Escola de Engenharia
Universidade do Minho

ESRG
EMBEDDED SYSTEMS RESEARCH GROUP

Marcelo Ribeiro PG54028

Pedro Pereira PG54155

MFS: Multi-Function Scanner

Master's in Industrial Electronic and Computer Engineering

Professor:
Dr. Adriano José Conceição Tavares

Contents

1 Introduction **10**

1.1 Introduction 10

1.2 Market Research 10

2 Analysis **11**

2.1 Problem Statement Analysis 11

2.2 System Overview 11

2.3 Requirements 11

2.4 Constrains 12

2.5 System Overview 12

2.5.1 Hardware Overview 12

2.5.2 Software Architecture 13

2.5.3 Technology Stack 14

2.5.4 State Chart 15

2.5.5 Events 16

2.5.6 Use Classes Diagram 16

2.5.7 Sequencial Diagram 17

2.5.8 Entity Relationship Diagram 19

2.5.9 Estimated Budget 19

3 Project Design **20**

3.1 Theoretical Foundations 20

3.1.1 Analysis Review 22

3.1.2 Hardware Component Specification 22

3.1.3 Raspberry Pi Model 4B 22

3.1.4 Barcode Reader 23
3.1.5 Touch Screen TFT LCD 24
3.1.6 RGB LED 25
3.1.7 Battery Holder 25
3.1.8 Temperature Sensor 26
3.1.9 Hardware Connection Diagram 27
3.1.10 Printed Components 27
3.2 Software Specification 28
3.2.1 Tools and COTS 28
3.3 GUI Layout 31
3.4 Thread Divison 36
3.4.1 Thread Description 37
3.4.2 Thread Priorities 38
3.4.3 Thread Synchronization 39
3.5 Class Diagrams 40
3.6 Flow Charts 42
3.7 Test Cases 48
3.8 Dry run test 51
3.9 Errors Handling Specification 51

4 Implementation **52**

4.1 Classes 52
4.1.1 Interfaces 52
4.1.2 Implementation 61
4.2 Device Driver 81
4.3 Buildroot 84
4.3.1 Packages 84

4.4 Final structure 87

4.5 Conclusion 88

4.6 Future Work 88

4.7 Acknowledgments 88

List of Figures

2.1 System Overview 11

2.2 Hardware Architecture 13

2.3 Software Architecture 14

2.4 Technology Stack 14

2.5 State Chart 15

2.6 Use Cases Diagram 16

2.7 Check Stock Sequence 17

2.8 Manage Stock Sequence 17

2.9 Manage Sale Sequence 18

2.10 GUI Sequence 18

2.11 Manage Sale Sequence 19

3.1 Mutex functionality 20

3.2 Device Drivers 21

3.3 Device Drivers 21

3.4 Raspberry Pi Model 4B 23

3.5 Raspberry Pi Model 4B 23

3.6 Barcode Reader 24

3.7 Touch Screen 24

3.8 RGB LED 25

3.9 Battery Holder 26

3.10 Tempeature Sensor 26

3.11 Hardware Connection Diagram 27

3.12 Printed Components 27

3.13 Printed Components 27
3.14 Printed Components 28
3.15 Buildroot Logo 28
3.16 C/C++ 29
3.17 GitHub Logo 29
3.18 Visual Studio Code Logo 29
3.19 Solid Works Logo 30
3.20 Overleaf Logo 30
3.21 Drawio Logo 30
3.22 PThreads Logo 30
3.23 SQLite Logo 31
3.24 QT Creator Logo 31
3.25 ThingSpeak Logo 31
3.26 Home Menu 32
3.27 Stock Management Menu 32
3.28 Pin Request Menu 33
3.29 Product Added Menu 33
3.30 Product Removed Menu 34
3.31 Stock Check Menu 1 34
3.32 Stock Check Menu 2 35
3.33 Sales Register Menu 1 35
3.34 Sales Register Menu 2 36
3.35 Sales Register Menu 3 36
3.36 Thread Division 37
3.37 Thread Priorities 39
3.38 Classes Diagram 40

3.39 Database Class 41

3.40 LED Class 41

3.41 Scanner Class 41

3.42 Main Thread 42

3.43 LED Thread 43

3.44 CheckTemperature Thread 44

3.45 Database Manage Thread 45

3.46 Database Daemon Thread 46

3.47 Startup FlowChart 47

3.48 Shutdown FlowChart 48

4.1 Graphical User Interface 52

4.2 Interface of Scanner Class 53

4.3 Interface of MLX90614 Class 54

4.4 Interface of rgbDevice Class 55

4.5 Interface of CProduct Class 56

4.6 Interface of CDatabaseManage Class 57

4.7 Interface of MFS Class - 1 58

4.8 Interface of MFS Class - Public Members 59

4.9 Interface of MFS Class - Private Members 60

4.10 Interface of MFS Class - Private Members 60

4.11 Interface of mfsWrapper Class 61

4.12 Implementation of the Scanner Class - 1 62

4.13 Implementation of the Scanner Class - 2 63

4.14 Implementation of the Scanner Class - 3 64

4.15 Implementation of the MLX90614 Class - 1 64

4.16 Implementation of the MLX90614 Class - 2 65

4.17 Implementation of the MLX90614 Class - 3 65

4.18 Implementation of the rgbDevice Class 66

4.19 Implementation of the CProduct Class 67

4.20 Implementation of the CProduct Class 67

4.21 Implementation of the CDatabaseManage Class - Constructors and Destructor 68

4.22 Implementation of the CDatabaseManage Class - Find product in database function . . . 68

4.23 Implementation of the CDatabaseManage Class - Initialization of database 2 70

4.24 Implementation of the CDatabaseManage Class - Add product to database method 70

4.25 Implementation of the CDatabaseManage Class - Remove product to database method . . 71

4.26 Implementation of the CDatabaseManage Class - Search for a product in the database method 71

4.27 Implementation of the CDatabaseManage Class - Sale for a product in the database method 72

4.28 Implementation of the CDatabaseManage Class - Search for a product in the database method 73

4.29 Sale's Charts - ThingSpeak 73

4.30 Implementation of class mfsWrapper 74

4.31 Implementation MFS Class - Constant Expressions 74

4.32 Implementation MFS Class - Public Methods 75

4.33 Implementation MFS Class - Public Methods 75

4.34 Implementation MFS Class - Public Methods 75

4.35 Implementation MFS Class - Public Methods 76

4.36 Implementation MFS Class - Public Methods 76

4.37 Implementation MFS Class - Setup Threads Method 77

4.38 Implementation MFS Class - Startup Sequence 77

4.39 Implementation MFS Class - tcheckTemperature thread function 78

4.40 Implementation MFS Class - tscan thread function 78

4.41 Implementation MFS Class - tLED thread function 79

4.42 Implementation MFS Class - tDatabaseManage thread function 80

4.43 Implementation MFS Class - tToGUI thread function 80

4.44 Implementation of Character Device Driver 1 81

4.45 Implementation of Character Device Driver 2 82

4.46 Implementation of Character Device Driver 3 82

4.47 Implementation of Character Device Driver 4 83

4.48 Implementation of Character Device Driver 5 83

4.49 Qt5 Buildroot Package 84

4.50 JSON Libraries 85

4.51 libcurl Library 86

4.52 sqlite Library 86

4.53 Front_view 87

4.54 Inside_view 87

List of Tables

2.1 Estimated Budget 19

3.1 Main Process 37

3.2 GUI Process 38

3.3 Daemon Process 38

3.4 Mutex 39

3.5 Message Queue 39

3.6 Condition Variable 40

3.7 Barcode Scanner Test Case 49

3.8 LED Test Cases 49

3.9 Temperature Sensor Test Case 49

3.10 Power Supply Test Case 50

3.11 GUI Test Cases 50

3.12 Database Test Cases 50

3.13 System Shutdown Test Case (Error Handling) 50

3.14 Dry run test 51

Chapter 1

Introduction

1.1 Introduction

From small retail stores to large industrial companies, businesses often struggle with efficiently managing their inventory and sales. Some stock management systems rely on manual data entry, outdated paper-based methods, or even to expensive, making it challenging for these businesses to track their products, prevent stockouts and overstocks, and effectively manage their sales. Such situations can result in lost sales and reduced productivity. To face these challenges, we aim to develop an affordable and user-friendly stock and sales management system that can contribute to the growth of these businesses. Our objective is to create a solution that is both economically viable and technologically feasible. This system will provide improved inventory accuracy, cost savings for companies, optimized resource utilization, and many other benefits.

