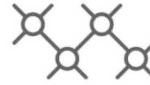



# Modern C++ in Embedded Systems

## Utilization of Modern C++ Language Features for Creating Zero-Overhead Firmware Architectures for Resource-Constrained Embedded Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Embedded Systems**

eingereicht von

**Marcell Ferenc Juhász, BSc**
Matrikelnummer 12034926

an der Fakultät für Elektrotechnik und Informationstechnik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Sauter Thilo
Dipl.-Ing. Albert Treytl
Dipl.-Ing. Thomas Bigler

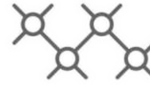
Wien, 22. November 2023

Marcell Ferenc Juhász

Technische Universität Wien
A-1040 Wien Karlsplatz 13 Tel. +43-1-58801-0 www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.




# Modern C++ in Embedded Systems

## Utilization of Modern C++ Language Features for Creating Zero-Overhead Firmware Architectures for Resource-Constrained Embedded Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Embedded Systems**

by

**Marcell Ferenc Juhász, BSc**

Registration Number 12034926

to the Faculty of Electrical Engineering and Information Technology

at the TU Wien

Supervisors: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Sauter Thilo
Dipl.-Ing. Albert Treytl
Dipl.-Ing. Thomas Bigler

Vienna, November 22, 2023

Marcell Ferenc Juhász

Technische Universität Wien
A-1040 Wien Karlsplatz 13 Tel. +43-1-58801-0 www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Declaration of Authorship

I, Marcell Ferenc JUHÁSZ, declare that this paper titled, "Modern C++ in Embedded Systems" and the work presented in it are my own. I confirm that I have written this work independently, have given full details of the sources and aids used, and have marked places in the work—including tables, maps and illustrations—which are taken from other works or from the Internet, either verbatim or in spirit, as borrowed, in any case indicating the source.

Vienna, August 8, 2023

Marcell Ferenc Juhász

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Abstract

Abstraction is a double-edged sword in programming languages. On one hand, it simplifies the management of complex systems, allowing developers to encapsulate intricate details behind intuitive interfaces. On the other hand, there is a longstanding perception that abstraction equates to increased runtime overhead, which can be a significant deterrent in resource-constrained environments.

One common misconception often encountered is the fear that object-oriented languages lead to runtime overhead, particularly in resource-constrained environments. This belief stems from the idea that abstraction, a key component of OOP, invariably adds performance cost. This fear is especially prevalent in the world of embedded systems, where every byte of memory and every millisecond of processing time is of paramount importance.

The aim of this thesis is to demolish the misconception that adding layers of abstraction inevitably results in less efficient software. This work serves as a demonstration of how modern C++ language features make it possible to write firmware that surpasses the equivalent C implementation in efficiency, maintainability and readability.

In the first part of the research, the focus was on how modern C++ language features can be used to optimize code commonly used in embedded firmware development. It is shown that by using C++, parts of the firmware that are inherently constant due to the embedded nature of the project could be executed at compile time.

In the second part, abstractions common to object-oriented programming languages are examined, analyzing their impact on runtime performance, and identifying which abstractions could be used without incurring additional runtime overhead. Many of these abstractions were found to be zero-cost, except for those explicitly designed for runtime.

In the final part, measurements on compilation time and runtime were performed to substantiate the conclusions based on the previous detailed analysis. The findings supported the earlier conclusions, indicating that C++ can be used to implement embedded firmware that is superior to its C counterpart.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

Abstraktion ist ein zweischneidiges Schwert in Programmiersprachen. Einerseits vereinfacht sie die Verwaltung komplexer Systeme und ermöglicht es Entwicklern, komplizierte Details hinter intuitiven Schnittstellen zu kapseln. Andererseits herrscht seit langem die Auffassung, dass Abstraktion mit einem erhöhten Laufzeit-Overhead gleichzusetzen ist, was in ressourcenbeschränkten Umgebungen ein erhebliches Hindernis darstellen kann.

Ein häufig anzutreffendes Missverständnis ist die Befürchtung, dass objektorientierte Sprachen zu einem höheren Laufzeit-Overhead führen, insbesondere in Umgebungen mit eingeschränkten Ressourcen. Dieser Glaube rührt von der Vorstellung her, dass Abstraktion, eine Schlüsselkomponente von OOP, unweigerlich zu Leistungseinbußen führt. Diese Befürchtung ist besonders in der Welt eingebetteter Systeme weit verbreitet, wo jedes Byte Speicher und jede Millisekunde Verarbeitungszeit von größter Bedeutung sind.

Ziel dieser Diplomarbeit ist es, zu widerlegen, dass das Hinzufügen von Abstraktionsschichten unvermeidlich zu weniger effizienter Software führt. Diese Arbeit dient als Demonstration, wie moderne C++ Features es ermöglichen, Firmware zu schreiben, die die entsprechende C-Implementierung an Effizienz, Wartbarkeit und Lesbarkeit übertrifft.

Im ersten Teil der Forschungsarbeit lag der Schwerpunkt darauf, wie moderne C++-Sprachfunktionen zur Optimierung von Code genutzt werden können, der häufig bei der Entwicklung eingebetteter Firmware verwendet wird. Es wird gezeigt, dass durch die Verwendung von C++ Teile der Firmware, die aufgrund des eingebetteten Charakters des Projekts von Natur aus konstant sind, zur Kompilierzeit ausgeführt werden können.

Im zweiten Teil werden Abstraktionen, die in objektorientierten Programmiersprachen üblich sind, untersucht, ihre Auswirkungen auf die Laufzeitleistung analysiert und festgestellt, welche Abstraktionen ohne zusätzlichen Laufzeit-Overhead verwendet werden können. Es wurde festgestellt, dass viele dieser Abstraktionen keinen Overhead verursachen, mit Ausnahme derjenigen, die explizit für die Laufzeit konzipiert sind.

Im letzten Teil wurden Messungen der Kompilierungs- und Laufzeit durchgeführt, um die Schlussfolgerungen aus der vorangegangenen detaillierten Analyse zu untermauern. Die Ergebnisse bestätigten die früheren Schlussfolgerungen und zeigten, dass C++ für die Implementierung eingebetteter Firmware verwendet werden kann, die ihrem C-Pendant überlegen ist.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

**Abstract** **v**

**Kurzfassung** **vii**

**Contents** **xii**

**1 Introduction** **1**

1.1 Motivation and Problem Statement . . . . . . . . . . . . . . . . . . . . . 1

1.1.1 Object-Oriented Programming Languages . . . . . . . . . . . . 1

1.1.2 Zero-Cost Abstraction . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Methodology and Outline . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Basic Concepts of Embedded Software** **5**

2.1 Compilation Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Firmware Structure and Initialization Process . . . . . . . . . . . . . . . 6

**3 Environment And Platform** **8**

3.1 Host Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.2 Target Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.3 Minimal Compilable Project . . . . . . . . . . . . . . . . . . . . . . . . 10

**4 Optimizing for Runtime** **12**

4.1 C Runtime Overhead . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

4.2 C++ Type System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.3 Templates and Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.4 Decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4.5 Calculations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**5 Overhead of Abstractions** **26**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.1 Encapsulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.2 Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.3 Polymorphism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.3.1 Runtime Polymorphism . . . . . . . . . . . . . . . . . . . . . . 35
5.3.2 Static Binding - "Compile-Time Polymorphism" . . . . . . . . . 39
5.3.3 Method Overloading . . . . . . . . . . . . . . . . . . . . . . . . 44

**6 Results and Practical Evaluation 46**

6.1 Measurements of Runtime Overhead Elimination . . . . . . . . . . . . 46
6.1.1 Measurement Methodology . . . . . . . . . . . . . . . . . . . . . 47
6.1.2 Binary Size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.1.3 Compilation Time . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.1.4 Link Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.1.5 Execution Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.2 Overhead Assessment of Abstractions . . . . . . . . . . . . . . . . . . 55
6.2.1 Measurement Methodology . . . . . . . . . . . . . . . . . . . . . 56
6.2.2 Binary Size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.2.3 Compilation Time . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.2.4 Link Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2.5 Execution Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.3 Practical Embedded Example . . . . . . . . . . . . . . . . . . . . . . . 59
6.3.1 Project Structure and Testing Environment . . . . . . . . . . . . 59
6.3.2 C++ Hardware Abstraction Layer . . . . . . . . . . . . . . . . . 61
6.3.3 Main Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.3.4 Fully Optimized C Code . . . . . . . . . . . . . . . . . . . . . . 64
6.3.5 Overhead Assessment and Conclusion . . . . . . . . . . . . . . . 66

**7 Further Considerations 68**

7.1 Code Bloating . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.2 Floating-Point Operations . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.3 Hidden Dynamic Memory Allocations . . . . . . . . . . . . . . . . . . 72
7.4 References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

**8 Conclusion** **76**

**A Startup File** **78**

**B Linker Script** **82**

**C Makefile** **85**

C.1 C Makefile . . . 85

C.2 C++ Makefile . . . 86

**D Benchmark Toolchain** **87**

D.1 Base main.c Jinja Template . . . 87

D.2 main.c Jinja Template . . . 87

D.3 Base main.cpp Jinja Template . . . 89

D.4 main.cpp Jinja Template . . . 89

D.5 JSON Configuration File Example . . . 91

D.6 Generator Script . . . 91

D.7 Plotting Script . . . 93

D.8 Shell Script for Running Benchmark for a Specific Configuration . . . 94

D.9 Shell Script for Running the Benchmark . . . 96

D.10 Generator Script for Abstraction Overhead Measurements . . . 96

D.11 Plotting Script for Abstraction Overhead Measurements . . . 104

D.12 Benchmarking Script for Abstraction Overhead Measurements . . . 105

**E Led Blinking** **110**

E.1 hal/base/constants.hpp . . . 110

E.2 hal/gpio/constants.hpp . . . 111

E.3 hal/gpio/gpio.hpp . . . 113

E.4 hal/gpio/registers.hpp . . . 114

E.5 hal/rcc/constants.hpp . . . 116

E.6 hal/rcc/registers.hpp . . . 117

E.7 hal/register.hpp . . . 118

E.8 hal/concepts.hpp . . . 119

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

E.9 hal/led.hpp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
E.10 hal/main.cpp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

**Bibliography** **124**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

# Chapter 1

# Introduction

## 1.1 Motivation and Problem Statement

With the advent of an increasingly digital world, the importance of efficient and highly structured programming languages has never been more relevant. The era of Internet of Things (IoT) has paved the way for embedded systems to become a common component of our day-to-day lives. These systems require firmware architectures that are both structurally sound and cost-effective in terms of performance overhead. Modern C++, with its extensive set of features, fits this criterion and provides the tools necessary to create such frameworks.

Abstraction is a double-edged sword in programming languages. On the one hand, it simplifies the management of complex systems, allowing developers to encapsulate intricate details behind intuitive interfaces. On the other hand, there is a longstanding perception that abstraction equates to increased runtime overhead, which can be a significant deterrent in resource-constrained environments. Modern C++, however, shatters this misconception through a key concept: zero-cost abstractions [1].

### 1.1.1 Object-Oriented Programming Languages

In the complex world of programming languages, two dominant paradigms reign: object-oriented programming (OOP) and procedural programming. OOP, utilized by languages such as C++, Python, and Java, centralizes its design around objects, i.e., instances of classes, which encapsulate both data and the functions that operate on that data. This approach promotes structured, reusable, and modular code, making it intuitive to design, read, and maintain. Features such as encapsulation, inheritance, and polymorphism facilitate high levels of abstraction and complex data interactions, offering programmers powerful tools to model the real world within their software systems[2].

Conversely, procedural programming languages, exemplified by C and FORTRAN, focus on the procedure or action to be performed rather than the data being manipulated. Programs in these languages are generally a series of tasks (procedures or functions) that are carried out sequentially[2]. While procedural languages can be

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

straightforward and efficient for simple tasks, their linear nature can lead to challenges in code maintenance, readability, and scalability when tackling complex systems.

One common misconception often encountered is the fear that object-oriented languages lead to runtime overhead, particularly in resource-constrained environments. This belief stems from the idea that abstraction, a key component of OOP, invariably adds a performance cost. This fear is especially prevalent in the world of embedded systems, where every byte of memory and every millisecond of processing time is of paramount importance. The aim of this thesis is to demolish the misconception that adding layers of abstraction inevitably results in a less efficient software. This work serves as a demonstration on how modern C++ language features make it possible to write firmware that surpasses the equivalent C implementation in efficiency, maintainability and readability. The focus will be on exploring the modern C++ language and its powerful features that enable the construction of highly structured firmware architectures with zero runtime overhead.

### 1.1.2 Zero-Cost Abstraction

Zero-cost abstraction refers to the ability to use higher-level programming constructs without incurring any additional runtime overhead. This is achieved by performing extensive work at compile-time, rather than runtime. The premise of zero-cost abstractions is that what can be done at compile-time should be done at compile-time. The result is a cleaner, more maintainable codebase without sacrificing runtime performance.

Modern C++ has robust support for this concept, offering advanced features that allow developers to leverage high-level abstractions with the assurance that they will not negatively impact performance. Features such as *templates*, *constant expressions*, and *immediate functions* are utilized to provide abstractions that are evaluated and optimized at compile-time, thereby avoiding additional runtime costs.

For instance, C++ *templates* allow developers to write generic, type-agnostic code without sacrificing type safety or incurring runtime overhead. The compiler generates the necessary specific implementations based on the types used, ensuring no extra cost at runtime. Similarly, *constant expressions* and *immediate functions* enable computations and function calls to be resolved at compile-time, removing any overhead they would have incurred if evaluated at runtime.

This zero-cost abstraction capability is a game-changer for resource-constrained environments, such as those often encountered in the embedded systems and IoT space. It opens the door for developers to leverage the benefits of object-oriented programming and high-level abstractions without fearing the penalty of increased runtime cost. The subsequent sections of this thesis are dedicated to exploring how modern C++ exploits these features to create highly structured, zero-overhead firmware architectures, thus dismantling the fear often associated with the use of OOP in resource-constrained environments.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## 1.2 Methodology and Outline

Throughout this document, the main focus will be on the runtime performance and binary size of different source code examples. The basis for meaningful comparison will be an empty project, containing a single main function with an empty infinite loop, alongside with the necessary file for successful cross-compilation. The runtime overhead assessment will be conducted in two distinct ways: firstly, by inspecting the size of the *.text* section of the compiled firmware, and secondly, through an exploration and analysis of the disassembled ELF file, focusing on the assembly code. Both methods are necessary, because although the size of the *.text* section (explained in detail in Section 2.2) is a simple and understandable metric, in and of itself it is not sufficient to assess runtime overhead, because it can also contain loops, branches and data. Therefore, the analysis will focus on the assembly code of the disassembled ELF file to gather meaningful insight into the factors contributing to changes in the binary size. Throughout this document, *text size* will be used to refer to the *size of the .text section in bytes*.

In the first part of the thesis, the demonstration will center on how modern language features can be effectively utilized to optimize runtime performance. This will include an examination of a simplified code snippet written in C, followed by an assessment of the runtime overhead. The code will mirror how real life Hardware Abstraction Layer (HAL) libraries are implemented in the embedded world.

The example codebase will then be iteratively reworked using modern C++ language features. Optimization steps that enable developers to shift runtime overhead into compile-time will be performed. The above mentioned methods will be used to assess the runtime overhead of the C++ code and compare the results with the functionally equivalent C implementation. The goal is to demonstrate that in many situations, especially in embedded projects, C++ can lead to a more efficient firmware.

In the subsequent part of the thesis, the main focus will be on assessing the runtime overhead of various OOP language features. The analysis will demonstrate how different C++ features can be utilized to build layers of abstractions without introducing unnecessary runtime overhead. Additionally, it will delve into some of the language's drawbacks, illustrating how careless implementations can lead to significant runtime overhead.

In the subsequent part of the thesis, the initial focus will be on summarizing the conclusions from previous chapters regarding the advantages and disadvantages of the discussed language features and programming methods. This will be complemented by presenting relevant measurements that elucidate the benefits of optimization steps and their impact on compilation time. Following this, some of the previously discussed programming techniques and presented architecture will be applied to construct an LED blinking application, regarded as the *Hello World* of embedded systems. An examination of the disassembled binary will reveal the optimization results performed by the compiler and assess the runtime overhead of the abstractions. Furthermore, a comparison will be made with a highly efficient and functionally equivalent C implementation. The overarching goal of this part of the thesis is to challenge the misconception that abstractions inherently carry runtime overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Lastly, the thesis will analyze and discuss some of the hidden dangers of the language that every embedded C++ developer should be aware of.

The driving goal of the thesis is to encourage embedded developers to start writing firmware more in C++, so that the embedded world can leave behind the remnants of a bygone era and move to a more sustainable and modern future. It is important to note right at the beginning that it is not intended to diminish the usefulness and robustness of C. Even with proficient knowledge of C++, there are situations where it is not possible to eliminate runtime overhead that is introduced by the language. In highly resource-constraint environments or in hard real-time applications, C can indeed still be the language of choice because of its predictability. On the other hand, in many cases, C++ can actually help to reduce runtime overhead, whereas in other cases, the benefits of using C++ outweigh the small potential for added runtime overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Chapter 2

# Basic Concepts of Embedded Software

Before diving into the specifics of embedded systems development, it's crucial to lay a solid groundwork of the key principles and concepts that underpin this field. This chapter aims to briefly illuminate these fundamentals, such as the significance of near-hardware languages and the role of cross-compilation in embedded systems programming.

## 2.1 Compilation Process

When it comes to embedded systems, C and C++ languages stand out as the go-to options, although there are new contestants in the field, like Rust and TinyGo [3]. This is primarily due to their classification as near-hardware languages. In essence, these languages allow for a closer interaction with the hardware by granting programmers direct access to the system's registers, the components that control and manage the hardware's behavior. By manipulating these registers, programmers can have fine-tuned control over the system's operations, making C and C++ the preferred choices for embedded systems programming.

The C/C++ compilation takes several steps. Although the exact process may vary depending on the specific compiler, target architecture, and build settings, the following gives a good overview of the process [4]:

- **Preprocessing** : In this stage, the preprocessor (a part of the compiler) takes the source code and deals with the *#include*, *#define*, and other preprocessor directives, replacing parts of the source code. The output is a "pure" C or C++ file without these directives.
- **Compilation** : The source code file is then compiled into an assembly language file specific to the target architecture. This file contains low-level, human-readable instructions specific to the machine architecture.
- **Assembly** : The assembler takes the assembly language file and converts it into an object file, which contains the machine language instructions (binary code) equivalent to the assembly instructions.
- **Linking** : Lastly, the linker takes one or more object files along with libraries needed by the program, and combines them to create the final executable. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

linker resolves references to functions and variables in different object files or libraries, and assigns final addresses to functions and variables.

The resulting executable file contains machine code that can be loaded into memory and executed by the system. If the program was written in C++, an additional name mangling step may occur before linking, which encodes additional information (such as function argument types) into the names of functions and variables, due to the more complex semantics of C++ compared to C.

When dealing with embedded systems, the compiler toolchains are not available on the target platform, therefore native compilation is not possible. In such cases, the solution is *cross-compilation*, where the code is compiled on one system (the host) to run on a different system (the target). The firmware binary that results from this process is loaded into the system's onboard memory, where it directs the system's operations. This binary is composed of different sections (discussed in Section 2.2) like the reset vector, which determines the initial point of execution after a system reset, and sections for code, initialized variables, uninitialized variables, and so on. Some sections must be at specific locations in the system's memory, otherwise the code will not execute [5].

In many cases, embedded developers do not have the support of any operating system on the target. This type of programming is called *bare-metal programming*, which refers to programming at the hardware level. Here, the programmer is responsible for controlling all aspects of the system, from managing interrupts and task scheduling to handling memory management. This approach is typically used in contexts where resources are limited or precise control over system operations is required [5]. Furthermore, developers often need to be aware of the time it takes for a specific code to execute on the target, because it can be crucial to the correct functionality of a system. For example, dynamic memory allocations or time-consuming algorithms should be avoided in an interrupt service routine [6].

## 2.2 Firmware Structure and Initialization Process

Understanding the firmware structure and how it is loaded into an embedded device's memory at startup is fundamental for embedded systems development.

Firmware in an embedded system is typically structured in sections, each having a specific purpose. It includes the following main sections [7]:

- **.text** : This section, often referred to as the code section, contains the actual machine code that the CPU executes. It includes executable instructions generated by compiling and linking the source code. Typically, this section is read-only to prevent accidental modifications.
- **.data** : This section holds initialized static and global variables used by the program. The values for these variables are specified in the program itself, and they retain their values throughout the program's lifetime.
- **.rodata** : Stands for "Read-Only Data". This section contains static constants rather than variables.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- **.bss** : Known as the Block Started by Symbol, this section includes uninitialized static and global variables. Unlike the *.data* section, variables in the *.bss* section are not explicitly initialized by the programmer and are set to zero by default.
- **.heap** : This section is used for dynamic memory allocation during program runtime.
- **.stack** : This is the section that holds the program stack, a LIFO structure typically used for storing local variables and function call information.

When the embedded device is powered on, or reset, the startup process kicks off. The processor's program counter jumps the a predefined location in memory, where the reset vector resides. The first field of the reset vector contains a memory address which is the address fo the reset handler, implemented by the programmer in a file known as the startup file. The reset handler is the entry point of the firmware and is the first piece of code executed after a reset. Its main role is to set up the environment to a known state before the main application runs. The firmware is typically stored in a non-volatile memory such as flash. During this preparations process, the firmware's different sections are either copied into RAM or prepared in some way [7]:

- **.text** : This section, being read-only, can be executed directly from flash memory, or in some systems for performance reasons, it might be copied into the RAM.
- **.data** : At startup, initialized data is copied from the flash memory to the RAM, so that the program can access and modify these variables.
- **.rodata** : Like the *.text* section, the *.rodata* section can also be left in the flash memory, or it can be loaded into RAM depending on the system's requirements and available resources.
- **.bss** : Since this section represents uninitialized variables that should start as zero, there is no need to take up space in the firmware image in flash for this section. Instead, during initialization, a region of RAM of the appropriate size is allocated and zeroed out.
- **.heap** and **.stack** : The sizes of these sections usually aren't known at compile time, as they depend on the program's dynamic behavior. Memory for the heap and stack is set aside in RAM during initialization, typically with the stack starting from one end of the remaining memory and the heap from the other, growing towards each other.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Chapter 3

# Environment And Platform

## 3.1 Host Environment

Every example code presented in this section was compiled and tested on *Ubuntu 22.04.2 LTS*. For cross-compilation, the GNU Arm Embedded Toolchain was used with the version *10.3.1*.

The GNU Arm Embedded Toolchain is a comprehensive suite of software development tools specifically designed for Arm-based microcontrollers and embedded systems. Developed and maintained by Arm, the toolchain provides an open-source platform for developers to create, debug, and optimize applications targeting these devices [8].

The GNU Arm Embedded Toolchain comprises several key components. The most essential component is the GCC (GNU Compiler Collection) for Arm processors, which includes compilers for both C and C++. The toolchain also includes a number of other software tools, some of which are used in this work. The GNU Assembler (*as*) allows developers to write assembly code for Arm processors, while the GNU Binary Utilities (*binutils*) provide a set of programming tools for creating, modifying, and analyzing binary files. For debugging, the toolchain offers GDB (GNU Debugger), a powerful debugging tool that supports a wide range of debugging techniques [8].

In addition to the tools, the GNU Arm Embedded Toolchain includes a standard C library optimized for embedded systems, known as Newlib. Newlib is a lightweight C library suitable for the limited resources of embedded systems, as it offers a smaller memory footprint compared to traditional C libraries.

## 3.2 Target Platform

Projects were compiled for, and tested on an STM32F030F4 microcontroller. It is a highly integrated, 32-bit microcontroller produced by STMicroelectronics, a global semiconductor leader. This microcontroller belongs to the STM32 F0 series, which is part of the wider STM32 family that is based on Arm Cortex-M processors. The STM32F030F4 specifically features the Arm Cortex-M0 core, the smallest and most

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

energy-efficient processor in the Arm Cortex-M family, making it well-suited for cost-sensitive, low-power applications [9].

Throughout this thesis, the disassembled binary file will be frequently analyzed to identify the exact cause of changes in the text size. In order to fully understand these code snippets, it is essential to get familiar with the Cortex-M0 processor core registers. An overview is shown in Figure 3.1. The most important ones, in the context of this thesis, are the general-purpose registers, the stack pointer, the link register and the program counter. The general-purpose registers, as the name suggests, are used for general data operations. The stack pointer stores the memory address of the last data element and is used to allocate memory on the stack for local variables, function parameters and other information. The link register stores return information for subroutines and function calls. The program counter contains the current program address and, among others, is used to jump to functions when they are called [10].

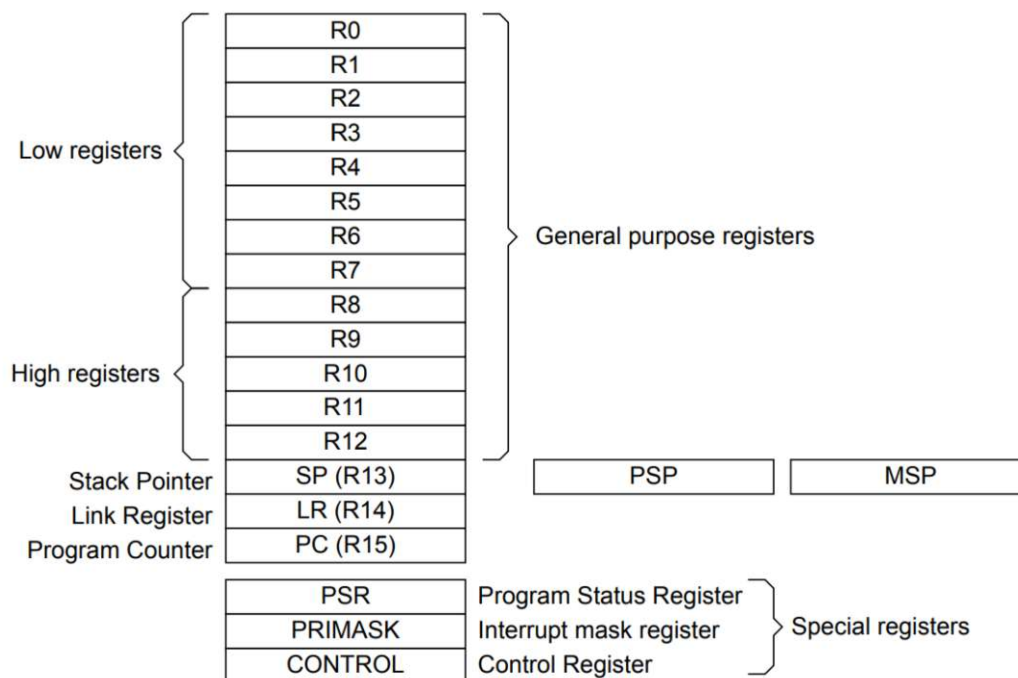



FIGURE 3.1: Processor core registers [10].

It is also important to gain a generic knowledge about the instruction set of the microcontroller. The processor implements a version of the *Thumb instruction set* [10]. Below is a list of common instructions that appear in the assembly code snippets throughout this work. Some operations listed here might have the letter *S* appended to them, which causes them to set internal flags based on the result of the operation. Note that this is not a complete and exhaustive list of instructions.

- **MOV** : This instruction copies the value from one register to another.
- **ADD** : Adds the values of two registers together and stores the result in a third register.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **SUB** : Subtracts the value of one register from another and stores the result in a third register.
- **MUL** : Multiplies the values in two registers and stores the result in a third register.
- **CMP** : Compares two register values and sets the condition flags.
- **AND** : Performs a bitwise AND operation on two register values and stores the result in a third register.
- **ORR** : Performs a bitwise OR operation on two register values and stores the result in a third register.
- **B** : Causes a jump to a different location in the program.
- **BL** : Causes a jump to a different location in the program, storing the return address in the link register.
- **BX** : Causes a jump to a different location, which is stored in a register, in the program.
- **BICS** : Performs a bit clear operation on a register based on masks stored in other registers.
- **LDR** : This instruction is used to load a value from memory into a register.
- **STR** : This instruction is used to store a value from a register into memory.
- **LSL** : Logical shift left instruction.
- **LSR** : Logical shift right instruction.
- **NOP** : No operation.
- **POP** : This instruction is used to pop registers from the stack.
- **PUSH** : This instruction is used to put registers on the stack.

## 3.3 Minimal Compilable Project

Two projects were set up for compiling the different code snippets and comparing the results between the traditional C and the modern C++ solutions. The following three files are necessary for compiling code for an STM32 microcontroller (and, in fact, for most microcontrollers):

- **Linker Script** : The linker script, usually having the .ld extension, is a file that guides the linker in arranging the compiled code and data in the memory of the target device. This includes the flash memory where the program code goes, the RAM for static and dynamic data, and any special sections like bootloaders or interrupt vector tables. Each STM32 microcontroller variant has its own memory layout, so the linker script should be written (or generated) specifically for the target device.
- **Startup File** : The startup file is a low-level code file, usually written in assembly language, that runs immediately after the microcontroller is powered up or reset. This file typically sets up important settings like the vector table, default interrupt handlers, and initial stack pointer. It may also initialize static data and BSS to zero. The startup file is specific to each microcontroller family, and

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

sometimes even specific to individual microcontroller models within a family, due to different peripherals and interrupt vector tables.

- **Main File** : This is the main source code file of the firmware, where the *main* function resides. The *main* function is called from the startup file after all the initial setup is done. This file will include application logic and calls to libraries or APIs for peripheral usage, which is up to the user to define. It is written in C or C++, as these languages offer higher-level abstractions and are more suitable for application development than the assembly language.

For the sake of repeatability, the startup files and the linker script, as well as two very simple makefiles are provided in Appendix A, Appendix B and Appendix C respectively. The files are based on the ones provided by the official STM32 Cube IDE.

The most crucial aspect of the compilation process, in relation to the goal of this study, is that the compiler is instructed to optimize for binary size. For each compiled code, the text size will be assessed and the disassembled binary analyzed. To render this assessment meaningful, a baseline is established for future comparisons. Consequently, an empty project has been set up in both C and C++ for this purpose. Empty in this context means that the application consists of a single main function containing only an infinite loop. Other than that, there are no variables and functions defined. Later sections will refer to this firmware as **base firmware**. The text size of the base firmware is **744 bytes** for both the C and the C++ versions. This value can be retrieved using the provided makefile or the *objdump* utility directly.

Please be aware that most of the code snippets in this document serve demonstration purposes only. This means that example codes are not written to perform some meaningful task on the given hardware, but rather to produce a simple assembly that can be meaningfully analyzed and compared to alternative solutions.

At the time of writing this document, the C++20 standard is the most current complete C++ standard and hence it is the version that is used in the project.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Chapter 4

# Optimizing for Runtime

In the realm of embedded systems, firmware architectures need to balance the structured organization provided by OOP and the performance efficiency traditionally associated with procedural languages. Here, modern C++ language features present a compelling case. These features, when utilized correctly, offer the potential to create highly structured, zero-overhead firmware architectures.

In this chapter, an exploration will be conducted into both basic and modern C++ language features that can assist developers in eliminating runtime overhead. The features to be examined in this section are as follows:

- C++ Type System
- Template Metaprogramming
- Constant Expressions
- Concepts
- Immediate Functions

## 4.1 C Runtime Overhead

In this section, an example code implemented in C will be examined to assess the runtime overhead added to the firmware. In Sections 4.2, 4.3, 4.4 and 4.5, it will be demonstrated how the runtime overhead can be eliminated using modern C++.

In the context of C-based Hardware Abstraction Layer (HAL) libraries, one often observes the use of defined structures designated for the initialization and management of diverse hardware components. This particular paradigm is evident in the HAL libraries of ESP32 and STM32 microcontrollers. The discussion will primarily focus on a simplified example of a HAL library that sufficiently mirrors the characteristics of a comprehensive HAL implementations but allows for meaningful analysis. A simplified example of such a structure defined in HAL libraries is shown in Listing 4.1. It has two fields that must be set by the developer according to the desired behavior. The *pin* integral value contains a value that is used to select the desired pin. The *mode* value is used to select between different pin operation modes, such as input, output, analog or alternate pin function.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
typedef struct {
    uint32_t pin;
    uint32_t mode;
} GPIO_InitStruct;
```

LISTING 4.1: Simplified example of C HAL library GPIO initialisation structure.

Note that in the structure, the *pin* and the *mode* values are of simple integral types. Although usually there are only sixteen pins on one IO port of a microcontroller and there are only around three or four possible GPIO operation modes, there is no enforced restriction on the values of these fields of the structure. Runtime checks can be - and usually are - introduced into the code to avoid incorrect initialization. The hardware abstraction layer usually defines the necessary values, macros, bitmasks and registers that the rest of the library and the firmware developer can use. An example of the defined values and value checking macros are given in Listing 4.2. Memory-mapped registers are given by address and converted to a pointer to be accessed. Such memory-mapped registers must be denoted with the *volatile* type qualifier to prevent compiler optimization. The *volatile* qualifier tells the compiler that the value might change independent from the code being compiled, and hence reading and writing the value must not be optimized. Register values can not only be changed by the programmer during the firmware execution, but are changed by the hardware itself as a consequence of external or internal hardware events and states.

Bitmasks, modes and other constant values are defined with specific integer values. Using these defined values, developers can read register values, apply the defined masks to retrieve the relevant pins and compare the result with the defined modes and other characteristics. Similarly, the masks can be used to set and clear only the relevant bits of the register and set them according to the desired functionality.

```
#define MODER   (*((volatile uint32_t*)0x48000000))
#define OTYPER  (*((volatile uint32_t*)0x48000004))
#define OSPEEDR (*((volatile uint32_t*)0x48000008))
/* ... */
#define GPIO_PIN_6  ((uint16_t)0x0006U)
/* ... */
#define MODER_MASK   (0x3UL)
#define OTYPER_MASK  (0x1UL)
#define OSPEEDR_MASK (0x3UL)
/* ... */
#define GPIO_MODE_INPUT  (0x00000000U)
#define GPIO_MODE_OUTPUT (0x00000001U)
/* ... */
#define  GPIO_SPEED_FREQ_LOW (0x00000000U)
/* ... */
#define GPIO_MODE        (0x00000003U)
#define GPIO_OUTPUT_TYPE (0x00000010U)
/* ... */

#define IS_GPIO_PIN(X) ((X) < 16 && (X) >= 0)
#define IS_GPIO_MODE(X) (((X) == GPIO_MODE_INPUT) || ((X) ==
    GPIO_MODE_OUTPUT))
```

LISTING 4.2: GPIO-related constants defined in HAL library.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In the main application, the developer needs to create a variable with the type *GPIO_InitStruct*, subsequently configuring its members to align with the application's requirements. To illustrate, suppose the application requires configuring pin 6 as an input. The code snippet provided in Listing 4.3 demonstrates this scenario. The *pin* field of the structure is set to the defined value *GPIO_PIN_6* in line 3, and the mode is set to the defined value *GPIO_MODE_INPUT* in line 4. The *GPIO_Init* function accepts a pointer to the instantiated data structure and performs the initialization of the requested GPIO pin.

