Republic of Tunisia

Ministry of Higher Education

and Scientific Research

University of Carthage

National Engineering School of Carthage

Summer Internship Project Report

By

Youssef HASNAOUI

Running Coremark on STM32 devices using Csolution project structure

Professional supervisor: **Mohamed HAMROUNI**

Project Elaborated Within STMicroelectronics

Academic Year 2024 - 2025

Dedications

I dedicate this work to my family and friends.

*To my mother **Monia Sellami**, your love and support means so much to me in every step of my journey. You're the best.*

*To my father **Makram Hasnaoui**, thank you so much for all the sacrifices you've made for my sake. I could never be here without you.*

*To my sister **Aicha**, your kindness and your belief in me have been a constant source of inspiration, and for that, I am very grateful.*

To my friends, you have my most sincere gratitude for your support, your guidance and your company throughout this journey.

Thank you,

Youssef HASNAOUI

Acknowledgements

I would like to sincerely thank everyone who had a hand in the development of this project.

Firstly, I want to express my utmost appreciation to ***Mohamed HAMROUNI****, his guidance and support has been invaluable, and has kept me motivated and determined to give my best effort.*

Last but not least, to every single person that contributed to the successful completion of this project with their constant support and motivation, you have my utmost thanks.

Contents

List of Figures

List of Tables

List of Acronyms

- **ABI** = **A**pplication **B**inary **I**nterface
- **API** = **A**pplication **P**rogramming **I**nterface
- **ARM** = **A**dvanced **R**ISC **M**achine
- **BRE** = **B**asic **R**egular **E**xpression
- **BSP** = **B**oard **S**upport **P**ack
- **CFSR** = **H**ard**F**ault **S**tatus **R**egister
- **CLI** = **C**ommand-**L**ine **I**nterface
- **CMSIS** = **C**ortex **M**icrocontroller **S**oftware **I**nterface **S**tandard
- **CPU** = **C**entral **P**rocessing **U**nit
- **DAP** = **D**ebug **A**dapter **P**ort
- **DFP** = **D**evice **F**amily **P**ack
- **DRAM** = **D**ynamic **R**andom **A**ccess **M**emory
- **DSP** = **D**igital **S**ignal **P**rocessor

- **DWARF** = **D**ebugging **W**ith **A**ttributed **R**ecord **F**ormats
- **EEMBC** = **E**mbedded **M**icroprocessor **B**enchmark **C**onsortium
- **ELF** = **E**xecutable and **L**inkable **F**ormat
- **ERE** = **E**xtended **R**egular **E**xpression
- **FPU** = **F**loating **P**oint **U**nit
- **FPU** = **F**loating-**P**oint **U**nit
- **GCC** = **G**NU **C**ompiler **C**ollection
- **GDB** = **G**NU **D**e**b**ugger
- **GNU** = **G**NU's **N**ot **U**nix
- **GUI** = **G**raphical **U**ser **I**nterface
- **HAL** = **H**ardware **A**bstraction **L**ayer
- **HFSR** = **H**ard**F**ault **S**tatus **R**egister
- **I/O** = **I**nput/**O**utput
- **IAR** = **I**ngenjörsfirma **A**nders **R**undgren

- **IDE** = **I**ntegrated **D**evelopement **E**nvironment
- **IEEE** = **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
- **IP** = **I**ntegrated **P**eripheral
- **IV** = **I**nitialization **V**ector
- **JSON** = **J**ava**S**cript **O**bject **N**otation
- **JTAG** = **J**oint **T**est **A**ction **G**roup
- **LL** = **L**ow-**L**ayer
- **LLVM** = **L**ow **L**evel **V**irtual **M**achine
- **MCU** = **M**icro **C**ontroller **U**nit
- **MFLOPS** = **M**ega **Fl**oating-point **O**perations **P**er **S**econd
- **MOPS** = **M**ega **O**perations **P**er **S**econd
- **MVE** = **M**-Profile **V**ector **E**xtension
- **OCD** = **O**n-**C**hip **D**ebugger
- **PLM** = **P**roject **L**ifetime **M**anagement

- **POSIX** = **P**ortable **O**perating **S**ystem **I**nterface
- **RAM** = **R**andom **A**ccess **M**emory
- **ROM** = **R**ead-**O**nly **M**emory
- **RTE** = **R**un**T**ime **E**nvironment
- **RTOS** = **R**eal-**T**ime **O**perating **S**ystem
- **RegEx** = **Reg**ular **Ex**pression
- **SCB** = **S**ystem **C**ontrol **B**lock
- **SCVD** = **S**oftware **C**omponent **V**iewer **D**escription
- **SDRAM** = **S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
- **SRAM** = **S**tatic **R**andom **A**ccess **M**emory
- **SRE** = **S**imple **R**egular **E**xpression
- **SVD** = **S**ystem **V**iew **D**escription
- **SWD** = **S**ingle-**W**ire **D**ebug
- **SWO** = **S**erial **W**ire **O**utput

- **UART** = **U**niversal **A**synchronous **R**eceiver **T**ransmitter

- **USART** = **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter

- **YAML** = **Y**et **A**nother **M**arkup **L**anguage

Introduction

In today's technological landscape, the importance of benchmarking in embedded systems cannot be overstated. Embedded systems are integral to a wide range of applications, from consumer electronics and industrial automation to healthcare devices and automotive systems. Therefore, evaluations provide objective data on performance, efficiency and reliability, making them essential for making informed design and purchasing decisions. As systems grow more complex and interconnected, the number of configurable parameters and potential performance bottlenecks expands, increasing the risk of suboptimal performance, wasted resources and failed deployments. Establishing a rigorous and standardized benchmarking methodology is therefore paramount to ensure systems meet their intended requirements and deliver value in both development and production environments.

Benchmarking provides the fundamental framework for this objective analysis by establishing a standardized set of metrics and tests to quantify a system's performance. Effective benchmarking allows developers to measure key characteristics such as processing speed, memory throughput, power consumption, and real-time task execution, creating a data-driven basis for comparison. However, executing benchmarks in embedded systems comes with challenges. Developers must navigate complexities such as isolating the unit under test, minimizing the influence of external systems, selecting appropriate tools, and ensuring that the test conditions are consistent and reproducible across different platforms and devices. Inaccurate methodology or poorly designed tests can cause misleading results, rendering the data useless for making decisions.

The dependency on proprietary and platform-specific vendor tools can lock projects into a single ecosystem and hinder reproducibility. There is a growing need to have development environments that prioritize portability and vendor independence, allowing benchmarks to be built, deployed, and executed consistently across a wide range of hardware. By abstracting away toolchain-specific complexities, such environments allow developers to focus on the benchmark results themselves rather than the intricacies of the different build processes.

This report explores the implementation of the CoreMark benchmark on STM32 devices using CMSIS-Toolbox, and the related Csolution project structure as a step towards this goal. This method demonstrates a move away from proprietary IDE-bound setups and towards a portable, platform-independent driven workflow. By leveraging a toolchain-agnostic approach, we establish a reproducible environment for evaluations. This reduces vendor lock-in, enhances cross-platform compatibility, ensures the data is a reliable reflection of the hardware capabilities, and, as a side effect, allows for a fair comparison between the available toolchains.

Chapter 1

Internship Context

Contents

Internship Context

This opening chapter focuses on introducing the host company STMicroelectronics, and provides an overview of the company's activities and global presence. It also includes an introduction to ST Tunis and the Support Solutions team, which played a pivotal role in the execution of this project, as well as an outline of the project's overall framework. Following the presentation of the challenges, we will explore the current state of the art and finally present the proposed solution which is the main objective of this project.

1.1 Host Organization

1.1.1 STMicroelectronics

STMicroelectronics (ST) is a global leader in the semiconductor industry, providing innovative solutions across various markets, including automotive, industrial, personal electronics, and communications equipment.

Founded in 1987 through the merger of SGS Microelettronica of Italy and Thomson Semiconducteurs of France, ST has grown to become one of the largest semiconductor companies in the world.

1.1.2 Activity Sectors

ST's mission is to be the undisputed leader in the semiconductor market, delivering sustainable and innovative solutions that make a positive impact on people's lives. The company's vision is to create technology that enables a more intelligent and connected world, driving progress in key areas such as smart driving, smart industry, smart home, and smart city applications.

The following illustration highlights the key end markets targetted by ST's solutions [1].

Figure 1.1: STMicroelectronics Activity Sectors

1.1.3 Global Presence

STMicroelectronics operates in more than 35 countries, with a strong presence in key markets around the world. Figure 1.2 illustrates the company's global network, which includes [1]:

- **Manufacturing Sites:** ST has 14 main manufacturing sites, strategically located to serve its global customer base efficiently.
- **Sales & Marketing Offices:** With a network of sales and marketing offices, ST provides localized support and services to its customers.
- **R&D Centers:** ST's R&D centers are spread across the globe, fostering innovation and collaboration.

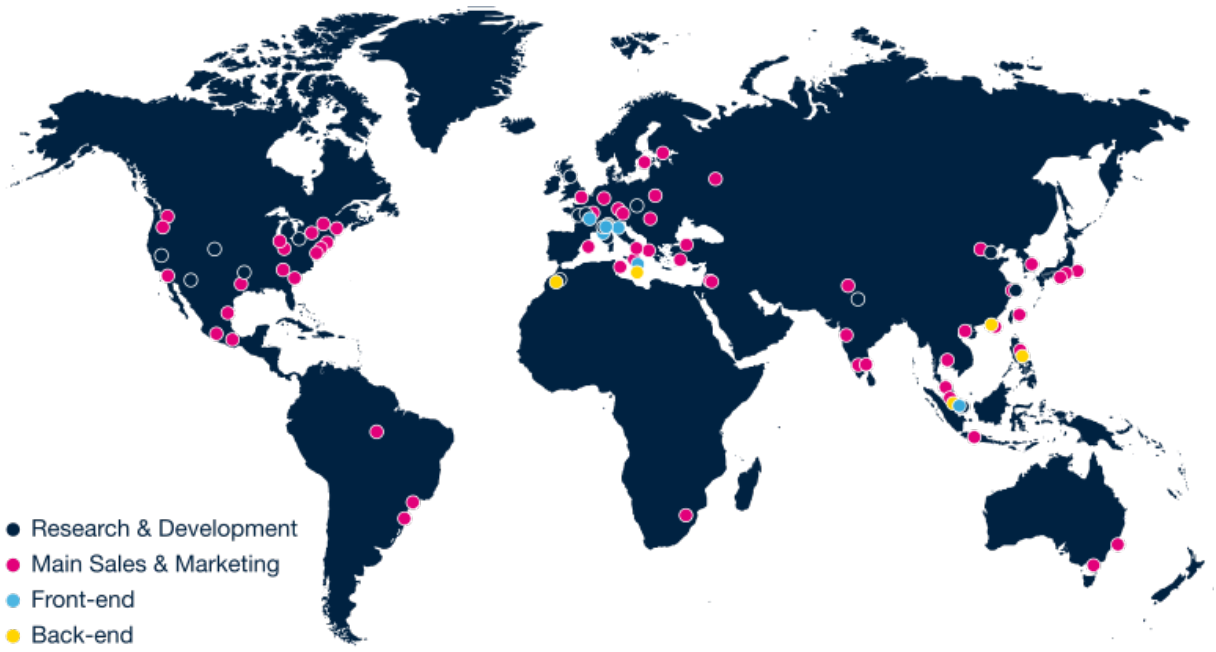

Figure 1.2: STMicroelectronics Worldwide Presence

1.1.4 ST Tunis

The STMicroelectronics Tunis site, located in the El Ghazela Technopark, employs over 100 professionals in various fields such as tools, applications, quality, and support functions. The facility is involved in all stages of the microelectronics industry, from initial circuit design to final verification before mass production.

