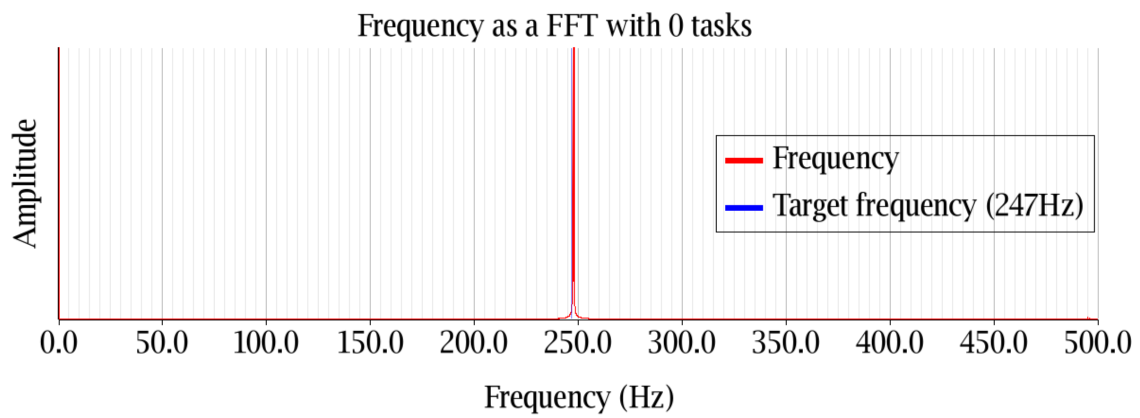


Figure B.1: FFT of the output signal when using no background tasks in RTIC.

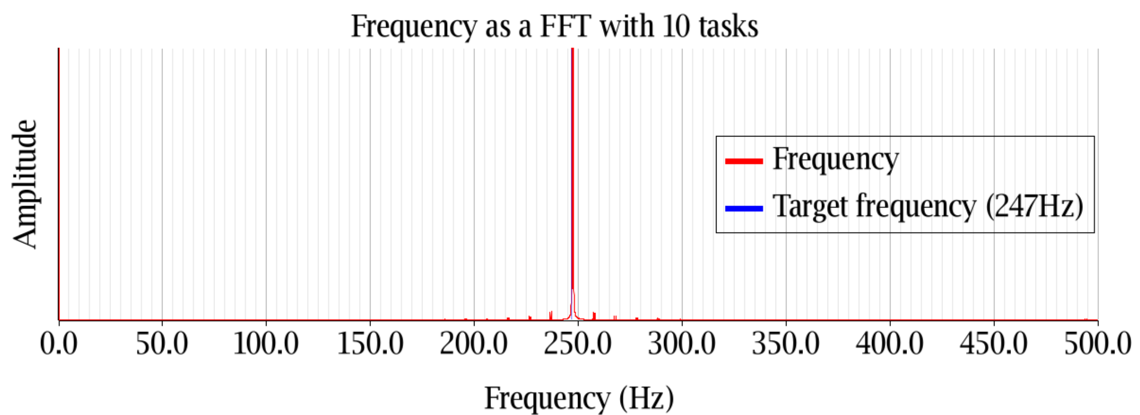


Figure B.2: FFT of the output signal when using 10 background tasks in RTIC.

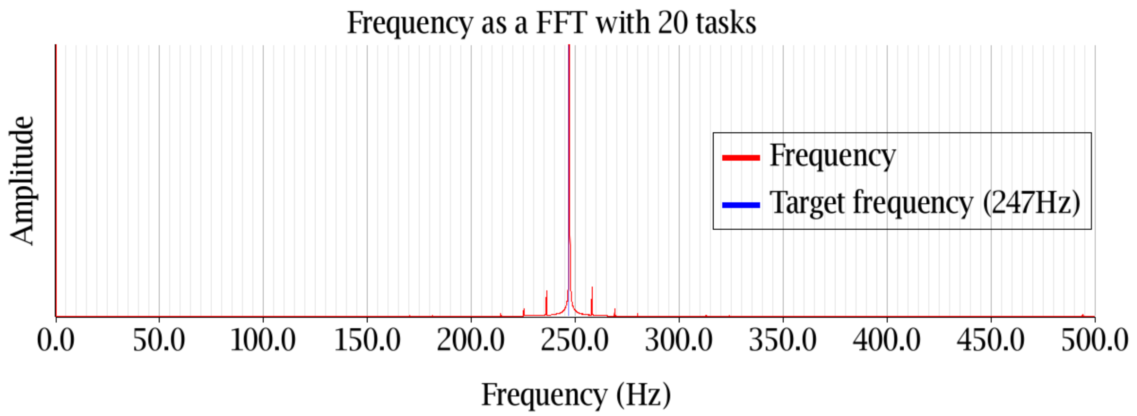


Figure B.3: FFT of the output signal when using 20 background tasks in RTIC.