```
1 GPIO_InitStruct gpio_init_struct = {0};
2
3 gpio_init_struct.pin = GPIO_PIN_6;
4 gpio_init_struct.mode = GPIO_MODE_INPUT;
5
6 GPIO_Init(&gpio_init_struct);
```

LISTING 4.3: Example of C GPIO initialization.

The function *GPIO_Init* is implemented in the HAL. It is either provided by the manufacturer of the microcontroller or it is written by the developer. It is a low-level code that deals with reading and writing registers, setting bits according to the desired configuration. An example implementation of this function in C is shown in Listing 4.4. As mentioned above the function takes a pointer to a structure. In lines 5 and 6, the relevant fields of the structure must be validated at runtime using the provided value validating macros. This is important, because invalid values might result in undesired values to be written to registers, which in turn can lead to unexpected behavior of the hardware which can potentially damage the device. For different configurations, different registers need to be written. Therefore, runtime branches are introduced into the software, like the one in line 9. The *OSPEEDR* (Output Speed Register) and the *OTYPER* (Output Type Register) need to be written only if the pin is configured in output mode. The *MODER* (Mode Register) needs to be set for every operation mode of the pin, hence it is done in lines 21, 22 23 and 24, outside of the conditional block. Only specific bits need to be set and cleared in the registers. Therefore, bitmasks need to be calculated by shifting a base bitmask by the value of the GPIO pin number, like it is done in lines 11 and 12.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
void GPIO_Init(GPIO_InitStruct* gpio_init_struct) {
    uint32_t temp;

    /* check the values */
    if (!IS_GPIO_PIN(gpio_init_struct->pin)) { return; }
    if (!IS_GPIO_MODE(gpio_init_struct->mode)) { return; }

    /* configure the GPIO based on the settings */
    if (gpio_init_struct->mode == GPIO_MODE_OUTPUT) {
        temp = OSPEEDR;
        temp &= ~(OSPEEDR_MASK << (gpio_init_struct->pin * 2u));
        temp |= (GPIO_SPEED_FREQ_LOW << (gpio_init_struct->pin * 2u));
        OSPEEDR = temp;

        temp = OTYPER;
        temp &= ~(OTYPER_MASK << gpio_init_struct->pin);
        temp |= (((gpio_init_struct->mode & GPIO_OUTPUT_TYPE) >> 4u) <<
            gpio_init_struct->pin);
        OTYPER = temp;
    }

    temp = MODER;
    temp &= ~(MODER_MASK << (gpio_init_struct->pin * 2u));
    temp |= ((gpio_init_struct->mode & GPIO_MODE) << (gpio_init_struct
        ->pin * 2u));
    MODER = temp;

    /* ... */
}
```

Listing 4.4: Example implementation of low-level C GPIO initialization function.

Such validity checks, if-else statements and bitmask calculations, as shown above in Listing 4.4, all add runtime overhead to the firmware. This additional runtime overhead can be verified by looking at the disassembled binary. Listing 4.5 shows the section where the pin number is verified. Its value is loaded into the *r3* register in line 1 and compared to the immediate value 15 in line 2. If the pin number is higher than 15, meaning that it is invalid, the program jumps to the end of the function and returns (line 3).

```
800019c:      6803      ldr     r3, [r0, #0]
80001a0:      2b0f      cmp     r3, #15
80001a2:      d825      bhi.n   80001f0 <GPIO_Init+0x54>
```

Listing 4.5: Assembly representation of the pin number validity check of the code from Listing 4.4 line 5.

This is not a large overhead, but occurs multiple times throughout the whole application. And not just in the initialization functions, but in every function that takes some integer parameter with no type restriction. This kind of input parameter check and runtime branching is unavoidable in a traditional C firmware. And it is noteworthy that low-cost microcontrollers usually do not include a branch predictor, and as such, the overhead is not mitigated. The same way, every bitmask calculation appears in the firmware as runtime overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

When examining the *main* function in the output of the *objdump* utility, as shown in Listing 4.6, the expected results are observed:

- Register values are pushed to the stack in lime 3.
- In the following lines, the local variable (initialization structure) is created and the values are set.
- The *bl* instruction in line 8 saves the return address in the link register (lr) and then jumps to the called function *GPIO_Init*.
- The *b.n* instruction in line 9 branches to the same address it is at, effectively locking the program in an infinite loop.

At first glance, this might not appear as an overhead, but a comparison with the equivalent C++ implementation will be made in a later section.

```
080001fc <main>:
 80001fc:       2300            movs    r3, #0
 80001fe:       b507            push    {r0, r1, r2, lr}
 8000200:       9301            str     r3, [sp, #4]
 8000202:       4668            mov     r0, sp
 8000204:       3306            adds    r3, #6
 8000206:       9300            str     r3, [sp, #0]
 8000208:       f7ff ffc8       bl      800019c <GPIO_Init>
 800020c:       e7fe            b.n     800020c <main+0x10>
```

LISTING 4.6: Call to the GPIO_Init function from the main function. Refer to Listing 4.3 for the implementation.

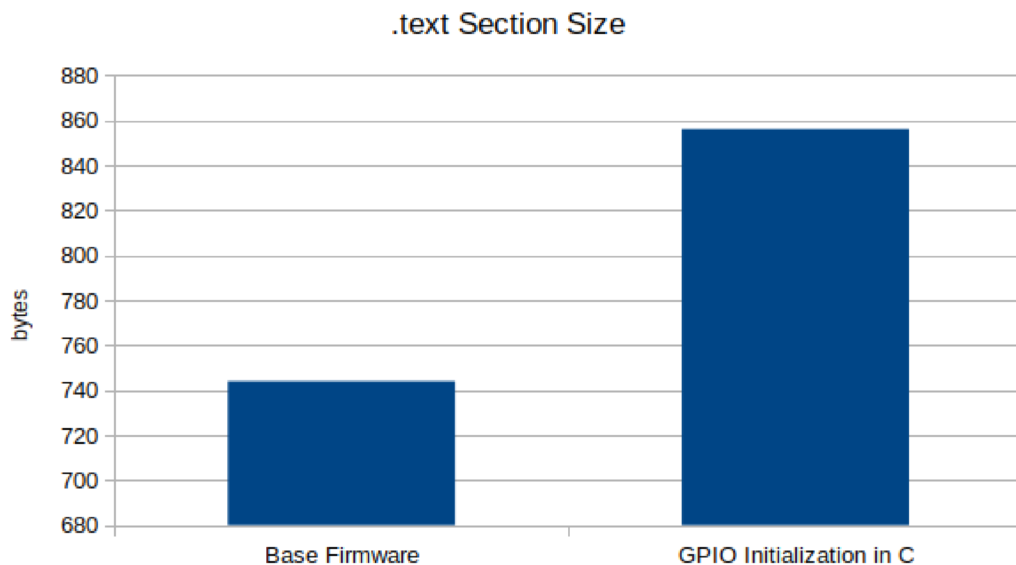



FIGURE 4.1: .text section size of GPIO initialization in C compared to the base firmware .text section size.

Subsequent sections will provide possible ways to eliminate runtime overhead, mostly by pushing previously runtime calculations to compile-time. Note that there

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

can be cases where the runtime overhead is unavoidable. For example, for applications designed to be configured by a configuration file at runtime, the necessary values are not available at compile-time to make use of these optimization methods. However, small and compact embedded firmware applications are usually not in this category and can be optimized using modern language features, compile-time calculations and decisions.

## 4.2 C++ Type System

C++ introduces a robust type system that offers several advantages over its predecessor, C. One of these advantages is the potential to reduce runtime overhead by establishing value constraints at compile time. This ability to perform checks during the compilation phase rather than at runtime enables the creation of more efficient and safer code, which is especially valuable in the domain of embedded systems where resources are limited and performance is paramount.

A prime example of this power can be found in C++'s *enum class* type. An enumeration in C++ is a distinct type whose value is restricted to a specific range of values. This differs from C-style enumerations, where the enumeration values are essentially treated as integral values and can be freely interchanged with them. This can lead to inadvertent errors if developers are not careful enough and assign an arbitrary integer to an *enum* variable. In contrast, *enum class* provides type safety, preventing these kinds of mistakes by not providing implicit conversion between the values of the scoped enumeration and the underlying integral type. As shown in Listing 4.7, the code in line 6 does not compile, since the integral type cannot be implicitly converted to the enum class value. Moreover, *enum class* allows for more efficient usage of memory and flexibility, as the size of a scoped enumeration type can be explicitly defined. In contrast, enumerations in C are usually implemented as integers.

```
enum class GPIO_Mode : std::uint8_t {
    input,
    output
};
/* ... */
GPIO_Mode mode = 5;    /* will not compile */
```

LISTING 4.7: Example of C++ enum class representing the possible GPIO modes and an example on type constraints enforced by the compiler.

The type system itself does not necessarily guarantee that the value is valid and holds one of the possible values of the enumerated type, but the developer must to put extra effort into introducing a bug to the application by an intentional type cast, as shown in Listing 4.8.

```
GPIO_Mode mode = static_cast<GPIO_Mode>(5);    /* will compile */
```

LISTING 4.8: Example on how an integral value can be converted to enumeration type in C++.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

By simply using *enum class* types instead of C-style enumerations, it could already be argued that the runtime value checks are not necessary, since it takes an extra effort to misuse the implemented functions. This way, the C++ type system not only provides mechanisms for improving runtime efficiency but also enhances type safety and code clarity.

## 4.3 Templates and Concepts

In embedded firmware applications, many parameters are declared by the surrounding hardware. Pin functions are dictated by the external components connected to the pins, and in most cases, these external components are fixed during board layout design. If a pin is connected to a status-indicating LED, it is known at the time of writing the firmware that the pin has to be configured as output. The same way, parameters like I2C frequency, clock source, clock frequency and ADC resolution are usually known at compile-time. Therefore it makes sense to use modern compiler features to eliminate the unnecessary runtime overhead. C does not provide support for differentiating between runtime and compile-time constants, but modern C++ gives a variety a features to deal with such concepts.

Templates are one of the most powerful and versatile features of C++, providing the ability to generate classes or functions at compile-time, based on template parameters. They constitute an essential component in any C++ programmer's toolkit, especially those working with embedded systems where runtime performance is of utmost importance. Templates offer the opportunity to shift much of the computational overhead from runtime to compile-time, which can yield substantial performance gains. The principle is straightforward: instead of deferring decisions until the runtime phase, the approach is to make as many decisions as possible during the compile-time phase. In other words, the price is paid once, at the time of compilation, rather than continuously paying a smaller price during runtime [11].

*Constraints* and *concepts*, introduced in C++20, are a significant enhancement to the template system in C++. They provide a high-level mechanism to specify and check the requirements on template arguments [12].

Before concepts were introduced, C++ templates were notorious for producing complicated and hard-to-understand error messages when constraints on types were not met. These constraints, or requirements for the type to support certain operations, were implicit and usually identified in the function or class template's body. When the actual type used did not meet the requirements, the compile-time error message was usually triggered deep within the template code, leading to error messages that were difficult to understand and difficult to link to the original problem.

Concepts address this issue by enabling developers to express the intended constraints explicitly. A *concept* is a kind of compile-time predicate that operates on types: a function or class template can specify a *concept* as a requirement for its template arguments [12]. The C++ compiler checks whether the template arguments satisfy the *concept*'s requirements and provides clear, understandable error messages when the check fails. This takes place at the point of the template's use, not inside its implementation, leading to more understandable code and easier debugging.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 4.9 shows the modifications that can be applied using *concepts* and *templates*. A concept called *is_valid_pin* is defined in lines 1 and 2. It specifies that the unsigned integer values that it receives as template argument is only valid if it is smaller than fifteen. The *GPIO_Init* function is modified to be a templated function, accepting two template arguments: the pin number as an 8 bit unsigned integer, and the mode as a C++ enumeration value. The is stated in line 5, the function requires the pin number to be a valid value by applying the previously defined concept. The function is called in line 11 from inside the main function using template parameters. Such method is extremely advantageous when the pin function is fixed and known at the time of writing the firmware. The if-else decisions and the bitmask calculations are still an unresolved issue inside the function body, which will be examined in further detail in later sections. However, this solution allows the developer to eliminate the runtime parameter value verifications by pushing the checks to compile-time, thus reducing the overall runtime overhead.

```
1  template <std::uint8_t pin>
2  concept is_valid_pin = (pin <= 15);
3
4  template <std::uint8_t pin, GPIO_Mode mode>
5  requires (is_valid_pin<pin>)
6  void GPIO_Init() {
7      /* ... */
8  }
9
10 int main (void) {
11     GPIO_Init<GPIO_PIN_6, GPIO_Mode::output>();
12     /* ... */
13 }
```

LISTING 4.9: Modified version of the GPIO_Init function using templates and concepts.

## 4.4 Decisions

An important keyword, introduced in C++11, is **constexpr**. The name stands for "constant expression" and the keyword can be used to create compile-time constant variables. Many believe that the "const" keyword creates a compile-time constant, but this is a misconception. While it might be true in some cases, in general, the value of a "const" may be calculated at run time [13].

The equivalent of a *constexpr* variable would be the *#define* preprocessor directive in C. In regard to performance, using a *constexpr* variable instead of a defined constant does not make a difference. However, *constexpr* is far more robust than *#define* when it comes to compile-time constants. Most importantly, it is type safe and it behaves just like any other variable. The *#define* preprocessor directive is a simple string substitution macro which is performed before the code is parsed by the compiler. Therefore, the macro does not respect scope or type and can lead to unexpected results, especially when the same name is used in different contexts or with different types. On the other hand, *constexpr* variables obey the C++ scope and type rules. They exist in a given scope and cannot be accessed outside of that scope. Another

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

aspect is the debuggability. Code full of preprocessor macros is generally harder to debug because they are expanded by the preprocessor before the compilation step. With *constexpr*, as it is part of the compilation process, the compiler can provide more meaningful error messages.

Listing 4.9 shows how the values defined by the preprocessor directive *#define* can be replaced by type-safe *constexpr* alternatives.

```
/* ... */
constexpr std::uint8_t GPIO_PIN_6 = 0x06;
/* ... */
constexpr std::uint32_t MODER_MASK = 0x3UL;
constexpr std::uint32_t OTYPER_MASK = 0x1UL;
constexpr std::uint32_t OSPEEDR_MASK = 0x3UL;
/* ... */
```

LISTING 4.10: C++ constexpr variables instead of preprocessor defines.

In this section, however, another feature of the *constexpr* specifier is used to optimize code performance: **if constexpr**. Using compile-time *if* statements, decisions can be made at compile-time rather than at runtime [14, p. 148]. Although the keyword was introduced in C++11, its combination with the *if* statement was only included in the C++17 standard [15].

Listing 4.11 shows how the concept of compile-time if statement can be used to further optimize the GPIO initializing function. In line 7, the runtime if statement from Listing 4.4 is replaced by a compile-time if statement. As will be shown in Listing 4.13, this code eliminates unnecessary runtime branching in the compiled firmware, since the compiler excludes the inactive branch from the binary altogether.

```
template <std::uint8_t pin, GPIO_Mode mode>
requires (is_valid_pin<pin>)
void GPIO_Init() {
    uint32_t temp;

    /* configure the GPIO based on the settings */
    if constexpr (mode == GPIO_Mode::output) {
        /* ... */
    }

    /* ... */
}
```

LISTING 4.11: C++ constexpr if statement converts runtime branching to compile-time decision.

## 4.5 Calculations

The next step in the optimization of the initial C code is the elimination of the runtime bitmask calculations. This can be achieved by using immediate functions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

designated with the **consteval** keyword to force the compiler to execute calculations at compile-time.

*constexpr* and *consteval* are both keywords in C++ that denote compile-time evaluation. While they serve similar purposes, they differ in the strictness of their compile-time evaluation requirements, with *consteval* being the more stringent of the two.

Introduced in C++11 and expanded in later standards, *constexpr* indicates that a function, variable, or object could potentially be computed at compile-time. *constexpr* functions can be called with arguments that are not known until runtime, which means that they can also be executed at runtime [13]. This feature allows for a mix of compile-time and runtime computation, providing flexibility to developers.

In contrast, *consteval*, introduced in C++20, guarantees that a function will always be evaluated at compile-time [13]. In other words, a *consteval* function cannot be invoked at runtime. If it cannot be evaluated at compile-time, because the values of its parameters are not known at compile-time, the compilation process will terminate with error.

Using this modern C++ feature, it is possible to eliminate the need for runtime bitmask calculation in the firmware. Listing 4.12 shows how *consteval* functions can be used for this optimization. Between lines 1 and 3, the immediate function *ospeedr_pin_mask* is defined. It takes the pin number as parameter and calculates the bitmask using the constant *OSPEEDR_MASK* value. Similarly, the function *moder_pin_mask_out* calculates the bitmask for the output mode register. Other functions, such as *ospeedr_pin_mask_low_freq* and *otyper_pin_mask* are defined similarly, but are not included in the presented code snippet. The functions are called in lines 17, 18, 22, 23, 28 and 29 just like regular functions. As long as all of the values are known at compile-time, this code compiles and eliminates the need for runtime bitmask calculations.

```
1  consteval std::uint32_t ospeedr_pin_mask(std::uint8_t pin) {
2      return OSPEEDR_MASK << (static_cast<std::uint32_t>(pin) * 2u);
3  }
4  /* ... */
5  consteval std::uint32_t moder_pin_mask_out(std::uint8_t pin) {
6      return GPIO_OUTPUT_MODE << (static_cast<std::uint32_t>(pin) * 2u);
7  }
8
9  template <std::uint8_t pin, GPIO_Mode mode>
10 requires (is_valid_pin<pin>)
11 void GPIO_Init() {
12     uint32_t temp;
13
14     /* configure the GPIO based on the settings */
15     if constexpr (mode == GPIO_Mode::output) {
16         temp = OSPEEDR;
17         temp &= ~ospeedr_pin_mask(pin);
18         temp |= ospeedr_pin_mask_low_freq(pin);
19         OSPEEDR = temp;
20
21         temp = OTYPER;
22         temp &= ~otyper_pin_mask(pin);
23         temp |= otyper_pin_mask(pin);
24         OTYPER = temp;
25     }
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
26
27      temp = MODER;
28      temp &= ~moder_pin_mask(pin);
29      temp |= moder_pin_mask_out(pin);
30      MODER = temp;
31
32      /* ... */
33  }
```

LISTING 4.12: C++ immediate functions used for bitmask calculation.

Since *consteval* functions are executed at compile time, the binary is expected to use precalculated immediate values. Listing 4.13 shows the main function part of the output of the *objdump* utility. By analyzing the assembly code, some drastic optimizations can be observed compared to the C implementation. The first thing that might occur is the lack of the call to the *GPIO_Init* function. In the C version, as shown in Listing 4.6, all the *main* function does is that it calls the *GPIO_Init* function with the given parameters. However, in the optimized C++ version, the whole function call is eliminated by the optimizer during compilation. It can also be observed, in contrast with the C implementation, that the bitmasks were calculated at compile-time, as expected, and are stored in memory as immediate values. These bitmask values are used to clear and set specific bits in the registers. Since the values of the template parameters are given and verified at compile-time, these steps are eliminated from the binary as well. Furthermore, the previously observed runtime branching based on the GPIO mode input parameter is absent from the assembly as well, because it was replaced with a compile-time if-else block in the source code. As a result, the whole function with parameter value comparisons, branchings and bitmask calculations was reduced to plain register operations in the *main* function. Since these optimization steps are of upmost importance in the context of the thesis, an explanation is provided for every line of Listing 4.13:

- **Line 2**: The literal value 16 (0x10 in hexadecimal) is moved into register *r0*.
- **Line 3**: Loads a value from a memory address computed by adding 28 to the program counter (*pc*) into register *r3*. This value is the address 0x48000008 as indicated at 0x80001bc in line 18.
- **Line 4**: Similarly, loads a value from an address offset by 28 from the *pc* into register *r1*. This value is 0xfffffcff as indicated at 0x80001c0 in line 19.
- **Line 5 and 6**: Load the value from the address in *r3* into *r2*, and then perform a bitwise AND operation with the value in *r1*, updating *r2*.
- **Line 7**: Store the result from the previous operation back into the memory location pointed to by *r3*.
- **Line 8**: Loads another value from memory (address offset by 24 from the *pc*) into *r3*. This address is 0x48000004 as indicated at 0x80001c4 in line 20.
- **Line 9 and 10**: Load the value from the address in *r3* into *r2*, then perform a bitwise OR operation with the value in *r0*, updating *r2*.
- **Line 11**: Store the result from the previous operation back into the memory location pointed to by *r3*.
- **Line 12 and 13**: Move the literal value 144 (0x90) into *r2* and then left shift it by 23 bits. This results in the address 0x48000000.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- **Line 14 and 15**: Load the value from the address in *r2* into *r3*, and then perform a bitwise AND operation with the value in *r1*, updating *r3*.
- **Line 16**: Store the result from the previous operation back into the memory location pointed to by *r2*.
- **Line 17**: An infinite loop is created by branching to the same address, effectively keeping the program executing at this point indefinitely.

```
0800019c <main>:
 800019c:       2010            movs    r0, #16
 800019e:       4b07            ldr     r3, [pc, #28]   ; (80001bc <
    main+0x20>)
 80001a0:       4907            ldr     r1, [pc, #28]   ; (80001c0 <
    main+0x24>)
 80001a2:       681a            ldr     r2, [r3, #0]
 80001a4:       400a            ands    r2, r1
 80001a6:       601a            str     r2, [r3, #0]
 80001a8:       4b06            ldr     r3, [pc, #24]   ; (80001c4 <
    main+0x28>)
 80001aa:       681a            ldr     r2, [r3, #0]
 80001ac:       4302            orrs    r2, r0
 80001ae:       601a            str     r2, [r3, #0]
 80001b0:       2290            movs    r2, #144        ; 0x90
 80001b2:       05d2            lsls    r2, r2, #23
 80001b4:       6813            ldr     r3, [r2, #0]
 80001b6:       400b            ands    r3, r1
 80001b8:       6013            str     r3, [r2, #0]
 80001ba:       e7fe            b.n     80001ba <main+0x1e>
 80001bc:       48000008        .word   0x48000008
 80001c0:       fffffcff        .word   0xfffffcff
 80001c4:       48000004        .word   0x48000004
```

LISTING 4.13: Assembly representation of the code from Listing 4.12.

As shown in Figure 4.2, the modern C++ version eliminated more than 60% of the text size increase compared to the C version. This results in a shorter execution time, although the relationship is not proportional, because the C binary includes sections that might not get executed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

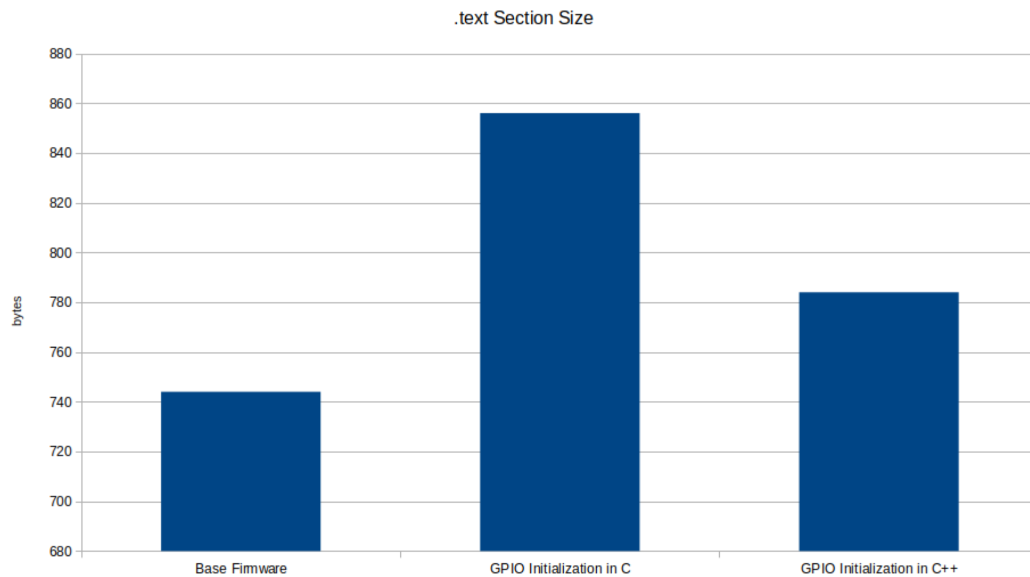



FIGURE 4.2: .text section size of GPIO initialization in C and C++ compared to the base firmware .text section size.

For demonstrating that these optimization steps are not only viable in initialization code, Listing 4.14 shows the conventional C implementation of the GPIO set-reset function. The code includes the runtime parameter value validity checks in lines 5 and 6, as well as the branching based on the given pin's state in line 8.

```
void GPIO_WritePin(uint16_t GPIO_Pin, GPIO_PinState PinState) {
    uint32_t temp;

    /* Check the parameters */
    assert_param(IS_GPIO_PIN(GPIO_Pin));
    assert_param(IS_GPIO_PIN_ACTION(PinState));

    if (PinState != GPIO_PIN_RESET)
    {
        BSRR = (uint32_t)GPIO_Pin;
    }
    else
    {
        BRR = (uint32_t)GPIO_Pin;
    }
}

int main (void) {
    /* ... */
    GPIO_WritePin(GPIO_PIN_10, GPIO_PIN_SET);
    /* ... */
}
```

LISTING 4.14: GPIO set-reset function implemented in C.

As demonstrated, in a conventional C firmware, every component initialization and even every blinking of an LED, achieved by calling the *GPIO_WritePin* function, adds mostly unnecessary additional runtime overhead. The functions first need to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

check the validity of the received arguments, since they are of integral types with arbitrary values. Then, in the case of the LED blinking, based in whether the pin is intended to be set or reset, the relevant bit in the BSRR (Bit Set-Reset Register) or in the BRR (Bit Reset Register) is set. This means that every time an LED is blinked, unnecessary runtime checks and branching is introduced.

I acknowledge that in some cases such overhead might be unavoidable and even necessary. However, the problem is that C does not offer a flexible solution for eliminating such overhead even if desired. C++, on the other hand, offers robust and flexible solutions to achieve the same behavior as conventional C would, while eliminating unnecessary runtime and binary size overhead. Concurrently, it helps to improve readability and maintainability.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Chapter 5

# Overhead of Abstractions

It was confirmed in the previous chapter that modern C++ can be used to eliminate unnecessary runtime overhead of a conventional C codebase. However, the power of C++ does more than simply allow us to optimize C code. One of the most important differences between C and C++ is that while the former is a procedural language, the latter is an object-oriented programming language. Object-oriented programming languages use classes to encapsulate data and methods for manipulating the data into a single entity [16]. Using these languages, developers can add layers of abstractions in order to create simple models of more complex systems.

This chapter will explore the following object-oriented programming paradigms and assess their impact on performance and binary size:

- Encapsulation
- Inheritance
- Polymorphism

## 5.1 Encapsulation

Encapsulation is one of the fundamental principles of object-oriented programming. The principle of encapsulation emphasizes data security and integrity, offering a way to bundle data, and the methods that operate on that data, within a single unit: the object [17]. At its core, encapsulation is about hiding the internal details of how an object operates and exposing only what is necessary to interact with the object [16]. This is usually accomplished through the use of classes, where data attributes (also called member variables) and methods (functions associated with an object) are defined. The class acts as a blueprint for creating individual objects, each with its own specific state and behavior.

In the context of encapsulation, the data of a class is usually made private, which means it can only be directly accessed from within the class itself. To access or modify this data from outside the class, public methods are provided, often referred to as 'getters' and 'setters'. This level of control allows the class to maintain a consistent and correct state, as the internal data cannot be changed arbitrarily.

Encapsulation brings several benefits to the software development process:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- **Increased data security**: By hiding the data attributes and exposing only required features, encapsulation helps maintain the integrity of the data by preventing unauthorized access and accidental modification.
- **Improved maintainability**: Encapsulated code is more modular, meaning that changes in one part of the system are less likely to affect others. This modularity makes the code easier to understand, modify, and debug.
- **Enhanced flexibility**: Encapsulated objects can be developed to be swapped out or updated without impacting the rest of the system, allowing for greater flexibility and scalability in evolving software systems.

Overall, encapsulation aids in creating a clear separation between an object's interface (how it communicates with the outside world) and its implementation (how it achieves its functionality), thereby enabling cleaner, more robust, and more secure programming. In C++, when encapsulating data and functions into a class, it does not come with runtime overhead. What does affect performance, is how these constructs are used. In this section, an examination of an example with surprising results will be conducted. The use of encapsulation to add a layer of abstraction over the *GPIO Mode Register* will be explored, followed by an analysis of the compiled binary.

Listing 5.1 contains the declaration of the scoped enumerations that represent the GPIO modes (*gpio_modes* in line 1) and the GPIO pins (*pin_numbers* in line 7). The enumerations have specific values that represent the bitmask for a given mode and the pin number for shifting the bitmasks.

```
1  enum class gpio_modes : std::uint32_t {
2      input  = 0b00,
3      output = 0b01,
4      analog = 0b10
5  };
6
7  enum class pin_numbers : std::uint32_t {
8      pin_0 = 0,
9      pin_1 = 1,
10     pin_2 = 2,
11     pin_3 = 3,
12     pin_4 = 4,
13     pin_5 = 5,
14     pin_6 = 6,
15     pin_7 = 7
16 };
```

LISTING 5.1: GPIO-related scoped enumerations.

Listing 5.2 shows the example implementation of a generic register class. As expected, a memory-mapped register has an address in memory, which is a private member of the class, as shown in line 3, because it should not be accessible from outside of the class. Furthermore, the register class offers a couple of member functions for interacting with the underlying hardware register. Such function is the *set* (lines 7..9), which allows for the setting of the entire register, the *get* (lines 11..13), which returns the value contained in the register, and another *set* function (lines 15..19), which makes it possible to set only certain bits in the register.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
class CRegister {
private:
    const std::uint32_t m_address;
public:
    CRegister (std::uint32_t address) : m_address(address) { }

    void set (std::uint32_t val) const {
        *(reinterpret_cast<volatile std::uint32_t *>(this->m_address))
            = val;
    }

    std::uint32_t get () const {
        return *(reinterpret_cast<volatile std::uint32_t *>(this->
            m_address));
    }

    void set (std::uint32_t value, std::uint32_t bitmask) const {
        std::uint32_t tmp = this->get();
        tmp &= ~bitmask;
        tmp |= value;
        this->set(tmp);
    }
};
```

LISTING 5.2: Example implementation of generic register class.

Listing 5.3 presents a sample implementation of the *Mode Register*, which is used to configure the mode for each GPIO pin. This register resides at the fixed memory address 0x48000000, which is represented by the constant value *ModeRegisterAddress*. It is worth noting that the class uses static member variables and functions. While this might be a subjective opinion, using static members often leads to clearer code, since it is not necessary to create an object to use the class's member functions. And since a hardware register at a specific memory location is a singleton by nature, it makes sense to use the static keyword. The class has private *consteval* member functions that allow for the computation of the necessary bitmasks at compile-time, as shown in lines 5 through 13. Such a mechanism is especially useful for a register class in an embedded project, since every register has a register-specific bit pattern that must be accounted for using bitmasks. The register offers a public member function in lines 15 through 18, which makes interaction with the register simple and clear. In a typical project, this register would likely be encapsulated in a higher-level abstractions, such as a *CGpio* class, which could be wrapped within even higher-level classes. However, for this discussion, the focus will be on a simplified example to facilitate a more detailed analysis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
class CModeRegister {
private:
    static inline const CRegister m_register { ModeRegisterAddress };

    template <pin_numbers pin, gpio_modes mode>
    static consteval std::uint32_t calculate_value () {
        return static_cast<std::uint32_t>(mode) << static_cast<std::
            uint32_t>(pin) * 2;
    }

    template <pin_numbers pin, gpio_modes mode>
    static consteval std::uint32_t calculate_bitmask () {
        return static_cast<std::uint32_t>(0b11) << static_cast<std::
            uint32_t>(pin) * 2;
    }
public:
    template <pin_numbers pin, gpio_modes mode>
    static inline void set_mode () {
        m_register.set(calculate_value<pin, mode>(), calculate_bitmask<
            pin, mode>());
    }
};

int main (void) {
    CModeRegister::set_mode<pin_numbers::pin_6, gpio_modes::output>();

    while (true) { /* ... */ }
}
```

LISTING 5.3: Example implementation of a specific GPIO register class using encapsulation with static members.

Listing 5.4 shows the assembly representation of the *main* function shown in Listing 5.3. The most surprising part of this code snippet is probably how the address of the GPIO Mode Register, 0x48000000, is handled. To better understand this, a summary of the code in Listing 5.4 is provided below:

- **Line 2**: Loads a value from a memory address offset by 16 from the program counter (*pc*) into register *r3*. This value is the address 0x20000088 as indicated at 0x80001b0 in line 12.
- **Line 3**: Similarly, loads a value from an address offset by 20 from the *pc* into register *r2*. This value is 0xffffcfff as indicated at 0x80001b4 in line 13.
- **Line 4**: Loads the value from the address in *r3* into *r1* (value at memory location 0x20000088 is loaded into *r1*). 0x20000088 is a memory address in the SRAM section of this microcontroller [18].
- **Line 5**: Loads the value from the address in *r1* into *r3* (value at memory location represented by the value at memory location 0x20000088 is loaded into *r3*). It will be demonstrated later, but for now, it can be assumed at this point that *r1* holds the address of the GPIO Mode Register, namely 0x48000000.
- **Line 6**: Performs a bitwise AND operation between the values in *r2* and *r3*, updating *r2*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

- **Line 7 and 8**: Moves the value 128 (0x80 in hexadecimal) into *r3*, and then left shifts it by 5 bits. This results in the value 4096 (0x1000 in hexadecimal).
- **Line 9**: Performs a bitwise OR operation between the values in *r3* and *r2*, updating *r3*.
- **Line 10**: Stores the result from the previous operation back into the memory location pointed to by *r1*.
- **Line 11**: An infinite loop is created by branching to the same address, effectively keeping the program executing at this point indefinitely.

```
0800019c <main>:
 800019c:       4b04            ldr     r3, [pc, #16]   ; (80001b0 <
    main+0x14>)
 800019e:       4a05            ldr     r2, [pc, #20]   ; (80001b4 <
    main+0x18>)
 80001a0:       6819            ldr     r1, [r3, #0]
 80001a2:       680b            ldr     r3, [r1, #0]
 80001a4:       401a            ands    r2, r3
 80001a6:       2380            movs    r3, #128        ; 0x80
 80001a8:       015b            lsls    r3, r3, #5
 80001aa:       4313            orrs    r3, r2
 80001ac:       600b            str     r3, [r1, #0]
 80001ae:       e7fe            b.n     80001ae <main+0x12>
 80001b0:       20000088        .word   0x20000088
 80001b4:       ffffcfff        .word   0xffffcfff
```

LISTING 5.4: Assembly representation of the code from Listing 5.3.

