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

Summer Internship

Implementation of use cases using GPDMA with different STM32 peripherals : PSSI and DCMI on STM32U5

Developed by :

Jaafer Hosni

Supervised by :

Mme. Kaouthar DJEMEL

3rd Year Telecommunication Engineering

Academic year: 2023/2024

Contents

List of Figures

Acknowledgements

I dedicate this page to express my 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).

I would especially like to thank Mme. Kaouthar DJEMEL for her technical guidance through the whole process of this project development. Her technical advises and shared experience would always be craved in my memory and help me go further into the path of embedded software development.

I would also like to express my gratitude and deep appreciation to the members of the jury for the honor they have given me by accepting to evaluate my work.

Introduction

This report documents a comprehensive exploration of the STM32U5 microcontroller, emphasizing its remarkable capabilities and potential applications, with a primary focus on the General-Purpose Direct Memory Access (GPDMA), Digital Camera Interface (DCMI), and Parallel Synchronous Slave Interface (PSSI). Developed by STMicroelectronics, the STM32U5 represents a relatively new addition to the STM32 family, boasting a powerful ARM Cortex-M33 core, an array of peripherals, and connectivity options that make it an ideal choice for high-performance and efficient projects.

This report delves into the STM32U5's unique features and the synergistic integration of GPDMA with DCMI, which paves the way for innovative solutions in computer vision and image processing. It highlights the potential for various applications, including object recognition and gesture detection, enabled by the STM32U5's processing capabilities and the versatility of the DCMI interface.

Additionally, we explore the utilization of the PSSI peripheral for parallel data communication, a feature that offers faster and more efficient data transfer between the STM32U5 and other devices. Despite its promise, the PSSI peripheral currently faces some challenges in terms of community support compared to more established interfaces like SPI and UART. However, we remain optimistic about its growth and widespread adoption in the STM32 community over time.

Throughout this report, we will delve into the tools and concepts behind the STM32U5, provide an overview of the hardware components used in our project, discuss relevant board support packages and software tools, and detail our project's initiation, implementation, and deployment phases. By the end of this document, you will gain a comprehensive understanding of the STM32U5's capabilities, its potential applications in computer vision and data communication, and the challenges and opportunities it presents in the ever-evolving world of embedded systems and microcontroller development.

Problematic

This Internship Project revolves around the STM32U5 microcontroller, its peripherals, and their potential applications. Key concerns of this internship include the lack of community support for the Parallel Synchronous Slave Interface (PSSI) in comparison to well-established interfaces, optimizing the Digital Camera Interface (DCMI) for computer vision, fostering the growth of the STM32U5 ecosystem, and ensuring compatibility and interoperability with other devices. These challenges and uncertainties necessitate further examination to facilitate the effective utilization of the STM32U5 microcontroller and its associated components in diverse applications and projects.

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 [**?**], Pasquale Pistorio [4], and Aldo Romano [**?**] 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[**?**] 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[**?**], 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 internship involving the implementation of use cases using GPDMA with different STM32 peripherals on the STM32U5 microcontroller is to leverage the capabilities of the hardware to achieve efficient data transfer, reduce CPU load, and enable various applications, especially those requiring high-speed data processing or real-time performance.

2.2 Key Concepts

2.2.1 General-Purpose Direct Memory Access

The General-Purpose Direct Memory Access (GPDMA) is a hardware feature commonly found in microcontrollers and microprocessors. Its primary purpose is to facilitate efficient and high-speed data transfers between various peripherals and memory without requiring continuous CPU intervention. GPDMA operates independently of the CPU, allowing it to move blocks of data between memory locations, peripherals, or even between different peripherals, such as UARTs, SPI controllers, and ADCs. This functionality is especially valuable in applications where data needs to be moved quickly and continuously, like audio and video streaming, sensor data acquisition, or communication protocols. GPDMA not only improves overall system performance but also reduces CPU load, making it a crucial component for real-time and resource-efficient embedded systems.

2.2.2 Parallel Synchronous Slave Interface

The Parallel Synchronous Slave Interface (PSSI) is a communication protocol and hardware interface commonly used in embedded systems and microcontroller-based applications. It operates as a slave device, meaning it responds to commands and data requests from a master device, often a microcontroller or a host controller. PSSI facilitates synchronized data exchange between the slave and the master in a parallel fashion, typically using multiple data lines to transfer data simultaneously. This parallelism allows for efficient and high-speed data transfer, making PSSI suitable for applications requiring rapid data communication, such as memory interfacing, sensor data acquisition, or interfacing with other digital peripherals. PSSI ensures precise timing control, ensuring that data bits are synchronized and transferred at the correct rate and maintaining data integrity in real-time or high-performance scenarios. It plays a vital role in enabling seamless communication and data transfer within embedded systems.

2.2.3 Digital Camera Interface

The Digital Camera Interface (DCMI) is a specialized hardware component found in microcontrollers and microprocessors, designed to interface with digital image sensors and cameras. DCMI serves as a bridge between the microcontroller and the image sensor, enabling the capture and transmission of digital image data. It provides a standardized way to connect to various image sensors, handling tasks such as data synchronization, pixel data retrieval, and control signals. DCMI is crucial for applications requiring image and video processing, such as surveillance systems, industrial automation, and multimedia devices. It simplifies the integration of cameras into embedded systems, allowing developers to capture high-quality images and video streams efficiently, making it an essential component for applications where visual data is a key input or output.

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.[**?**]

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.[2]

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.

TheOverall, 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 STM32U575I_EV

The STM32U575I_EV is a development board designed to provide an accessible and versatile platform for developers and engineers working with STMicroelectronics' STM32U5 series microcontrollers. This development board is equipped with an STM32U575 microcontroller, which is part of the STM32U5 family known for its high performance, advanced security features, and energy efficiency.[6]

The STM32U5 microcontroller features a powerful Arm Cortex-M33 core running at up to 480 MHz, providing high-performance computing capabilities. Additionally, the STM32U575I_EV incorporates advanced security features, including hardware-based secure boot and cryptographic accelerators, making it suitable for applications demanding robust security. With a comprehensive set of peripherals, including UART, SPI, I2C, USB, and more, this microcontroller facilitates versatile connectivity and control. Its low-power modes enhance energy efficiency, making it ideal for battery-powered applications.
This board is a great choice for projects that require a Digital Camera Interface (DCMI) to interface with image sensors. The choice of such board is due to having such important features like:

1. DCMI interface: the STM32U575I-EV board features a dedicated DCMI interface that allows it to interface with a wide range of image sensors. This interface supports various image formats, including YUV, RGB, and JPEG.
2. High-performance microcontroller: The STM32U575 microcontroller is a high-performance device with an Arm Cortex-M33 core that can run at up to 480 MHz It also has a large amount of flash memory and RAM, which makes it ideal for image processing applications.
3. Rich set of peripherals: The board has a rich set of peripherals, including USB, Ethernet, CAN, and UART interfaces. These peripherals can be used to communicate with other devices or to control external components.
4. On-board sensors: The board also features a range of on-board sensors, including an accelerometer, a gyroscope, and a magnetometer. These sensors

can be used to detect motion and orientation, which can be useful in image stabilization applications.

5. Expansion options: The board has a range of expansion options, including Arduino and STMod+ connectors. These connectors allow you to add additional functionality to the board, such as wireless connectivity or additional sensors.

6. Power management: The STM32U575I-EV board features advanced power management options, including low-power modes and dynamic voltage scaling. These features help to optimize power consumption and extend battery life in portable applications.

Figure 2.1: a front view of the STM32U575I_EV

2.3.3 MB1379A Camera Module

he MB1379A camera module is an excellent choice for video processing applications that require high performance and low power consumption. Its composite video input, sync separation capabilities, STM32 microcontroller interface, low power consumption, and small form factor make it an ideal choice for video processing applications that require high performance and low power consumption.[6] The MB1379A camera module is a highly integrated device that includes several features that make it ideal for video processing applications. Some of the key

features of the MB1379A camera module include:

1. Composite video input: The MB1379A camera module includes a composite video input that allows it to receive video signals from a variety of sources, including cameras, VCRs, and DVD players.
2. Sync separation: The MB1379A camera module is designed to extract horizontal and vertical synchronization signals from composite video signals. These signals are used to synchronize the display of video images on a monitor or TV screen.
3. STM32 microcontroller interface: The MB1379A camera module is designed to work with the STM32 microcontroller family. It includes an interface that allows it to communicate with the microcontroller and transfer video data.
4. Low power consumption: The MB1379A camera module is designed to operate on low power, making it ideal for battery-powered applications.
5. Small form factor: The MB1379A camera module is available in a small form factor package, which makes it easy to integrate into a variety of video processing applications.

Figure 2.2: a front view of the MB1379A camera module

To mount such module to the STM32U575I_EV board you need to follow the provided schematic by STMicroelectronics.

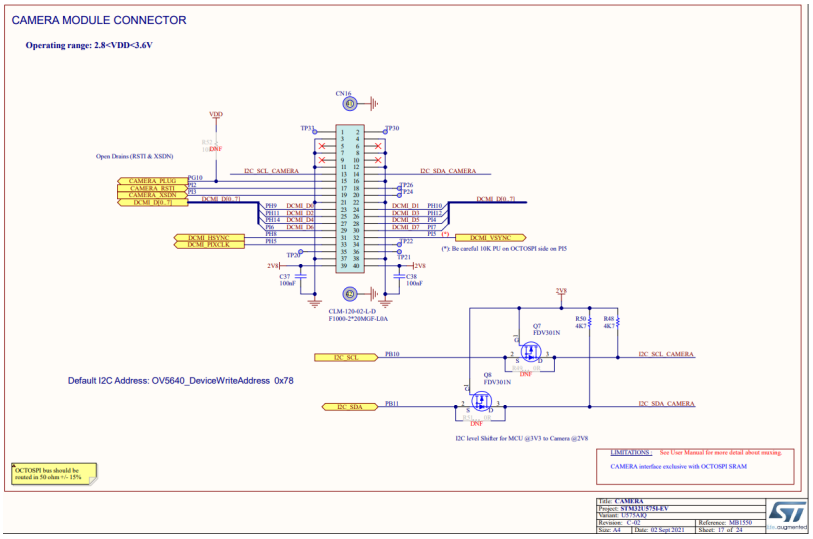

Figure 2.3: mounting schematic of the MB1379A camera module to the STM32U575I_EV board

2.3.4 MB989C LCD Module

The MB989C is a 3.5-inch TFT LCD display module that can be interfaced with an STM32 microcontroller. This display module features a resolution of 320x240 pixels and supports 16-bit color depth. It also includes a resistive touch panel for user input.[6]
It uses a 16-bit parallel interface to communicate with the STM32 microcontroller. This interface allows for fast data transfer and enables the display to update quickly.

Figure 2.4: a front view of the MB989C LCD module

To mount such module to the STM32U575I_EV board you need to follow the provided schematic by STMicroelectronics.