1.2 Market Research

Our market research revealed that this type of product is not common and is typically extremely expensive. Despite all its features, the pricing can serve as a limitation for many companies. This finding prompted our project to offer a very similar service at a significantly lower cost. Below is an example of two products already available in the market. For many companies, these products can be financially unattainable. Therefore, our product will aim to provide a budget-friendly alternative without compromising on quality and functionality.

(a) Tera Digital Mobile Computer

(b) Zebra Mobile Computer

Chapter 2

Analysis

2.1 Problem Statement Analysis

This project aims to develop a wireless multi-functional barcode scanner integrated with a display to enhance data accessibility and efficiency in various industries. Employees equipped with this tool will have a user-friendly interface displayed on a screen, allowing them to add items to the stock, remove items from it, and view all the inventory information. Additionally, there will be a sales mode where the employees can process sales using the machine. The managers will also have access to a website from the database where they can access more detailed information about their stock, including graphics and other analytical tools. This data can be analyzed to make informed decisions for the economic evolution of companies.

2.2 System Overview

In the Figure 2.1, we present a high level system overview diagram illustrating the main components of this project.

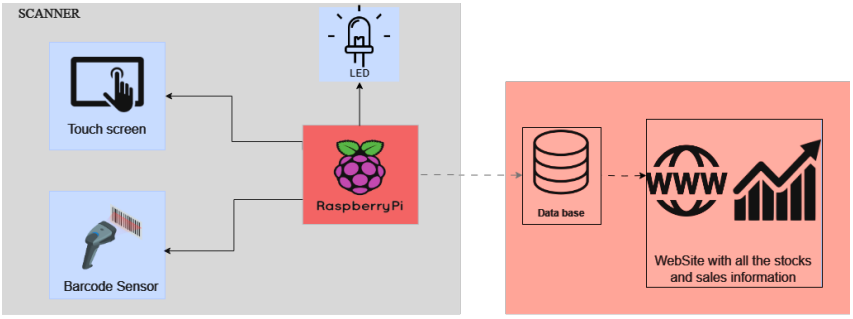

Figure 2.1: System Overview

The entire system consists of two parts: the local system and the remote system. The local system consists of the user-facing part, which include both hardware and software components located at the user's end. This will be constituted by the barcode scanner with a user-friendly interface. The remote system is composed of the database and the connected website.

2.3 Requirements

To ensure a successful and functional project, our system must meet the following Functional and Non-Functional requirements.

Functional Requirements:
- Scan the product's code;
- Ensure a reliable power supply;
- Inform the user with various warnings/notifications when necessary;

Non-Functional Requirements:
- User-friendly system;
- Scalable for future software updates;
- Structurally strong to all types of industrial environments.

2.4 Constrains

Technical Constraints:
- Use Raspberry Pi as the development board;
- Use Buildroot to generate embedded Linux systems;
- Object-oriented programming using C++;
- The system must be Real-Time;
- Implement, at least, one device driver;
- Linux OS;

Non-Technical Constraints:
- Only two students per group;
- Low-cost project;
- The project has a deadline on the end of the semester;

2.5 System Overview

2.5.1 Hardware Overview

Describing our hardware architecture can be done by outlining the key components and their interactions.
•**The Raspberry Pi 4 Model B** which serves as the core hardware platform;
•**The Barcode Reader** that is the essential input device in the system;
•**The Display Interface** is the user-facing output device;
•**The Power Module** must ensure a stable and reliable power supply to the entire system.
•**The RGB LED** will help the employee verify if the product was scanned correctly.
•**The Temperature Sensor** which informs the current temperature of the system.

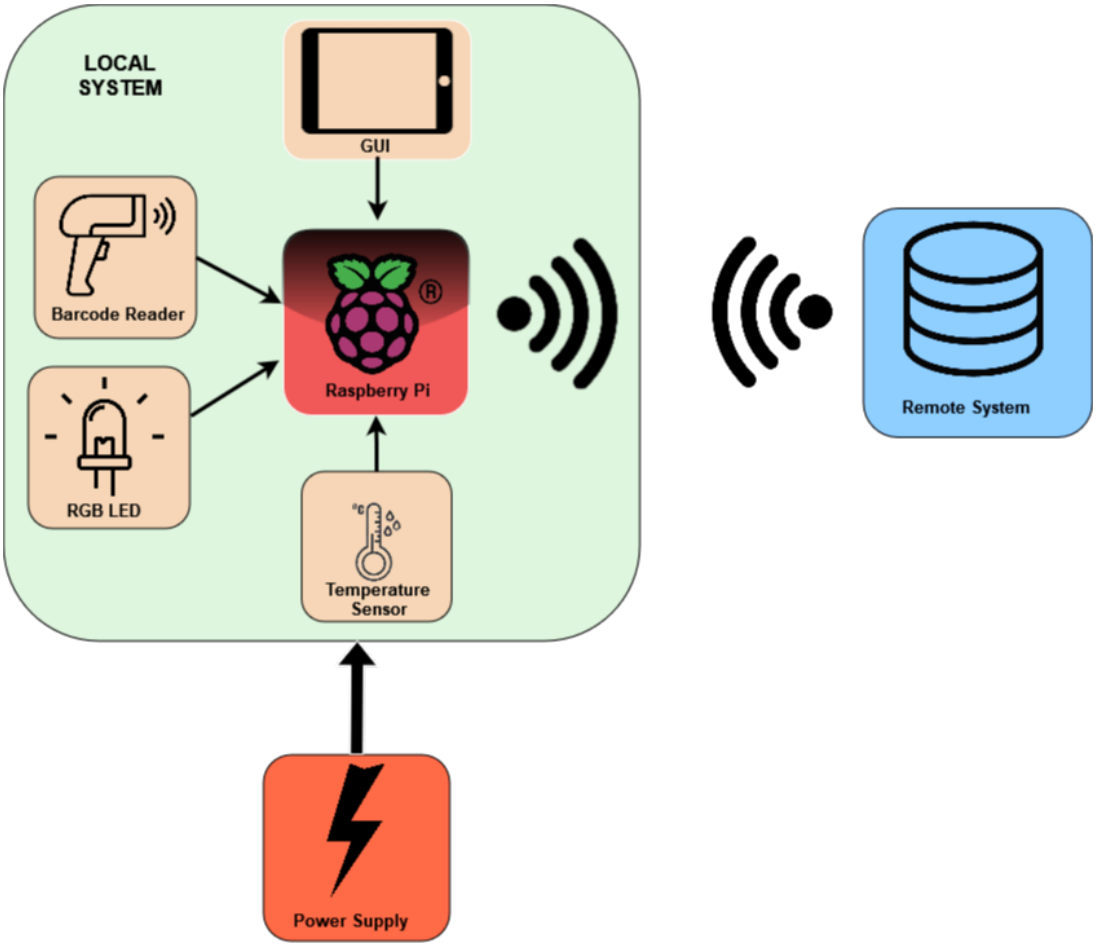

Figure 2.2: Hardware Architecture

2.5.2 Software Architecture

The next figure represents the software architecture of this system. Our software architecture is designed to make a robust and efficient system barcode scanning, sales tracking, inventory management, and data visualization. It consists of key components divided into three layers that, when combined, can accomplish the project needs.

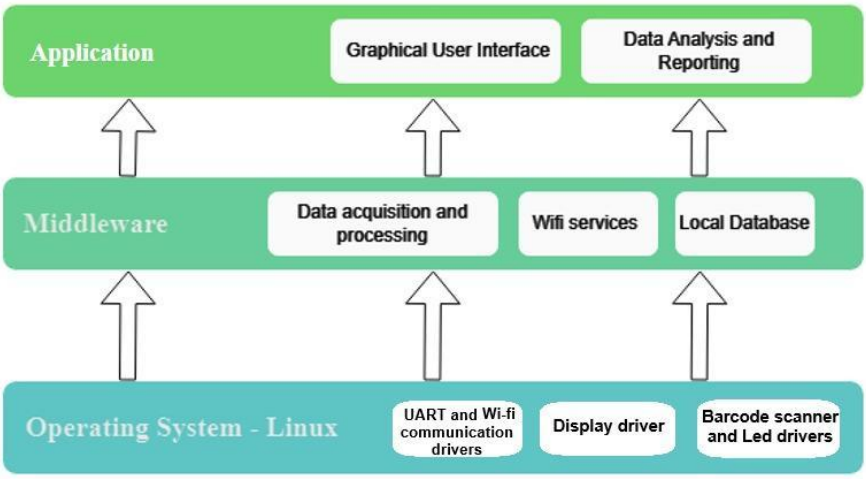

Figure 2.3: Software Architecture

2.5.3 Technology Stack

The Technology Stack is an important part of any project analysis, due to the simple but ordered Hardware and Software component specifications. In the figure below, we can visualize all the layers of our technology stack. In the bottom layer, all the hardware components used are represented. Moving to the next layer, we have the communication protocols to interface the hardware with the software. The subsequent layer comprises the Device Drivers and the Operating System, in this case, the Linux OS. Following this, we encounter the application software, where we can find the application logic and data processing using objected-oriented C++ programming language along with the respective POSIX Threads APIs. Next, we have the application layer, where the graphical user interface application can be found. Lastly, there is the Database and all the analytics tools available on the website.

