Tunisian Republic
Ministry of Higher Education and Scientific Research
University of Tunis El Manar
National School of Engineering of Tunis

End Of The Year Project

Thread Metric benchmark tool on CMSIS RTOS v2 API

Developed by :

Iheb Kesraoui
&
Jaafer Hosni

Supervised by :

Mr. Tahar Ezzedine
&
Mr. Haithem Rahmani

2nd Year Info

2nd Year Telecom

Academic year: 2022/2023

Contents

List of Figures

Acknowledgements

We dedicate this page to express our gratitude to the administrative staff of our Honorable school the National School of Engineering Tunis (ENIT) as well as to all the professors of our department of Information and Communication Technologies (ICT).

We would especially like to thank Mr. TAHAR EZZDINE for the enriching and interesting experience he gave us during the supervision of this project, especially for the efforts he made, for his trust and for the knowledge he shared with us. We also thank him for his availability and the quality of his supervision during the realization of our report.

We would especially like to thank Mr. HAITHEM RAHMANI for his technical guidance through the whole process of this project development. His technical advises and shared experience would always be craved in our memory and help us go further into the path of embedded software development.

We would also like to express our gratitude and deep appreciation to the members of the jury for the honor they have given us by accepting to evaluate our work.

Introduction

Embedded systems are a rapidly growing field in the technology industry, with a wide range of applications ranging from consumer electronics to industrial automation. These systems are designed to perform specific tasks and operate within a predefined set of constraints, including limited resources such as processing power, memory, and energy. To accomplish these tasks, embedded systems often rely on real-time operating systems (RTOS) that provide efficient and reliable solutions for managing system resources and responding to real-time events.

The CMSIS RTOS, developed by ARM, is one such RTOS that has gained widespread adoption in the embedded systems community. The CMSIS RTOS is designed to provide a standardized interface for programming multiple RTOS platforms, making it a popular choice for many embedded system applications. Its performance characteristics, such as response time, context switching, and memory footprint, make it a suitable option for various hardware platforms and applications.

The choice of an RTOS is a crucial factor for developers in designing an embedded system. STMicroelectronics recently switched from FreeRTOS middleware to Azure RTOS ThreadX and provided justifications for the change, citing memory footprint and performance as key factors. To assess the performance of these two RTOS platforms and provide a comprehensive evaluation, this project was undertaken. A common set of benchmark tests was used to compare the performance of FreeRTOS and ThreadX, including measuring response time, context switching, and memory footprint.

Problematic

As the original RTOS offer was based on the FreeRTOS middleware, STmicroelectronics should provide evidences and arguments to justify the switch to Azure RTOS threadx. Among these reasons there are the memory footprint and the performance. These two factors are curcial to convince customers about selecting one RTOS or another. This project is dedicated to assess performance of FreeRTOS and ThreadX using a common set of benchmark tests.

By evaluating the results of these tests, developers can make informed decisions about selecting an RTOS for their embedded system projects, and gain insights into the performance characteristics of these two popular RTOS platforms. The findings of this project will contribute to the growing body of knowledge on RTOS platforms, and provide a valuable resource for developers in the embedded systems community.

Chapter 1

Introduction to the company

1.1 Introduction

The purpose of this chapter is to provide an overview of STMicroelectronics [1] and its position in the semiconductor industry. The chapter will cover key facts and figures about the company, as well as an analysis of its products and services, target markets, and competitive landscape. Additionally, the chapter will explore current trends and challenges in the semiconductor industry and how STMicroelectronics is positioned to address these challenges.

As a leading global semiconductor company, STMicroelectronics plays a critical role in the development of electronic devices that are used in a wide range of industries, from automotive to consumer electronics. By providing insights into the company's products and services, this chapter will help readers to better understand the role that STMicroelectronics plays in the industry and the unique value that it offers to its customers.

1.2 History and Missions

STMicroelectronics is a global semiconductor company that was founded in 1987 through a merger between SGS Microelettronica [**?**] of Italy and Thomson Semiconducteurs of France. The company has its headquarters in Geneva, Switzerland, and operates in more than 35 countries worldwide.

The founders of the company were Carlo Gavazzi [9], Pasquale Pistorio [16], and Aldo Romano [17] from SGS Microelettronica, and Alain Gomez and Joel de Rosnay from Thomson Semiconducteurs.

STMicroelectronics primary goal is to design, develop, and market innovative semiconductor solutions that meet the evolving needs of customers in a wide range of industries, including automotive, industrial, telecommunications, and consumer electronics. The company is committed to providing high-quality products and services that help customers improve their performance and competitiveness.

STMicroelectronics has a strong focus on sustainability and corporate responsibility, and is committed to minimizing the environmental impact of its operations and products. The company has been recognized for its efforts in this area, including being named to the Dow Jones Sustainability World Index and the Carbon Disclosure Project's Climate A List.

1.3 Key facts and Figures

1. STMicroelectronics is a global semiconductor company with more than 46,000 employees worldwide.
2. The company operates in more than 35 countries and has research and development centers in Europe, Asia, and the United States.
3. STMicroelectronics' net revenues in 2021 were $12.5 billion USD, up 32.8% from the previous year.
4. The company's product portfolio includes microcontrollers, digital and analog semiconductors, MEMS[2] sensors, and power management ICs, among others.
5. STMicroelectronics is a leading supplier of semiconductors for the automotive industry, with a market share of approximately 10%.
6. The company has been recognized for its sustainability efforts, including being named to the Dow Jones Sustainability World Index and the Carbon Disclosure Project's Climate A List.
7. In 2021, STMicroelectronics announced its goal to become carbon neutral by 2027, and to reduce its total greenhouse gas emissions by 50% by 2030.
8. STMicroelectronics is listed on the New York Stock Exchange (NYSE) and the Euronext Paris stock exchange under the ticker symbol "STM".

Figure 1.1: STMicroelectronics logo

1.4 Products and Services

STMicroelectronics is a global semiconductor company that designs, develops, and manufactures a wide range of products for a variety of industries. The company's product portfolio includes microcontrollers[3], sensors, power management devices, analog and mixed-signal ICs, MEMS and sensors, and RF and wireless ICs. STMicroelectronics' products are used in a wide range of applications, including automotive, industrial, communications, and personal electronics.

One of the company's key areas of focus is on developing energy-efficient solutions that help reduce the environmental impact of electronic devices. For example, STMicroelectronics has developed a range of power management devices that are designed to reduce energy consumption in a variety of applications, from smartphones to home appliances. The company also offers a range of sensors and MEMS (micro-electromechanical systems) that enable the development of more efficient and reliable devices.

STMicroelectronics' products are targeted at a diverse range of customers, from large multinational corporations to small start-ups. The company's customers span a wide range of industries, including automotive, industrial, consumer electronics, and telecommunications. By offering a broad range of products and services, STMicroelectronics is able to meet the needs of a diverse customer base and stay ahead of competitors in the semiconductor industry.

1.5 Conclusion

In conclusion, STMicroelectronics is a leading global semiconductor company that has established itself as a key player in the industry. With a focus on innovation, quality, and customer service, STMicroelectronics has developed a broad portfolio of products and services that are used in a wide range of applications, from automotive to consumer electronics.

Chapter 2

Tools & Concepts

2.1 Introduction

The purpose of this project is to evaluate the performance of the CMSIS RTOS v2 API using two different real-time operating systems, ThreadX and freeRTOS, on the STM32 microcontroller. To achieve this, we utilized the Thread Metric benchmark tool, which allows us to measure the responsiveness, efficiency, and reliability of the RTOS implementation. By comparing the performance of ThreadX and freeRTOS on the STM32 using the CMSIS RTOS v2 API, we aim to identify any strengths or weaknesses of each system and determine which one is better suited for our specific application. This project can provide valuable insights into the selection of an RTOS for embedded systems and help improve the overall performance of the system.

2.2 Key Concepts

2.2.1 Real Time

In computing, real-time refers to a system or process that can respond to an event or input within a predetermined timeframe or deadline. This is particularly important in applications where timing is critical, such as control systems, telecommunications, and multimedia applications. Real-time systems often require dedicated hardware and software designed to meet strict timing requirements and ensure reliable and predictable performance.[13]

2.2.2 RTOS

RTOS (Real-Time Operating System) is an operating system specifically designed for use in real-time embedded systems, where the timely and accurate response to external events is crucial. An RTOS is built to handle tasks with strict deadlines, where a failure to complete a task within a specific timeframe could result in a catastrophic system failure. RTOSs include a real-time kernel that manages tasks, interrupts, and communication between tasks, as well as providing mechanisms for synchronization, memory management, and resource allocation.[13]

2.2.3 Thread

A thread is a single sequence of instructions that can be executed independently by a program. Threads are lightweight processes that can be created within a single process to perform tasks concurrently. Threads have their own set of registers, stack, and program counter, but they share the same memory space as other threads in

the process. This allows threads to communicate and synchronize with each other efficiently, which is essential for multi-tasking and multi-user systems. Threads are commonly used in modern operating systems, programming languages, and application frameworks to improve performance and scalability.[4]

2.2.4 Semaphore

In computer science, a semaphore is a synchronization mechanism used to control access to a shared resource in a concurrent system. It is typically a simple integer variable that can be accessed and modified by atomic operations, such as incrementing and decrementing.

A semaphore is initialized with a value that represents the number of available resources. When a process or thread wants to access the resource, it first acquires the semaphore by decrementing its value. If the value becomes negative, the process or thread is blocked until the semaphore value becomes positive again, indicating that a resource is available. When a process or thread is done using the resource, it releases the semaphore by incrementing its value, which allows other processes or threads to acquire the resource.

There are two types of semaphores: binary and counting. A binary semaphore can have only two values, typically 0 and 1, and is used to protect a single shared resource. A counting semaphore can have multiple values, typically greater than or equal to 0, and is used to control access to a finite number of resources, such as a pool of database connections.[5]

2.3 Hardware

2.3.1 ARM Cortex-M Series

The ARM Cortex-M series is a family of microcontroller architectures developed by ARM Holdings, a British multinational semiconductor and software design company. The Cortex-M series is specifically designed for use in embedded systems and real-time applications, and is characterized by its low power consumption, small footprint, and high performance.[6]

The Cortex-M series includes a range of microcontroller architectures, each optimized for different applications and use cases. For example, the Cortex-M0 is designed for low-power applications, while the Cortex-M7 is optimized for high-performance applications. The Cortex-M3 and Cortex-M4 are widely used in a range of applications, from automotive and industrial control systems to home automation and consumer electronics.

One of the key features of the Cortex-M series is its efficient instruction set architecture, which enables high performance with low power consumption. The Cortex-M series also includes a range of integrated peripherals, including timers, ADCs, DACs, and communication interfaces such as UART, SPI, and I2C, which make it easier to interface with external devices and sensors.[7]

Another important feature of the Cortex-M series is its support for the ARM Cortex Microcontroller Software Interface Standard (CMSIS), which provides a standardized interface for software development and code reuse across different microcontroller platforms.

Overall, the ARM Cortex-M series is a popular choice for embedded systems and real-time applications, offering a combination of low power consumption, high performance, and a range of integrated peripherals and software tools. Its flexibility and scalability also make it well-suited to a wide range of applications, from low-power wearable devices to high-performance industrial control systems.

2.3.2 STM32H723ZG_Nucleo

The STM32H723ZG_Nucleo board is a development board based on the STM32H723ZG microcontroller, which is part of the STM32H7 series of high-performance microcontrollers from STMicroelectronics. The board is designed to provide developers with a platform for prototyping and testing applications that require high-performance processing, advanced connectivity, and rich multimedia capabilities.