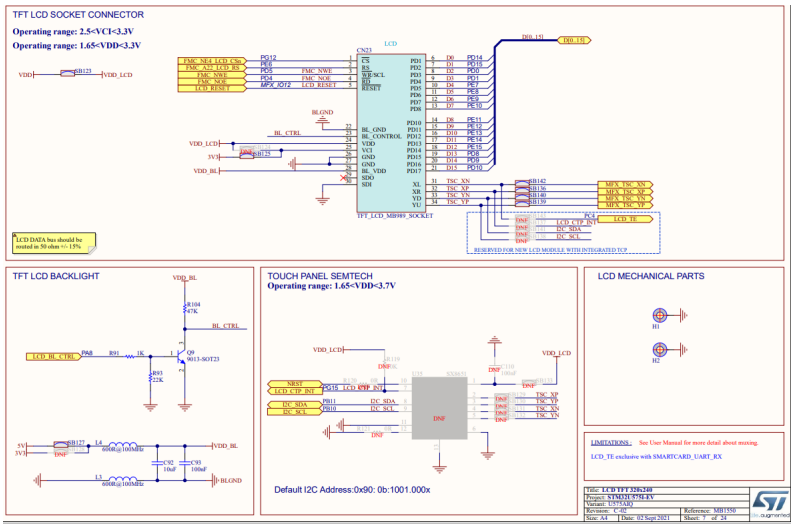

Figure 2.5: mounting schematic of the MB989C LCD module to the STM32U575I_EV board

2.3.5 STM32U575ZI-Q_Nucleo

The STM32U575ZI-Q Nucleo is a development board designed to provide an accessible and versatile platform for developers and engineers working with STMicroelectronics' STM32U5 series microcontrollers. This development board is equipped with an STM32U575ZI-Q microcontroller, which is part of the STM32U5 family known for its high performance, advanced security features, and energy efficiency. The Nucleo board offers a wide range of peripherals and interfaces, making it an ideal choice for prototyping and evaluating applications in various domains, including IoT, industrial automation, and embedded systems. With its user-friendly design, extensive documentation, and compatibility with popular integrated development environments (IDEs) like STM32CubeIDE, the STM32U575ZI-Q Nucleo accelerates the development process and allows engineers to harness the full potential of the STM32U5 microcontroller for their projects.[0]

The STM32U575ZI-Q microcontroller features a powerful Arm Cortex-M33 core running at up to 160 MHz, providing high-performance computing capabilities. Additionally, the STM32U575ZI-Q incorporates advanced security features, including hardware-based secure boot and cryptographic accelerators, making it suitable for applications demanding robust security. With a comprehensive set

of peripherals, including UART, SPI, I2C, USB, and more, this microcontroller facilitates versatile connectivity and control. Its low-power modes enhance energy efficiency, making it ideal for battery-powered applications.

Figure 2.6: a front view of the STM32U575ZI-Q_Nucleo

2.4 Board Support Package

To ensure the right usage of all the peripherals with the hardware and the boards we need to use certain firmware to ensure the transfer of data, quick accessibility, and easier code such firmware is provided either by 3rd parties or official release from STMicroelectronics like the BSP
The Board Support Package(BSP), refers to a software layer or collection of software components designed to provide an abstraction layer between the hardware of a specific embedded computing platform, like a microcontroller or microprocessor, and the higher-level software applications running on it. BSPs include device drivers, initialization code, hardware abstraction libraries, and other essential software elements tailored to the specific hardware configuration of the target board. They serve to simplify the process of developing software for embedded systems by offering standardized interfaces and functions for accessing and controlling the underlying hardware, thus enabling developers to focus more on application development and

less on hardware intricacies. BSPs play a crucial role in ensuring the portability and scalability of software across different hardware platforms, making them a fundamental component in embedded system development.

2.5 Software

2.5.1 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..[5]

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 2.7: STM32cubeMX logo

2.5.2 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.[5]

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.

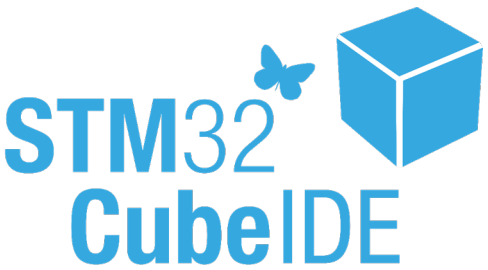

Figure 2.8: STM32cubeIDE logo

2.5.3 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.[3]
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 2.9: GitHub Desktop logo

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 STM32U575I-EV and STM32U575ZI-Q_Nucleo boards which we used for deployment.

Additionally, we have introduced the various peripherals (mainly the DCMI and thee PSSI) that we will be working with, as well as how they are hundled by the GPDMA which is the central focus of my projects.

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

In this chapter, we explore the integration of use cases for the STM32U5 microcontroller, focusing on its Digital Camera Interface (DCMI) and Parallel Synchronous Slave Interface (PSSI). We will also leverage the General-Purpose Direct Memory Access (GPDMA) controller for efficient data transfer. This combination of features enables a wide range of applications, from image processing to sensor interfacing, and enhances data processing efficiency.Furthermore, the incorporation of the GPDMA controller serves as a catalyst, enhancing data transfer efficiency and minimizing processor load, making your applications more robust and efficient.

3.2 DCMI Use Case application

In the context of image capturing and image processing, it is essential to understand various image formats. When working with the STM32U5 microcontroller, especially in conjunction with the Digital Camera Interface (DCMI), three prominent image formats come to the forefront: RGB, YCbCr, and Y-only.

1. **RGB (Red, Green, Blue):** RGB is a full-color image representation that utilizes red, green, and blue channels. It is invaluable for applications where color accuracy is paramount, such as digital cameras. When paired with the STM32U5's DCMI, RGB image capture harnesses the microcontroller's capabilities for handling vibrant, high-resolution visual data.

2. **YCbCr:** YCbCr separates luminance (Y) and chrominance (Cb and Cr) information, offering efficient data storage and processing. In the context of STM32U5's DCMI, YCbCr provides the flexibility needed for applications like video compression, granting precise control over image components.

3. **Y-only:** Y-only, a grayscale format, simplifies image data to shades of gray without color information. This format is suitable for applications that do not require color details, such as document scanning. STM32U5's DCMI efficiently captures Y-only images, tailored to specific use cases.

3.2.1 Project Objective

In this section, we delve into various typical examples of Digital Camera Interface (DCMI) use, each designed to demonstrate the versatility and capabilities of this interface. These examples encompass the following key objectives:

1. **Capture and Display of RGB Data:** Our first objective is to capture RGB data in the RGB565 format with a QVGA (320x240) resolution. The captured data is then efficiently stored in the Synchronous Dynamic Random-Access Memory (SDRAM) and subsequently displayed on the LCD-TFT screen. This use case showcases the STM32U5's prowess in handling real-time image acquisition and display.

2. **Capture of YCbCr Data:** The second scenario involves capturing data in the YCbCr format, also with a QVGA (320x240) resolution. Here, the primary aim is to demonstrate the STM32U5's ability to efficiently capture and store YCbCr data in the SDRAM, a valuable feature for applications such as video processing and compression.

3. **Capture of Y-Only Data:** In our final example, we configure the DCMI to receive Y-only data, emphasizing the flexibility of this interface. The Y-only data is recorded and stored in the SDRAM, highlighting the STM32U5's adaptability to various data formats and its suitability for diverse applications.

Through these use cases, we aim to showcase the STM32U5's capabilities in image and data acquisition, storage, and manipulation, offering valuable insights for developers looking to harness the full potential of this microcontroller in their projects.

3.2.2 Hardware Configuration

For this project, we are going to use the STM32U575I-EV with the pre-mounted peripherals the MB1379A Camera Module and the MB989C LCD Module which are mentioned in the previous chapter with their prospective mounting schematics. The camera module connects to the Digital Camera Interface (DCMI) in the STM32 microcontroller to enable efficient image acquisition and processing.
The DCMI serves as the intermediary between the camera sensor and the microcontroller, acting as a bridge for the parallel data produced by the camera. Typically, the camera module interfaces with the DCMI through a dedicated connector, with wires carrying the image data and an I2C communication for clock signals, and control signals.
The DCMI, integrated into the STM32 microcontroller, provides the necessary hardware and software support to handle the incoming image data, facilitating real-time processing, storage, and display.

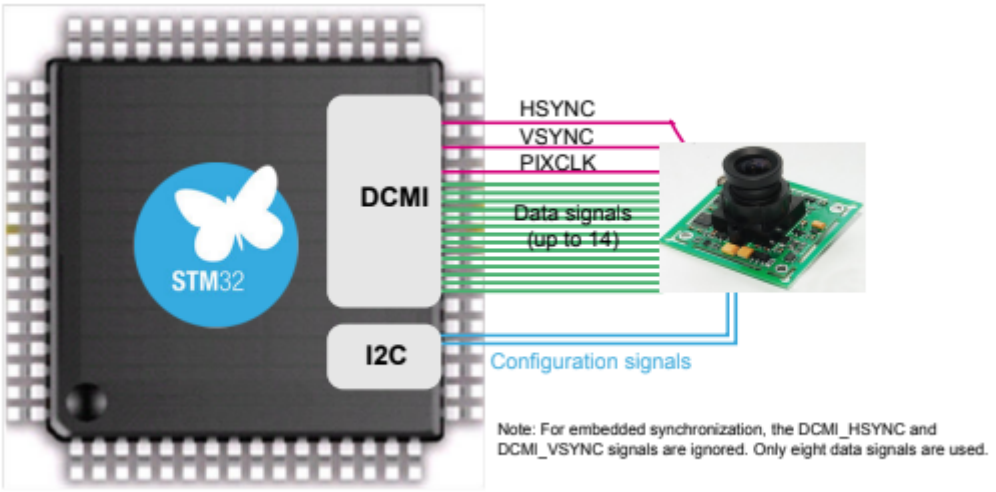

Figure 3.1: STM32 MCU and camera module interconnection.

In the STM32U5 microcontroller, the transfer of image data from the Digital Camera Interface (DCMI) to the General-Purpose Direct Memory Access (GPDMA) controller through the Advanced High-Performance Bus 2 (AHB2).

The AHB2 bus is part of the Advanced Microcontroller Bus Architecture (AMBA) that is widely used in ARM-based microcontrollers. It is primarily used to connect high-performance peripherals and interfaces, such as memory interfaces, external memory controllers, and other high-bandwidth components. It provides a dedicated high-speed data transfer pathway to ensure efficient and high-throughput data exchange between these peripherals.

The GPDMA controller is configured to receive data from the DCMI via AHB2. This setup includes specifying the source (DCMI) and destination (e.g., memory or peripheral) for data transfer, as well as the data size and any necessary control signals.
Once configured, the DCMI begins capturing image data, which is then sent directly through AHB2 to the GPDMA without MPU intervention. This data flow is critical for real-time processing and ensures minimal latency in handling the captured images.

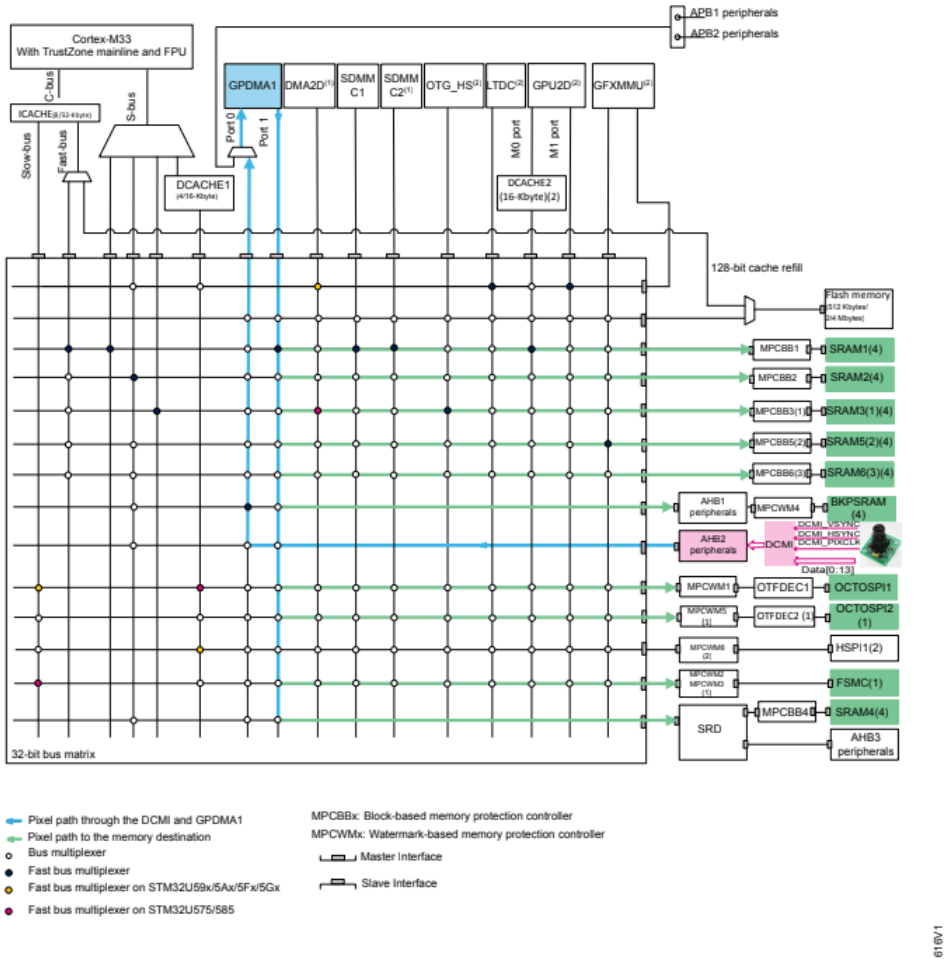

Figure 3.2: DCMI slave AHB2 peripheral in STM32U5.