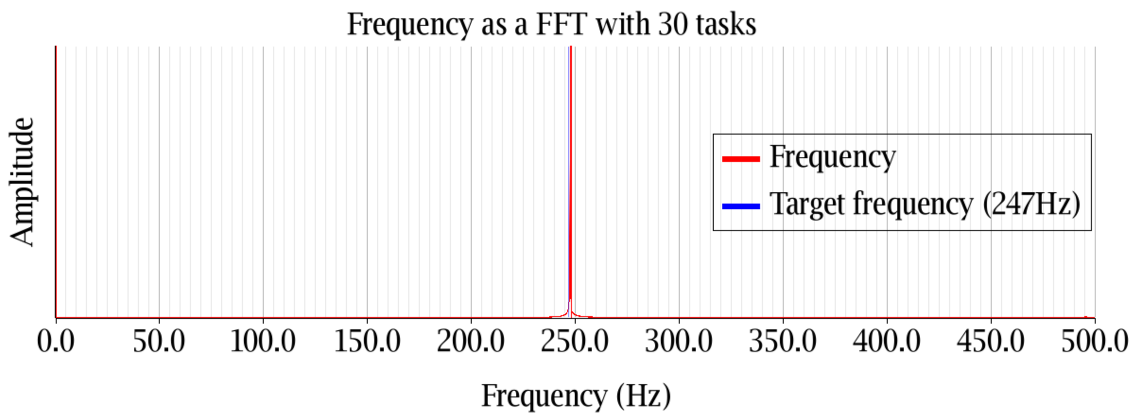


Figure B.4: FFT of the output signal when using 30 background tasks in RTIC.

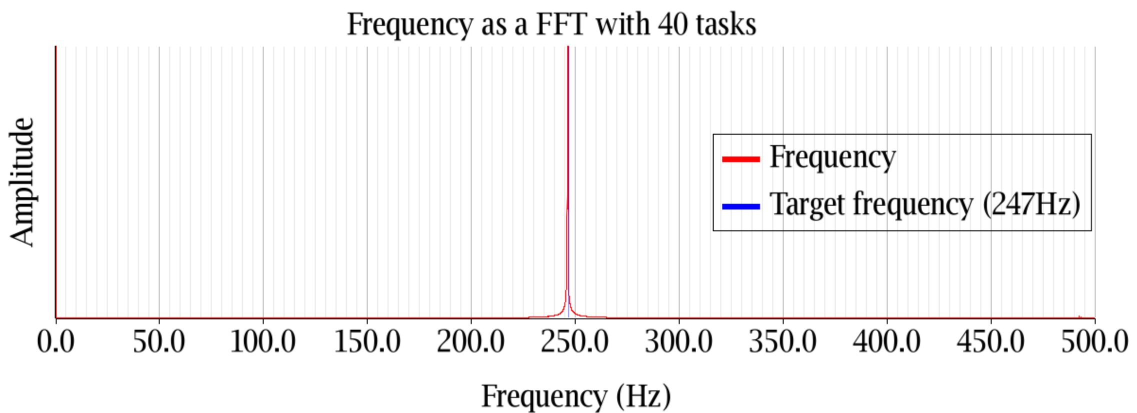


Figure B.5: FFT of the output signal when using 40 background tasks in RTIC.