The STM32H723ZG_Nucleo board features an STM32H723ZG microcontroller, which is based on the Arm Cortex-M7 core and includes 1 MB of Flash memory, 512 KB of RAM, and a range of integrated peripherals, including USB, Ethernet, CAN, SPI, and I2C interfaces. The board also includes an external quad-SPI flash memory for additional storage, and an ST-LINK/V2-1 debugger and programmer for debugging and programming the microcontroller.[20]

The STM32H723ZG_Nucleo board is designed to be compatible with a wide range of software development tools, including the STM32CubeIDE integrated development environment (IDE) and the STM32CubeMX software configuration tool. The board also supports a range of operating systems, including FreeRTOS,ThreadX and Linux.

One of the key features of the STM32H723ZG_Nucleo board is its support for a range of multimedia applications. The microcontroller includes a graphics

accelerator that supports advanced graphics rendering, and the board includes a TFT-LCD display connector and a camera interface for developing multimedia applications.

The board also includes a range of connectivity options, including Ethernet, USB, and CAN interfaces, which make it well-suited to applications that require high-speed communication with other devices or networks. The STM32H723ZG_Nucleo board also supports wireless connectivity options, such as Wi-Fi and Bluetooth, through external modules that can be connected to the board.

Figure 2.1: a front and rear view of the STM32H723ZG_Nucleo

2.3.3 STM32F429I_Discovery

The STM32F429I_Discovery board is a development board based on the STM32F429ZI microcontroller from STMicroelectronics, a leading semiconductor company. It is designed to provide a low-cost platform for developing and prototyping embedded systems and applications.

The STM32F429I_Discovery board features a wide range of integrated peripherals and interfaces, including a high-resolution 2.4-inch TFT LCD display with touch panel, a stereo audio codec, a 3-axis accelerometer, a 3-axis gyroscope, and a digital microphone. It also includes USB OTG, Ethernet, and CAN communication interfaces, as well as a range of GPIO pins and headers for connecting external

devices and sensors.[18]

The STM32F429ZI microcontroller at the heart of the board is based on the ARM Cortex-M4 core, and features a 180 MHz clock speed, 2 MB of flash memory, and 256 KB of SRAM. It also includes a range of hardware acceleration features, including a hardware floating-point unit (FPU), a DMA controller, and a CRC calculation unit, which make it well-suited to high-performance real-time applications.

The board is compatible with a range of development tools and software, including the STM32CubeMX code generation tool, the STM32CubeIDE integrated development environment (IDE), and the STM32CubeF4 firmware package, which includes a range of software libraries and examples for getting started with the board.

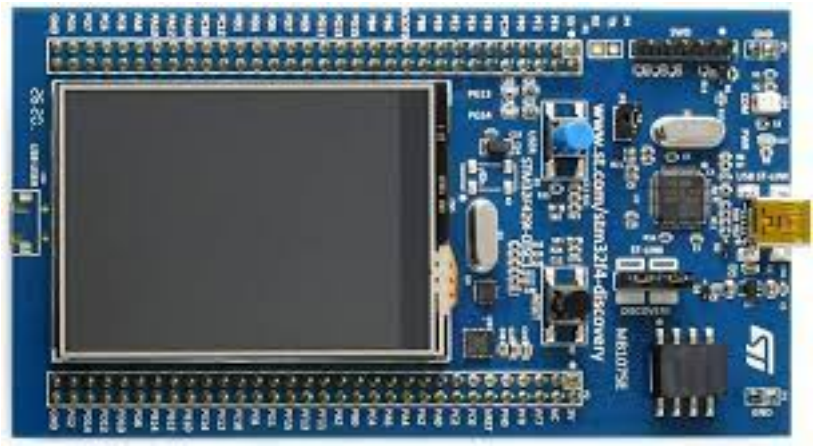

Figure 2.2: a front view of the STM32F429I_Discovery

2.4 Software

2.4.1 STM32cubeMX

STMicroelectronics STM32CubeMX is a graphical tool that allows developers to generate initialization code for STM32 microcontrollers. It offers an intuitive interface that helps users configure the hardware settings of the microcontroller and generate code based on the selected configuration. STM32CubeMX supports a wide range of STM32 microcontrollers, and provides users with various configuration options for different peripherals such as timers, ADCs, and communication interfaces..[19]

With STM32CubeMX, developers can easily configure pin assignments, clock frequencies, and other system parameters. The tool generates a project file that can be imported into an integrated development environment (IDE) such as Keil, IAR, or Eclipse, which enables developers to start writing application code without having to worry about the low-level initialization code.

STM32CubeMX also provides a powerful code generation engine that generates code for the selected configuration in various programming languages such as C, C++, and Assembler. The generated code is highly optimized and follows a modular architecture, which allows for easy customization and reuse.

Overall, STM32CubeMX is a powerful and user-friendly tool that streamlines the process of configuring and initializing STM32 microcontrollers, and provides developers with a solid foundation to build their applications upon.

Figure 2.3: STM32cubeMX logo

2.4.2 STM32cubeIDE

STMicroelectronics STM32CubeIDE is an integrated development environment (IDE) that is designed to simplify the development of applications for STM32 microcontrollers. It is based on the Eclipse platform and includes all the necessary tools for developing, debugging, and deploying STM32 applications.[19]

STM32CubeIDE features an intuitive graphical user interface that provides users with access to a wide range of tools, including code editors, debuggers, and project management tools. The IDE also includes a powerful code generation engine that generates optimized code based on the selected configuration.

The IDE supports a wide range of STM32 microcontrollers and provides developers with a comprehensive set of drivers, middleware, and example projects to help them get started quickly. It also includes a real-time operating system

(RTOS) that enables developers to create complex applications with multiple tasks and threads.

STM32CubeIDE is fully integrated with the STM32CubeMX configuration tool, which enables developers to configure their STM32 microcontrollers and generate initialization code that can be imported directly into the IDE. This allows developers to focus on writing application code rather than worrying about low-level initialization code.

Overall, STM32CubeIDE is a powerful and user-friendly IDE that streamlines the development process for STM32 microcontrollers and provides developers with all the necessary tools to create high-quality applications with ease.

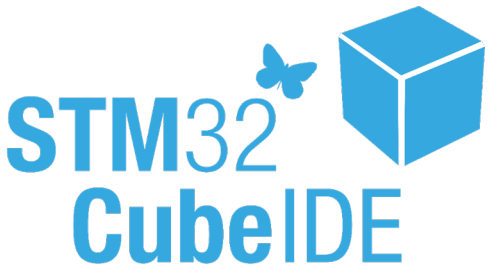

Figure 2.4: STM32cubeIDE logo

2.4.3 GitHub Desktop

GitHub Desktop is a graphical user interface (GUI) for GitHub, the popular web-based hosting service for software development projects. It provides an easy-to-use interface for users to interact with their repositories and collaborate with others on their projects.[10]
With GitHub Desktop, users can manage their repositories, create and review pull requests, and merge changes into their codebase with ease. The application provides a streamlined workflow that simplifies the process of contributing to open source projects or collaborating on private repositories.

One of the key benefits of GitHub Desktop is its integration with the GitHub platform. This integration enables users to easily clone repositories, create new branches, and push changes to their codebase, all from within the application. The application also includes a number of built-in tools for resolving merge conflicts and reviewing code changes.

GitHub Desktop is available for both Windows and Mac operating systems and is free to download and use. It has a simple and intuitive interface that makes it easy for developers of all skill levels to get started with GitHub and collaborate with others on their projects.

Figure 2.5: GitHub Desktop logo

2.5 Firmware

2.5.1 ThreadX

ThreadX is a real-time operating system (RTOS) developed by Express Logic. It is designed to be highly efficient and reliable, with a small memory footprint and fast context-switching times. ThreadX provides a preemptive, priority-based scheduling algorithm, which allows for deterministic and predictable system behavior. It also includes a comprehensive set of synchronization mechanisms, memory management features, and inter-process communication facilities, making it suitable for a wide range of embedded applications, such as consumer electronics, medical devices, and aerospace and defense systems. ThreadX has been widely adopted in the embedded industry due to its ease of use, scalability, and fast time-to-market.[15]

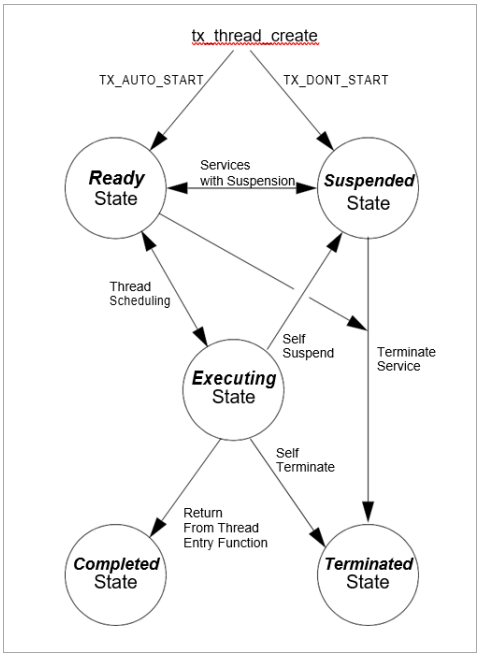

Figure 2.6: ThreadX thread managment system

2.5.2 FreeRtos

FreeRTOS is a real-time operating system (RTOS) designed for use in embedded systems. It is a popular open-source software platform that provides a small, efficient kernel with pre-emptive multitasking and time-slicing capabilities. FreeRTOS supports a variety of architectures and microcontrollers, making it a versatile and widely used solution in the embedded industry. It includes a range of features, including task prioritization, inter-task communication, synchronization mechanisms, and memory management, which make it suitable for a wide range of applications, such as industrial control systems, automotive, and aerospace systems. FreeRTOS is widely known for its portability, flexibility, and reliability, and it has a large community of developers who actively contribute to its development and support.[8]

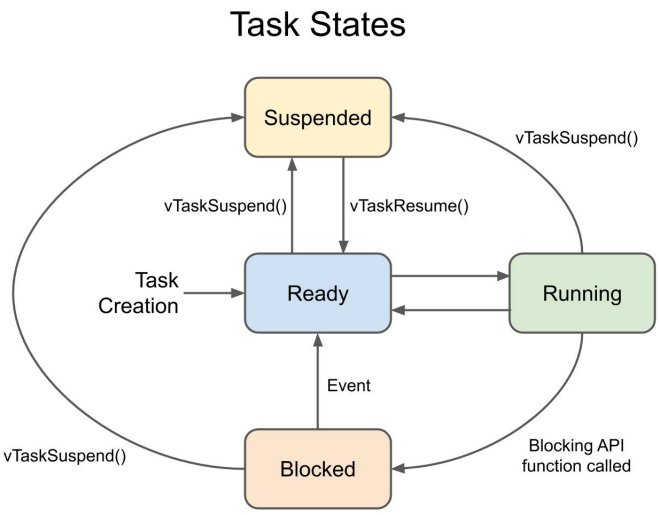

Figure 2.7: FreeRTOS thread managment system

2.5.3 CMSIS API

The Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized interface for microcontrollers that allows for greater code reusability and portability across different microcontroller platforms. The CMSIS API is a collection of functions and data structures that provide a consistent and predictable interface to the hardware peripherals of a microcontroller, allowing developers to write code that can be easily ported between different devices. The CMSIS API is divided into several modules, each of which corresponds to a different aspect of the microcontroller hardware, such as the core processor, interrupt controller, and peripheral interfaces.[12]

The CMSIS API provides a high-level abstraction of the hardware, shielding developers from the complexities of low-level hardware programming, while still providing access to the full capabilities of the microcontroller. The API is designed to be efficient and lightweight, minimizing the overhead of function calls and data transfers, and is optimized for use with the ARM Cortex-M series of microcontrollers (such as the STM32 Boards), although it can be used with other microcontroller architectures as well.

One of the key benefits of the CMSIS API is its ability to simplify the development process by providing a consistent interface across different microcontroller platforms, reducing the need for developers to learn and adapt to new hardware

interfaces with each new project. This also makes it easier to reuse code across projects, as well as to share code with other developers.