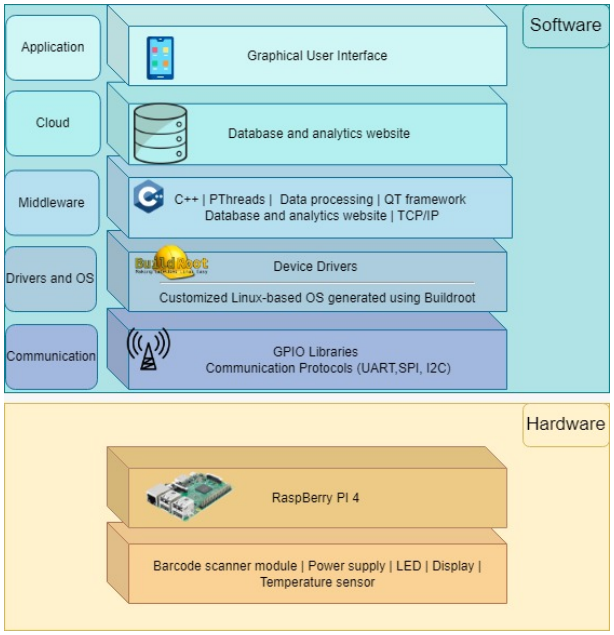

Figure 2.4: Technology Stack

2.5.4 State Chart

As depicted in the next figure, our system initializes a startup sequence after powering up or resetting. If no errors occur during this sequence, the system proceeds to the next state, which involves displaying the Graphical User Interface on our touchscreen. Upon selecting an option from the menu, the system analyzes the choice and can transition to one of three different states:

• **Stock Management**: In this state, the user can add or remove a product. After selecting the respective action, the product must be scanned. Subsequently, the database is updated, and the screen is refreshed.

• **Stock Check**: Upon selecting this option, a product must be scanned. Information is then requested from the database. Once all the necessary information is gathered, it is displayed on the screen. To return to the menu, the operator must tap on the "Return to Menu" option.

• **Sales Register**: In this state, scanning a product prompts a payment request. After the payment is confirmed, the product is removed from the database, and the sale is recorded in the database analytics tools. After choosing the operation, there is always the option "Menu" to go back and choose another operation.

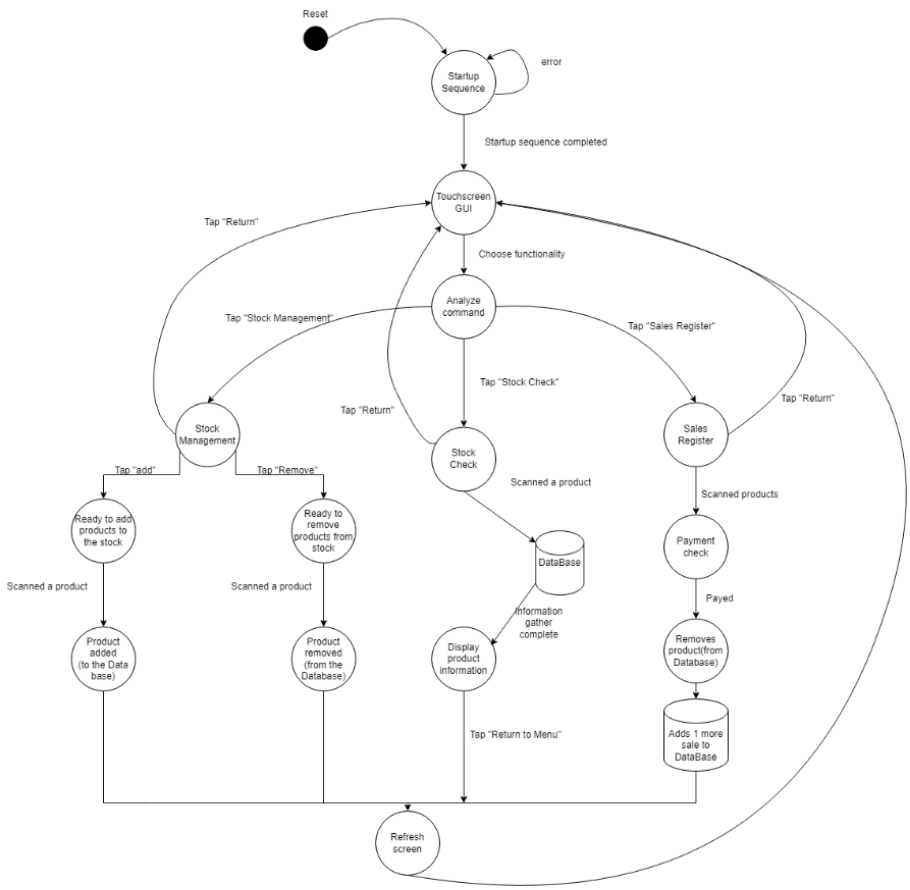

Figure 2.5: State Chart

2.5.5 Events

An event table is an efficient way to organize and demonstrate our system behavior towards an event specifying its reaction to it. In the table below we can see our project's event table:

Event	System Response	Source or Trigger	Type
Power ON	Initializes the rasp and its startup commands sequence	User	Asynchronous
Touchscreen input	Receive and process information	User	Asynchronous
Scan on SM mode	Accesses the database to add/remove a product	User	Asynchronous
Scan on SC mode	Acccesses the database to check the product's stock	User	Asynchronous
Scan on SR	Accesses the database to remove a product and regist a sale	User	Asynchronous
Any Scan	Causes the buzzer to produce a sound and a LED blink for acknowledgement	User	Asynchronous

Used Acronyms: **SM** - Stock management **SC**- Stock check **SR**- Sales Register

2.5.6 Use Classes Diagram

Use cases play a vital role in the development of embedded systems by defining how the system interacts with its environment and specifying its expected behavior in different scenarios. In Figure 2.6 it's provided the use cases diagram of the MSF system. In this system the user will be able to choose from a variety of operations such as checking the sales, check the current stock, register and proceed to the confirmation of a sale, add or remove items from the stock and, of course, turn on and off the system. The "Check Sales" option allows the user to easily review the sales history, while in the "Confirm Sale" option, he will be able to confirm the sale of an item or many items. The "Check Stock" option will allow the user to view the stock list or search for an item by its barcode/QR code. Lastly, the "Add/Remove Stock Items" option will allow the user to add or remove items from the current inventory.