The GPDMA can efficiently transfer the image data to a specified destination, such as internal memory, external memory, or a peripheral module for further processing, analysis, or display. In our case, the GPDMA will send the data to the SRAM so that it could be fetched by the LCD Module to display the captured images.

Project Initialization

The STM32 development environment offer many ways to configure peripherals and allocate pins and processor registers, yet the most easy and more used is using

the STM32CubeMX for such task.

Firstly, we initialize a project using board selecter and you select the right bord in this case the STM32U575I-EV borad

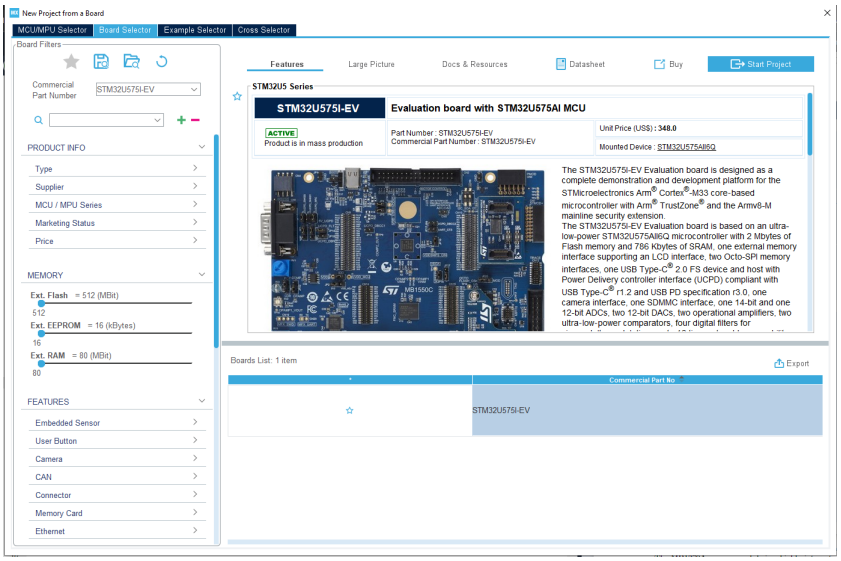

Figure 3.3: DCMI project initialization using STM32CubeMX.

DCMI Pins and Peripheral Configuration

After initializing the project, we select "DCMI" in the "Multimedia" section where you well be met with the graphical interface to configure the DCMI perepheral settings.

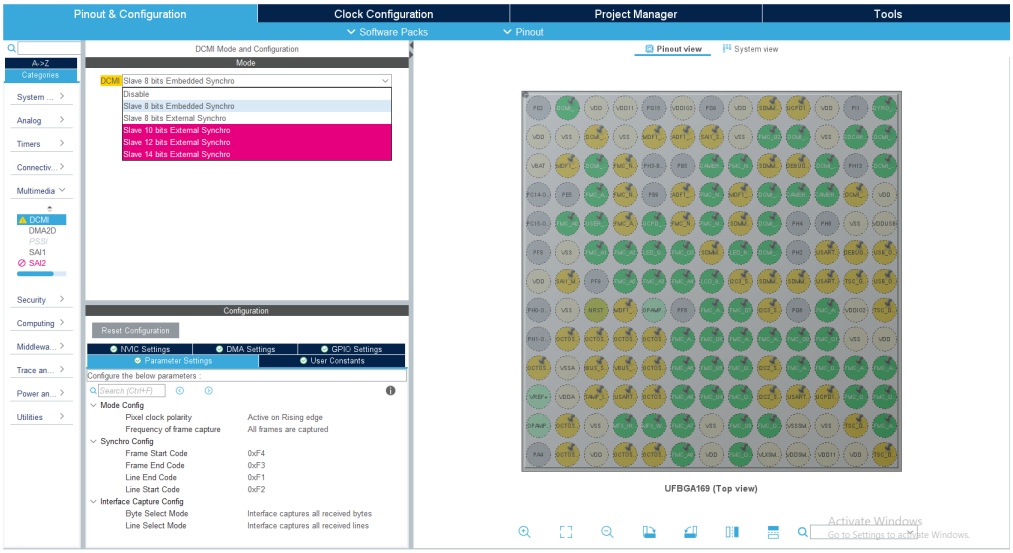

Figure 3.4: DCMI peripheral configuration using STM32CubeMX.

The DCMI is being configured, with various modes selectable in a dropdown menu such as:

- Slave 8 bits Embedded Synchrono
- Slave 8 bits External Synchrono
- Slave 10 bits External Synchrono
- Slave 12 bits External Synchrono
- Slave 14 bits External Synchrono

In this project we are going with "Slave 8 bits Embedded Synchrono" configuration. In terms of DCMI configuration, the following parameters are visible:

Mode Config: Settings related to the pixel clock polarity and frequency of frame capture.

Synchro Config: Configuration of synchronization codes for frame start, end, line end, and line start.

Interface Capture Config: Settings for byte select mode and line select mode, determining how the interface captures data.

This configuration must met the ones mentioned in **Figure 3.4** to ensure the correct performing of the project.

In digital camera interfaces like the DCMI (Digital Camera Interface) on STM32 microcontrollers, synchronization codes are crucial for correctly interpreting the data stream coming from a camera module. These codes are used to distinguish the different parts of the data frame that the camera sends. Here's what each synchronization code means in the context of the DCMI configuration:

Frame Start Code: This code indicates the beginning of a new frame of data. When the DCMI detects this code, it knows that the data following it represents the start of a new image frame.

Frame End Code: This code signals the end of the current frame. Upon detecting this code, the DCMI understands that the image frame is complete, and no more data will be received for this frame.

Line End Code: At the end of each line of pixels within a frame, this code is transmitted. It helps the DCMI to separate each line of image data.

Line Start Code: This code marks the beginning of a new line of pixels within a frame.

These synchronization codes allow the DCMI to parse the incoming data stream and organize it into a coherent frame structure that can then be processed or displayed by the system. They are typically specific patterns or sequences of bits that are unlikely to occur randomly in the pixel data, ensuring reliable detection by the hardware.

In the Synchro Config settings, you can specify the actual values for these codes. This needs to be matched with the camera module's output. If these are not set correctly, the microcontroller could misinterpret the incoming data, resulting in corrupted or skewed images.

GPDMA Configuration

After configuring the DCMI peripheral, we must configure the GPDMA to properly handling the data from the DCMI and transport it to the MPU of the STM32U575I-EV.
To do that we need firstly to select a channel and configured. A channel 0 to 11 can be allocated,unless a channel 12 to 15 is free/unallocated and the required performances in terms of bandwidth are significant for this AHB peripheral. A

channel 12 to 15 may be also allocated if the accessed memory is external.[4] For our project, we are going with channel 12 to ensure the best performance possible as well as to take advantage of the 2D addressing feature.

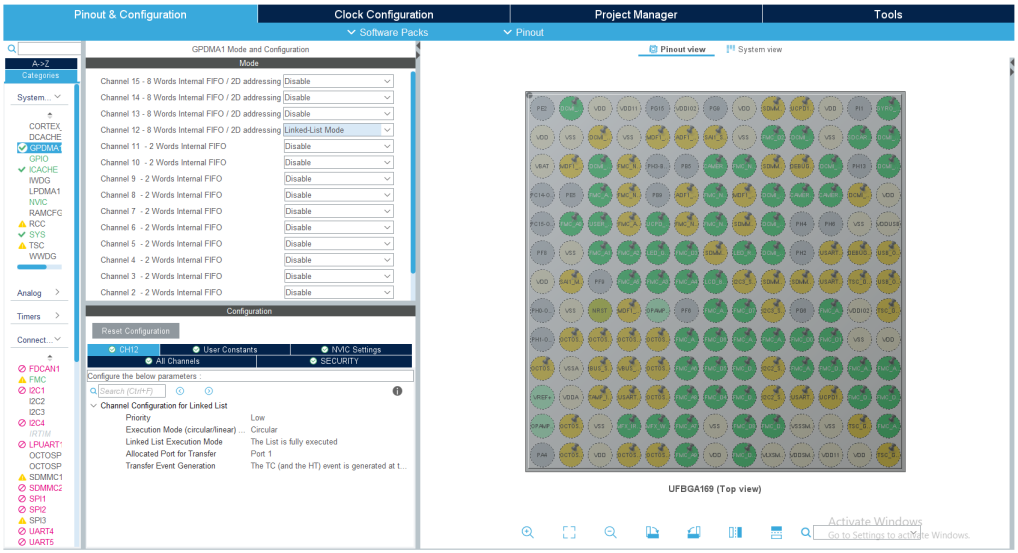

Figure 3.5: GPDMA channel configuration using STM32CubeMX.

Then we create a linked list that is going to store the data and handle the transfert of data between the DCMI and GPDMA.

The configuration is shown as follow in these figures.

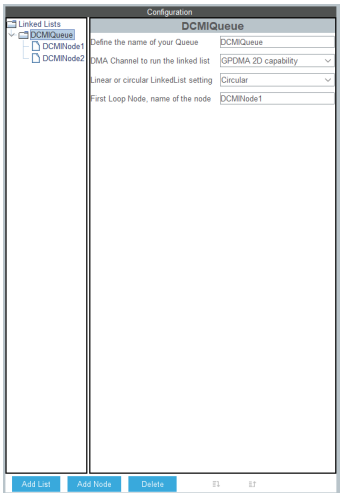

Figure 3.6: linked list intialization

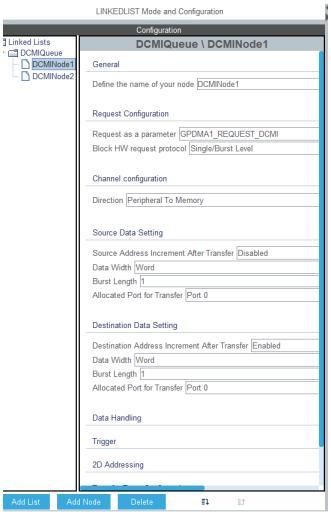

Figure 3.7: First node : GPDMA to DCMI

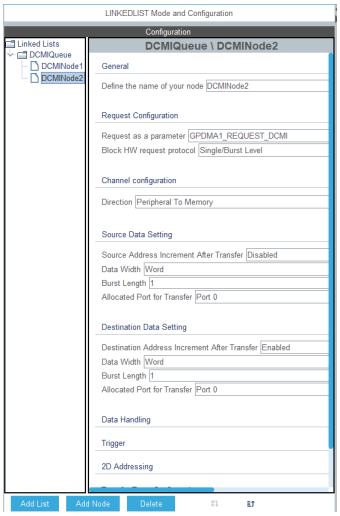

Figure 3.8: Second node : DCMI to GPDMA

Figure 3.9: GPDMA LinkedList configuration using STM32CubeMX

SRAM Configuration

Finally after finishing the GPDMA initialization, we need to configure the SRAM that will assure the data transfert from the MPU to the LCD so that we be able to project the image taken from the camera module into the LCD module. The configuration is as shown in this figure:

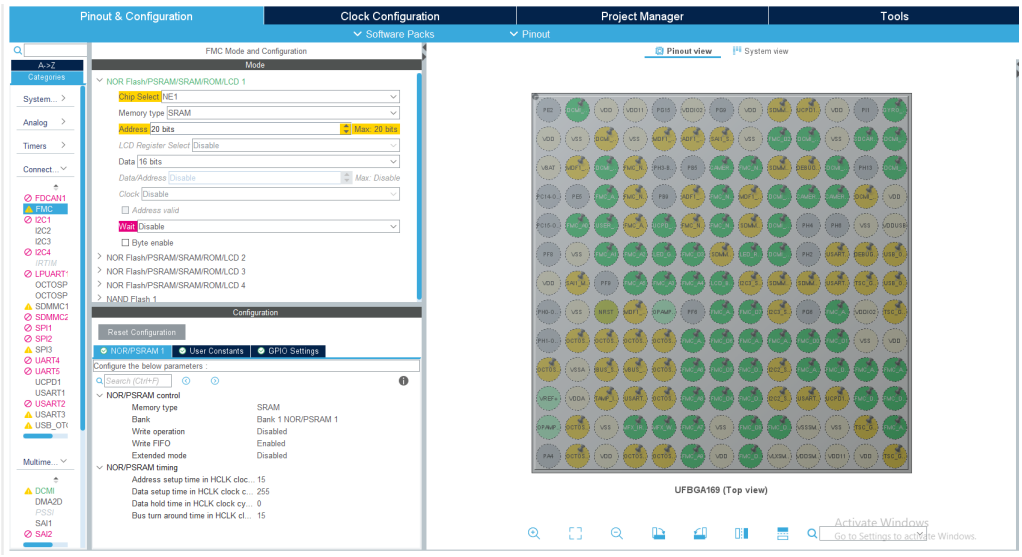

Figure 3.10: SRAM configuration using STM32CubeMX.

Code Generation

After finishing the initialization of all peripherals, we save the project and select one of the proposed IDEs (In our case, we are going with the STM32CubeIDE) to continue the application development as the STM32CubeMX offers the possibility to generate the necessary code for the mapped pins and the necessary settings of each peripheral

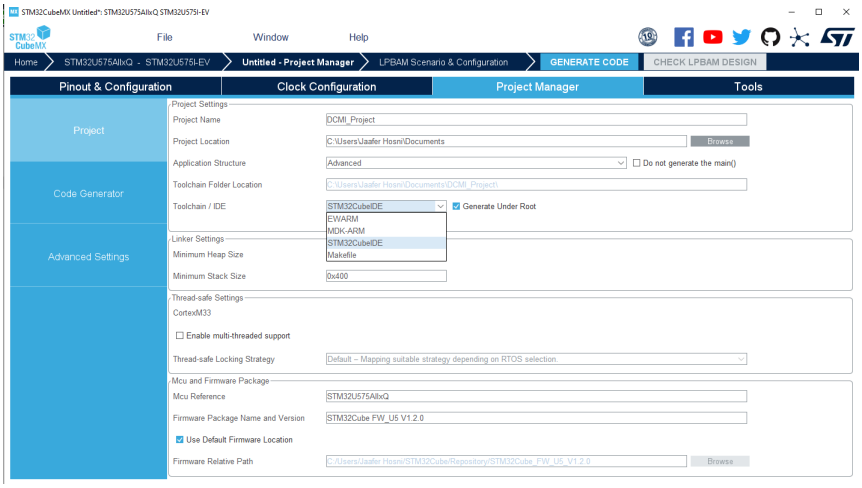

Figure 3.11: Code generation using STM32CubeMX.

3.2.3 Software Implementation

Importing packages and setting developing environment

In order to progress with the development of this project, it is necessary to include specific software packages that will be instrumental in its creation. Those packages are :

1. **Utilities Package**: This package includes functions for initializing the LCD display, configuring its orientation (e.g., landscape or portrait), drawing basic shapes like lines, rectangles, and circles, displaying text and graphics at specific screen coordinates, managing colors and pixel manipulation, handling touch screen input (if the LCD has a touch interface), controlling backlight settings, and providing higher-level abstractions for creating graphical user interfaces (GUIs). These utilities significantly simplify the process of working with LCD displays on STM32 boards, making it more accessible for developers to design visually appealing applications and user interfaces.

2. **Board Support Package**: The BSP acts as an intermediary layer between the hardware and application software, offering a comprehensive set of functions and drivers. These components encompass the configuration and control of various hardware peripherals on the board, such as GPIO pins, UART, SPI, I2C, USB, and more. Additionally, the BSP includes drivers for handling camera modules, display, touch screen, audio, communication protocols, and even file systems when relevant. It simplifies application development by providing standardized interfaces, allowing developers to focus on their application's logic while abstracting low-level hardware intricacies. STMicroelectronics typically tailors BSPs to specific STM32 microcontroller families and development boards, facilitating the development process for users of these platforms.

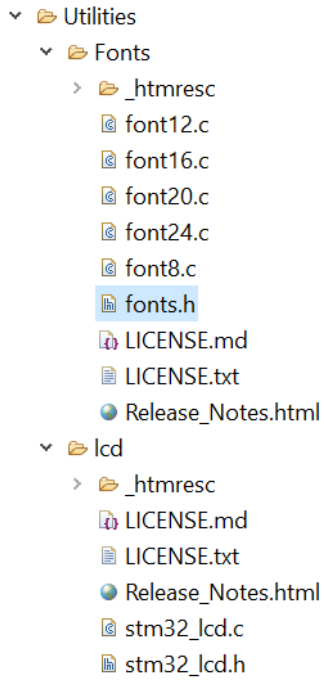

Figure 3.12: Utilities Packages

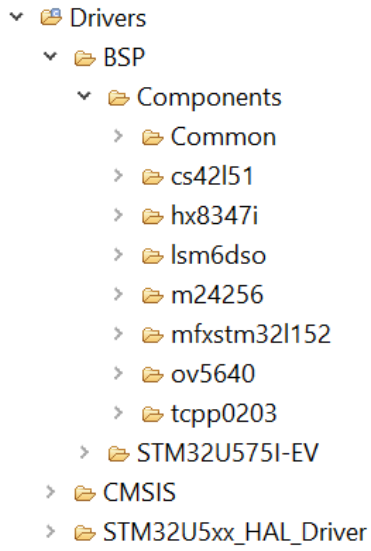

Figure 3.13: BSP Packages

After that, we need to add the packages path to the building paths of the STM32CubeIDE

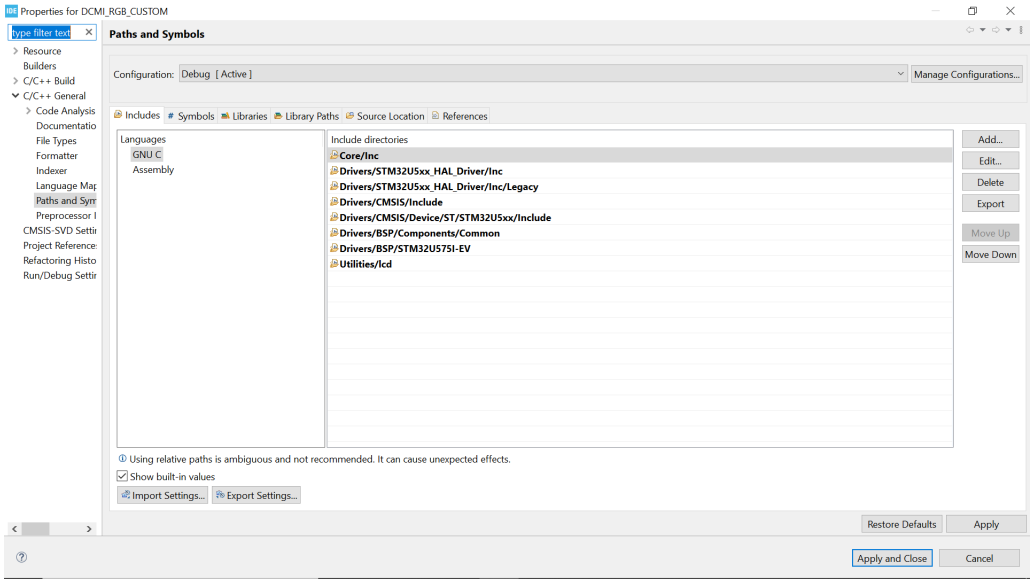

Figure 3.14: variables implementation in code.

Variables initialization

Before we start to dive into the application we must declare certain variables.These variables are crucial for managing and controlling the behavior of the code, particularly when interfacing with hardware peripherals like the camera (DCMI).

```
/* Private variables ---------------------------------------------
DCMI_HandleTypeDef hdcmi;

DMA_HandleTypeDef handle_GPDMA1_Channel12;

SRAM_HandleTypeDef hsram1;

/* USER CODE BEGIN PV */
__IO uint32_t frame_suspended   = 0;
__IO uint32_t frame_captured    = 0;
uint32_t FRAME_BUFFER_SIZE = (FRAME_WIDTH*FRAME_HEIGHT*2)/4;
uint32_t CAMERA_FRAME_BUFFER[(FRAME_WIDTH*FRAME_HEIGHT*2)/4];
OV5640_SyncCodes_t pSyncroCodes;
DCMI_SyncUnmaskTypeDef SyncUnmask;

extern DMA_QListTypeDef  DCMIQueue;
__IO FlagStatus UserButtonPressed = RESET;
/* USER CODE END PV */
```

Figure 3.15: variables implementation in code.

1. **hdcmi**: This structure is crucial for managing the configuration and control of the DCMI (Digital Camera Interface) peripheral. It plays a central role in interfacing with digital cameras or image sensors, ensuring that the image data is captured and processed effectively.

2. **handle_GPDMA1_Channel12**: This structure is essential for handling the DMA (Direct Memory Access) channel associated with GPDMA1, allowing for efficient data transfers between memory and various peripherals. In this case, it specifically refers to DMA Channel 12 in GPDMA1, which is responsible for data transfer operations.

3. **hsram1**: This structure plays a key role in configuring and controlling an SRAM (Static Random-Access Memory) peripheral. SRAM serves as a crucial memory resource, often used for storing data and program code, and this structure ensures its proper management.

4. **frame_suspended**: This variable serves as an indicator of the suspension state of frame capture. It is essential for controlling the flow of image capture

operations, allowing for the suspension and resumption of frame capture as needed.

5. **frame_captured**: This variable is instrumental in tracking whether a frame has been successfully captured. It enables the code to detect when a frame is ready for processing and indicates when the capture process has been completed.

6. **FRAME_BUFFER_SIZE**: The size of the frame buffer, calculated and stored in this variable, is critical for ensuring that there is enough memory allocated to store the captured image data efficiently.

7. **CAMERA_FRAME_BUFFER**[ ]: This array is the designated location for storing image data captured by the camera. It is a crucial data structure for collecting and processing images, facilitating further image analysis and display.

8. **pSyncroCodes**: As an essential configuration variable, this structure holds synchronization codes and settings specific to the OV5640 camera. It ensures that the camera operates with the desired synchronization parameters.

9. **SyncUnmask**: This structure is responsible for configuring settings related to the synchronization and unmasking of the DCMI peripheral. It plays a significant role in ensuring the proper synchronization of image data.

10. **DCMIQueue**: This structure likely defines a data structure or queue specialized for DMA configuration. It aids in the management and control of data transfers between the camera and memory, facilitating efficient data flow.