The indirection examined above in Listing 5.4 is not the only thing that captured my attention. An estimation can be made of the expected increase in text section size based on a given section in the disassembled ELF file. In the Thumb instruction set, most instructions occupy 2 bytes in memory, while data words take up 4 bytes. By counting the number of instruction lines and the number of added data lines, it can be estimated that the text section should increase by around 26 bytes. However, this does not align with the observation. When compared to the text size of the base firmware (744 bytes), there is an observed increase of 56 bytes. This suggests that there is another location in the binary where additional code has been introduced.

Listing 5.5 shows the code section that was additionally introduced by the compiler. This function is responsible for constructing and destructing the global objects. Since the *CModeRegister* has a static member variable, that member variable must exist for the duration of the whole application, meaning that the lifetime of the object extends beyond the end of the *main* function. The compiler cannot optimize this away, it has to make sure that global objects are created before the *main* function starts and are destroyed after the *main* function returns.

In the previous discussion of Listing 5.4, it was noted that the register address, namely 0x48000000, resides in the microcontroller's SRAM section. It is not immediately evident, but the assembly in Listing 5.5 writes this value to the memory location 0x20000088. In line 8, the immediate value 0x90 is loaded into the register, *r2*. The subsequent line then loads the value 0x20000088 into *r3*. In line 10, the value in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

*r2* is subjected to a left shift by 23, producing the expected value of 0x48000000. The following line then stores the *r2* value at memory location 0x20000088. This is the memory location from where the *main* function retrieves the GPIO Mode Register address.

```
080001b4 <_GLOBAL__sub_I_main>:
 80001b4:       2201            movs    r2, #1
 80001b6:       4b05            ldr     r3, [pc, #20]   ; (80001cc <
    _GLOBAL__sub_I_main+0x18>)
 80001b8:       6819            ldr     r1, [r3, #0]
 80001ba:       4211            tst     r1, r2
 80001bc:       d104            bne.n   80001c8 <_GLOBAL__sub_I_main+
    0x14>
 80001be:       601a            str     r2, [r3, #0]
 80001c0:       2290            movs    r2, #144        ; 0x90
 80001c2:       4b03            ldr     r3, [pc, #12]   ; (80001d0 <
    _GLOBAL__sub_I_main+0x1c>)
 80001c4:       05d2            lsls    r2, r2, #23
 80001c6:       601a            str     r2, [r3, #0]
 80001c8:       4770            bx      lr
 80001ca:       46c0            nop                     ; (mov r8, r8)
 80001cc:       20000084        .word   0x20000084
 80001d0:       20000088        .word   0x20000088
```

LISTING 5.5: Function responsible for constructing and destructing global objects.

The example code discussed in Listing 5.5 indicates that if not managed properly, encapsulation could lead to an increase in code size. However, it is important to note that this extra code would only execute before the *main* function is called and after it completes. As such, the core performance of the application wouldn't be impacted. The only changes would occur in the startup and shutdown times of the application.

However, it is possible to eliminate this overhead by getting rid of the runtime static member variable using templates. Listing 5.6 presents the modified version of the code from Listing 5.3 using templates. As shown in line 1, the generic register class receives the memory address as template parameter, and is instantiated in line 22 with the memory address 0x48000000 of the mode register.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
template<std::uint32_t address>
class CRegister {
public:
    void set (std::uint32_t val) const {
        *(reinterpret_cast<volatile std::uint32_t *>(address)) = val;
    }

    std::uint32_t get () const {
        return *(reinterpret_cast<volatile std::uint32_t *>(address));
    }

    void set (std::uint32_t value, std::uint32_t bitmask) const {
        std::uint32_t tmp = this->get();
        tmp &= ~bitmask;
        tmp |= value;
        this->set(tmp);
    }
};

class CModeRegister {
private:
    static inline const CRegister<0x48000000> m_register { };

    template <pin_numbers pin, gpio_modes mode>
    static consteval std::uint32_t calculate_value () {
        return static_cast<std::uint32_t>(mode) << static_cast<std::
            uint32_t>(pin) * 2;
    }

    template <pin_numbers pin, gpio_modes mode>
    static consteval std::uint32_t calculate_bitmask () {
        return static_cast<std::uint32_t>(0b11) << static_cast<std::
            uint32_t>(pin) * 2;
    }
public:
    template <pin_numbers pin, gpio_modes mode>
    static inline void set_mode () {
        m_register.set(calculate_value<pin, mode>(), calculate_bitmask<
            pin, mode>());
    }
};

int main (void) {
    CModeRegister::set_mode<pin_numbers::pin_6, gpio_modes::output>();

    while (true) { /* ... */ }
}
```

LISTING 5.6: Address of register as template parameter leads to optimized runtime performance.

With the modified code shown in Listing 5.6, the text size is reduced from 800 bytes to 764 bytes, which is only a 20 bytes increase compared to the base firmware's text size. The function, responsible for the construction and destruction of global objects (Listing 5.5), is eliminated from the binary, and the *main* function, shown in Listing 5.7, becomes much more readable than it was in Listing 5.3. The address 0x48000000 is not stored in the RAM at runtime anymore, but constructed directly in the *main* function from immediate values and bitwise operations with the following steps:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- The instruction at 0x800019c (movs r1, #144; 0x90) moves the value 0x90 into register r1.
- The instruction at 0x800019e (lsls r1, r1, #23) performs a left shift logical operation on the value in r1 by 23 bits. This means 0x90 (which is 10010000 in binary) is shifted 23 bits to the left, creating the number 0x48000000.

As a side-note, it is interesting to see how precise the optimization is. The value 0x48000000 could be stored as a read-only data in the binary at a specific address and loaded into a register if needed. However, since it is loaded only once in the program, it is more efficient to construct the value at runtime than storing it as data and load from memory. Constructing the value at runtime takes two instructions, which take up 4 bytes altogether in the binary, because an instruction is stored on 2 bytes in memory. On the other hand, storing the value as a *word* in memory takes up 4 bytes, and the additional loading instruction takes 2 bytes from the memory space, since it is a single instruction, adding up to 6 bytes altogether. This is only true if the value needs to be loaded into a register once in the program, and if the value can be constructed from two operations.

```
1  0800019c <main>:
2   800019c:       2190            movs    r1, #144        ; 0x90
3   800019e:       05c9            lsls    r1, r1, #23
4   80001a0:       680b            ldr     r3, [r1, #0]
5   80001a2:       4a03            ldr     r2, [pc, #12]   ; (80001b0 <
       main+0x14>)
6   80001a4:       401a            ands    r2, r3
7   80001a6:       2380            movs    r3, #128        ; 0x80
8   80001a8:       015b            lsls    r3, r3, #5
9   80001aa:       4313            orrs    r3, r2
10  80001ac:       600b            str     r3, [r1, #0]
11  80001ae:       e7fe            b.n     80001ae <main+0x12>
12  80001b0:       ffffcfff        .word   0xffffcfff
```

LISTING 5.7: Assembly representation of the code from Listing 5.6

Based on the examples presented in this section, I can conclude that it is possible to use encapsulation without affecting the runtime performance of the firmware application. This is just as well applicable for software applications running on more powerful computers. This gives C++ a huge advantage over C, because it allows the grouping and organization of the code with basically no runtime or binary size penalty. This versatile feature of C++ makes writing efficient, readable and maintainable code possible.

## 5.2 Inheritance

Inheritance is a fundamental concept in object-oriented programming that enables an object or class to take on the properties and methods of another class. This allows for code reuse, reduced redundancy and increased maintainability.

The class that is being inherited from is known as the *parent*, *base*, or *superclass*. The class that inherits from the base class is known as the *child*, *derived*, or *subclass*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The derived class can acquire the attributes and behavior of the base class, while also having the ability to introduce specific attributes and behaviors of its own [13].

It is worth noting that while inheritance provides significant advantages in the abstraction and organization of code, it should be used judiciously. Overuse or inappropriate use of inheritance can lead to overly complex code and potential issues with data integrity.

To assess the runtime overhead of this OOP feature, a reworked version of the code from Listings 5.6 and 5.3 was analyzed. Instead of having an instance of the *CRegister* class as member variable, the *CModeRegister* inherits from it, as shown in line 1 of Listing 5.8. The object is instantiated in line 21 of the *main* function, and the member function *set_mode* is called in the subsequent line.

```
1  class CModeRegister : public CRegister<0x48000000> {
2  private:
3
4      template <pin_numbers pin, gpio_modes mode>
5      static consteval std::uint32_t calculate_value () {
6          return static_cast<std::uint32_t>(mode) << static_cast<std::
               uint32_t>(pin) * 2;
7      }
8
9      template <pin_numbers pin, gpio_modes mode>
10     static consteval std::uint32_t calculate_bitmask () {
11         return static_cast<std::uint32_t>(0b11) << static_cast<std::
               uint32_t>(pin) * 2;
12     }
13 public:
14     template <pin_numbers pin, gpio_modes mode>
15     inline void set_mode () {
16         set(calculate_value<pin, mode>(), calculate_bitmask<pin, mode
               >());
17     }
18 };
19
20 int main (void) {
21     CModeRegister mode_register {};
22     mode_register.set_mode<pin_numbers::pin_6, gpio_modes::output>();
23
24     while (true) { /* ... */ }
25 }
```

LISTING 5.8: Example code for inheritance.

The text size of the compiled code from Listing 5.8 remained the same compared to the code from Listing 5.6. The disassembled ELF files are identical as well. As such, I can conclude that adding inheritance to the project does not result in additional runtime overhead for objects typical on embedded systems. However, it can be generally concluded that inheritance in and on itself, without the use of virtual functions, does not result in penalty regarding performance and binary size. This feature of C++ offers another great advantage over C implementations, namely that it supports code reusability to a high extent.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## 5.3 Polymorphism

Polymorphism, deriving from Greek terms *poly* meaning many and *morph* meaning forms, is a fundamental principle of object-oriented programming and a core feature of C++. In essence, polymorphism provides a way to structure code so that one interface can be used to manipulate objects of different types, leading to greater code reusability and cleaner design [16].

In C++, polymorphism is primarily achieved through inheritance and virtual functions. With inheritance, derived classes inherit the attributes and methods of the base class, but also have the ability to extend or override these elements. This forms the basis for the two main types of polymorphism in C++: static (or compile-time) and dynamic (or runtime) polymorphism.

Static polymorphism is realized at compile-time through function overloading and templates. Function overloading allows functions with the same name but different parameters to be defined, with the compiler determining which function to call based on the arguments. Templates, another form of static polymorphism, enable the creation of functions or classes that operate on different data types without having to rewrite the code for each type [12].

Dynamic polymorphism, on the other hand, is facilitated by virtual functions and abstract base classes. A virtual function defined in a base class can be overridden in derived classes, and the decision of which function to call is made at runtime based on the type of the object pointed to, rather than the type of pointer. This enables a single interface to be used for general actions on different types of objects, encapsulating variation in behavior in a clean and flexible manner [12][19].

Critically, C++'s support for polymorphism is designed with performance in mind. The language offers fine control over when dynamic dispatch is used, allowing programmers to make informed trade-offs between abstraction and efficiency. With careful design, polymorphic systems in C++ can exhibit both high-level expressiveness and low-level speed.

The following sections will dive into the different types of polymorphism and demonstrate their usefulness and effectiveness as well as their drawbacks and pitfalls.

### 5.3.1 Runtime Polymorphism

Runtime polymorphism leverages the language's ability to resolve a function call to different concrete functions based on the type of object the call is made on during program execution. Under the hood, runtime polymorphism is implemented using a mechanism called the virtual table, or *vtable*. A short overview on how runtime polymorphism works is described in the following three points:

- **Virtual table**: When a class in C++ has at least one virtual function, the compiler generates a *virtual table* for that class. The virtual table is essentially an array where each entry is a function pointer that points to the virtual function

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

applicable to that class. Each object of the class will have a hidden pointer (usually called a *vpointer*) that points to this *vtable*.

- **Inheritance and vtables**: When a derived class inherits from a base class and overrides some of its virtual functions, the compiler creates a new *vtable* for the derived class. This *vtable* contains pointers to the derived class's overridden functions and pointers to any virtual functions not overridden, pointing back to the base class's version.
- **Virtual function call**: When a virtual function is called on an object, the compiler generates code to first fetch the object's *vpointer*, then index into the *vtable* to find the correct function to execute. The function is then invoked using the function pointer.

This mechanism introduces an overhead for the call to the virtual function and for each object that has a *vpointer*, but it provides the powerful ability to call the correct function based on the actual type of the object at runtime, even when handled through a pointer or reference to the base type. The overhead introduced by virtual interfaces in C++ is generally relatively small, but can be significant in certain cases. There are two main sources of this overhead:

- **Memory Overhead**: Each object of a class with virtual functions needs to store a pointer to its virtual method table (*vtable*). This increases the size of each object by the size of a pointer. For classes with many small objects, this can add up. Additionally, each distinct class with virtual functions needs to have its own *vtable* stored somewhere in the binary, which also consumes space.
- **Runtime Overhead**: Calls to virtual functions are slightly slower than calls to non-virtual functions. This is because a virtual function call requires two pointer dereferences (one to get the *vtable* pointer from the object, and another to get the function pointer from the *vtable*), whereas a non-virtual function call is a direct branch to a known location and hence requires no pointer dereferences. Furthermore, because the target of a virtual function call is not known until runtime, it can potentially disrupt branch prediction in the CPU, causing a pipeline stall and further slowing down execution. However, modern CPUs and compilers are quite good at minimizing the impact of this.

In most situations, these overheads are small enough so that they are not of significant concern. However, in performance-critical code, or on very memory-constrained systems, they might be relevant. Developers need to consider whether the benefits of using virtual functions (e.g., more flexible and maintainable code via polymorphism) outweigh these costs in specific situations.

Listing 5.9 presents an example on how polymorphism can be introduced to the firmware. An *IPin* interface is created in lines 1 through 4 which has a single pure virtual method. The derived classes that implement this interface must provide the implementation for the pure virtual function. *CPinA* in lines 6 through 12 and *CPinB* in lines 14 through 20 are two classes that implement the interface. The *HandlePin* in lines 22 through 26 function represents an entity that operates with the interface only and is independent from the underlying implementation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
class IPin {
public:
    virtual std::uint8_t get_pin_number () const = 0;
};

class CPinA : public IPin {
private:
    std::uint8_t m_pin;
public:
    CPinA (std::uint8_t pin) : m_pin(pin) {}
    std::uint8_t get_pin_number () const override { return m_pin + 1; }
};

class CPinB : public IPin {
private:
    std::uint8_t m_pin;
public:
    CPinB (std::uint8_t pin) : m_pin(pin) {}
    std::uint8_t get_pin_number () const override { return m_pin - 1; }
};

std::uint8_t HandlePin (IPin* pin) {
    std::uint8_t pin_number = pin->get_pin_number();
    ++pin_number;
    return pin_number;
}

int main (void) {
    GPIO_Init<GPIO_PIN_6, GPIO_Mode::output>();

    CPinA pin_a { GPIO_PIN_6 };
    std::uint8_t pin_number = HandlePin(&pin_a);
    CPinB pin_b { GPIO_PIN_6 };
    pin_number = HandlePin(&pin_b);

    while (true) { /* ... */}
}
```

LISTING 5.9: Runtime polymorphism implemented in C++.

A relevant part of the disassembled binary of the code in Listing 5.9 is shown in Listing 5.10. The first instruction in line 2 loads the contents of the memory location pointed to by the *r0* register (the first argument, which is the object pointer) into the *r3* register. This effectively loads the *vtable* pointer into *r3*, since the *vtable* pointer is the first element of an object instance with virtual methods. The next line pushes the *link register* and *r4* onto the stack, which is a standard way to save the return location before a branch operation. Line 4 loads the contents of the memory location pointed to by *r3* (the *vtable* pointer) into *r3*. This retrieves the address of the first virtual function from the *vtable*. Then the code branches to the location stored in *r3*, and also stores the next instruction's address (the return address) into the *lr*. This is how a call to a virtual function is achieved using assembly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
080001c0 <_Z9HandlePinP4IPin>:
 80001c0:       6803            ldr     r3, [r0, #0]
 80001c2:       b510            push    {r4, lr}
 80001c4:       681b            ldr     r3, [r3, #0]
 80001c6:       4798            blx     r3
 80001c8:       3001            adds    r0, #1
 80001ca:       b2c0            uxtb    r0, r0
 80001cc:       bd10            pop     {r4, pc}
```

LISTING 5.10: Low-level handling of interface pointer.

However, this is not the only overhead that appears when using polymorphism. The size of the text section increased significantly (by around 6 kilobytes) after introducing the code presented in Listing 5.9. In the disassembled binary, many sections appear as a result of introducing this indirection.

On the one hand, assembly sections like *__gnu_unwind_frame* and *__gnu_unwind_execute* can be found. These are not directly related to polymorphism, but are part of the exception handling mechanism used by the GNU GCC compiler. When an exception is thrown in C++, the program needs to unwind the stack, cleaning up the objects in each stack frame until it finds a handler for the exception [20][21]. This process involves finding and executing the right destructors for objects in the stack frames. In many embedded environments, exception handling is disabled because, as observed above, it can add significant overhead and unpredictability. For GCC, the *-fno-exceptions* flag can be used to disable exceptions, which will prevent these sections from being generated. However, disabling exceptions means that the developer needs to ensure that the code can handle error conditions without them. Developers will also not be able to use any code that relies on exceptions, including part of the C++ Standard Library.

On the other hand, assembly sections like *__class_type_info11__do_upcast* and *__class_type_info* appeared as well. These are symbols introduced by the C++ ABI (Application Binary Interface) and are a part of the C++ runtime support. Such symbols are typically introduced when certain features of C++, such as RTTI (Run-Time Type Information), dynamic casting and exception handling are used [20]. Depending on the use-case and the constraints of the environment, it is recommended to carefully consider which C++ features are used. Some features, like RTTI, can add significant code and data size overhead, and may not be suitable for all environments [22]. Note that some compilers or build systems allow for disabling RTTI, which can reduce code size. However, if such features are required, then the code for supporting them must be present in the binary.

Note that the large increase in the binary size does not necessarily influence the runtime performance as long as the code segments are not executed. The stack unwinding functions will not run until an exception is thrown. However, if memory is a constrained resource on the embedded device, this could result in the firmware not fitting into the device's memory.

The goal of this section was to show that although modern C++ can be used to optimize code if handled correctly, it has its limitations in the embedded world. Indeed, significant overhead can be introduced if developers are not aware of such traps. Developers who develop software for powerful PCs usually do not encounter

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

such constraints and considerations, however, embedded developers must be aware of such limitations. The following section will try to give an alternative method for eliminating such overhead.

### 5.3.2 Static Binding - "Compile-Time Polymorphism"

In computer programming, binding refers to the process by which names are associated with objects in memory. This can happen at different stages in the programming process: during compilation, during link time, or during program execution. When binding occurs during compilation or link time, it if known as static binding (also known as early binding or compile-time binding)[23].

In C++, static binding is the default binding mechanism for most constructs. This includes function calls, variable accesses, and similar operations. When the compiler encounters a function call, for example, it decides at compile-time which function implementation to associate with that call. This is based on the function name, as well as the number and types of the function arguments (this process is known as function overloading resolution) [19].

Static binding in C++ provides several advantages:

- **Performance**: Since the binding is done at compile-time, the associated overhead is handled before the program's execution, thus leading to faster runtime performance.
- **Type Checking**: The types of variables and expressions are checked at compile-time, leading to early detection of type mismatch errors.
- **Readability and Maintainability**: The code is more straightforward to read and maintain since the binding of variables and functions is determined and can be traced at compile-time.

The introduction of the C++ *concepts* in 2020 opened up a field of new opportunities in mimicking the behavior of dynamic polymorphism at compile-time. Clearly, true dynamic polymorphism is not possible using this method, since it is an inherently runtime paradigm. But in many cases, reusable code can be implemented without the need for dynamic polymorphism. This is especially the case for embedded systems, where code written for a specific platform is most likely not compatible with other platforms.

Listing 5.11 shows how the previous example in Listing 5.9 can be modified to eliminate all of the runtime penalties presented before. Using *concepts*, requirements can be defined for types which are checked at compile-time. The *IPin* concept in lines 1 through 4 states that a type only satisfies the requirement if it implements a function called get_pin_number which takes no arguments and returns an *std::uint8_t* type. The classes *CPinA* in lines 6 through 12 and *CPinB* in lines 14 through 20 both fulfill this requirement. The *HandlePin* function in lines 22 through 27 can only handle types that are of type *IPin*, meaning that they implement the said function. The code of this function is reusable on every platform without porting by replacing the required platform-specific pin class, while still being a compile-time paradigm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
template <typename T>
concept IPin = requires (T pin) {
    { pin.get_pin_number() } -> std::same_as<std::uint8_t>;
};

class CPinA {
private:
    std::uint8_t m_pin;
public:
    CPinA (std::uint8_t pin) : m_pin(pin) {}
    std::uint8_t get_pin_number () const { return m_pin + 1; }
};

class CPinB {
private:
    std::uint8_t m_pin;
public:
    CPinB (std::uint8_t pin) : m_pin(pin) {}
    std::uint8_t get_pin_number () const { return m_pin - 1; }
};

template <IPin TPin>
std::uint8_t HandlePin (TPin* pin) {
    std::uint8_t pin_number = pin->get_pin_number();
    ++pin_number;
    return pin_number;
}

int main (void) {
    volatile std::uint8_t pin_number;

    CPinA pin_a { GPIO_PIN_6 };
    pin_number = HandlePin<CPinA>(&pin_a);
    CPinB pin_b { GPIO_PIN_6 };
    pin_number = HandlePin<CPinB>(&pin_b);

    while (true) { /* ... */}
}
```

LISTING 5.11: Implementation of polymorphism-like relationship using static binding.

The size of the text section of the binary reduces back into the expected range, specifically to 756 bytes, which is only a slight increase when compared to the reference 744 byte. Note that in Listing 5.11, the variable *pin_number* is denoted with the *volatile* qualifier. This is necessary because the compiler recognizes that the function calls only change the value of a local variable that is not used afterwards, and hence it has no effect and can be optimized away. If the variable is optimized away entirely by the compiler, only an empty main function and the while loop can be found in the disassembled binary. With the *volatile* keyword, the compiler can be instructed not to eliminate the variable during the compilation process. This ensures proper analysis of the disassembled binary for the specified purpose.

Listing 5.12 shows the *main* function (Listing 5.11) from the disassembled binary. When examining the C++ code as a whole, it would be expected for the value of *pin_number* to take the following values:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- First, the *HandlePin* function is called with *CPinA* as template parameter. The function calls the *get_pin_number* member function on the *pin_a* object, which returns *GPIO_PIN_6 + 1*. Then then pin number is further incremented by one in the *HandlePin* function before returning it. Meaning that the value of *pin_number* first takes the value *GPIO_PIN_6 + 2*.
- Then, the *HandlePin* function is called with *CPinB* as template parameter. The function calls the *get_pin_number* member function on the *pin_b* object, which returns *GPIO_PIN_6 - 1*. Then then pin number is incremented by one in the *HandlePin* function before returning it. Meaning that the value of *pin_number* takes the value *GPIO_PIN_6*.

The defined value of *GPIO_PIN_6* is 4. The assembly in Listing 5.12 reveals that the compiler was able to optimize away the function calls and most of the performed operations. The code snippet performs the following steps:

- First, 8 bytes are allocated on the stack for local data in line 2. This might be a surprising step, because the *pin_number* is an *std::uint8_t*, which should only take up 1 byte in memory. However, this could have many reasons, for example that the microcontroller has specific requirements for alignment and performs better when the stack is 8-byte aligned.
- The value of the current stack pointer is then moved into the *r3* register in line 3. This indicates that the *r3* register is used as a pointer to the local data on the stack.
- The next operation stores the immediate value 6 into the *r2* register in line 4. The *strb* operation stores the value in *r2* (which is 6) to the address pointed to by *r3*, offset by 7 (line 5). This effectively writes the value 6 to the eighth byte of the allocated stack space, and hence the value of the local variable *pin_number* becomes 6, as expected.
- Then 2 is subtracted from the value in *r2* in line 6, resulting in the value 4, which is then stored in the local variable in line 7.

```
0800019c <main>:
 800019c:       b082            sub     sp, #8
 800019e:       466b            mov     r3, sp
 80001a0:       2206            movs r2, #6
 80001a2:       71da            strb r2, [r3, #7]
 80001a4:       3a02            subs r2, #2
 80001a6:       71da            strb r2, [r3, #7]
 80001a8:       e7fe            b.n     80001a8 <main+0xc>
```

LISTING 5.12: Assembly of the code from Listing 5.11.

As mentioned above, the binary size reduced back into the expected range, being only a couple of bytes larger than the base firmware. The sections dealing with exception handling and runtime type information are not present in the binary anymore. This is due to the fact that only static binding happens in the code, which is resolved by the compiler. Runtime error handling is no longer required, because if the type does not satisfy the type requirement specified by the concept, the compiler will generate a meaningful error message. Listing 5.13 shows a modified version of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

*CPinB* class which does not satisfy the constraint, because it returns an *std::uint16_t* instead of the required *std::uint8_t*. The error message is shown in Figure 5.1. The error message states that the constraints are not satisfied, and it even specifies that the problem is that the return type of the *get_pin_number* function does not match the type required by the constraint *IPin*.

```
1 class CPinB {
2 private:
3     std::uint8_t m_pin;
4 public:
5     CPinB (std::uint8_t pin) : m_pin(pin) {}
6     std::uint16_t get_pin_number () const { return m_pin - 1; }
7 };
```

LISTING 5.13: Example of a class that does not satisfy the type constraints.

```
main.cpp: In function 'int main()':
main.cpp:115:41: error: use of function 'uint8_t HandlePin(TPin*) [with TPin = CPinB; uint8_t = unsigned
 char]' with unsatisfied constraints
  115 |     pin_number = HandlePin<CPinB>(&pin_b);
      |                                         ^
main.cpp:101:14: note: declared here
  101 | std::uint8_t HandlePin (TPin* pin) {
      |              ^~~~~~~~~
main.cpp:101:14: note: constraints not satisfied
main.cpp: In instantiation of 'uint8_t HandlePin(TPin*) [with TPin = CPinB; uint8_t = unsigned char]':
main.cpp:115:41:   required from here
main.cpp:80:9:   required for the satisfaction of 'IPin<TPin>' [with TPin = CPinB]
main.cpp:80:16:   in requirements with 'T pin' [with T = CPinB]
main.cpp:81:25: note: 'pin.get_pin_number()' does not satisfy return-type-requirement
   81 |     { pin.get_pin_number() } -> std::same_as<std::uint8_t>;
      |       ~~~~~~~~~~~~~~~~~~~^~
cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail
make: *** [makefile:88: prog] Error 1
```

FIGURE 5.1: Compile-time error message generated for constraint violation.

This feature can be very useful in certain situations. Most importantly, it can replace dynamic polymorphism in hardware-agnostic higher layer constructs. To give an example, imagine a *CLed* class, which should be implemented in a hardware-independent way, so that it can be used on both an ESP32 and an STM32 platform. The easiest solution before *concepts* was runtime polymorphism: An *IGpio* interface was defined which then was implemented for both platforms. The *CLed* class can then operate with the interface. The result is a hardware-independent abstraction of a blinking LED. This, however, adds unnecessary runtime overhead, as it was demonstrated in Chapter 5.3.1.

Since it is clearly known at compile-time whether the underlying platform is an STM32 or an ESP32 microcontroller, by using modern C++ concepts, the compiler can eliminate the additional runtime overhead. The *GPIO concept* and *CLed* class can be used as shown in Listing 5.14. All of the necessary functions are required by the concept and are implemented by a hardware-specific GPIO class. This way, the *CLed* class can be used on any hardware as long as the GPIO implementations satisfy the constraints defined by the *GPIO concept*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
template <typename T>
concept GPIO = requires(T gpio) {
    { gpio.set() } -> std::same_as<void>;
    { gpio.reset() } -> std::same_as<void>;
    { gpio.toggle() } -> std::same_as<void>;
    { gpio.get() } -> std::same_as<bool>;
};

template <GPIO TGpio>
class CLed {
private:
    TGpio* m_gpio;
public:
    constexpr CLed (TGpio* gpio) : m_gpio(gpio) {}
    void turn_on () { m_gpio->set(); }
    void turn_off () { m_gpio->reset(); }
    void toggle () { m_gpio->toggle(); }
};
```

LISTING 5.14: Example usage of concepts in firmware applications.

On the flip side, it is important to note that this feature has some disadvantages when compared to runtime polymorphism. While the virtual interface, shown in Listing 5.15, can only be implemented by classes that define the pure virtual function with the identical signature, the concept, shown in 5.16, only checks for the existence of a *func* function that can accept an *uint32_t* parameter and returns void. For example, this will allow for a function that takes an *uint8_t* parameter, because an *uint8_t* can indeed accept a *uint32_t* due to *integral promotion*. Integral promotion is a concept in C++ and several other programming languages that determines how integer types (i.e., integral types) are handled in various situations. In many cases, integral types smaller than *int* or *unsigned int* are converted or "promoted" to *int* or *unsigned int* before an operation is performed. This conversion process is done automatically by the compiler in certain circumstances, usually to maximize the efficiency of operations and to ensure consistent behavior across different hardware architectures.

```
class IA {
public:
    virtual ~IA() {}
    virtual void func (std::uint32_t val) = 0;
};
```

LISTING 5.15: Runtime polymorphism.