With expertise in both hardware and software, the Tunis center's activities include electronic circuit design, software tool development, application software support, and the electrical validation and verification of new circuits. This diverse skill set allows the center to contribute significantly to ST's global operations.

Figure 1.3: STMicroelectronics Tunis

1.1.5 ST Support Solutions

The ST Support Solutions team is dedicated to delivering comprehensive customer support while managing and publishing STM32 product documentation. Their role includes providing in-depth technical support for ST's tools and products, helping customers troubleshoot problems and optimize their use of these technologies. They ensure the accuracy, clarity, and consistency of all technical documentation, which is regularly updated to reflect the latest developments and innovations. The team is also involved in enhancing and maintaining existing products, as well as defining new products, derivatives, and application references to meet evolving market needs. Additionally, they lead initiatives related to intellectual property applications, offering specific support through the development of application notes, training programs, and other resources that help customers fully understand and utilize ST's products. This multifaceted approach ensures both customer satisfaction and continuous improvement in the STM32 ecosystem.

1.2 Project Context

Benchmarking microcontrollers is a fundamental step in evaluating their computational capabilities, power efficiency, and suitability for various applications, such as IoT, robotics, or industrial control. Among the many available benchmarks, CoreMark, developed by EEMBC, has become the de facto standard for embedded systems because it provides a well-defined, portable, and reliable performance metric. STM32 microcontrollers, are widely adopted in the embedded systems industry due to their scalability, rich peripheral set, and performance-to-cost ratio.

The **CMSIS** initiative, led by Arm, and its Open-CMSIS-Pack ecosystem aim to standardize microcontroller development workflows. Within this ecosystem, Csolution provides a modern, metadata-driven project structure

that enables cross-platform builds, device abstraction, and automation. Leveraging Csolution can significantly simplify benchmarking workflows by providing a unified and reproducible project generation and build process.

1.3 Problem Statement

Running CoreMark on STM32 devices traditionally requires manual setup for each device, including:

- Creating and configuring individual projects for different STM32 families.
- Managing peripheral initialization.
- Providing a clock source to Coremark.
- Handling different compiler options and toolchains.
- Customizing linker scripts manually.
- Maintaining multiple project files for IDEs like Keil µVision, IAR EWARM, or others.

This manual process is time-consuming, error-prone, and non-scalable, especially when benchmarking a wide range of devices. There is currently no standardized, automated workflow that allows developers to quickly generate CoreMark-ready projects for different STM32 targets while ensuring consistency and portability.

1.4 State of the Art

The current approaches to running benchmarks on STM32 devices typically rely on:

- **Vendor-Specific IDEs:** IDEs provide an easy way to manage projects via a graphical interface, they typically provide tools for peripheral configuration, memory management and compiler options. They also offer integrated build and debug environments.
- **Standalone CoreMark Implementations:** EEMBC provides CoreMark source code with minimal reference implementations, it is up to the developer to manually adapt it to the target device, providing startup and initialization code, memory configuration and a clock source.
- **Custom Build Systems:** Some environments require the use of custom build systems such as CMake and Make, these tools allow the developer to manually manage dependencies, defines and other C/C++ related build configuration.
- **CMSIS Packs:** CMSIS packs offer a standardized way of packaging drivers, middleware and device specific files, they provide metadata to describe the device's memory and peripheral layout.

1.5 Critique of Current State of the Art

While the above methods work, they exhibit several limitations:

Table 1.1: Comparison of Approaches for Embedded Benchmarking and Automation

Approach	Advantages	Limitations
Vendor-Specific IDEs	Intuitive GUI, built-in drivers, easy to use	Projects are IDE-specific, hard to automate, poor scalability
Standalone CoreMark	Portable reference code on computers	Significant manual effort for adaptation to each MCU
Custom Build Systems	Flexible, automation-friendly	Requires deep expertise, no standardized metadata integration
CMSIS Packs	Standardized packaging and metadata support	Lack seamless integration with benchmarking and automation

1.6 Proposed Solution

The proposed solution centers around the aspects of automation and standardization, leveraging the **CMSIS Solution project structure**, part of the **Open-CMSIS-Pack** ecosystem in order to achieve the desired outcome. The key aspects of the solution are:

- **Unified Project structure** By making use of the csolution project files, we can describe build configuration, dependencies, and device parameters in a centralized way, simplifying maintenance and reuse.
- **Automating Project Generation** While the **Csolution project structure** offers a standardized manner of configuring projects, the source code of the application has to be managed in some way, although generators like **STM32CubeMX** does a decent job of generating peripheral configuration, it is still slow and GUI based, so it can't be used in a fully automated environment. Therefore, we can make use of creating a tool to handle automating source code and linker script generation, device configuration and even generating the csolution project files.
- **Portable and Cross-Toolchain Support** Allow the project to be compiled by the main toolchains used in embedded applications, through **CMSIS-Toolbox**, as well as ensuring the project can be built and debugged in multiple IDE environments.
- **CoreMark Integration** Embed the **CoreMark** source code into the standardized generation template, making it easily portable amongst **all** STM32 MCU targets.

1.7 Objective

The main objective of this project is to develop a standardized, automated workflow for running CoreMark on STM32 devices using the Csolution project structure.

1.8 Conclusion

Benchmarking embedded systems is essential for performance evaluation and hardware selection. However, the current manual approaches to running CoreMark on STM32 devices are inefficient, inconsistent, and hard to scale.

This project addresses these challenges by introducing a Csolution-based workflow that integrates device metadata, automation, and portable build configurations. By leveraging the Open-CMSIS-Pack ecosystem, the proposed solution provides a unified, reproducible, and scalable benchmarking framework.

The outcome is a streamlined process where CoreMark can be quickly deployed to any STM32 target, results can be gathered reliably, and the entire workflow can serve as a foundation for future benchmarking or performance evaluation tasks across diverse embedded platforms.

CHAPTER 2

THEORY AND KEY CONCEPTS

Contents

Introduction

In this chapter, we will explore the key concepts essential to our project. We begin by introducing the STM32 ecosystem, providing an overview of its components and their functionalities. Following this, we delve into the theoretical aspects of benchmarking and embedded build systems relevant to our work. We then present the open-CMSIS-pack ecosystem. Afterwards, we will talk about loading and debugging embedded systems applications. Finally we will discuss the essentials of automating project generation.

2.1 STM32 Microcontrollers (MCUs)

2.1.1 STM32 Series

STM32 microcontrollers (MCUs) are a family of 32-bit microcontrollers based on the ARM Cortex-M processor. Developed by STMicroelectronics, the STM32 series offers a wide range of products that cater to various applications, from simple embedded systems to complex industrial automation. The STM32 MCUs are categorized into several series, each designed to meet specific application requirements. Some of the most commonly used series are the **STM32F** and the **STM32H** series.

2.1.2 ARM Cortex-M

The STM32 microcontrollers are built around the ARM Cortex-M cores, which are designed for efficient and high-performance processing in embedded systems. The ARM Cortex-M family includes several cores, such as Cortex-M0, Cortex-M4, Cortex-M7, and Cortex-M85, each offering different levels of performance and features.

STM32 MCUs are grouped into families, a combination of a series and an ARM Cortex-M core, the number proceeding the STM32 series indicates the core, for example, STM32H7 family indicates a microcontroller from the STM32H series with a Cortex-M7 core, while the STM32N6 family indicates a STM32N series microcontroller with a Cortex-M55 core.

2.1.3 STM32 UART Peripheral

STM32 microcontrollers come with a rich set of integrated peripherals (IPs) that enhance their functionality and enable developers to build complex and feature-rich applications. The one of interest in this report is the UART peripheral. UART is a hardware protocol and communication interface that uses two wires for data exchanges between peripherals that do not share a common clock signal. It fits our use case of transmitting data from the microcontroller to the host computer in order to have access to Coremark results.

2.1.4 STM32Cube Firmware Package

The STM32Cube Firmware Package is a comprehensive software suite provided by STMicroelectronics to accelerate development on STM32 microcontrollers. Its components include but are not limited to:

- **CMSIS:** A vendor-independent standard defined by Arm that provides a consistent API for Cortex-M cores, STM32Cube Firmware provides an implementation of the CMSIS-Core layer, these include device and family specific header files, as well as templates for startup files and linker scripts.
- **HAL:** Simplifies peripheral configuration and access through high-level APIs, reducing development complexity and allows for re-usability across different hardware.
- **LL Drivers:** Provide fine-grained control of peripherals with minimal overhead, suitable for performance-critical applications.

2.2 Benchmarking

2.2.1 Overview

Benchmarking is the systematic process of evaluating and comparing the performance of a computing system, or a specific component within that system, by running a standardized set of tasks and synthetic workloads. The key aspects of benchmarks include:

- **Metrics:** Performance is measured using quantifiable units, Common metrics include:
  - **Throughput:** The amount of work done per unit of time (e.g., operations/second, frames/second, MB/s).
  - **Latency:** The time taken to complete a single operation (e.g., microseconds per operation).
  - **Power Efficiency:** Performance achieved per watt of power consumed (e.g., points per watt, inferences per joule).
  - **Memory Usage:** The amount of RAM or cache consumed during a task.

- **Benchmark Types:**
  - **Synthetic Benchmarks:** These are specialized programs designed to stress specific subsystems like the CPU, memory, or GPU. They provide standardized but often abstract results.
  - **Application Benchmarks:** Use real-world software and workloads (e.g., rendering a video file, compiling a large code-base, running a specific game at a set quality). These measure performance in practical, user-facing scenarios.
  - **Micro-benchmarks:** Isolate and test a very specific, low-level operation (e.g., floating-point multiplication speed, memory access latency).

2.2.2 Benchmarking Microcontrollers

Although the same benchmarks can be run on most pieces of technology, the implementation differs slightly for microcontrollers. Computers can delegate tasks such as scheduling and printing to the underlying operating system, which is not the case when it comes to embedded devices. We have to take into account handling threads, I/O operations, memory regions, and code sections. This can make the implementation tricky at times. The following steps are necessary to ensure proper benchmarking on a microcontroller:

- **Clock Source Provision:** Benchmarks rely on CPU ticks in order to get an accurate estimate of the time taken to run the workloads, providing access to the tick counter, whether it be the internal System Clock or an external timer, as well as the frequency of said clock is crucial for proper measurements.
- **Print Logic Implementation:** Computers can offer multiple ways of getting data from a program, such as logging them into files or printing them to the standard output, this option is not available on microcontrollers, therefore, we have to provide our own mechanism of data transmission. In our context, the print logic was implemented to send data via the UART peripheral.
- **Code Execution Management:** For high-end systems relying on an operating system, the most common approach is to load the program from disk into volatile memory(RAM) and begin execution without any extra steps. However, in our case, we have to specify the regions of memory where each part of the program will reside, the most common types of memory are: FLASH, SRAM, ITCM/DTCM, SDRAM, NVMe, all of them can be either external or internal. Depending on multiple factors, such as wait-states, clock synchronization and the communication interface, the results can have vastly varying results depending on the chosen regions.