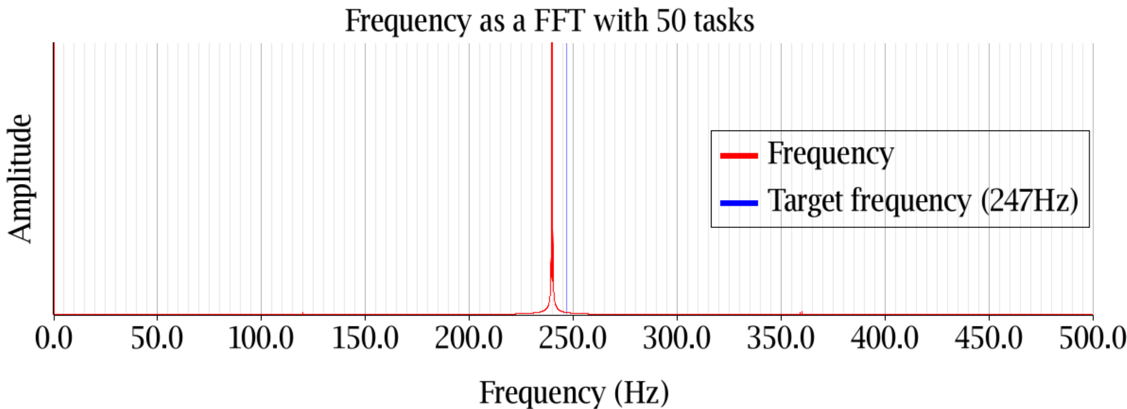


Figure B.6: FFT of the output signal when using 50 background tasks in RTIC.

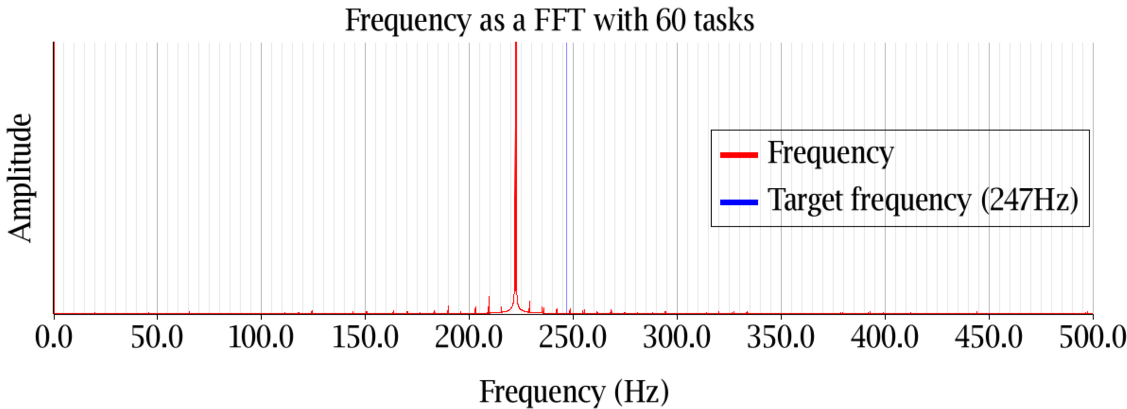


Figure B.7: FFT of the output signal when using 60 background tasks in RTIC.

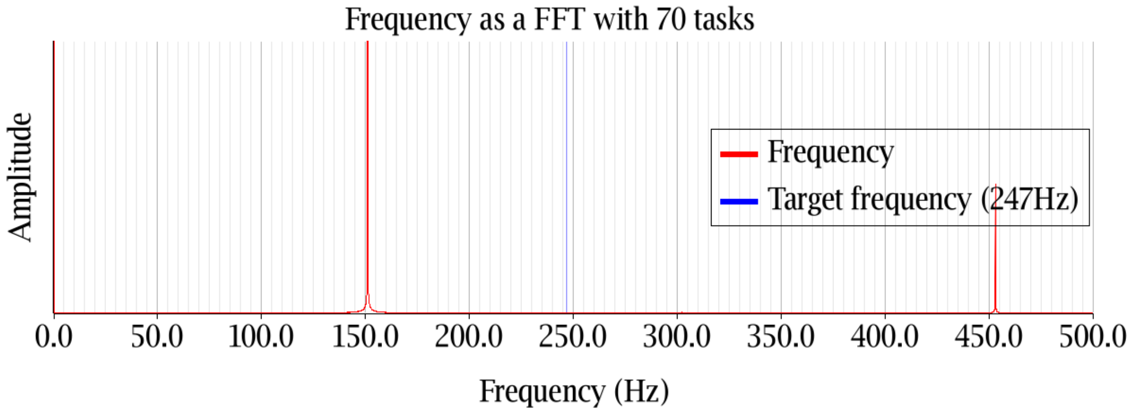


Figure B.8: FFT of the output signal when using 70 background tasks in RTIC.