```
template <typename T>
concept IA = requires (T obj, uint32_t val) {
    { obj.func(val) } -> std::same_as<void>;
};
```

LISTING 5.16: Polymorphism using static binding.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

I believe that this new feature of C++ solves a long-standing issue in the embedded world: writing reusable code in a hardware agnostic way without paying for the penalties at runtime and without using complicated configuration steps that use preprocessor directives to activate and deactivate code segments. This can be a huge advantage of C++ in contrast to C, which does not give a solution to this problem. The way C HAL libraries tackle the issue of reusability is by designing the HAL interfaces of different devices of the same manufacturer similarly, so that the code written by developers can be easily ported and reused across different families of their own devices. However, the reusability between platforms of different manufacturers is still unresolved. This new C++ feature has the potential to solve this issue in an elegant and efficient way.

### 5.3.3 Method Overloading

Method overloading, a significant feature of many object-oriented programming languages, is a form of compile-time polymorphism. It allows for the existence of multiple methods within a class that share the same name, but possess different parameters [24]. This feature enhances code readability and reusability by enabling the use of a single method name with different quantities or types of arguments.

At the heart of method overloading lies the concept of the method signature, which combines the method name with the parameter list. Each variant of an overloaded method must have a unique method signature. It is crucial to note that the method's return type is not considered part of this signature. As such, having two methods with identical names and parameter lists, but different return types, will typically trigger a compile error [25].

During compilation, the compiler discerns which method to invoke based on the number, types, and order of the arguments provided. If it fails to identify an appropriate method, it typically results in an error.

By diminishing the need for differently named methods performing similar actions on varying types or quantities of arguments, method overloading enhances code readability. It also improves code reusability by presenting a single interface for executing the same action across diverse types or quantities of arguments.

This feature is not necessarily considered as a type of abstraction, however, for the sake of completeness, the runtime overhead of this OOP feature is briefly mentioned as well. Even though it was not explicitly stated at that point, an example code containing function overloading was already examined. In Listing 5.6, the *CRegister* class has two *set* member functions with the same name but different signatures. One of them only takes a single parameter, the value to be written into the register. This function overwrites the whole register. On the other hand, the other function takes not only a value parameter, but also a bitmask parameter. The bitmask parameter is used to select the bits in the register that can be modified by the set function. The two functions implement similar functionalities, therefore it makes sense to use overloading. As determined in Chapter 5.1, there was no runtime overhead added to the firmware. The compiler resolves the overloaded methods at compile-time with no effect on runtime. This feature is not supported by C, function names need to unique. Even though defining many function with different names is the same

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

regarding performance as defining just as many functions with the same name but different signatures, code readability can be greatly increased through method overloading.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

# Chapter 6

# Results and Practical Evaluation

The goal of this section is to give a summary about the overhead of the previously discussed C++ language features and programming techniques.

## 6.1 Measurements of Runtime Overhead Elimination

In Chapter 4, a detailed explanation and analysis was given on how C++ can help eliminating the unnecessary runtime overhead inherent to software written in C. Analysis of the disassembled binary files showed that the software written in C++ was more efficient at runtime than its C counterpart. This is due to the following C++ language features and their consequences:

- **templates and concepts** (Section 4.3) provide a mechanism to enforce compile-time evaluation of function parameters. This is especially useful in the area of embedded systems, where the functionality of a device is known at manufacturing and does not change over the device's lifetime. Compile-time parameter passing allows for the elimination of runtime function calls, which are inherently expensive operations.
- **Type system constraints** (Section 4.2) of C++ eliminate the need for function parameter validity checking at runtime. Developers can pass dedicated enumerated values instead of plain integral types, which not only results in a more efficient runtime execution but helps with code readability as well.
- **constexpr if statements** (Section 4.4) can be used to push the branching of the execution from runtime to compile-time. As such, inactive code sections are excluded from the binary, and therefore the result is a faster and more memory-efficient binary.
- **consteval function calls** (Section 4.5) allow for functions to be executed at compile-time. This is highly usable in embedded systems applications, where register values are set using bitmask calculations.

Although the thesis mostly focuses on embedded systems applications, the discussed advantages of C++ apply the same way to applications running on well-known platforms, such as Linux or Windows.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

### 6.1.1 Measurement Methodology

It was shown how C++ can eliminate runtime overhead based on the disassembled binary. However, the effects can also be demonstrated during execution and the conclusions can be verified based on actual measurements.

It is not trivial to perform meaningful measurements of the time it takes to compile and execute a software. Small projects are not suitable for such measurements, because the time it takes to compile and run them on a modern and powerful PC is very small and, as such, measurements are very error-prone. Creating a large, functional code-base takes a lot of time and effort and do not show effect of single C++ language features. The solution developed involves using *Python* and the *Jinja2* template engine [26] to generate large, functionally equivalent codebases in both C and C++ in order to allow for meaningful measurements. This includes generating preprocessor defines, functions, templated functions, enumerated values and even up to tens of thousands of function calls. The generator and benchmarking toolchain consists of several shell and python scripts, as well as JSON configuration files and Jinja2 templates, all of which can be found in Appendix D.

The key variables of the measurements are the following:

- **Number of generated functions**: if a measurement is performed with twenty distinctly generated functions, it means, that in the C source code, there are twenty different functions that can be randomly called from the main function. Similarly in the C++ version, the same functions are generated as templated function. In the case of C, every distinct function (which is actually called) is included in the binary. And in the case of C++, a new template specialization is generated for every distinct function call, meaning that more function templates result in more work for the compiler. By changing this variable, it can be observed how the two compilers handle the increasing load, affecting both compilation time and binary size.
- **number of function calls**: if a measurement is performed with the *number of function calls* set to ten, then there will be ten randomized function calls generated in the main file. Tn the case of C++, function calls result in template specialization, meaning that the compiler need to generate a function for every distinct function call. The more functions are called, the more functions need to be generated by the compiler. Similarly for C, functions are only included in the binary if they are actually called in the source code. This parameter is also necessary for the runtime performance measurement. By increasing the number of function calls, the runtime of the applications can be increased into a range where they can measured with higher accuracy, mitigating the tolerances of time measurement.

As illustrated in Figure 6.1, each measurement is performed for a fixed number of generated functions and a variable number of function calls, increasing with each iteration. The number of generated functions is changed between measurements. For better understanding, the following is a summary of how the measurements are executed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- The measurements are performed in a PC running Ubuntu 22.04.1 and not on an embedded device. There are multiple reasons for this decision. On the one hand, an embedded device usually has limited available memory, and hence large binaries do not fit. On the other hand, time measurement is much simpler using an operating system like Ubuntu. Finally, thousands upon thousands of automated tests were performed, taking several hours on four dedicated processor cores, which is simply not feasible on an embedded platform as part of this work.
- The measurements are done parallel on four dedicated processor cores to speed up the process. Even so, executing one whole set of measurements takes nearly up to eight hours. The shell script that runs the testing script on the dedicated cores is shown included in Appendix D.9.
- The processor cores are isolated to eliminate scheduling effects. This helps with mimicking the behavior of an embedded system, where processes run on a single processor without the interference of a scheduler.
- The functionality of the generated C and C++ code is the same. They both perform operations on an array just as they would on a memory register in an embedded firmware. The structure of the generated code is similar to that presented in Chapter 4. The C code performs a number of randomized function calls with randomized parameters. The functions have runtime decisions and bitmask calculations, just as they would have on an embedded platform. The C++ code performs the same randomized function calls with the same parameters as the C version, however, it uses enum classes, templates, concepts and consteval functions to achieve the same functionality. The two Jinja templates can be found in Appendix D.2 and D.4. The generator Python script that parses the Jinja templates and generates the actual C and C++ files can be found in Appendix D.6.
- Each measurement generates the source code, compiles, links and executes the project while measuring the time it takes to execute each step. Figure 6.1 illustrates the measurement process. To eliminate the overhead of program execution on a PC platform, the program execution is measured by the programs themselves using the *clock_gettime* [27] function starting at the beginning of the main and ending before the main function returns. This also helps bringing the measurement closer to that of an actual embedded device. Compile time and link time are measured by the executing shell script.
- The measurements also measure the size of the binary compared to a base binary. The base binary contains the array and the code responsible for time measurement but does not contain the generated functions and function calls. And as such, the additional overhead of those features are accounted for an eliminated from the results. The Jinja templates for the C and the C++ base software can be found in Appendix D.1 and D.3 respectively.
- Compilations are performed using GCC version 11.4.0.
- The script that does the actual measurements can be found in Appendix D.8. The script writes the results into CSV files which can be plotted using the plotting script from Appendix D.7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

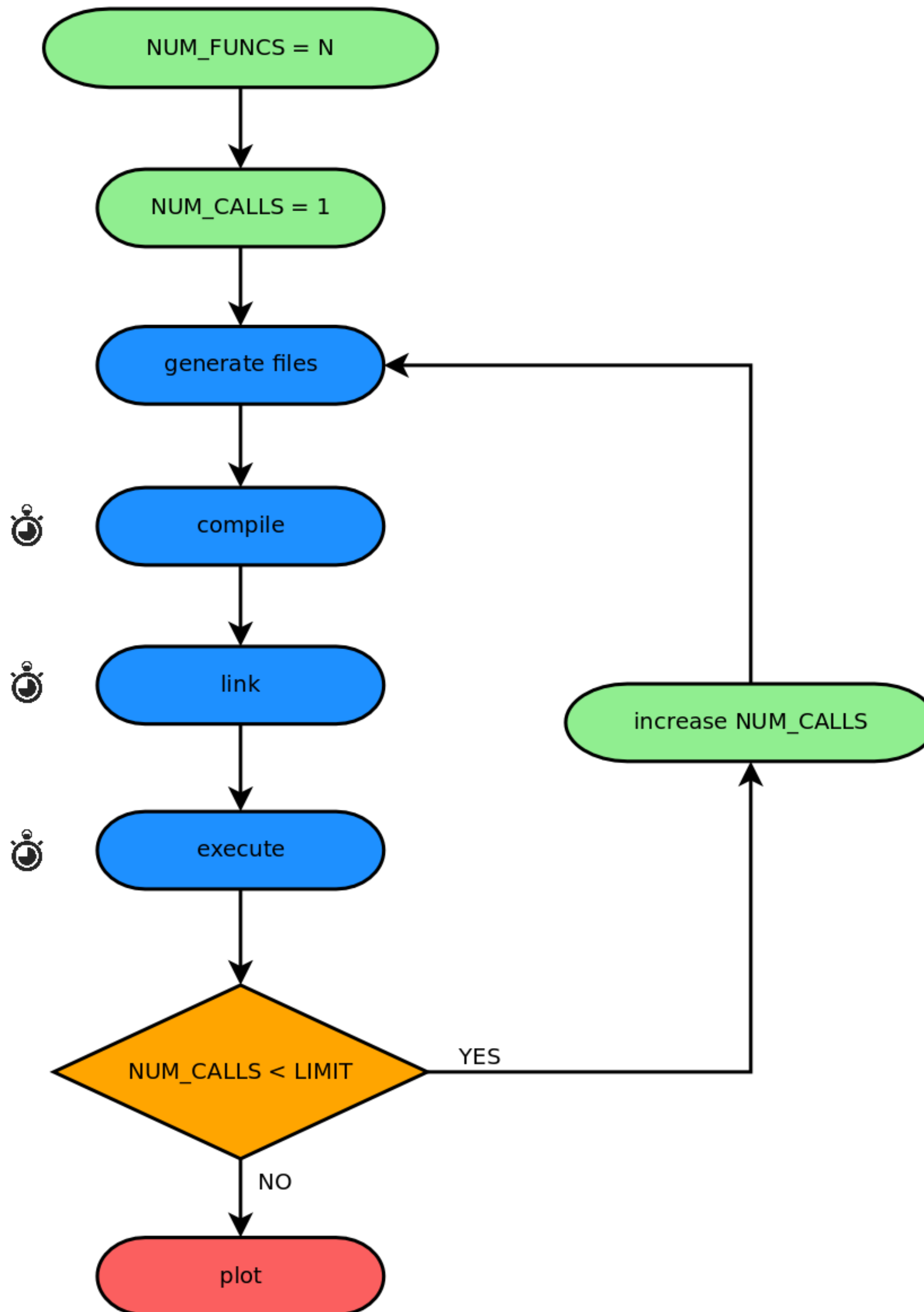



FIGURE 6.1: Measurement flowchart.

The goal of the measurements is to assess not only the execution time but also the compilation time of the applications. The execution time is expected to be faster for the C++ code, because many of the computations are pushed to compile-time. Consequently, it would be expected that the compilation time for the C++ software is longer, given that the compiler needs to resolve the templates and perform the calculations during compilation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

### 6.1.2 Binary Size

The conclusion derived from the analysis of the disassembled binaries in Chapter 4 directly correlates with the measurement result shown in Figure 6.2. Figure 6.2 shows the size overhead of the binaries in bytes in relation to the number of function calls in the main function for a single generated function. This means that a single function was generated by the scripts and called a variable amounts of times, each time assessing the binary size.

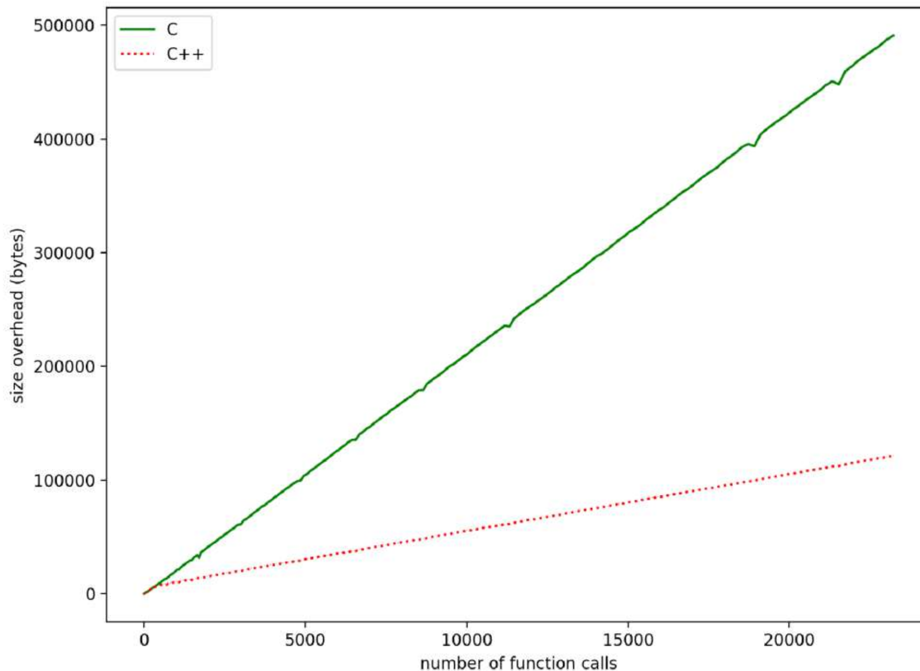



FIGURE 6.2: Binary size overhead for 1 generated function.

As demonstrated in previous chapters, the C binary contains a single instance of the function. This function is called from the main function each time with different parameters. However, the C++ compiler has a much higher flexibility, since it works with templated functions. Analyzing the disassembled binary for different function call numbers, it shows that if the function is called only a couple of times, the compiler eliminates the function call altogether and inlines the operations. However, when a certain specialization of the templated function is called multiple times, the compiles decides to generate the function, includes it in the binary and performs function calls to the function rather than inlining it every time. This is the reason why up until 300-400 function calls, in Figure 6.2, the binary size of the C++ application seems to increase similarly to the C application.

This phenomenon is more visible in Figure 6.3. In this case, there are twenty different function generated by the generator scripts. The C version is straightforward, it has the twenty functions in the binary, those functions are called each time. It has a linear relationship to the number of function calls, the more times the functions are called, the larger the binary grows. On the other hand, the C++ compiler has a non-linear relationship to the number of function calls. The binary starts the increase

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

similarly to C as the compiler starts to generate the code for every function specialization. However, when every function is generated, then a C++ function execution is less expensive than the same C function execution, since the runtime branching is eliminated at compile-time.

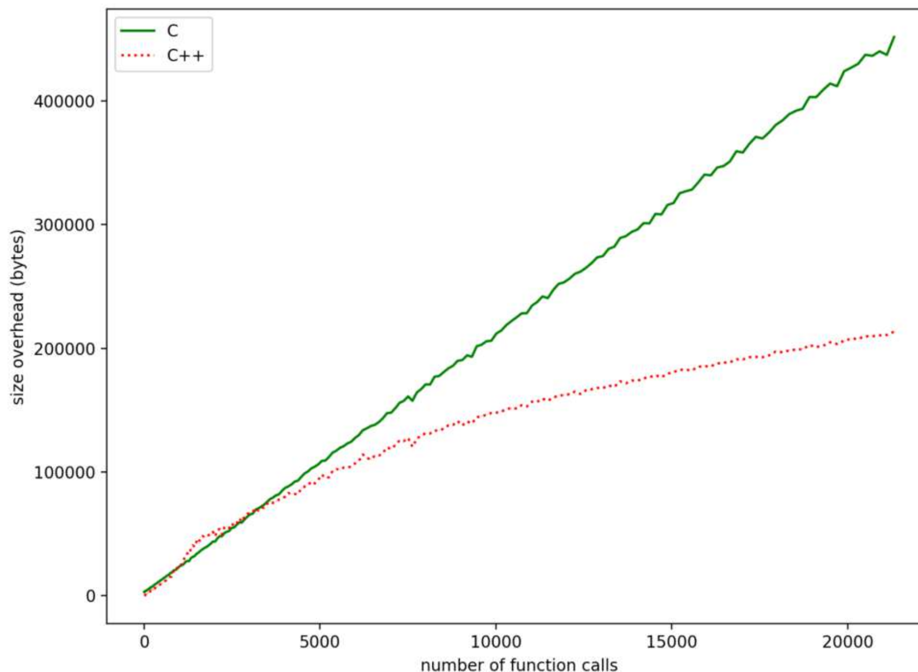



FIGURE 6.3: Binary size overhead for 20 generated functions.

There are cases where the C++ binary size can increase more rapidly than the C binary size. This occurs due to code bloating, which is discussed in Section 7.1.

These results prove conclusively by the measurements that C++ code can be just as effective, or even more effective than its C counterpart, when talking about binary size.

### 6.1.3 Compilation Time

The conclusions of the compilation time measurements were a surprise to me. I assumed that the compilation time of the C++ version is going to be higher than the compilation time of the C source code, since the compiler needs to do far more work due to template specialization and compile-time bitmask calculations. However, this is not what the measurements showed. As shown in Figure 6.4, C++ start off with a slightly higher compilation time, but increases more slowly compared to the C compilation time. It shows that the compiler performs a lot of work during compilation to generate the specialized code for each template instantiation, however, once the template is instantiated for a particular type, reusing it does not significantly increase the compilation time, because the compiler does not need to redo all the work. On the other hand, C treats each function call more independently, leading to rapidly increasing compilation time with more calls.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

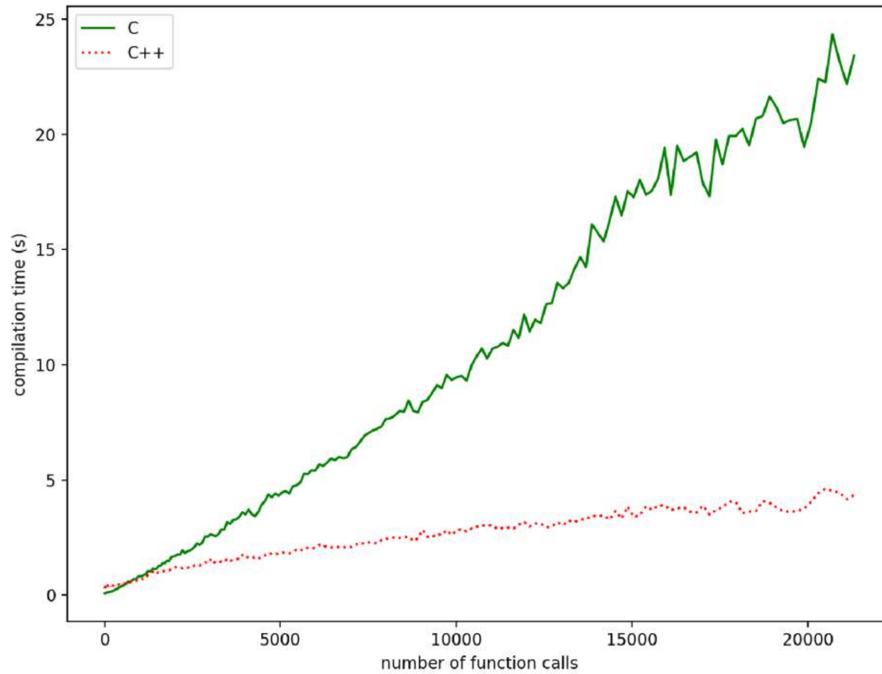



FIGURE 6.4: Compilation time for 20 generated functions.

With increasing number of function calls, the compilation time of C++ starts to increase similarly to C. For a hundred different generated function and function templates, the C++ compilation time is only slightly better than that of the C project, as shown in Figure 6.5.

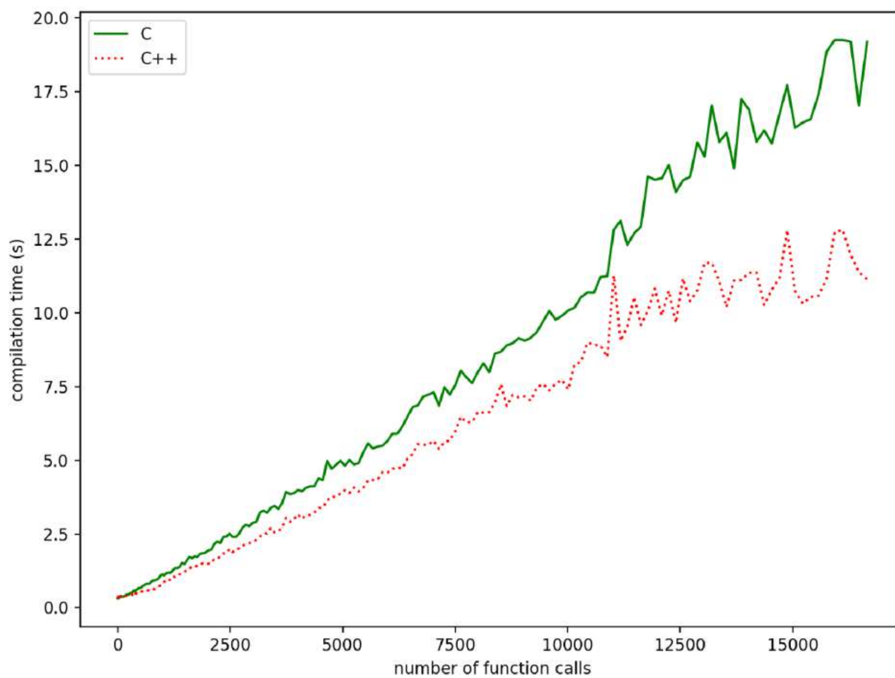



FIGURE 6.5: Compilation time for 100 generated functions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The conclusion of these measurements is that the compilation time of a C++ software can be better than that of a C project, even if extensive compile-time calculations and template specializations are required from the compiler. Pushing otherwise runtime operations to compile-time does not come with large compile-time overhead, which might be counter-intuitive.

### 6.1.4 Link Time

The link time measurements are not of high relevance, because the source code is generated as a single source file. Therefore, linking is not used intensively. However, for the sake of completeness, it can be concluded that C++ default link time is higher than that of C. The measurement result for a hundred generated functions is shown in Figure 6.6. It shows that the C++ compile time increases slightly with the increasing number of function calls, while the C link time remains fairly constant. However, both values are insignificant when compared to the compilation time of these projects.

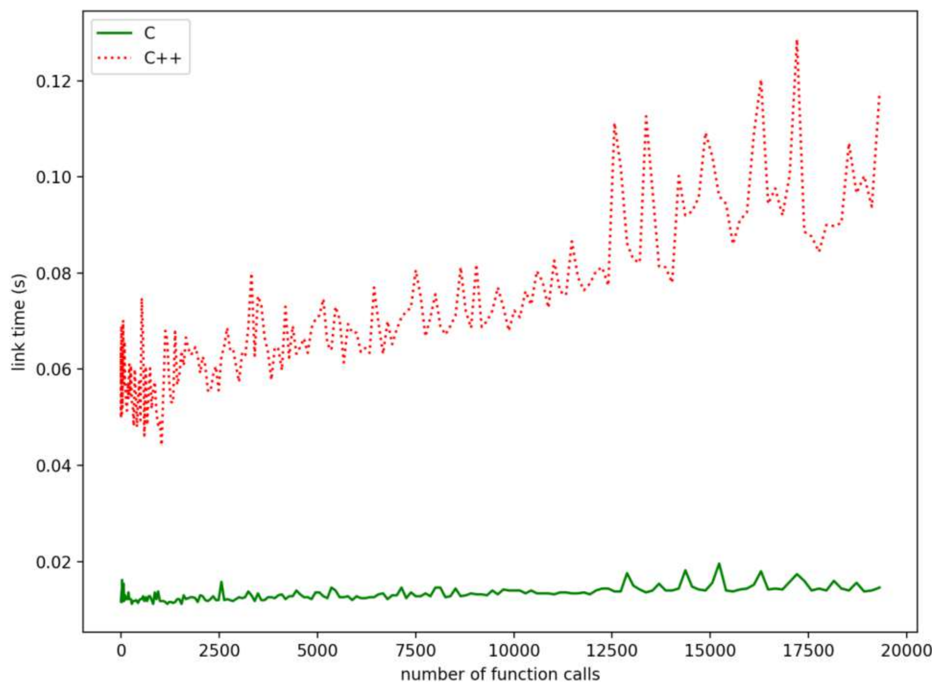



FIGURE 6.6: Link time for 50 generated functions.

### 6.1.5 Execution Time

The measurement of execution time is the most important measurement, since this is the main focus of the thesis. Previously, execution time was inferred based on the analysis of the disassembled binaries in Chapter 4. These measurements allowed for the verification of those findings.

Figure 6.7 shows that the measurements for a single generated function, the C++ version runs approximately twice as fast as its C counterpart. Both execution times

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

increase linearly with increasing function calls, but the C++ version increases more slowly due to the fact that each function execution is optimized during compilation. Runtime decision-makings and bitmask calculations are both eliminated during the compilation process.

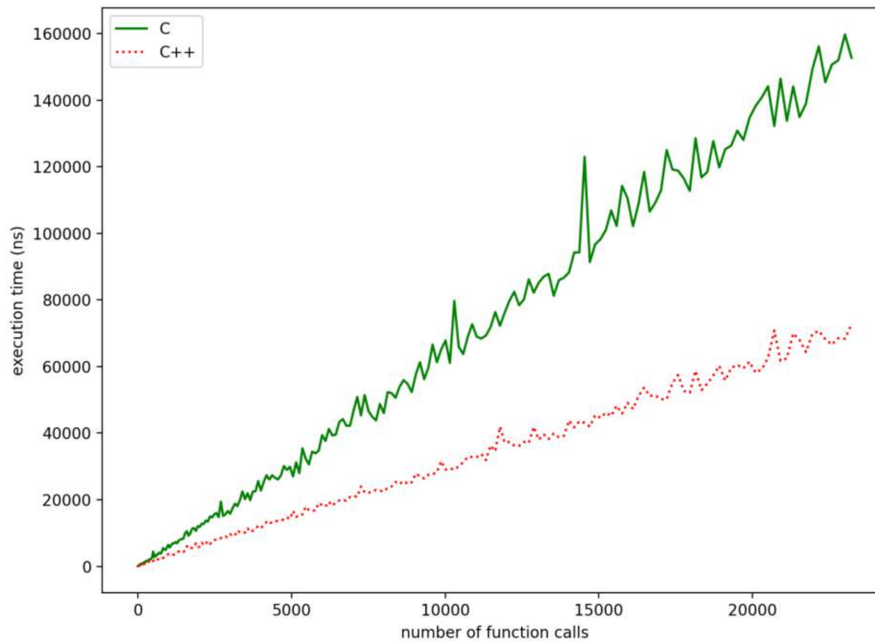



FIGURE 6.7: Execution time for 1 generated function.

The relation does not change drastically even for a hundred distinctly generated functions, as shown in Figure 6.8.

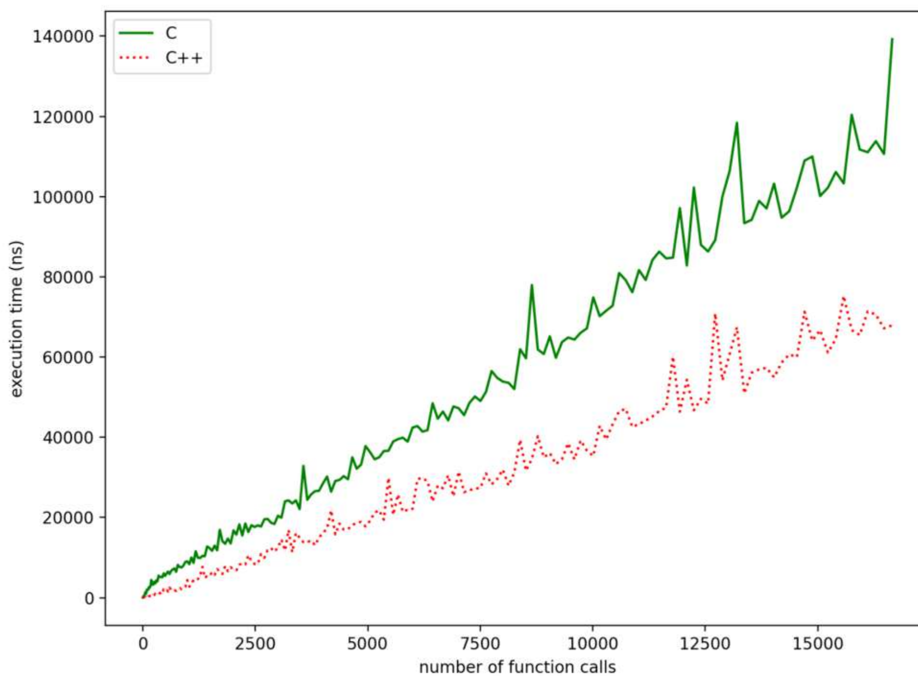



FIGURE 6.8: Execution time for 100 generated functions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The conclusion of this measurement is that execution time can indeed be optimized using modern C++ language features. It also proves that the inferences previously made on execution time based on purely the disassembled binary were correct. And since in the context of electronic devices, energy consumption is directly related to execution time, this shows that using modern C++ instead of C can lead to a more sustainable product in certain situations.

## 6.2 Overhead Assessment of Abstractions

In Chapter 5, the runtime overhead of abstractions inherent to object-oriented programming languages was shown. It was demonstrated that most abstractions in C++ come with zero runtime overhead. This section provides a brief summary of the results of the analysis.

- **Encapsulation** (Section 5.1) allows developers to organize data and logic into classes and handle them as objects. As demonstrated, such an abstraction does not inherently lead to runtime overhead. Objects are compile-time entities and are not present in the disassembled binary.
- **Inheritance** (Section 5.2) makes it possible to share certain features between the base class and its derived classes. As it was demonstrated, in its base form, meaning without the use of virtual functions, it does not lead to runtime overhead. The compiler is capable of resolving the inheritance structure at compile time and does not introduce runtime penalty for using the abstraction.
- **Polymorphism** (Section 5.3) provides a way to structure code so that a single common interface can be used to manipulate objects of different types.
  - As it was demonstrated, the most well-known form of polymorphism, namely **dynamic polymorphism**, does come with inherent runtime overhead, since runtime type information must be stored and handled at runtime to achieve the desired functionality. Virtual tables and virtual function calls introduce runtime overhead as well as binary size increase, although the effects are small.
  - **Static binding or compile-time polymorphism**, on the other hand, forces the compiler to resolve the inheritance at compile time. This mechanism, similarly to dynamic polymorphism, allows for writing reusable code that uses a common interface to handle objects of different types. However, the interface in this case is a compile-time concept, which makes it possible to eliminate the need for runtime type information, while it is still featuring a high degree code reusability.

These abstractions can make a C++ codebase superior to the same project implemented in the conventional way using C. They make the code more readable, maintainable and less error-prone. The fact that most of these features come with zero runtime overhead leaves developers with no excuse for not using them in modern embedded projects.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

### 6.2.1 Measurement Methodology

For measuring the overhead of abstractions, a similar method to the one described in Section 6.1.1 was used. The only difference in this measurement setup is that the source codes are generated directly using Python, because it offers the necessary flexibility for generating different architectures in the C++ code. The generator script, the plotting script and the benchmarking script can be found in Appendix D.10, Appendix D.11 and Appendix D.12 respectively.

To perform a through comparison, the generator script generates the following, functionally equivalent sources:

- C codebase similar to the one generated and benchmarked in Section 6.1. A number of functions are generated, which perform randomized register operations. Function calls are generated in the main file.
- C++ codebase similar to the one generated and benchmarked in Section 6.1. It uses templated functions, concepts for parameter validation and compile-time branching.
- A C++ codebase, where the register operations are encapsulated in register classes. The generated register classes are all derived from the same register base class, which implements the reading and writing of the actual register. The registers are further encapsulated in classes representing some kind of peripheral or device driver. These classes instantiate the registers and set their values using their presented member functions. The code structure is similar to the one presented in Listing 5.8 in Section 5.2.
- Another C++ codebase, which introduces static polymorphism to the code, similarly to the code presented in Listing 5.11 in Section 5.3.2.
- A C++ codebase, which uses dynamic polymorphism similar to the code in Listing 5.9 in Section 5.3.1.

The goal of the measurements is to assess to overhead of the abstractions compared to each other as well as to the original C code.

### 6.2.2 Binary Size

The measurement of the binary size substantiates the conclusions made in Chapter 5, namely that using abstractions that can be resolved at compile-time does not add overhead to the binary. As it is shown in Figure 6.9, the binary size of the compiled code that uses runtime polymorphism is far greater than the binary size of the other projects. However, it is still recognizable even on this larger scale that the second greatest binary size is produced by compiling the C source code. The other C++ variants produce very similar binary sizes, meaning that the usage of these abstractions does not result in a binary size increase.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

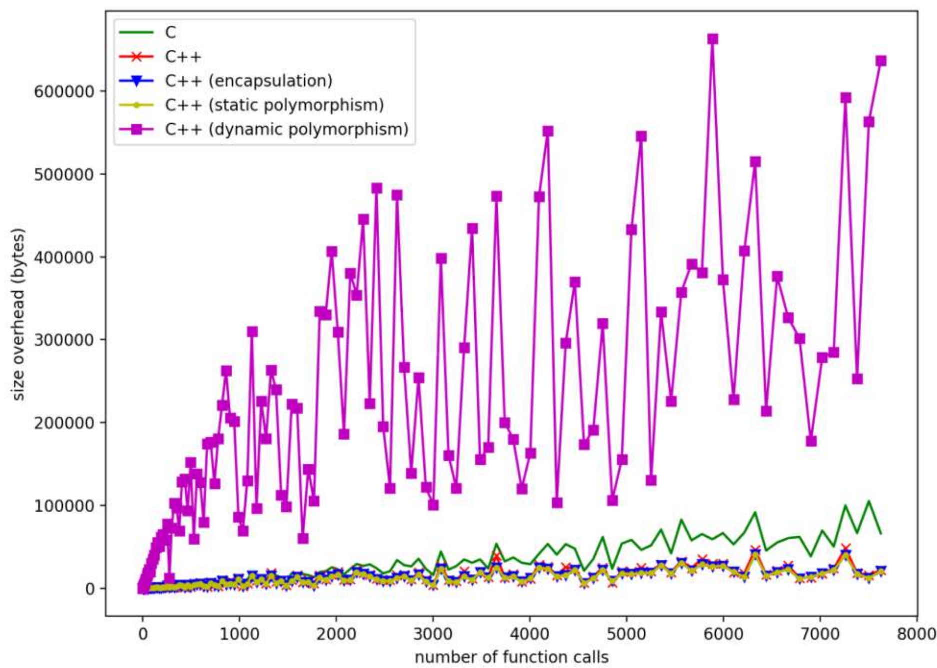



FIGURE 6.9: Binary size for 20 generated functions.

### 6.2.3 Compilation Time

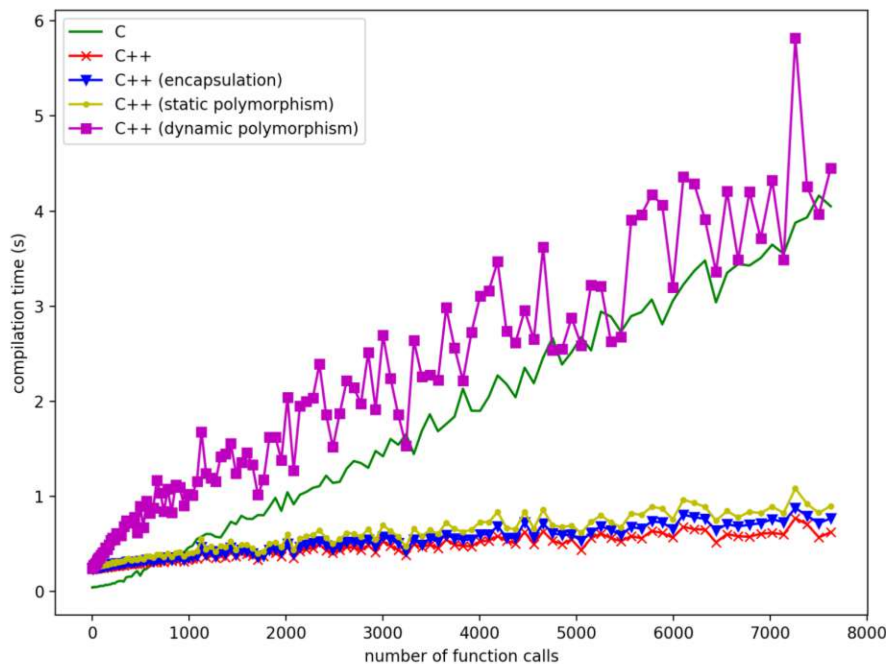



FIGURE 6.10: Compilation time for 20 generated functions.

As shown in Figure 6.10, the compilation time of the C++ sources correlates with the complexity of the abstractions. The version that does not use any abstractions has the lowest compile time. It is followed by the version that encapsulates the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

functionalities in classes. The third is the version that adds a further layer of abstraction and uses static polymorphism in the code. By far the longest compile time is measured for the source code that uses runtime polymorphism. However, it is still comparable with the compilation time of the C project. Although the C version starts with the best initial compilation time, it increases faster than the compile time of the C++ projects that use compile-time abstractions.

### 6.2.4 Link Time

The link times shown in Figure 6.12 are very small compared to the compilation times shown in Figure 6.10. However, the C++ version that uses runtime polymorphism performs tha worst in this measurement as well. The C link time outperforms the C++ link times. However, it is important to note that the sources are generated as single files, and as such, the linker is not used extensively.

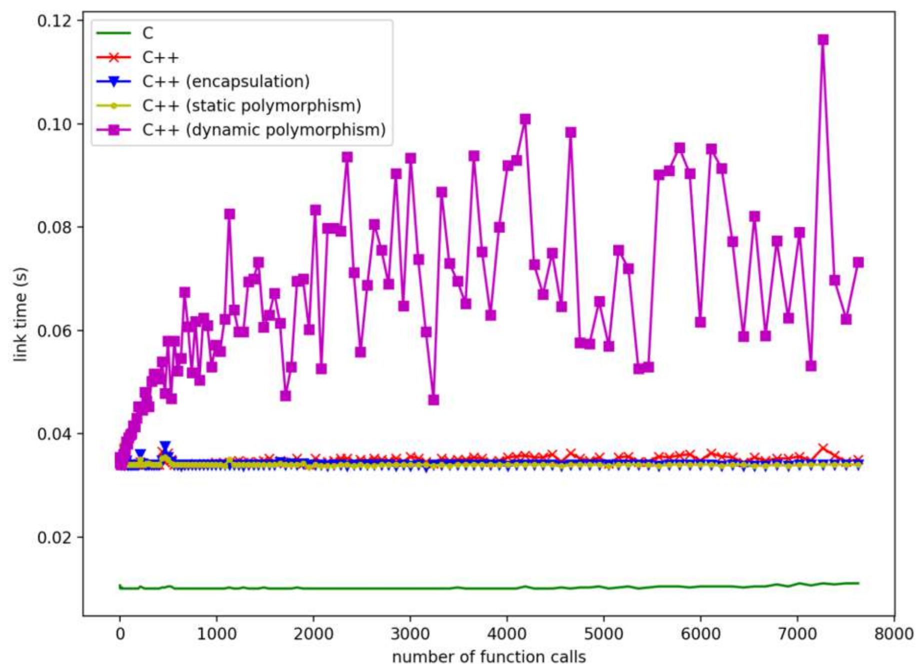



FIGURE 6.11: Link time for 20 generated functions.

### 6.2.5 Execution Time

Figure 6.12 shows the measured execution times of the different codebases. Dynamic polymorphism, being an inherently runtime paradigm, adds extensive runtime overhead. However, it also shows that the layers of abstractions added by the other C++ codebases do not affect the runtime performance. Encapsulation, inheritance and static polymorphism are resolved at compile time and thus do not introduce additional runtime overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

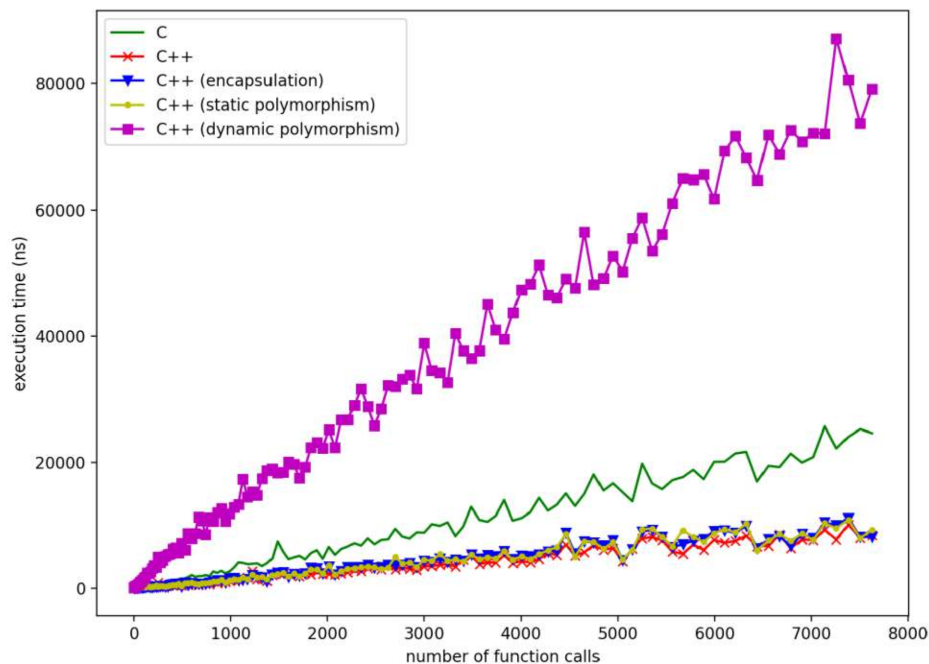



FIGURE 6.12: Execution time for 20 generated functions.

The measurement of execution time substantiates the conclusions of Chapter 5, namely that many C++ abstractions can be used in resource-constrained environments, because they do not add runtime overhead. In fact, they can be used to reduce runtime overhead inherent to C implementations. The measurement also provides practical evidence for the analysis done in Section 5.3.1, which showed that runtime polymorphism can greatly affect runtime performance.

## 6.3 Practical Embedded Example

The goal of the previous subsection was to introduce reproducible results for individual measurements using example programs specifically designed with the purpose of getting analyzable results that help assessing the quality of the compiled code. This section, on the other hand, presents a codebase that features many of the previously discussed language features and performs a meaningful task on the target STM32 platform. It will examined how an embedded program, the program that blinks an LED, preforms when using modern C++. Note that only the necessary parts of the *Hardware Abstraction Layer* are implemented to keep the codebase small enough for a meaningful discussion. Many registers are not modified, because the reset value is right for the given task.

### 6.3.1 Project Structure and Testing Environment

The test setup is shown in Figure 6.13. An LED is connected to the *PA6* pin of the microcontroller. The device is programmed using the *ST-Link/V2* in-circuit debugger and programmer over the *Serial Wire Debug (SWD)* interface of the microcontroller.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

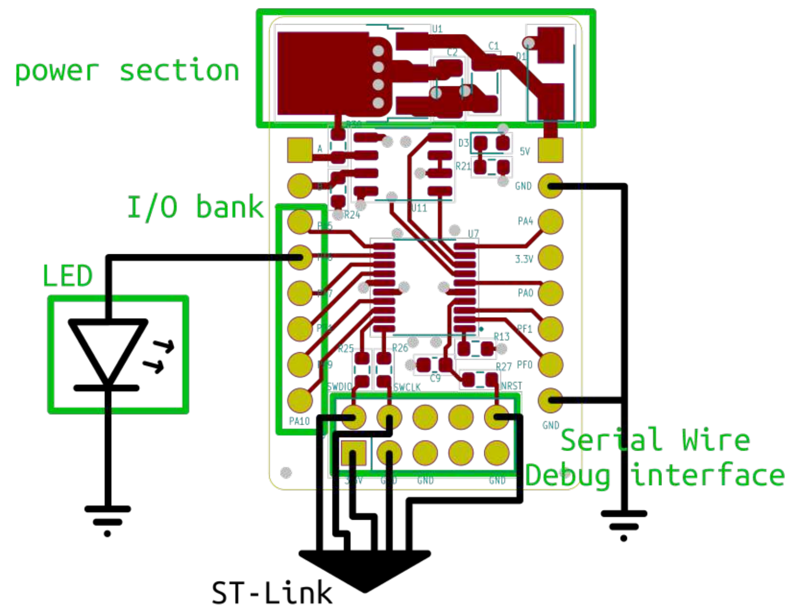



FIGURE 6.13: Test setup.

The firmware was tested on the test board shown in Figure 6.14. The board was specifically designed for this project, as it is a low-cost and simple device. The PCB contains the STM32F030F4 microcontroller, operates from either a 5V external power supply through the on-board voltage regulator or directly from 3.3V.

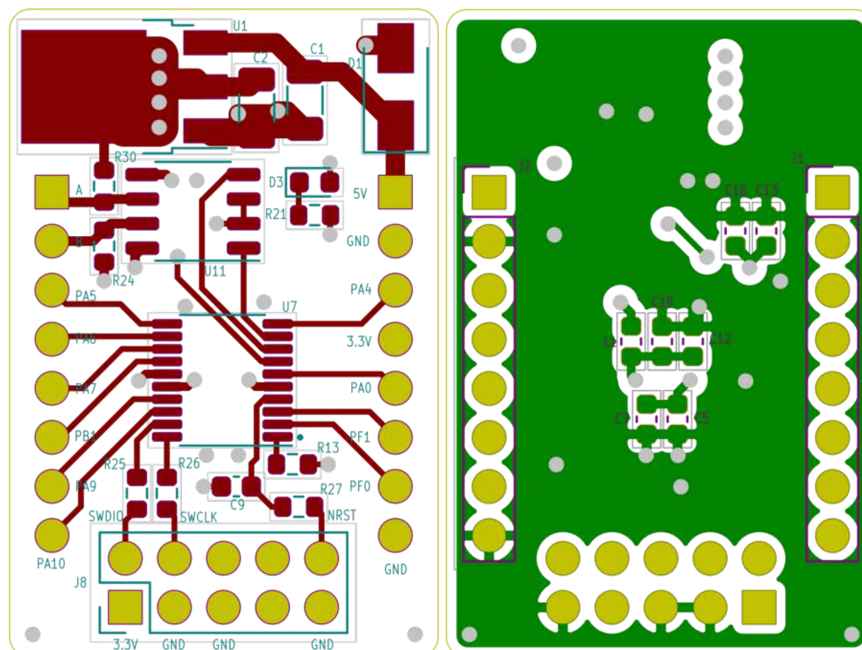



FIGURE 6.14: Test board layout.

The directory structure of the project is shown in Figure 6.15. The project features a header-only HAL library implementation, containing the necessary abstractions for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the GPIO (General Purpose Input/Output) and the RCC (System-Reset and Clock Control) peripherals. The implementations of these peripherals can be found in the *gpio* and the *rcc* directories respectively. The *constants.hpp* in the *base* directory contains the register base addresses of the peripherals. The *register.hpp* file contains the generic register class, which implements the generic register operations as member functions. The *concepts.hpp* contains the *compile-time interface* of a GPIO class, requiring the necessary functions to be implemented by a hardware-specific implementation. The *led.hpp* contains the hardware-independent device driver for an LED device, which operates with the hardware-independent GPIO type. The codebase is too large for presenting all of it here, so please refer to Appendix E to find the source code. For code compilation and for flashing the device, the same linker script, startup code and makefile were used, which can be found in Appendix B, Appendix A and Appendix C respectively.

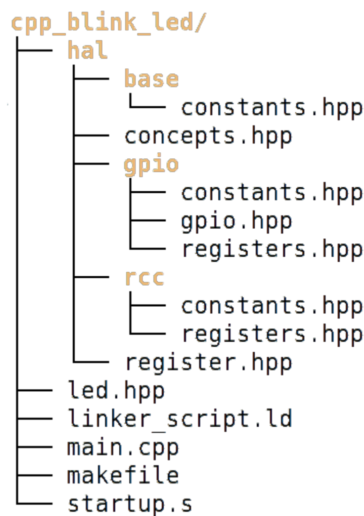



FIGURE 6.15: Project structure of the demo project.

### 6.3.2 C++ Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is a crucial concept in the realm of embedded systems and system-level software development. In essence, a HAL is a layer of programming that allows software to interact with hardware in a way that is independent of the specifics of the underlying hardware. This abstraction layer sits between the system's hardware and software, serving as a bridge that facilitates interaction between the two.

As previously discussed, the firmware only encompasses the necessary registers and functionalities required to make the LED blink. Typically, the application would start with configuring the system clock, establishing the clock's source and setting its frequency. Nonetheless, for this example, the default configuration is adequate. The same is true for certain aspects of the GPIO configuration. A fully-fledged Hardware Abstraction Layer would necessitate the implementation of an extensive array of registers. However, in this illustrative firmware, many GPIO functions are not employed and certain other GPIO registers already hold suitable reset values for this application. Therefore, it was only necessary to implement the following registers:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **RCC_AHBENR**: AHB peripheral clock enable register. Used to enable peripheral clock for different peripherals, including the GPIO ports.
- **GPIOx_MODER**: Controls the mode of the GPIO pins. The reset state of the register sets most pins to input mode.
- **GPIOx_BSRR**: GPIO port bit set/reset register. Used to set the logic level of output GPIO pins.

Figure 6.16 shows the UML diagram of the register classes. The *CRegister* class is the abstraction of the generic register concept. It can be used to read and write a memory-mapped register, either as a whole, or only selectively using bitmasks. The classes *CAhbEnRegister*, *CBSRRegister* and *CModeRegister* add another layer of abstraction over the *CRegister* class and provide meaningful interfaces to handle the underlying registers as intended.

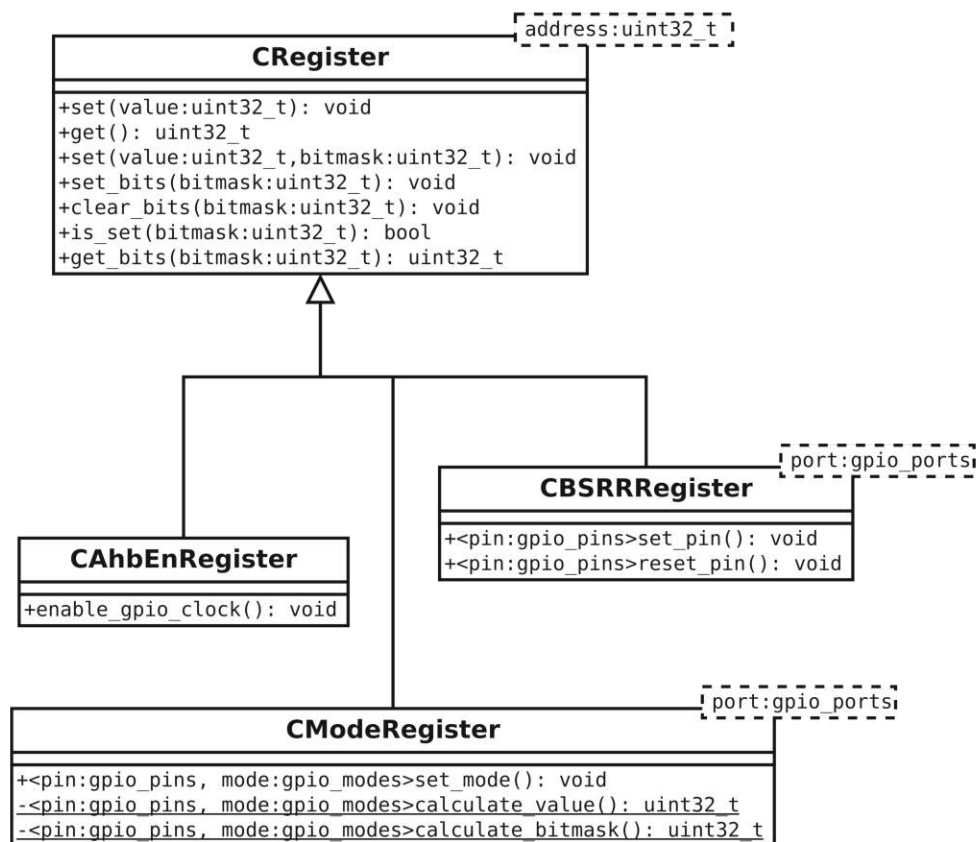



FIGURE 6.16: UML diagram of the registers.

Figure 6.17 shows the UML diagram of the higher-level abstractions. Note that strictly speaking the UML diagram is intentionally not entirely correct, because the *CGpio* class does not implement the *GPIO concept* in a conventional way. However, to my current understanding, UML does not provide a way to represent this kind of relationship between classes. Similarly to the example shown in Listing 5.11 in Chapter 5.3.2, the *GPIO* is defined as a *concept* to enforce constraints on types. To keep the example small enough for meaningful analysis, a class is considered a GPIO class if it implements the following functions:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- **configure**: Set the configuration bits in the necessary registers.
- **set**: Set the pin to logic high.
- **reset**: Set the pin to logic low.

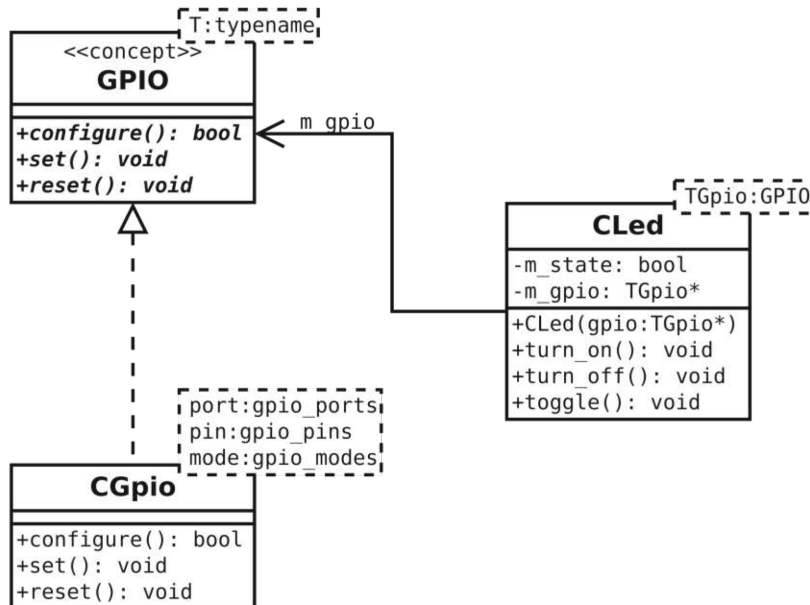



FIGURE 6.17: UML diagram of the GPIO and LED classes.

The *CLed* class is a high-level abstraction of a single LED device, and is implemented in a hardware-agnostic way. It accepts types that satisfy the type constraints defined by the *GPIO concept* and offers functions to turn on, turn off and to toggle and LED connected to a GPIO pin. The *CGpio* class meets the type requirements and thus can be used as template parameter for the *CLed* class.

The *CLed* class can be thought of as a hardware-independent device driver. The hardware-related implementations are encapsulated in the *CGpio* class, which means that the *CLed* device driver can be ported to any platform without the need to modify its implementation. Replacing the underlying STM32-specific GPIO class is the only necessary step in the porting process. This might not seem important at first glance, because an LED is a fairly simple device which can be implemented in only a couple of lines of code. However, other external devices can be quite complicated and the driver implementation can even extend to thousands of lines. Porting such device drivers to a new hardware can take a lot of work and effort if the device directly interacts with the hardware or the hardware-specific HAL library.

### 6.3.3 Main Function

Listing 6.1 shows the implementation of the *main* function. A brute-force *delay* function is implemented in lines 6 through 9 so that the blinking of the LED can be recognized by the human eye. The *CGpio* class is instantiated with the required template parameters. It was mentioned before that in embedded systems, many parameters of the applications are known at compile-time, and this is a good example

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

to support that statement. Pin *PA6* is hard-wired to an LED, it is known before the development of the firmware is started that it is going to operate in output mode. The *decltype* specifier in line 17 can be used to explicitly specify the type for the template class instantiation, although the compiler is able to deduce it as well.