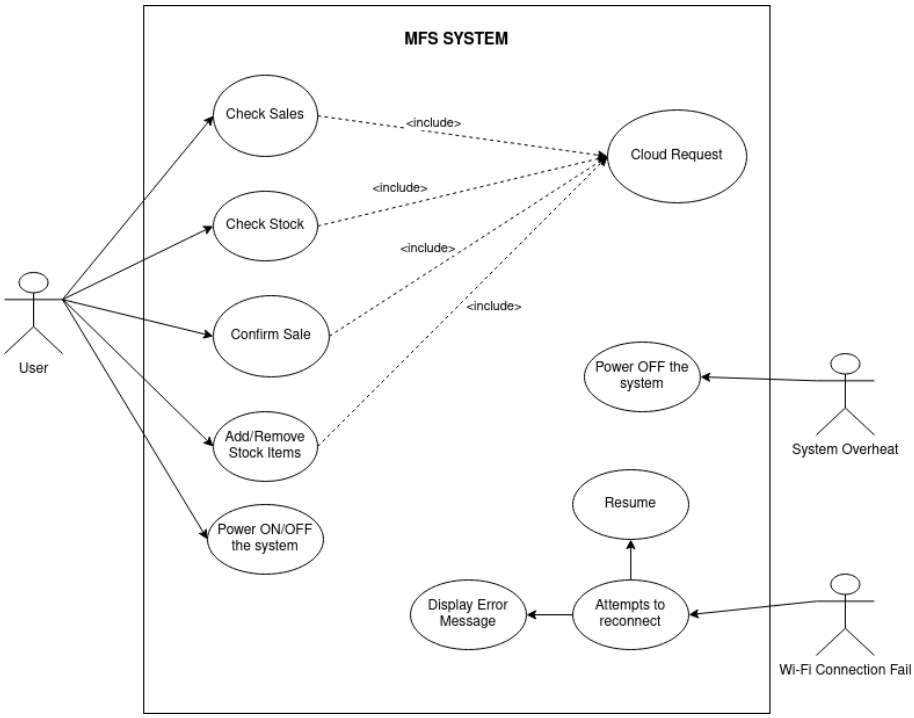

Figure 2.6: Use Cases Diagram

2.5.7 Sequencial Diagram

Sequential diagrams are a useful tool for modeling the dynamic behavior of a system. **Check Stock Sequence**: When the user scans the product's code, the system will analyze it and will search for it on the Database and on the GUI whether the product is in stock or not.

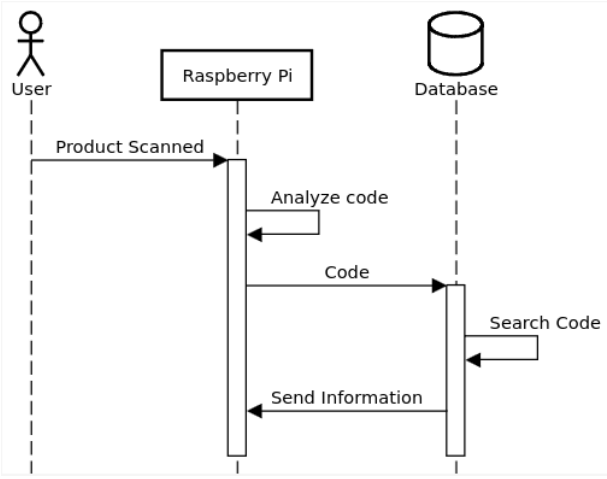

Figure 2.7: Check Stock Sequence

Manage Stock Sequence: This scenario is very similar to the previous one, with the difference that the system needs to analyze not only the product's code but also the operation (add or remove) before proceeding to manage the stock on the Database correctly.

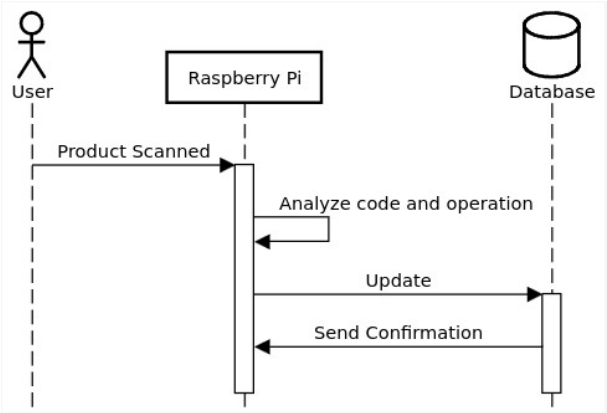

Figure 2.8: Manage Stock Sequence

Manage Sale Sequence: In this case, the user will scan a product and then proceed to confirm the payment by an option that will appear on the GUI display. Subsequently, the database will be updated.

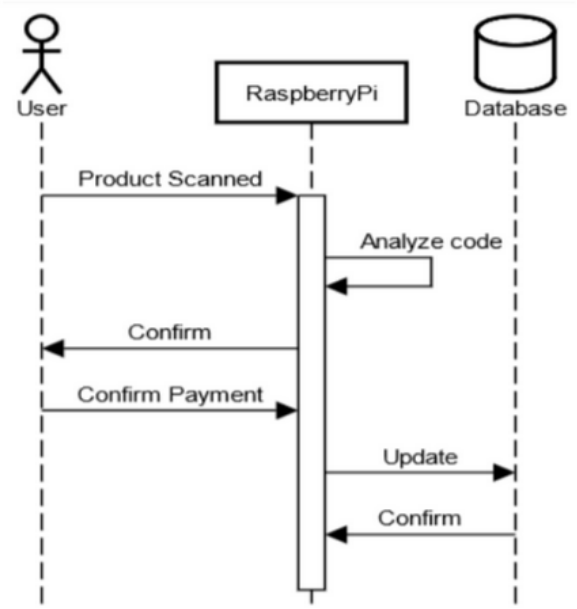

Figure 2.9: Manage Sale Sequence

GUI Sequence:

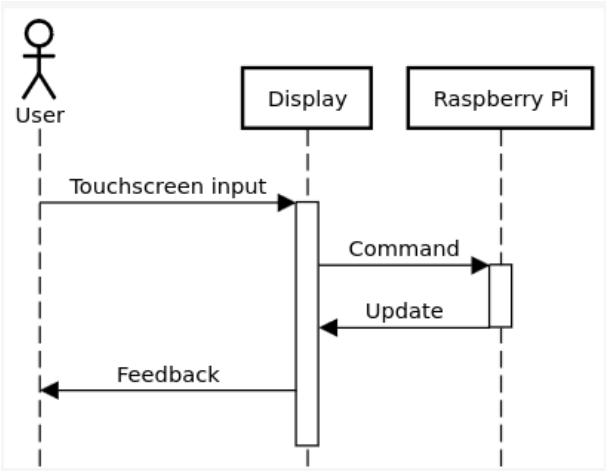

Figure 2.10: GUI Sequence

2.5.8 Entity Relationship Diagram

An entity relationship diagram is a type of flowchart that illustrates how entities are related to each other within a system. Entity relationship diagrams are most often used to design and relation databases. The following figure represents the entity relationship diagram, which makes it possible to analyze all the entities present in the database storage, along with their attributes and how these entities are related to each other. Each product will have a price, a quantity currently available in stock and the name of the product. Each product will also have a one and only relation with the respective supplier. Lastly, the sales will have a date and the respective total amount of the transaction.

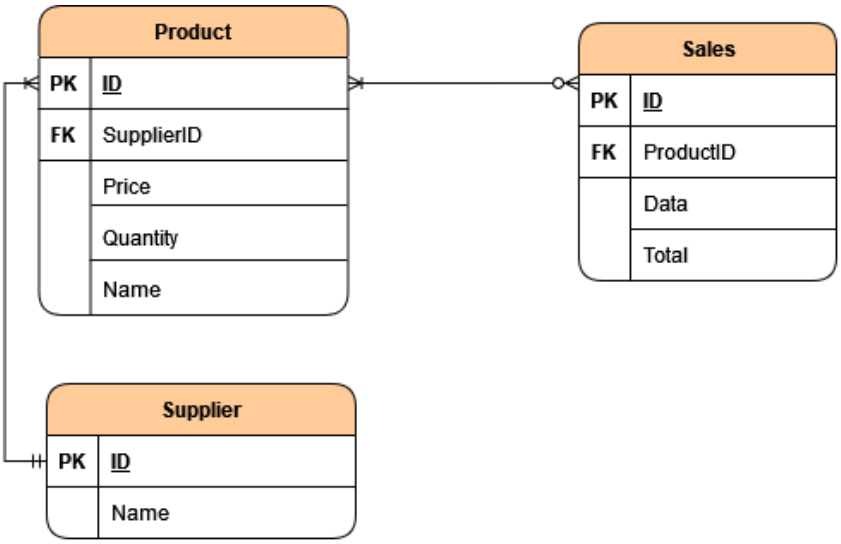

Figure 2.11: Manage Sale Sequence

2.5.9 Estimated Budget

Table 2.1: Estimated Budget

Item	Quantity	Price
Raspberry Pi 4 Model B	1	70€
Touchscreen 3.5-inch	1	27,80€
Barcode Reader	1	38,65€
Power supply	1	8,40€
Temperature Sensor	1	18,90€
Total		163,75€ height

Chapter 3

Project Design

3.1 Theoretical Foundations

Thread

A thread is the smallest unit of execution within a process in a multi-tasking operating system. A process can have multiple threads, and these threads can execute concurrently, allowing for more efficient utilization of resources. There are different types of multi-tasking models that are used to manage threads:

•**Round-Robin model**: where each thread (all with the same priority) is executed at a time and runs until the end.