Overall, the CMSIS API is an important tool for developers working with microcontrollers, allowing them to write efficient, portable code that can be easily adapted to a wide range of hardware platforms.

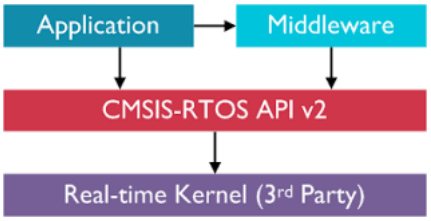

Figure 2.8: CMSIS API interface

2.6 Conclusion

In conclusion, this chapter has provided a comprehensive overview of the key concepts and tools used in our project. We have discussed the different software and development environments of the STM32, including the STM32cubeMX and the STM32cubeIDE.

We introduced also the STM32H723ZG_Nucleo and STM32F429I_Discovery boards which we used for deployment.

Additionally, we have introduced the various real-time operating systems (RTOS) that we will be working with, as well as the CMSIS API which is the central focus of our project.

This chapter has laid the foundation for the following chapters, where we will dive deeper into these concepts and explore their implementation in greater detail. By understanding these fundamental concepts and tools, we are well-equipped to design and develop a robust and efficient embedded system using the STM32 microcontroller platform.

Chapter 3

Project Initialization & Implementations

3.1 Intoduction

As we dive in the specifics of our project, we need to use the provided CMSIS RTOS v2 API implementation on top of ThreadX API by STmicroelectronics to import the CMSIS API to our project and develop the necessary functionalities to adapted to the benchmark tests and compare the test results to those of the native implementation of ThreadX.
As CMSIS is originally provided by the STM32Cube ecosystem on top of the FreeRTOS, we are also tasked to run the thread_metric and compare the performance of the CMSIS API using FreeRTOS and ThreadX.
In order to get this results we needed to use the STM32CubeIDE to create multiple projects for STM32H723-nucleo board and the STM32F429I-Discovery board running the thread_metrics using native ThreadX API as well as over the CMSIS RTOS 2 API (using the FreeRTOS and ThreadX) which are pushed into a github repository with the necessary documentation to explain how to run the tests and how to configure the tests.

3.2 Project creation and boards configuration

Before we could begin the development of our code, we need to create a set of projects using the STM32CubeMX each project is going to be specified for a test and intialized a specified RTOS.
This process is quite simple we just need to follow some sample steps:

1. Using STM32CubeMX create a new project and select the specific board to work on.

Figure 3.1: Board Selection on the STM32CubeMX

2. Configure the USART port tx/rx using the graphical interface so that the serial print is enabled to be able to print the test results on the console.

   Figure 3.2: Configuration of the USART ports using STM32CubeMX

3. Configure the RTOS drivers and settings using the graphical interface : for the CMSIS API it is naively offered by the STM32 environment on top of the FreeRTOS only so for the CMSIS API on top of ThreadX we actually intialize the project with ThreadX native and then we manually add the CMSIS drivers and files manually to the project that we cloned from the provided GitHub repository by STMicroelectronics.

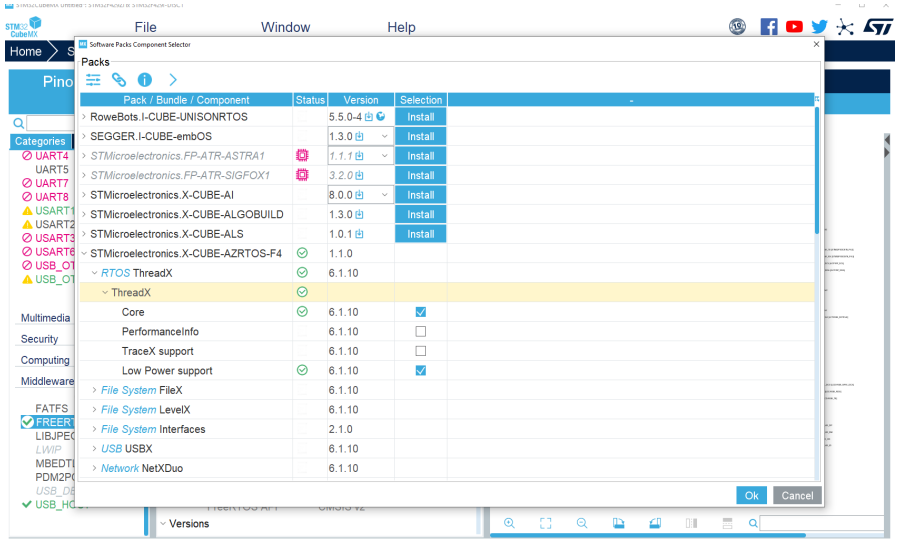

Figure 3.3: Selection of ThreadX RTOS packages

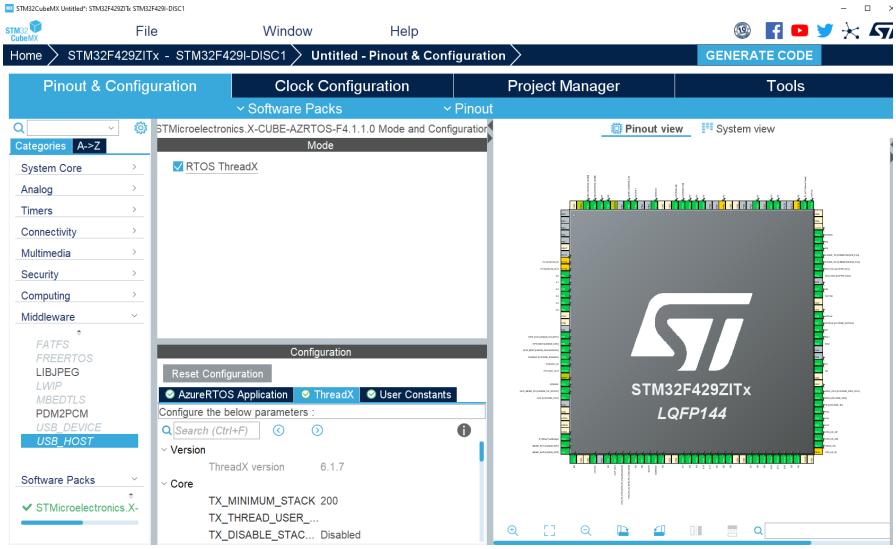

Figure 3.4: Configuration of ThreadX RTOS using STM32CubeMX

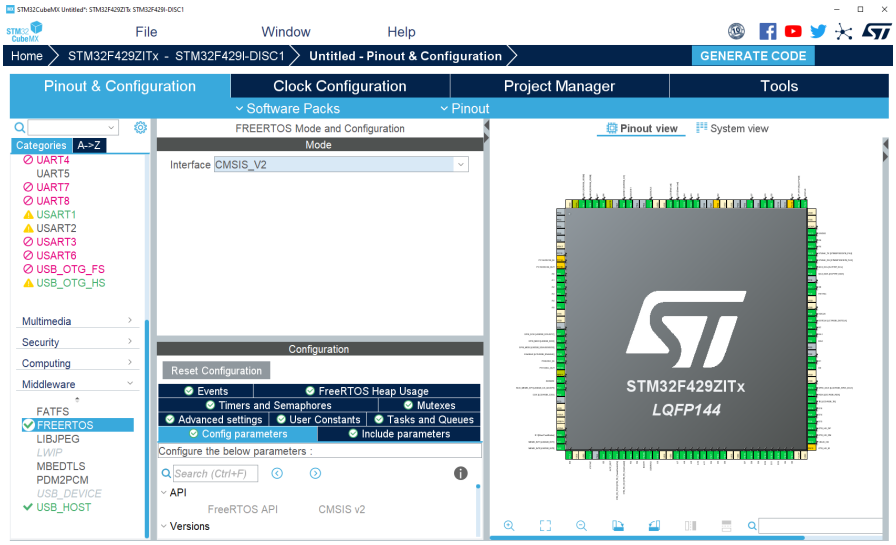

Figure 3.5: Configuration of CMSIS/FreeRTOS using STM32CubeMX

4. Implement the porting layer of the specified RTOS and run the tests.

3.3 Implementation of CMSIS API with ThreadX

While STmicroelectronics still does not offer CMSIS RTOS v2 API implementation on top of ThreadX API in their ecosystem, They opt with this project to offer this service to there clients. To achieve this goal and to assure the correct implementation they have provided us with an initial wrapper to the CMSIS API using the ThreadX API.
The CMSIS-RTOS wrapper for ThreadX provides a standardized and portable interface to the ThreadX RTOS functions, simplifying the development of efficient and portable software for microcontrollers. This file is the implementation of functions to wrap CMSIS RTOS2 onto AzureRTOS ThreadX based on API published by Arm Limited in cmsis-os2.h file. The implementation of these functions is inspired from an original work from Arm Limited to wrap CMSIS RTOS2 onto FreeRTOS.
The CMSIS-RTOS API defines two levels of abstraction :
See the appendix .1
The CMSIS-RTOS API provides a wrapper for ThreadX, which implements the KAL and RTOS API layers for ThreadX. The CMSIS-RTOS wrapper for ThreadX provides the following benefits :
See the appendix .2

3.4 Porting Layer

The porting layer is needed provides the necessary interfaces to integrate the Thread-Metric Component with a particular RTOS to be able to perform the benchmarking.

During this project, we have developed two versions: one specified for the ThreadX RTOS native and the other is written for the CMSIS-RTOS so that we can compare the result of the benchmarking later. It is worth to mention that even we have used both the CMSIS_RTOS API over FreeRTOS and Threadx the porting layer is the same due to using the build in native functions of the CMSIS_RTOS API.

The porting layer defines constants and data structures that are specified depending on the used RTOS(either ThreadX or CMSIS), such as the maximum number of threads, queues, semaphores, and memory pools that can be created. It also defines the size of the thread stack, the size of the queue, and the size of the memory pool. The porting layer provides implementations for a set of functions that makes use of the specific RTOS APIs to create and manage threads, queues, and semaphores. The porting layer also defines data structures and memory areas that are used by the Thread-Metric Component to keep track of threads, queues, semaphores, and memory pools wich provides the necessary interface to integrate the Thread-Metric Component.

3.4.1 tm-initialize

This function called from main performs basic RTOS initialization,calls the test initialization function, and then starts the RTOS function.It takes as a parameter a void pointer named "test-initialization-function" which is a pointer to a function that takes no arguments and returns nothing. The function itself returns void.Then, the function that "test-initialization-function" points to is executed. This function initializes the tests that will be performed in the RTOS.

```
void  tm_initialize(void (*test_initialization_function)(void))
{

    /* Save the test initialization function.  */
    tm_initialization_function =  test_initialization_function;

    /* Call the previously defined initialization function.  */
    (tm_initialization_function)();
}
```

Figure 3.6: tm-intialize source code

3.4.2 tm-thread-create

This function creates a new thread in an RTOS system. It takes three arguments: thread-id, priority, and a function pointer named "entry-function". Depending on the used RTOS, the function core varies:

- For the ThreadX implementation we used the predefined function **"tx-thread-create"** which takes for parameters:
  See the appendix .3
- For the CMSIS implementation we used the predefined function **"osThread-New"** which takes for parameters:
  See the appendix .4

  note that we need to suspend the thread after creation in the CMSIS API implementation over ThreadX as the thread on creation will be active and it will cause problems during the tests.

```
int  tm_thread_create(int thread_id, int priority, void (*entry_function)(void))
{

UINT    status;

    /* Remember the actual thread entry.  */
    tm_thread_entry_functions[thread_id] =  (void *) entry_function;

    /* Create the thread under ThreadX.  */
    status =  tx_thread_create(&tm_thread_array[thread_id], "Thread-Metric test", tm_thread_entry, (ULONG) thread_id,
                  &tm_thread_stack_area[thread_id*TM_THREADX_THREAD_STACK_SIZE], TM_THREADX_THREAD_STACK_SIZE,
                  (UINT) priority, (UINT) priority, TX_NO_TIME_SLICE, TX_DONT_START);

    /* Determine if the thread create was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.7: ThreadX Native

```
int tm_thread_create(int thread_id, int priority, void (*entry_function)(void))
{
    // Remember the actual thread entry.
       // tm_thread_entry_functions[thread_id] = entry_function;
        char str[100];
        sprintf(str, "%d",thread_id);

        // Create the thread under CMSIS-RTOS.
        osThreadAttr_t thread_attr = {
            .name = str,
            //.stack_mem = tm_thread_stack_area[thread_id],
            .stack_size = 3*1024,
            .priority = priority
        };
        tm_thread_array[thread_id] = osThreadNew((osThreadFunc_t)entry_function, NULL, &thread_attr);

        // Determine if the thread create was successful.
        if (tm_thread_array[thread_id] != NULL)
        {
            osThreadSuspend(tm_thread_array[thread_id]);

            return TM_SUCCESS;
        }

        else
        {

             return TM_ERROR;
        }
}
```

Figure 3.8: Cmsis

Figure 3.9: tm-thread-create source code

3.4.3 tm-thread-resume

This function resumes the specified thread. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.

- For the ThreadX implementation we used the predefined function **"tx-threadresume"**which takes for parameters a TX-THREAD pointer called **"threadptr"** that refers to the designated thread to resume.
- For the CMSIS implementation we used the predefined function **"osThread-Resume"** which takes for parameters an osThreadId-t variable that refers to the designated thread to resume.