2.2.3 Coremark

There have been many attempts to provide a single number that can totally quantify the ability of a CPU. Be it MHz, MOPS, MFLOPS - all are simple to derive but misleading when looking at actual performance potential. EEMBC's CoreMark is a benchmark that measures the performance of microcontrollers (MCUs) and central processing units (CPUs) used in embedded systems. It is designed to run on devices from 8-bit microcontrollers to 64-bit microprocessors. CoreMark ties a performance indicator to execution of simple code, but rather than being entirely arbitrary and synthetic, the code for the benchmark uses basic data structures and algorithms that are common in practically any application

Coremark Composition

To appreciate the value of CoreMark, it's worthwhile to dissect its composition, which in general is comprised of lists, strings, and arrays (matrixes to be exact). Lists commonly exercise pointers and are also characterized by non-serial memory access patterns. In terms of testing the core of a CPU, list processing predominantly tests how fast data can be used to scan through the list. For lists larger then the CPU's available cache, list processing can also test the efficiency of cache and memory hierarchy.

2.2.3.1 List Processing

List processing consists of reversing, searching or sorting the list according to different parameters, based on the contents of the list data items. In particular, each list item can either contain a pre-computed value or a directive to invoke a specific algorithm with specific data to provide a value during sorting. To verify correct operation, CoreMark performs a 16b cyclic redundancy check (CRC) based on the data contained in elements of the list. Since CRC is also a commonly used function in embedded applications, this calculation is included in the timed portion of the CoreMark.

2.2.3.2 Matrix Processing

Many algorithms use matrixes and arrays, warranting significant research on optimizing this type of processing. These algorithms test the efficiency of tight loop operations as well as the ability of the CPU and associated toolchain to use ISA accelerators such as MAC units and SIMD instructions. These algorithms are composed of tight loops that iterate over the whole matrix. CoreMark performs simple operations on the input matrixes, including multiplication with a constant, a vector, or another matrix. CoreMark also tests operating on part of the data in the matrix in the form of extracting bits from each matrix item for operations. To validate that all operations have been performed, CoreMark again computes a CRC on the results from the matrix test.

2.2.3.3 State machine processing

An important function of a CPU core is the ability to handle control statements other than loops. A state machine based on switch or 'if' statements is an ideal candidate for testing that capability. There are 2 common methods for state machines – using switch statements or using a state transition table. Because CoreMark already utilizes the latter method in the list processing algorithm to test load/store behavior, CoreMark uses the former method, switch and 'if' statements, to exercise the CPU control structure. The state machine tests an input string to detect if the input is a number, if it is not a number it will reach the "invalid" state. This is a simple state machine with 9 states. The input is a stream of bytes, initialized to ensure we pass all available states, based on an input that is not available at compile time. The entire input buffer is scanned with this state machine.

2.2.3.4 CoreMark Profiling

Since CoreMark contains multiple algorithms, it is interesting to demonstrate how the behavior changes over time. For example, looking at the percentage of control code executed (samples taken at each 1000 cycles) and branch mis-predictions in Figure 2.1, it is obvious where the matrix algorithm is being called. This is portrayed by the low mis- prediction rate and high percentage of control operations, indicative of tight loops) (for example, between points 330-390).

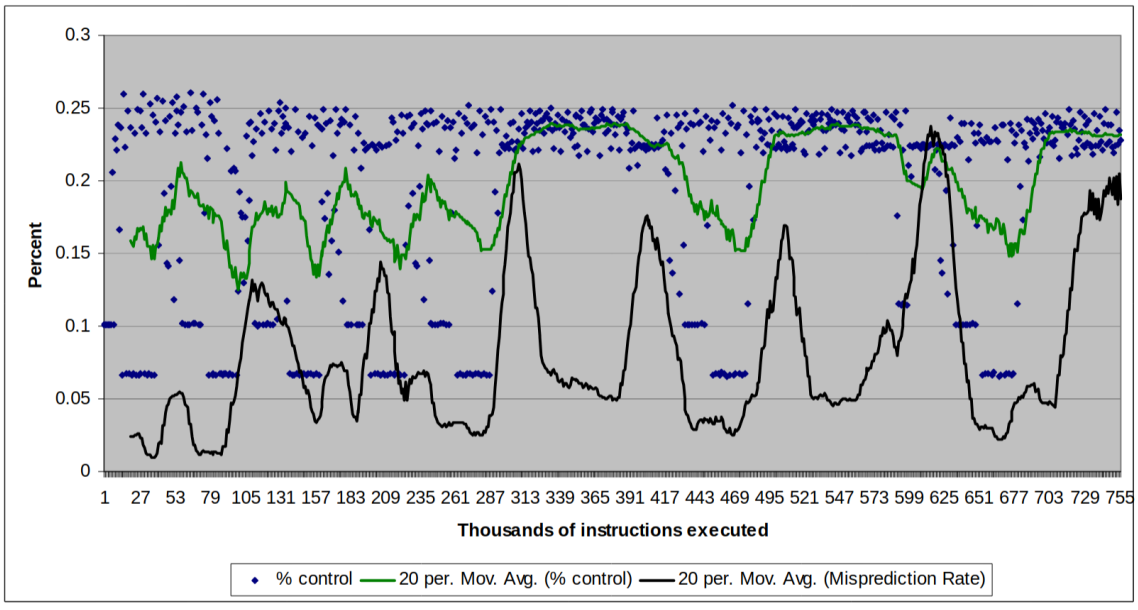

Figure 2.1: Distribution of control instructions and mispredictions over CoreMark execution.

Overall CoreMark is well suited to comparing embedded processors. It is small, highly portable, well understood, and highly controlled. CoreMark verifies that all computations were completed correctly during execution, which helps debug any issues that may come up. The run rules are clearly defined and reporting rules are enforced on the CoreMark web site.

2.3 C/C++ Build Process

The construction of executable firmware for embedded systems necessitates interventions throughout the compilation toolchain, this requirement is driven by the absence of standardized hardware and resource constraints. Consequently, in our context, the build process is characterized by explicit configuration at each stage, including compiler optimizations, targeted assembly integration, and memory management via linker scripts.

2.3.1 Compilation Toolchain

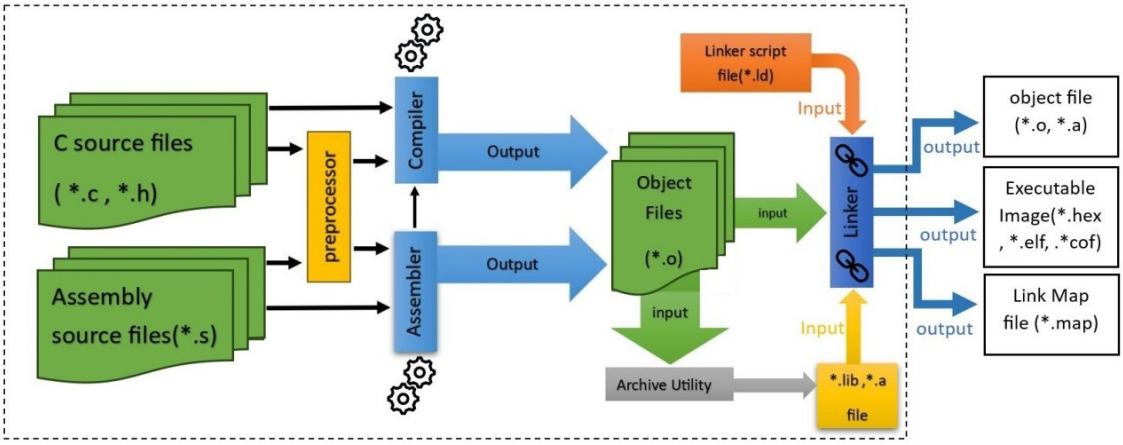

Figure 2.2: Compilation Process Diagram

Going from source code in C/C++ to executable firmware involves multiple steps, in this subsection, we will go over them, highlighting the importance of each procedure.

- **Preprocessor :** preprocessing is the first step of the build process, it expands macros and resolves defines as well as stripping comments from the code.
- **Assembler :** The assembler is the part of the toolchain that translates the high-level code into assembly code, which is the human-readable format of machine instruction, this step is done on a per-file basis.
- **Compiler :** Although the we use the term compilation as an equivalent to the whole process, compilation at its core is translating code into machine instructions, this step is what takes assembly mnemonics from each file and turns them into machine code also known as object files.
- **Linker :** The linking stage is the final and most important part of the process, since every c file is compiled separately into its own object file, they are not executable by default, the linker is what declares memory regions and associates symbols to the appropriate sections.

2.3.2 Embedded Systems Toolchains

While a standard toolchain for native development produces binaries that run directly on the host machine, an **embedded systems toolchain** targets a separate device with its own architecture, memory constraints, and hardware interfaces. This difference introduces several additional requirements beyond those of a regular host toolchain.

The most significant distinction is that an embedded toolchain must generate **cross-compiled code**. The compiler, assembler, and linker are configured to emit machine code compatible with the target processor's instruction set (e.g., ARM Cortex-M), rather than the host CPU. This involves selecting the correct ABI and handling architectural details such as endianness, hardware floating-point support, or specialized instruction extensions. In addition to basic code generation, embedded toolchains typically include:

- **Device-Specific Startup Code:** Since embedded systems do not run an operating system by default, the toolchain must provide initialization code that configures the processor state after reset. This includes setting up the stack pointer, initializing memory sections, and defining the interrupt vector table.
- **Linker Scripts for Memory Mapping:** Unlike host systems where memory layout is abstracted by the operating system, embedded applications must explicitly define where code, data, and peripherals reside in memory. The linker script enforces this mapping, ensuring that the binary fits within the device's regions.
- **Runtime Support Libraries:** Embedded toolchains include specialized implementations of standard libraries designed to operate without an operating system. These lightweight libraries provide essential functionality such as math operations or memory management while avoiding features that rely on system calls.
- **Debugging Interfaces:** Many embedded toolchains integrate support for hardware debugging through interfaces like SWD or JTAG. This enables loading binaries onto the target, setting breakpoints, and inspecting registers or memory directly on the device.
- **Output Conversion Utilities:** Since microcontrollers typically require firmware images in raw binary or hex formats, embedded toolchains include utilities to convert the standard elf into target-specific formats such as Intel HEX or Motorola S-Record.

2.3.3 Build Runners

While small embedded projects can be compiled by invoking individual commands manually, this approach quickly becomes impractical as the codebase grows. Modern projects often contain hundreds of source files, complex dependencies, and different build configurations for debugging or optimization.

A **build runner** (or build automation tool) automates this process by defining a set of rules that specify how source files should be transformed into the desired output. At its core, a build runner:

- Tracks dependencies between source files and generated files.
- Determines which parts of the project need to be rebuilt when a file changes.
- Invokes the appropriate compiler, assembler, or linker commands in the correct order.