```
#include <cstdint>

#include "hal/gpio/gpio.hpp"
#include "led.hpp"

void delay (int cycles) {
    volatile int i;
    for (i = 0; i < cycles;) { i = i + 1; }
}

using namespace hal;

int main (void) {

    CGpio<gpio_ports::port_a, gpio_pins::pin_06, gpio_modes::output>
        gpio {};

    CLed<decltype(gpio)> led { &gpio };

    while (true) {
        led.toggle();
        delay(200000);
    }
}
```

LISTING 6.1: main.cpp of the LED blinking example.

### 6.3.4 Fully Optimized C Code

To better illustrate the efficiency this code is, it is compared to a functionally equivalent C implementation. The C code presented in this section is manually optimized to an extreme extent, containing only the necessary register operations to blink the LED. Normally, firmware is not written this way, however, the goal here is to compare the regular C++ implementation to its fully optimized C counterpart.

The following register operations are necessary to blink an LED:

- **initialization**:
    1. Enable the peripheral clock for the GPIO port A by setting the 17th bit in the *RCC_AHBENR* register to 1. Without this step, the modifications to the registers implemented in the source code would not take effect.
    2. Set the mode of pin PA6 to output by setting the 13th bit to 0 and the 12th bit to 1 in the *GPIOA_MODER* register. Setting the pin to output is necessary for being able to not just read, but even set the voltage of the pin from the firmware.
    3. Set pin PA6 low by setting the 22nd bit to 1 in the *GPIOx_BSR* register. This is done to make sure that the pin is in a known state after reset.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **main loop**:
  1. Set pin PA6 to high or to low using the *GPIOx_BSR* register, depending on its previous state. Setting the pin to high will cause the pin voltage to increase to 3.3V, turning the LED on. Setting the pin low results in the pin voltage to drop to 0V and thus turning the LED off.
  2. Call the *delay* function, so that even the human eye can perceive the blinking of the LED.

Listing 6.2 shows a possible implementation of the above described steps. The code is probably as efficient as it gets, there are no function calls, no indirections, only simple register operations are performed with manually precalculated addresses, values and bitmasks. The AHB peripheral bus is enabled in line 17 by flipping the necessary bit using a simple binary or operation on the register. The code in lines 19 through 22 clear and set the necessary bits in the mode register of port A, making sure that the pin *PA6* is in output mode. In line 24, the pin is set to its default low state by setting the dedicated bit in the ser-reset register. In the while loop, the LED is toggled and the delay function is called.

```
1  #define AHBENR       (*((volatile uint32_t*)0x40021014))
2  #define GPIO_A_MODER (*((volatile uint32_t*)0x48000000))
3  #define GPIO_A_BSRR  (*((volatile uint32_t*)0x48000018))
4
5  void delay (int cycles) {
6      volatile int i;
7      for (i = 0; i < cycles;) { i = i + 1; }
8  }
9
10 int main (void) {
11     uint32_t tmp;
12     bool led_on = false;
13
14     AHBENR = AHBENR | 0x00020000;
15
16     tmp = GPIO_A_MODER;
17     tmp &= ~0x00003000;
18     tmp |= 0x00001000;
19     GPIO_A_MODER = tmp;
20
21     GPIO_A_BSRR = GPIO_A_BSRR | 0x00400000;
22
23     while (1) {
24         if (led_on) {
25             GPIO_A_BSRR = GPIO_A_BSRR | 0x00400000;
26             led_on = false;
27         }
28         else {
29             GPIO_A_BSRR = GPIO_A_BSRR | 0x00000040;
30             led_on = true;
31         }
32         delay(500000);
33     }
34 }
```

LISTING 6.2: Manually optimized and functionally equivalent C code for blinking an LED.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

### 6.3.5 Overhead Assessment and Conclusion

When the code from Listing 6.2 is compiled and the result are compared with that from Sections 6.3.2 and 6.3.3, it is found that two projects produced nearly the exact same ELF file and binary. The text size of both projects is 864 bytes (the base firmware's text size was 744 bytes), and the binary itself is 976 bytes in both cases. Comparing the *objdump* outputs reveals that the two are nearly identical. There is some minor difference in the *delay* function, but nothing in the *main* function.

Given the length of this example, it is not practical to closely examine each and every assembly instruction. However, a brief look at the *main* function reveals a lack of function calls, apart from invoking the *delay* function in line 11. Listing 6.3 showcases the initial and final lines of the *main* function. This is no surprise for the C version from Listing 6.2, because the code was manually optimized to an extreme extent. However, for the C++ project, the compiler achieved the same optimized binary as was manually accomplished with C. The main in Listing 6.3 function begins by loading the address of the *RCC_AHBENR* register, 0x40021014, in line 3. In the C++ code, this operation is performed when the *CLed* class's constructor calls the *configure* function of the *CGpio* object. Although this operation is hidden beneath layers of abstraction, these layers are optimized away by the compiler. Consequently, all that remains in the *main* function are simple register operations.

```
080001b4 <main>:
 80001b4:       2380            movs    r3, #128        ; 0x80
 80001b6:       4a14            ldr     r2, [pc, #80]   ; (8000208 <
    main+0x54>)
 80001b8:       029b            lsls    r3, r3, #10
 80001ba:       6811            ldr     r1, [r2, #0]
 80001bc:       b570            push    {r4, r5, r6, lr}
 80001be:       430b            orrs    r3, r1
 80001c0:       2190            movs    r1, #144        ; 0x90
 80001c2:       6013            str     r3, [r2, #0]
 ...
 80001f6:       f7ff ffd1       bl      800019c <_Z5delayi>
 80001fa:       e7f3            b.n     80001e4 <main+0x30>
 80001fc:       6823            ldr     r3, [r4, #0]
 80001fe:       2501            movs    r5, #1
 8000200:       4333            orrs    r3, r6
 8000202:       6023            str     r3, [r4, #0]
 8000204:       e7f6            b.n     80001f4 <main+0x40>
 8000206:       46c0            nop                     ; (mov r8, r8)
 8000208:       40021014        .word   0x40021014
 800020c:       ffffcfff        .word   0xffffcfff
 8000210:       48000018        .word   0x48000018
 8000214:       00030d40        .word   0x00030d40
```

LISTING 6.3: Parts of the assembly of the compiled project.

Based on the above measurements, it can be concluded that with modern C++ it is possible to develop firmware with a well-structured architecture and with zero runtime overhead. The compiled code can be just as efficient as the manually fully optimized C implementation. Even when the actual register manipulating code sections are hidden under layers upon layers of abstractions, the compiler is still capable of collapsing the architecture and producing a highly efficient binary, while

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

developers can enjoy the readability, extendability and maintainability of a modern, well-structured project.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Chapter 7

# Further Considerations

## 7.1 Code Bloating

In Chapter 4.3, the usage of templates and concepts was discussed with the goal of runtime overhead reduction. On the flip side, templates are not without drawbacks. They can often result in an issue commonly referred to as *code bloat*. Code bloat happens when the code size becomes significantly larger due to the excessive use of templates. Since every distinct instantiation of a template generates a new piece of code, it can quickly lead to a drastic increase in the size of the binary executable, particularly when templates are used indiscriminately or without a proper understanding of their implications. This not only increases the footprint of the application, which is a critical issue for memory-constrained embedded systems, but it can also detrimentally affect the performance of otherwise powerful systems due to increased cache misses and longer load times [11].

As such, the use of C++ templates in embedded systems is a balancing act. On the one hand, they provide powerful tools for improving runtime performance and promoting generic, reusable code. On the other hand, they carry the potential for significant code bloat if used carelessly. Therefore, developers must tread carefully, understanding and taking into account both the benefits and potential drawbacks of templates in their design and coding practices. In this section, the goal is to present an example when the usage of templates leads to code bloat. It is worth noting that the compiler optimization can address many of the code bloat issues associated with templates. A potential source of bloat that may resist optimization is related to large template classes or functions, as well as excessive template instantiation.

As mentioned above, the compiler is very good at optimizing for code bloat, especially if the optimization is set for code size. Implementing a single template class or function is not sufficient to trigger the phenomenon, because these issues occur in much larger codebases. However, it is possible to enforce that the compiler does not optimize out function calls by adding the *noinline* attribute to the function. Note that the *noinline* specifier is not standard C++, but a GNU-specific function attribute. Listing 7.1 shows the example code. A template class is defined in lines 1 through 11 with a member function that is prevented from being considered for inlining (*noinline* attribute in line 6). The class is instantiated for both *int* and *char* types in lines 15 and 19 respectively. The init function is called on both objects in lines 16 and 19.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
template <typename T>
class Example {
private:
    T m_data[10000];
public:
    __attribute__ ((noinline)) void init() {
        for (int i = 0; i < 10000; i++) {
            m_data[i] = i;
        }
    }
};

int main (void) {

    Example<int> example_int;
    example_int.init();

    Example<char> example_char;
    example_char.init();

    while (true) { /* ... */ }
}
```

LISTING 7.1: Example for C++ code bloat.

The compiler generates the function for both types, as shown in Listing 7.2. Examining the mangled names of the functions in lines 1 and 12 reveals that they differ in only a single letter and thus it can determined which one is for the integer type (*_ZN7Example***i***E4initEv* in line 1) and which one is for the character type (*_ZN7Example***c***E4initEv* in line 12). The assumption can be verified by analyzing the assembly code. Line 4 left shifts *r3* by 2 bits, effectively multiplying it by 4. This is a key point because multiplying by 4 suggests that it is indexing an array of integers (since an *int* is 4 bytes). Line 15 stores the byte in *r3* to an array at the position *r3* (offset from base pointer *r0*). The instruction *strb* is used for storing a byte, indicating that this snippet is working with *char* types. That the compiler specializes the templated functions for every type it is used with is expected, but it can have more severe consequences when nested templates or multiple template parameters are used extensively, and the compiler needs to generate a distinct definition for every combination of the template parameters.

```
0800019c <_ZN7ExampleIiE4initEv>:
 800019c:       2300            movs    r3, #0
 800019e:       4a03            ldr     r2, [pc, #12]   ; (80001ac <
    _ZN7ExampleIiE4initEv+0x10>)
 80001a0:       0099            lsls    r1, r3, #2
 80001a2:       5043            str     r3, [r0, r1]
 80001a4:       3301            adds    r3, #1
 80001a6:       4293            cmp     r3, r2
 80001a8:       d1fa            bne.n   80001a0 <_ZN7ExampleIiE4initEv+
    0x4>
 80001aa:       4770            bx      lr
 80001ac:       00002710        .word   0x00002710

080001b0 <_ZN7ExampleIcE4initEv>:
 80001b0:       2300            movs    r3, #0
 80001b2:       4a03            ldr     r2, [pc, #12]   ; (80001c0 <
    _ZN7ExampleIcE4initEv+0x10>)
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
15 80001b4:     54c3        strb   r3, [r0, r3]
16 80001b6:     3301        adds   r3, #1
17 80001b8:     4293        cmp    r3, r2
18 80001ba:     d1fb        bne.n  80001b4 <_ZN7ExampleIcE4initEv+
     0x4>
19 80001bc:     4770        bx     lr
20 80001be:     46c0        nop                    ; (mov r8, r8)
21 80001c0:     00002710    .word  0x00002710
```

LISTING 7.2: Assembly result of the code bloat example.

Code bloat was briefly mentioned during the evaluation of the measurements done in Subsection 6.1.2. The more templated functions are generated into the source code by the generator script, the more functions the compiler needs to generate in the binary. For every combination of template parameters that occurs in the main function where the functions are specialized and called, the compiler generates a new function with the given signature. This is the reason why, as illustrated by Figure 7.1, the size of the C++ binary increases rapidly until every possible function is generated, while the C version increases linearly as the number of function calls increase.

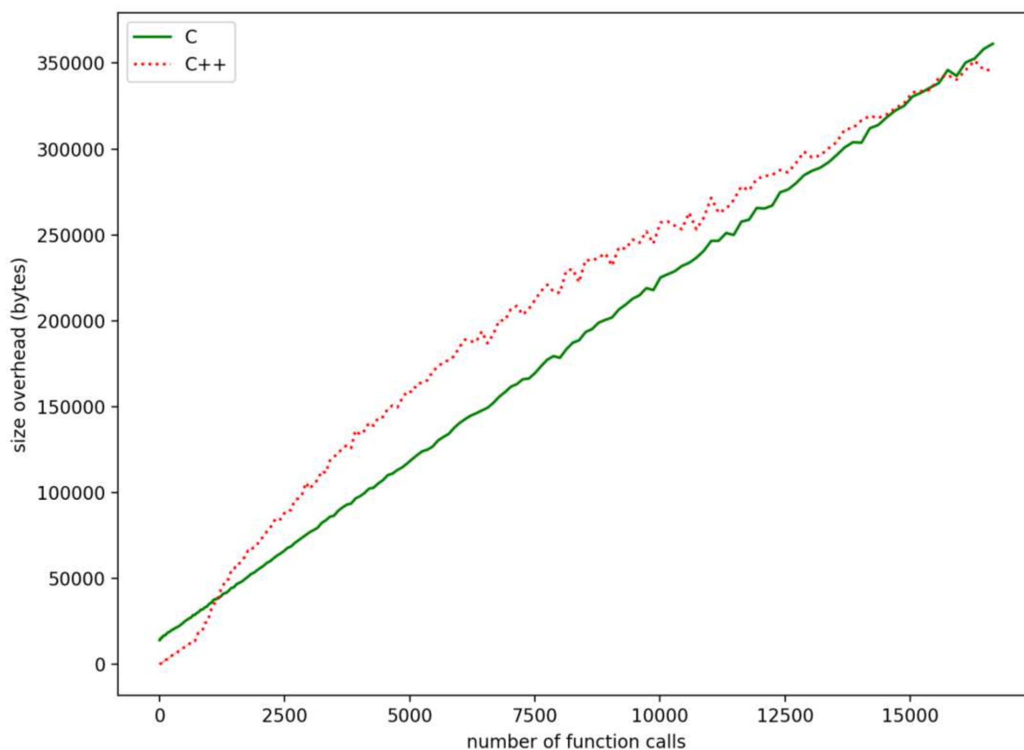



FIGURE 7.1: Binary size overhead for 100 generated functions.

## 7.2 Floating-Point Operations

This section discusses an aspect that cannot be left unmentioned when talking about the runtime performance of embedded systems. This aspect is the usage of floating-point operations in the source code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The FPU, also known as a math coprocessor, is a specialized component of the CPU designed to carry out operations on floating-point numbers efficiently. When present, it can perform arithmetic operations like addition, subtraction, multiplication, division, and more complex operations such as square root and trigonometric calculations much faster than the CPU can using software routines [28].

However, not all embedded systems feature an FPU, either due to cost, power, or physical size constraints. In such systems, any floating-point operation must be performed via software emulation. This emulation is often achieved using complex routines that convert floating-point operations into a series of integer operations that the CPU can handle [11]. The runtime overhead of such operations is not dependent on the language, but rather on the hardware itself.

There are several implications of this. On the one hand, floating-point operations on devices without an FPU incur a significant runtime overhead. A simple operation, such as adding two floating-point numbers, can require dozens or even hundreds of CPU instructions when emulated in software, compared to a single instruction on a device equipped with an FPU. This can drastically slow down the execution of the program and increase power consumption, a critical concern for battery-operated devices [11].

On the other hand, the absence of an FPU can complicate programming and limit the use of certain software tools and libraries. Many mathematical libraries and algorithms are designed with the assumption of hardware floating-point support, and adapting these for software emulation can be a non-trivial task [29].

Given these considerations, embedded system developers must carefully consider the impact of floating-point operations on their designs. If high-precision fractional calculations are necessary, hardware with an FPU should be considered despite the added cost. Alternatively, problems must be reformulated to minimize the use of floating-point operations, using fixed-point arithmetic or integer operations where possible, and ensuring that any necessary floating-point routines are efficiently implemented to mitigate the overhead on FPU-less devices.