```
int  tm_thread_resume(int thread_id)
{

UINT    status;


    /* Attempt to resume the thread.  */
    status =  tx_thread_resume(&tm_thread_array[thread_id]);

    /* Determine if the thread resume was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.10: ThreadX Native

```
int tm_thread_resume(int thread_id)
{
    /* Resume the thread with the specified ID. */
    if (osThreadResume(tm_thread_array[thread_id]) == osOK) {
        /* Thread resume successful. */
        return TM_SUCCESS;

    }
    else
    {

    }
    /* Thread resume failed. */
    return TM_ERROR;
}
```

Figure 3.11: Cmsis

Figure 3.12: tm-thread-resume source code

3.4.4 tm-thread-suspend

This function resumes the specified thread. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.

- For the ThreadX implementation we used the predefined function **"tx-threadresume"**which takes for parameters a TX-THREAD pointer called **"threadptr"** that refers to the designated thread to resume.
- For the CMSIS implementation we used the predefined function **"osThread-Resume"** which takes for parameters an osThreadId-t variable that refers to the designated thread to resume.

```
int  tm_thread_suspend(int thread_id)
{

UINT    status;


    /* Attempt to suspend the thread.  */
    status =  tx_thread_suspend(&tm_thread_array[thread_id]);

    /* Determine if the thread suspend was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.13: ThreadX Native

```
int  tm_thread_suspend(int thread_id)
{

        if (osThreadSuspend(tm_thread_array[thread_id]) == osOK)
        {
            return TM_SUCCESS;
        }
        else
        {
            return TM_ERROR;
        }
}
```

Figure 3.14: Cmsis

Figure 3.15: tm-thread-suspend source code

3.4.5 tm-thread-relinquish

This function relinquishes to other ready threads at the same priority.

- For the ThreadX implementation we used the predefined function **"tx-thread-relinquish"**.
- For the CMSIS implementation we used the predefined function **"osThreadYield"** .

```
void tm_thread_relinquish(void)
{

    /* Relinquish to other threads at the same priority.  */
    tx_thread_relinquish();
}
```

Figure 3.16: ThreadX Native

```
void tm_thread_relinquish(void)
{
    osThreadYield();
}
```

Figure 3.17: Cmsis

Figure 3.18: tm-thread-relinquish source code

3.4.6 tm-thread-sleep

This function chage the status of the specified thread to sleep mode. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.

- For the ThreadX implementation we used the predefined function **"tx-thread-sleep"** which takes for parameters an integer variable that refers to the number of seconds the thread is going to sleep for.

- For the CMSIS implementation we used the predefined function **"osDelay"** which takes for parameters an integer variable that refers to the number of seconds the thread is going to sleep for. note that the ThreadX clock is still slower 10X of that of the FreeRTOS so we need to adjust the numbers in the porting layer.

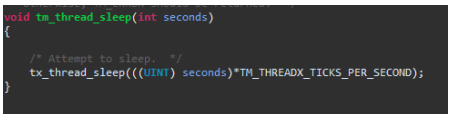

Figure 3.19: ThreadX Native

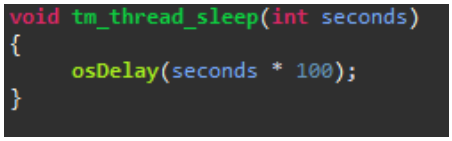

Figure 3.20: Cmsis/ThreadX

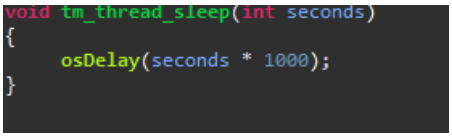

Figure 3.21: Cmsis/FreeRtos

Figure 3.22: tm-thread-sleep source code

3.4.7 tm-queue-create

This function creates a new queue in an RTOS system. It takes one argument: an integer variable called "queue-id".

- For the ThreadX implementation we used the predefined function **"tx-queue-create"** which takes for parameters:
  See the appendix .5

- For the CMSIS implementation we used the predefined function **"osMessageQueueNew"** which takes for parameters:
  See the appendix .6

```c
int  tm_queue_create(int queue_id)
{

UINT    status;


    /* Create the specified queue with 16-byte messages.  */
    status =  tx_queue_create(&tm_queue_array[queue_id], "Thread-Metric test", TX_4_ULONG,
                              &tm_queue_memory_area[queue_id*TM_THREADX_QUEUE_SIZE], TM_THREADX_QUEUE_SIZE);

    /* Determine if the queue create was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.23: ThreadX Native

```c
int  tm_queue_create(int queue_id)
{
    char str[100];
    sprintf(str, "%d",queue_id);
    osMessageQueueAttr_t queue_attr = {
            .name = str,            // Queue name is not used
            .attr_bits = 0U,        // No special attributes needed
            .cb_mem = NULL,         // Allocate memory from the heap
            .cb_size = 0U           // Use default size
        };

        osMessageQueueId_t queue = osMessageQueueNew(TM_CMSIS_QUEUE_SIZE, sizeof(unsigned long), &queue_attr);
        if (queue == NULL) {
            return TM_ERROR;
        }

        // Save the queue ID in the global variable for future use
        tm_queue_array[queue_id] = queue;

        return TM_SUCCESS;
}
```

Figure 3.24: Cmsis

Figure 3.25: tm-queue-create source code

3.4.8 tm-queue-send

This function sends a 16-byte message to the specified queue. If successful,the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned. It takes two arguments: an integer variable called "queue-id" and an unsigned long pointer that stores the message.

- For the ThreadX implementation we used the predefined function**"tx-queue-send"** which takes for parameters:
  See the appendix.7

- For the CMSIS implementation we used the predefined function **"osMessageQueuePut"** which takes for parameters:
  See the appendix .8

```c
int  tm_queue_send(int queue_id, unsigned long *message_ptr)
{

UINT    status;


    /* Send the 16-byte message to the specified queue.  */
    status =  tx_queue_send(&tm_queue_array[queue_id], message_ptr, TX_NO_WAIT);

    /* Determine if the queue send was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.26: ThreadX Native

```c
int  tm_queue_send(int queue_id, unsigned long *message_ptr)
{
    osMessageQueueId_t queue = tm_queue_array[queue_id];
        osStatus_t status = osMessageQueuePut(queue, message_ptr, 0U, 0U);
        if (status != osOK) {
            return TM_ERROR;
        }

        return TM_SUCCESS;
}
```

Figure 3.27: Cmsis

Figure 3.28: tm-queue-send source code

3.4.9 tm-queue-receive

This function receives a 16-byte message to the specified queue. If successful,the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.It takes two arguments: an integer variable called "queue-id" and an unsigned long pointer that stores the message.

- For the ThreadX implementation we used the predefined function**"tx-queue-recieve"** which takes for parameters:
  See the appendix .9
- For the CMSIS implementation we used the predefined function **"osMessageQueueGet"** which takes for parameters:
  See the appendix .10

```c
int  tm_queue_receive(int queue_id, unsigned long *message_ptr)
{

UINT    status;


    /* Receive a 16-byte message from the specified queue.  */
    status =  tx_queue_receive(&tm_queue_array[queue_id], message_ptr, TX_NO_WAIT);

    /* Determine if the queue receive was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.29: ThreadX Native

```c
int  tm_queue_receive(int queue_id, unsigned long *message_ptr)
{
    osMessageQueueId_t queue = tm_queue_array[queue_id];
       uint32_t msg_priority;
       osStatus_t status = osMessageQueueGet(queue, message_ptr, &msg_priority, osWaitForever);
       if (status != osOK) {
           return TM_ERROR;
       }

       return TM_SUCCESS;
}
```

Figure 3.30: Cmsis

Figure 3.31: tm-queue-receive source code

3.4.10 tm-semaphore-create

This function creates the specified semaphore. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned. It takes one argument: an integer variable called "semaphore-id".

- For the ThreadX implementation we used the predefined function **"tx-semaphore-create"** which takes for parameters:
  See the appendix.11
- For the CMSIS implementation we used the predefined function **"osSemaphore-New"** which takes for parameters:
  See the appendix .12

```c
int  tm_semaphore_create(int semaphore_id)
{

UINT    status;


    /*  Create semaphore.  */
    status =  tx_semaphore_create(&tm_semaphore_array[semaphore_id], "Thread-Metric test", 1);

    /* Determine if the semaphore create was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.32: ThreadX Native

```c
int  tm_semaphore_create(int semaphore_id)
{
    char str[100];
    sprintf(str, "%d",semaphore_id);
     osSemaphoreAttr_t semaphore_attr = {
            .name = str,           // Semaphore name is not used
            .attr_bits = 0U,       // No special attributes needed
            .cb_mem = NULL,        // Allocate memory from the heap
            .cb_size = 0U          // Use default size
        };

        osSemaphoreId_t semaphore = osSemaphoreNew(TM_CMSIS_MAX_SEMAPHORES, TM_CMSIS_MAX_SEMAPHORES, &semaphore_attr);
        if (semaphore == NULL) {
            return TM_ERROR;
        }

        // Save the semaphore ID in the global variable for future use
        tm_semaphore_array[semaphore_id] = semaphore;

        return TM_SUCCESS;
}
```

Figure 3.33: Cmsis

Figure 3.34: tm-semaphore-create source code

3.4.11 tm-semaphore-get

This function gets the specified semaphore. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned. It takes one argument: an integer variable called "semaphore-id".

- For the ThreadX implementation we used the predefined function **"tx-semaphore-get"** which takes for parameters:
  See the appendix .13
- For the CMSIS implementation we used the predefined function **"osSemaphore-Acquire"** which takes for parameters:
  See the appendix .14

```
int  tm_semaphore_get(int semaphore_id)
{

UINT    status;


    /*  Get the semaphore.  */
    status =  tx_semaphore_get(&tm_semaphore_array[semaphore_id], TX_NO_WAIT);

    /* Determine if the semaphore get was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.35: ThreadX Native

```
int  tm_semaphore_get(int semaphore_id)
{
    osSemaphoreId_t semaphore = tm_semaphore_array[semaphore_id];
        osStatus_t status = osSemaphoreAcquire(semaphore, osWaitForever);
        if (status != osOK) {
            return TM_ERROR;
        }

        return TM_SUCCESS;
}
```

Figure 3.36: Cmsis

Figure 3.37: tm-semaphore-get source code

3.4.12 tm-semaphore-put

This function puts the specified semaphore. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.It takes one argument: an integer variable called "semaphore-id".