Build runners operate based on explicit instructions provided through configuration files or scripts.

2.3.4 Build Systems

A **build system** is a higher-level abstraction built on top of build runners. While a build runner executes rules, a build system focuses on generating and managing those rules for complex projects, often in a platform-agnostic way.

Build systems provide several key advantages:

- **Portability:** They enable the same project to be built on different platforms or with different toolchains without rewriting build scripts.
- **Configuration Management:** They support multiple build configurations, such as debug or release builds, with different compiler flags and options.
- **Scalability:** They handle large projects with multiple modules, libraries, and external dependencies.

In practice, a build system generates low-level build instructions for a runner. For example, it may produce a set of dependency files and rules that a runner can execute efficiently. This separation of concerns allows developers to describe the project structure and relationships at a higher level, without directly managing every compiler or linker invocation.

In the context of embedded systems, build systems are particularly valuable because they can integrate device-specific settings, such as memory layouts or hardware abstraction layers, into the build process while remaining flexible enough to support multiple target devices and toolchains.

2.4 Debugging

Debugging embedded systems involves identifying and resolving issues that may occur in both hardware and software components. Unlike traditional software debugging, embedded debugging must account for real-time constraints, limited system visibility, and direct interaction with hardware peripherals. STM32 microcontrollers provide built-in debugging interfaces and features that greatly facilitate this process.

2.4.1 Challenges of Embedded Debugging

Embedded debugging presents unique challenges:

- **Limited Visibility:** Unlike desktop systems, embedded devices lack standard output and comprehensive logging mechanisms, making it harder to inspect internal states.
- **Real-Time Behavior:** Debugging can disrupt timing-sensitive operations, potentially masking or introducing bugs.
- **Hardware Dependence:** Failures may stem from peripheral misconfiguration, incorrect clock settings, or external circuitry issues.
- **Resource Constraints:** Limited memory and processing power restrict the use of advanced debugging techniques.

2.4.2 Debugging Interfaces and Tools

In the case of STM32 devices, they integrate hardware support for debugging through two primary interfaces:

- **SWD:** A two-pin protocol widely used for STM32 devices. It allows memory inspection, breakpoint handling, and peripheral monitoring with minimal I/O overhead.
- **JTAG:** A more comprehensive four-wire interface offering additional features such as boundary scan testing.

These interfaces are typically accessed via debug probes such as ST-LINK, J-Link, or CMSIS-DAP. They are supported by a wide range of IDEs and toolchains, including STM32CubeIDE, Keil µVision, and open-source solutions like OpenOCD and GDB.

2.4.3 Core Debugging Techniques

The following techniques are commonly used when debugging MCU applications:

- **Breakpoints and Stepping:** Halt execution at specific lines to inspect variables and system state.
- **Watchpoints:** Trigger a halt when a specific memory location is read or written, useful for diagnosing memory corruption or stack overflows.
- **Peripheral Inspection:** Live monitoring of hardware registers to verify correct peripheral configuration.
- **Tracing:** Using the SWO pin to stream real-time events such as ‘printf‘-style logs without blocking the CPU.

2.4.4 Firmware-Based Debugging Aids

Alongside hardware debuggers , simple firmware techniques can provide additional help in order to diagnose some issues:

- **UART Logging:** Redirecting output to a UART peripheral for runtime logging. Though simple, it can introduce timing delays.
- **LED and GPIO Debugging:** Toggling pins to indicate execution flow or measure timing with an oscilloscope or logic analyzer.
- **Watchdog Management:** Pausing watchdog timers during debugging using the DBGMCU registers to prevent unintended resets.

2.5 Open-CMSIS-Pack

Software compatibility for component re-use has long been a challenge in the microcontroller space, which is much more diverse at the hardware level compared to PCs. Open-CMSIS-Pack removes this complexity, delivering a standard for software component packaging and related foundation tools for validation, distribution, integration, management, and maintenance.

2.5.1 CMSIS-Packs

CMSIS-Packs are software components that contain device specific startup code, peripheral drivers, middleware, and board support packages. They provide a standardized delivery mechanism for software components and enable consistent project configuration across different development environments.

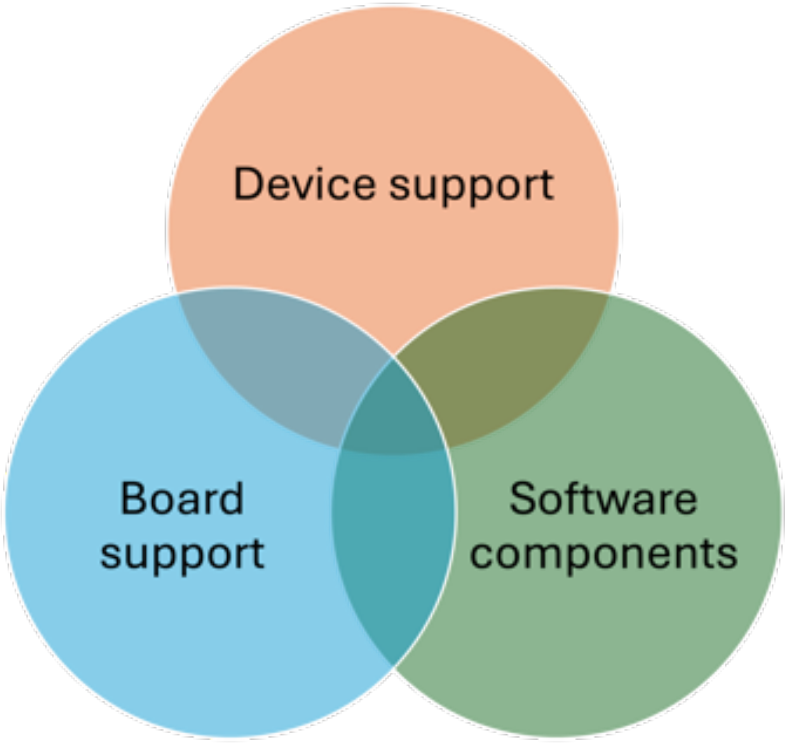

Figure 2.3: Software Packs Types

2.5.2 CMSIS-Pack Format

The CMSIS-Pack format is used to deliver a software package and is aimed to be scalable for future requirements. It provides a management process and supports a tool independent distribution for:

- **Device Support :**
  - Information about the processor and it's features.
  - C and assembly files for the device startup and access to the memory mapped peripheral registers.
  - Parameters, technical information, and data sheets about the device family and the specific devices.
  - Device description and available peripherals.
  - Memory layout of internal and external RAM and ROM address ranges.
  - Flash algorithms for programming the device.
  - Debug and trace configurations as well as System View Description files for device specific display of the memory mapped peripheral registers.
- **Board Support :**
  - Information about the development board and it's features.
  - Parameters, technical information, and data sheets about the board, the mounted microcontroller, and peripheral devices.

  - Drivers for on-board peripheral devices

- **Software components :**

  - A collection of source modules, header and configuration files as well as libraries.
  - Documentation of the software, including features and APIs.

2.5.3 Software Components

A software component encapsulates a set of related functions. They can contain C/C++ source files, object code, assembler files, header files, or libraries. The interfaces of software components should be defined with APIs to make them substitutable by other compatible components at design time. CMSIS software components can also refer to multiple interfaces of other software components. This could be also a hardware abstraction layer for a device peripheral. Configuration files contain application specific parameters for a software component. These files are typically copied to the user project workspace; all other files are not modified by the user and can remain in a separate location which avoids that a project workspace is polluted by many source files that should be considered as "black-box" elements by the application programmer.

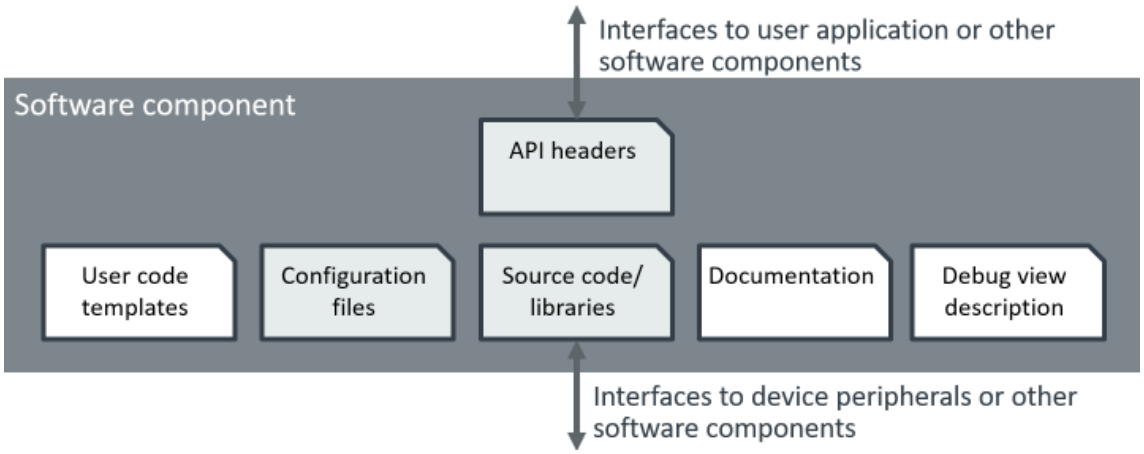

Figure 2.4: Software Component Interface

2.5.3.1 Component classification

A component lists the files that belong to it and that are relevant for a project. The component itself or each individual file may refer to a condition that must resolve to true; if it is false, the component or file is not applicable in the given context.

Each software component must have the following attributes that are used to identify the component:

- **Component Class (Cclass):** A component class which is a top-level component name, for example CMSIS, Device, File System
- **Component Group (Cgroup):** A component group name, for example CMSIS:RTOS, Device:Startup,

File System:CORE

- **Component Version (Cversion):** the version number of the software component.

2.5.4 CMSIS Solution Project Structure

As an effort to standardize the embedded software ecosystem, the csolution(CMSIS Solution) project structure came into place, it is a set of configuration files that is meant to describe your overall application, from the workspace, to projects, software layers and default configurations. This approach allows the user to centralize all the configuration of an application.