Listing 7.3 shows a very simple example code that demonstrates how substantial the effects of floating-point operations can be. It is simple performing three floating-point operations in a row, addition, multiplication and division. This couple of lines of code resulted in a text size of 5696 bytes. As a reminder, the base firmware had a text size of only 744 bytes, and performing register operations in the previous examples resulted in only a slight text size increase. The increase happens, because the compiler introduced the functions for software floating-point operations into the code, such as *__aeabi_dmul*, *__aeabi_ddiv* and *__aeabi_dadd*. In embedded devices with limited memory, it might be necessary to avoid floating-point operations altogether. Even in environments with sufficient memory but with no FPU, it might be worth it to convert floating-point operations to integer arithmetic with sufficient resolution for a given application, because it will substantially improve runtime performance.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
int main (void) {

    volatile double a = 15.5, b = 20.5, c;

    c = a + b;
    a = c * b;
    b = a / c;

    while (true) { /* ... */ }
}
```

LISTING 7.3: Demonstration of the problems with floating-point operations without hardware support.

## 7.3 Hidden Dynamic Memory Allocations

Dynamic memory allocation is a fundamental aspect of most computer systems, allowing for the efficient use of memory resources. It is typically performed using functions such as *malloc* and *free* in C, or operators such as *new* and *delete* in C++. However, in the context of embedded systems, dynamic memory allocation introduces a number of potential issues that can have significant impacts on system performance and reliability.

One notable issue is the impact on the binary size of the embedded application. When dynamic memory allocation functions are used, their code must be included in the binary executable. This increases the size of the binary significantly, which can be problematic in resource-constrained embedded systems where memory is limited. Listing 7.4 shows an example code that demonstrates this issue. The memory is allocated in line 5 using *malloc*, it allocates an array of 100 *uint32_t*s. The text size increases to 1620 bytes, which is more than two times the size of the base firmware, which was 744 bytes.

```
#define SOME_REGISTER (*((volatile uint32_t*)0x22000000))

int main (void) {
    volatile uint32_t* arr = NULL;
    arr = (uint32_t*)malloc(100 * sizeof(uint32_t));

    int index = 0;
    while (1) {
        arr[index] = SOME_REGISTER;
        ++index;
        if (index == 100) index = 0;
    }
}
```

LISTING 7.4: Example of dynamic memory allocation.

An examination of the assembly reveals that functions like *_sbrk*, *_reclaim_reent*, *malloc* and *free* were added to the executable by the compiler. In the compilation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

commands, the flag *–specs=nano.specs* causes *arm-none-eabi-gcc* to link against *newlib-nano*, a smaller variant of *newlib*, optimized for size. This library includes implementations for *malloc*, *free*, *sbrk* and many other standard C functions. However, note that while *newlib* provides default implementations for these functions, the behavior of *malloc* and *sbrk* in particular depend on support from the microprocessor. The STM32CubeIde, for example, provides its own *sbrk* implementation specifically designed for ST microcontrollers. Therefore, when working on a bare metal system and it is unavoidable to use dynamic memory allocation, it is likely necessary to provide an own *sbrk* implementation [11].

The use of dynamic memory allocation can also introduce non-deterministic behavior into an embedded system. Dynamic memory allocation involves complex management of memory resources, which can lead to unpredictable delays as the system searches for a sufficiently large block of free memory. This non-deterministic behavior can be detrimental in hard real-time applications, where predictable, deterministic execution is required to meet strict timing constraints. The unpredictable delays introduced by dynamic memory allocation can lead to missed deadlines, potentially resulting in system failure [11].

Furthermore, the use of dynamic memory allocation introduces the risk of allocation failure. When a program calls *malloc* or *new*, there is always the chance that the system will not have enough free memory to fulfill the request. In such cases, *malloc* and *new* return a null pointer, indicating the allocation failure. Unfortunately, many programs fail to check the return value, assuming that the allocation will always succeed. If the return value is not checked and is subsequently used, it can lead to a program crash or other undefined behavior, undermining the reliability of the system.

Insufficient knowledge of the C++ library features can be problematic in the embedded world, because in many cases, dynamic memory allocation can be embedded in a container or algorithm. Software developers for modern operating systems, such as Linux or Windows, rarely care about the impact of dynamic memory allocation, because it can be considered negligible in those environments. It still can impact the performance, but the computational power and available memory of a PC is in a whole other league when compared to a microcontroller.

Listing 7.5 and Listing 7.6 show two examples that demonstrate this issue. The STL container *array* used in Listing 7.6 is a static container, it takes the size of the array as a template parameter. The container *vector*, used in Listing 7.5, on the other hand, is a container that has dynamic size and relies on the *new* operator internally to allocate the necessary memory. While the two programs seemingly perform the same action, they have a very different impact on an embedded platform. The code in Listing 7.5 increases the code size significantly, because all of the code responsible for dynamic memory allocation is included again. Furthermore, the constructor of the *vector* might even throw an exception if the allocation of the memory is not successful. Similarly, the use of the *string* class, and many other classes from the standard library can cause similar problems.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
#include <cstdint>
#include <vector>

int main (void) {
    int index = 0;
    while (true) {
        std::vector<int> arr (3);
        arr[0] = index;
        arr[3] = arr[2] + 3;
    }
}
```

LISTING 7.5: Usage of std::vector implies dynamic memory allocation.

```
#include <cstdint>
#include <array>

int main (void) {
    int index = 0;
    while (true) {
        std::array<int, 3> arr;
        arr[0] = index;
        arr[3] = arr[2] + 3;
    }
}
```

LISTING 7.6: The std::array container has static size, hence it does not require dynamic memory allocation at runtime.

On the other side, this feature, which might cause unexpected problems to inexperienced developers, is one of the biggest advantages of C++ over C. RAII stands for *Resource Acquisition Is Initialization*, a programming idiom used in several object-oriented, statically-typed programming languages like C++. The principle of RAII is that the acquisition of a resource, which could be memory allocation, a file handle, a lock, or any other system resource, is tied to the lifetime of an object [30]. Just like in the example with the *vector*, when an object is created (initialized), resources are acquired in the constructor. The resources are then released in the object's destructor when the object goes out of scope or is otherwise destroyed. This approach is designed to provide a convenient and reliable means of resource management. RAII means that less manual intervention is required in resource cleanup, and it reduces the chance of errors like memory leaks and dangling pointers.

## 7.4 References

A reference in C++ is a type of variable that acts as an alias to another object or value. Conceptually, a reference behaves like a pointer with two key differences: a reference always refers to an object (it cannot be null), and it cannot be changed to refer to another object once it is initialized. The mechanism of passing parameters by references was introduced in C++, providing an additional level of flexibility and efficiency over C's pass-by-pointer and pass-by-value mechanisms.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

In the context of function argument passing, references enable the function to modify the original variables, similar to pointers. However, due to their non-null and immutable nature, references provide a safer and more user-friendly alternative to pointers. While pointers may lead to potential issues with null or dangling pointers, references alleviate these concerns, promoting safer and more readable code.

Listing 7.7 and Listing 7.8 show the same functionality implemented in C and in C++ respectively. The *noinline* attribute is applied to the function in both cases so that the compiler is forced not to inline the function. The two source codes compile to exactly the same assembly. The assembly shows that in both cases the address of the variable is passed to the function, the value is loaded, incremented and stored. However, the C implementation is not safe, because the function can be called with a null pointer, which would lead to a segmentation fault or undefined behavior.

```
1  __attribute__ ((noinline)) void increment(int* value) {
2      ++(*value);
3  }
4
5  int main (void) {
6      int i = 0;
7      while (1) {
8          increment(&i);
9      }
10 }
```

LISTING 7.7: Example of passing a variable by pointer in C.

```
1  __attribute__ ((noinline)) void increment(int& value) {
2      ++value;
3  }
4
5  int main (void) {
6      int i = 0;
7      while (true) {
8          increment(i);
9      }
10 }
```

LISTING 7.8: Example of passing a variable by reference in C++.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Chapter 8

# Conclusion

I have demonstrated that C++ can indeed be used to build zero-overhead abstractions. I have observed how the runtime performance of code written in C can be optimized using modern C++ compile-time constructs. I have examined how structured projects can be built without adding runtime overhead, how layers upon layers of abstractions get reduced to a series of register operations by the compiler. The demonstrated methods provide a basis for writing effective firmware for embedded systems.

The key conclusion of the thesis is that it is in fact possible and plausible to write efficient code with zero runtime overhead using modern C++. Even if some features, such as dynamic polymorphism, come with inherent runtime overhead, C++ gives flexibility to both the developer and the compiler to choose between compile-time and runtime constructs, while C forces both the developers and the compiler to perform most of the operations at runtime without offering any alternatives. Embedded C++ developers, alongside with the C++ compiler, can create firmware that perfectly balances compile-time and runtime overhead, leading to a faster and more sustainable product.

I personally cannot see any drawbacks of using C++ in embedded projects, but can see many of its advantages over C. I hope that by researching the topic in detail, I could provide sufficient evidence for my case for the use of modern C++ in embedded systems.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Appendix A

# Startup File

```
.syntax unified
.cpu cortex-m0
.fpu softvfp
.thumb

.global g_pfnVectors
.global Default_Handler

.word _sidata
.word _sdata
.word _edata
.word _sbss
.word _ebss

.section .text.Reset_Handler
.weak Reset_Handler
.type Reset_Handler, %function

Reset_Handler:
    ldr   r0, =_estack
    mov   sp, r0
    ldr r0, =_sdata
    ldr r1, =_edata
    ldr r2, =_sidata
    movs r3, #0
    b LoopCopyDataInit

CopyDataInit:
    ldr r4, [r2, r3]
    str r4, [r0, r3]
    adds r3, r3, #4

LoopCopyDataInit:
    adds r4, r0, r3
    cmp r4, r1
    bcc CopyDataInit

    ldr r2, =_sbss
    ldr r4, =_ebss
    movs r3, #0
    b LoopFillZerobss

FillZerobss:
    str  r3, [r2]
    adds r2, r2, #4
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
46
47 LoopFillZerobss:
48     cmp r2, r4
49     bcc FillZerobss
50
51     bl __libc_init_array
52     bl main
53
54 LoopForever:
55     b LoopForever
56
57 .size Reset_Handler, .-Reset_Handler
58
59
60
61 .section .text.Default_Handler,"ax",%progbits
62 Default_Handler:
63 Infinite_Loop:
64     b Infinite_Loop
65     .size Default_Handler, .-Default_Handler
66
67
68 .section .isr_vector,"a",%progbits
69 .type g_pfnVectors, %object
70 .size g_pfnVectors, .-g_pfnVectors
71
72 g_pfnVectors:
73   .word  _estack
74   .word  Reset_Handler
75   .word  NMI_Handler
76   .word  HardFault_Handler
77   .word  0
78   .word  0
79   .word  0
80   .word  0
81   .word  0
82   .word  0
83   .word  0
84   .word  SVC_Handler
85   .word  0
86   .word  0
87   .word  PendSV_Handler
88   .word  SysTick_Handler
89   .word  WWDG_IRQHandler
90   .word  0
91   .word  RTC_IRQHandler
92   .word  FLASH_IRQHandler
93   .word  RCC_IRQHandler
94   .word  EXTI0_1_IRQHandler
95   .word  EXTI2_3_IRQHandler
96   .word  EXTI4_15_IRQHandler
97   .word  0
98   .word  DMA1_Channel1_IRQHandler
99   .word  DMA1_Channel2_3_IRQHandler
100  .word  DMA1_Channel4_5_IRQHandler
101  .word  ADC1_IRQHandler
102  .word  TIM1_BRK_UP_TRG_COM_IRQHandler
103  .word  TIM1_CC_IRQHandler
104  .word  0
105  .word  TIM3_IRQHandler
106  .word  0
107  .word  0
108  .word  TIM14_IRQHandler
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
109  .word   0
110  .word   TIM16_IRQHandler
111  .word   TIM17_IRQHandler
112  .word   I2C1_IRQHandler
113  .word   0
114  .word   SPI1_IRQHandler
115  .word   0
116  .word   USART1_IRQHandler
117  .word   0
118  .word   0
119  .word   0
120  .word   0
121
122
123  .weak       NMI_Handler
124  .thumb_set  NMI_Handler,Default_Handler
125
126  .weak       HardFault_Handler
127  .thumb_set  HardFault_Handler,Default_Handler
128
129  .weak       SVC_Handler
130  .thumb_set  SVC_Handler,Default_Handler
131
132  .weak       PendSV_Handler
133  .thumb_set  PendSV_Handler,Default_Handler
134
135  .weak       SysTick_Handler
136  .thumb_set  SysTick_Handler,Default_Handler
137
138  .weak       WWDG_IRQHandler
139  .thumb_set  WWDG_IRQHandler,Default_Handler
140
141  .weak       RTC_IRQHandler
142  .thumb_set  RTC_IRQHandler,Default_Handler
143
144  .weak       FLASH_IRQHandler
145  .thumb_set  FLASH_IRQHandler,Default_Handler
146
147  .weak       RCC_IRQHandler
148  .thumb_set  RCC_IRQHandler,Default_Handler
149
150  .weak       EXTI0_1_IRQHandler
151  .thumb_set  EXTI0_1_IRQHandler,Default_Handler
152
153  .weak       EXTI2_3_IRQHandler
154  .thumb_set  EXTI2_3_IRQHandler,Default_Handler
155
156  .weak       EXTI4_15_IRQHandler
157  .thumb_set  EXTI4_15_IRQHandler,Default_Handler
158
159  .weak       DMA1_Channel1_IRQHandler
160  .thumb_set  DMA1_Channel1_IRQHandler,Default_Handler
161
162  .weak       DMA1_Channel2_3_IRQHandler
163  .thumb_set  DMA1_Channel2_3_IRQHandler,Default_Handler
164
165  .weak       DMA1_Channel4_5_IRQHandler
166  .thumb_set  DMA1_Channel4_5_IRQHandler,Default_Handler
167
168  .weak       ADC1_IRQHandler
169  .thumb_set  ADC1_IRQHandler,Default_Handler
170
171  .weak       TIM1_BRK_UP_TRG_COM_IRQHandler
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
172  .thumb_set TIM1_BRK_UP_TRG_COM_IRQHandler,Default_Handler
173
174  .weak      TIM1_CC_IRQHandler
175  .thumb_set TIM1_CC_IRQHandler,Default_Handler
176
177  .weak      TIM3_IRQHandler
178  .thumb_set TIM3_IRQHandler,Default_Handler
179
180  .weak      TIM14_IRQHandler
181  .thumb_set TIM14_IRQHandler,Default_Handler
182
183  .weak      TIM16_IRQHandler
184  .thumb_set TIM16_IRQHandler,Default_Handler
185
186  .weak      TIM17_IRQHandler
187  .thumb_set TIM17_IRQHandler,Default_Handler
188
189  .weak      I2C1_IRQHandler
190  .thumb_set I2C1_IRQHandler,Default_Handler
191
192  .weak      SPI1_IRQHandler
193  .thumb_set SPI1_IRQHandler,Default_Handler
194
195  .weak      USART1_IRQHandler
196  .thumb_set USART1_IRQHandler,Default_Handler
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Appendix B

# Linker Script

```
ENTRY(Reset_Handler)

_estack = ORIGIN(RAM) + LENGTH(RAM);

_Min_Heap_Size = 0x200 ;
_Min_Stack_Size = 0x400 ;

MEMORY
{
  RAM     (xrw)     : ORIGIN = 0x20000000,   LENGTH = 4K
  FLASH    (rx)     : ORIGIN = 0x8000000,    LENGTH = 16K
}

SECTIONS
{
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector))
    . = ALIGN(4);
  } >FLASH

  .text :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
    *(.glue_7)
    *(.glue_7t)
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;
  } >FLASH

  .rodata :
  {
    . = ALIGN(4);
    *(.rodata)
    *(.rodata*)
    . = ALIGN(4);
  } >FLASH
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
46
47  .ARM.extab   : {
48    . = ALIGN(4);
49    *(.ARM.extab* .gnu.linkonce.armextab.*)
50    . = ALIGN(4);
51  } >FLASH
52
53  .ARM : {
54    . = ALIGN(4);
55    __exidx_start = .;
56    *(.ARM.exidx*)
57    __exidx_end = .;
58    . = ALIGN(4);
59  } >FLASH
60
61  .preinit_array     :
62  {
63    . = ALIGN(4);
64    PROVIDE_HIDDEN (__preinit_array_start = .);
65    KEEP (*(.preinit_array*))
66    PROVIDE_HIDDEN (__preinit_array_end = .);
67    . = ALIGN(4);
68  } >FLASH
69
70  .init_array :
71  {
72    . = ALIGN(4);
73    PROVIDE_HIDDEN (__init_array_start = .);
74    KEEP (*(SORT(.init_array.*)))
75    KEEP (*(.init_array*))
76    PROVIDE_HIDDEN (__init_array_end = .);
77    . = ALIGN(4);
78  } >FLASH
79
80  .fini_array :
81  {
82    . = ALIGN(4);
83    PROVIDE_HIDDEN (__fini_array_start = .);
84    KEEP (*(SORT(.fini_array.*)))
85    KEEP (*(.fini_array*))
86    PROVIDE_HIDDEN (__fini_array_end = .);
87    . = ALIGN(4);
88  } >FLASH
89
90  _sidata = LOADADDR(.data);
91
92  .data :
93  {
94    . = ALIGN(4);
95    _sdata = .;
96    *(.data)
97    *(.data*)
98    *(.RamFunc)
99    *(.RamFunc*)
100
101   . = ALIGN(4);
102   _edata = .;
103
104 } >RAM AT> FLASH
105
106 . = ALIGN(4);
107 .bss :
108 {
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
109     _sbss = .;
110     __bss_start__ = _sbss;
111     *(.bss)
112     *(.bss*)
113     *(COMMON)
114 
115     . = ALIGN(4);
116     _ebss = .;
117     __bss_end__ = _ebss;
118   } >RAM
119 
120   ._user_heap_stack :
121   {
122     . = ALIGN(8);
123     PROVIDE ( end = . );
124     PROVIDE ( _end = . );
125     . = . + _Min_Heap_Size;
126     . = . + _Min_Stack_Size;
127     . = ALIGN(8);
128   } >RAM
129 
130   /DISCARD/ :
131   {
132     libc.a ( * )
133     libm.a ( * )
134     libgcc.a ( * )
135   }
136 
137   .ARM.attributes 0 : { *(.ARM.attributes) }
138 }
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Appendix C

# Makefile

## C.1 C Makefile

```
all: prog

prog:
	arm-none-eabi-gcc -mcpu=cortex-m0 -c -specs=nano.specs -mfloat-abi=soft -mthumb -o startup.o startup.s
	arm-none-eabi-gcc -mcpu=cortex-m0 -c -std=gnu11 -Os -ffunction-sections -fdata-sections -Wall --specs=nano.specs -mfloat-abi=soft -mthumb -I. -o main.o main.c
	arm-none-eabi-gcc -mcpu=cortex-m0 -o main.elf main.o startup.o -T linker_script.ld --specs=nosys.specs --specs=nano.specs -mfloat-abi=soft -mthumb
	arm-none-eabi-objcopy -O binary main.elf main.bin

objdump:
	arm-none-eabi-objdump -h -S main.elf

size:
	arm-none-eabi-size main.elf

flash:
	st-flash write main.bin 0x08000000

reset:
	st-flash reset

clean:
	@if [ -f main.o ]; then rm main.o; fi
	@if [ -f main.s ]; then rm main.s; fi
	@if [ -f startup.o ]; then rm startup.o; fi
	@if [ -f main.elf ]; then rm main.elf; fi
	@if [ -f main.bin ]; then rm main.bin; fi
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## C.2 C++ Makefile

```
all: prog

prog:
	arm-none-eabi-g++ -mcpu=cortex-m0 -c -specs=nano.specs -mfloat-abi=soft -mthumb -o startup.o startup.s
	arm-none-eabi-g++ -mcpu=cortex-m0 -c -std=c++20 -Os -ffunction-sections -fdata-sections -Wall --specs=nano.specs -mfloat-abi=soft -mthumb -I. -o main.o main.cpp
	arm-none-eabi-g++ -mcpu=cortex-m0 -o main.elf main.o startup.o -T linker_script.ld --specs=nosys.specs --specs=nano.specs -mfloat-abi=soft -mthumb
	arm-none-eabi-objcopy -O binary main.elf main.bin

objdump:
	arm-none-eabi-objdump -h -S main.elf

size:
	arm-none-eabi-size main.elf

flash:
	st-flash write main.bin 0x08000000

reset:
	st-flash reset

clean:
	@if [ -f main.o ]; then rm main.o; fi
	@if [ -f main.s ]; then rm main.s; fi
	@if [ -f startup.o ]; then rm startup.o; fi
	@if [ -f main.elf ]; then rm main.elf; fi
	@if [ -f main.bin ]; then rm main.bin; fi
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Appendix D

# Benchmark Toolchain

## D.1 Base main.c Jinja Template

```
#include <stdint.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

uint32_t m_memory [{{array_size}}];

int main (void) {

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start);
    uint64_t elapsed_ns = 0;

    clock_gettime(CLOCK_REALTIME, &stop);

    elapsed_ns = (stop.tv_sec - start.tv_sec) * (uint64_t)1e9;
    elapsed_ns += stop.tv_nsec - start.tv_nsec;

    printf("%lu\n", elapsed_ns);

    return 0;
}
```

## D.2 main.c Jinja Template

```
#include <stdint.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

uint32_t m_memory [{{array_size}}];


{% for i in range(array_size) -%}
#define ELEM_{{i}} (*((volatile uint32_t*)&m_memory[{{i}}]))
{% endfor %}

{% for i in range(num_choices) -%}
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
#define CHOICE_{{i}} ((uint16_t){{ "0x%0x" | format(i|int) }})
{% endfor %}

{% for i in range(num_modes) -%}
#define MODE_{{i}} ({{ "0x%0x" | format(i|int) }}U)
{% endfor %}

{% for i in range(num_masks) -%}
#define MASK_{{i}} ({{ "0x%0x" | format(i|int) }}U)
{% endfor %}

#define IS_CHOICE(X) ((X) < 0xFFFF && (X) >= 0)
#define IS_MODE(X) ({% for i in range(num_modes) %}((X) == MODE_{{i}}) {% if not loop.last %} || {% endif %}{% endfor %})

typedef struct {
    uint32_t choice ;
    uint32_t mode ;
} A_Struct ;

{% for func in funcs -%}
void DoStuff_{{func["index"]}}(A_Struct* m_struct) {
    uint32_t temp;

    if (!IS_CHOICE(m_struct->choice)) { return; }
    if (!IS_MODE(m_struct->mode)) { return; }

    if (m_struct->mode == MODE_{{func["mode_index"]}}) {
        temp = ELEM_{{func["elem_index_0"]}};
        temp &= ~(MASK_{{func["mask_index_0"]}} << (m_struct->mode * 2u));
        temp |= (MASK_{{func["mask_index_1"]}} << (m_struct->choice * 2u));
        ELEM_{{func["elem_index_0"]}} = temp;

        temp = ELEM_{{func["elem_index_1"]}};
        temp &= ~(MASK_{{func["mask_index_1"]}} << m_struct->choice);
        temp |= (((m_struct->mode & MASK_{{func["mask_index_1"]}}) >> 4u) << m_struct->mode);
        ELEM_{{func["elem_index_1"]}} = temp;
    }

    temp = ELEM_{{func["elem_index_1"]}};
    temp &= ~(MASK_{{func["mask_index_1"]}} << (m_struct->choice * 2u));
    temp |= ((m_struct->mode & MASK_{{func["mask_index_1"]}}) << (m_struct->choice * 2u));
    ELEM_{{func["elem_index_1"]}} = temp;
}
{% endfor %}

int main (void) {

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start);
    uint64_t elapsed_ns = 0;

    {% for st in a_structs -%}
    {
        A_Struct m_struct;
        m_struct.choice = CHOICE_{{st["choice"]}};
        m_struct.mode = MODE_{{st["mode"]}};
        DoStuff_{{st["func"]}}(&m_struct) ;
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
    }
    {% endfor %}

    clock_gettime(CLOCK_REALTIME, &stop);

    elapsed_ns = (stop.tv_sec - start.tv_sec) * (uint64_t)1e9;
    elapsed_ns += stop.tv_nsec - start.tv_nsec;

    printf("%lu\n", elapsed_ns);

    return 0;
}
```

## D.3 Base main.cpp Jinja Template

```
#include <stdint.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

uint32_t m_memory [{{array_size}}];

int main (void) {

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start);
    uint64_t elapsed_ns = 0;

    clock_gettime(CLOCK_REALTIME, &stop);

    elapsed_ns = (stop.tv_sec - start.tv_sec) * (uint64_t)1e9;
    elapsed_ns += stop.tv_nsec - start.tv_nsec;

    printf("%lu\n", elapsed_ns);

    return 0;
}
```

## D.4 main.cpp Jinja Template

```
#include <cstdint>
#include <concepts>
#include <type_traits>
#include <iostream>
#include <unistd.h>

std::uint32_t m_memory [{{array_size}}];


{% for i in range(array_size) -%}
#define ELEM_{{i}} (*((volatile uint32_t*)&m_memory[{{i}}]))
{% endfor %}

enum class choices : std::uint16_t {
{%- for i in range(num_choices) %}
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
16     choice_{{i}} = {{ "0x%0x" | format(i|int) }}{% if not loop.last
          %},{% endif %}
17 {%- endfor %}
18 };
19
20 enum class modes : std::uint32_t {
21 {%- for i in range(num_modes) %}
22     mode_{{i}} = {{ "0x%0x" | format(i|int) }}{% if not loop.last %},{%
           endif %}
23 {%- endfor %}
24 };
25
26 enum class masks : std::uint32_t {
27 {%- for i in range(num_masks) %}
28     mask_{{i}} = {{ "0x%0x" | format(i|int) }}{% if not loop.last %},{%
           endif %}
29 {%- endfor %}
30 };
31
32 consteval std::uint32_t calculate_mask_0(masks mask, modes mode) {
33     return static_cast<std::uint32_t>(mask) << (static_cast<std::
          uint32_t>(mode) * 2u);
34 }
35
36 consteval std::uint32_t calculate_mask_1(masks mask, choices choice) {
37     return static_cast<std::uint32_t>(mask) << (static_cast<std::
          uint32_t>(choice) * 2u);
38 }
39
40 consteval std::uint32_t calculate_mask_2(masks mask, choices choice) {
41     return static_cast<std::uint32_t>(mask) << static_cast<std::
          uint32_t>(choice);
42 }
43
44 consteval std::uint32_t calculate_mask_3(masks mask, modes mode) {
45     return ((static_cast<std::uint32_t>(mode) & static_cast<std::
          uint32_t>(mask)) >> 4) << static_cast<std::uint32_t>(mode);
46 }
47
48 consteval std::uint32_t calculate_mask_4(masks mask, modes mode,
       choices choice) {
49     return (static_cast<std::uint32_t>(mode) & static_cast<std::
          uint32_t>(mask)) << (static_cast<std::uint32_t>(choice) * 2);
50 }
51
52 template <choices choice>
53 concept is_valid_choice = (static_cast<std::uint16_t>(choice) < 0xFFFF)
       ;
54
55 template <modes mode>
56 concept is_valid_mode = ({% for i in range(num_modes) %}(mode == modes
       ::mode_{{i}}){% if not loop.last %} || {% endif %}{% endfor %});
57
58 {% for func in funcs -%}
59 template <choices choice, modes mode>
60 requires (is_valid_choice<choice> && is_valid_mode<mode>)
61 void DoStuff_{{func["index"]}}() {
62     uint32_t temp;
63
64     if constexpr (mode == modes::mode_{{func["mode_index"]}}) {
65         temp = ELEM_{{func["elem_index_0"]}};
66         temp &= ~calculate_mask_0(masks::mask_{{func["mask_index_0"]}},
              mode);
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
        temp |= calculate_mask_1(masks::mask_{{func["mask_index_1"]}},
            choice);
        ELEM_{{func["elem_index_0"]}} = temp;

        temp = ELEM_{{func["elem_index_1"]}};
        temp &= ~calculate_mask_2(masks::mask_{{func["mask_index_1"]}},
             choice);
        temp |= calculate_mask_3(masks::mask_{{func["mask_index_1"]}},
            mode);
        ELEM_{{func["elem_index_1"]}} = temp;
    }

    temp = ELEM_{{func["elem_index_1"]}};
    temp &= ~calculate_mask_1(masks::mask_{{func["mask_index_1"]}},
        choice);
    temp |= calculate_mask_4(masks::mask_{{func["mask_index_1"]}}, mode
        , choice);
    ELEM_{{func["elem_index_1"]}} = temp;
}
{% endfor %}

int main (void) {

    struct timespec start, stop;
    clock_gettime(CLOCK_REALTIME, &start);
    uint64_t elapsed_ns = 0;

    {% for st in a_structs -%}
    {
        DoStuff_{{st["func"]}}<choices::choice_{{st["choice"]}}, modes
            ::mode_{{st["mode"]}}>();
    }
    {% endfor %}

    clock_gettime(CLOCK_REALTIME, &stop);

    elapsed_ns = (stop.tv_sec - start.tv_sec) * (uint64_t)1e9;
    elapsed_ns += stop.tv_nsec - start.tv_nsec;

    printf("%lu\n", elapsed_ns);
    return 0;
}
```

## D.5 JSON Configuration File Example

```
{
    "array_size":20,
    "num_choices":10,
    "num_modes":10,
    "num_masks":15,
    "num_funcs":10,
    "num_calls":91
}
```

## D.6 Generator Script

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```python
import jinja2
import json
import random
import sys

N = sys.argv[1]

data_file = 'data_' + N + '.json'
base_c = 'c/base_' + N + '.c'
main_c = 'c/main_' + N + '.c'
base_cpp = 'cpp/base_' + N + '.cpp'
main_cpp = 'cpp/main_' + N + '.cpp'

data = json.load(open(data_file))
array_size = data['array_size']
num_choices= data['num_choices']
num_modes = data['num_modes']
num_masks = data['num_masks']
num_calls = data['num_calls']
num_funcs = data['num_funcs']

templateLoader = jinja2.FileSystemLoader(searchpath="./")
templateEnv = jinja2.Environment(loader=templateLoader)

# create functions that use random masks and elements

funcs = []
for i in range(num_funcs) :
    funcs.append({
        "index": i,
        "mode_index": random.randint(0, num_modes - 1),
        "mask_index_0" : random.randint(0, num_masks - 1),
        "mask_index_1" : random.randint(0, num_masks - 1),
        "elem_index_0" : random.randint(0, array_size - 1),
        "elem_index_1" : random.randint(0, array_size - 1)
    })

# create structs that contain random choices and modes

a_structs = []
for i in range(num_calls) :
    a_structs.append({
        "choice": random.randint(0, num_choices - 1),
        "mode": random.randint(0, num_modes - 1),
        "func": random.randint(0, num_funcs - 1)
    })

c_template = templateEnv.get_template("c/main.jinja")
c_base_template = templateEnv.get_template("c/base.jinja")
c_main = c_template.render(
        array_size=array_size,
        num_choices=num_choices,
        num_modes=num_modes,
        num_masks=num_masks,
        a_structs=a_structs,
        funcs=funcs
    )
c_base = c_base_template.render(array_size=array_size)
with open(main_c, 'w') as fp:
    fp.write(c_main)
with open(base_c, 'w') as fp:
    fp.write(c_base)

```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
64 cpp_template = templateEnv.get_template("cpp/main.jinja")
65 cpp_base_template = templateEnv.get_template("cpp/base.jinja")
66 cpp_main = cpp_template.render(
67         array_size=array_size,
68         num_choices=num_choices,
69         num_modes=num_modes,
70         num_masks=num_masks,
71         a_structs=a_structs,
72         funcs=funcs
73     )
74 cpp_base = cpp_base_template.render(array_size=array_size)
75 with open(main_cpp, 'w') as fp:
76     fp.write(cpp_main)
77 with open(base_cpp, 'w') as fp:
78     fp.write(cpp_base)
```

## D.7 Plotting Script

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4 import sys
5
6 N = sys.argv[1]
7
8 my_dpi=200
9
10 data = pd.read_csv("data_" + N + ".csv")
11
12 fig, axs = plt.subplots(2, 2)
13
14 fig.suptitle('evaluation')
15 fig.set_size_inches(20, 15)
16
17 axs[0, 0].plot(data['num_calls'].values, data['c_size_o'].values, label
      ="C", color='g', linestyle='solid', marker=',')
18 axs[0, 0].plot(data['num_calls'].values, data['cpp_size_o'].values,
      label="C++", color='r', linestyle='solid', marker=',')
19 axs[0, 0].set(xlabel='number␣of␣function␣calls', ylabel='size␣overhead␣
      (bytes)')
20 axs[0, 0].legend(loc="upper␣left")
21
22 axs[0, 1].plot(data['num_calls'].values, data['c_comp_t'].values, label
      ="C", color='g', linestyle='solid', marker=',')
23 axs[0, 1].plot(data['num_calls'].values, data['cpp_comp_t'].values,
      label="C++", color='r', linestyle='solid', marker=',')
24 axs[0, 1].set(xlabel='number␣of␣function␣calls', ylabel='compilation␣
      time␣(s)')
25 axs[0, 1].legend(loc="upper␣left")
26
27 axs[1, 0].plot(data['num_calls'].values, data['c_link_t'].values, label
      ="C", color='g', linestyle='solid', marker=',')
28 axs[1, 0].plot(data['num_calls'].values, data['cpp_link_t'].values,
      label="C++", color='r', linestyle='solid', marker=',')
29 axs[1, 0].set(xlabel='number␣of␣function␣calls', ylabel='link␣time␣(s)'
      )
30 axs[1, 0].legend(loc="upper␣left")
31
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
axs[1, 1].plot(data['num_calls'].values, data['c_exec_t'].values, label
    ="C", color='g', linestyle='solid', marker=',')
axs[1, 1].plot(data['num_calls'].values, data['cpp_exec_t'].values,
    label="C++", color='r', linestyle='solid', marker=',')
axs[1, 1].set(xlabel='number of function calls', ylabel='execution time
    (ns)')
axs[1, 1].legend(loc="upper left")


#plt.show()
plt.savefig('plt_' + N + '.png', dpi=my_dpi)
```

## D.8 Shell Script for Running Benchmark for a Specific Configuration

```
#!/bin/bash

export TIMEFORMAT=%3R

NUM_EPOCHS=5

ARRAY_SIZE=20
NUM_CHOICES=10
NUM_MODES=10
NUM_MASKS=15

NUM_FUNCS=$1
CSV_FILE="data_$NUM_FUNCS.csv"
JSON_FILE="data_$NUM_FUNCS.json"

BASE_C_SRC="c/base_$NUM_FUNCS.c"
BASE_C_BIN="base_c_$NUM_FUNCS"
BASE_CPP_SRC="cpp/base_$NUM_FUNCS.cpp"
BASE_CPP_BIN="base_cpp_$NUM_FUNCS"

MAIN_C_SRC="c/main_$NUM_FUNCS.c"
MAIN_C_OBJ="main_c_$NUM_FUNCS.o"
MAIN_C_BIN="main_c_$NUM_FUNCS"

MAIN_CPP_SRC="cpp/main_$NUM_FUNCS.cpp"
MAIN_CPP_OBJ="main_cpp_$NUM_FUNCS.o"
MAIN_CPP_BIN="main_cpp_$NUM_FUNCS"

echo "num_calls,c_size_o,cpp_size_o,c_comp_t,cpp_comp_t,c_link_t,
    cpp_link_t,c_exec_t,cpp_exec_t" > $CSV_FILE