- For the ThreadX implementation we used the predefined function **"tx-semaphore-put"** which takes for parameter a TX-SEMAPHORE pointer called **"semaphore-ptr"**: which is a pointer to a previously created counting semaphore.
- For the CMSIS implementation we used the predefined function **"osSemaphoreRelease"** which takes for parameter an osSemaphoreId-t variable called **"semaphore-id"**: refers to the handle of the semaphore to acquire.

```
int  tm_semaphore_put(int semaphore_id)
{

UINT    status;


    /*  Put the semaphore.  */
    status =  tx_semaphore_put(&tm_semaphore_array[semaphore_id]);

    /* Determine if the semaphore put was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.38: ThreadX Native

```
int  tm_semaphore_put(int semaphore_id)
{
    osSemaphoreId_t semaphore = tm_semaphore_array[semaphore_id];
        osStatus_t status = osSemaphoreRelease(semaphore);
        if (status != osOK) {
            return TM_ERROR;
        }

        return TM_SUCCESS;

}
```

Figure 3.39: Cmsis

Figure 3.40: tm-semaphore-put source code

3.4.13 tm-memory-pool-create

This function creates the specified memory pool that can support one or more allocations of 128 bytes.If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.It takes one argument: an integer variable called "pool-id".

- For the ThreadX implementation we used the predefined function **"tx-block-pool-create"** which takes for parameters:
  See the appendix .15
- For the CMSIS implementation we used the predefined function **"osMemoryPoolNew"** which takes for parameters:
  See the appendix.16

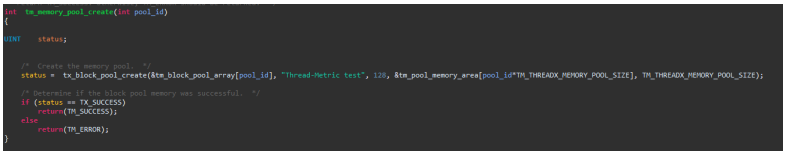

Figure 3.41: ThreadX Native

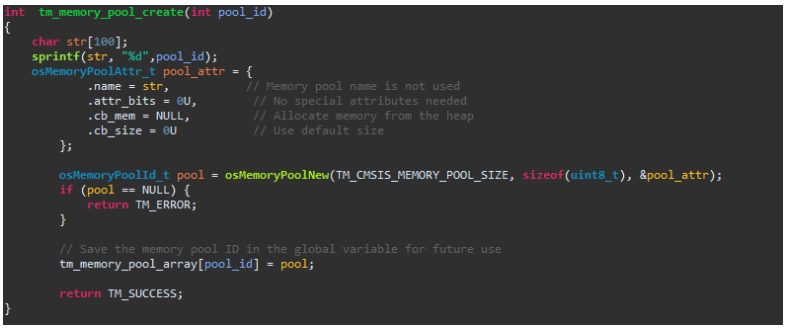

Figure 3.42: Cmsis

Figure 3.43: tm-memory-pool-create source code

3.4.14 tm-memory-pool-allocate

This function allocatess a 128 byte block from the specified memory pool.If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.It takes two arguments: an integer variable called "pool-id" and an unsigned char pointer called "memory-ptr".

- For the ThreadX implementation we used the predefined function **"tx-block-allocate"** which takes for parameters:
  See the appendix .17
- For the CMSIS implementation we used the predefined function **"osMemoryPoolAlloc"** which takes for parameters:
  See the appendix .18

```
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr)
{

UINT    status;


    /*  Allocate a 128-byte block from the specified memory pool.  */
    status =  tx_block_allocate(&tm_block_pool_array[pool_id], (void **) memory_ptr, TX_NO_WAIT);

    /* Determine if the block pool allocate was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.44: ThreadX Native

```
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr)
{
    osMemoryPoolId_t pool = tm_memory_pool_array[pool_id];
        void *memory = osMemoryPoolAlloc(pool, osWaitForever);
        if (memory == NULL) {
            return TM_ERROR;
        }

        // Return the memory block pointer to the caller
        *memory_ptr = (unsigned char *)memory;

        return TM_SUCCESS;
}
```

Figure 3.45: Cmsis

Figure 3.46: tm-memory-pool-allocate source code

3.4.15 tm-memory-pool-deallocate

This function releases a 128 byte block from the specified memory pool. If successful, the function should return TM-SUCCESS. Otherwise, TM-ERROR should be returned.It takes two arguments: an integer variable called "pool-id" and an unsigned char pointer called "memory-ptr".

- For the ThreadX implementation we used the predefined function **"tx-block-release"** which takes for parameter a void pointer called **"block-ptr"**: which is a pointer to a pointer to the previously allocated memory block.
- For the CMSIS implementation we used the predefined function **"osMemoryPoolFree"** which takes for parameters:
  See the appendix.19

```
int  tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr)
{

UINT    status;


    /*  Release the 128-byte block back to the specified memory pool.  */
    status =  tx_block_release((void *) memory_ptr);

    /* Determine if the block pool release was successful.  */
    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Figure 3.47: ThreadX Native

```
int  tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr)
{
        osMemoryPoolId_t pool = tm_memory_pool_array[pool_id];
        osStatus_t status = osMemoryPoolFree(pool, memory_ptr);
        if (status != osOK) {
            return TM_ERROR;
        }

        return TM_SUCCESS;
}
```

Figure 3.48: Cmsis

Figure 3.49: tm-memory-pool-deallocate source code

3.5 Project Deployment

After finishing the development of our project and to share our code with STMicroelectronics for an easy supervision, we have created a GitHub repository where we have deployed succesfully our project using GitHub Desktop (mentioned earlier in chapter 2).

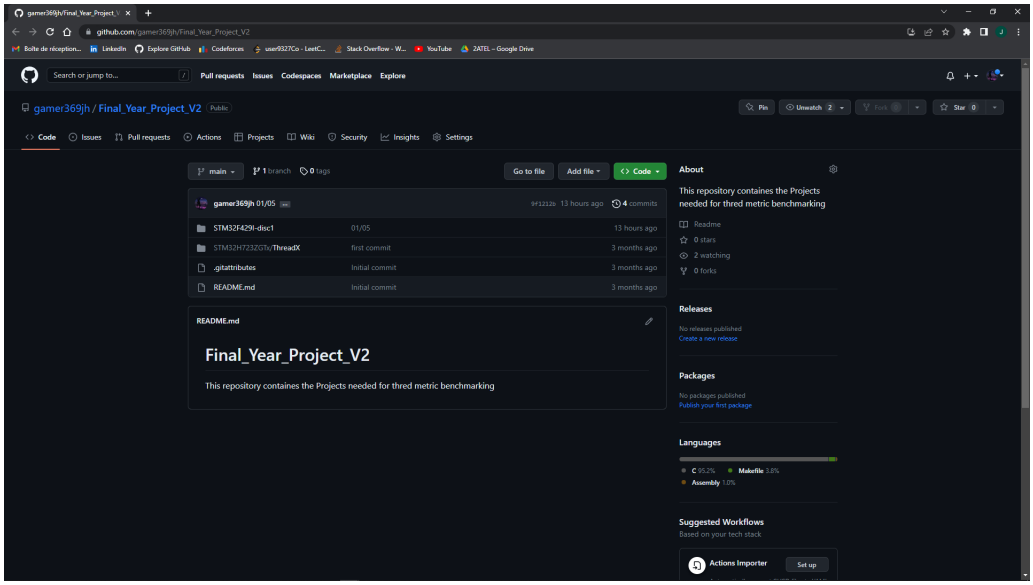

Figure 3.50: Project architecture on GitHub

The choice of using GitHub was taken for several reasons we mention from them.

- Backups and Version Control: GitHub provides a reliable backup for your code and offers version control, which allows you to keep track of changes made to your code over time. This feature is particularly useful when working on large-scale projects with multiple contributors.
- Collaboration: GitHub makes it easy for developers to collaborate with each other on a project. Developers can work together on the same codebase and easily share code, track issues, and discuss changes, all in one place. This enables a faster and more efficient development process.
- Continuous Integration and Deployment: GitHub can be integrated with various continuous integration and deployment (CI/CD) tools, such as Jenkins and Travis CI. This enables developers to automate the testing and deployment process, reducing the risk of errors and speeding up the release process.

Figure 3.51: QR code link to the GitHub repository full project. [11]

3.6 Conclusion

In conclusion, this project has offered us the opportunity to work in depth with the STM32 technologies and to interact with the provided developing environment as well as the chance to look into the different real-time operating systems and to have a practical touch in manipulating the different functionalities offered by them and grasp the basics and the low-level architecture of such technologies to be able to apply this knowledge on this project and expended on others.

Chapter 4

Benchmark & Test results

4.1 Intoduction

As one of the big leaders in the IOT and embedded system industry, STmicroelectronics has adapted the Microsoft Azure RTOS embedded suites, in particular the ThreadX RTOS.
ThreadX comes with a performance benchmark called thread_metrics. In this project our job is to dive deep into the ThreadX RTOS and learn about its APIs and functionality and test its performance using the thread_metrics then collecting the results.
All data then is collected and presented in an Excel document comparing the benchmark results (Native vs CMSIS RTOS2) in order to be shared by ST with internal and external customers.
In the end , we present the benchmarking results so that it helps in the decision to adapt CMSIS RTOSv2 API on top of ThreadX RTOS.

4.2 Test presentation

Thread-Metric benchmarking is an important process for ensuring that thread-metric components meet the required standards and are fit for their intended use. It involves measuring and evaluating the performance and quality of the components against established standards and best practices, identifying areas for improvement, implementing changes, and monitoring and evaluating performance. The benchmarking includes several tests that we are going to explore them and collect the results to ensure that thread-metric components meet the required specifications and are fit for their intended use.

4.2.1 Basic processing test

The thread appendix.20
tm-basic-processing-thread-0-entry():
Clear the tm-basic-processing-array loopthrough the array and in each iteration you add the previous contents with the content of the tm-basic-processing-entry
tm-basic_processing-thread-report():
Preempt the previous thread(as it has higher priority) run the test for a declared tm-test-duration (30sec defined in tm-api.h) if the test run successfully it prints the relative time and the time period total else it prints an error message

4.2.2 Cooperative scheduling test

The result provides information about the performance of the cooperative scheduling test. it shows the number of times each of the five cooperative threads ran during a test period of TM_TEST_DURATION (defined in the code). Compare different scheduling algorithms and determine which one is best suited for an application. It creates 5 threads and a reporting thread, and tests their cooperative behavior by having them relinquish the CPU to each other after each increment of their respective counters. The reporting thread prints the test results, including the average time per thread for each iteration of the test.

At a high level, the test initializes the threads and reporting thread, then enters an infinite loop in each of the 5 threads, where each thread relinquishes the CPU after incriminating its counter. The reporting thread sleeps for a specified duration, then calculates and prints the test results.

4.2.3 Interrupt preemption processing test

The test creates two threads, one with a higher priority than the other. The higher-priority thread generates an interrupt, which is handled by an interrupt service routine (ISR) that resumes the lower-priority thread. The ISR saves the full context of the interrupted thread during the preemption processing. Both threads run in a loop, incriminating their respective counters and suspending themselves. The test runs for a defined duration, and then the results are printed to the console.

The purpose of this test is to measure the performance of the interrupt handling mechanism of an operating system. It does this by measuring the amount of time it takes to handle an interrupt, including the time required to save and restore the context of the interrupted thread. The test can be used to evaluate the real-time performance of an operating system, as well as its ability to handle multiple tasks concurrently.

4.2.4 Interrupt processing test

The main function initializes the test by calling tm_initialize function and passing the address of the tm_interrupt_processing_initialize function, which creates a thread that generates an interrupt at priority 10, a semaphore that will be posted from the interrupt handler, and the reporting thread that will print out the test results.

The tm_interrupt_thread_0_entry function generates an interrupt and increments the tm_interrupt_thread_ 0_counter counter. The tm_interrupt_handler function increments the tm_interrupt_handler_counter counter and puts the semaphore from the interrupt handler.

The tm_interrupt_thread_report function calculates the total of all the counters, calculates the average of all the counters, and checks if the tm_interrupt_thread_0_counter and tm_interrupt_handler_counter counter values are within the range of the average ±1. If the test fails, it prints an error message.

4.2.5 Memory allocation test

The thread appendix .20
m_memory_allocation_thread_0_entry():

1. allocate memory from the pool: allocates a fixed-size memory block from the specified memory pool. The actual size of the memory block is determined during memory pool creation.

2. release the memory back to the pool: releases a previously allocated block back to its associated memory pool. If there are one or more threads suspended waiting for memory blocks from this pool, the first thread suspended is given this memory block and resumed.

3. increment the number of memory allocations sent and received.

tm_memory_allocation_thread_report(): preempt the previous thread(as it has higher priority) run the test for a declared tm-test-duation (30sec defined in tm-api.h)
if the test run successfully it prints the relative time and the time period total else it prints an error message

4.2.6 Message processing

The thread appendix .20

tm_message_processing_thread_0_entry():

1. initialize the source message as folow
   (a) tm_message_sent[0] = 0x11112222;
   (b) tm_message_sent[1] = 0x33334444;
   (c) tm_message_sent[2] = 0x55556666;
   (d) tm_message_sent[3] = 0x77778888;

2. send a message to the queue: sends a message to the specified message queue. The sent message is copied to the queue from the memory area specified by the source pointer.

3. receive a message from the queue: retrieves a message from the specified message queue. The retrieved message is copied from the queue into the memory area specified by the destination pointer. That message is then removed from the queue.

4. increment the last word of the 16-byte message.

5. increment the number of messages sent and received.

tm_message_processing_thread_report(): preemept the previous thread(as it has higher priority) run the test for a declared tm-test-duation (30sec defined in tm-api.h)
if the test run successfully it prints the relative time and the time period total else it prints an error message

4.2.7 Preemptive scheduling test

This is a preemptive scheduling test implemented in C using the Thread-Metric Component library. The test creates five threads with different priorities and tests the preemptive behavior of the scheduler.

The five threads are created with the following priorities: thread 0 at priority 10, thread 1 at priority 9, thread 2 at priority 8, thread 3 at priority 7, and thread 4 at priority 6. Thread 0 is initially resumed, and it resumes thread 1, which resumes thread 2, and so on until all threads have executed and self-suspended.

Each thread increments its respective counter and then self-suspends, allowing the next thread to execute. The reporting thread is created at priority 2 and prints the counter values of each thread, demonstrating the preemptive behavior of the scheduler. You can use this implementation to test the performance of the preemptive scheduling mechanism in ThreadX

4.2.8 Synchronization processing test

This is a C language source code file that defines a test for measuring the performance of synchronization processing in a threaded environment. The test uses a semaphore to synchronize the execution of threads.
The tm_synchronization_processing_initialize function creates a thread, a semaphore, and a reporting thread. The thread waits on the semaphore by calling tm_semaphore_get, then releases the semaphore by calling tm_semaphore_put. The number of successful semaphore get/put operations is counted and stored in tm_synchronization_processing_counter. The reporting thread periodically wakes up, calculates the number of semaphore get/put operations performed in the previous time period, and reports the result to the console.

The tm_main function initializes the test by calling tm_initialize and passing in a function pointer to tm_synchronization_processing_initialize.

4.3 Test results and observed behavior

4.3.1 Basic processing test

Test exemple running on ThreadX Native :