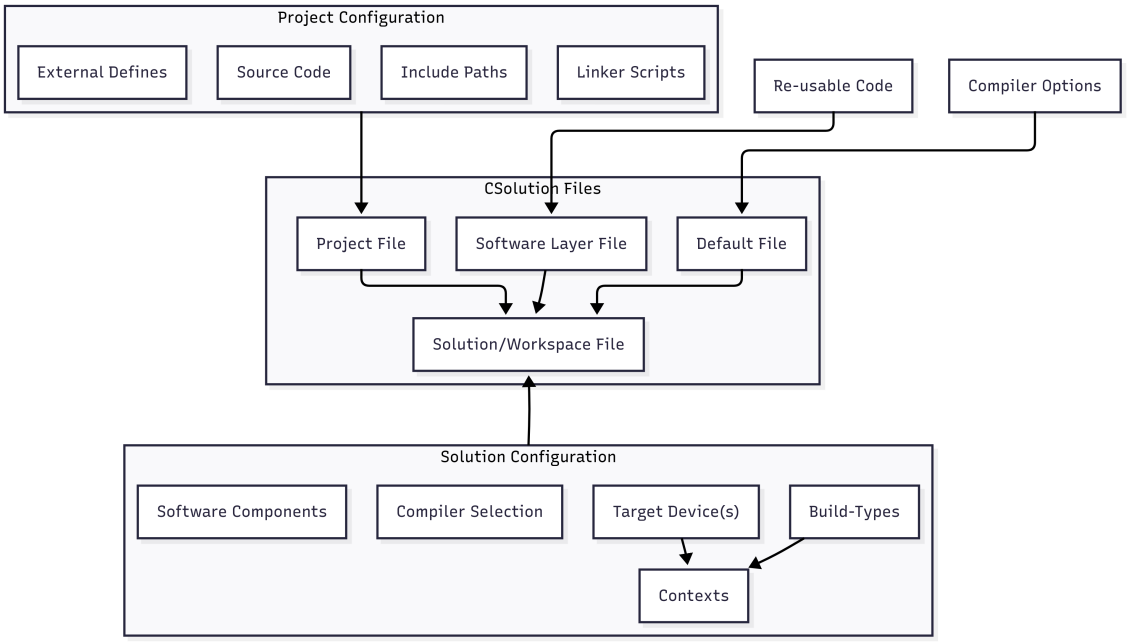

Figure 2.5: CMSIS Solution Project Structure

2.5.4.1 CMSIS solution files

The CMSIS-Toolbox gets its information from the csolution project files, these are YAML configuration files used to describe the application's context:

- **Solution file** A solution is the software view of the complete system. It combines projects that can be generated independently and therefore, manages related projects. It also specifies the targeted device(s) and build type(s). The file has the format *.csolution.yml.
- **Project file** The *.cproject.yml file has the content of a single independent build step, it specifies the source files to compile, sets the include directories and the necessary defines.
- **Layer file** Software layers collect source files and software components along with configuration files for reuse in different projects. Software Layers gives projects a better structure and simplifies:
  - Development flows with evaluation boards and production hardware.

  - Evaluation of middleware and hardware modules across different microcontroller boards.
  - Code reuse across projects, i.e. board support for test-case deployment.
  - Test-driven software development on simulation model and hardware.

- **Default Configuration file** The cdefault.yml file contains a common set of compiler-specific settings that select reasonable defaults with miscellaneous controls for each compiler.

2.6 Project Generation

When trying to run benchmarks across a wide variety of MCUs, the project structure follows the same pattern, and the dependencies can be expressed in a logical way relating to the device's metadata, therefore, automating the process will prove handy. There are multiple techniques to be used to generate a fully working project from scratch.

2.6.1 Regular Expressions:

The phrase regular expressions, or regexes, is often used to mean the specific, standard textual syntax for representing patterns for matching text. Each character in a regular expression (that is, each character in the string describing its pattern) is either a metacharacter, having a special meaning, or a regular character that has a literal meaning. A regular expression, often called a pattern, specifies a set of strings required for a particular purpose.

2.6.1.1 Operations

Most formalisms provide the following operations to construct regular expressions.

- **Boolean "OR":** A vertical bar separates alternatives. For example, gray|grey can match "gray" or "grey".
- **Grouping** Parentheses are used to define the scope and precedence of the operators (among other uses). For example, gray|grey and gr(a|e)y are equivalent patterns which both describe the set of "gray" or "grey".
- **Quantification** A quantifier after an element (such as a token, character, or group) specifies how many times the preceding element is allowed to repeat. The available quantifications are:
  - **Zero or One:** Denoted by the question mark **"?"** symbol.
  - **Zero or More:** Denoted by the asterisk **"*"** (derived from the Kleene star).
  - **One or More:** Denoted by the plus symbol **"+"**.
  - **N or More:** Denoted by the expression : **"{N, }"** where N is a positive integer.
  - **N or Less:** Denoted by the expression : **"{ ,N}"** where N is a positive integer.

  - **Between N and M times:** Denoted by the expression: **"{N, M}"** where N and M are positive integers such that N < M.

- **Wildcard:** Denoted by the dot **"."** it matches any single character

These operations can be combined interchangeably to create complex expressions to describe the desired system.

2.6.1.2 IEEE POSIX Standard

The IEEE POSIX standard has three sets of compliance: BRE (Basic Regular Expressions), ERE (Extended Regular Expressions), and SRE (Simple Regular Expressions). SRE is deprecated, in favor of BRE, as both provide backward compatibility. BRE and ERE work together. ERE adds ?, +, and |, and it removes the need to escape the metacharacters ( ) and { }, which are required in BRE. Furthermore, as long as the POSIX standard syntax for regexes is adhered to, there can be, and often is, additional syntax to serve specific (yet POSIX compliant) applications. Although POSIX.2 leaves some implementation specifics undefined, BRE and ERE provide a "standard" which has since been adopted as the default syntax of many tools, where the choice of BRE or ERE modes is usually a supported option. Perl regexes have become a de facto standard, having a rich and powerful set of atomic expressions. Perl has no "basic" or "extended" levels. As in POSIX EREs, ( ) and { } are treated as metacharacters unless escaped; other metacharacters are known to be literal or symbolic based on context alone. Additional functionality includes lazy matching, backreferences, named capture groups, and recursive patterns.

2.6.2 Templates & File Manipulation

File generation templates are pre-defined structures or blueprints used to create new files with standardized content and formatting.

2.6.2.1 Template Definition

A template file is created containing boilerplate text, placeholders for dynamic content (variables), and sometimes logic for conditional generation.

2.6.2.2 Parameterization

The template defines parameters that can be filled in by the user or a program when generating a new file. These parameters can be single-valued, arrays or maps containing relevant information.

2.6.2.3 Template Execution

When a new file is needed, the template is selected, and the required parameters are provided. A generation engine then processes the template, replacing placeholders with the provided values and executing any embedded

logic to produce the final file.

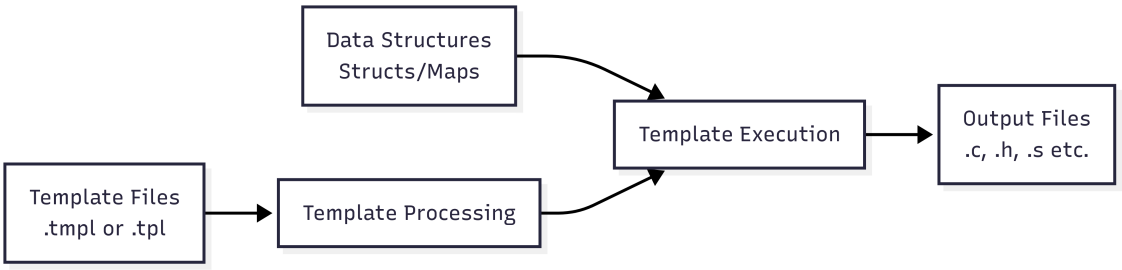

Figure 2.6: Templating Process

2.6.3 CMSIS-Toolbox Generators

Generators, such as STM32CubeMX or MCUXpresso Config Tools, simplify the configuration for devices and boards. The CMSIS-Toolbox implements a generic interface for generators. They may be used to:

- Configure device and/or board settings, such as clock configuration or pinout.
- Add and configure software drivers, for example, for UART, SPI, or I/O ports.
- Configure parameters of an algorithm, such as DSP filter design or motor control parameters.

2.6.3.1 Generator Integration (STM32CubeMX Example)

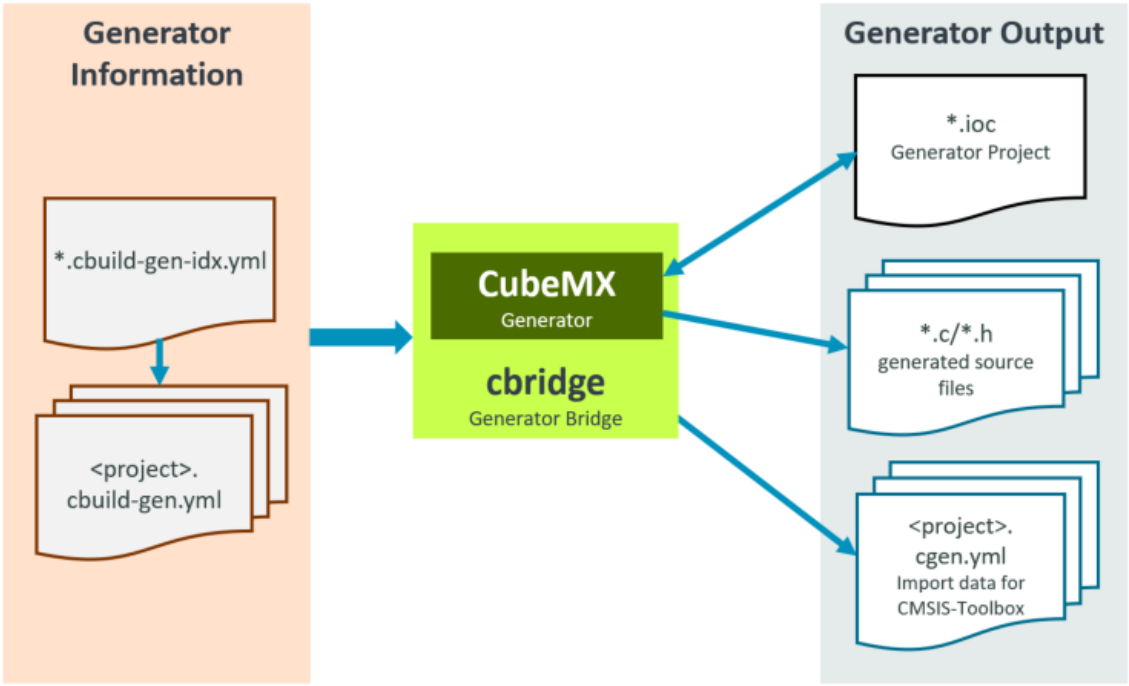

Figure 2.7: Generator Integration in CMSIS-Toolbox

The Figure 2.7 shows how the STM32CubeMX generator is integrated into the CMSIS build process. The data flow is exemplified on STM32CubeMX (Generator ID for this example is CubeMX). The information about the

project is delivered to the generator using the Generator Information files (<solution-name>.cbuild-gen-idx.yml and <context>.cbuild-gen.yml). This information provides CubeMX with the project context, such as the selected board or device, and CPU mode, such as TrustZone, disabled/enabled. The utility cbridge gets as parameter the <solution-name>.cbuild-gen-idx.yml and calls the generator. For the CubeMX generator example, these files are created:

- *.ioc CubeMX project file with current project settings
- *.c/.h source files, i.e. for interfacing with drivers
- <project-name>.cgen.yml (created by cbridge) provides the data for project import into the csolution build process.

Conclusion

This chapter has provided an overview of the key theoretical concepts and technologies relevant to the project. We began by exploring the STM32 microcontroller ecosystem, including the various series and ARM Cortex-M cores that form the foundation of these devices. The STM32Cube firmware package was introduced as a comprehensive software suite for STM32 development.

We then examined the principles of benchmarking, with a particular focus on the CoreMark benchmark and its application to microcontrollers. The C/C++ build process for embedded systems was detailed, highlighting the unique requirements and toolchains needed for cross-compilation.