•**Round-Robin with time slice**: where each thread (all with the same priority) is allowed to run for a specified time slice and then suspended to let other threads use the CPU for that same time slice until all tasks finish executing. This is useful to simulate thread's parallelism.

•**Preemptive model**: The scheduler decides, based on the priority level (i.e., how critical is the thread), which thread, on the ready-to-execute state, will be executed. The scheduler can also interrupt a thread if a high priority one is ready.

Mutex

Mutex, short for "Mutal exclusion", is a synchronization mechanism used to control access of multiple threads to shared resources. In a multi-threaded such as ours, where multiple threads can access and modify shared data simultaneously, conflicts and data inconsistencies can arise. A mutex provides a way to prevent these issues by allowing only one thread to access the shared resource at a time. When a thread acquires a mutex, it gains exclusive access to the protected resource. If another thread tries to access the resource while it is locked by a mutex, it will be blocked until the mutex is released by the owning thread. The Figure 3.1 depics the functionality of a mutex.

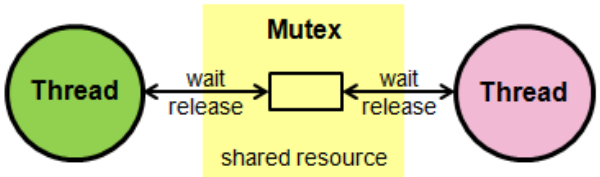

Figure 3.1: Mutex functionality

Device Drivers

As mentioned before, in our project we will be implementing two device drivers. A device driver is a set of functions and data, in the kernel, that make the interface with an external I/O device. Implementing device drivers is required because, besides the fact that was a technical constrain of our project, there is the need to make the bridge between the hardware and the programmer.

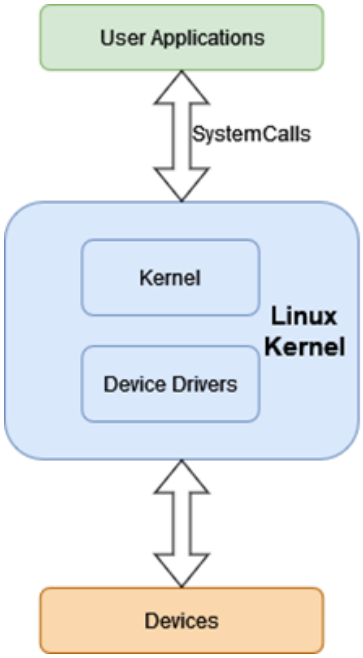

Figure 3.2: Device Drivers

Each I/O device provide a driver-specific set of APIs to the user applications. The Linux Kernel offers a set of functions to the user space to interact with the hardware. For each of these functions there is a matching function in the kernel space allowing data transfer between kernel and the user space.

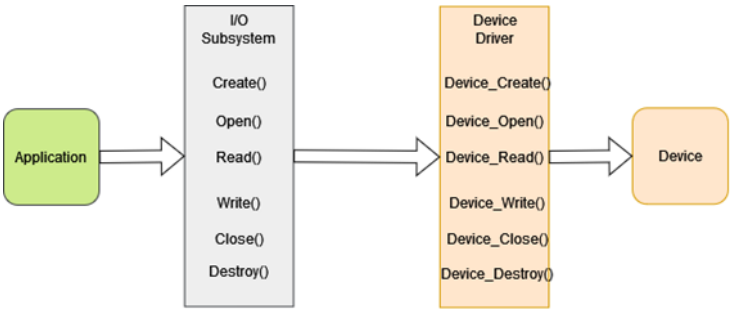

Figure 3.3: Device Drivers

Daemon Process

In computing, a daemon process is a background process, which runs continuously and wakes up to handle period service requests, without any user input. One of the main examples of a daemon is the Hypertext Transfer Protocol daemon (HTTPd), which runs on every web server, continually waiting in dormant mode until requests come in from web clients. The simple steps to create a daemon process are:

1. Fork a child process and exit in the parent process;
2. Use setsid call to create a new session in the child process.
3. Make the root directory the child process's working directory;
4. Change the child process's unmask to 0;
5. Lastly, close any unneeded file descriptor the child might inherited.

3.1.1 Analysis Review

This system consists of a wireless multifunctional scanner, designed to be beneficial across various types of industries, from retail stores to large warehouses. It enhances data accuracy across various scanners and the respective cloud, making it extremely effective in overcoming stockouts and overstocks. Moreover, contributes to informed decision-making, enabling businesses to make strategic decisions based on reliable, up-to-date information.
The system will be temperature-monitored to ensure the safety of the user and, of course, the hardware components. It will also be able to scan different types of product codes, enabling the possibility of managing the current stock and process sales accordingly. All of this will be updated in real-time in the respective database.
User will interact with the system through a user-friendly touch-based GUI (Graphical User Interface). This interface allows the respective user to control the whole system and choose the desired action. Through this intuitive GUI, the system will be able to deliver notifications and/or warnings to the user, ensuring a smooth and accessible user experience.

3.1.2 Hardware Component Specification

After going through an intense study of which components would offer the best price-quality, durability, and reliability, the ones described below were our options. In this chapter, there will to be a description of all the components used in this project, as well as the connection layout.

3.1.3 Raspberry Pi Model 4B

This development board serves as the brain of this project, functioning as the central unit behind its operation. It is responsible for controlling and managing the entire system, integrating all components to create a fully functional unit. It has a pivotal role, ensuring the smooth execution of the system.

Here are features about this development board: • Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
• 1GB, 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
• 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE • Gigabit Ethernet
• 2 USB 3.0 ports; 2 USB 2.0 ports.
• Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
• 2 × micro-HDMI® ports (up to 4kp60 supported)
• 2-lane MIPI DSI display port
• 2-lane MIPI CSI camera port
• 4-pole stereo audio and composite video port
• H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
• OpenGL ES 3.1, Vulkan 1.0
• Micro-SD card slot for loading operating system and data storage

- 5V DC via USB-C connector (minimum 3A)
- 5V DC via GPIO header (minimum 3A)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C

Figure 3.4: Raspberry Pi Model 4B

The Figure 3.5 represents all the 40 GPIOs available on this development board.

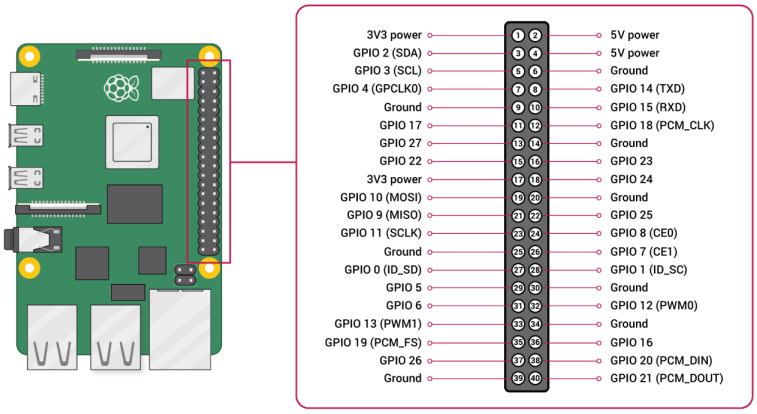

Figure 3.5: Raspberry Pi Model 4B

3.1.4 Barcode Reader

The barcode reader will be one of the main components of this project, as without it, the scanner would be useless. This module is capable of reading 1D/2D codes such as barcodes and QR codes and will be responsible for registering the products. Its efficient and accurate scanning capability ensures swift data capture, facilitating seamless inventory management and sales processing.

Figure 3.6: Barcode Reader

The following are some specifications about this component:
• Operating voltage: 5V
• Operating current: 135mA (scanning), 58mA (standby), 2mA (sleep)
• Operating temperature: 0°C 60°C
• Operating humidity: 5• Communication interface: UART, USB
• Decodes 1D codes: Codebar, Code 11, Code 39/Code 93, UPC/EAN,
Code 128/EAN128, Interleaved 2 of 5, Matrix 2 of 5, MSI Code, Industrial 2 of 5, GS1 Databar(RSS)
• Decodes 2D codes: QR code, Data Matrix, PDF417
• Light source: white
• Scan angels: roll 360°, skew ±65°, pitch ±60°
• Field of view: 28° (horizontal), 21.5° (vertical)
• Dimension: 53.3mm x 21.4mm

3.1.5 Touch Screen TFT LCD

This display serves as the primary link between the user and our project. It provides a convenient interface managing the system, allowing users to select various operation options and access essential information about the stock and sales effortlessly. Its user-friendly design enhances the overall user experience and facilitates efficient navigation through the system's functionalities.