11. **UserButtonPressed**: This variable, defined as an enumeration with values like `SET` and `RESET`, is pivotal for tracking the status of a user button press. It enables the code to respond to user input effectively, allowing the system to wait for and detect button presses, which can trigger various actions or interruptions.

RGB Image Display Application

This application's primary purpose is to capture and display video frames obtained from an OV5640 camera module. The initialization phase sets up various hardware peripherals, including the System Clock, System Power, GPIOs, DMA, DCMI (Digital Camera Interface), and FMC (Flexible Memory Controller). Additionally, it configures an LCD display to visualize the captured video frames. The

OV5640 camera module is appropriately configured for operation, specifically in QVGA (Quarter VGA) mode, with parameters set for image capture.

User interaction is an integral part of this application. The code waits for a user button press to commence the continuous capture of video frames. The user can press the button to either suspend or resume this capture operation, with the state controlled by the frame_suspended variable.

As frames are continuously captured from the camera, they are stored in the CAMERA_FRAME_BUFFER. These frames are subsequently displayed on the LCD screen, utilizing the DMA for efficient data transfer. LEDs provide visual feedback to the user: LED5 is toggled when frames are actively being captured, and LED6 may serve other purposes as needed. The code operates in an infinite loop, continually displaying frames and monitoring user input to suspend or resume frame capture, making it suitable for applications involving video streaming or computer vision on embedded systems.

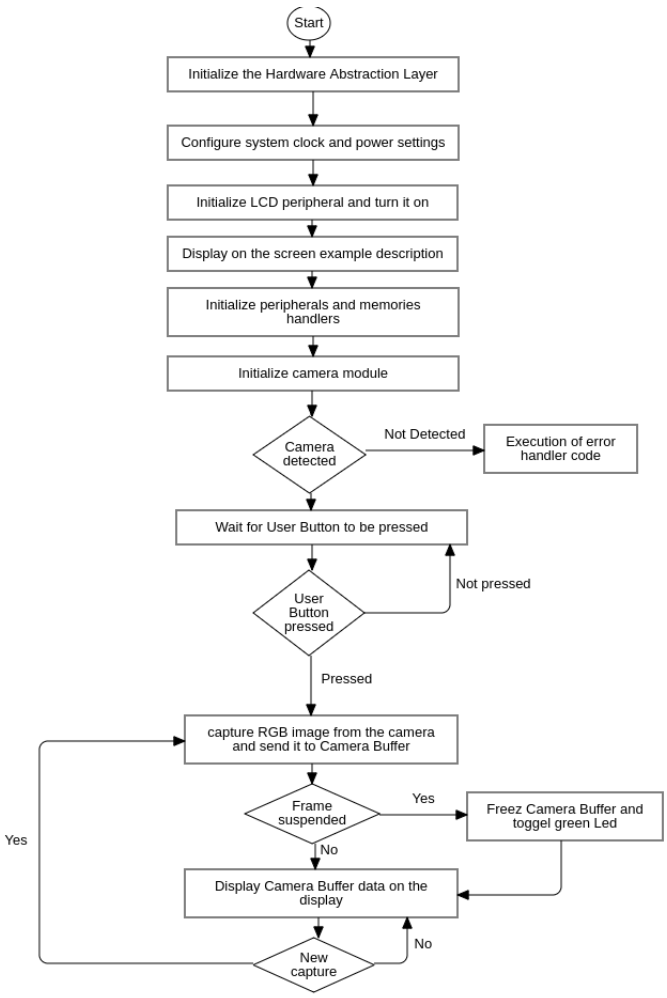

Figure 3.16: Flowchart Diagram for the RGB-application realized using StarUML.

YCrCb Image Display Application

This application toggle the same objectives as the RGB application with an added feature: Processing images in the YCrCb color format.
First of all, we need an algorithm that make the conversion of RGB image to an YCrCb image that follows this mathematical equation:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad \begin{matrix} Y \in [0, 255] \\ C_b \in [0, 255] \\ C_r \in [0, 255] \end{matrix}$$

Figure 3.17: Conversion from RGB to YCbCr .

To do that, we start by making a conversion on image data stored in the `CAMERA_FRAME_BUFFER`. The process begins by defining the `pixelCount` variable, which calculates the number of pixels in the image, assuming a pixel format of RGB565, where each pixel is represented by 16 bits. It also initializes two pointers, `rgbData` and `ycrcbData`, both pointing to the same memory location, which allows the code to reuse the same buffer for storing the YCrCb data, saving memory.

The conversion itself is accomplished within a `for` loop that iterates through each pixel in the image. For each pixel, the code extracts the red (`r`), green (`g`), and blue (`b`) color components from the RGB565 data. It then computes the luminance (`Y`), chroma red (`Cr`), and chroma blue (`Cb`) values based on these RGB components using the standard formulas for the YCrCb color space conversion. The luminance (`Y`) is calculated as a weighted sum of the RGB components, while the chroma red (`Cr`) and chroma blue (`Cb`) are derived from the differences between the original color values and the luminance. Finally, the converted YCrCb data is stored back in the `ycrcbData` buffer, with proper bit shifting to maintain the correct format. This conversion is essential in various image and video processing applications, as it separates image brightness and color information, making it easier to perform tasks such as image enhancement, color correction, and compression while preserving perceptual image quality.

```
uint32_t pixelCount = FRAME_BUFFER_SIZE / 2;  // Assuming RGB565 pixel format
uint16_t *rgbData = (uint16_t *)CAMERA_FRAME_BUFFER;
uint16_t *ycrcbData = (uint16_t *)CAMERA_FRAME_BUFFER;  // Reuse the same buffer for YCrCb data



/* Perform RGB to YCrCb conversion */
for (uint32_t i = 0; i < pixelCount; ++i)
{
    uint16_t rgbPixel = rgbData[i];

    uint8_t r = (rgbPixel >> 11) & 0x1F;
    uint8_t g = (rgbPixel >> 5) & 0x3F;
    uint8_t b = rgbPixel & 0x1F;

    uint8_t y = (uint8_t)(0.299 * r + 0.587 * g + 0.114 * b);
    uint8_t cr = (uint8_t)(128 + 0.564 * (b - y));
    uint8_t cb = (uint8_t)(128 + 0.713 * (r - y));

    ycrcbData[i] = (y << 8) | (cr << 3) | (cb >> 3);
}
```

Figure 3.18: Code implementation for RGB to YCrCb conversion.

As shown in the following diagram, the application is no different from the previous one.

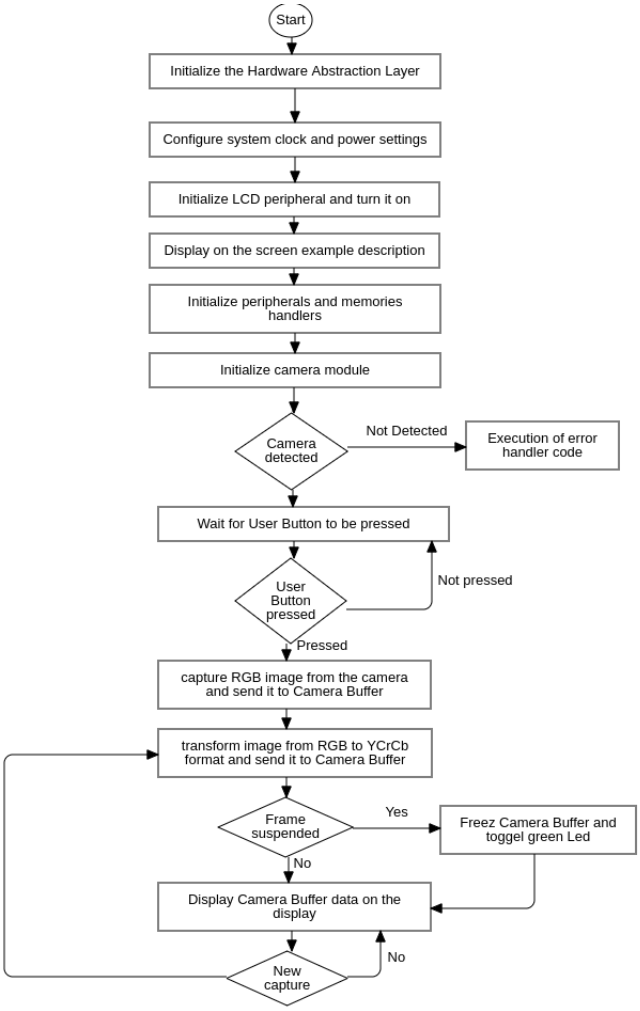

Figure 3.19: Flowchart Diagram for the YCrCb-application realized using StarUML.

Y-Only Image Display Application

The "Y-Only Image Display Application" could be considered as a sub-application to the "YCrCb Image Display Application" as in this application we are going only to show the Y variable only which is commonly used when only grayscale or brightness information is needed from an RGB image.
To assure this type of conversion we implement the following code:

```
uint16_t *rgbData = (uint16_t *)CAMERA_FRAME_BUFFER;
  uint8_t *yData = (uint8_t *)CAMERA_FRAME_BUFFER;   // Reuse the same buffer for YCrCb data
    for (uint16_t y = 0; y < FRAME_HEIGHT; y++)
    {
       for (uint16_t x = 0; x < FRAME_WIDTH; x++)
       {
           // Calculate the index for the current pixel in the RGB image data array
           uint32_t pixelIndex = (y * FRAME_WIDTH + x) * 3;
           // Get the RGB components of the pixel
           uint8_t r = rgbData[pixelIndex];
           uint8_t g = rgbData[pixelIndex + 1];
           uint8_t b = rgbData[pixelIndex + 2];
           // Convert RGB to YCbCr
           uint8_t yCbCr[3];
           yCbCr[0] = (uint8_t)(0.299 * r + 0.587 * g + 0.114 * b);
           yCbCr[1] = (uint8_t)(-0.1687 * r - 0.3313 * g + 0.5 * b + 128);
           yCbCr[2] = (uint8_t)(0.5 * r - 0.4187 * g - 0.0813 * b + 128);
           // Extract the Y component
           uint8_t yValue = yCbCr[0];
           // Store the Y component in the grayscale image data array
           uint32_t grayIndex = y * FRAME_WIDTH + x;
           yData[grayIndex] = yValue;
       }
    }
```

Figure 3.20: Code implementation for RGB to Y-Only conversion.

The ‘pixelCount‘ variable is calculated based on the size of the image buffer, assuming an RGB565 pixel format, where each pixel is represented by 16 bits. Two pointers, ‘rgbData‘ and ‘ycrcbData‘, are initialized to point to the same memory location, allowing the code to reuse the same buffer for storing the Y (luminance) data, which conserves memory.

Within a ‘for‘ loop that iterates through each pixel in the image, the code extracts the red (r), green (g), and blue (b) color components from the RGB565 data. It then calculates the luminance (Y) for each pixel using the standard formula:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

The resulting luminance (Y) values are then stored back in the ‘ycrcbData‘ buffer. This Y-Only conversion is useful when you need grayscale information from an RGB image and don’t require the full YCrCb color space with chroma components. It’s a lightweight way to extract brightness data for various image processing tasks, such as edge detection or grayscale image display.