```
                              ***This test is running on the STM32H723ZG Nucleo with ThreadX_RTOS ***
this is tm_main
this is tm_basic_processing_initialize
tm_basic_processing_thread_report
tm_basic_processing_thread_0_entry
**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 30
Time Period Total:  153739

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 60
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 90
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 120
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 150
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 180
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 210
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 240
Time Period Total:  153725

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 270
Time Period Total:  153727

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 300
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 330
Time Period Total:  153726

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 360
Time Period Total:  153725

**** Thread-Metric Basic Single Thread Processing Test **** Relative Time: 390
Time Period Total:  153727
```

Figure 4.1: ThreadX Native

Based on the code provided, it seems that the Basic Processing Test is designed to measure the processing capabilities of a system by performing a series of calculations on a test array.
The test was executed on three different RTOS platforms: ThreadX Native, CMSIS/ThreadX, and CMSIS/FreeRTOS. The results obtained were 153,729; 748,693; and 715,457, respectively.
It is important to note that the results are highly dependent on the hardware and system configuration being used, as well as the version and implementation of the RTOS. Therefore, it is difficult to make a direct comparison between the results obtained on different platforms.
That being said, based on the results obtained, it appears that CMSIS/ThreadX

has the highest processing capabilities, followed by CMSIS/FreeRTOS, and finally ThreadX Native. However, it is important to note that these results should be taken with a grain of salt and may not necessarily reflect the overall performance of the RTOS platforms in all scenarios.

4.3.2 Cooperative scheduling test

Test exemple running on ThreadX Native :

```
**** Thread-Metric Cooperative Scheduling Test **** Relative Time: 30
tm_cooperative_thread_0_counter: 12075110
tm_cooperative_thread_1_counter: 12075110
tm_cooperative_thread_2_counter: 12075109
tm_cooperative_thread_3_counter: 12075109
tm_cooperative_thread_4_counter: 12075109
Time Period Total:  60375547

**** Thread-Metric Cooperative Scheduling Test **** Relative Time: 60
tm_cooperative_thread_0_counter: 24150917
tm_cooperative_thread_1_counter: 24150917
tm_cooperative_thread_2_counter: 24150917
tm_cooperative_thread_3_counter: 24150916
tm_cooperative_thread_4_counter: 24150916
Time Period Total:  60379036

**** Thread-Metric Cooperative Scheduling Test **** Relative Time: 90
tm_cooperative_thread_0_counter: 36226689
tm_cooperative_thread_1_counter: 36226689
tm_cooperative_thread_2_counter: 36226689
tm_cooperative_thread_3_counter: 36226689
tm_cooperative_thread_4_counter: 36226688
Time Period Total:  60378861

**** Thread-Metric Cooperative Scheduling Test **** Relative Time: 120
tm_cooperative_thread_0_counter: 48302461
tm_cooperative_thread_1_counter: 48302461
tm_cooperative_thread_2_counter: 48302461
tm_cooperative_thread_3_counter: 48302461
tm_cooperative_thread_4_counter: 48302461
Time Period Total:  60378861

**** Thread-Metric Cooperative Scheduling Test **** Relative Time: 150
tm_cooperative_thread_0_counter: 60378198
tm_cooperative_thread_1_counter: 60378198
tm_cooperative_thread_2_counter: 60378198
tm_cooperative_thread_3_counter: 60378198
tm_cooperative_thread_4_counter: 60378197
Time Period Total:  60378684
```

Figure 4.2: ThreadX Native

The code provided is a cooperative scheduling test that consists of five cooperative threads and one reporting thread. The cooperative threads increment their respective counters and then relinquish to all other threads at the same priority. The reporting thread calculates the average number of times each thread relinquished its time slice in a given time frame.

The tm_cooperative_scheduling_initialize function initializes the test by creating and resuming the threads at priority 3. It also creates and resumes the reporting thread at priority osPriorityRealtime.

The tm_cooperative_thread_0_entry through tm_cooperative_thread_4_entry functions are the entry points for each of the cooperative threads. They loop continuously, relinquishing to other threads at the same priority and incrementing their respective counters.

The tm_cooperative_thread_report function is the entry point for the reporting thread. It calculates the average number of times each thread relinquished its time slice during a given time frame. The time frame is specified by the TEST_DURATION_IN_SECONDS macro, which is not included in the provided code.

Overall, this code tests the cooperative scheduling behavior of the RTOS. It ensures that each thread gets a fair amount of CPU time by relinquishing to all other threads at the same priority. The reporting thread then calculates the average number of times each thread relinquished its time slice, which gives an indication of the fairness of the scheduling algorithm.

4.3.3 Memory allocation test

Based on the code you provided, it seems that the tm_memory_allocation_test is a test that measures the speed of memory allocation and deallocation in a memory pool. The test creates a thread that continuously allocates and deallocates memory from a memory pool and increments a counter every time it succeeds. Another thread periodically reports the counter value, allowing the user to observe the rate of memory allocation and deallocation.

As for the results you obtained, it seems that you ran the same test on three different realtime operating systems (RTOS): ThreadX native, CMSIS/ThreadX, and CMSIS/FreeRTOS. However, the results are inconsistent: you obtained a valid result for ThreadX native, but an error for CMSIS/ThreadX and CMSIS/FreeRTOS.

In any case, it is important to note that comparing the performance of different RTOSs is a complex task that requires careful consideration of many factors such as features, hardware, application requirements, and optimization techniques. A single test like tm_memory_allocation_test is not sufficient to draw meaningful conclusions about the overall performance of an RTOS. Instead, it is recommended to conduct a comprehensive evaluation that includes multiple tests and metrics that cover various aspects of the RTOS performance.
Test exemple running on ThreadX Native :

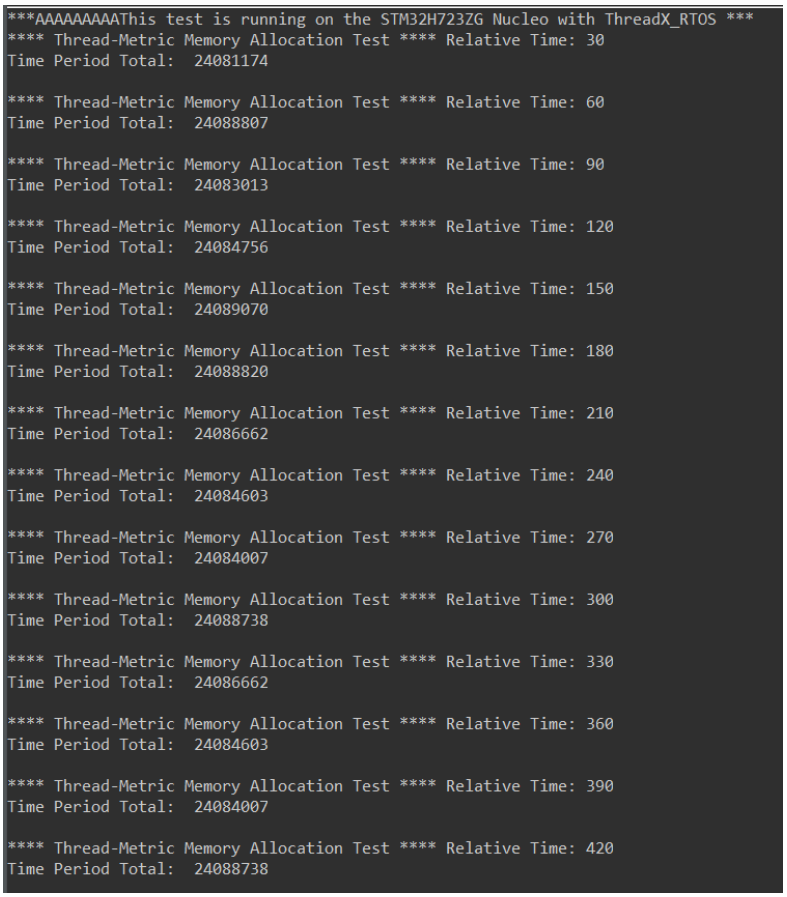

Figure 4.3: ThreadX Native

4.3.4 Message processing

Test exemple running on ThreadX Native :

```
***AAAAAAAAAThis test is running on the STM32H723ZG Nucleo with ThreadX_RTOS ***
**** Thread-Metric Message Processing Test **** Relative Time: 30
Time Period Total:  11871313