Figure 3.7: Touch Screen

Features and Specifications:
- 480x320 hardware resolution;
- Resistive touch control;
- Supports any revision of Raspberry Pi (directly pluggable);
- 125MHz High-Speed SPI signal transmission, clear and stable display effect;
- Drivers provided;
- Supports FBCP software driver as well, allows to config software resolution and set up dual-display;
- High quality immersion gold surface plating;
- Refresh rate up to 50Hz.

3.1.6 RGB LED

This LED will be useful in assisting the user while working with the system. As an RGB LED, it will provide the user with notifications, such as indicating the temperature level and providing acknowledgment of whether the scan was successful or not through different colors.

Figure 3.8: RGB LED

3.1.7 Battery Holder

To ensure the system's mobility, it will incorporate a battery holder, providing a reliable and long-lasting power supply to the entire system. This feature ensures uninterrupted functionality, enabling the system to operate efficiently even in environments where a stable power source is not readily available.
- Supports 3V 1A voltage outputs.
- Supports 5V 2.2A voltage outputs.
- Micro USB charging current is 600mA-800mA.
- Supports up to two Li-ion batteries.

Figure 3.9: Battery Holder

3.1.8 Temperature Sensor

The temperature sensor will be responsible for monitoring the unit's temperature, preventing potential damage and malfunctions to the system. This crucial function ensures that the system operates within optimal temperature ranges, safeguarding its performance and longevity.
Specifications:
• Operating Voltage: 3.3V - 5V
• Operating Current: 1.2mA
• Temperature: -70.01℃ to +382.19℃, (0.01 ℃ resolution)
• Interface Type: I2C
• Interface Line Sequence: VCC, GND, SCL, SDA
• Dimensions: 31.5*18 mm/1.24 x 0.7 inches
• Weight: 15g

Figure 3.10: Tempeature Sensor

3.1.9 Hardware Connection Diagram

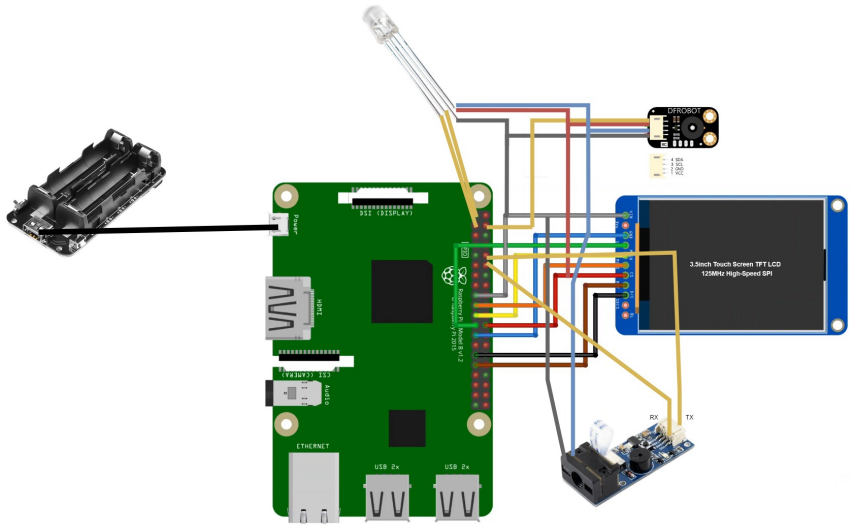

Figure 3.11: Hardware Connection Diagram

3.1.10 Printed Components

To integrate the various components into a unified structure, we will print a custom. The figures below represent the design of the 3D model.

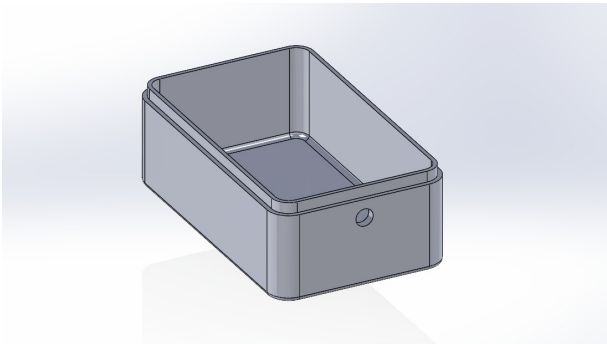

Figure 3.12: Printed Components

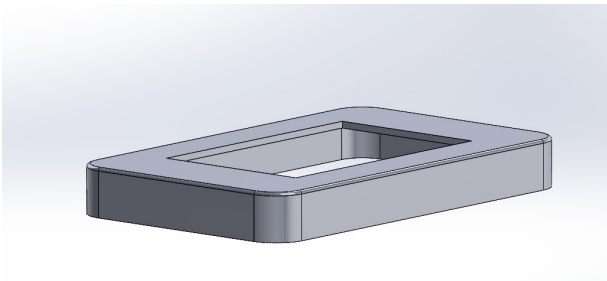

Figure 3.13: Printed Components

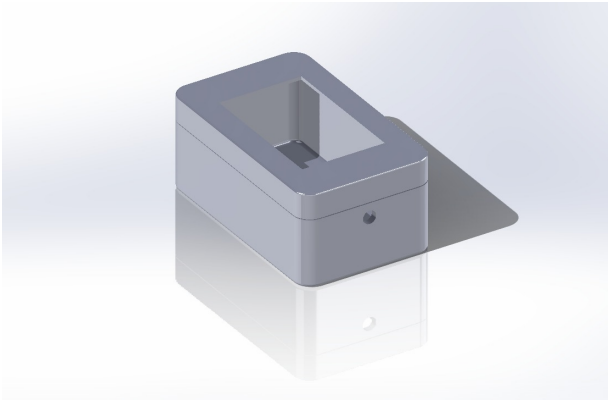

Figure 3.14: Printed Components

3.2 Software Specification

3.2.1 Tools and COTS

In this chapter we will be presenting the tools and COTS (commercial off-the-shelf) applications which are indispensable factors for ensuring efficient and effective project implementation. The software tools refer to applications that are designed to carry out specific tasks or functions to aid in software development. The COTS software is developed, sold, and distributed for the general market and can be easily integrated into existing systems, providing cost-effective and time-effective solutions

Tools:

• **Buildroot**,used to configure and generate custom embedded Linux systems by an image file that includes the Kernel, bootloader, and root filesystem.

Figure 3.15: Buildroot Logo

• **C/C++ Programming Languages**

Figure 3.16: C/C++

• **GitHub**,web-based platform that provides hosting for software development and version control using Git.

Figure 3.17: GitHub Logo

• **Git**, DevOps tool used for source code management. Free and open-source version control system used to handle small to very large projects efficiently.

• **Visual Studio Code**, lightweight but powerful source code editor with several extensions available.

Figure 3.18: Visual Studio Code Logo

• **SolidWorks**, is a parametric CAD (Computer-Aided Design) and CAE (Computer-Aided Engineering) software. It's one of the most popular design and engineering software in the market, known for its range of features and high functionality.

Figure 3.19: Solid Works Logo

• **Overleaf**, An online LaTeX editor that's easy to use. No installation, real-time collaboration, version control, hundreds of LaTeX templates, and more.

Figure 3.20: Overleaf Logo

• **Draw.io**, is free online diagram software for making flowcharts, process diagrams, org charts, UML, ER and network diagrams.

Figure 3.21: Drawio Logo

COTS:

• **PThreads**, short for POSIX Threads, for the standard APIs for all thread related functions.

Figure 3.22: PThreads Logo

• **SQLite**, is a C-language library that implements small, fast, self-contained, high-reliability, SQL database engine. SQLite is an embedded SQL database engine. Unlike most other SQL databases, the SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files.

Figure 3.23: SQLite Logo

• **QT Creator**, is a multi-platform IDE that is part of the Qt SDK and is aimed at simplifying the development of cross-platform GUI applications.

Figure 3.24: QT Creator Logo

• **ThingSpeak**, provides the ability to collect, analyze, and visualize data from various IoT devices and sensors in real-time.