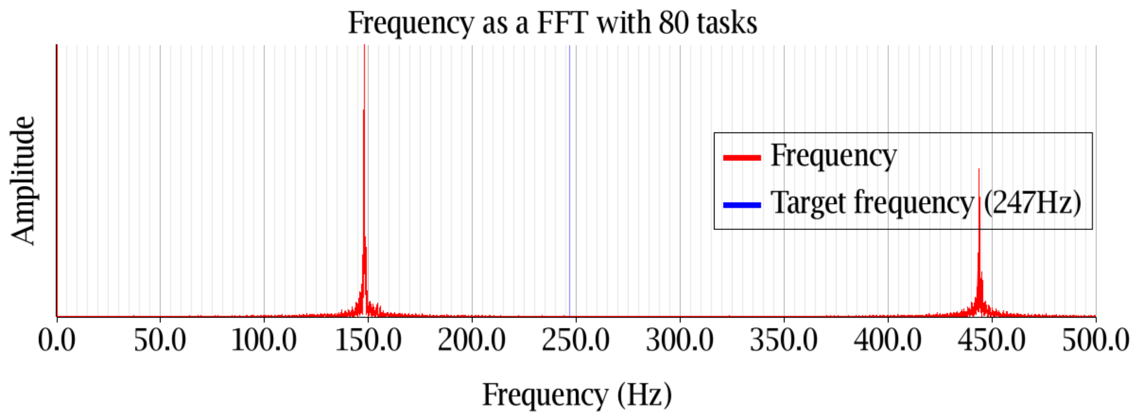


Figure B.9: FFT of the output signal when using 80 background tasks in RTIC.

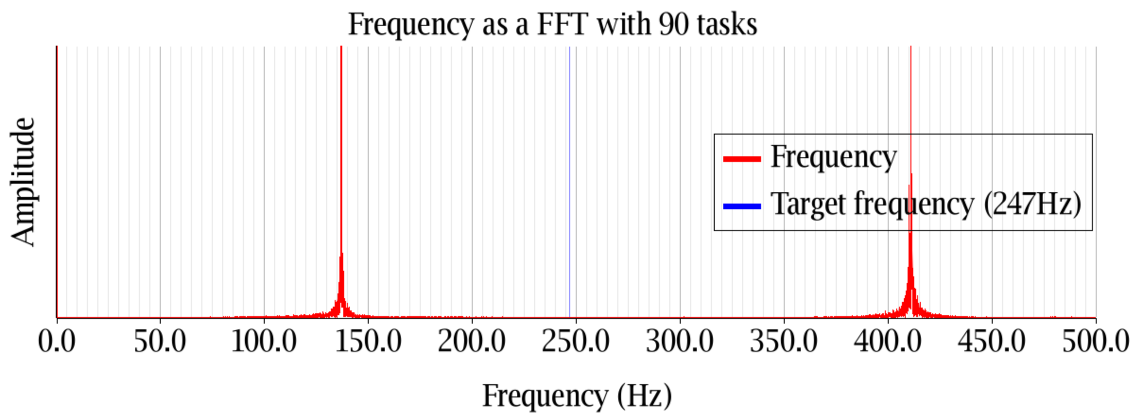


Figure B.10: FFT of the output signal when using 90 background tasks in RTIC.