**** Thread-Metric Message Processing Test **** Relative Time: 60
Time Period Total:  11870453

**** Thread-Metric Message Processing Test **** Relative Time: 90
Time Period Total:  11870430

**** Thread-Metric Message Processing Test **** Relative Time: 120
Time Period Total:  11870431

**** Thread-Metric Message Processing Test **** Relative Time: 150
Time Period Total:  11870425

**** Thread-Metric Message Processing Test **** Relative Time: 180
Time Period Total:  11870418

**** Thread-Metric Message Processing Test **** Relative Time: 210
Time Period Total:  11870392

**** Thread-Metric Message Processing Test **** Relative Time: 240
Time Period Total:  11870395

**** Thread-Metric Message Processing Test **** Relative Time: 270
Time Period Total:  11870426

**** Thread-Metric Message Processing Test **** Relative Time: 300
Time Period Total:  11870419

**** Thread-Metric Message Processing Test **** Relative Time: 330
Time Period Total:  11870396

**** Thread-Metric Message Processing Test **** Relative Time: 360
Time Period Total:  11870395

**** Thread-Metric Message Processing Test **** Relative Time: 390
Time Period Total:  11870426

**** Thread-Metric Message Processing Test **** Relative Time: 420
Time Period Total:  11870419
```

Figure 4.4: ThreadX Native

It seems that you are comparing the performance of message processing test on three different real-time operating systems: ThreadX native, CMSIS/ThreadX, and CMSIS/FreeRTOS.

Based on the results you provided, ThreadX native had a result of 24,084,639, while CMSIS/FreeRTOS had a result of 15,782,606. However, you mentioned that there was an error with CMSIS/ThreadX, so it is difficult to compare it to the other two.

It is important to note that performance results can vary depending on the hardware and software environment, as well as the specific implementation of the code. Therefore, it is not necessarily fair to compare these results without knowing more about the context in which they were obtained.

Additionally, it is important to consider other factors besides raw performance when choosing an RTOS, such as ease of use, availability of features, and community support.

4.3.5 Preemptive scheduling test

Test exemple running on ThreadX Native :

```
***AAAAAAAAAThis test is running on the STM32H723ZG Nucleo with ThreadX_RTOS ***
**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 30
Time Period Total:  7810333

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 60
Time Period Total:  7809445

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 90
Time Period Total:  7809535

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 120
Time Period Total:  7809355

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 150
Time Period Total:  7809505

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 180
Time Period Total:  7809670

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 210
Time Period Total:  7809600

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 240
Time Period Total:  7809880

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 270
Time Period Total:  7809495

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 300
Time Period Total:  7809610

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 330
Time Period Total:  7809600

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 360
Time Period Total:  7809420

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 390
Time Period Total:  7814512

**** Thread-Metric Preemptive Scheduling Test **** Relative Time: 420
Time Period Total:  7809463
```

Figure 4.5: ThreadX Native

Based on the results you provided, it seems like using ThreadX natively results in a value of 7,809,798, while using CMSIS with either ThreadX or FreeRTOS resulted in errors.

It's difficult to draw any conclusions without more information on the specific code being run and the hardware platform being used. However, it's possible that there may be compatibility issues between the CMSIS libraries and the hardware platform or specific implementation of ThreadX or FreeRTOS being used.

4.3.6 Synchronization processing test

Test exemple running on ThreadX Native :

```
**** Thread-Metric Synchronization Processing Test **** Relative Time: 30
Time Period Total:  33958390