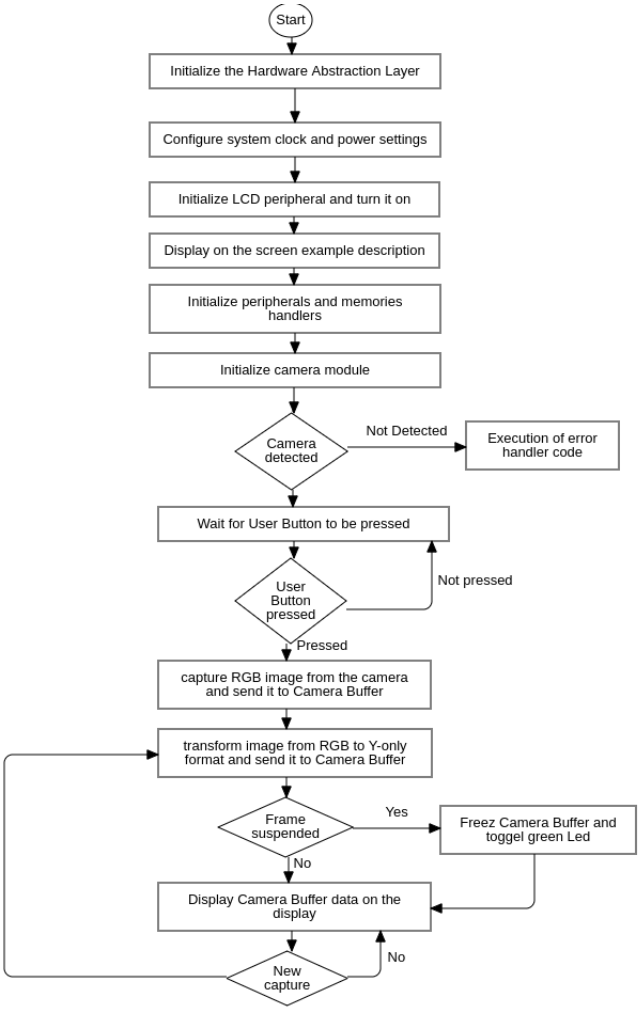

Figure 3.21: Flowchart Diagram for the YCrCb-application realized using StarUML.

As shown in this diagram, the application is no different from the previous two.

3.3 PSSI Use Case application

Effective communication between a master device and one or more slave devices is a foundational element of modern embedded systems. In the context of the STM32U5 microcontroller, the choice of communication protocol is pivotal to achieving the desired functionality and efficiency. This chapter explores the three primary communication interfaces available for slave-master communication: SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and PSSI (Parallel Synchronous Slave Interface).

To facilitate this understanding, I present a comparison table outlining the strengths and use cases of SPI, I2C, and PSSI on the STM32U5.

Aspect	**SPI**	**I2C**	**PSSI**
Data Rate	High (up to several MHz)	Moderate (up to 400 kHz)	Configurable, suitable for high-speed applications
Number of Devices	Typically 1 master, multiple slaves	Multiple masters and slaves	Multiple masters and slaves
Hardware Pins	SCLK, MISO, MOSI, CS	SCL, SDA	Configurable pins, flexible connection
Data Transfer Direction	Full duplex	Half duplex	Full duplex
Addressing	Typically uses CS for selection	7 or 10-bit addressing	Variable addressing, flexible selection
Error Handling	No built-in error detection	Error flags and detection	Error flags and detection, CRC support
Use Cases	High-speed data transfer (e.g., sensors, displays)	Multi-device communication, sensors, EEPROM	High-speed data transfer, real-time processing

Table 3.1: Comparison of SPI, I2C, and PSSI on STM32U5 for Slave-Master Communication

3.3.1 Project Objective

Efficient master-slave communication is the cornerstone of many embedded systems, enabling seamless data exchange. The STM32U5 microcontroller offers a powerful solution by integrating the Parallel Synchronous Slave Interface (PSSI) with the General-Purpose Direct Memory Access (GPDMA) controller.

The PSSI is a synchronous 8/16-bit parallel data input/output slave interface that facilitates rapid data transmission between a transmitter and a receiver. It equips the transmitter with a data valid signal, signaling when the data is ready for transmission, and enables the receiver to output a flow control signal, indicating its readiness to sample the data. This parallel interface can be configured as 8 or 16 bits and boasts an integrated FIFO capable of storing up to 8 words. For efficient flow control, two vital signals come into play: Data Enable and Data Ready.

This project explores the integration of PSSI and GPDMA on the STM32U5 microcontroller to unlock real-time data exchange for a wide range of applications.

3.3.2 Hardware Configuration

For the purpose of establishing master-slave communication using the Parallel Synchronous Slave Interface (PSSI), two STM32U575ZI-Q Nucleo boards will be employed in this project. The PSSI interface will serve as the communication channel between the master and slave devices. Each board will be configured with PSSI and connected accordingly. The master board will be responsible for initiating and controlling the communication process, while the slave board will listen for incoming data and respond as per the communication protocol.
It is important to note that the PSSI and the DCMI use some shared pins as well as the same memory bus to communicate with the GPDMA which is the "AHB2". As a result, we can not use both these peripherals at the same time.

Project Initialization

The STM32 development environment offer many ways to configure peripherals and allocate pins and processor registers, yet the most easy and more used is using the STM32CubeMX for such task.

Firstly, we initialize a project using board selecter and you select the right bord in this case the STM32U575ZI-Q_Nucleo borad

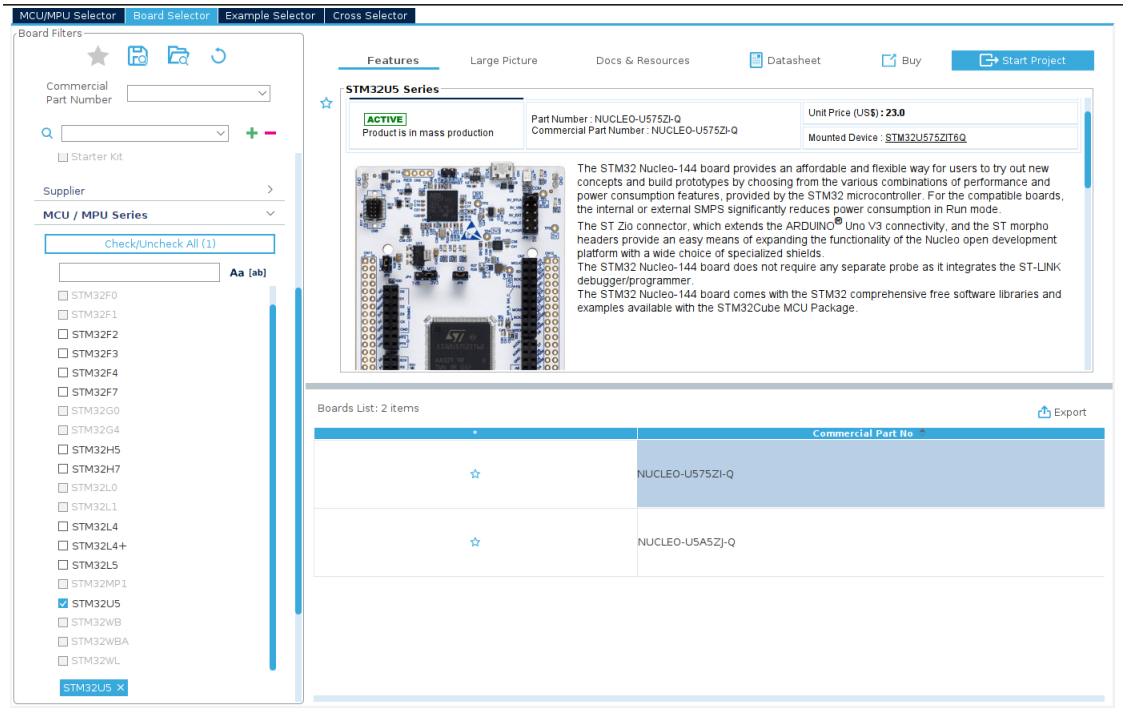

Figure 3.22: PSSI project initialization using STM32CubeMX.

PSSI Pins and Peripheral Configuration

After initializing the project, we select "PSSI" in the "Multimedia" section where you well be met with the graphical interface to configure the PSSI peripheral settings.

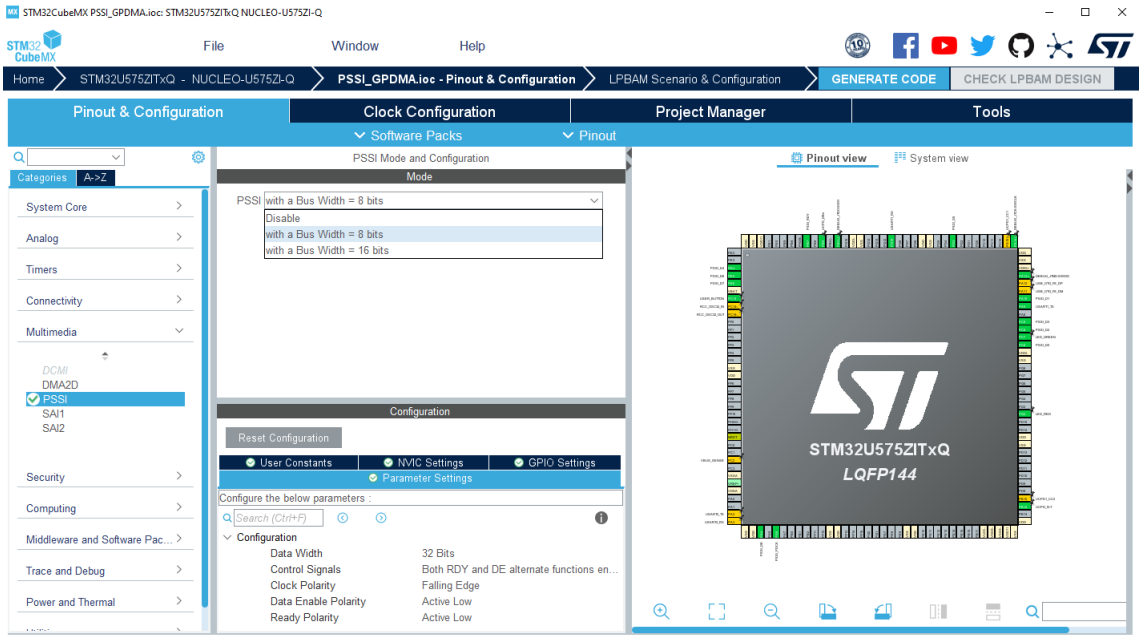

Figure 3.23: PSSI peripheral configuration using STM32CubeMX.

The PSSI is being configured, with two modes selectable in a dropdown menu such as:

- with a BUS Width = 8 bits
- with a BUS Width = 16 bits

In this project we are going with "with a BUS Width = 8 bits" configuration.
In terms of DCMI configuration, the following parameters are visible:

Data Width: Refers to the number of bits transferred in parallel during each data transfer cycle. It determines how many bits are sent or received simultaneously.

Control Signals: Refer to the additional signals used alongside data lines to control and manage data transfer operations. These signals include read/write control, chip select, and other signals that control the flow and synchronization of data.

Clock Polarity: Determine the state (high or low) of the clock signal when it is inactive or idle. It specifies the clock signal's initial and resting state.

This configuration must met the ones mentioned in **Figure 3.23** to ensure the correct performing of the project.