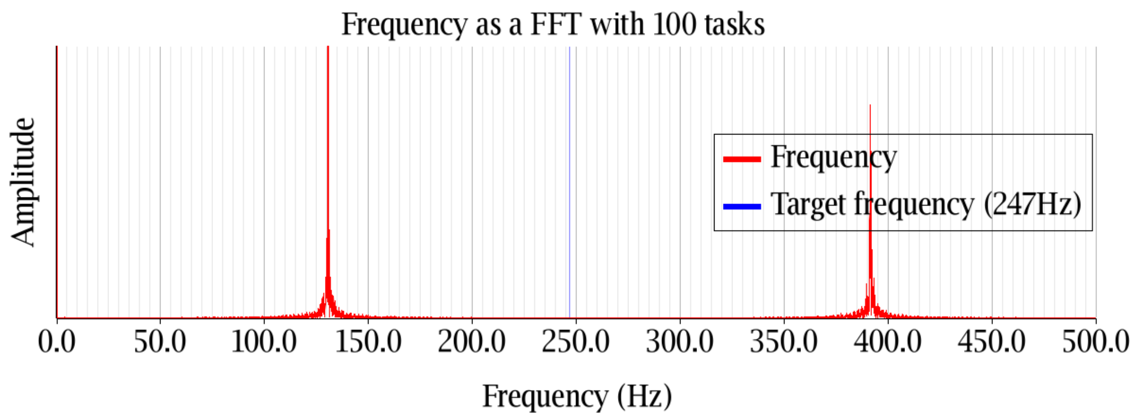


Figure B.11: FFT of the output signal when using 100 background tasks in RTIC.

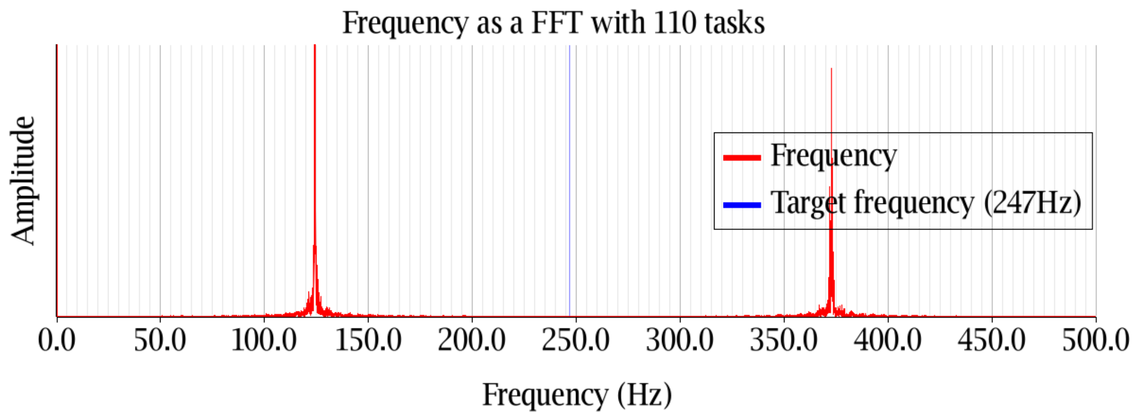


Figure B.12: FFT of the output signal when using 110 background tasks in RTIC.

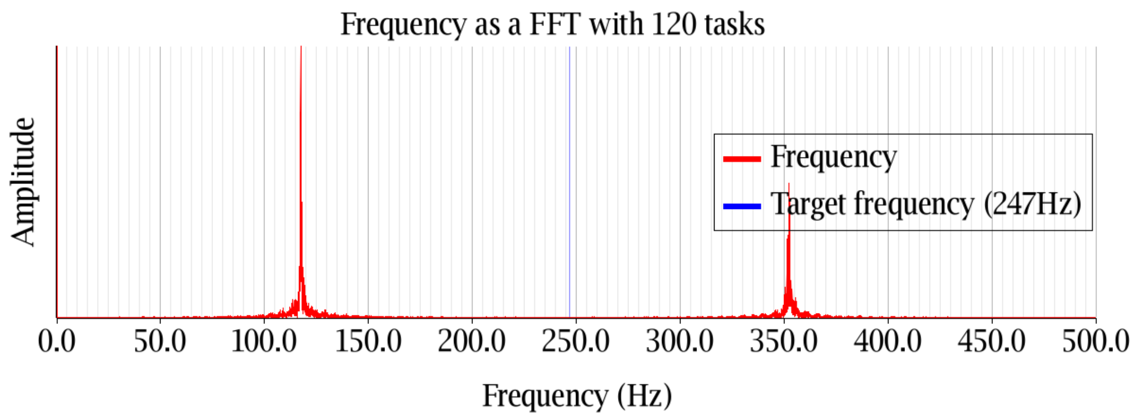


Figure B.13: FFT of the output signal when using 120 background tasks in RTIC.