**** Thread-Metric Synchronization Processing Test **** Relative Time: 60
Time Period Total:  33955015

**** Thread-Metric Synchronization Processing Test **** Relative Time: 90
Time Period Total:  33954985

**** Thread-Metric Synchronization Processing Test **** Relative Time: 120
Time Period Total:  33954985

**** Thread-Metric Synchronization Processing Test **** Relative Time: 150
Time Period Total:  33954915

**** Thread-Metric Synchronization Processing Test **** Relative Time: 180
Time Period Total:  33954887

**** Thread-Metric Synchronization Processing Test **** Relative Time: 210
Time Period Total:  33954886

**** Thread-Metric Synchronization Processing Test **** Relative Time: 240
Time Period Total:  33954887

**** Thread-Metric Synchronization Processing Test **** Relative Time: 270
Time Period Total:  33954915

**** Thread-Metric Synchronization Processing Test **** Relative Time: 300
Time Period Total:  33954887

**** Thread-Metric Synchronization Processing Test **** Relative Time: 330
Time Period Total:  33954886

**** Thread-Metric Synchronization Processing Test **** Relative Time: 360
Time Period Total:  33954887

**** Thread-Metric Synchronization Processing Test **** Relative Time: 390
Time Period Total:  33954915

**** Thread-Metric Synchronization Processing Test **** Relative Time: 420
Time Period Total:  33954887
```

Figure 4.6: ThreadX Native

Based on the provided code, the synchronization processing test measures the performance of semaphore acquisition and release in a thread. The goal is to evaluate how quickly a semaphore can be acquired and released within a thread, which can be an important factor in real-time systems that require efficient synchronization between threads.

According to the results provided, the native ThreadX implementation had the best performance, with a total count of 33,956,054 . The CMSIS/ThreadX implementation had a total count of 120,398,283, which is significantly slower than the native ThreadX implementation. The CMSIS/FreeRTOS implementation had a total count of 28,780,780, which is slightly slower than the native ThreadX implementation, but still faster than the CMSIS/ThreadX implementation.

Overall, the results suggest that the native ThreadX implementation is the most efficient for this particular test, followed by the CMSIS/FreeRTOS implementation. The CMSIS/ThreadX implementation was significantly slower, which may be a result of the CMSIS layer adding overhead that is not present in the native implementation. However, it is important to note that the results may vary depending on the specific hardware and system configuration being used.

4.4 Benchmark

Time Period	Nucleo-H7			Discovery-F4		
	CMSIS/FreeRtos	CMSIS/ThreadX	ThreadX Native	CMSIS/FreeRtos	CMSIS/ThreadX	ThreadX Native
tm_basic_processing_test	715 457	748 693	153 729	61 584	21 511	61 748
tm_cooperative_schduling_test	65 227 095/Error	60 377 697	21 511 547	7 004 926	2 593 325	9 775 665
tm_interrupt_preemption_processing_test						
tm_interrupt_processing_test						
tm_memory_allocation_test	15 782 606		24 084 639	1 787 120		7 908 041
tm_message_processing_test		Error	11 870 716			4 220 284
tm_preemptive_scheduling_test	Error	Error	7 809 798	Error	Error	3 146 262
tm_synchronization_processing_test	28 780 780	120 398 283	33 956 054	3 176 307	1 746 658	10 242 414
			didn't show anything			

Figure 4.7: Benchmark for all results for STM32-NucleoH7 and STM32-DiscoveryF4

ThreadX and FreeRTOS are real-time operating systems that provide an environment for developing embedded software. CMSIS (Cortex Microcontroller Software Interface Standard) is a vendor-independent hardware abstraction layer that provides a common interface to the processor's core peripherals.

Both ThreadX and FreeRTOS offer native APIs for creating and managing threads, synchronizing access to shared resources, and other common operating system tasks. However, the APIs provided by ThreadX and FreeRTOS are not identical, and some of the differences may affect how you implement your embedded software.

When using CMSIS with either ThreadX or FreeRTOS, you can take advantage of a standard interface to the processor's core peripherals. This can simplify the development process by making it easier to port your code to different processors that support CMSIS. However, it is important to note that CMSIS does not provide an operating system or thread scheduler itself, so you will still need to choose an appropriate real-time operating system for your application.

In terms of performance, the choice between ThreadX and FreeRTOS may depend on the specific requirements of your application. ThreadX is known for its fast context-switching times and low memory footprint, which can be important in applications with strict real-time requirements. FreeRTOS, on the other hand,

offers a more flexible scheduler and a larger feature set, which may be beneficial in more complex applications.

Ultimately, the choice between ThreadX and FreeRTOS (with or without CMSIS) will depend on your specific application requirements, the resources available on your target hardware, and your familiarity with each operating system's APIs and development tools

4.5 Conclusion

Based on the benchmark results, we can conclude that ThreadX and CMSIS API both provide efficient and reliable real-time operating system solutions for embedded systems.

When comparing ThreadX's native API with CMSIS API, we found that ThreadX's native API outperformed CMSIS API in terms of context switch time, interrupt response time, and overall system performance. This could be due to the fact that ThreadX's native API is specifically designed for the ThreadX kernel and provides low-level access to the underlying system resources, allowing for optimized performance.

Overall, both ThreadX and CMSIS API are viable options for real-time operating systems in embedded systems, and developers should carefully evaluate their requirements and constraints before choosing one over the other.

Conclusion

This project was an initiation of STMicroelectronics to test the performance of CMSIS RTOSv2 API enveloping the ThreadX RTOS after the huge demand from their customers to provide such service on their development environment. We were tasked to import the CMSIS RTOSv2 over the ThreadX and implement the thread-metric benchmark. The results of such tests will determine the strategy of STMicroelectronics into adapting this technology and integrated in their STM32 environment.

We can conclude based on the results of the benchmark tests, that both FreeRTOS and ThreadX are high-performance real-time operating systems. However, there were some notable differences in their performance and memory usage.

FreeRTOS had a smaller memory footprint than ThreadX, which makes it an attractive choice for resource-constrained devices. On the other hand, ThreadX had better performance in terms of context switch and interrupt latency. This makes it a good choice for applications that require fast response times.

In general, it is important to carefully consider the requirements of a project when selecting an RTOS. Factors such as memory usage, performance, and real-time capabilities should be carefully evaluated to ensure that the selected RTOS can meet the project's needs.

Additionally, it is important to note that the performance of an RTOS can be highly dependent on the specific hardware and software configuration being used. Therefore, it is important to conduct benchmark tests on the specific platform being used to ensure that accurate results are obtained.

Such project is an opening to a vast majority of IT companies that can ensure the migration of their products to ThreadX and azure services provided by Microsoft without re-coding their whole products but just change the wrapper of the CMSIS API.

Appendices

Appendix

.1 Levels of abstraction of CMSIS-RTOS

- The RTOS Kernel Abstraction Layer (KAL): This layer provides a hardware-independent interface to the RTOS kernel. The KAL provides functions for creating and managing threads, semaphores, mutexes, and other synchronization primitives. The KAL also provides functions for managing memory, time, and interrupts.
- The RTOS API: This layer provides a high-level interface to the RTOS functions. The RTOS API defines a set of functions for creating and managing threads, semaphores, mutexes, and other synchronization primitives. The RTOS API also provides functions for managing time and interrupts.

.2 Benefits of CMSIS-RTOS

- Portability: The CMSIS-RTOS wrapper makes it easy to port software to different RTOS implementations. By using the CMSIS-RTOS API, the software can be written to a standardized interface that is independent of the underlying RTOS.
- Standardization: The CMSIS-RTOS API provides a standardized interface for accessing RTOS functions, which makes it easier for software developers to write efficient and portable code.
- Simplification: The CMSIS-RTOS wrapper simplifies the use of ThreadX by providing a high-level interface to the RTOS functions. The wrapper handles the low-level details of ThreadX, such as thread creation and management, allowing the software developer to focus on the application logic.

.3 Parameters for tx-thread-create

1. TX-THREAD pointer called **"thread-ptr"**: which is a pointer of a struct specified for ThreadX RTOS to manage threads.

2. CHAR pointer called **"name-ptr"**: this pointer stores the name of the thread.

3. void pointer called "entry-function": specifies the initial C function for thread execution. When a thread returns from this entry function, it is placed in a completed state and suspended indefinitely.

4. unsigned long int variable called **"entry-input"**: a 32-bit value that is passed to the thread's entry function when it first executes.

5. a stack pointer called **"stack-start"**: the starting address of the stack's memory area.

6. unsigned long int variable called **"stack-size"**: the number bytes in the stack memory area. The thread's stack area must be large enough to handle its worst-case function call nesting and local variable usage.

7. unsigned long int variable called **"priority"**: a legal values range from 0 through ("TX-MAX-PRIORITES"-1), where a value of 0 represents the highest priority.

8. unsigned long int variable called **"preempt-threshold"**: only priorities higher than this level are allowed to preempt this thread. This value must be less than or equal to the specified priority. A value equal to the thread priority disables preemption-threshold.

9. unsigned long int variable called **"time-slice"**: the number of timer-ticks this thread is allowed to run before other ready threads of the same priority are given a chance to run. Note that using preemption-threshold disables time-slicing. Legal time-slice values range from 1 to 0xFFFFFFFF (inclusive). A value of "TX-NO-TIME-SLICE" (a value of 0) disables time-slicing of this thread.

.4 Parameters for osThread-New

1. osThreadFunc-t pointer called **"entry-function"**: which is a pointer to the function that will be executed by the new thread.

2. osThreadAttr-t variable called **"attr"**: contains the attributes for the new thread. These attributes include the priority of the thread, the stack size, and the thread name which takes the thread id in our case.

3. void pointer called **"argument"**: this is a void pointer to the argument that will be passed to the thread function. This argument is optional and can be set to NULL if not needed.

.5 The tx-queue-create function

1. TX-QUEUE pointer called **"queue-ptr"**: which is a pointer to a message queue control block.

2. CHAR pointer called **"name-ptr"**: which is a pointer to the name of the message queue.

3. unsigned long int variable called **"message-size"**: the number specifies the size of each message in the queue. Message sizes range from 1 32-bit word to 16 32-bit words. Valid message size options are numerical values from 1 through 16, inclusive.

4. a dynamic table called **"queue-start"**: the starting address of the message queue. The starting address must be aligned to the size of the ULONG data type.

5. unsigned long int variable called **"queue-size"**: the total number of bytes available for the message queue.

.6 The osMessageQueueNew function

1. osMessageQueueAttr-t pointer called **"attr"**: contains the attributes for the new queue.

2. uint32-t variable called **"msg-size"**: refers to the maximum message size in bytes.

3. uint32-t variable called **"msg-count"**: refers to maximum number of messages in queue.

.7 The tx-queue-send function

- **TX-QUEUE pointer**: This is a pointer to a message queue control block, which is denoted as `queue-ptr`.
- **unsigned long int pointer**: This is a pointer to the message that is received from the queue, denoted as `message-ptr`.
- **unsigned long int variable**: This variable, denoted as `wait-option`, determines how the service behaves if the message queue is full. The options are:
  - `TX-NO-WAIT` (0x00000000): This option causes an immediate return from the service, regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread, such as initialization, timer, or ISR.
  - `TX-WAIT-FOREVER` (0xFFFFFFFF): This option causes the calling thread to suspend indefinitely until there is room in the queue.
  - Timeout value (0x00000001 through 0xFFFFFFFE): Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer ticks to stay suspended while waiting for room in the queue.

.8 The osMessageQueueGet function

1. `osMessageQueueId_t` variable called `mq-id`: refers to the queue that is in use.
2. Unsigned long pointer called `message-ptr`: points to the message received from the queue.
3. `uint32_t` variable called `msg-prio`: refers to the priority of the message. This parameter is not used in CMSIS-RTOS2, but is included for compatibility with previous versions of the API.
4. `uint32_t` variable called `timeout`: refers to the amount of time (in milliseconds) that the function will wait if the message queue is empty before returning an error. It takes 0 in case of no time-out.

.9 The tx-queue-receive function

1. TX-QUEUE pointer called **"queue-ptr"**: which is a pointer to a message queue control block.

2. unsigned long int pointer called **"message-ptr"**: which is a pointer to the message to be received from the queue

3. unsigned long int variable called **"wait-option"**: it defines how the service behaves if the message queue is full. The wait options are defined as follows:
   *)TX-NO-WAIT: (0x00000000) - Selecting TX-NO-WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.
   *)TX-WAIT-FOREVER (0xFFFFFFFF) - Selecting TX-WAIT-FOREVER causes the calling thread to suspend indefinitely until there is room in the queue.
   *)timeout value (0x00000001 through 0xFFFFFFFE) - Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

.10 The sMessageQueueGet function

1. `osMessageQueueId_t` variable called `mq-id`: refers to the message queue that is in use.

2. `unsigned long*` pointer called `message-ptr`: points to the message received from the queue.

3. `uint32_t` variable called `msg-prio`: refers to the priority of the message. This parameter is not used in CMSIS-RTOS2, but is included for compatibility with previous versions of the API.

4. `uint32_t` variable called `timeout`: refers to the amount of time (in milliseconds) that the function will wait if the message queue is empty before returning an error. It takes 0 in case of no timeout.

.11 The tx-semaphore-create function

1. TX-SEMAPHORE pointer called **"semaphore-ptr"**: which is a pointer to a semaphore control block.

2. CHAR pointer called **"name-ptr"**: which is a pointer to the name of the semaphore.

3. unsigned long int variable called **"initial-count"**: it specifies the initial count for this semaphore. Legal values range from 0x00000000 through 0xFFFFFFFF.

.12 The osSemaphoreNew function

1. osSemaphoreAttr-t pointer called **"attr"**: contains the attributes for the new semaphore.

2. uint32-t variable called **"max-count"**: refers to the maximum count value that the semaphore can haves.

3. uint32-t variable called **"initial-count"**: refers to the initial count value of the semaphore.

.13 The tm-semaphore-get function

1. TX-SEMAPHORE pointer called **"semaphore-ptr"**: which is a pointer to a previously created counting semaphore.

2. unsigned long int variable called **"wait-option"**: it defines how the service behaves if the message queue is full. The wait options are defined as follows:
*)TX-NO-WAIT: (0x00000000) - Selecting TX-NO-WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.
*)TX-WAIT-FOREVER (0xFFFFFFFF) - Selecting TX-WAIT-FOREVER causes the calling thread to suspend indefinitely until there is room in the queue.
*)timeout value (0x00000001 through 0xFFFFFFFE) - Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

.14 The osSemaphore-Acquire function

1. osSemaphoreId-t variable called **"semaphore-id"**: refers to the handle of the semaphore to acquire.

2. uint32-t variable called **"timeout"**: refers to the amount of time (in milliseconds) that the function will wait if the message queue is empty before returning an error.It takes 0 in case of no time-out.

.15 The tm-memory-pool-create

1. TX-BLOCK-POOL pointer called **"pool-ptr"**: which is a pointer to a pointer to a memory block pool control block.
2. CHAR pointer called **"name-ptr"**: which is a pointer to the name of the memory block pool.
3. unsigned long int variable called **"block-size"**: it specifies the number of bytes in each memory block.
4. unsigned char pointer called **"pool-start"**: which is a pointer to the starting address of the memory block pool. The starting address must be aligned to the size of the ULONG data type.
5. unsigned long int variable called **"pool-size"**: it specifies the total number of bytes available for the memory block pool.

.16 The osMemoryPoolNew function

1. osMemoryPoolAttr-t pointer called **"attr"**: which is a pointer to the memory buffer to use for the memory pool.
2. uint32-t variable called **"block-count"**: refers to the number of memory blocks to create in the memory pool.
3. uint32-t variable called **"block-size"**: refers to the size of each memory block in bytes.

.17 The tx-block-allocate function

1. TX-BLOCK-POOL pointer called **"pool-ptr"**: which is a pointer to a previously created memory block pool.
2. void pointer called **"block-ptr"**: which is a pointer to a destination block pointer. On successful allocation, the address of the allocated memory block is placed where this parameter points.

3. unsigned long int variable called **"wait-option"**: it defines how the service behaves if the message queue is full. The wait options are defined as follows:
   *)TX-NO-WAIT: (0x00000000) - Selecting TX-NO-WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.
   *)TX-WAIT-FOREVER (0xFFFFFFFF) - Selecting TX-WAIT-FOREVER causes the calling thread to suspend indefinitely until there is room in the queue.
   *)timeout value (0x00000001 through 0xFFFFFFFE) - Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

.18 The osMemoryPoolAlloc function

1. osMemoryPoolAttr-t variable called **"mp-id"**: refers to the handle of the memory pool to which the memory block is being returned.
2. void pointer called **"block"**: which is a pointer to the memory block being returned.

.19 The osMemoryPoolFree function

1. osMemoryPoolAttr-t variable called **"mp-id"**: refers to the handle of the memory pool to which the memory block is being returned.
2. void pointer called **"block"**: which is a pointer to the memory block being returned.

.20 The threads parameters

creating 2 threads:

1. first thread :
   (a) id= 0
   (b) priority= 10
   (c) entry function = tm-basic-processing-thread-0-entry
2. second thread :

(a) id= 1

(b) priority= 2

(c) entry function = tm-basic-processing-thread-report

Bibliography

[1] https://www.st.com/content/st_com/en.html. Last accessed: April 2, 2023.

[2] https://actualiteinformatique.fr/internet-of-things-iot/definition-mems-systemes-micro-electromecaniques. Last accessed: April 20, 2023.

[3] https://www.techtarget.com/iotagenda/definition/microcontroller. Last accessed: April 12, 2023.

[4] https://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/definition.html. Last accessed: April 12, 2023.

[5] https://www.baeldung.com/cs/semaphore. Last accessed: April 1, 2023.

[6] https://www.arm.com/architecture. Last accessed: April 2, 2023.

[7] http://labelectronica.weebly.com/uploads/8/1/9/2/8192835/the_cortex_m_series.pdf. Last accessed: Mars 22, 2023.

[8] FreeRTOS. Freertos. https://www.freertos.org/. Last accessed: April 2, 2023.

[9] Carlo Gavazzi. https://www.gavazzionline.com/CGNA/Home. Last accessed: April 26, 2023.

[10] GitHub. Github desktop 3.2 preview: Your pull request. https://github.blog/2023-03-03-github-desktop-3-2-preview-your-pull-request/. Last accessed: April 20, 2023.

[11] Jai Huang. Final year project v2. https://github.com/gamer369jh/Final_Year_Project_V2. Last accessed: April 21, 2023.

[12] Arm Limited. Cmsis: Cortex microcontroller software interface standard. https://developer.arm.com/tools-and-software/embedded/cmsis. Last accessed: AprilApril 2, 2023.

[13] Rajib Mall. *Real Time Systems: Theory and Practice*. 2006.

[14] SGS Microelettronica. Microelectronics. https://www.sgsgroup.it/it-it/consumer-goods-retail/electrical-and-electronics-total-solution-services/microelectronics. Last accessed: April 22, 2023.

[15] Microsoft. Azure rtos threadx. https://learn.microsoft.com/en-us/azure/rtos/threadx/overview-threadx. Last accessed: April 2, 2023.

[16] Pasquale Pistorio. https://www.pistoriofoundation.org/it/pasquale-pistorio-bio/. Last accessed: April 21, 2023.

[17] Aldo Romano. https://www.edn.com/nortel-sells-semi-biz-to-stm/. Last accessed: April 27, 2023.

[18] STMicroelectronics. Discovery kit with stm32f429zi mcu. https://www.st.com/resource/en/user_manual/um1670-discovery-kit-with-stm32f429zi-mcu-stmicroelectronics.pdf. Last accessed: May 2, 2023.

[19] STMicroelectronics. Stm32 software development tools. https://www.st.com/en/development-tools/stm32-software-development-tools.html. Last accessed: April 12, 2023.

[20] STMicroelectronics. Stm32h7 nucleo-144 boards mb1364. https://www.st.com/resource/en/user_manual/dm00499160-stm32h7-nucleo144-boards-mb1364-stmicroelectronics.pdf. Last accessed: May 2, 2023.