NUM_CALLS=0
INCREMENT=0
STOP_AT=40000

while true; do

    NUM_CALLS=$((NUM_CALLS + INCREMENT))
    INCREMENT=$((INCREMENT + 1))

    if [ $NUM_CALLS -gt $STOP_AT ]; then
        break
    fi
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
43
44    C_COMPILE_TIME_AVG=0
45    C_LINK_TIME_AVG=0
46    C_SIZE=0
47    C_EXEC_TIME_AVG=0
48    BASE_C_SIZE=0
49    CPP_COMPILE_TIME_AVG=0
50    CPP_LINK_TIME_AVG=0
51    CPP_SIZE=0
52    CPP_EXEC_TIME_AVG=0
53    BASE_CPP_SIZE=0
54
55    for i in $(seq 1 $NUM_EPOCHS); do
56
57        printf '{"array_size":' > $JSON_FILE
58        printf $ARRAY_SIZE >> $JSON_FILE
59        printf ',"num_choices":' >> $JSON_FILE
60        printf $NUM_CHOICES >> $JSON_FILE
61        printf ',"num_modes":' >> $JSON_FILE
62        printf $NUM_MODES >> $JSON_FILE
63        printf ',"num_masks":' >> $JSON_FILE
64        printf $NUM_MASKS >> $JSON_FILE
65        printf ',"num_funcs":' >> $JSON_FILE
66        printf $NUM_FUNCS >> $JSON_FILE
67        printf ',"num_calls":' >> $JSON_FILE
68        printf $NUM_CALLS >> $JSON_FILE
69        printf '}' >> $JSON_FILE
70
71        python3 generate.py $NUM_FUNCS
72
73        gcc -Os -Wall $BASE_C_SRC -o $BASE_C_BIN
74        BASE_C_SIZE=`size $BASE_C_BIN | tail -n1 | awk '{print␣$1}'`
75
76        g++ -Os -std=c++20 -Wall $BASE_CPP_SRC -o $BASE_CPP_BIN
77        BASE_CPP_SIZE=`size $BASE_CPP_BIN | tail -n1 | awk '{print␣$1}'
          `
78
79        C_COMPILE_TIME=`(time gcc -c -Os $MAIN_C_SRC -o $MAIN_C_OBJ)
          2>&1 >/dev/null`
80        C_LINK_TIME=`(time gcc -Os -Wall $MAIN_C_OBJ -o $MAIN_C_BIN)
          2>&1 >/dev/null`
81
82        C_SIZE=`size $MAIN_C_BIN | tail -n1 | awk '{print␣$1}'`
83
84        C_EXEC_TIME=`./$MAIN_C_BIN`
85
86
87        CPP_COMPILE_TIME=`(time g++ -std=c++20 -c -Os $MAIN_CPP_SRC -o
          $MAIN_CPP_OBJ) 2>&1 >/dev/null`
88        CPP_LINK_TIME=`(time g++ -std=c++20 -Os -Wall $MAIN_CPP_OBJ -o
          $MAIN_CPP_BIN) 2>&1 >/dev/null`
89
90        CPP_SIZE=`size $MAIN_CPP_BIN | tail -n1 | awk '{print␣$1}'`
91
92        CPP_EXEC_TIME=`./$MAIN_CPP_BIN`
93
94
95        C_COMPILE_TIME_AVG=`echo $C_COMPILE_TIME_AVG + $C_COMPILE_TIME
          | bc`
96        C_LINK_TIME_AVG=`echo $C_LINK_TIME_AVG + $C_LINK_TIME | bc`
97        C_EXEC_TIME_AVG=`echo $C_EXEC_TIME_AVG + $C_EXEC_TIME | bc`
98        CPP_COMPILE_TIME_AVG=`echo $CPP_COMPILE_TIME_AVG +
          $CPP_COMPILE_TIME | bc`
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
99          CPP_LINK_TIME_AVG=`echo $CPP_LINK_TIME_AVG + $CPP_LINK_TIME |
                bc`
100         CPP_EXEC_TIME_AVG=`echo $CPP_EXEC_TIME_AVG + $CPP_EXEC_TIME |
                bc`
101
102         rm $MAIN_C_BIN
103         rm $MAIN_C_OBJ
104         rm $MAIN_CPP_BIN
105         rm $MAIN_CPP_OBJ
106         rm $MAIN_C_SRC
107         rm $MAIN_CPP_SRC
108         rm $BASE_C_SRC
109         rm $BASE_CPP_SRC
110         rm $BASE_C_BIN
111         rm $BASE_CPP_BIN
112         rm $JSON_FILE
113
114     done
115
116     C_COMPILE_TIME_AVG=`echo $C_COMPILE_TIME_AVG / $NUM_EPOCHS | bc -l`
117     C_LINK_TIME_AVG=`echo $C_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
118     C_EXEC_TIME_AVG=`echo $C_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
119     CPP_COMPILE_TIME_AVG=`echo $CPP_COMPILE_TIME_AVG / $NUM_EPOCHS | bc
            -l`
120     CPP_LINK_TIME_AVG=`echo $CPP_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
121     CPP_EXEC_TIME_AVG=`echo $CPP_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
122
123     echo "$NUM_CALLS,$((C_SIZE-BASE_C_SIZE)),$((CPP_SIZE-BASE_CPP_SIZE)
        ),$C_COMPILE_TIME_AVG,$CPP_COMPILE_TIME_AVG,$C_LINK_TIME_AVG,
        $CPP_LINK_TIME_AVG,$C_EXEC_TIME_AVG,$CPP_EXEC_TIME_AVG" >>
        $CSV_FILE
124
125 done
126
127 python3 plot.py $NUM_FUNCS
```

## D.9 Shell Script for Running the Benchmark

```
1 #!/bin/bash
2
3 taskset -c 2 ./test.sh 1 &
4 taskset -c 3 ./test.sh 20 &
5 taskset -c 4 ./test.sh 50 &
6 taskset -c 5 ./test.sh 100 &
```

## D.10 Generator Script for Abstraction Overhead Measurements

```
1 import json
2 import random
3 import sys
4
5 num_functions = int(sys.argv[1])
6 num_calls = int(sys.argv[2])
7
8 MAX_REGISTERS = 20
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
9  MAX_MODES = 10
10 MAX_CHOICES = 10
11 MAX_MASKS = 10
12 BRANCHING_THRESHOLD = 0.5
13 STOP_THRESHOLD = 0.3
14
15 def include_header (file, headers) :
16     file.write("/*␣includes␣*/\n")
17     for header in headers :
18         file.write("#include␣<" + header + ">\n")
19     file.write("\n")
20
21 def add_enum_class (file, name, prefix, underlying_type, num_enums) :
22     file.write("/*␣enum␣" + name + "␣*/\n")
23     file.write("enum␣class␣" + name + "␣:␣" + underlying_type + "␣{\n")
24     for i in range(num_enums) :
25         file.write("␣␣␣␣" + prefix + "_" + str(i) + "␣=␣" + str(i))
26         if (i == num_enums - 1) :
27             file.write("\n")
28         else :
29             file.write(",\n")
30     file.write("};\n")
31     file.write("template␣<" + name + "␣val>␣concept␣is_valid_" + prefix
        + "␣=␣(")
32     for i in range(num_enums) :
33         file.write("(val␣==␣" + name + "::" + prefix + "_" + str(i) + "
            )")
34         if (i < num_enums - 1) :
35             file.write("␣||␣")
36     file.write(");\n\n")
37
38 def add_enum (file, name, prefix, num_enums) :
39     file.write("/*␣enum␣" + name + "␣*/\n")
40     file.write("typedef␣enum␣{\n")
41     for i in range(num_enums) :
42         file.write("␣␣␣␣" + prefix + "_" + str(i) + "␣=␣" + str(i))
43         if (i == num_enums - 1) :
44             file.write("\n")
45         else :
46             file.write(",\n")
47     file.write("}␣" + name + ";\n")
48     file.write("#define␣IS_" + name + "(X)␣(")
49     for i in range(num_enums) :
50         file.write("((X)␣==␣" + prefix + "_" + str(i) + ")")
51         if (i < num_enums - 1) :
52             file.write("␣||␣")
53     file.write(")\n\n")
54
55 def define_reg_base_class (file) :
56     file.write("template<std::uint32_t␣address>\n")
57     file.write("class␣Register␣{\n")
58     file.write("private:\n")
59     file.write("␣␣␣␣void␣set␣(std::uint32_t␣val)␣const␣{\n")
60     file.write("␣␣␣␣␣␣␣␣*(reinterpret_cast<volatile␣std::uint32_t␣*>(&
        m_memory[address]))␣=␣val;\n")
61     file.write("␣␣␣␣}\n")
62     file.write("\n")
63     file.write("␣␣␣␣std::uint32_t␣get␣()␣const␣{\n")
64     file.write("␣␣␣␣␣␣␣␣return␣*(reinterpret_cast<volatile␣std::
        uint32_t␣*>(&m_memory[address]));\n")
65     file.write("␣␣␣␣}\n")
66     file.write("protected:\n")
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
67      file.write("␣␣␣␣void␣set␣(std::uint32_t␣value,␣std::uint32_t␣
            bitmask)␣const␣{\n")
68      file.write("␣␣␣␣␣␣␣␣std::uint32_t␣tmp␣=␣this->get();\n")
69      file.write("␣␣␣␣␣␣␣␣tmp␣&=␣~bitmask;\n")
70      file.write("␣␣␣␣␣␣␣␣tmp␣|=␣value;\n")
71      file.write("␣␣␣␣␣␣␣␣this->set(tmp);\n")
72      file.write("␣␣␣␣}\n")
73      file.write("};\n\n")
74
75  def define_reg_class (file, registers) :
76      for i, reg in enumerate(registers) :
77          file.write("class␣Register_" + str(i) + "␣:␣public␣Register<" +
             str(i) + ">␣{\n")
78          file.write("private:\n")
79          file.write("␣␣␣␣static␣consteval␣std::uint32_t␣calc_set_bits␣(
                modes␣mode,␣choices␣choice)␣{\n")
80          file.write("␣␣␣␣␣␣␣␣return␣static_cast<std::uint32_t>(mode)␣<<␣
                static_cast<std::uint32_t>(choice)␣*␣2;\n")
81          file.write("␣␣␣␣}\n")
82          file.write("␣␣␣␣static␣consteval␣std::uint32_t␣calc_clear_bits␣
                (modes␣mode,␣choices␣choice)␣{\n")
83          file.write("␣␣␣␣␣␣␣␣return␣(static_cast<std::uint32_t>(masks::
                mask_" + str(reg['mask_id']) + ")␣|␣static_cast<std::
                uint32_t>(mode))␣<<␣static_cast<std::uint32_t>(choice)␣*␣
                2;\n")
84          file.write("␣␣␣␣}\n")
85          file.write("public:\n")
86          file.write("␣␣␣␣template<modes␣mode,␣choices␣choice>\n")
87          file.write("␣␣␣␣requires␣(is_valid_choice<choice>␣&&␣
                is_valid_mode<mode>)\n")
88          file.write("␣␣␣␣inline␣void␣set_mode␣()␣{\n")
89          file.write("␣␣␣␣␣␣␣␣set(calc_set_bits(mode,␣choice),␣
                calc_clear_bits(mode,␣choice));\n")
90          file.write("␣␣␣␣}\n")
91          file.write("};\n\n")
92
93  def batch_write (string, files) :
94      for file in files :
95          file.write(string)
96
97  c_main = open("main_" + str(num_functions) + ".c", "w")
98  base_c = open("base_" + str(num_functions) + ".c", "w")
99  base_cpp = open("base_" + str(num_functions) + ".cpp", "w")
100 cpp_main = open("main_" + str(num_functions) + ".cpp", "w")
101 cpp_encapsulation = open("main_encapsulation_" + str(num_functions) + "
        .cpp", "w")
102 cpp_static_poly = open("main_static_poly_" + str(num_functions) + ".cpp
        ", "w")
103 cpp_dynamic_poly = open("main_dynamic_poly_" + str(num_functions) + ".
        cpp", "w")
104
105 # adding includes
106
107 include_header(base_c, [ "stdint.h", "time.h", "stdio.h", "unistd.h" ])
108 include_header(c_main, [ "stdint.h", "time.h", "stdio.h", "unistd.h" ])
109
110 include_header(base_cpp, [ "cstdint", "concepts", "type_traits", "
        iostream", "unistd.h" ])
111 include_header(cpp_main, [ "cstdint", "concepts", "type_traits", "
        iostream", "unistd.h" ])
112 include_header(cpp_encapsulation, [ "cstdint", "concepts", "type_traits
        ", "iostream", "unistd.h" ])
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
113 include_header(cpp_static_poly, [ "cstdint", "concepts", "type_traits",
      "iostream", "unistd.h" ])
114 include_header(cpp_dynamic_poly, [ "cstdint", "concepts", "type_traits"
      , "iostream", "unistd.h" ])
115
116 # adding modes
117
118 num_modes = random.randint(1, MAX_MODES)
119
120 add_enum(c_main, "MODE", "MODE", num_modes)
121 add_enum_class(cpp_main, "modes", "mode", "std::uint32_t", num_modes)
122 add_enum_class(cpp_encapsulation, "modes", "mode", "std::uint32_t",
      num_modes)
123 add_enum_class(cpp_static_poly, "modes", "mode", "std::uint32_t",
      num_modes)
124 add_enum_class(cpp_dynamic_poly, "modes", "mode", "std::uint32_t",
      num_modes)
125
126 # adding choices
127
128 num_choices = random.randint(1, MAX_CHOICES)
129
130 add_enum(c_main, "CHOICE", "CHOICE", num_choices)
131 add_enum_class(cpp_main, "choices", "choice", "std::uint16_t",
      num_choices)
132 add_enum_class(cpp_encapsulation, "choices", "choice", "std::uint16_t",
      num_choices)
133 add_enum_class(cpp_static_poly, "choices", "choice", "std::uint16_t",
      num_choices)
134 add_enum_class(cpp_dynamic_poly, "choices", "choice", "std::uint16_t",
      num_choices)
135
136 # adding masks
137
138 num_masks = random.randint(1, MAX_MASKS)
139
140 add_enum(c_main, "MASK", "MASK", num_masks)
141 add_enum_class(cpp_main, "masks", "mask", "std::uint32_t", num_masks)
142 add_enum_class(cpp_encapsulation, "masks", "mask", "std::uint32_t",
      num_masks)
143 add_enum_class(cpp_static_poly, "masks", "mask", "std::uint32_t",
      num_masks)
144 add_enum_class(cpp_dynamic_poly, "masks", "mask", "std::uint32_t",
      num_masks)
145
146 # adding registers
147
148 num_registers = random.randint(1, MAX_REGISTERS)
149
150 c_main.write("uint32_t␣m_memory[" + str(num_registers) + "];\n\n")
151 base_c.write("uint32_t␣m_memory[" + str(num_registers) + "];\n\n")
152 cpp_main.write("std::uint32_t␣m_memory[" + str(num_registers) + "];\n\n
      ")
153 base_cpp.write("std::uint32_t␣m_memory[" + str(num_registers) + "];\n\n
      ")
154 cpp_encapsulation.write("std::uint32_t␣m_memory[" + str(num_registers)
      + "];\n\n")
155 cpp_static_poly.write("std::uint32_t␣m_memory[" + str(num_registers) +
      "];\n\n")
156 cpp_dynamic_poly.write("std::uint32_t␣m_memory[" + str(num_registers) +
      "];\n\n")
157
158 for i in range(num_registers) :
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
159     c_main.write("#define␣ELEM_" + str(i) + "␣(*((volatile␣uint32_t*)&
            m_memory[" + str(i) + "]))\n")
160     cpp_main.write("#define␣ELEM_" + str(i) + "␣(*((volatile␣uint32_t*)
            &m_memory[" + str(i) + "]))\n")
161
162 define_reg_base_class(cpp_encapsulation)
163 define_reg_base_class(cpp_static_poly)
164 define_reg_base_class(cpp_dynamic_poly)
165
166 registers = []
167 for i in range(num_registers) :
168     registers.append({
169         "mask_id" : random.randint(0, num_masks - 1)
170     })
171
172
173 cpp_main.write("static␣consteval␣std::uint32_t␣calc_set_bits␣(modes␣
        mode,␣choices␣choice)␣{\n")
174 cpp_main.write("␣␣␣␣return␣static_cast<std::uint32_t>(mode)␣<<␣
        static_cast<std::uint32_t>(choice)␣*␣2;\n")
175 cpp_main.write("}\n")
176
177 cpp_main.write("static␣consteval␣std::uint32_t␣calc_clear_bits␣(masks␣
        mask,␣modes␣mode,␣choices␣choice)␣{\n")
178 cpp_main.write("␣␣␣␣return␣(static_cast<std::uint32_t>(mask)␣|␣
        static_cast<std::uint32_t>(mode))␣<<␣static_cast<std::uint32_t>(
        choice)␣*␣2;\n")
179 cpp_main.write("}\n")
180
181 define_reg_class(cpp_encapsulation, registers)
182 define_reg_class(cpp_static_poly, registers)
183 define_reg_class(cpp_dynamic_poly, registers)
184
185 # C structs
186
187 c_main.write("typedef␣struct␣{\n")
188 c_main.write("␣␣␣␣uint32_t␣choice␣;\n")
189 c_main.write("␣␣␣␣uint32_t␣mode␣;\n")
190 c_main.write("}␣A_Struct␣;\n")
191
192 # static polymorphism
193 cpp_static_poly.write("template␣<typename␣T>\n")
194 cpp_static_poly.write("concept␣IPeripheral␣=␣requires␣(T␣peripheral)␣{\
        n")
195 cpp_static_poly.write("␣␣␣␣{␣peripheral.DoStuff()␣}␣->␣std::same_as<
        void>;\n")
196 cpp_static_poly.write("};\n\n")
197
198 cpp_static_poly.write("template␣<IPeripheral␣TPeripheral>\n")
199 cpp_static_poly.write("void␣HandlePeripheral␣(TPeripheral&␣peripheral)␣
        {\n")
200 cpp_static_poly.write("␣␣␣␣peripheral.DoStuff();\n")
201 cpp_static_poly.write("}\n")
202
203 # dynamic polymorphism
204 cpp_dynamic_poly.write("class␣IPeripheral␣{\n")
205 cpp_dynamic_poly.write("public:\n")
206 cpp_dynamic_poly.write("␣␣␣␣virtual␣~IPeripheral()␣{␣}\n")
207 cpp_dynamic_poly.write("␣␣␣␣virtual␣void␣DoStuff␣()␣=␣0;\n")
208 cpp_dynamic_poly.write("};\n")
209
210 cpp_dynamic_poly.write("void␣HandlePeripheral␣(IPeripheral&␣peripheral)
        ␣{\n")
```

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
211 cpp_dynamic_poly.write("␣␣␣␣peripheral.DoStuff();\n")
212 cpp_dynamic_poly.write("}\n")
213
214 # functions
215
216 cpp_class_files = [cpp_encapsulation, cpp_static_poly, cpp_dynamic_poly
        ]
217
218 for i in range(num_functions) :
219     c_main.write("void␣DoStuff_" + str(i) + "(A_Struct*␣m_struct)␣{\n")
220     c_main.write("␣␣␣␣uint32_t␣temp;\n")
221     c_main.write("\n")
222     c_main.write("␣␣␣␣if␣(!IS_CHOICE(m_struct->choice))␣{␣return;␣}\n")
223     c_main.write("␣␣␣␣if␣(!IS_MODE(m_struct->mode))␣{␣return;␣}\n")
224     c_main.write("\n")
225
226     cpp_main.write("template␣<choices␣choice,␣modes␣mode>\n")
227     cpp_main.write("requires␣(is_valid_choice<choice>␣&&␣is_valid_mode<
            mode>)\n")
228     cpp_main.write("void␣DoStuff_" + str(i) + "()␣{\n")
229     cpp_main.write("␣␣␣␣uint32_t␣temp;\n")
230     cpp_main.write("\n")
231
232     batch_write("template␣<choices␣choice,␣modes␣mode>\n", [
            cpp_static_poly, cpp_dynamic_poly])
233     batch_write("requires␣(is_valid_choice<choice>␣&&␣is_valid_mode<
            mode>)\n", [cpp_static_poly, cpp_dynamic_poly])
234     batch_write("class␣Peripheral_" + str(i) + "␣{\n", [
            cpp_encapsulation, cpp_static_poly])
235     batch_write("class␣Peripheral_" + str(i) + "␣:␣public␣IPeripheral␣
            {\n", [cpp_dynamic_poly])
236     batch_write("public:\n", cpp_class_files)
237     batch_write("␣␣␣␣template␣<choices␣choice,␣modes␣mode>\n", [
            cpp_encapsulation])
238     batch_write("␣␣␣␣requires␣(is_valid_choice<choice>␣&&␣is_valid_mode
            <mode>)\n", [cpp_encapsulation])
239     batch_write("␣␣␣␣void␣DoStuff()␣{\n", [cpp_encapsulation,
            cpp_static_poly])
240     batch_write("␣␣␣␣void␣DoStuff()␣override␣{\n", [cpp_dynamic_poly])
241     batch_write("\n", cpp_class_files)
242
243     branching_prob = random.random()
244     if branching_prob > BRANCHING_THRESHOLD :
245         mode_index = random.randint(0, num_modes - 1)
246         reg_index = random.randint(0, num_registers - 1)
247
248         c_main.write("␣␣␣␣if␣(m_struct->mode␣==␣MODE_" + str(mode_index
                ) + ")␣{\n")
249         c_main.write("␣␣␣␣␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + ";\n")
250         c_main.write("␣␣␣␣␣␣␣␣temp␣&=␣~((MASK_" + str(registers[
                reg_index]['mask_id']) + "␣|␣m_struct->mode)␣<<␣m_struct->
                choice␣*␣2u);\n")
251         c_main.write("␣␣␣␣␣␣␣␣temp␣|=␣m_struct->mode␣<<␣m_struct->
                choice␣*␣2u;\n")
252         c_main.write("␣␣␣␣␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp;\n")
253
254         cpp_main.write("␣␣␣␣if␣constexpr␣(mode␣==␣modes::mode_" + str(
                mode_index) + ")␣{\n")
255         cpp_main.write("␣␣␣␣␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + ";\n")
256         cpp_main.write("␣␣␣␣␣␣␣␣temp␣&=␣~calc_clear_bits(masks::mask_"
                + str(registers[reg_index]['mask_id']) + ",mode,␣choice);\
                n")
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
257         cpp_main.write("␣␣␣␣␣␣␣␣temp␣|=␣calc_set_bits(mode,␣choice);\n"
                )
258         cpp_main.write("␣␣␣␣␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp;\n")
259 
260         batch_write("␣␣␣␣␣␣␣␣if␣constexpr␣(mode␣==␣modes::mode_" + str(
                mode_index) + ")␣{\n", cpp_class_files)
261         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣{\n", cpp_class_files)
262         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Register_" + str(reg_index) + "␣
                reg␣{};\n", cpp_class_files)
263         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg.set_mode<mode,␣choice>();\n",
                cpp_class_files)
264         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣}\n", cpp_class_files)
265 
266         while (random.random() < STOP_THRESHOLD) :
267             mode_index = random.randint(0, num_modes - 1)
268             reg_index = random.randint(0, num_registers - 1)
269             c_main.write("␣␣␣␣␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + ";\n
                    ")
270             c_main.write("␣␣␣␣␣␣␣␣temp␣&=␣~((MASK_" + str(registers[
                    reg_index]['mask_id']) + "␣|␣m_struct->mode)␣<<␣
                    m_struct->choice␣*␣2u);\n")
271             c_main.write("␣␣␣␣␣␣␣␣temp␣|=␣m_struct->mode␣<<␣m_struct->
                    choice␣*␣2u;\n")
272             c_main.write("␣␣␣␣␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp;\n
                    ")
273 
274             cpp_main.write("␣␣␣␣␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + "
                    ;\n")
275             cpp_main.write("␣␣␣␣␣␣␣␣temp␣&=␣~calc_clear_bits(masks::
                    mask_" + str(registers[reg_index]['mask_id']) + ",␣mode
                    ,␣choice);\n")
276             cpp_main.write("␣␣␣␣␣␣␣␣temp␣|=␣calc_set_bits(mode,␣choice)
                    ;\n")
277             cpp_main.write("␣␣␣␣␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp
                    ;\n")
278 
279             batch_write("␣␣␣␣␣␣␣␣␣␣␣␣{\n", cpp_class_files)
280             batch_write("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Register_" + str(reg_index) +
                    "␣reg␣{};\n", cpp_class_files)
281             batch_write("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg.set_mode<mode,␣choice>();\
                    n", cpp_class_files)
282             batch_write("␣␣␣␣␣␣␣␣␣␣␣␣}\n", cpp_class_files)
283 
284         c_main.write("␣␣␣␣}\n")
285         cpp_main.write("␣␣␣␣}\n")
286         batch_write("␣␣␣␣␣␣␣␣}\n", cpp_class_files)
287 
288     while (random.random() < STOP_THRESHOLD) :
289         mode_index = random.randint(0, num_modes - 1)
290         reg_index = random.randint(0, num_registers - 1)
291         c_main.write("␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + ";\n")
292         c_main.write("␣␣␣␣temp␣&=␣~((MASK_" + str(registers[reg_index][
                'mask_id']) + "␣|␣m_struct->mode)␣<<␣m_struct->choice␣*␣2u)
                ;\n")
293         c_main.write("␣␣␣␣temp␣|=␣m_struct->mode␣<<␣m_struct->choice␣*␣
                2u;\n")
294         c_main.write("␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp;\n")
295 
296         cpp_main.write("␣␣␣␣temp␣=␣ELEM_" + str(reg_index) + ";\n")
297         cpp_main.write("␣␣␣␣temp␣&=␣~calc_clear_bits(masks::mask_" +
                str(registers[reg_index]['mask_id']) + ",␣mode,␣choice);\n"
                )
298         cpp_main.write("␣␣␣␣temp␣|=␣calc_set_bits(mode,␣choice);\n")
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
299         cpp_main.write("␣␣␣␣ELEM_" + str(reg_index) + "␣=␣temp;\n")
300 
301         batch_write("␣␣␣␣␣␣␣␣{\n", cpp_class_files)
302         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣Register_" + str(reg_index) + "␣reg␣
                {};\n", cpp_class_files)
303         batch_write("␣␣␣␣␣␣␣␣␣␣␣␣reg.set_mode<mode,␣choice>();\n",
                cpp_class_files)
304         batch_write("␣␣␣␣␣␣␣␣}\n", cpp_class_files)
305 
306     c_main.write("}\n")
307     cpp_main.write("}\n")
308     batch_write("␣␣␣␣}\n", cpp_class_files)
309     batch_write("};\n", cpp_class_files)
310 
311 
312 # function calls
313 
314 all_files = [base_c, c_main, base_cpp, cpp_main, cpp_encapsulation,
        cpp_static_poly, cpp_dynamic_poly]
315 
316 batch_write("int␣main␣(void)␣{\n", all_files)
317 batch_write("␣␣␣␣struct␣timespec␣start,␣stop;\n", all_files)
318 batch_write("␣␣␣␣clock_gettime(CLOCK_REALTIME,␣&start);\n", all_files)
319 batch_write("␣␣␣␣uint64_t␣elapsed_ns␣=␣0;\n", all_files)
320 
321 for i in range(num_calls) :
322     choice_id = random.randint(0, num_choices - 1)
323     mode_id = random.randint(0, num_modes - 1)
324     func_id = random.randint(0, num_functions - 1)
325 
326     c_main.write("␣␣␣␣{\n")
327     c_main.write("␣␣␣␣␣␣␣␣A_Struct␣m_struct;\n")
328     c_main.write("␣␣␣␣␣␣␣␣m_struct.choice␣=␣CHOICE_" + str(choice_id) +
            ";\n")
329     c_main.write("␣␣␣␣␣␣␣␣m_struct.mode␣=␣MODE_" + str(mode_id) + ";\n"
            )
330     c_main.write("␣␣␣␣␣␣␣␣DoStuff_" + str(func_id) + "(&m_struct)␣;\n")
331     c_main.write("␣␣␣␣}\n")
332 
333     cpp_main.write("␣␣␣␣{\n")
334     cpp_main.write("␣␣␣␣␣␣␣␣DoStuff_" + str(func_id) + "<choices::
            choice_" + str(choice_id) + ",␣modes::mode_" + str(mode_id) + "
            >();\n")
335     cpp_main.write("␣␣␣␣}\n")
336 
337     cpp_encapsulation.write("␣␣␣␣{\n")
338     cpp_encapsulation.write("␣␣␣␣␣␣␣␣Peripheral_" + str(func_id) + "␣
            peripheral␣{};\n")
339     cpp_encapsulation.write("␣␣␣␣␣␣␣␣peripheral.DoStuff<choices::
            choice_" + str(choice_id) + ",␣modes::mode_" + str(mode_id) + "
            >();\n")
340     cpp_encapsulation.write("␣␣␣␣}\n")
341 
342     cpp_static_poly.write("␣␣␣␣{\n")
343     cpp_static_poly.write("␣␣␣␣␣␣␣␣Peripheral_" + str(func_id) + "<" +
            "␣choices::choice_" + str(choice_id) + ",␣modes::mode_" + str(
            mode_id) + ">␣peripheral␣{};\n")
344     cpp_static_poly.write("␣␣␣␣␣␣␣␣HandlePeripheral<Peripheral_" + str(
            func_id) + "<choices::choice_" + str(choice_id) + ",␣modes::
            mode_" + str(mode_id) + ">>␣(peripheral);\n")
345     cpp_static_poly.write("␣␣␣␣}\n")
346 
347     cpp_dynamic_poly.write("␣␣␣␣{\n")
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
348     cpp_dynamic_poly.write("␣␣␣␣␣␣␣␣Peripheral_" + str(func_id) + "<" + "␣choices::choice_" + str(choice_id) + ",␣modes::mode_" + str(mode_id) + ">␣peripheral␣{};\n")
349     cpp_dynamic_poly.write("␣␣␣␣␣␣␣␣HandlePeripheral␣(peripheral);\n")
350     cpp_dynamic_poly.write("␣␣␣␣}\n")
351 
352 batch_write("␣␣␣␣clock_gettime(CLOCK_REALTIME,␣&stop);\n", all_files)
353 batch_write("␣␣␣␣\n", all_files)
354 batch_write("␣␣␣␣elapsed_ns␣=␣(stop.tv_sec␣-␣start.tv_sec)␣*␣(uint64_t)1e9;\n", all_files)
355 batch_write("␣␣␣␣elapsed_ns␣+=␣stop.tv_nsec␣-␣start.tv_nsec;\n", all_files)
356 batch_write("␣␣␣␣\n", all_files)
357 batch_write("␣␣␣␣printf(\"%lu\\n\",␣elapsed_ns);\n", all_files)
358 batch_write("␣␣␣␣return␣0;\n", all_files)
359 batch_write("}\n", all_files)
360 
361 
362 c_main.close()
363 base_c.close()
364 cpp_main.close()
365 cpp_encapsulation.close()
366 cpp_static_poly.close()
367 cpp_dynamic_poly.close()
368 base_cpp.close()
```