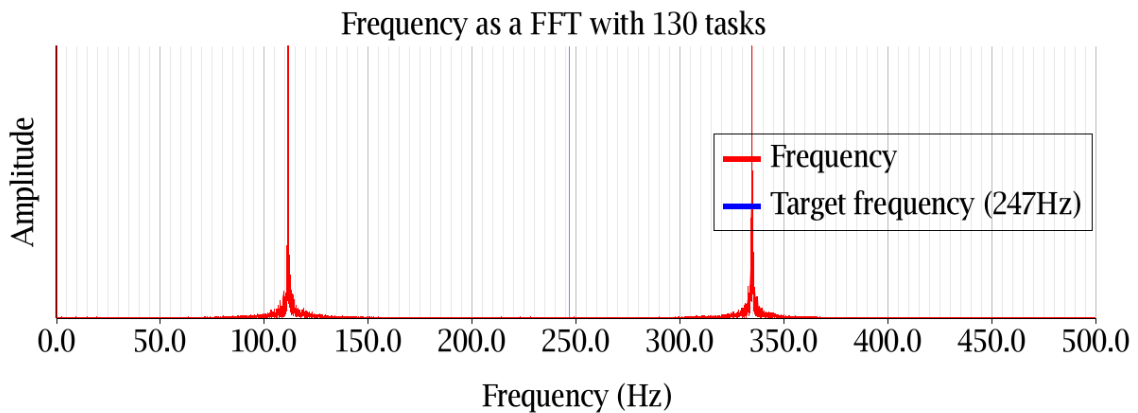


Figure B.14: FFT of the output signal when using 130 background tasks in RTIC.

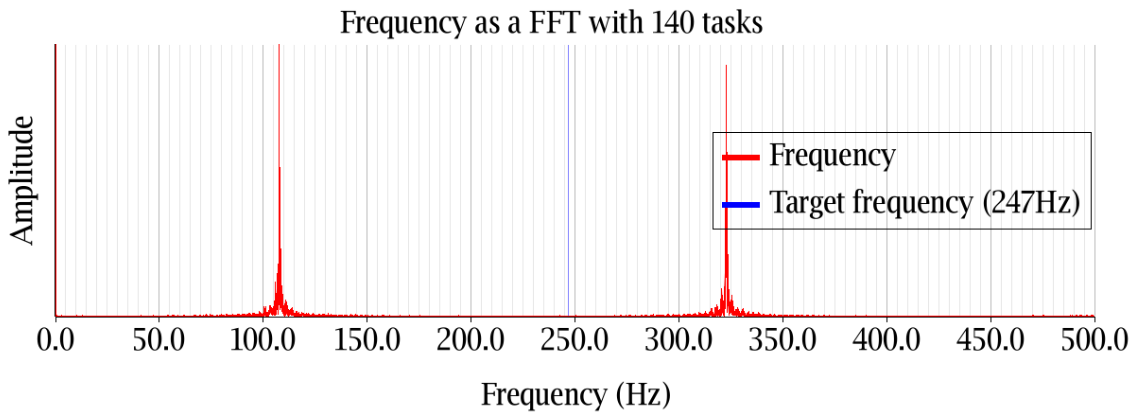


Figure B.15: FFT of the output signal when using 140 background tasks in RTIC.

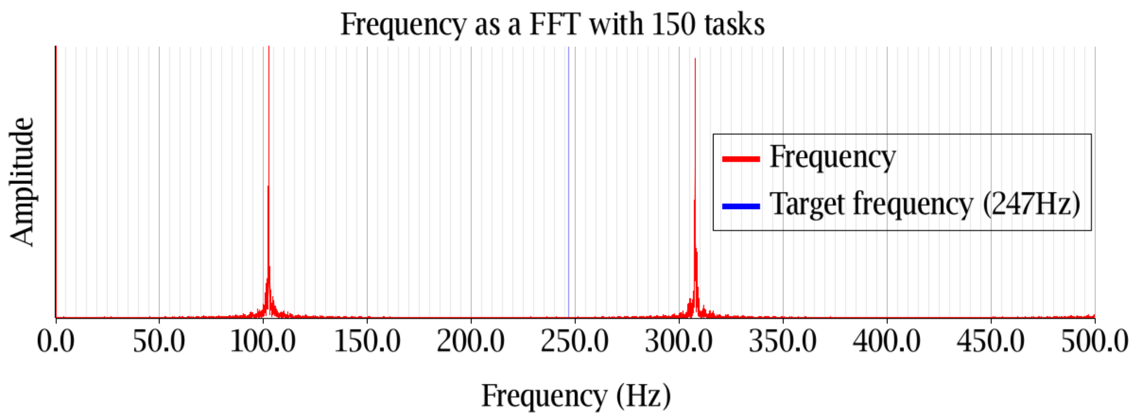


Figure B.16: FFT of the output signal when using 150 background tasks in RTIC.