GPDMA Configuration

After configuring the PSSI peripheral, we must configure the GPDMA to properly handling the data from the DCMI and transport it to the MPU of the STM32U575I-EV.
To do that we need firstly to select a channel and configured. A channel 0 to 11 can be allocated,unless a channel 12 to 15 is free/unallocated and the required performances in terms of bandwidth are significant for this AHB peripheral. A channel 12 to 15 may be also allocated if the accessed memory is external.
For our project, we are going with channel 12 and 13 to ensure the best performance possible as well as to take advantage of the 2D addressing feature.[4]

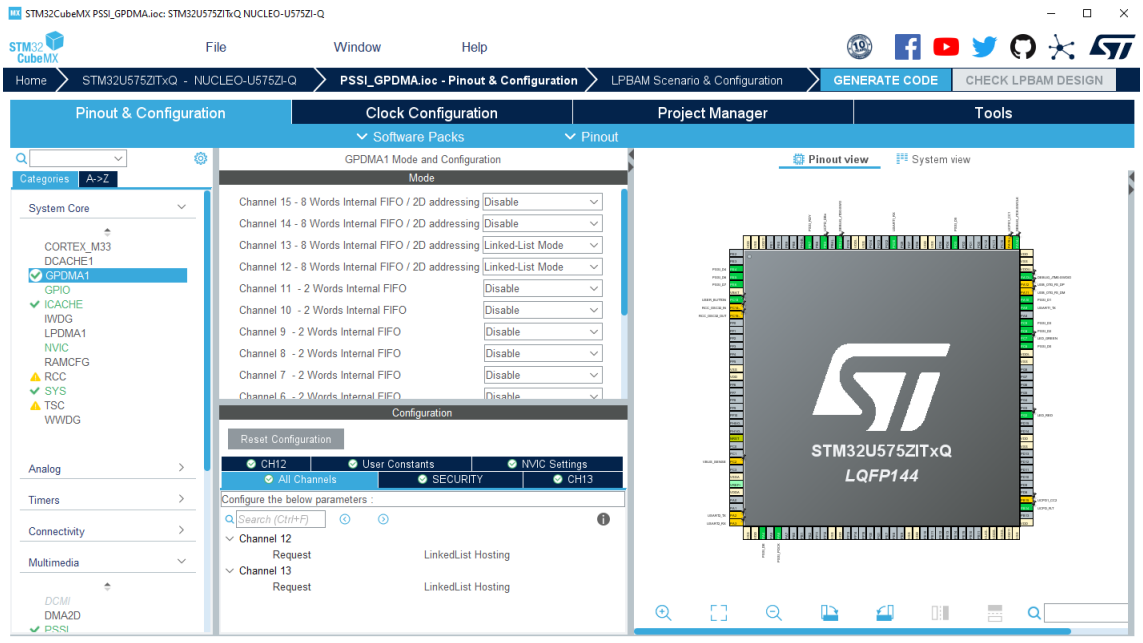

Figure 3.24: GPDMA channel configuration using STM32CubeMX.

Then we create two linked lists that are going to store the data and handle the transfer of data between the PSSI and GPDMA.One for Transmission functionality and the other for Receiving functionality.

The configuration is shown as follow in these figures.

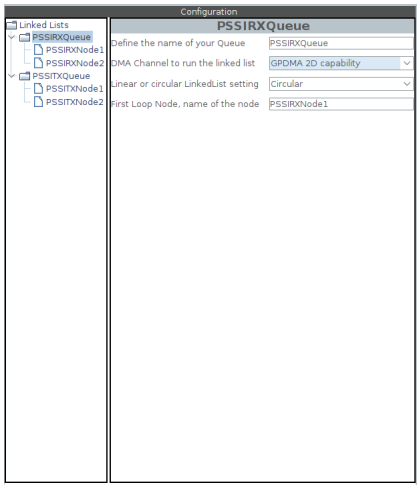

Figure 3.25: linked list initialization

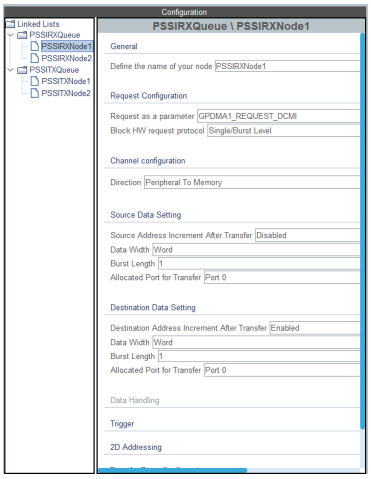

Figure 3.26: First node : GPDMA to PSSI

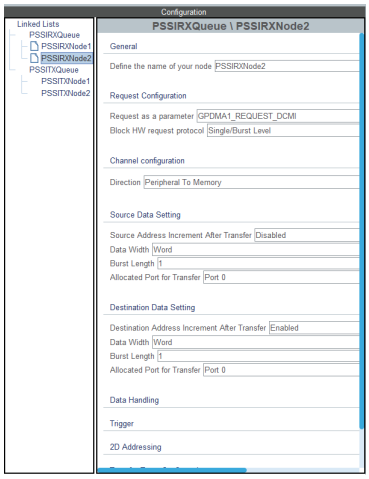

Figure 3.27: Second node : PSSI to GPDMA

Figure 3.28: Receiving LinkedList configuration using STM32CubeMX

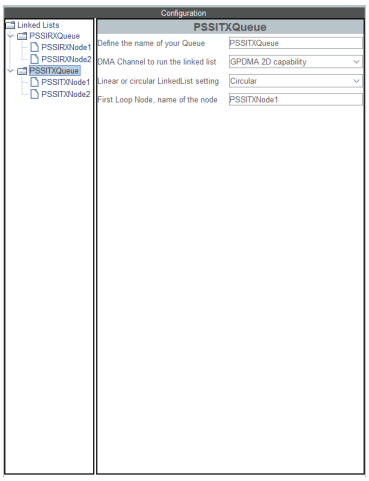

Figure 3.29: linked list initialization

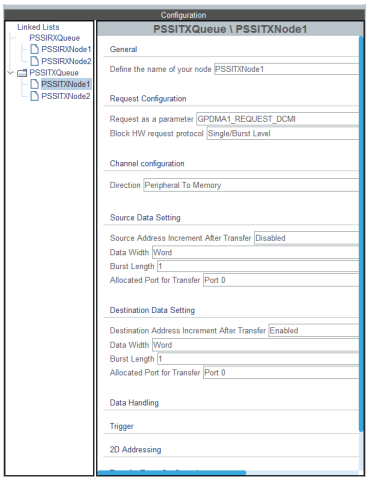

Figure 3.30: First node : GPDMA to PSSI

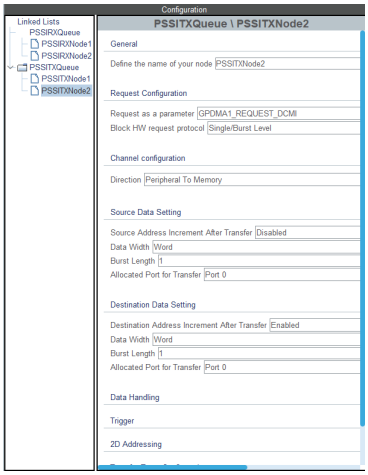

Figure 3.31: Second node : PSSI to GPDMA

Figure 3.32: Transmission LinkedList configuration using STM32CubeMX

USART Configuration

To be able to use the "printf" function and display data on the console we need to establish a serial communication between the board and the targeted console in our machine.
To do that, we need to configure "USART1" being the one linked to the ST-LINK directly.This process is quite straight forward. In the "Pinout & Configuration" tab, locate the USART1 peripheral and select the appropriate pins for its Tx (transmit)

and Rx (receive) functions. Ensure the UART mode is chosen, and configure it as follows:

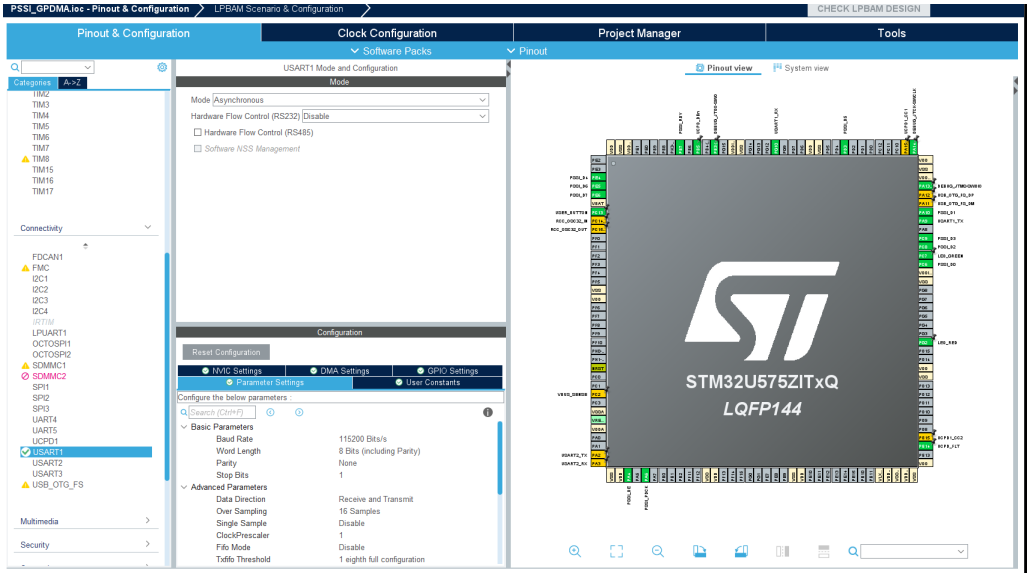

Figure 3.33: USART1 configuration using STM32CubeMX.

Code Generation

After finishing the initialization of all peripherals, we save the project and select one of the proposed IDEs (In our case, we are going with the STM32CubeIDE) to continue the application development as the STM32CubeMX offers the possibility to generate the necessary code for the mapped pins and the necessary settings of each peripheral

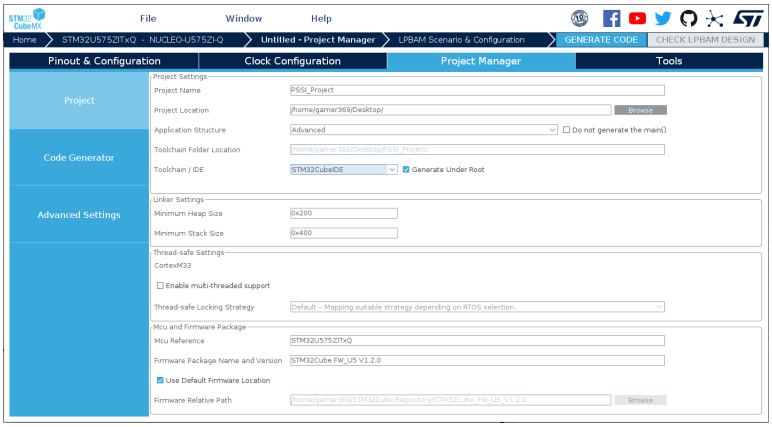

Figure 3.34: Code generation using STM32CubeMX.

3.3.3 Software Implementation

Variables initialization

Before we start to dive into the project, we must declare certain variables. These variables are crucial for managing and controlling the behavior of the code, particularly when interfacing with hardware peripherals like the PSSI in the context of a master-slave application.

1. **uhPrescalerValue**: This variable represents the prescaler value. In a master-slave application using PSSI, it may be used to adjust the clock frequency or synchronization timing.

2. **Request_received**: This variable indicates whether a request has been received in the context of a master-slave communication. It can serve as a flag to trigger actions upon receiving requests.

3. **data_cmp**: This variable is used for data comparison purposes. It may be involved in verifying the integrity or correctness of received data.

4. **PSSI_HAL_PSSI_TransmitComplete_count**: This variable keeps track of the count of completed transmit operations in the PSSI (Parallel Synchronous Serial Interface) communication. It helps monitor successful data transmission.

5. **PSSI_HAL_PSSI_ReceiveComplete_count**: Similar to the previous variable, this one counts completed receive operations in the PSSI communication, providing visibility into received data.

6. **PSSI_HAL_PSSI_ErrorCallback_count**: This variable is used to count the occurrences of error callbacks in the PSSI communication. It is valuable for error handling and diagnostics.

7. **pData8_S_TRSMT[64]** (Slave): Data to transmit from the slave device. This variable holds the data that the slave device intends to send to the master. It is aligned to 32 bytes for efficient memory access.

8. **pData8_S_RCV[64]** (Slave): Data received by the slave device. This variable stores data received by the slave from the master. It is aligned to 32 bytes for efficient memory access.

9. **pData8_M_RCV[64]** (Master): Data received by the master device. This variable holds data received by the master from the slave. It is aligned to 32 bytes for efficient memory access.