The chapter also covered debugging techniques specific to embedded systems, outlining both hardware interfaces like SWD and JTAG, as well as firmware-based approaches. The Open-CMSIS-Pack standard was presented as a solution for software component packaging and project configuration in the microcontroller space.

Finally, we discussed methods for automating project generation, including the use of regular expressions, file templates, and specialized tools like the CMSIS-Toolbox generators. These concepts and tools form the theoretical and practical basis for the work conducted during this internship, setting the stage for the implementation and results that will be presented in subsequent chapters.

CHAPTER 3

OBJECTIVES SPECIFICATION AND WORK ENVIRONMENT

Contents

Introduction

This chapter will first define the project's functional and non-functional requirements, then we will discuss the chosen structural decisions, explaining the reasoning behind their selection. Finally we will specify the hardware and software resources necessary for this project.

3.1 Project Specification

Requirements analysis is a fundamental phase in every project realization process. It is based on the study of the project's features as well as the constraints. In this section, we will cover both the functional and non-functional requirements.

3.1.1 Functional Requirements

The goal of the project is to generate a running CoreMark project on STM32 devices using CMSIS-Toolbox in an automated way. Within this context, this means guaranteeing the following concepts:

Table 3.1: Functional Requirements for CMSIS-Toolbox based automated benchmarking

Requirements	Description
Automation	The system must automatically generate a CoreMark benchmarking project for STM32 devices without requiring manual configuration.
Device Adaptability	The generated project must adapt to different STM32 families and configurations.
CoreMark Integration	The CoreMark benchmark must be integrated and ready to run immediately on the target hardware.
Result Reporting	The system must provide a standardized mechanism for reporting benchmark results.
Multi-Toolchain Support	The project must support multiple compilers such as GCC, ARM Compiler, and IAR.

3.1.2 Non-Functional Requirements

Below are the non-functional requirements or the constraints that the functional expectations must abide by:

- **Performance Requirements:**
  - **Low Overhead** The project must not add significant, if any overhead besides the CoreMark runtime, this insures the integrity of the benchmarking results.
  - **Compiler Optimizations** The project must provide the most optimal compiler configuration to ensure the maxmium performance results.

- **Reliability Requirements:**
  - **Correctness** The project must be functional and able to run without manual fixes.
  - **Consistency** The project must produce consistent results amongst STM32 devices.
  - **Error Handling** The project must fail gracefully and generate meaningful error messages.
- **Usability Requirements:**
  - **Ease of use** The generation process must be simple and intuitive, the project structure must be clear and consistent.
  - **Standardized Output** The results must be presented in a clear format to facilitate further automation and data collection.

3.2 Structural Decisions

Before delving further into the implementation details, it is crucial to understand the rationale behind the structural decisions chosen, for both the generation tool and the CoreMark project. This sections elucidates the reasons behind the choices and their relevence to the project's objectives.

3.2.1 CoreMark Project

The CoreMark project is the end result of the generation tool, it is meant to be runnable out of the box with the 3 main compilers (GCC, ARM, and IAR).

Ensuring the cross-toolchain support is done by providing the necessary compiler specific directives:

- **Linker Scripts** Each compiler has their own unique format for linker scripts:
  - **GCC:** GCC uses the .ld file extension and the LDScript syntax for the linking phase, it is structued in a declarative way to generate regions, sections and describe behavior within elements as well as exporting symbols.
  - **IAR:** IAR uses the .icf file extension as well as proprietary syntax for the linking phase, it allows the user to simply declare regions and optionally sections and takes care of the rest.
  - **ARM:** ARM compiler uses the .sct file extension, you only have to declare memory regions and the rest is handled either in C code or by the compiler directly.
- **Startup Files** The startup file is written in assembly (this approach is changed in HAL2), and although most of the code is common between the 3 compilers, some specific instructions/symbols are different, as well as the libc initialization function is different as each compiler uses a different implementation.

- **Compiler Options** Each compiler has their own set of compiler flags to be used, providing each one with the appropriate configuration is crucial to ensure consistent running.
- **Generic Print Function Implementation** As mentioned prior, each compiler uses a different implementation of libc, meaning that the syscall behind the standard library's printf function are different, providing our own implementation and passing it to CoreMark for result reporting is the most efficient and platform-agnostic solution.

The table 3.2 provides an overview of the architecture of the CoreMark project.

Table 3.2: CoreMark Project Structure

Element	Description
User Code	This component provides the definition for the user application's entry point as well as the UART peripheral configuration.
Driver	This component includes the necessary HAL drivers to enable the UART peripheral.
Toolchain_Specific	This component contains the toolchain specific source and header files to modify if the user wants to modify system configurations such as linker scripts and startup files.
CoreMark	This component contains the source code of the CoreMark benchmark to be invoked within the usercode's entry point.
coremark.csolution.yml	This is the workspace file for CMSIS-Toolbox to have knowledge of the project, it defines the DFP, ARM's CORE component, alongside the target device, build types and compiler selection.
coremark.cproject.yml	This is the project description file, it contains the source files to be compiled, user defines for UART functionality and the include paths.
cdefault.yml	This is the file that contains all the compiler options, in our context, the configuration is meant to provide the most performance-oriented setup.

3.2.2 Project Generation Tool

The architecture of the CoreMark project is quite complex and contains multiple pieces that must be either generated dynamically or fetched from different resources.

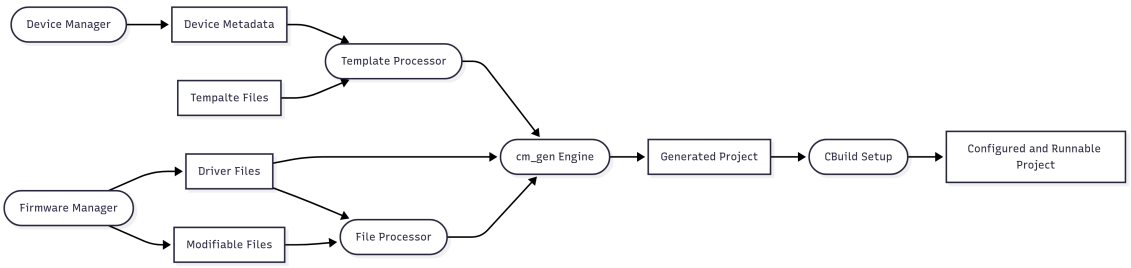

Figure 3.1: Project Generation Tool Architecture

The figure 3.1 shows the multiple modules within the project generation tool.

Table 3.3: Project Generation Tool Structure

Module	Description
Device Manager	Extracts the device metadata
Firmware Manager	Manages the device's firmware pack and extracting the necessary files
Template Processor	Generates files from the template files and the gathered data
File Processor	Modifies firmware files that cannot be templated to accomodate for our project's context
cm_gen Engine	Orchestrates and runs the different modules

3.3 Use-Case Diagram

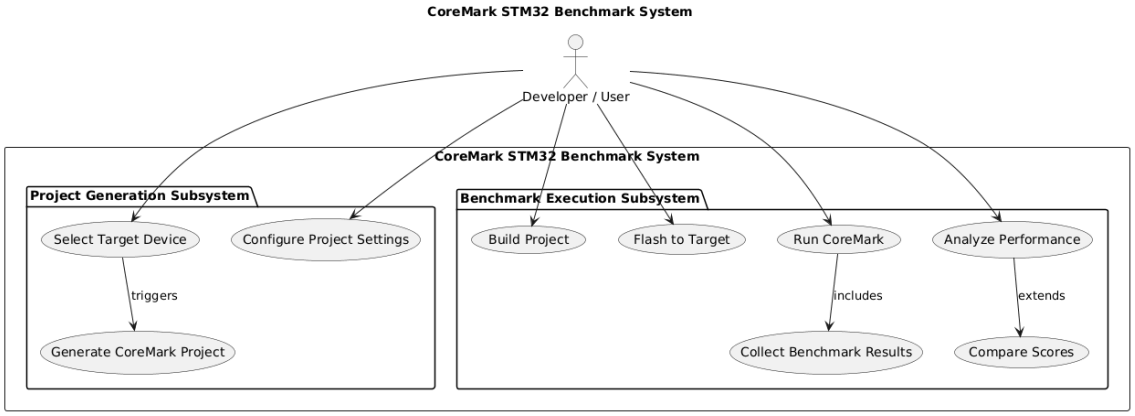

Figure 3.2: Automated CoreMark Benchmarking Use-Case Diagram

The Use-Case diagram in Figure 3.2 illustrates the interactions between a user, the project generation tool and the project's execution space. Refer to Table 3.4 for an in-depth explanation:

Table 3.4: Automated CoreMark Benchmarking

Use Case	Relevance
	Project Generation Subsystem
Select Target Device	The developer selects the STM32 device or family for which the CoreMark project should be generated. This ensures that the automation tool generates the correct configuration files and CMSIS device support.
Configure Project Settings	The developer configures CoreMark-specific parameters such as iteration counts, seed values, and compiler optimization flags before building and running the benchmark.
Generate CoreMark Project	The automation tool generates a complete project structure, including the `csolution` files and necessary source code, ready to be built and deployed for the selected STM32 target.
	CoreMark Project Subsystem
Build Project	The CoreMark project is compiled using the CMSIS build system and toolchain to produce a firmware image for the target hardware.
Flash Target	The compiled binary is flashed to the STM32 device to prepare it for execution.
Run CoreMark	The benchmark executes on the STM32 target, performing operations such as list processing, matrix manipulation, and state machine handling to measure CPU performance.
Collect CoreMark Results	During execution, CoreMark outputs the cycle count, execution time, and final score.
Analyze Performance	The developer interprets the CoreMark/MHz score and other performance metrics to determine how well the device performs within the specified context.
Compare Scores	Performance results are compared across different devices, toolchains, or configurations to guide hardware selection or optimization decisions.

Work Environment

In this section, we outline the hardware and software resources utilized during the internship project that were essential for the development testing, and validation.