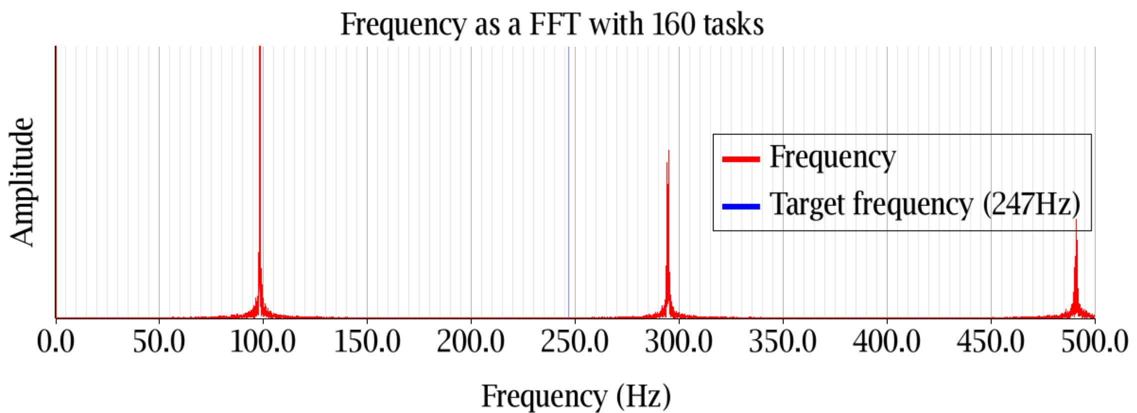


Figure B.17: FFT of the output signal when using 160 background tasks in RTIC.

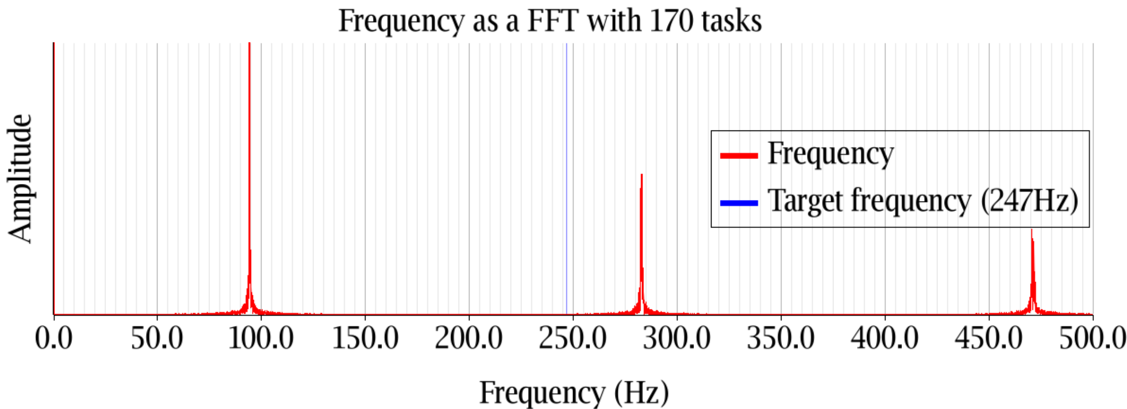


Figure B.18: FFT of the output signal when using 170 background tasks in RTIC.