## D.11 Plotting Script for Abstraction Overhead Measurements

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4 import sys
5 
6 N = sys.argv[1]
7 
8 my_dpi=200
9 
10 data = pd.read_csv("data_" + N + ".csv")
11 
12 fig, axs = plt.subplots(2, 2)
13 
14 fig.suptitle('evaluation')
15 fig.set_size_inches(20, 15)
16 
17 axs[0, 0].plot(data['num_calls'].values, data['c_size_o'].values, label="C", color='g', linestyle='solid', marker=',')
18 axs[0, 0].plot(data['num_calls'].values, data['cpp_size_o'].values, label="C++", color='r', linestyle='solid', marker='x')
19 axs[0, 0].plot(data['num_calls'].values, data['cpp_encapsulation_size_o'].values, label="C++␣(encapsulation)", color='b', linestyle='solid', marker='v')
20 axs[0, 0].plot(data['num_calls'].values, data['cpp_static_poly_size_o'].values, label="C++␣(static␣polymorphism)", color='y', linestyle='solid', marker='.')
21 axs[0, 0].plot(data['num_calls'].values, data['cpp_dynamic_poly_size_o'].values, label="C++␣(dynamic␣polymorphism)", color='m', linestyle='solid', marker='s')
22 axs[0, 0].set(xlabel='number␣of␣function␣calls', ylabel='size␣overhead␣(bytes)')
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
23 axs[0, 0].legend(loc="upper␣left")
24
25 axs[0, 1].plot(data['num_calls'].values, data['c_comp_t'].values, label
       ="C", color='g', linestyle='solid', marker=',')
26 axs[0, 1].plot(data['num_calls'].values, data['cpp_comp_t'].values,
       label="C++", color='r', linestyle='solid', marker='x')
27 axs[0, 1].plot(data['num_calls'].values, data['cpp_encapsulation_comp_t
       '].values, label="C++␣(encapsulation)", color='b', linestyle='solid
       ', marker='v')
28 axs[0, 1].plot(data['num_calls'].values, data['cpp_static_poly_comp_t'
       ].values, label="C++␣(static␣polymorphism)", color='y', linestyle='
       solid', marker='.')
29 axs[0, 1].plot(data['num_calls'].values, data['cpp_dynamic_poly_comp_t'
       ].values, label="C++␣(dynamic␣polymorphism)", color='m', linestyle=
       'solid', marker='s')
30 axs[0, 1].set(xlabel='number␣of␣function␣calls', ylabel='compilation␣
       time␣(s)')
31 axs[0, 1].legend(loc="upper␣left")
32
33 axs[1, 0].plot(data['num_calls'].values, data['c_link_t'].values, label
       ="C", color='g', linestyle='solid', marker=',')
34 axs[1, 0].plot(data['num_calls'].values, data['cpp_link_t'].values,
       label="C++", color='r', linestyle='solid', marker='x')
35 axs[1, 0].plot(data['num_calls'].values, data['cpp_encapsulation_link_t
       '].values, label="C++␣(encapsulation)", color='b', linestyle='solid
       ', marker='v')
36 axs[1, 0].plot(data['num_calls'].values, data['cpp_static_poly_link_t'
       ].values, label="C++␣(static␣polymorphism)", color='y', linestyle='
       solid', marker='.')
37 axs[1, 0].plot(data['num_calls'].values, data['cpp_dynamic_poly_link_t'
       ].values, label="C++␣(dynamic␣polymorphism)", color='m', linestyle=
       'solid', marker='s')
38 axs[1, 0].set(xlabel='number␣of␣function␣calls', ylabel='link␣time␣(s)'
       )
39 axs[1, 0].legend(loc="upper␣left")
40
41 axs[1, 1].plot(data['num_calls'].values, data['c_exec_t'].values, label
       ="C", color='g', linestyle='solid', marker=',')
42 axs[1, 1].plot(data['num_calls'].values, data['cpp_exec_t'].values,
       label="C++", color='r', linestyle='solid', marker='x')
43 axs[1, 1].plot(data['num_calls'].values, data['cpp_encapsulation_exec_t
       '].values, label="C++␣(encapsulation)", color='b', linestyle='solid
       ', marker='v')
44 axs[1, 1].plot(data['num_calls'].values, data['cpp_static_poly_exec_t'
       ].values, label="C++␣(static␣polymorphism)", color='y', linestyle='
       solid', marker='.')
45 axs[1, 1].plot(data['num_calls'].values, data['cpp_dynamic_poly_exec_t'
       ].values, label="C++␣(dynamic␣polymorphism)", color='m', linestyle=
       'solid', marker='s')
46 axs[1, 1].set(xlabel='number␣of␣function␣calls', ylabel='execution␣time
       ␣(ns)')
47 axs[1, 1].legend(loc="upper␣left")
48
49 plt.savefig('plt_cpp_v_cpp' + N + '.png', dpi=my_dpi)
```

## D.12 Benchmarking Script for Abstraction Overhead Measurements

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```bash
#!/bin/bash

export TIMEFORMAT=%3R

NUM_EPOCHS=5

NUM_FUNCS=$1

CSV_FILE="data_$NUM_FUNCS.csv"
JSON_FILE="data_$NUM_FUNCS.json"

BASE_C_SRC="base_$NUM_FUNCS.c"
BASE_C_BIN="base_c_$NUM_FUNCS"
BASE_CPP_SRC="base_$NUM_FUNCS.cpp"
BASE_CPP_BIN="base_cpp_$NUM_FUNCS"

MAIN_C_SRC="main_$NUM_FUNCS.c"
MAIN_C_OBJ="main_c_$NUM_FUNCS.o"
MAIN_C_BIN="main_c_$NUM_FUNCS"

MAIN_CPP_SRC="main_$NUM_FUNCS.cpp"
MAIN_CPP_OBJ="main_cpp_$NUM_FUNCS.o"
MAIN_CPP_BIN="main_cpp_$NUM_FUNCS"

MAIN_ENCAPSULATION_CPP_SRC="main_encapsulation_$NUM_FUNCS.cpp"
MAIN_ENCAPSULATION_CPP_OBJ="main_cpp_encapsulation_$NUM_FUNCS.o"
MAIN_ENCAPSULATION_CPP_BIN="main_cpp_encapsulation_$NUM_FUNCS"

MAIN_STATIC_POLY_CPP_SRC="main_static_poly_$NUM_FUNCS.cpp"
MAIN_STATIC_POLY_CPP_OBJ="main_cpp_static_poly_$NUM_FUNCS.o"
MAIN_STATIC_POLY_CPP_BIN="main_cpp_static_poly_$NUM_FUNCS"

MAIN_DYNAMIC_POLY_CPP_SRC="main_dynamic_poly_$NUM_FUNCS.cpp"
MAIN_DYNAMIC_POLY_CPP_OBJ="main_cpp_dynamic_poly_$NUM_FUNCS.o"
MAIN_DYNAMIC_POLY_CPP_BIN="main_cpp_dynamic_poly_$NUM_FUNCS"

printf "num_calls," > $CSV_FILE
printf "c_size_o,cpp_size_o,cpp_encapsulation_size_o,
    cpp_static_poly_size_o,cpp_dynamic_poly_size_o," >> $CSV_FILE
printf "c_comp_t,cpp_comp_t,cpp_encapsulation_comp_t,
    cpp_static_poly_comp_t,cpp_dynamic_poly_comp_t," >> $CSV_FILE
printf "c_link_t,cpp_link_t,cpp_encapsulation_link_t,
    cpp_static_poly_link_t,cpp_dynamic_poly_link_t," >> $CSV_FILE
printf "c_exec_t,cpp_exec_t,cpp_encapsulation_exec_t,
    cpp_static_poly_exec_t,cpp_dynamic_poly_exec_t\n" >> $CSV_FILE

NUM_CALLS=0
INCREMENT=0
STOP_AT=20000

while true; do

    NUM_CALLS=$((NUM_CALLS + INCREMENT))
    INCREMENT=$((INCREMENT + 1))

    if [ $NUM_CALLS -gt $STOP_AT ]; then
        break
    fi

    C_COMPILE_TIME_AVG=0
    C_LINK_TIME_AVG=0
    C_SIZE=0
    C_EXEC_TIME_AVG=0
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
60  BASE_C_SIZE=0
61  CPP_COMPILE_TIME_AVG=0
62  CPP_LINK_TIME_AVG=0
63  CPP_SIZE=0
64  CPP_EXEC_TIME_AVG=0
65  BASE_CPP_SIZE=0
66  CPP_ENCAPSULATION_COMPILE_TIME_AVG=0
67  CPP_ENCAPSULATION_LINK_TIME_AVG=0
68  CPP_ENCAPSULATION_SIZE=0
69  CPP_ENCAPSULATION_EXEC_TIME_AVG=0
70  CPP_STATIC_POLY_COMPILE_TIME_AVG=0
71  CPP_STATIC_POLY_LINK_TIME_AVG=0
72  CPP_STATIC_POLY_SIZE=0
73  CPP_STATIC_POLY_EXEC_TIME_AVG=0
74  CPP_DYNAMIC_POLY_COMPILE_TIME_AVG=0
75  CPP_DYNAMIC_POLY_LINK_TIME_AVG=0
76  CPP_DYNAMIC_POLY_SIZE=0
77  CPP_DYNAMIC_POLY_EXEC_TIME_AVG=0
78
79  for i in $(seq 1 $NUM_EPOCHS); do
80
81      python3 generate.py $NUM_FUNCS $NUM_CALLS
82
83      gcc -Os -Wall $BASE_C_SRC -o $BASE_C_BIN
84      BASE_C_SIZE=`size $BASE_C_BIN | tail -n1 | awk '{print␣$1}'`
85
86      g++ -Os -std=c++20 -Wall $BASE_CPP_SRC -o $BASE_CPP_BIN
87      BASE_CPP_SIZE=`size $BASE_CPP_BIN | tail -n1 | awk '{print␣$1}'
        `
88
89      C_COMPILE_TIME=`(time gcc -c -Os $MAIN_C_SRC -o $MAIN_C_OBJ)
           2>&1 >/dev/null`
90      C_LINK_TIME=`(time gcc -Os -Wall $MAIN_C_OBJ -o $MAIN_C_BIN)
           2>&1 >/dev/null`
91      C_SIZE=`size $MAIN_C_BIN | tail -n1 | awk '{print␣$1}'`
92      C_EXEC_TIME=`./$MAIN_C_BIN`
93
94
95      CPP_COMPILE_TIME=`(time g++ -std=c++20 -c -Os $MAIN_CPP_SRC -o
           $MAIN_CPP_OBJ) 2>&1 >/dev/null`
96      CPP_LINK_TIME=`(time g++ -std=c++20 -Os -Wall $MAIN_CPP_OBJ -o
           $MAIN_CPP_BIN) 2>&1 >/dev/null`
97      CPP_SIZE=`size $MAIN_CPP_BIN | tail -n1 | awk '{print␣$1}'`
98      CPP_EXEC_TIME=`./$MAIN_CPP_BIN`
99
100     CPP_ENCAPSULATION_COMPILE_TIME=`(time g++ -std=c++20 -c -Os
           $MAIN_ENCAPSULATION_CPP_SRC -o $MAIN_ENCAPSULATION_CPP_OBJ)
            2>&1 >/dev/null`
101     CPP_ENCAPSULATION_LINK_TIME=`(time g++ -std=c++20 -Os -Wall
           $MAIN_ENCAPSULATION_CPP_OBJ -o $MAIN_ENCAPSULATION_CPP_BIN)
            2>&1 >/dev/null`
102     CPP_ENCAPSULATION_SIZE=`size $MAIN_ENCAPSULATION_CPP_BIN | tail
            -n1 | awk '{print␣$1}'`
103     CPP_ENCAPSULATION_EXEC_TIME=`./$MAIN_ENCAPSULATION_CPP_BIN`
104
105     CPP_STATIC_POLY_COMPILE_TIME=`(time g++ -std=c++20 -c -Os
           $MAIN_STATIC_POLY_CPP_SRC -o $MAIN_STATIC_POLY_CPP_OBJ)
           2>&1 >/dev/null`
106     CPP_STATIC_POLY_LINK_TIME=`(time g++ -std=c++20 -Os -Wall
           $MAIN_STATIC_POLY_CPP_OBJ -o $MAIN_STATIC_POLY_CPP_BIN)
           2>&1 >/dev/null`
107     CPP_STATIC_POLY_SIZE=`size $MAIN_STATIC_POLY_CPP_BIN | tail -n1
            | awk '{print␣$1}'`
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
108         CPP_STATIC_POLY_EXEC_TIME=`./$MAIN_STATIC_POLY_CPP_BIN`
109 
110         CPP_DYNAMIC_POLY_COMPILE_TIME=`(time g++ -std=c++20 -c -Os
                $MAIN_DYNAMIC_POLY_CPP_SRC -o $MAIN_DYNAMIC_POLY_CPP_OBJ)
                2>&1 >/dev/null`
111         CPP_DYNAMIC_POLY_LINK_TIME=`(time g++ -std=c++20 -Os -Wall
                $MAIN_DYNAMIC_POLY_CPP_OBJ -o $MAIN_DYNAMIC_POLY_CPP_BIN)
                2>&1 >/dev/null`
112         CPP_DYNAMIC_POLY_SIZE=`size $MAIN_DYNAMIC_POLY_CPP_BIN | tail -
                n1 | awk '{print␣$1}'`
113         CPP_DYNAMIC_POLY_EXEC_TIME=`./$MAIN_DYNAMIC_POLY_CPP_BIN`
114 
115         C_COMPILE_TIME_AVG=`echo $C_COMPILE_TIME_AVG + $C_COMPILE_TIME
                | bc`
116         C_LINK_TIME_AVG=`echo $C_LINK_TIME_AVG + $C_LINK_TIME | bc`
117         C_EXEC_TIME_AVG=`echo $C_EXEC_TIME_AVG + $C_EXEC_TIME | bc`
118         CPP_COMPILE_TIME_AVG=`echo $CPP_COMPILE_TIME_AVG +
                $CPP_COMPILE_TIME | bc`
119         CPP_LINK_TIME_AVG=`echo $CPP_LINK_TIME_AVG + $CPP_LINK_TIME |
                bc`
120         CPP_EXEC_TIME_AVG=`echo $CPP_EXEC_TIME_AVG + $CPP_EXEC_TIME |
                bc`
121         CPP_ENCAPSULATION_COMPILE_TIME_AVG=`echo
                $CPP_ENCAPSULATION_COMPILE_TIME_AVG +
                $CPP_ENCAPSULATION_COMPILE_TIME | bc`
122         CPP_ENCAPSULATION_LINK_TIME_AVG=`echo
                $CPP_ENCAPSULATION_LINK_TIME_AVG +
                $CPP_ENCAPSULATION_LINK_TIME | bc`
123         CPP_ENCAPSULATION_EXEC_TIME_AVG=`echo
                $CPP_ENCAPSULATION_EXEC_TIME_AVG +
                $CPP_ENCAPSULATION_EXEC_TIME | bc`
124         CPP_STATIC_POLY_COMPILE_TIME_AVG=`echo
                $CPP_STATIC_POLY_COMPILE_TIME_AVG +
                $CPP_STATIC_POLY_COMPILE_TIME | bc`
125         CPP_STATIC_POLY_LINK_TIME_AVG=`echo
                $CPP_STATIC_POLY_LINK_TIME_AVG + $CPP_STATIC_POLY_LINK_TIME
                 | bc`
126         CPP_STATIC_POLY_EXEC_TIME_AVG=`echo
                $CPP_STATIC_POLY_EXEC_TIME_AVG + $CPP_STATIC_POLY_EXEC_TIME
                 | bc`
127         CPP_DYNAMIC_POLY_COMPILE_TIME_AVG=`echo
                $CPP_DYNAMIC_POLY_COMPILE_TIME_AVG +
                $CPP_DYNAMIC_POLY_COMPILE_TIME | bc`
128         CPP_DYNAMIC_POLY_LINK_TIME_AVG=`echo
                $CPP_DYNAMIC_POLY_LINK_TIME_AVG +
                $CPP_DYNAMIC_POLY_LINK_TIME | bc`
129         CPP_DYNAMIC_POLY_EXEC_TIME_AVG=`echo
                $CPP_DYNAMIC_POLY_EXEC_TIME_AVG +
                $CPP_DYNAMIC_POLY_EXEC_TIME | bc`
130 
131         rm $MAIN_C_SRC
132         rm $MAIN_C_OBJ
133         rm $MAIN_C_BIN
134         rm $MAIN_CPP_SRC
135         rm $MAIN_CPP_OBJ
136         rm $MAIN_CPP_BIN
137         rm $MAIN_ENCAPSULATION_CPP_SRC
138         rm $MAIN_ENCAPSULATION_CPP_OBJ
139         rm $MAIN_ENCAPSULATION_CPP_BIN
140         rm $MAIN_STATIC_POLY_CPP_SRC
141         rm $MAIN_STATIC_POLY_CPP_OBJ
142         rm $MAIN_STATIC_POLY_CPP_BIN
143         rm $MAIN_DYNAMIC_POLY_CPP_SRC
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
144         rm $MAIN_DYNAMIC_POLY_CPP_OBJ
145         rm $MAIN_DYNAMIC_POLY_CPP_BIN
146         rm $BASE_C_SRC
147         rm $BASE_C_BIN
148         rm $BASE_CPP_SRC
149         rm $BASE_CPP_BIN
150 
151     done
152 
153     C_COMPILE_TIME_AVG=`echo $C_COMPILE_TIME_AVG / $NUM_EPOCHS | bc -l`
154     C_LINK_TIME_AVG=`echo $C_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
155     C_EXEC_TIME_AVG=`echo $C_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
156     CPP_COMPILE_TIME_AVG=`echo $CPP_COMPILE_TIME_AVG / $NUM_EPOCHS | bc
         -l`
157     CPP_LINK_TIME_AVG=`echo $CPP_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
158     CPP_EXEC_TIME_AVG=`echo $CPP_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
159     CPP_ENCAPSULATION_COMPILE_TIME_AVG=`echo
        $CPP_ENCAPSULATION_COMPILE_TIME_AVG / $NUM_EPOCHS | bc -l`
160     CPP_ENCAPSULATION_LINK_TIME_AVG=`echo
        $CPP_ENCAPSULATION_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
161     CPP_ENCAPSULATION_EXEC_TIME_AVG=`echo
        $CPP_ENCAPSULATION_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
162     CPP_STATIC_POLY_COMPILE_TIME_AVG=`echo
        $CPP_STATIC_POLY_COMPILE_TIME_AVG / $NUM_EPOCHS | bc -l`
163     CPP_STATIC_POLY_LINK_TIME_AVG=`echo $CPP_STATIC_POLY_LINK_TIME_AVG
        / $NUM_EPOCHS | bc -l`
164     CPP_STATIC_POLY_EXEC_TIME_AVG=`echo $CPP_STATIC_POLY_EXEC_TIME_AVG
        / $NUM_EPOCHS | bc -l`
165     CPP_DYNAMIC_POLY_COMPILE_TIME_AVG=`echo
        $CPP_DYNAMIC_POLY_COMPILE_TIME_AVG / $NUM_EPOCHS | bc -l`
166     CPP_DYNAMIC_POLY_LINK_TIME_AVG=`echo
        $CPP_DYNAMIC_POLY_LINK_TIME_AVG / $NUM_EPOCHS | bc -l`
167     CPP_DYNAMIC_POLY_EXEC_TIME_AVG=`echo
        $CPP_DYNAMIC_POLY_EXEC_TIME_AVG / $NUM_EPOCHS | bc -l`
168 
169     printf "$NUM_CALLS," >> $CSV_FILE
170     printf "$((C_SIZE-BASE_C_SIZE)),$((CPP_SIZE-BASE_CPP_SIZE)),$((
        CPP_ENCAPSULATION_SIZE-BASE_CPP_SIZE)),$((CPP_STATIC_POLY_SIZE-
        BASE_CPP_SIZE)),$((CPP_DYNAMIC_POLY_SIZE-BASE_CPP_SIZE))," >>
        $CSV_FILE
171     printf "$C_COMPILE_TIME_AVG,$CPP_COMPILE_TIME_AVG,
        $CPP_ENCAPSULATION_COMPILE_TIME_AVG,
        $CPP_STATIC_POLY_COMPILE_TIME_AVG,
        $CPP_DYNAMIC_POLY_COMPILE_TIME_AVG," >> $CSV_FILE
172     printf "$C_LINK_TIME_AVG,$CPP_LINK_TIME_AVG,
        $CPP_ENCAPSULATION_LINK_TIME_AVG,$CPP_STATIC_POLY_LINK_TIME_AVG
        ,$CPP_DYNAMIC_POLY_LINK_TIME_AVG," >> $CSV_FILE
173     printf "$C_EXEC_TIME_AVG,$CPP_EXEC_TIME_AVG,
        $CPP_ENCAPSULATION_EXEC_TIME_AVG,$CPP_STATIC_POLY_EXEC_TIME_AVG
        ,$CPP_DYNAMIC_POLY_EXEC_TIME_AVG\n" >> $CSV_FILE
174 
175 done
176 
177 python3 plot.py $NUM_FUNCS
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Appendix E

# Led Blinking

## E.1 hal/base/constants.hpp

```
#pragma once

#include <cstdint>

namespace hal {

/* Peripheral base address in the alias region */
constexpr std::uint32_t PERIPH_BASE_ADDR = UINT32_C(0x40000000);

/* Peripheral memory map */
constexpr std::uint32_t AHB1PERIPH_BASE_ADDR = PERIPH_BASE_ADDR +
    UINT32_C(0x00020000);
constexpr std::uint32_t AHB2PERIPH_BASE_ADDR = PERIPH_BASE_ADDR +
    UINT32_C(0x08000000);

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.2 hal/gpio/constants.hpp

```
#pragma once

#include <cstdint>
#include <concepts>

#include "hal/base/constants.hpp"

namespace hal {

constexpr std::uint32_t GPIOA_BASE_ADDR = AHB2PERIPH_BASE_ADDR + UINT32_C(0x0000);
constexpr std::uint32_t GPIOB_BASE_ADDR = AHB2PERIPH_BASE_ADDR + UINT32_C(0x0400);
constexpr std::uint32_t GPIOC_BASE_ADDR = AHB2PERIPH_BASE_ADDR + UINT32_C(0x0800);
constexpr std::uint32_t GPIOD_BASE_ADDR = AHB2PERIPH_BASE_ADDR + UINT32_C(0x0C00);
constexpr std::uint32_t GPIOF_BASE_ADDR = AHB2PERIPH_BASE_ADDR + UINT32_C(0x1400);

constexpr std::uint32_t MODER_OFFSET   = UINT32_C(0x00);     /* GPIO port mode register,               Address offset: 0x00 */
constexpr std::uint32_t OTYPER_OFFSET  = UINT32_C(0x04);     /* GPIO port output type register,        Address offset: 0x04 */
constexpr std::uint32_t OSPEEDR_OFFSET = UINT32_C(0x08);     /* GPIO port output speed register,       Address offset: 0x08 */
constexpr std::uint32_t PUPDR_OFFSET   = UINT32_C(0x0C);     /* GPIO port pull-up/pull-down register,  Address offset: 0x0C */
constexpr std::uint32_t IDR_OFFSET     = UINT32_C(0x10);     /* GPIO port input data register,         Address offset: 0x10 */
constexpr std::uint32_t ODR_OFFSET     = UINT32_C(0x14);     /* GPIO port output data register,        Address offset: 0x14 */
constexpr std::uint32_t BSRR_OFFSET    = UINT32_C(0x18);     /* GPIO port bit set/reset register,      Address offset: 0x18 */
constexpr std::uint32_t LCKR_OFFSET    = UINT32_C(0x1C);     /* GPIO port configuration lock register, Address offset: 0x1C */
constexpr std::uint32_t AFRL_OFFSET    = UINT32_C(0x20);     /* GPIO alternate function registers,     Address offset: 0x20 */
constexpr std::uint32_t AFRH_OFFSET    = UINT32_C(0x24);     /* GPIO alternate function registers,     Address offset: 0x24 */
constexpr std::uint32_t BRR_OFFSET     = UINT32_C(0x28);     /* GPIO port bit reset registers,         Address offset: 0x28 */

template <std::uint32_t GPIOx_BASE_ADDR>
concept is_valid_gpio_base_address = ((GPIOx_BASE_ADDR == GPIOA_BASE_ADDR) ||
                                      (GPIOx_BASE_ADDR == GPIOB_BASE_ADDR) ||
                                      (GPIOx_BASE_ADDR == GPIOC_BASE_ADDR) ||
                                      (GPIOx_BASE_ADDR == GPIOD_BASE_ADDR) ||
                                      (GPIOx_BASE_ADDR == GPIOF_BASE_ADDR));

enum class gpio_ports : std::uint8_t {
    port_a,
    port_b,
    port_c,
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
39     port_d,
40     port_f
41 };
42
43 template <gpio_ports port>
44 concept is_valid_gpio_port = ((port == gpio_ports::port_a) ||
45                               (port == gpio_ports::port_b) ||
46                               (port == gpio_ports::port_c) ||
47                               (port == gpio_ports::port_d) ||
48                               (port == gpio_ports::port_f));
49
50 template <gpio_ports port>
51 requires (is_valid_gpio_port<port>)
52 consteval std::uint32_t port_to_base_address () {
53     if (port == gpio_ports::port_a) return GPIOA_BASE_ADDR;
54     else if (port == gpio_ports::port_b) return GPIOB_BASE_ADDR;
55     else if (port == gpio_ports::port_c) return GPIOC_BASE_ADDR;
56     else if (port == gpio_ports::port_d) return GPIOD_BASE_ADDR;
57     else return GPIOF_BASE_ADDR;
58 }
59
60 enum class gpio_pins : std::uint8_t {
61     pin_00 = 0,
62     pin_01 = 1,
63     pin_02 = 2,
64     pin_03 = 3,
65     pin_04 = 4,
66     pin_05 = 5,
67     pin_06 = 6,
68     pin_07 = 7,
69     pin_08 = 8,
70     pin_09 = 9,
71     pin_10 = 10,
72     pin_11 = 11,
73     pin_12 = 12,
74     pin_13 = 13,
75     pin_14 = 14,
76     pin_15 = 15
77 };
78
79 } /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

## E.3 hal/gpio/gpio.hpp

```cpp
#pragma once

#include <cstdint>

#include "hal/gpio/constants.hpp"
#include "hal/gpio/registers.hpp"
#include "hal/rcc/registers.hpp"

namespace hal {

template <gpio_ports port,
          gpio_pins pin,
          gpio_modes mode>
class CGpio {
public:
    bool configure () const {
        CModeRegister<port> mode_register {};
        CAhbEnRegister ahb_en_register {};

        ahb_en_register.enable_gpio_clock<port>();
        mode_register.template set_mode<pin, mode>();
        reset();

        return true;
    }

    void set () const {
        CBSRRegister<port> bsrr_register {};
        bsrr_register.template set_pin<pin>();
    }

    void reset () const {
        CBSRRegister<port> bsrr_register {};
        bsrr_register.template reset_pin<pin>();
    }
};

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.4 hal/gpio/registers.hpp

```
#pragma once

#include <cstdint>

#include "hal/gpio/constants.hpp"
#include "hal/register.hpp"

namespace hal {

/* GPIOx_MODER */

enum class gpio_modes : std::uint32_t {
    input    = 0b00,
    output   = 0b01,
    alt_func = 0b10,
    analog   = 0b11
};

template<gpio_ports port>
requires (is_valid_gpio_port<port>)
class CModeRegister : public CRegister<port_to_base_address<port>() +
    MODER_OFFSET> {
private:
    template <gpio_pins pin, gpio_modes mode>
    static consteval std::uint32_t calculate_value () {
        return static_cast<std::uint32_t>(mode) << static_cast<std::
            uint32_t>(pin) * 2;
    }

    template <gpio_pins pin, gpio_modes mode>
    static consteval std::uint32_t calculate_bitmask () {
        return static_cast<std::uint32_t>(0b11) << static_cast<std::
            uint32_t>(pin) * 2;
    }
public:
    template <gpio_pins pin, gpio_modes mode>
    inline void set_mode () {
        this->set(calculate_value<pin, mode>(), calculate_bitmask<pin,
            mode>());
    }
};

/* GPIOx_BSRR */

template<gpio_ports port>
requires (is_valid_gpio_port<port>)
class CBSRRegister : public CRegister<port_to_base_address<port>() +
    BSRR_OFFSET> {
public:
    template <gpio_pins pin>
    inline void set_pin () {
        this->set_bits(static_cast<std::uint32_t>(1) << static_cast<std
            ::uint32_t>(pin));
    }

    template <gpio_pins pin>
    inline void reset_pin () {
        this->set_bits(static_cast<std::uint32_t>(0x10000) <<
            static_cast<std::uint32_t>(pin));
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
53      }
54  };
55
56  } /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

## E.5 hal/rcc/constants.hpp

```
#pragma once

#include <cstdint>
#include <concepts>

#include "hal/base/constants.hpp"

namespace hal {

constexpr std::uint32_t RCC_BASE_ADDR = AHB1PERIPH_BASE_ADDR + UINT32_C
    (0x1000);

constexpr std::uint32_t RCC_AHBENR_OFFSET = UINT32_C(0x14);

constexpr std::uint32_t RCC_AHBENR_ADDR = RCC_BASE_ADDR +
    RCC_AHBENR_OFFSET;

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.6 hal/rcc/registers.hpp

```
#pragma once

#include <cstdint>

#include "hal/rcc/constants.hpp"
#include "hal/gpio/constants.hpp"
#include "hal/register.hpp"

namespace hal {

/* RCC_AHBENR */

class CAhbEnRegister : public CRegister<RCC_AHBENR_ADDR> {
public:
    template<gpio_ports port>
    requires (is_valid_gpio_port<port>)
    inline void enable_gpio_clock () {
        if constexpr (port == gpio_ports::port_a) set_bits(0x00020000);
        else if constexpr (port == gpio_ports::port_b) set_bits(0
            x00040000);
        else if constexpr (port == gpio_ports::port_c) set_bits(0
            x00080000);
        else if constexpr (port == gpio_ports::port_d) set_bits(0
            x00080000);
        else set_bits(0x00400000);
    }
};

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.7 hal/register.hpp

```
#pragma once

#include <cstdint>

namespace hal {

template <std::uint32_t address>
class CRegister {
public:
    inline void set (std::uint32_t value) const {
        *(reinterpret_cast<volatile std::uint32_t *>(address)) = value;
    }

    inline std::uint32_t get () const {
        return *(reinterpret_cast<volatile std::uint32_t *>(address));
    }

    inline void set (std::uint32_t value, std::uint32_t bitmask) const
        {
        std::uint32_t tmp = get();
        tmp &= ~bitmask;
        tmp |= value;
        set(tmp);
    }

    inline void set_bits (std::uint32_t bitmask) const {
        *(reinterpret_cast<volatile std::uint32_t *>(address)) = *(
            reinterpret_cast<volatile std::uint32_t *>(address)) |
            bitmask;
    }

    inline void clear_bits (std::uint32_t bitmask) const {
        *(reinterpret_cast<volatile std::uint32_t *>(address)) = *(
            reinterpret_cast<volatile std::uint32_t *>(address)) & ~
            bitmask;
    }

    inline bool is_set (std::uint32_t bitmask) const {
        return static_cast<bool>(*(reinterpret_cast<volatile std::
            uint32_t *>(address)) & bitmask);
    }

    inline std::uint32_t get_bits (std::uint32_t bitmask) const {
        return *(reinterpret_cast<volatile std::uint32_t *>(address)) &
             bitmask;
    }
};

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.8 hal/concepts.hpp

```
#pragma once

#include <concepts>

namespace hal {

template <typename T>
concept GPIO = requires (T pin) {
    { pin.configure() } -> std::same_as<bool>;
    { pin.set() } -> std::same_as<void>;
    { pin.reset() } -> std::same_as<void>;
};

} /* namespace hal */
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

## E.9 hal/led.hpp

```
#pragma once

#include "hal/concepts.hpp"

template <hal::GPIO TGpio>
class CLed {
private:
    bool m_state { false };
    TGpio* m_gpio { nullptr };
public:
    CLed () = delete;
    CLed (TGpio* gpio) : m_gpio(gpio) {
        m_gpio->configure();
    }

    void turn_on () {
        m_state = true;
        m_gpio->set();
    }

    void turn_off () {
        m_state = false;
        m_gpio->reset();
    }

    void toggle () {
        if (m_state) {
            turn_off();
        }
        else {
            turn_on();
        }
    }
};
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## E.10 hal/main.cpp

```
1 #include <cstdint>
2
3 #include "hal/gpio/gpio.hpp"
4 #include "led.hpp"
5
6 void delay (int cycles) {
7     volatile int i;
8     for (i = 0; i < cycles;) { i = i + 1; }
9 }
10
11 using namespace hal;
12
13 int main (void) {
14
15     CGpio<gpio_ports::port_a, gpio_pins::pin_06, gpio_modes::output>
           gpio {};
16
17     CLed<decltype(gpio)> led { &gpio };
18
19     while (true) {
20         led.toggle();
21         delay(200000);
22     }
23 }
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography

[1] B. A. Viktor Sehr, *C++ High Performance*. Packt Publishing, 2018, ISBN: 1787120953; 9781787120952.

[2] A. Rahnev, N. Pavlov, N. Valchanov, and T Terzieva, "Object oriented programming", *Lightning Source UK Ltd., London*, 2014.

[3] I. Plauska, A. Liutkevičius, and A. Janavičiūtė, "Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller", *Electronics*, vol. 12, no. 1, 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12010143. [Online]. Available: https://www.mdpi.com/2079-9292/12/1/143.

[4] M. S. (auth.), *Advanced C and C++ Compiling*, 1st ed. Apress, 2014, ISBN: 9781430266679; 1430266678; 9781430266686; 1430266686.

[5] S. Oualline, *Bare metal C : Embedded Programming for the Real world*, 1st ed. No Starch Press, 2022.

[6] J. Beningo, *Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques*, 1st ed. Apress, 2022, ISBN: 9781484282786; 1484282787; 9781484282793; 1484282795.

[7] E. Sutter, *Embedded Systems Firmware Demystified*, 1st edition. CMP, 2002, ISBN: 9781578200993; 1578200997.

[8] *Arm gnu toolchain*, Arm Limited, Accessed on 16.07.2023. [Online]. Available: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain.

[9] *Stm32f030f4*, STMicroelectronics, Accessed on 16.07.2023. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f030f4.html.

[10] *Stm32f0 series cortex-m0 programming manual*, PM0215, Rev. 2, STMicroelectronics, Feb. 2023.

[11] C. K. (auth.), *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*, 4th ed. Springer, 2021, ISBN: 3662629968; 9783662629963; 366262995X; 9783662629956.

[12] V. G. S. Wu, *Expert C++: Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features*. Packt Publishing Ltd, 2020, ISBN: 9781838554767; 1838554769.

[13] B. Stroustrup, *A Tour of C++*, 3rd ed. Addison-Wesley Professional, 2022, ISBN: 0136816487; 9780136816485.

[14] I. JTC1/SC22/WG21, "Working draft, standard for programming language C++", International Organization for Standardization, Technical Report N4861, 2020. [Online]. Available: https://wg21.link/n4861.

[15] B. Filipek, *C++17 In Detail: Learn the Exciting Features of The New C++ Standard!*, 1.0. Independently published, 2019, ISBN: 1798834065; 9781798834060.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[16] M. Weisfeld, *Object-Oriented Thought Process, 3rd Edition*. Addison-Wesley, 2008, ISBN: 9780672330162; 0672330164.

[17] A. T. Cohen, "Data abstraction, data encapsulation and object-oriented programming", *ACM SIGPLAN Notices*, vol. 19, no. 1, pp. 31–35, 1984.

[18] *Stm32f030x4 stm32f030x6 stm32f030x8 stm32f030xc*, DS9773, Rev. 4, STMicroelectronics, Jan. 2019.

[19] J. Jarvi, "Compile time recursive objects in c++", in *Proceedings Technology of Object-Oriented Languages. TOOLS 27 (Cat. No.98EX224)*, 1998, pp. 66–77. DOI: 10.1109/TOOLS.1998.713588.

[20] B. E. M. Stanley B. Lippman Josée Lajoie, *C++ Primer*, 5th ed. Addison-Wesley Professional, 2012, ISBN: 0321714113; 9780321714114.

[21] C. de Dinechin, "C++ exception handling", *IEEE Concurrency*, vol. 8, no. 4, pp. 72–79, 2000. DOI: 10.1109/4434.895109.

[22] J. Renwick, T. Spink, and B. Franke, "Low-cost deterministic c++ exceptions for embedded systems", in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019, Washington, DC, USA: Association for Computing Machinery, 2019, 76–86, ISBN: 9781450362771. DOI: 10.1145/3302516.3307346. [Online]. Available: https://doi.org/10.1145/3302516.3307346.

[23] M. Bancila, *Template Metaprogramming with C++: Learn everything about C++ templates and unlock the power of template metaprogramming*, 1st ed. Packt Publishing, 2022, ISBN: 1803243457; 9781803243450.

[24] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Comput. Surv.*, vol. 17, no. 4, 471–523, 1985, ISSN: 0360-0300. DOI: 10.1145/6041.6042. [Online]. Available: https://doi.org/10.1145/6041.6042.

[25] D. B. Kahanwal and D. T. Singh, "Function overloading implementation in c++", *arXiv preprint arXiv:1311.7203*, 2013.

[26] *Jinja2 template engine*, Accessed on 18.11.2023. [Online]. Available: https://palletsprojects.com/p/jinja/.

[27] *Clock_gettime - linux man page*, Accessed on 18.11.2023. [Online]. Available: https://linux.die.net/man/3/clock_gettime.

[28] T. Vladimirova, D. Eamey, S. Keller, and P. S. M. Sweeting, "Floating-point mathematical co-processor for a single-chip on-board computer", *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices and Technologies (MAPLD'2003)*, 2003.

[29] F. Merchant, A. Chattopadhyay, S. Raha, S. Nandy, and R. Narayan, "Accelerating blas and lapack via efficient floating point architecture design", *Parallel Processing Letters*, vol. 27, no. 03n04, p. 1 750 006, 2017.

[30] B. Stroustrup, "Exception safety: Concepts and techniques", in *Advances in Exception Handling Techniques*, A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 60–76, ISBN: 978-3-540-45407-6. DOI: 10.1007/3-540-45407-1_4. [Online]. Available: https://doi.org/10.1007/3-540-45407-1_4.


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub