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

- **API** = **A**pplication **P**rogramming **I**nterface
- **ARM** = **A**dvanced **R**ISC **M**achine
- **BSP** = **B**oard **S**upport **P**pack
- **CMSIS** = **C**ortex **M**icrocontroller **S**oftware **I**nterface **S**tandard
- **DFP** = **D**evice **F**amily **P**ack
- **DWARF** = **D**ebugging **W**ith **A**ttributed **R**ecord **F**ormats
- **ELF** = **E**xecutable and **L**inkable **F**ormat
- **FPU** = **F**loating **P**oint **U**nit
- **GCC** = **G**NU **C**ompiler **C**ollection
- **GNU** = **G**NU's **N**ot **U**nix
- **HAL** = **H**ardware **A**bstraction **L**ayer
- **IAR** = **I**ngenjörsfirma **A**nders **R**undgren
- **IDE** = **I**ntegrated **D**evelopement **E**nvironment
- **IP** = **I**ntegrated **P**eripheral
- **IV** = **I**nitialization **V**ector
- **JSON** = **J**ava**S**cript **O**bject **N**otation
- **LL** = **L**ow-**L**ayer
- **LLVM** = **L**ow **L**evel **V**irtual **M**achine
- **MCU** = **M**icro **C**ontroller **U**nit
- **PLM** = **P**roject **L**ifetime **M**anagement
- **RTE** = **R**un**T**ime **E**nvironment
- **RTOS** = **R**eal-**T**ime **O**perating **S**ystem
- **RegEx** = **Reg**ular **Ex**pression

- **SCVD** = **S**oftware **C**omponent **V**iewer **D**escription
- **SVD** = **S**ystem **V**iew **D**escription
- **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.

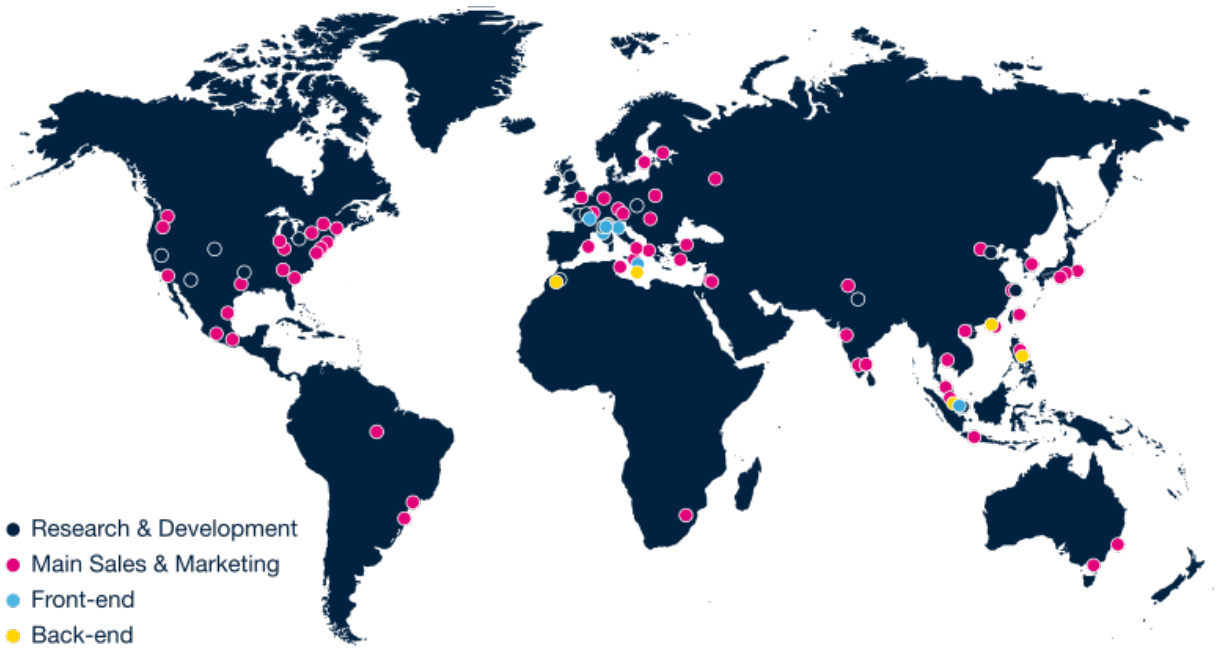

Figure 1.2: STMicroelectronics Worldwide Presence

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:

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 developing a practical demonstration of the "CryptoEngine," the new cryptography peripheral featured in the STM32XX MCU series. This demo is designed to help users, especially those unfamiliar with cryptography, to understand and utilize the advanced features of this new technology.

To achieve this, the demonstration sets up a secure communication system between two boards: one equipped with the STM32XX MCU using the "CryptoEngine" and the other with an STM32U545 MCU employing existing cryptographic solutions. By highlighting the security enhancements, and ease of use provided by the "CryptoEngine," this demo will serve as a valuable resource for developers, offering a hands-on introduction to the new hardware's capabilities.

1.7 Objective

The primary objective of this project is to develop a practical demonstration that showcases the capabilities of the new "CryptoEngine" in enhancing security and simplifying cryptographic operations. This demonstration will serve as a reference for developers and engineers, providing them with a clear understanding of the new hardware's features and practical applications.

By achieving this objective, the project aims to establish a good starting guide to users of the new "CryptoEngine", which will contribute to the development of more secure and robust cryptographic solutions, thereby enhancing the overall security posture of IoT and connected devices.

1.8 Conclusion

To conclude, this chapter has introduced STMicroelectronics, including its global operations, the ST Tunis site, and the Support Solutions team. We have highlighted the growing importance of security for microcontrollers, especially within the IoT landscape, and identified the existing limitations in the current cryptographic solutions available for STM32 products.

The proposed "CryptoEngine" is designed to address these challenges by providing enhanced security features combined with improved usability. By developing a practical demonstration of this new peripheral, the project aims to offer a comprehensive introduction to its capabilities, thus enabling users to effectively implement and benefit from advanced cryptographic solutions.

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. 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** series and the **STM32H** series. Within each 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.

Together, these components create an environment where developers can choose the appropriate level of abstraction based on their specific requirements, all while maintaining compatibility and code re-usability across the STM32 ecosystem.

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).

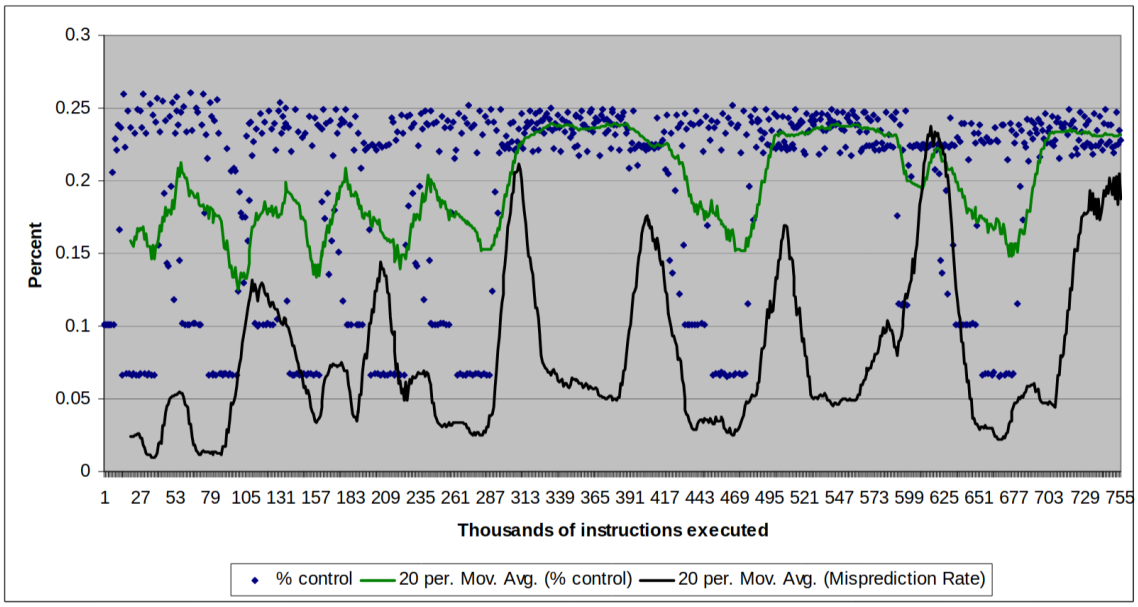

Figure 2.1: Distribution of control instructions and mispredictions over CoreMark execution.

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

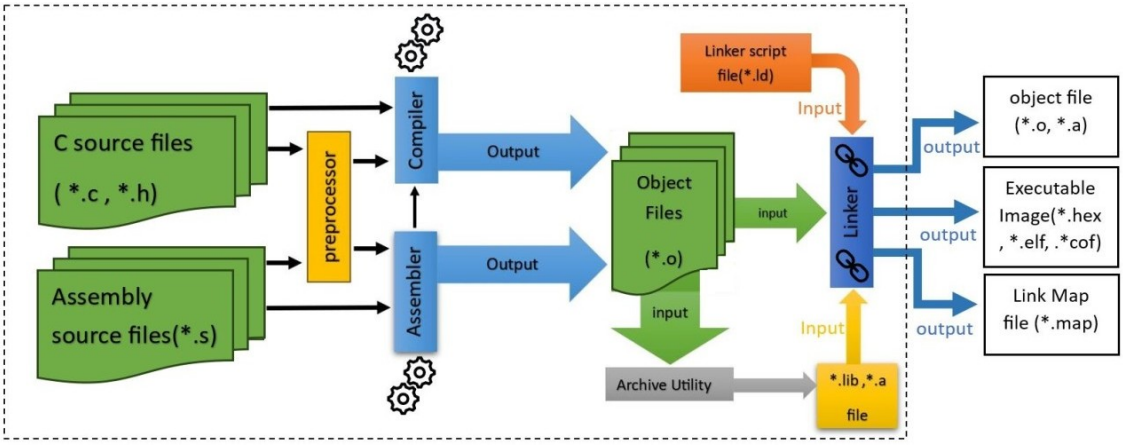

Figure 2.2: Compilation Process Diagram

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.
- **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 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

Building applications for embedded systems is quite different from a native build process, although you can create your own cross-compilation toolchain, there are available battle-tested available toolchains that are used across the industry. The most relevant ones being:

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

2.3.3 Build Runners

Whereas simple projects can be compiled directly by invoking the corresponding command via an IDE or a terminal, as the project grows more complex, maintaining dependencies and flags becomes hard; that's where build runners or build system generators come in. They apply a rule-based mechanism to produce the desired output whether it be a library or an executable from its input which is the project's source code. We find two main runners used:

- **Make:** Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. Make is based on a file named "Makefile" at the top level of the project, it contains the necessary informations describing the build process and how to compute the desired output files.
- **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.

2.3.4 CMake

Although build runners alone work in a small scope, they do not provide enough flexibility to work across different environments or have multiple targets. This is the problem that CMake solves; it is a cross-platform build-system generator, that allows your project to run in different contexts without having to write platform specific configuration. It supports dependency management, integrates seamlessly with multiple IDEs and toolchains and offers builtin test and packaging support.

2.4 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 or the data center. Open-CMSIS-Pack removes this complexity, delivering a standard for software component packaging and related foundation tools for validation, distribution, integration, management, and maintenance.

2.4.1 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.

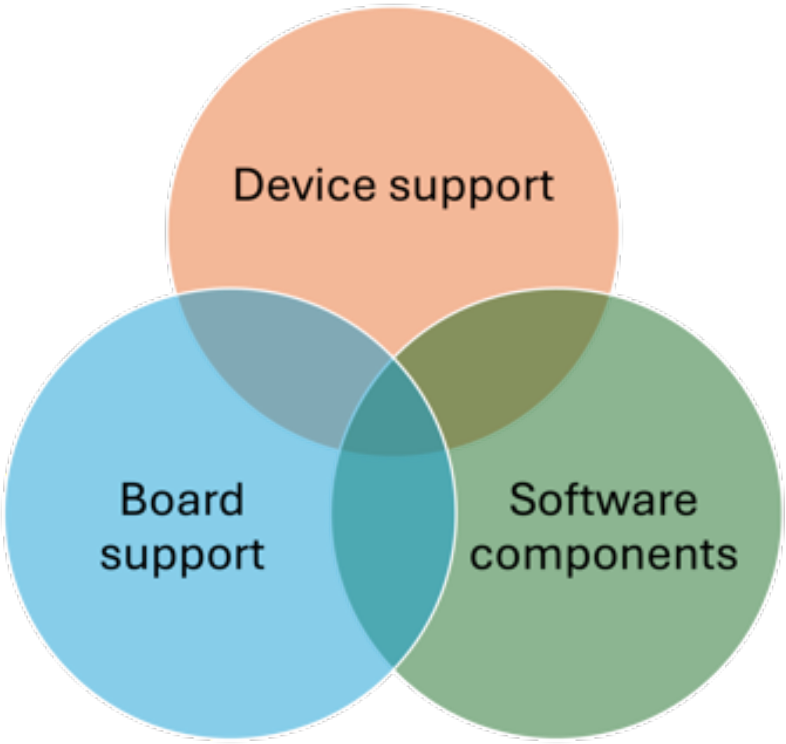

Figure 2.3: Software Packs Types

2.4.1.1 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.

2.4.1.2 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.

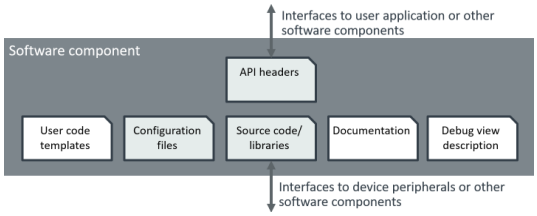

Figure 2.4: Software Component Interface

2.4.1.3 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.

2.4.2 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 you with software pack creation, maintenance, and distribution utilizing the CMSIS-Pack format.

2.4.2.1 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.

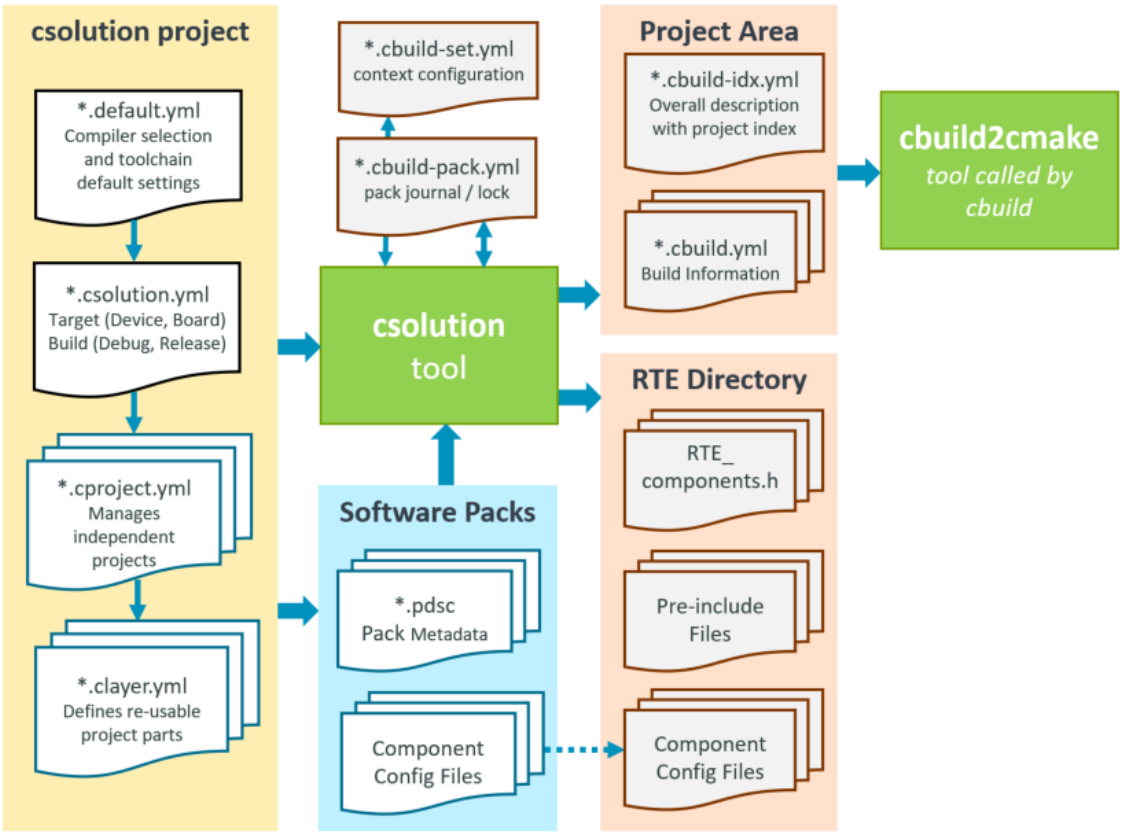

Figure 2.5: Csolution Operation

- **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.

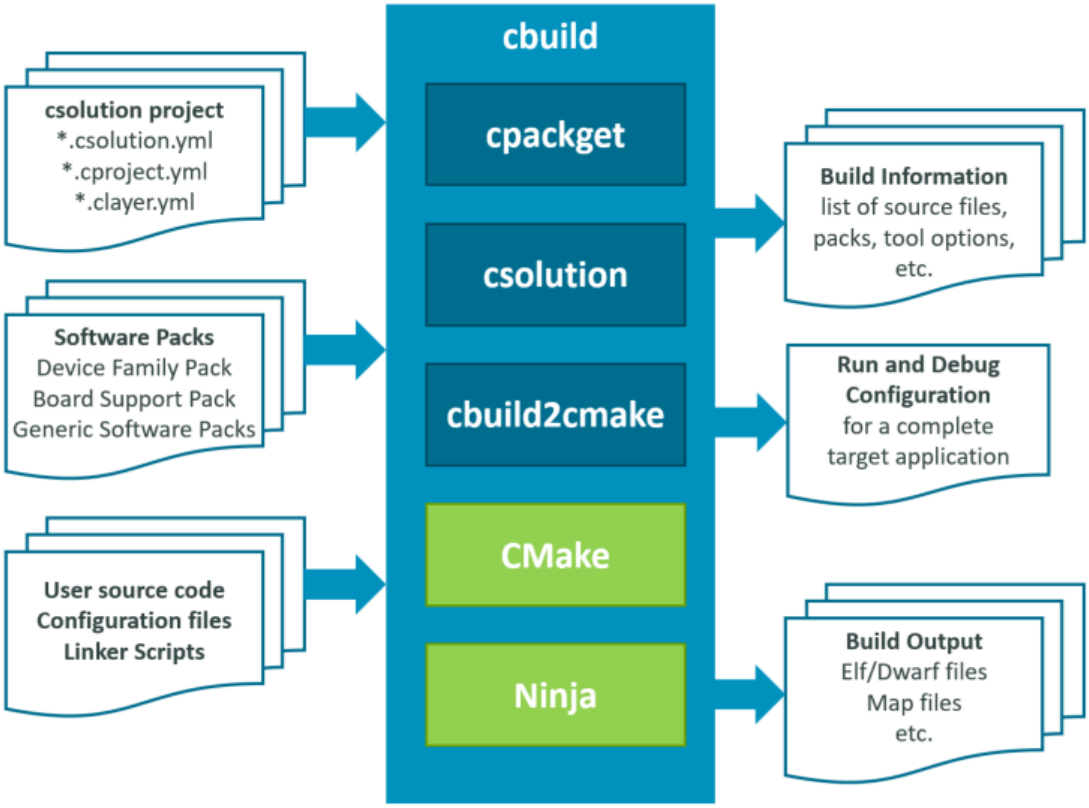

Figure 2.6: Cbuild Workflow

2.4.2.2 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.

2.4.2.3 Cbuild2cmake

cbuild2cmake reads the build information files *.cbuild-idx.yml and *.cbuild.yml to get all relevant project information for the build process. It generates the output files for CMake build system; describing the overall application build process with the current context configuration, as well as the toolchain specific configuration and source file definitions for groups and software components. CMake is then invoked twice.

- CMake configuration command defines the build generator, source, and build directory with:

```
cmake –G Ninja –S <tmpdir> –B <tmpdir> –Wnodev
```

- CMake build command to build the application program for each context with:

```
cmake ––build <tmpdir> –j <n> ––target <context>
```

2.4.2.4 Run and Debug Management

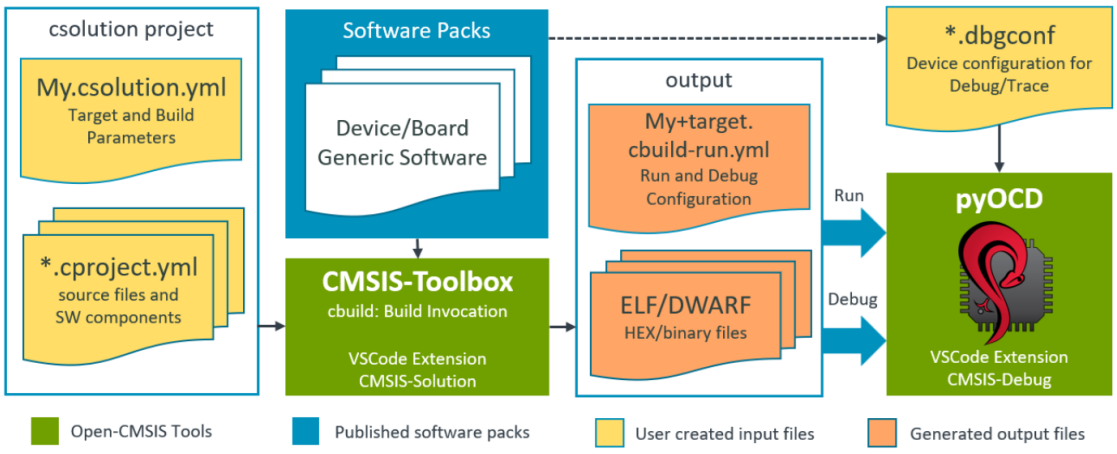

Figure 2.7: Running and Debugging Workflow

The CMSIS-Toolbox build system manages software packs that contain information about device, board, and software components. It controls the build output (typically ELF/DWARF files), and has provisions for HEX, BIN and post-processing.

Configuration & Settings

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 that support more complex Cortex-A/R/M configurations.
- 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).

2.5 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 since every single step can be automated. There are multiple techniques to be used to generate a fully working project from scratch.

2.5.1 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.

2.5.1.1 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.

2.5.1.2 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.

2.5.2 Templates & File Manipulation

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

2.5.2.1 Template Definition

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

2.5.2.2 Parameterization

The template defines parameters that can be filled in by the user or a program when generating a new file. These parameters can include names, dates, specific values, or other relevant information.

2.5.2.3 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.

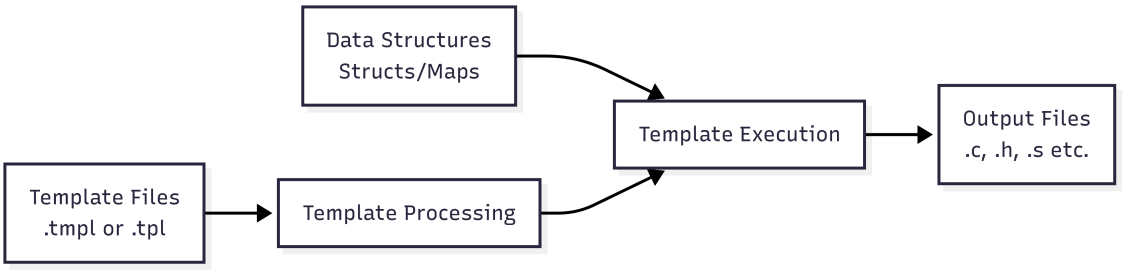

Figure 2.8: Templating Process

2.5.3 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.

2.5.3.1 Generator Integration (STM32CubeMX Example)

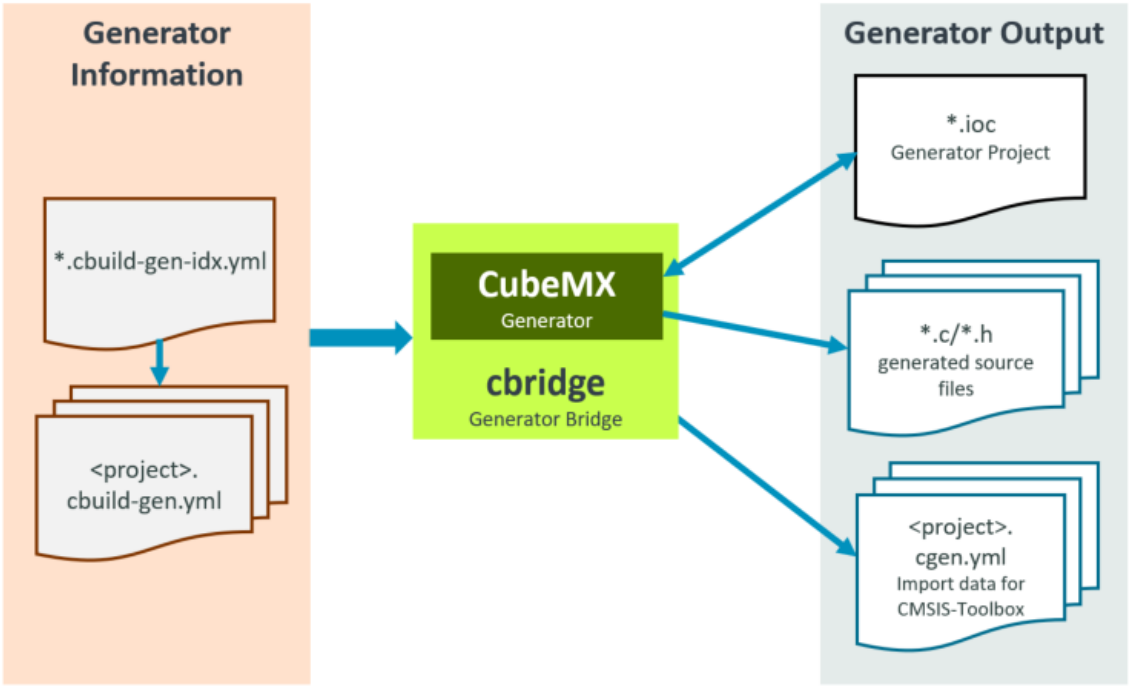

Figure 2.9: Generator Integration in CMSIS-Toolbox

The Figure 2.9 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 provided the foundational knowledge required to understand the subsequent work in this report. We began by exploring the STM32 ecosystem, examining the wide range of available microcontrollers, their ARM Cortex-M cores, and critical peripherals such as UART. We also reviewed the STM32Cube firmware package, which streamlines development through standardized libraries like CMSIS, HAL, and LL drivers.

Next, we covered benchmarking principles with a focus on embedded systems. We highlighted the unique challenges of benchmarking microcontrollers, including clock source provisioning, memory mapping, and custom I/O implementations. CoreMark was introduced as a reliable and widely adopted benchmark for evaluating MCU performance in a standardized manner, providing meaningful metrics for comparison.

Afterwards, we discussed the embedded systems build process, emphasizing the importance of toolchains, linking, and memory management. We examined build systems like Make, Ninja, and CMake as a higher-level generator for complex, cross-platform builds.

A key part of this chapter was the introduction of Open-CMSIS-Pack and the CMSIS-Toolbox, which address the complexity of managing device- and board-specific software components. We detailed their role in packaging, versioning, and automating builds.

Finally, we explored project generation techniques, from basic methods like regular expressions and templating to advanced integration with generators like STM32CubeMX.

In summary, this chapter established a comprehensive theoretical framework, connecting concepts of hardware, benchmarking, toolchains, and automation. These concepts form the backbone of the methodology applied in later chapters, ensuring that the processes for benchmarking and project generation are both systematic and scalable.

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 cryptographic algorithms, explaining the reasoning behind their selection and offering an overview of how each algorithm works. Finally we will specify the hardware and software resources necessary for this demonstration.

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 "CryptoEngine" demo is to establish secure communication between an STM32XX MCU using "CryptoEngine" and an STM32U5 MCU using PKA and AES. Secure communication means guaranteeing the three key security concepts: confidentiality, integrity, and authenticity.

- **Confidentiality:** Ensuring that only authorized parties can understand the messages exchanged. This is achieved through:
  - **Asymmetric Key Management:** Implement asymmetric key management for ECDSA and ECDH keys, ensuring secure storage and handling of private keys and generation of public keys.
  - **Shared Secret Computation:** Compute the ECDH shared secret using the exchanged public keys, which will be used as the basis for deriving symmetric encryption keys.
- **Integrity:** Ensuring that the message has not been altered during transmission. This is achieved through:
  - **Symmetric Message Encryption and Decryption:** Use AES GCM symmetric encryption to encrypt messages and verify their integrity during transmission. This also serves as an authentication check.
- **Authenticity:** Ensuring that the communicating parties are who they claim to be. This is achieved through:
  - **Signature Generation and Verification:** Enable the generation and verification of digital signatures using ECDSA to ensure the authenticity of the communicating parties.

Establishing a secure, authenticated encryption system using modern cryptographic standards, including Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures, Elliptic Curve Diffie-Hellman (ECDH) for key exchange, and Advanced Encryption Standard Galois Counter Mode (AES GCM) for encryption and authentication, is the primary goal.

Table 3.1 neatly summarizes the functional requirements for our demonstration.

Table 3.1: Functional Requirements for Secure Communication

Security Requirement	**Component**	**Description**
Confidentiality	Key Management	Implement key management for ECDSA and ECDH keys, ensuring secure storage and handling of private keys and generation of public keys.
	Shared Secret Computation	Compute the ECDH shared secret using the exchanged public keys, which will be used as the basis for deriving symmetric encryption keys.
Integrity	Message Encryption and Decryption	Use AES GCM to encrypt messages and verify their integrity during transmission. This also serves as an additional authenticity verification.
Authenticity	Signature Generation and Verification	Enable the generation and verification of digital signatures using ECDSA to ensure the authenticity of the communicating parties.

3.1.2 Non-Functional Requirements

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

- **Clarity:** The demo should be clear and easily understandable, enabling users without extensive technical expertise to grasp the concepts.
- **Transferability:** The knowledge and experience gained from the demo should empower users to apply cryptographic techniques in their own projects with ease.
- **Modularity:** The demo should be designed in a modular way, allowing individual components to be easily reused or adapted for different applications.
- **Feature Comparison:** The demo should highlight the new features and enhancements introduced by the "CryptoEngine" compared to its predecessors, demonstrating the improvements in security and usability.

3.2 Cryptographic Decisions and Context

Before delving further into the implementation details, it is crucial to understand the rationale behind the cryptographic algorithms and protocols chosen for this demo. This section elucidates the reasons for selecting

specific cryptographic methods and their relevance to the project's objectives.

3.2.1 Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) is a widely adopted key exchange protocol that facilitates secure and efficient key exchange between parties. We opted for ECDH due to its significant advantages over traditional algorithms like RSA, particularly in resource-constrained environments.

- **Resource Efficiency:** ECDH is less resource-intensive compared to RSA, making it suitable for embedded systems with limited computational power and memory.
- **Optimal Key Size:** A 256-bit key size strikes a balance between security and performance, making it suitable for embedded applications, including our demo.

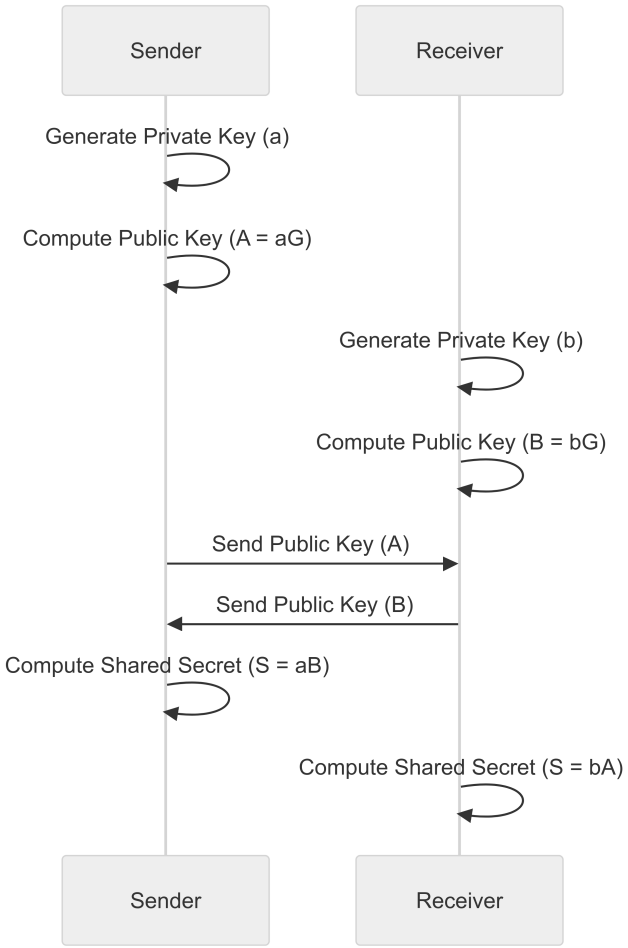

Figure 3.1: Elliptic Curve Diffie Hellman Process

The ECDH process as shown in Figure 3.1, begins with both the sender and the receiver independently generating their own private and public key pairs.

The sender generates a private key a and computes their corresponding public key $A = aG$, where G is a predefined generator point on the elliptic curve. Similarly, the receiver generates a private key b and computes

their corresponding public key $B = bG$.

Once the key pairs are generated, the sender and receiver exchange their public keys over the insecure channel. The sender sends their public key A to the receiver, and the receiver sends their public key B to the sender. Despite the exchange occurring over an insecure channel, the private keys a and b remain confidential and are never transmitted.

After receiving the public key from the other party, each participant computes the shared secret using their own private key and the received public key. The sender calculates the shared secret $S = aB$ by multiplying their private key a with the receiver's public key B. Conversely, the receiver calculates the shared secret $S = bA$ by multiplying their private key b with the sender's public key A. Due to the properties of elliptic curves, both computations result in the same shared secret S, which can be used to derive a symmetric key for encrypting and decrypting subsequent communications.

3.2.2 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a renowned digital signature algorithm that enables the generation and verification of digital signatures, ensuring data authenticity and integrity.

- **Efficiency:** ECDSA is highly efficient, offering faster computations and smaller key sizes compared to other digital signature algorithms like RSA.
- **Compatibility:** ECDSA is supported by the PKA (Public Key Accelerator) peripheral in STM32 products, making it an ideal choice for our demo.
- **Security:** ECDSA provides strong security guarantees, leveraging the hardness of the elliptic curve discrete logarithm problem.

The graph in Figure 3.2 illustrates a typical digital signature process that applies to many algorithms, including ECDSA. It involves two main phases: signing by the sender and verification by the receiver.

On the sender side, the process begins with key generation, where a pair of keys—a private key and a public key—are created. The sender then prepares a message that needs to be signed. This message is passed through a hashing algorithm to produce a message hash, ensuring that the message is represented in a fixed-size format. The private key and the message hash are then input into the signing algorithm to generate a signature. Subsequently, the message, signature, and public key are transmitted to the receiver.

On the receiver side, the verification algorithm uses the public key, message hash, and signature to verify the authenticity of the message. The receiver independently generates the message hash using the same hashing algorithm as the sender. If the signature is valid, the sender is authenticated, providing confidence that the message has not been tampered with and is from the legitimate sender. Conversely, if the signature is invalid, the

sender is not authenticated, indicating potential tampering or an illegitimate sender. This process ensures both the integrity and authenticity of the message, making digital signatures a necessity for secure communications.

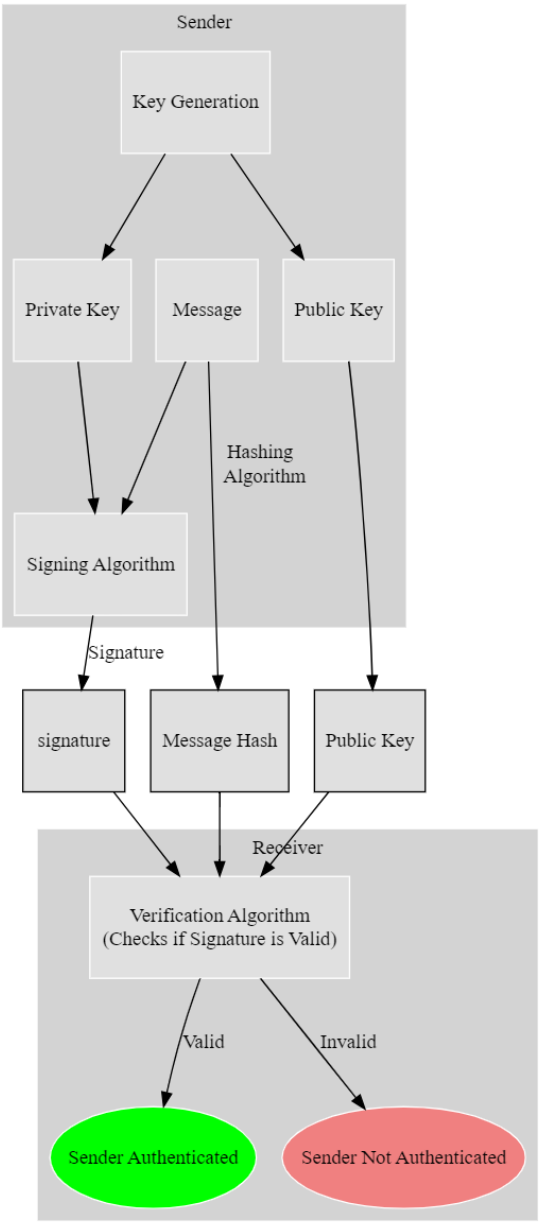

Figure 3.2: Typical Digital Signature Process

3.2.3 NIST P-256 Curve

For this demo, we have chosen the NIST P-256 curve for both the ECDSA algorithm and the ECDH protocol. This decision was based on several factors:

- **NIST Recommendation:** The P-256 curve is recommended by the National Institute of Standards and Technology (NIST) for its security and efficiency [2].
- **Popularity:** The P-256 curve is widely used and well-supported across various cryptographic libraries and tools, facilitating interoperability and ease of use [3].

- **Documentation:** Extensive documentation and implementation examples are available for the P-256 curve, simplifying the development and debugging processes.
- **Smaller Key Size:** ECDH achieves the same security level as RSA with much smaller key sizes. For instance, a 256-bit key in ECDH provides equivalent security to a 3072-bit key in RSA, as illustrated in Table 3.2.

Table 3.2: Comparison of Key Sizes and Security Levels between RSA and ECC

Algorithm	Key Size (bits)	Security Level (bits)
RSA	2048	112
RSA	3072	128
RSA	15360	256
ECC (P-224)	224	112
ECC (P-256)	256	128
ECC (P-521)	521	256

Note: Using a 256-bit curve means that private keys will be 256 bits long, and our public key's x and y coordinates will each be 256 bits long, totaling 512-bit size for the public keys.

3.2.4 AES Galois Counter Mode

AES Galois Counter Mode (GCM) is a robust symmetric encryption algorithm that provides both confidentiality and integrity. We chose AES GCM for the following reasons:

- **High Security:** AES GCM offers strong security guarantees, making it resistant to various cryptographic attacks. It is widely regarded as one of the most secure modes of operation for AES.
- **Integrity and Authenticity:** AES GCM generates an authentication tag during encryption, which can be used to verify the integrity and authenticity of the encrypted data. This dual functionality is particularly valuable in ensuring data security.
- **Standardization:** AES GCM is a standardized encryption mode, widely adopted in various security protocols and applications, ensuring compatibility and interoperability.

Figure 3.3 [4] illustrates the AES GCM process. It begins with the initialization vector being fed into a counter. The counter's value, along with the symmetric encryption key, is used by the AES encryption function to generate a series of encrypted blocks. These encrypted blocks are then XORed with the corresponding plaintext blocks to produce the ciphertext.

Simultaneously, the encrypted blocks are processed through a Galois field multiplication (GF2mul), which contributes to generating an authentication tag (TAG) for integrity verification. The output of the AES GCM process is both the encrypted message (ciphertext) and the authentication tag, ensuring confidentiality and integrity of the data. The diagram also shows the handling of additional blocks and the finalization step to complete the authentication process.

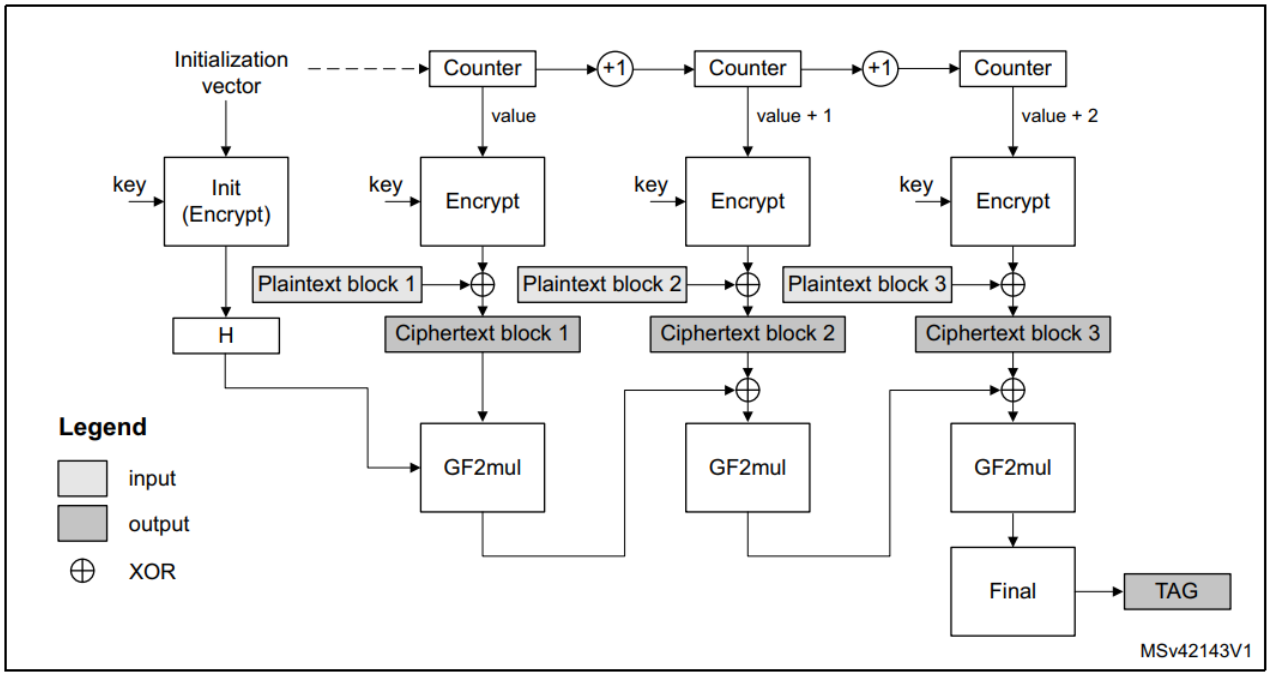

Figure 3.3: AES Galois Counter Mode Process

3.3 Use-Case Diagram

Figure 3.4: Use-Case Diagram for Cryptographic Peripherals (AES and PKA)

The Use-Case diagram in Figure 3.4 illustrates the interactions between a user and the cryptographic peripherals available on STM32 MCUs. Refer to Table 3.3 for an in-depth explanation:

Table 3.3: Use Cases for Cryptographic Peripherals

Use Case	**Relevance**	**Peripheral Used**
Generate Private ECDSA Key	Part of the key management process, ensuring secure storage and handling of private keys, which contributes to confidentiality.	PKA (Public Key Accelerator)
		Continued on next page

Table 3.3 – continued from previous page

Use Case	**Relevance**	**Peripheral Used**
Generate Private ECDH Key	Part of the key management process and essential for establishing a shared secret, contributing to confidentiality.	PKA (Public Key Accelerator)
Compute ECDSA Public Key	Necessary for signature verification, which ensures authenticity.	PKA (Public Key Accelerator)
Compute ECDH Public Key	Necessary for the ECDH key exchange process, contributing to confidentiality.	PKA (Public Key Accelerator)
Generate ECDSA Signature	Ensures the authenticity of the communicating parties by providing a way to verify identities.	PKA (Public Key Accelerator)
Verify ECDSA Signature	Ensures the authenticity of the communicating parties by verifying the provided signatures.	PKA (Public Key Accelerator)
Encrypt Message	Ensures confidentiality and integrity of the message during transmission.	AES (Advanced Encryption Standard)
Generate Tag	Ensures the integrity of the message and also serves as an authentication check.	AES (Advanced Encryption Standard)
Decrypt Message	Ensures confidentiality and integrity of the message during transmission.	AES (Advanced Encryption Standard)
Verify Tag	Ensures the integrity of the message and also serves as an authentication check.	AES (Advanced Encryption Standard)

3.4 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 demonstration of the "CryptoEngine" peripheral.

3.4.1 Hardware Resources

- **STM32XX MCU:** The STM32XX microcontroller unit (MCU) is the primary hardware platform used for implementing and testing the "CryptoEngine" peripheral. The STM32XX series offers advanced cryptographic features and peripherals, making it ideal for secure communication applications.
- **STM32U545 MCU:** The STM32U545 microcontroller unit is used for comparison and interoperability with the STM32XX MCU. This board utilizes the AES and PKA peripherals to perform cryptographic operations, enabling secure key exchange and encryption.

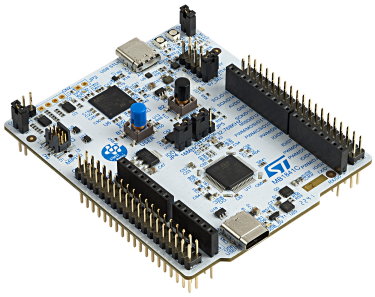

Figure 3.5: STM32U545 Nucleo Board

3.4.2 Software Resources

The following software tools were used for the development, debugging and testing of the "CryptoEngine" demonstration:

3.4.2.1 IAR Embedded Workbench

IAR Embedded Workbench is an integrated development environment (IDE) used for programming, debugging, and optimizing embedded applications. It provides comprehensive support for STM32 microcontrollers, including the STM32XX and STM32U5 series. Key features include:

- Advanced debugging capabilities with breakpoints, watch windows, and real-time data visualization.
- Code optimization tools to improve performance and reduce memory footprint.
- Integrated support for STM32CubeMX, allowing seamless project setup and configuration.

Figure 3.6: IAR Embedded Workbench

3.4.2.2 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.

Figure 3.7: STM32CubeMX

3.4.2.3 Yet Another Terminal (YAT)

Yet Another Terminal (YAT) is a terminal application used for serial communication with the STM32 microcontrollers. It provides a user-friendly interface for monitoring and interacting with the microcontroller's output. Key features include:

- Support for multiple communication protocols, including USART.
- Real-time data logging and visualization capabilities.
- Customizable settings for baud rate, data bits, parity, and stop bits.

Conclusion

The work environment for this internship project includes both hardware and software resources that are essential for the successful implementation and demonstration of the "CryptoEngine" peripheral. The STM32XX

MCU provides the necessary hardware capabilities, while the STM32U545 MCU is used for comparison and interoperability, utilizing AES and PKA peripherals. IAR Embedded Workbench and STM32CubeMX offer powerful tools for software development and configuration. Together, these resources enable the creation of a secure, efficient, and user-friendly cryptographic demonstration.

CHAPTER 4

SOLUTION IMPLEMENTATION

Contents

Introduction

In this chapter, we provide a detailed step-by-step explanation of the implementation of the "CryptoEngine" demo. This demo showcases the use of cryptographic techniques to establish secure communication between two microcontrollers, specifically the STM32XX and STM32U545 MCUs.

4.1 Demo Overview

The following section outlines the sequence of operations involved in the "CryptoEngine" demo. The steps are detailed in Table 4.1:

Table 4.1: Steps for "CryptoEngine" Demo

Step	**Alice**	**Bob**
1	Generate ECDSA Key Pair	Generate ECDSA Key Pair
	(ECDSA keys are used for creating and verifying digital signatures)	
2	Generate ECDH Key Pair	Generate ECDH Key Pair
	(ECDH keys are used for establishing a shared secret)	
3	Send ECDSA Public Key + Signature to Bob	Send ECDSA Public Key + Signature to Alice
4	Verify Bob's Signature using his ECDSA Public Key	Verify Alice's Signature using her ECDSA Public Key
	(Ensures the authenticity of the participants)	
5	Send ECDH Public Key to Bob	Send ECDH Public Key to Alice
6	Compute ECDH Shared Secret using own Private ECDH Key and Bob's Public ECDH Key	Compute ECDH Shared Secret using own Private ECDH Key and Alice's Public ECDH Key
	(Shared secret is used as the basis for deriving encryption keys)	

Continued on next page

Table 4.1 – continued from previous page

Step	Alice	Bob
7	Derive AES GCM Key and IV from ECDH Shared Secret	Derive AES GCM Key and IV from ECDH Shared Secret
8	Encrypt and Send Message using AES GCM	Decrypt and Verify Message using AES GCM
9	Decrypt and Verify Response using AES GCM	Encrypt and Send Response using AES GCM

4.2 Sequence Diagram

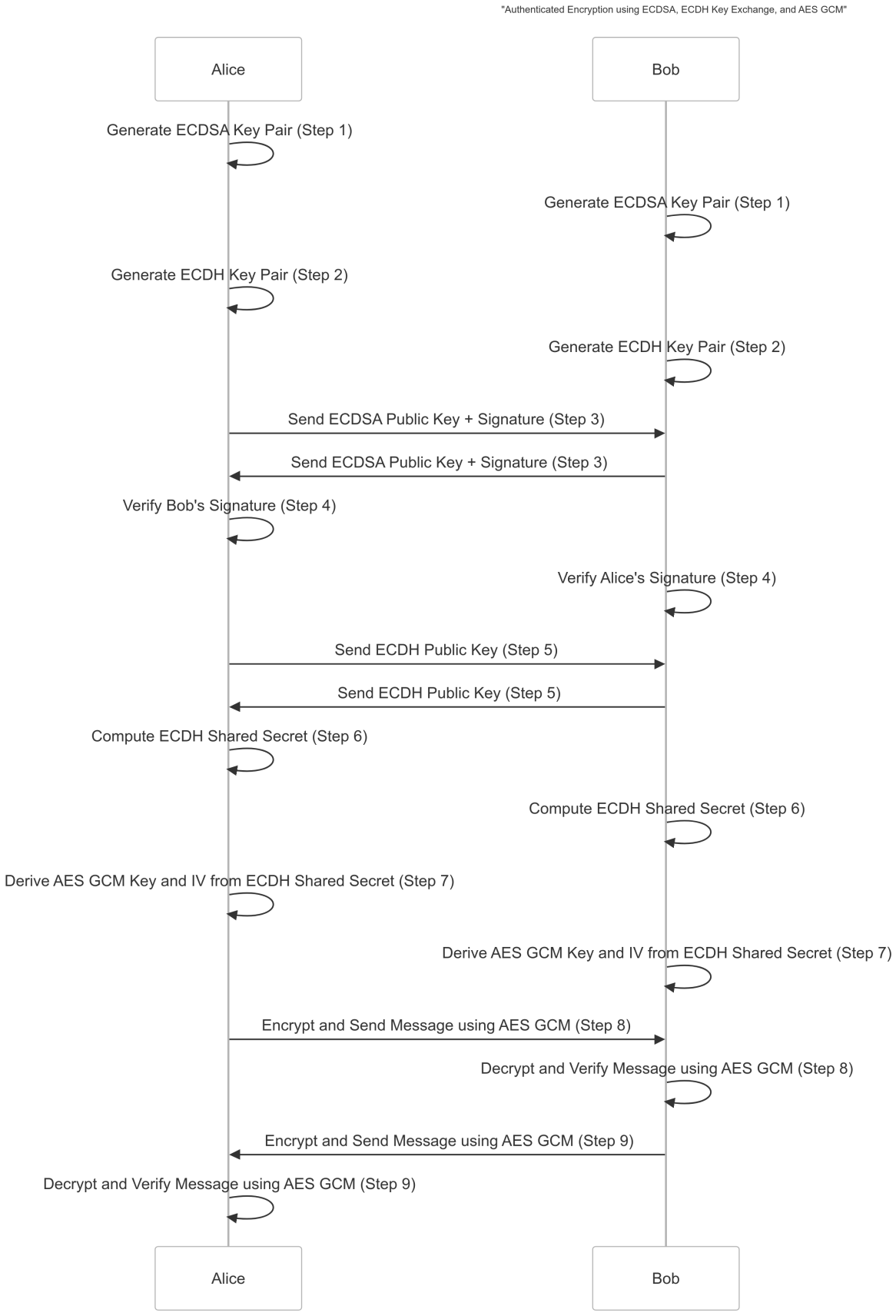

Figure 4.1: Sequence Diagram for "CryptoEngine" Demo

The Sequence Diagram in Figure 4.1 illustrates the process of authenticated encryption using ECDSA for authentication, ECDH for key exchange, and AES GCM for encryption. The diagram shows the interactions between two participants, Alice and Bob, as they perform cryptographic operations to securely exchange messages.

Note : In our demo, the STM32XX and STM32U545 MCUs will play the roles of Alice and Bob respectively.

This sequence diagram demonstrates a secure communication process where both participants authenticate each other using ECDSA, establish a shared secret using ECDH, and securely exchange messages using AES GCM. The use of these cryptographic techniques ensures the confidentiality, integrity, and authenticity of the exchanged messages.

4.3 Demonstration Details and Explanation

For confidentiality reasons we will only detail the "Bob" side of the demo, since it is using the STM32U545 MCU.

4.3.1 ECDSA Key Pair Generation

This is the first step of the demo. It involves setting up PKA peripheral to generate the public ECDSA key.

4.3.1.1 PKA Peripheral Initialization

To compute ECDSA public key we need to set the PKA mode to "ECC Fp scalar multiplication", then we need to input the cryptographic parameters in the correct addresses in PKA RAM.

4.3.1.2 ECC Scalar Multiplication

To input cryptographic parameters in PKA RAM, we need to refer to the STM32U5 reference manual [4] where we can find the specifications for using ECC Fp scalar multiplication. The details of this operation can be found in Figure 4.2.

This figure details the address and size of every cryptography parameter involved in ECC multiplication, which we are using at this level to compute the ECDSA public key. Note that in our demonstration, since we are using a 256-bit curve, the size labeled as EOS (ECC Operand Size) is equal to 320 bits which are 256 bits of actual data and 64 bits of zeros which are necessary when inputting certain data to PKA RAM, as mentioned in the reference manual for U5 [4].

Table 505. ECC Fp scalar multiplication

Parameters with direction		Value (note)	Storage	Size
IN	MODE	0x20	PKA_CR	6 bits
IN	Curve prime order *n* length	(in bits, not null,)	RAM@0x400	64 bits
	Curve modulus *p* length	(in bits, not null, 8 < value < 640)	RAM@0x408	
	Curve coefficient *a* sign	– 0x0: positive – 0x1: negative	RAM@0x410	

Table 505. ECC Fp scalar multiplication (continued)

Parameters with direction		Value (note)	Storage	Size
IN	Curve coefficient |*a*|	(absolute value, |a| < p)	RAM@0x418	EOS
	Curve coefficient *b*	(positive integer)	RAM@0x520	
	Curve modulus value *p*	(odd integer prime, $0 < p < 2^{640}$)	RAM@0x1088	
	Scalar multiplier k	($0 \le k < 2^{640}$)	RAM@0x12A0	
	Point P coordinate x_P	(x < *p*)	RAM@0x578	
	Point P coordinate y_P	(y < *p*)	RAM@0x470	
	Curve prime order *n*	(integer prime)	RAM@0xF88	
OUT	Result: k x P coordinate *x'*	(result < *p*)	RAM@0x578	
	Result: k x P coordinate *y'*	(result < *p*)	RAM@0x5D0	
ERROR	Error k x P	– No errors: 0xD60D – Errors: 0xCBC9	RAM@0x680	64 bits

Figure 4.2: PKA ECC Fp Scalar Multiplication

In our case the parameters that we need to use are provided by NIST [5] and are specified in the following table.

Table 4.2: NIST P-256 Curve Specifications

Name	Value
Curve Modulus p	0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
Curve Coefficient a	0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc
Curve Coefficient b	0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
Generator Point G	(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296, 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)

Continued on next page

NIST P-256 Curve Specifications (continued)

Name	Value
Curve prime order n	0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

Note : These same curve parameters will be reused for ECDSA signature generation and verification as well as ECDH key pair generation.

These curve specifications have to be put in their specific adresses in PKA RAM in order to set the stage for elliptic curve operations on the P-256 curve.

Figure 4.3 demonstrates the code implementation of the curve parameters.

```
  /*Curve Modulus P */
const uint8_t prime256v1_Prime[] = {
    0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff
};
  /*Curve Coefficient a */
const uint8_t prime256v1_A[] = {
    0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xfc
};
  /*Curve Coefficient b */
const uint8_t prime256v1_B[] = {
    0x5a, 0xc6, 0x35, 0xd8, 0xaa, 0x3a, 0x93, 0xe7, 0xb3, 0xeb, 0xbd, 0x55, 0x76, 0x98, 0x86,
    0xbc, 0x65, 0x1d, 0x06, 0xb0, 0xcc, 0x53, 0xb0, 0xf6, 0x3b, 0xce, 0x3c, 0x3e, 0x27, 0xd2,
    0x60, 0x4b
};

  /*Generator G(X,Y) */
 uint8_t prime256v1_GeneratorX[] = {
    0x6b, 0x17, 0xd1, 0xf2, 0xe1, 0x2c, 0x42, 0x47, 0xf8, 0xbc, 0xe6, 0xe5, 0x63, 0xa4, 0x40,
    0xf2, 0x77, 0x03, 0x7d, 0x81, 0x2d, 0xeb, 0x33, 0xa0, 0xf4, 0xa1, 0x39, 0x45, 0xd8, 0x98,
    0xc2, 0x96
};

 uint8_t prime256v1_GeneratorY[] = {
    0x4f, 0xe3, 0x42, 0xe2, 0xfe, 0x1a, 0x7f, 0x9b, 0x8e, 0xe7, 0xeb, 0x4a, 0x7c, 0x0f, 0x9e,
    0x16, 0x2b, 0xce, 0x33, 0x57, 0x6b, 0x31, 0x5e, 0xce, 0xcb, 0xb6, 0x40, 0x68, 0x37, 0xbf,
    0x51, 0xf5
};
  /*Prime Order n */
const uint8_t prime256v1_Order[] = {
    0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xbc, 0xe6, 0xfa, 0xad, 0xa7, 0x17, 0x9e, 0x84, 0xf3, 0xb9, 0xca, 0xc2, 0xfc,
    0x63, 0x25, 0x51
};
```

Figure 4.3: Code Implementation of NIST P-256 Parameters

We will use PKA structures provided by the Hardware Abstraction Layer (HAL) to configure our curve parameters, as illustrated in Figure 4.4. We have developed a function to fill in this structure's members with the NIST P-256 curve parameters as illustrated in Figure 4.5

```
typedef struct
{
  uint32_t scalarMulSize;             /*!< Number of element in scalarMul array */
  uint32_t modulusSize;               /*!< Number of element in modulus, coefA, pointX and pointY arrays */
  uint32_t coefSign;                  /*!< Curve coefficient a sign */
  const uint8_t *coefA;               /*!< Pointer to curve coefficient |a| (Array of modulusSize elements) */
  const uint8_t *coefB;               /*!< pointer to curve coefficient b */
  const uint8_t *modulus;             /*!< Pointer to curve modulus value p (Array of modulusSize elements) */
  const uint8_t *pointX;              /*!< Pointer to point P coordinate xP (Array of modulusSize elements) */
  const uint8_t *pointY;              /*!< Pointer to point P coordinate yP (Array of modulusSize elements) */
  const uint8_t *scalarMul;           /*!< Pointer to scalar multiplier k   (Array of scalarMulSize elements) */
  const uint8_t *primeOrder;          /*!< pointer to order of the curve */
} PKA_ECCMulInTypeDef;
```

Figure 4.4: HAL PKA ECC Input Structure

```
void Setup_ECC_Input(PKA_ECCMulInTypeDef* ecdh_input_pointer)
{

  ecdh_input_pointer->modulusSize =     prime256v1_Prime_len;
  ecdh_input_pointer->coefSign =        prime256v1_A_sign;
  ecdh_input_pointer->coefA =           prime256v1_absA;
  ecdh_input_pointer->coefB =           prime256v1_B;

  ecdh_input_pointer->modulus =         prime256v1_Prime;
  ecdh_input_pointer->scalarMulSize =   prime256v1_Prime_len;
  ecdh_input_pointer->primeOrder =      prime256v1_Order;
}
```

Figure 4.5: PKA ECC Setup

After setting up the elliptic curve parameters, we can generate the ECDSA public key using the ECDSA private key illustrated in Figure 4.6. Figure 4.7 shows the function used to generate the public key, and Figure 4.8 shows the generated key, transmitted through USART to our terminal application. As previously mentioned in 3.2.3, the ECDSA public key is a point on the elliptic curve which has a X and Y coordinate, each of them being 256 bits.

```
uint8_t ECDSA_UserKey[32] = {
  0x0f, 0x56, 0xdb, 0x78, 0xca, 0x46, 0x0b, 0x05, 0x5c, 0x50, 0x00, 0x64, 0x82, 0x4b, 0xed, 0x99,
  0x9a, 0x25, 0xaa, 0xf4, 0x8e, 0xbb, 0x51, 0x9a,0xc2, 0x01, 0x53, 0x7b, 0x85, 0x47, 0x98, 0x13
};
```

Figure 4.6: ECDSA Private Key

```
void pka_ecdsa_generate_publickey(PKA_HandleTypeDef* hpka,
                                  PKA_ECCMulInTypeDef* ecdsa_input_pointer,
                                  uint8_t* ecdsa_private_key,
                                  PKA_ECCMulOutTypeDef* ECDSA_GeneratedPublicKey)
{

  ecdsa_input_pointer->pointX =          prime256v1_GeneratorX;
  ecdsa_input_pointer->pointY =          prime256v1_GeneratorY;
  ecdsa_input_pointer->scalarMul =       ecdsa_private_key;

   if(HAL_PKA_ECCMul(hpka, ecdsa_input_pointer, 5000) != HAL_OK)
  {
    Error_Handler();
  }

  ECDSA_GeneratedPublicKey->ptX =        malloc(prime256v1_Prime_len);
  ECDSA_GeneratedPublicKey->ptY =        malloc(prime256v1_Prime_len);

  HAL_PKA_ECCMul_GetResult(hpka,ECDSA_GeneratedPublicKey);
}
```

Figure 4.7: ECDSA Public Key Generation Function

```
ECDSA PubKeyX:
âfÝýÁ&hÛ0ÔÊ><0x8f>wIC,A`DòÒ¸Á<0x0b>óÔ<0x01>*ïú<0x8a>
ECDSA PubKeyY:
¿¨d<0x04>¢éÿæ}GÅ<0x87>ïz<0x97>§ôV¸c´Ð,üi(<0x97>:µ±Ë9
```

Figure 4.8: Generated ECDSA Public Key

4.3.2 ECDH Key Pair Generation

This is the second step of the demo. Similarly to the previous step, we will use PKA to generate the ECDH public key.

4.3.2.1 ECC Scalar Multiplication

The key generation process using "ECC Fp scalar multiplication" is the same for ECDSA and ECDH, as described in Figure 4.2. We will use the same PKA structure as in ECDSA key generation as illustrated in Figure 4.4.

The ECDH private key illustrated in Figure 4.9 is used along with the function shown in Figure 4.10 in order to generate the ECDH public key, which is shown in Figure 4.11.

```
uint8_t ECDH_UserKey[32] = {
 0x51, 0x9b, 0x42, 0x3d, 0x71, 0x5f, 0x8b, 0x58, 0x1f, 0x4f, 0xa8, 0xee, 0x59, 0xf4, 0x77, 0x1a,
 0x5b, 0x44, 0xc8, 0x13, 0x0b, 0x4e, 0x3e, 0xac, 0xca, 0x54, 0xa5, 0x6d, 0xda, 0x72, 0xb4, 0x64
};
```

Figure 4.9: ECDH Private Key

```
void pka_ecdh_generate_publickey(PKA_HandleTypeDef* hpka,
                                 PKA_ECCMulInTypeDef* ecdh_input_pointer,
                                 uint8_t* ecdh_private_key,
                                 PKA_ECCMulOutTypeDef* ECDH_GeneratedPublicKey)
{

  ecdh_input_pointer->pointX =          prime256v1_GeneratorX;
  ecdh_input_pointer->pointY =          prime256v1_GeneratorY;
  ecdh_input_pointer->scalarMul =       ecdh_private_key;

  if(HAL_PKA_ECCMul(hpka, ecdh_input_pointer, 5000) != HAL_OK)
  {
    Error_Handler();
  }

  ECDH_GeneratedPublicKey->ptX =        malloc(prime256v1_Prime_len);
  ECDH_GeneratedPublicKey->ptY =        malloc(prime256v1_Prime_len);

  HAL_PKA_ECCMul_GetResult(hpka,ECDH_GeneratedPublicKey);
}
```

Figure 4.10: ECDH Public Key Generation Function

```
ECDH PubKeyX:
<0x1c>Ëé<0x1c><0x07>_Çðð3¿¢HÛ<0x8f>ÌÓV]éK¿±/<YÿFÂq¿<0x83>

ECDH PubKeyY:
Î@<0x14>Æ<0x88><0x11>ù¢<0x1a><0x1f>Û,<0x0e>a<0x13>àm·Ê<0x93>·@NxÜ|Í\¨<0x9a>L©
```

Figure 4.11: Generated ECDH Public Key

4.3.3 ECDSA Signature Generation

To generate an ECDSA signature, we will use the PKA mode illustrated in Figure 4.12 [4].

Parameters with direction		Value (note)	Storage	Size
IN	MODE	0x24	PKA_CR	6 bits
	Curve prime order n length (*nlen*)	(in bits, not null)	RAM@0x400	64 bits
	Curve modulus p length	(in bits, 8 < value < 640)	RAM@0x408	
	Curve coefficient a sign	0x0: positive 0x1: negative	RAM@0x410	
	Curve coefficient |a|	(absolute value, |a| < p)	RAM@0x418	EOS
	Curve coefficient b	(positive integer)	RAM@0x520	
	Curve modulus value *p*	(odd integer prime, $0 < p < 2^{640}$)	RAM@0x1088	
	Integer k(1)	($0 \le k < 2^{640}$)	RAM@0x12A0	
	Curve base point G coordinate x	(x < *p*)	RAM@0x578	
	Curve base point G coordinate y	(y < *p*)	RAM@0x470	
	Hash of message z	(hash size equal to *nlen*)(2)	RAM@0xFE8	
	Private key d	(0 < d)	RAM@0xF28	
	Curve prime order n	(integer prime)	RAM@0xF88	

Figure 4.12: ECDSA Sign Operation

We will use the PKA ECDSA signature structure provided by the HAL driver shown in Figure 4.13 along with the function shown in 4.14 to input the signature parameters in PKA RAM.

```
typedef struct
{
  uint32_t primeOrderSize;          /*!< Number of element in primeOrder array */
  uint32_t modulusSize;             /*!< Number of element in modulus array */
  uint32_t coefSign;                /*!< Curve coefficient a sign */
  const uint8_t *coef;              /*!< Pointer to curve coefficient |a|     (Array of modulusSize elements) */
  const uint8_t *coefB;             /*!< Pointer to B coefficient             (Array of modulusSize elements) */
  const uint8_t *modulus;           /*!< Pointer to curve modulus value p     (Array of modulusSize elements) */
  const uint8_t *integer;           /*!< Pointer to random integer k          (Array of primeOrderSize elements) */
  const uint8_t *basePointX;        /*!< Pointer to curve base point xG       (Array of modulusSize elements) */
  const uint8_t *basePointY;        /*!< Pointer to curve base point yG       (Array of modulusSize elements) */
  const uint8_t *hash;              /*!< Pointer to hash of the message       (Array of primeOrderSize elements) */
  const uint8_t *privateKey;        /*!< Pointer to private key d             (Array of primeOrderSize elements) */
  const uint8_t *primeOrder;        /*!< Pointer to order of the curve n      (Array of primeOrderSize elements) */
} PKA_ECDSASignInTypeDef;
```

Figure 4.13: HAL PKA ECDSA Input Structure

```
void Setup_ECDSA_Input(PKA_ECDSASignInTypeDef* ecdsa_input_pointer)
{

  ecdsa_input_pointer->primeOrderSize =  prime256v1_Order_len;
  ecdsa_input_pointer->modulusSize =     prime256v1_Prime_len;
  ecdsa_input_pointer->coefSign =        prime256v1_A_sign;
  ecdsa_input_pointer->coef =            prime256v1_absA;
  ecdsa_input_pointer->coefB =           prime256v1_B;

  ecdsa_input_pointer->modulus =         prime256v1_Prime;
  ecdsa_input_pointer->primeOrder =      prime256v1_Order;
  ecdsa_input_pointer->basePointX =      prime256v1_GeneratorX;
  ecdsa_input_pointer->basePointY =      prime256v1_GeneratorY;

}
```

Figure 4.14: HAL PKA ECDSA Input Structure

Compared to public key generation, generating the signature requires a new private parameter called "k" which is a 256-bit number only used once to generate the signature. The integer "k" used for this demo is shown in Figure 4.15. The signature generation also uses a hash message, which in our demonstration is pre-shared between the two parties.

```
uint8_t ECDSA_K_Integer[32] = {
  0x94, 0xa1, 0xbb, 0xb1, 0x4b, 0x90, 0x6a, 0x61, 0xa2, 0x80, 0xf2, 0x45, 0xf9, 0xe9, 0x3c, 0x7f,
  0x3b, 0x4a, 0x62, 0x47, 0x82, 0x4f, 0x5d, 0x33, 0xb9, 0x67, 0x07, 0x87, 0x64, 0x2a, 0x68, 0xde
};
```

Figure 4.15: ECDSA Integer "K"

After setting up the ECDSA HAL structure, we can launch the ECDSA signing operation using the function shown in Figure 4.16.

```
void pka_ecdsa_generate_signature(PKA_HandleTypeDef* hpka,
                                  PKA_ECDSASignInTypeDef* ecdsa_input_pointer,
                                  uint8_t* ecdsa_private_key,
                                  uint8_t* ECDSA_K_Integer,
                                  uint8_t* hash_msg,
                                  PKA_ECDSASignOutTypeDef* ecdsa_output_pointer)
{
  ecdsa_input_pointer->privateKey =     ecdsa_private_key;
  ecdsa_input_pointer->hash       =     hash_msg;
  ecdsa_input_pointer->integer    =     ECDSA_K_Integer  ;


  /* Allocate required space */
  ecdsa_output_pointer->RSign=     malloc(prime256v1_Order_len);
  ecdsa_output_pointer->SSign=     malloc(prime256v1_Order_len);
  if (ecdsa_output_pointer->RSign== NULL || ecdsa_output_pointer->SSign == NULL)
  {
    /* Not enough memory in heap */
    Error_Handler();
  }
    /* Launch the verification */
  if(HAL_PKA_ECDSASign(hpka, ecdsa_input_pointer, 5000) != HAL_OK)
  {
    Error_Handler();
  }
  HAL_PKA_ECDSASign_GetResult(hpka ,ecdsa_output_pointer, NULL);
}
```

Figure 4.16: ECDSA Signing Function

The generated signature is composed of two parts "Signature R" and "Signature S", each being 256 bits long. Figure 4.17 shows the output of the signing operation.

```
Signature R:
ó¬<0x80>aµ<0x14>y[<0x88>CãÖb<0x95>'í*ýk<0x1f>jUZzÊ»^oyÈÂ¬

Signature S:
<0x8b>÷x<0x19>Ê<0x05>¦²xlv&+÷7<0x1c>ï<0x97>²<0x18>éo<0x17>Z<ÍÚ*Ì<0x05><0x89><0x03>
```

Figure 4.17: Generated ECDSA Signature

4.3.4 ECDSA Public Key and Signature Exchange

After having generated the ECDSA public keys and signatures, the two MCUs exchange these parameters using USART. Exchanging these parameters is crucial for the following step, which is signature verification, to ensure the authenticity of the communication.

Figure 4.18 shows the ECDSA public key and signature received by "Bob" which were generated by "Alice".

```
Received ECDSA Public Key X:
<0x85>$Å<0x02>N-<0x9a>s½èÇ-<0x91>)õxs»<0xad><0x0e>ÐR<0x15>£r¨OÛÇ<0x8f>.h
Received ECDSA Public Key Y:
<0x0f>gWj0¸âB2ØS<0x0b>RûL<0x89>ËÅ<0x89>íâ<0x91>ä<0x99>ÝÑ_èp«<0x96>

Received Signature R:
0Åb7ö<0x17>µ<0x81>ÈJ$hÌ<0x8b>Õ<0x0b>±Ëó"ÞA·<0x88>|à|<0x0e>X<0x84>Ê

Received Signature S:
´m<0x9f>-<0x8c>Kø5Fÿ<0x17><0x8f><0x1d>x<0x93>|<0x8d>dèìÅË¸%Ë!ÙMg<0x89>
```

Figure 4.18: Received ECDSA Signature

4.3.5 ECDSA Signature Verification

In this step, each device verifies the ECDSA signature sent by the other party using the PKA mode "ECDSA Verification" shown in Figure 4.19 [4]. A lot of the same curve parameters are the same as ECDSA signature generation , with the addition of the received public key and signature parameters.

Parameters with direction		Value (note)	Storage	Size
IN	MODE	0x26	PKA_CR	6 bits
	Curve prime order n length (*nlen*)	(in bits, not null)	RAM@0x408	64 bits
	Curve modulus p length	(in bits, not null, 8 < value < 640)	RAM@0x4C8	
	Curve coefficient a sign	0x0: positive 0x1: negative	RAM@0x468	
	Curve coefficient |a|	(absolute value, |a| < p)	RAM@0x470	EOS
	Curve modulus value *p*	(odd integer prime, $0< p < 2^{640}$)	RAM@0x4D0	
	Curve base point G coordinate x	(x < *p*)	RAM@0x678	
	Curve base point G coordinate y	(y < *p*)	RAM@0x6D0	
	Public-key curve point Q coordinate x_Q	(x_Q < *p*)	RAM@0x12F8	
	Public-key curve point Q coordinate y_Q	(y_Q < *p*)	RAM@0x1350	
	Signature part *r*	(0 < *r* < n)	RAM@0x10E0	
	Signature part *s*	(0 < *s* < n)	RAM@0xC68	
	Hash of message *z*	(hash size equal to *nlen*)(1)	RAM@0x13A8	
	Curve prime order *n*	(integer prime)	RAM@0x1088	

Figure 4.19: PKA "ECDSA Verification" Mode

Figure 4.20 shows the code implementation of the ECDSA signature verification process.

```
uint32_t pka_ecdsa_sigVerif(PKA_HandleTypeDef hpka,
                            PKA_ECDSAVerifInTypeDef* inPointer,
                            const uint8_t *PublicKeyCurvePtX,
                            const uint8_t *PublicKeyCurvePtY,
                            const uint8_t *Hash_Msg,
                            const uint8_t *Signature_R,
                            const uint8_t *Signature_S)
{
  /*This function returns 1 when signature is verified, 0 otherwise.
  Set the parameters of ECDSA Signature Verification then verify and check signature validity.*/
  /*Fill in ECDSA Verif Params*/

  inPointer->primeOrderSize =  prime256v1_Order_len;
  inPointer->modulusSize =     prime256v1_Prime_len;
  inPointer->coefSign =        prime256v1_A_sign;
  inPointer->coef =            prime256v1_absA;
  inPointer->modulus =         prime256v1_Prime;
  inPointer->basePointX =      prime256v1_GeneratorX;
  inPointer->basePointY =      prime256v1_GeneratorY;
  inPointer->primeOrder =      prime256v1_Order;
  inPointer->hash =            Hash_Msg; //The sender and receiver have to use the same hash
  inPointer->pPubKeyCurvePtX = PublicKeyCurvePtX; //Received Public Key
  inPointer->pPubKeyCurvePtY = PublicKeyCurvePtY;
  inPointer->RSign = Signature_R;//Received Signature
  inPointer->SSign = Signature_S;

  HAL_PKA_ECDSAVerif(&hpka, inPointer, 5000);
  return HAL_PKA_ECDSAVerif_IsValidSignature(&hpka);
}
```

Figure 4.20: ECDSA Verification Function

There are two outcomes to signature verification. Figure 4.21 shows the output of a successful signature verification process, which allows to go on to establish ECDH key exchange, while Figure 4.22 shows the output of a failed signature verification which abruptly ends the communication.

```
Received Signature R:
×Åb7ö<0x17>µ<0x81>ÈJ$hÌ<0x8b>Õ<0x0b>±Ëó"ÞA·<0x88>|à|<0x0e>X<0x84>Ê

Received Signature S:
´m<0x9f>-<0x8c>Kø5Fÿ<0x17><0x8f><0x1d>x<0x93>|<0x8d>dèìÅË¸%Ë!ÙMg<0x89>

Signature verified succesfully!

Waiting to receive ECDH public Key...
-->
```

Figure 4.21: Successful ECDSA Signature Verification

```
Received Signature R:
0Åb7ö<0x17>µ<0x81>ÈJ$hÌ<0x8b>Õ<0x0b>±Ëó"ÞA·<0x88>|à|<0x0e>X<0x84>Ê

Received Signature S:
´m<0x9f>-<0x8c>Kø5Fÿ<0x17><0x8f><0x1d>x<0x93>|<0x8d>dèìÅË¸%Ë!ÙMg<0x89>

Signature Verification Failed !
```

Figure 4.22: Failed ECDSA Signature Verification

4.3.6 ECDH Public Key Exchange

Now that the two parties have verified each other's signature, we can proceed knowing that authenticity has been assured. The next step is to establish a shared secret which will serve as a basis for establishing a symmetric encryption key. For that, the two devices must exchange the ECDH public keys generated at the start of the demo via USART. Figure 4.23 shows the terminal output of the ECDH public key reception process.

```
Waiting to receive ECDH public Key...
-->

Received ECDH Public Key X:
2/<0x80>7<0x1b>öàD%I9<0x1d><0x97>ÁqJ¸<0x7f><0x99><0x0b><0x94><0x9b>ÁxË|C·Â-<0x89>á

Received ECDH Public Key Y:
<<0x15>ÕJ\Æ¹ð<0x9d>èE~<0x87>>³Þ±üëTº²<0x95>Ú`P)O®<0x7f>Ù<0x99>
```

Figure 4.23: Received ECDH Public Key

4.3.7 ECDH Shared Secret Generation

Having exchanged the ECDH public keys, each device will independently calculate the ECDH shared secret which will be a point on the elliptic curve. This point is obtained through multiplying the device's own ECDH private key with the received ECDH public key. Naturally, both devices should obtain the same common 512-bit shared secret.

Shared secret generation is done through ECC scalar multiplication, meaning we will use the PKA mode "ECC Fp scalar multiplication" which has been detailed in Figure 4.2 [4].

Figure 4.24 illustrates the code implementation of ECDH shared secret generation, and Figure 4.25 shows the output result of this operation.

```
void pka_ecdh_generate_shared_secret(PKA_HandleTypeDef* hpka,
                                     PKA_ECCMulInTypeDef*  ecdh_input_pointer,
                                     PKA_ECCMulOutTypeDef* ECDH_ReceivedPublicKey,
                                     uint8_t* ecdh_private_key,
                                     PKA_ECCMulOutTypeDef* ECDH_SharedSecret)
{

  uint8_t  ECDH_Received_X[32]={0};
  uint8_t  ECDH_Received_Y[32]={0};
  uint8_t  ECDH_PrivKey[32]={0};


  ecdh_input_pointer->pointX =         (uint8_t *)ECDH_Received_X;
  ecdh_input_pointer->pointY =         (uint8_t *)ECDH_Received_Y;
  ecdh_input_pointer->scalarMul =      (uint8_t *)ECDH_PrivKey;


   if(HAL_PKA_ECCMul(hpka, ecdh_input_pointer, 5000) != HAL_OK)
  {
    Error_Handler();
  }
  ECDH_SharedSecret->ptX =               malloc(prime256v1_Prime_len);
  ECDH_SharedSecret->ptY =               malloc(prime256v1_Prime_len);

  HAL_PKA_ECCMul_GetResult(hpka,ECDH_SharedSecret);
}
```

Figure 4.24: ECDH Shared Secret Generation Function

```
ECDH Shared Secret X:
Qù<0x9d>-RÔ¦ç4HJ<0x01><0x8b>|¢ø<0x95>Â<0x92><0x9b>gT£<0xa0>2$Ðzæ<0x11>fÎ

ECDH Shared Secret Y:
G7Ú<0x96><n÷$<0x7f>,<0x8d><0x19>ù°ÆgÊÇþ<0x12><0x83><0x7f>Ú,<0x8c>fñ<<0x14>ÊÑ
```

Figure 4.25: ECDH Shared Secret

4.3.8 AES GCM Key and IV Derivation

Having established the shared secret, we will now derive from it the symmetric encryption key as well as the initialization vector which are needed for AES GCM . Usually, this process would involve a key derivation function and the use of complicated algorithms, but for the purposes of the simplicity of the demo we opted to take the X coordinate of the ECDH shared secret as the 256-bit AES Key and the first 96 bits of the Y coordinate as the initialization vector.

4.3.9 AES GCM Symmetric Encryption

For this step, we will use the AES peripheral structure provided by the HAL driver provided in Figure 4.26. We will initialize the peripheral's structure as shown in Figure 4.27, where we select a 256-bit key size and AES GCM mode.

```
typedef struct
{
  uint32_t DataType;                  /*!< 32-bit data, 16-bit data, 8-bit data or 1-bit string.
                                       This parameter can be a value of @ref CRYP_Data_Type */
  uint32_t KeySize;                   /*!< Used only in AES mode : 128, 192 or 256 bit key length in CRYP1.
                                       128 or 256 bit key length in TinyAES This parameter can be a value
                                       of @ref CRYP_Key_Size */
  uint32_t *pKey;                     /*!< The key used for encryption/decryption */
  uint32_t *pInitVect;                /*!< The initialization vector used also as initialization
                                        counter in CTR mode */
  uint32_t Algorithm;                 /*!<  DES/ TDES Algorithm ECB/CBC
                                       AES Algorithm ECB/CBC/CTR/GCM or CCM
                                       This parameter can be a value of @ref CRYP_Algorithm_Mode */
  uint32_t *Header;                   /*!< used only in AES GCM and CCM Algorithm for authentication,
                                       GCM : also known as Additional Authentication Data
                                       CCM : named B1 composed of the associated data length and Associated Data. */
  uint32_t HeaderSize;                /*!< The size of header buffer */
  uint32_t *B0;                       /*!< B0 is first authentication block used only  in AES CCM mode */
  uint32_t DataWidthUnit;             /*!< Payload Data Width Unit, this parameter can be value of @ref CRYP_Data_Width_Unit */
  uint32_t HeaderWidthUnit;           /*!< Header Width Unit, this parameter can be value of @ref CRYP_Header_Width_Unit */
  uint32_t KeyIVConfigSkip;           /*!< CRYP peripheral Key and IV configuration skip, to config Key and Initialization
                                          Vector only once and to skip configuration for consecutive processings.
                                          This parameter can be a value of @ref CRYP_Configuration_Skip */
  uint32_t KeyMode;                   /*!< Key mode selection, this parameter can be value of @ref CRYP_Key_Mode */
  uint32_t KeySelect;                 /*!< Only for SAES : Key selection, this parameter can be value of @ref CRYP_Key_Select */
  uint32_t KeyProtection;             /*!< Only for SAES : Key protection, this parameter can be value of @ref CRYP_Key_Protection */

} CRYP_ConfigTypeDef;
```

Figure 4.26: AES HAL Structure

```
static void AES_GCM_Init(void)
{

  hcryp_aes.Instance = AES;
  hcryp_aes.Init.DataType = CRYP_NO_SWAP;
  hcryp_aes.Init.KeySize = CRYP_KEYSIZE_256B;
  hcryp_aes.Init.pInitVect = (uint32_t *)pInitVectAES;
  hcryp_aes.Init.Algorithm = CRYP_AES_GCM_GMAC

  hcryp_aes.Init.HeaderSize = 0;

  hcryp_aes.Init.DataWidthUnit = CRYP_DATAWIDTHUNIT_WORD;


  if (HAL_CRYP_Init(&hcryp_aes) != HAL_OK)
  {
    Error_Handler();
  }

}
```

Figure 4.27: AES Structure Initialization

All that is left is to derive the AES key and initialization vector from the ECDH shared secret as shown in Figure 4.28.

```
//Format ECDH Shared Secret to 32 Bit for use as symmetric AES key.
format_Buffer_8to32(ECDH_Shared_Secret.ptX,32,AES_Symmetric_Key,8);

//Take a part of ECDH secret to use as IV
format_Buffer_8to32(ECDH_Shared_Secret.ptY,32,AES_IV_Source,8);

 Create_AES_IV(AES_IV_Source,AES_IV);

 /* Get the AES parameters */
  HAL_CRYP_GetConfig(&hcryp_aes, &Conf);

/*AES GCM Configuration*/
 Conf.pKey      = AES_Symmetric_Key;
 Conf.pInitVect = AES_IV;
 HAL_CRYP_SetConfig(&hcryp_aes, &Conf);
```

Figure 4.28: AES GCM Key and Initialization Vector Derivation

On the "Bob" side, we will encrypt and send the message "This is Bob" along with the tag that will help in verifying the integrity of the message. Figure 4.29 demonstrates the plaintext, ciphertext and tag generated by the encryption process. Note that the "%" symbol was added for extra padding since the AES peripheral expects a 128-bit input.

```
Plaintext Message :
This is Bob%%%%

Ciphertext Message:
Ù¥§2<0x81><0x17>ô<0x8b>K<0x8d>ØÁ}®r.uk?öN

Encrypt Tag:
/ÂÏøÍÔ<0x86>k<0x1d>tä[<0x07>Ó3¯Fg·<0x08><0x88>
```

Figure 4.29: AES GCM Key and Initialization Vector Derivation

4.3.10 Message Decryption and Tag Verification

The two devices exchange the encrypted messages and the generated tags. They must now decrypt the message while regenrating a tag. If the received tag is the same as the generated decryption tag, that means the message has been decrypted succesfully. Figure 4.30 shows the decryption process on the "Bob" side which shows the correct decryption of the "Alice" message ("This is Alice").

```
Received Tag:
éç<0x95>xø<0x98><0x8b><0x16><0x8e>ßñ¨³JÍ<0x1f>í6q

Decrypted Text:
"This is Alice%%"

Generated Tag:
éç<0x95>xø<0x98><0x8b><0x16><0x8e>ßñ¨³JÍ<0x1f>í6q

Decrypt Tag matches Received Tag .

End of Demo.
```

Figure 4.30: Message Decryption Output

The matching of the tags proves that the message was transmitted correctly and without any manipulation. It also ensures that the two sides have managed to derive the exact same symmetric key from the shared secret generation process, meaning that a secure line of communication has been succesfully established between the two devices.

Conclusion

This chapter provided a comprehensive review of the implementation steps for the "CryptoEngine" demo, which demonstrates secure communication between two microcontrollers using cryptographic techniques. We explored the generation and exchange of cryptographic keys, including ECDSA and ECDH keys, and the processes of signature generation, verification, and secure message exchange using AES GCM. We also included relevant code snippets and expected outputs for each step of the demonstration to provide a complete overview.

Bibliography

[1] STMicroelectronics, "St company presentation," 2024, [Accessed 7 February 2024]. [Online]. Available: https://www.st.com/resource/en/company_presentation/company_presentation.pdf

[2] N. I. of Standards and Technologies, "Recommendations for discrete logarithm-based cryptography: elliptic curve domain parameters," 2023, [Accessed 24 July 2024]. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf

[3] D. Warburton, "The 2021 tls telemetry report," 2021, [Accessed 7 March 2024]. [Online]. Available: https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report

[4] STMicroelectronics, "Rm0456 stm32u5 series arm®-based 32-bit mcus," 2024, [Accessed 8 February 2024]. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0456-stm32u5-series-armbased-32bit-mcus-stmicroelectronics.pdf

[5] J. Jancar, "Standard curve database," 2020, [Accessed 17 June 2024]. [Online]. Available: https://neuromancer.sk/std/nist/P-256

[6] STMicroelectronics, "Stm32 mooc part 3: security features," 2020, [Accessed 13 February 2024]. [Online]. Available: https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/STM32_security_features

[7] STMicroelectronics, "Stm32 mooc part 2:cryptography basics," 2020, [Accessed 12 February 2024]. [Online]. Available: https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/STM32_security_features

[8] STMicroelectronics, "St application note 5156 : introduction to security for stm32 mcus," 2018, [Accessed 27 February 2024]. [Online]. Available: https://www.st.com/resource/en/application_note/an5156-introduction-to-security-for-stm32-mcus-stmicroelectronics

[9] D. A.Popovic, "Understanding blockchain for insurance use cases," 2020, [Accessed 1 July 2024]. [Online]. Available: https://www.researchgate.net/publication/342136274_Understanding_blockchain_for_insurance_use_cases

Annexes

Annexe 1. Exemple d'annexe

Les chapitres doivent présenter l'essentiel du travail. Certaines informations-trop détaillées ou constituant un complément d'information pour toute personne qui désire mieux comprendre ou refaire une expérience décrite dans le document- peuvent être mises au niveau des annexes. Les annexes, **placées après la bibliographie**, doivent donc être numérotées avec des titres (Annexe1, Annexe2, etc.).

Le tableau annexe 1.1 présente un exemple d'un tableau dans l'annexe.

Tableau annexe 1.1: Exemple tableau dans l'annexe

0	0
1	1
2	2
3	3
4	4

Annexe 2. Entreprise

La figure annexe 2.1 présente le logo entreprise.

Figure annexe 2.1: Logo d'entreprise

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.