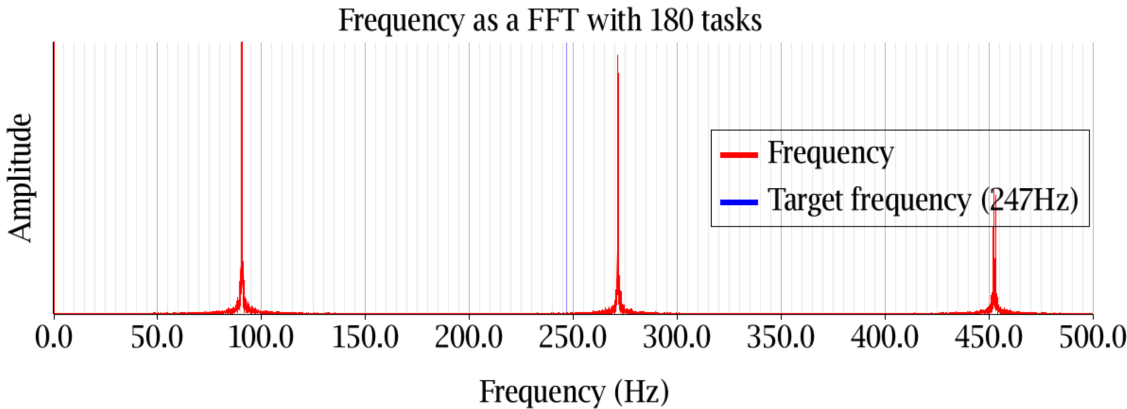


Figure B.19: FFT of the output signal when using 180 background tasks in RTIC.

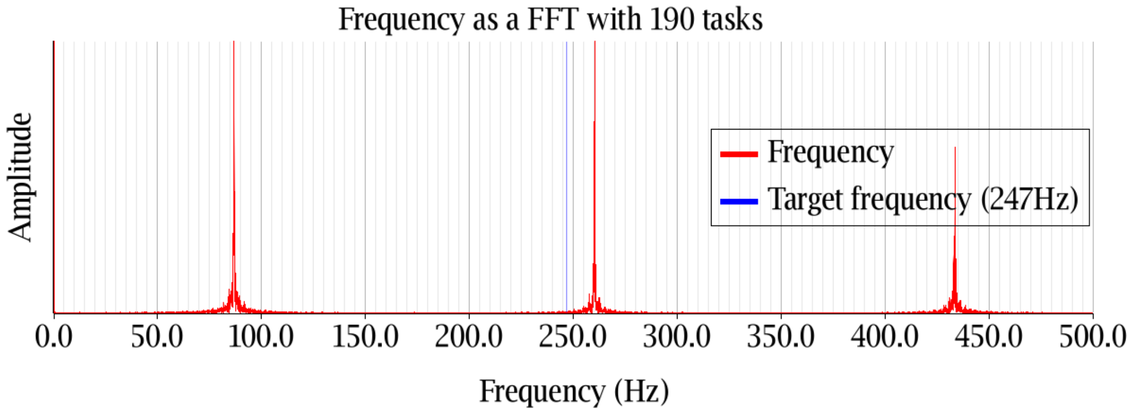


Figure B.20: FFT of the output signal when using 190 background tasks in RTIC.

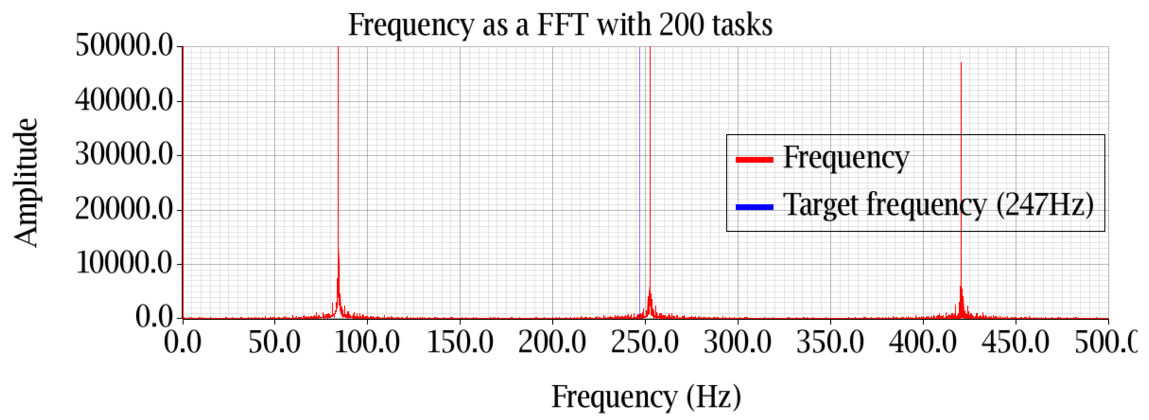


Figure B.21: FFT of the output signal when using 200 background tasks in RTIC.