3.4 Hardware Resources

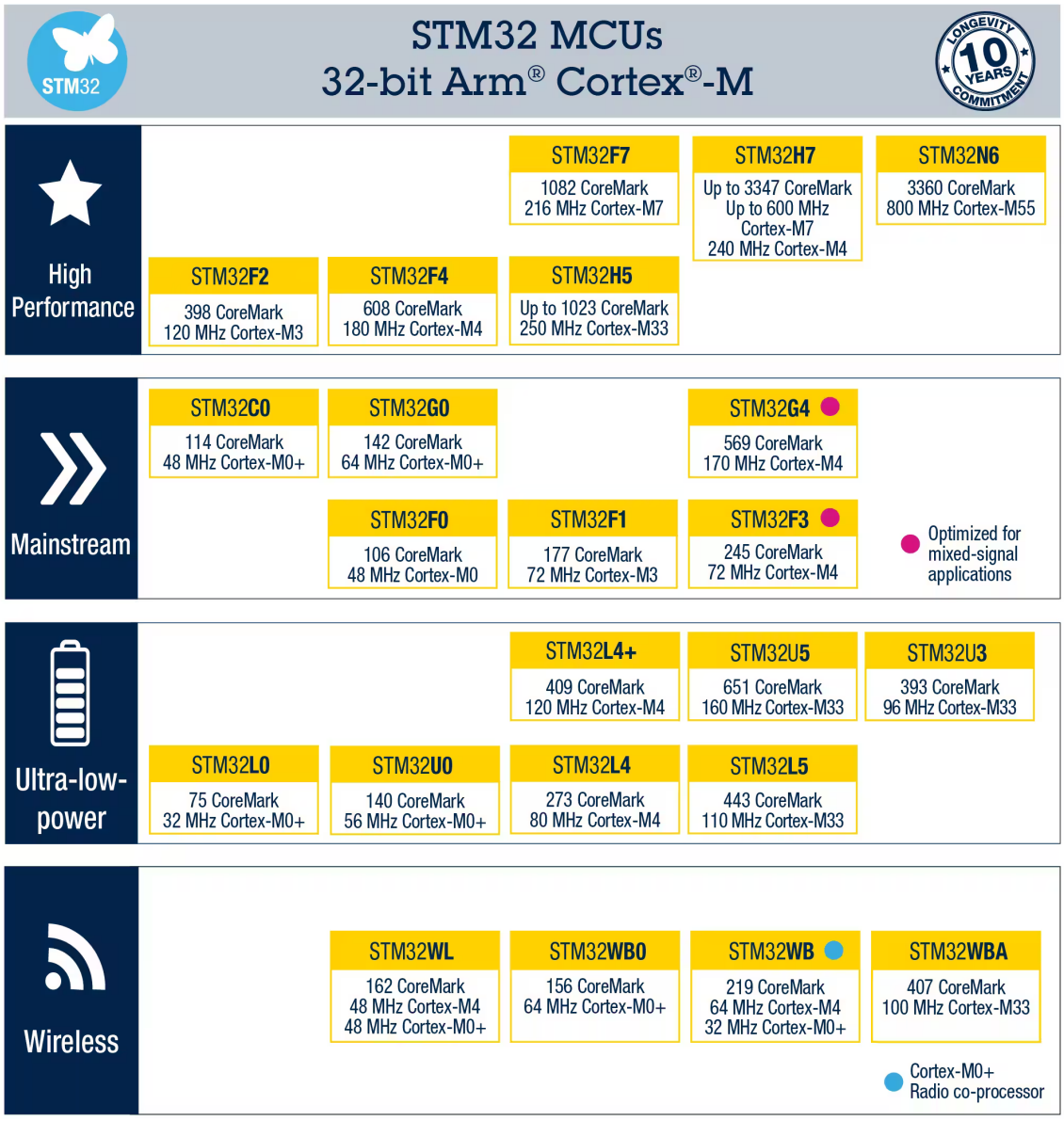

Figure 3.3: STM32 Microcontroller Portfolio

The project is meant to work on all STM32 MCUs, therefore as a base unit, the validation was done on a single board from all STM32 MCU families. Table 3.5 describes all the available families:

Table 3.5: STM32 Microcontroller Families

Family	Core	Maximum Frequency(MHz)	Read-Only Memory	Random Access Memory	Additional Features
STM32C0	Arm Cortex-M0+	48MHZ	16-256KB	6-36KB	-
STM32F0	Arm Cortex-M0	48MHz	16-256KB	4-32KB	-
STM32F1	Arm Cortex-M3	72MHz	16KB-1MB	4-96KB	-
STM32F2	Arm Cortex-M3	120MHz	128KB-1MB	64KB-128KB	ART Accelerator
STM32F3	Arm Cortex-M4	72MHz	32-512KB	16-80KB	CCM-SRAM and FPU/DSP instructions
STM32F4	Arm Cortex-M4	180MHz	64KB-2MB	32-384KB	Chrom-ART Accelerator
STM32F7	Arm Cortex-M7	216MHz	64K-2MB	256-512KB	L1 Data and Instruction Caches, TCM, Read-While-write Flash capabilities
STM32G0	Arm Cortex-M0+	64MHz	16-512KB	8-144KB	
STM32G4	Arm Cortex-M4	170MHz	32-512KB	32-128KB	CCM-SRAM and FPU/DSP instructions
STM32H5	Arm Cortex-M33	250MHz	128KB-2MB	32-640KB	Integrated Instruction and Data Cache peripherals

Continued on next page

Table 3.5 – Continued from previous page

Family	Core	Maximum Frequency	Read-Only Memory	Random Access Memory	Additional Features
STM32H7	Arm Cortex-M7	600MHz	64KB-2MB	536KB-1.4MB	L1 Data and Instruction Cache, TCM, Chrome-ART and NeoChrom Accelerators.
STM32L0	Arm Cortex-M0+	32MHz	8-192KB	2-20KB	Up to 6KB of EEPROM
STM32L1	Arm Cortex-M3	32MHz	32-512KB	4-80KB	Up to 16KB of EEPROM
STM32L4	Arm Cortex-M4	80MHz	64KB-1MB	40-320KB	Chrom-ART Accelerator
STM32L4+	Arm Cortex-M4	120MHz	512KB-2MB	320-640KB	Chrom-ART Accelerator
STM32L5	Arm Cortex-M33	110MHz	256-512MB	256KB	ART Accelerator
STM32N6	Arm Cortex-M55	800MHz	0	4.2MB	Neural-ART Accelerator(Up to 1GHz Clock Speed)
STM32U0	Arm Cortex-M0+	56MHz	16-256KB	12-40KB	-
STM32U3	Arm Cortex-M33	96MHz	512KB-1MB	256KB	Integrated Instruction and Data Cache peripherals

Continued on next page

Table 3.5 – Continued from previous page

Family	Core	Maximum Frequency	Read-Only Memory	Random Access Memory	Additional Features
STM32U5	Arm Cortex-M33	160MHz	128-4MB	274KB-3MB	Integrated Instruction and Data Cache peripherals
STM32WB	Arm Cortex-M4	64MHz	256KB-1MB	48-256KB	-
STM32WB0	Arm Cortex-M0+	64MHz	192-512KB	24-64KB	-
STM32WBA	Arm Cortex-M33	100MHz	512KB-2MB	64-512KB	Integrated Instruction and Data Cache peripherals
STM32WL	Arm Cortex-M4	64MHz	64-256KB	8-64KB	-

3.5 Software Resources

The following software tools were used for the development, debugging and testing of the project:

3.5.1 Programming Languages

- **C:** C is a general-purpose programming language. By design, it gives the programmer relatively direct access to the features of the typical CPU architecture, customized for the target instruction set. It has been and continues to be used to implement operating systems (especially kernels), device drivers, protocol stacks, and embedded applications. C is used on computers that range from the largest supercomputers to the smallest microcontrollers.

- **Go:** Go (or Golang) is a high-level general purpose programming language that is statically typed and compiled. It is known for the simplicity of its syntax and the efficiency of development that it enables by the inclusion of a large standard library supplying many needs for common projects.

- **Python:** Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically type-checked and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming.

3.5.2 Integrated Development Environments and Text Editors

3.5.2.1 Visual Studio Code

Visual Studio Code (VS Code) is an extensible code editor developed by Microsoft for Windows, Linux, macOS and web browsers. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded version control with Git. Users can install extensions that add functionality. Table 3.6 goes over the most impactful extensions during the development of the project.

Table 3.6: Relevant VSCode Extensions

Extension	Description
Arm CMSIS Solution	The Arm CMSIS Solution extension is a graphical user interface for csolution projects that use the CMSIS-Toolbox. The extension supports microcontroller devices that incorporate Arm Cortex®-M processors and works with various C/C++ compilers and debuggers.
Arm CMSIS Debugger	The Arm CMSIS Debugger extension pack is a comprehensive debug platform for Arm Cortex-M processor-based devices that uses the GDB/MI protocol. It supports single and multi-core processor systems, it includes pyOCD for target connection and image download, as well as GDB for core debugging features.
Serial Monitor	The Serial Monitor extension provides a serial monitor to view output from as well as send messages to serial ports. This is often useful when testing or debugging programs on embedded devices.
ELFInsight	ElfInsight is an extension to analyze ARM Cortex-M ELF files, it offers a streamlined interface to view symbol tables, inspect memory usage and visualize function call graph.

3.5.3 Compiler Toolchains

3.5.3.1 GCC

The GCC compiler is an open-source C/C++ compiler, and is part of the GNU utilities, ARM provides their own implementation, prefixed with "arm-none-eabi" and can target most ARM architectures such as ARMv7 and ARMv8.

3.5.3.2 ARM Compiler v6

As well as providing a cross-compiling gcc toolchain, ARM has their own compiler based on CLANG and LLVM technologies called ARM compiler.

3.5.3.3 IAR Toolchain

IAR Systems is a Swedish computer software company that offers development tools for embedded systems. They are known for having a comprehensive development suite and an optimized compiler based on GCC.

3.5.4 Development Tools

3.5.4.1 CMake

CMake is a free, open-source, cross-platform, software development tool for building applications via compiler-independent instructions. It also can automate testing, packaging and installation. It runs on a variety of platforms and supports many programming languages. As a meta-build tool, CMake configures native build tools which in turn build the codebase. CMake generates configuration files for other build tools based on CMake-specific configuration files. The other tools are responsible for more directly building; using the generated files. A single set of CMake-specific configuration files can be used to build a codebase using the native build tools of multiple platforms.

3.5.4.2 Ninja

Ninja is a small build system with a focus on speed. It differs from other build systems in two major respects: it is designed to have its input files generated by a higher-level build system, and it is designed to run builds as fast as possible.

3.5.4.3 CMSIS-Toolbox

The CMSIS-Toolbox provides command-line tools for project creation and build of embedded applications utilizing CMSIS-Packs. It supports multiple compilation tools. It also helps with software pack creation, maintenance, and distribution utilizing the CMSIS-Pack format.

Overall Workflow

The CMSIS-Toolbox includes the following tools for the creation of embedded applications:

- **Pack Manager (cpackget):** install and manage software packs in the host development environment.
- **Project Manager (csolution):** create build information for embedded applications that consist of one or more related projects. It performs the following operations:

- **In the Project Area:** Generate build information files *.cbuild-idx.yml and *.cbuild.yml with all relevant project information for the build process.
- **In the RTE Directory:**
  - Generate for each context the RTE_components.h file and pre-include files from the software pack (*.pdsc) metadata.
  - Copy the configuration files from selected software componentsand provide PLM information.