10. **pData8_M_TRSMT[64]** (Master): Data to transmit from the master device. This variable contains data that the master intends to send to the slave. It is aligned to 32 bytes for efficient memory access.

```
uint32_t uhPrescalerValue = 0;
uint32_t Request_received = 0 ;
uint32_t data_cmp = 0 ;
uint32_t i;
uint32_t PSSI_HAL_PSSI_TransmitComplete_count = 0;
uint32_t PSSI_HAL_PSSI_ReceiveComplete_count = 0;
uint32_t PSSI_HAL_PSSI_ErrorCallback_count = 0;

#ifndef MASTER_BOARD
//ALIGN_32BYTES (char     pData8_S_TRSMT[64] ="Hello from Slave");  /* Data to transmit from Slave */
#define ALIGN_32BYTES __attribute__((aligned(32)))
char pData8_S_TRSMT[64] ALIGN_32BYTES = "Hello from Slave";
ALIGN_32BYTES char pData8_S_RCV[64];
#else
ALIGN_32BYTES (char     pData8_M_RCV[64]);
ALIGN_32BYTES (char     pData8_M_TRSMT[64]); /* Data to transmit from Master */
#endif
/* USER CODE END PV */
```

Figure 3.35: variables implementation in code.

Master application

The operation of the master board is initiated with an initialization phase, signifying its identity as the master.
Additionally, a clock signal is configured to ensure synchronization in the data transfer process. Within a continuous operation loop, the master board monitors the receipt of requests from the slave. The `Request_received` flag signifies the arrival of a request, while the GPDMA capabilities facilitate data reception into the `pData8_M_RCV` buffer.
Upon receiving data, it is scrutinized for specific requests, and if a request is detected, the `Request_received` flag is set. Data transmission to the slave is executed through DMA, utilizing the `pData8_M_TRSMT` buffer. Successful transmission is validated when the `PSSI_HAL_PSSI_TransmitComplete_count` reaches 1.
Following successful transmission, a "Transmission completed" message is displayed, with a brief delay to ensure stability. A subsequent delay precedes the reset of the `Request_received` flag, indicating readiness for new requests. To optimize power and minimize interference during idle periods, the clock signal is deactivated with `CLK_Off()`.
This process repeats in a loop, ensuring efficient and synchronized communication between the master and slave devices, enabling effective data exchange in this project.

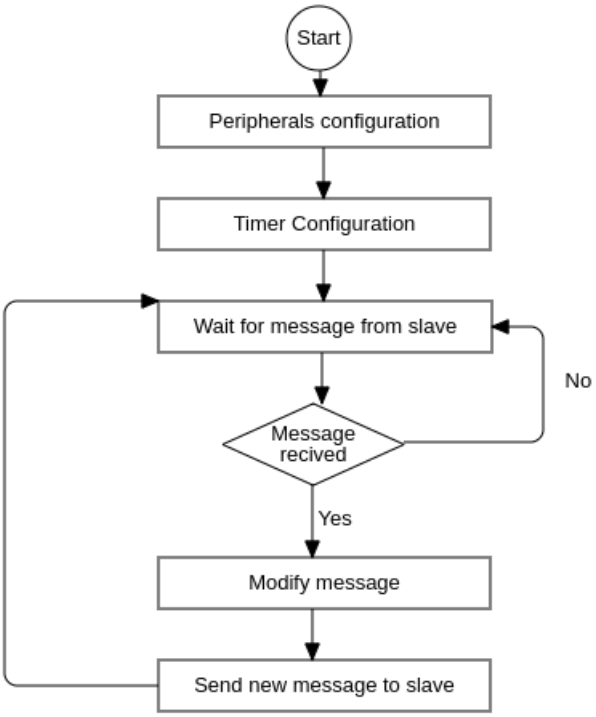

Figure 3.36: Flowchart Diagram for the Master application realized using StarUML.

Slave application

The operation of the slave board is data transmission and reception through the PSSI. It commences with introducing the board's identity to the communication network. Following a brief delay for initialization, data transmission is initiated with the help of the `HAL_PSSI_Transmit_DMA` function. This function leverages the PSSI's DMA capabilities to transmit data from the `pData8_S_TRSMT` buffer. The code then enters a waiting phase, holding off until the `PSSI_HAL_PSSI_TransmitComplete_count` reaches 1, signaling the successful completion of data transmission.
After a short delay, the code proceeds to receive data from the master device using the `HAL_PSSI_Receive_DMA` function. A similar waiting phase ensues, with the code monitoring the `PSSI_HAL_PSSI_ReceiveComplete_count` to verify the successful reception of data.
Following the completion of data reception, a validation process is initiated to ensure data integrity. The code iterates through the received data, comparing it to the transmitted data character by character. The `data_cmp` variable keeps track of any discrepancies, incrementing when differences are encountered. If the

transmitted and received data match without any discrepancies and the received data includes the string "Master," indicating the master board's modification, the code enters an infinite loop. Within this loop, it prints a "data has been modified" message, displays the received data, and introduces a delay of 1 second between iterations. If data integrity is not maintained or the modification indicator is absent, the code invokes the `Error_Handler()`.

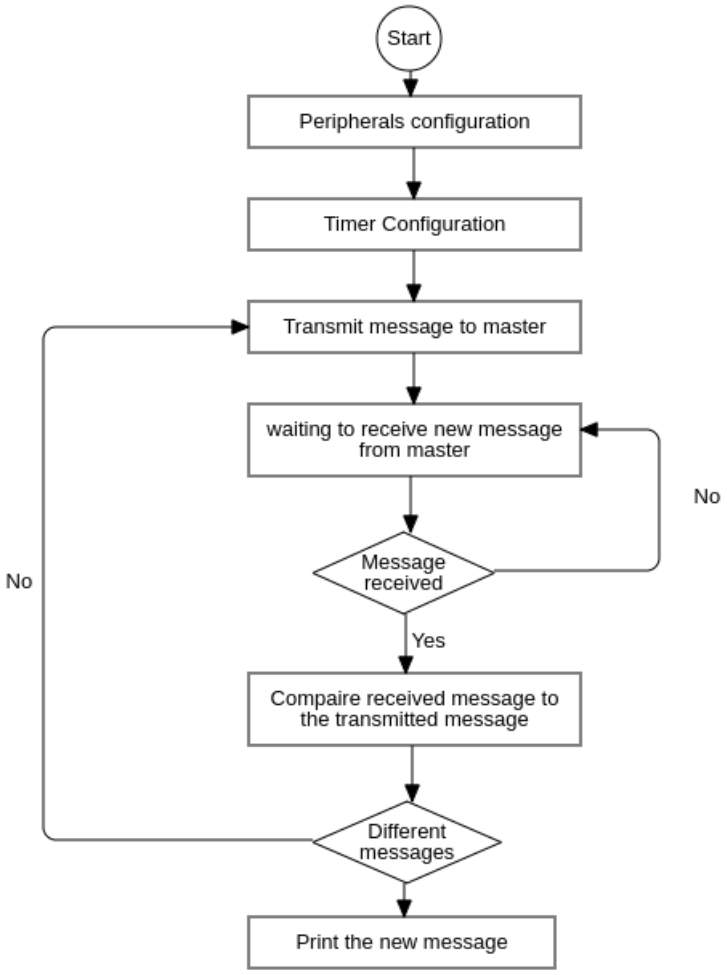

Figure 3.37: Flowchart Diagram for the Slave application realized using StarUML.

3.4 Project Deployment

After finishing the development of my projects and sharing my code with STMicroelectronics for an easy supervision, I have created a GitHub repository where I have deployed successfully my project using GitHub Desktop (mentioned earlier in Chapter 2).

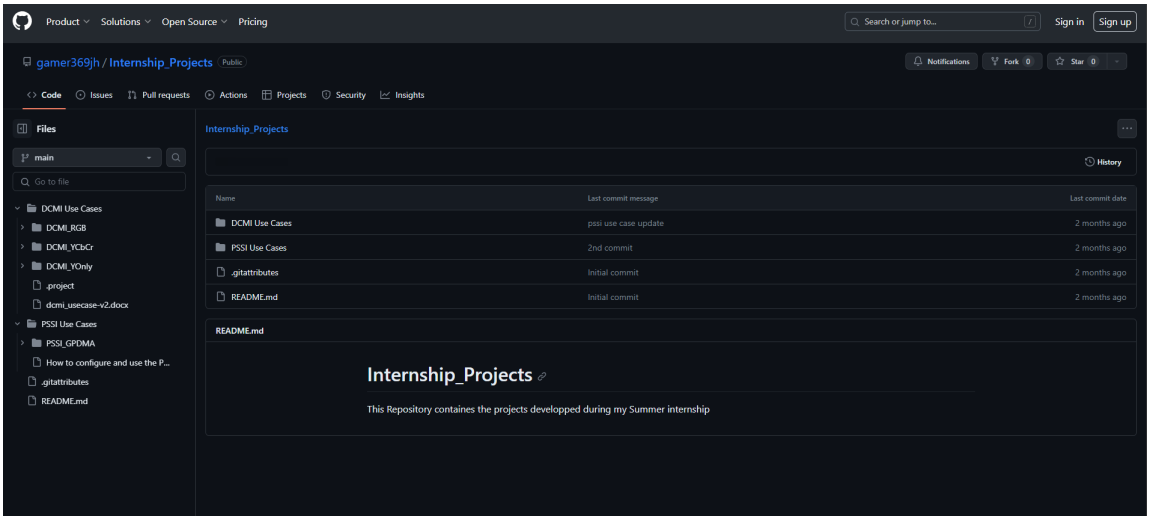

Figure 3.38: Project architecture on GitHub

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 3.39: QR code link to the GitHub repository full project.

3.5 Conclusion

In conclusion, these projects have offered me 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.

Conclusion

This project was an initiation of STMicroelectronics to showcase the potential of the STM32U5 and the GPDMA, DCMI, and PSSI peripherals. I was tasked to shed light on the incredible potential of the STM32U5 microcontroller, a relatively new but highly capable addition to the STM32 family. By exploring various use cases for the General-Purpose Direct Memory Access (GPDMA) in conjunction with the Digital Camera Interface (DCMI) and Parallel Synchronous Slave Interface (PSSI),

The STM32U5's combination of features and capabilities, including its powerful ARM Cortex-M33 core, extensive peripherals, and connectivities, make it an ideal choice for projects that demand high performance and efficiency. The integration of GPDMA with DCMI enables us to process and manipulate image data efficiently, opening doors to a multitude of computer vision applications. From object recognition to gesture detection, this combination empowers developers to create innovative solutions in image-based processing.In addition to that it offers the possibility to have parallel data communication with other boards using the PSSI to achieve faster and sustainable data transfer.

However, it's important to note that the PSSI peripheral, while promising, currently faces a challenge in terms of community support when compared to more established interfaces like SPI and UART. Despite its potential for various communication and data transfer tasks, it may require additional efforts to gain broader acceptance and widespread adoption within the STM32 community.

Nonetheless, the STM32U5's rapid evolution and increasing adoption rates suggest that the community will likely address these challenges over time, fostering a more extensive ecosystem of resources, libraries, and expertise. As developers continue to explore the capabilities of the STM32U5 and its peripherals, it is certain that this microcontroller will become an increasingly attractive option for a wide range of applications.

Bibliography

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

[2] ARM. https://developer.arm.com/Processors/Cortex-M33. Last accessed: July 20, 2023.

[3] 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: July 20, 2023.

[4] STMicroelectronics. https://www.st.com/resource/en/application_note/an5593-how-to-use-the-gpdma-for-stm32u575585-microcontrollers-stmicroelectronic pdf. Last accessed: July 21, 2023.

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

[6] STMicroelectronics. Stm32u575i$_e$*vdatasheet*.. Last accessed: July 20, 2023.

STMicroelectronics. Stm32u575zi-q datasheet. https://www.st.com/en/microcontrollers-micro stm32u575zi.html#st_circuit-diagram_sec-nav-tab. Last accessed: July 20, 2023.