Figure 3.25: ThingSpeak Logo

3.3 GUI Layout

The main interface that this system will have with the user will be the display touchscreen. This chapter will provide a quick and simple description of all the menus that will be available to the user. When the system is initialized, the user has 3 options he can choose: Stock Management, Stock Check and Sales Register.

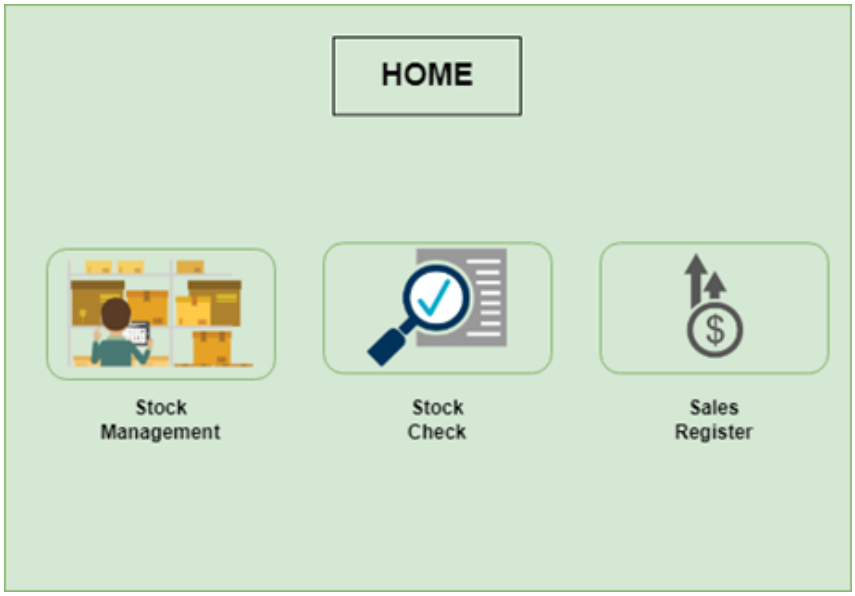

Figure 3.26: Home Menu

When selecting Stock Management the operator has 2 options, either add a product to the stock or, if it is an administrator, remove a product from it.

Figure 3.27: Stock Management Menu

If he tries to remove a product, an “admin” PIN is requested to prevent a regular employee from stealing products.

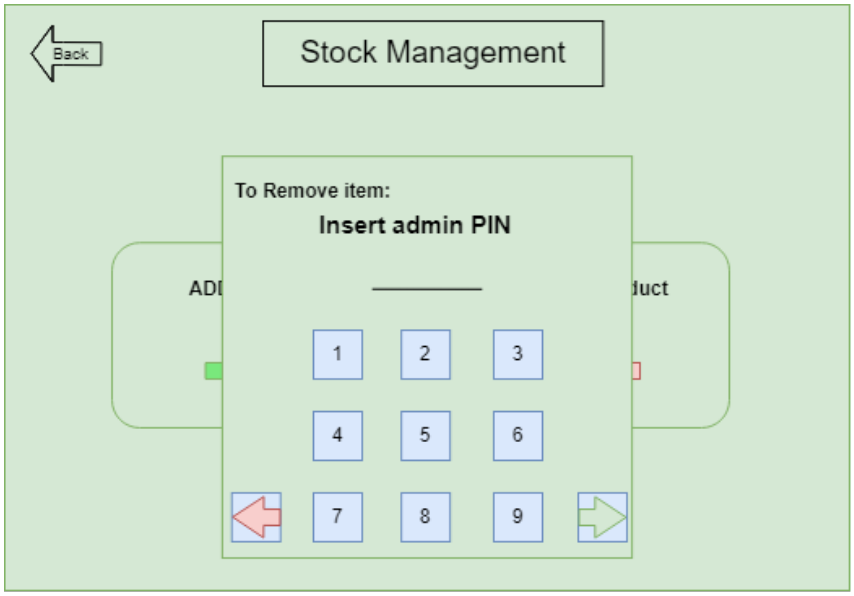

Figure 3.28: Pin Request Menu

After choosing between add or remove, the user must scan the desired product to complete their task and a message will be displayed to inform the user about the success of the operation.

Figure 3.29: Product Added Menu

Figure 3.30: Product Removed Menu

Another option on the Home menu is the Stock Check, where the user can check all the stock information about the scanned product. It will be displayed the product's image and the product's stock information, the quantity available and the respective price.

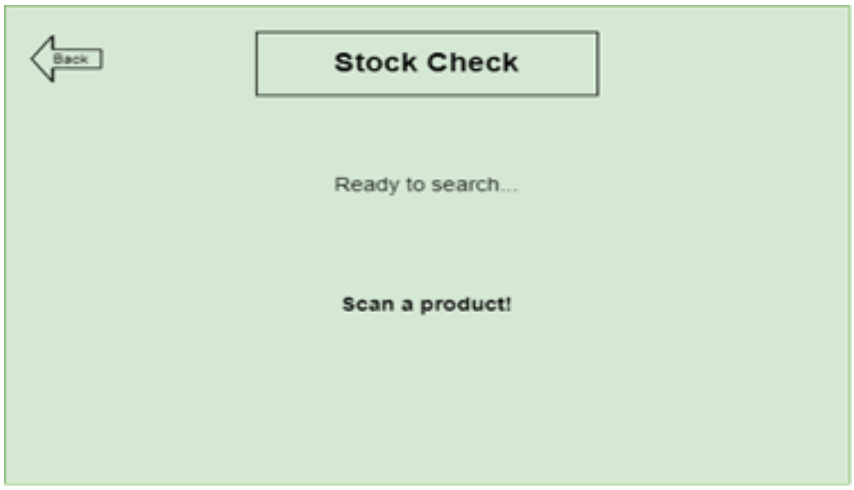

Figure 3.31: Stock Check Menu 1

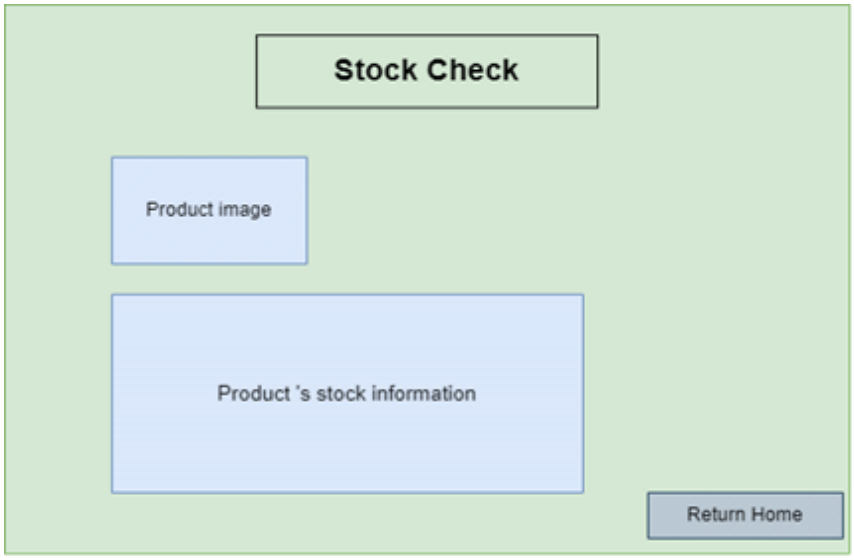

Figure 3.32: Stock Check Menu 2

The third option is the Sales Register, where the user can process the sale of multiple products. After scanning them all, the user must select the option "Proceed to Payment", to preview the bill.