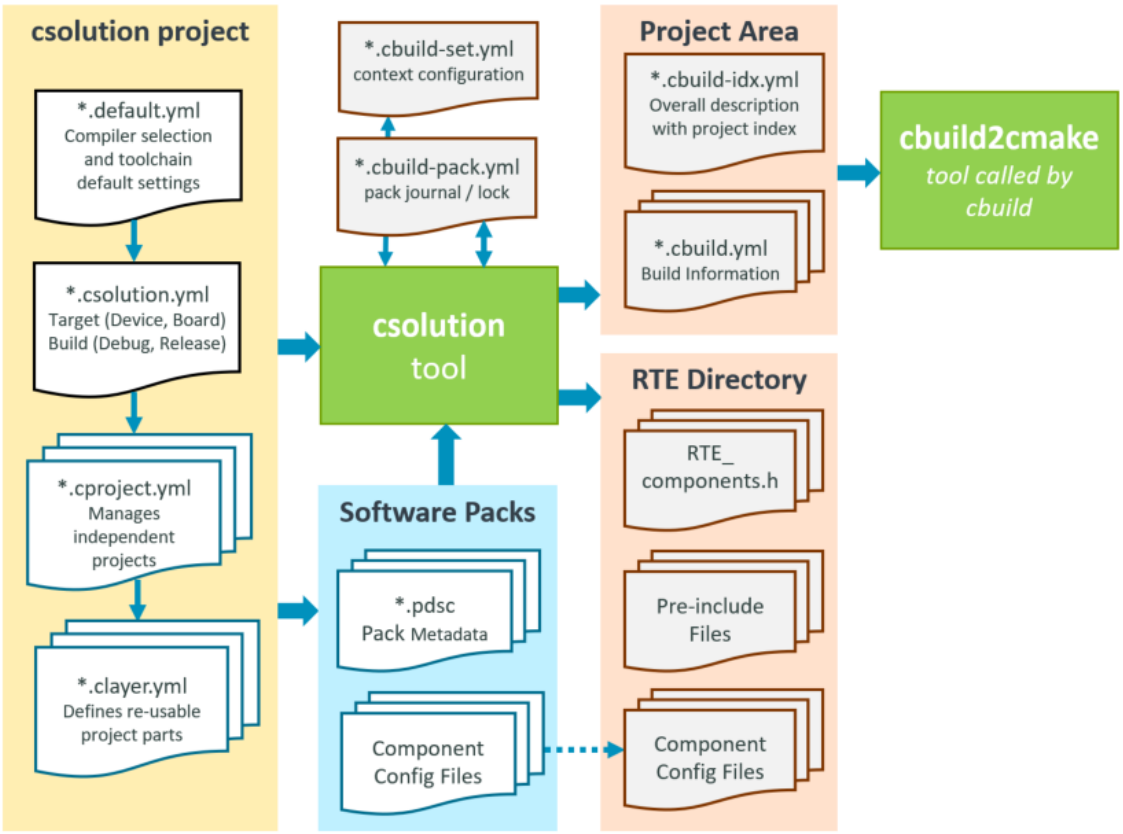

Figure 3.4: Csolution Operation

- **Build Invocation (cbuild):** orchestrate the build steps utilizing CMSIS tools and a CMake compilation process. It has two overall modes :
  - **build mode:** generates the application and is the default command.
    - When option –packs is used, it downloads missing software packs using cpackget.
    - It calls csolution to process the the <name>.csolution.yml project. The output are build information files with all relevant project information for the build process.
  - **setup mode:** generates the setup information for an IDE to populate dialogues, IntelliSense, and project outline views.
    - Check YML file syntax against schema for all files specified by *.csolution.yml.

∗ Check the correctness of all files specified by *.csolution.yml.

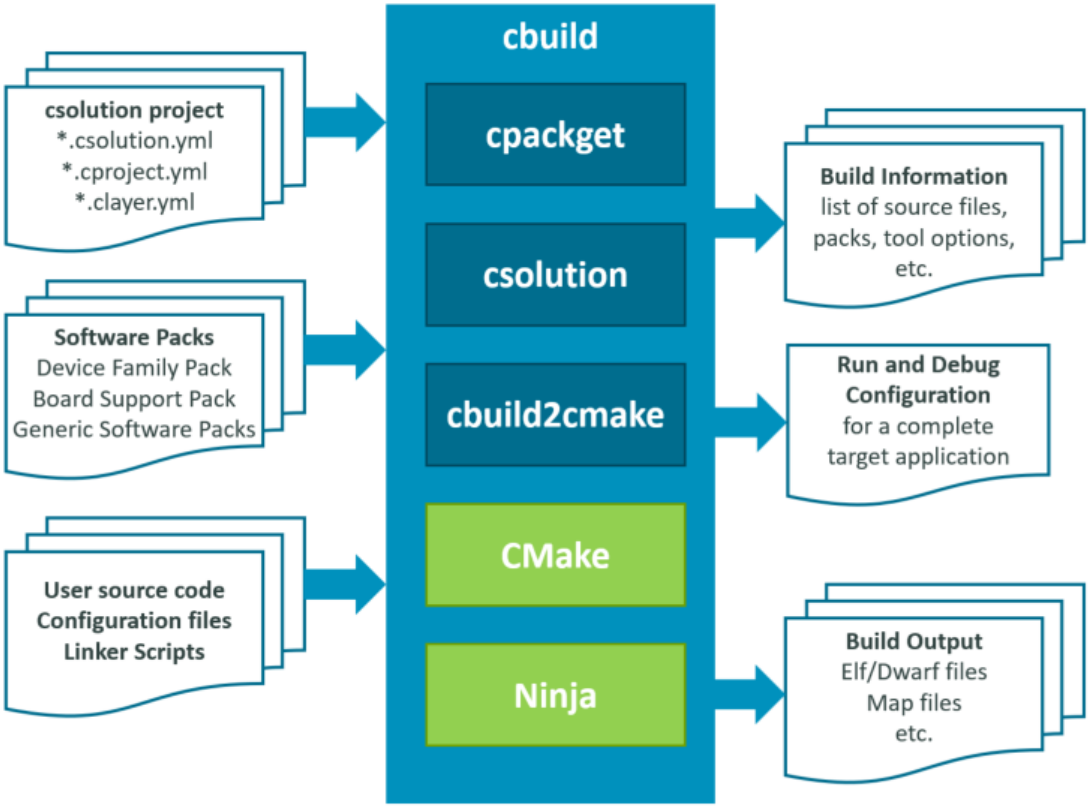

Figure 3.5: Cbuild Workflow

3.5.4.4 PyOCD

pyOCD is an open source Python based tool and package for programming and debugging Arm Cortex-M microcontrollers with a wide range of debug probes. It is fully cross-platform, with support for Linux, macOS, Windows, and FreeBSD. Beyond its standalone capabilities, pyOCD has been integrated into the CMSIS-Toolbox ecosystem as a backend for device flashing and debugging. As a result, it simplifies the automation of programming and debugging tasks across different hardware targets and development environments.

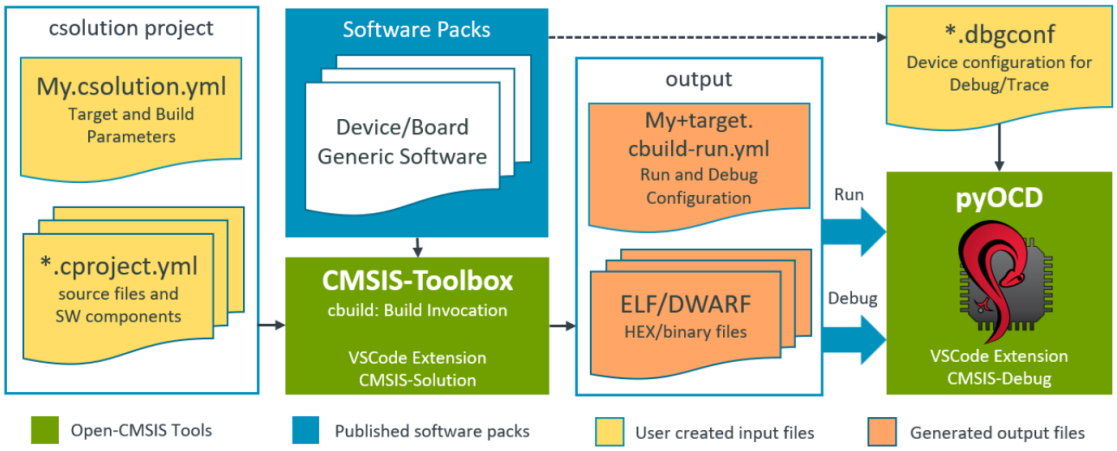

Figure 3.6: PyOCD CMSIS-Toolbox-Integrated Running and Debugging Workflow

The CMSIS-Toolbox uses the information from the DFP and BSP to simplify the debugger configuration. It generates the file *.cbuild-run.yml that contains for one target of a csolution project all information for run and debug. The software packs contain information that is the basis for debug and run settings:

- Flash algorithms of device memory (in DFP) and board memory (in BSP).
- On-board debug adapter (a default programming/debug channel) including features.
- Available memory of device and board.
- Device parameters such as processor core(s) and clock speed.
- Debug Access Sequences and System Description Files.
- Debug Configuration files (*.dbgconf) that configure device properties such as trace pins.
- CMSIS-SVD SVD files for viewing device peripherals.
- CMSIS-View SCVD files for analysis of software components (RTOS, Middleware).

3.5.4.5 STM32CubeProgrammer

STM32CubeProgrammer is a proprietary, all-in-one programming tool provided by STMicroelectronics that facilitates the flashing and configuration of STM32 microcontrollers. Supporting a wide array of connectivity options, it offers a unified interface for erasing, programming, and verifying device memory across the entire STM32 portfolio. The tool provides advanced features like secure programming, option byte management, and lifecycle management for TrustZone-enabled devices. Its support for scripting and CLI operation further enables automation in production and development environments.

3.5.4.6 STM32CubeMX

STM32CubeMX is a graphical software configuration tool that simplifies the development of STM32-based applications. It allows developers to:

- Configure microcontroller peripherals and middleware components through an intuitive graphical interface.
- Generate initialization code for STM32 microcontrollers, reducing development time.
- Integrate with various IDEs, including IAR Embedded Workbench, for a streamlined development workflow.

Conclusion

The work environment for this internship includes both hardware and software resources that are essential for the successful implementation of the project.

Abstract

This report describes the work carried out during my summer internship at STMicroelectronics, focused on platform-agnostic benchmarking. The main objective was to run Coremark on STM32 devices using Csolution project structure. To achieve this, I put to use my knowledge in software and embedded development,C code compilation and code generation, using tools such as CMSIS-Toolbox, C and Golang. The results helped to drastically reduce the time required to setup and run Coremark projects using multiple compilers, as well as generating the most optimal results. This internship also allowed me to strengthen my skills in the compilation process, embedded projects architecture design and gain insight into the specific challenges of embedded systems engineering.

Keywords : C/C++, Golang, Embedded Systems, STM32, Compilation, Code Generation.

Résumé

Ce rapport décrit les travaux réalisés lors de mon stage d'été chez STMicroelectronics, axé sur l'analyse comparative indépendante de la plateforme. L'objectif principal était d'exécuter Coremark sur des microcontrôleurs STM32 en utilisant la structure de projet Csolution. Pour ce faire, j'ai mis à profit mes connaissances en développement logiciel et embarqué, en compilation et génération de code C, à l'aide d'outils tels que CMSIS-Toolbox, C et Golang. Les résultats ont permis de réduire considérablement le temps nécessaire à la configuration et à l'exécution des projets Coremark avec plusieurs compilateurs, tout en optimisant les résultats. Ce stage m'a également permis de renforcer mes compétences en compilation, en conception d'architecture de projets embarqués et de mieux comprendre les défis spécifiques de l'ingénierie des systèmes embarqués.

Mots clés : C/C++, Golang, Systèmes embarqués, STM32, Compilation, Génération de Code.