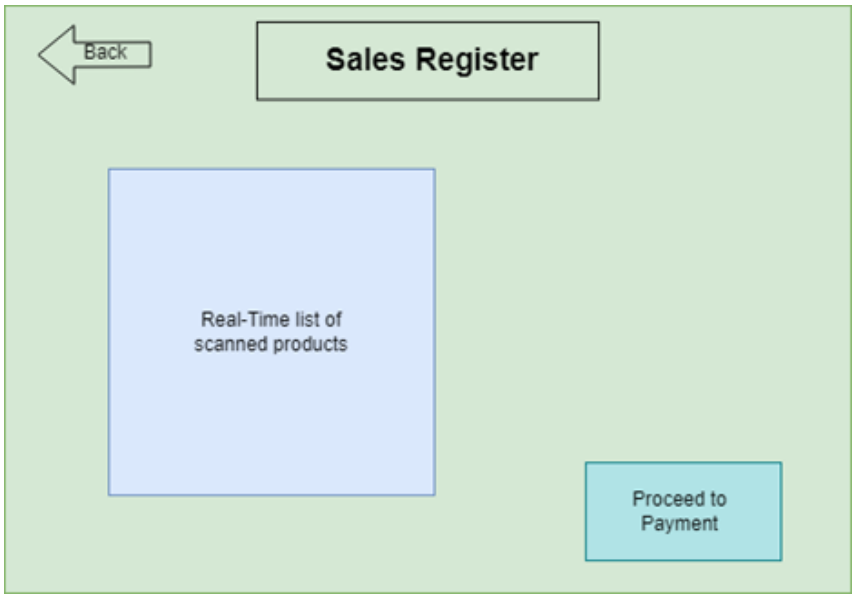

Figure 3.33: Sales Register Menu 1

Moving to the next menu, here the user can preview the bill as mentioned before, which includes quantities and individual prices of the products. It is expected that the employee confirms that he received the right amount of money and then proceeds to the payment confirmation

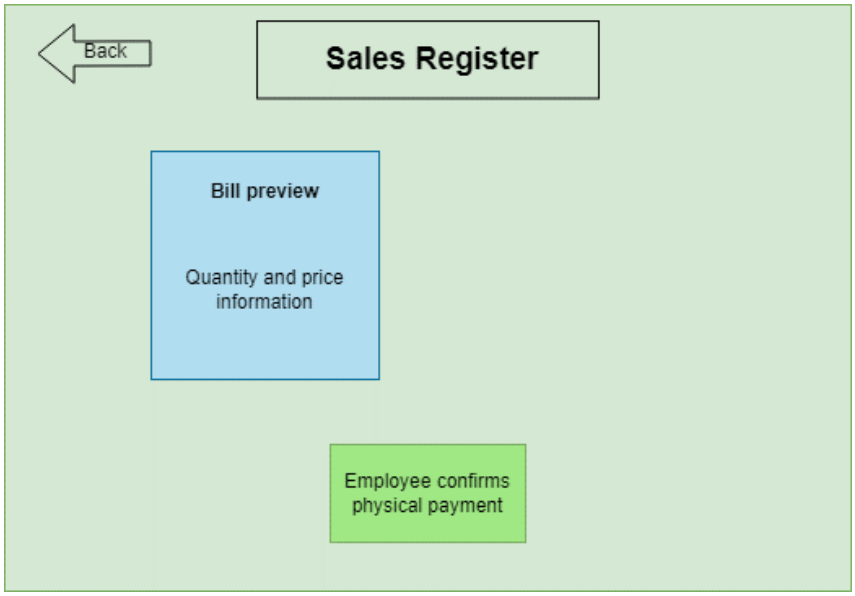

Figure 3.34: Sales Register Menu 2

Lastly, an acknowledgment message is displayed, with an option to return to the Home Menu.

Figure 3.35: Sales Register Menu 3

3.4 Thread Divison

This project will be divided into three parts:
The Main Process, which will be responsible for controlling all the main threads of the system, particularly those dedicated to the peripherals.
The Daemon Process, which is responsible for handling the Database management.
Lastly, the GUI Process will be responsible for issuing commands to the Main Process and refreshing the display with data received by the Main Process.

All three processes will communicate via message queues, with each communication using two message queues to ensure a secure communication between them. Communication to the GUI Process will be executed through the msgFromGUI and msgToGUI message queues, respectively receiving the commands and updating the GUI. The same procedure applies to the database. To ensure safe communication, mutexes will be used to enforce the atomicity of read and write operations in the message queues. The Figure 3.36 represents the thread division of this system.

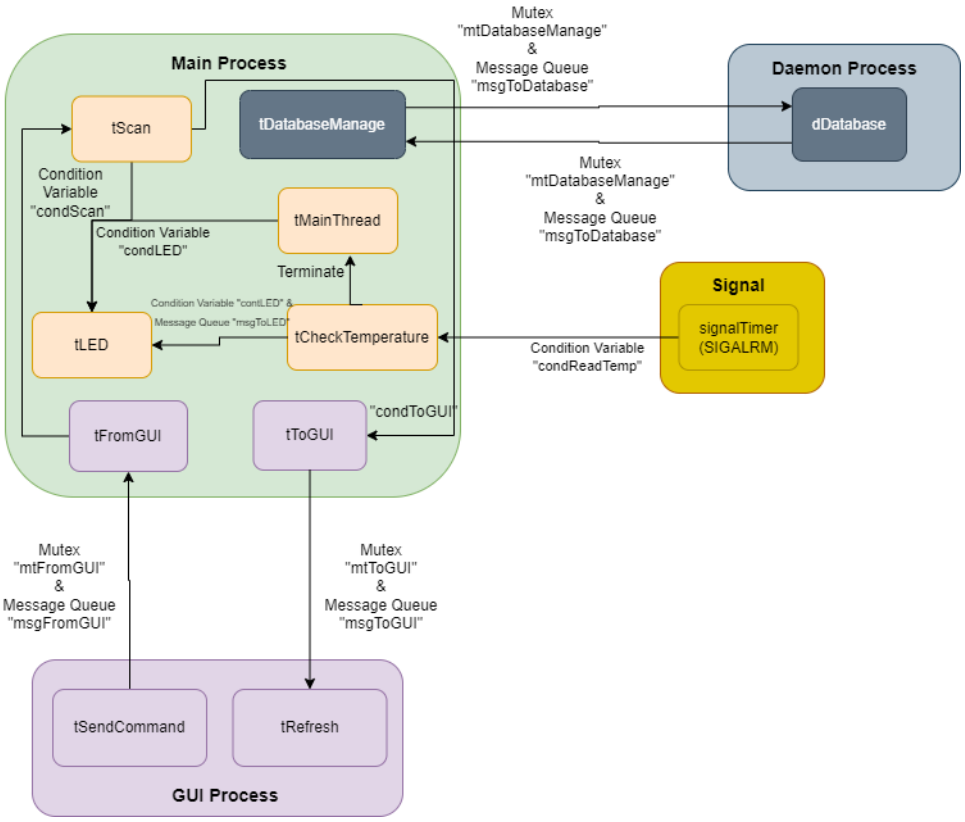

Figure 3.36: Thread Division

3.4.1 Thread Description

On table 3.1 we can see the Main process Threads and their respective description.

Table 3.1: Main Process

Thread Name	Description
tScan	Processes the product's code
tLED	Blinks the LED to acknowledge the scan was successful
tToGUI	Send the information to refresh the GUI
tFromGUI	Receives the information from the GUI
tDatabaseManage	Updates and receives database information
tCheckTemperature	Checks the temperature of the system

On table 3.2 we can see the GUI process Threads and their description.

Table 3.2: GUI Process

Thread Name	Description
tRefresh	Updates the GUI with data sent from the Main Process
tSendCommand	Sends the command the user choose to be processed by the Main Process

On table 3.3 we can see the Daemon Process Threads and their description.

Table 3.3: Daemon Process

Thread Name	Description
dDatabase	Manages the database

3.4.2 Thread Priorities

In our multi-tasking system, the efficient management of thread priorities is vital to ensure smooth execution and optimal resource utilization. In this chapter, we describe our thread's priorities. For safety reasons, the tCheckTemperature thread is prioritized the highest, as it is vital to ensure that the system doesn't overheat and damage the unit.

Managing the database is crucial for the core functionality of our scanner. This thread involves storing, retrieving, and updating item information, making it essential for accurate scanning and data integrity, that's why we've stablished medium priority to the tDatabaseManage thread.

The scanning module is a fundamental component of our system. It interacts with the environment, collects data. Without this thread, our system won't perform its primary function, so tScan thread is also placed in the medium priority level.

Moving to the low priority threads:

The tLED thread provides a visual acknowledgment feedback to the user and can indicate the status of the system's actual temperature. While important for user experience, it is not as critical as the tasks above.

The tFromGUI thread manages the data send by the GUI process, which is essential for configuring the system and providing user input commands. However, it is placed below database management and scanning because it depends on these functionalities.

Finally, the tToGUI thread which is crucial to send data to the GUI to display important information and status updates to users. However, it is of lower priority compared to tasks involving critical system functions, data management, and user-inputted data.

In the figure 3.37, there's a representation of this description.

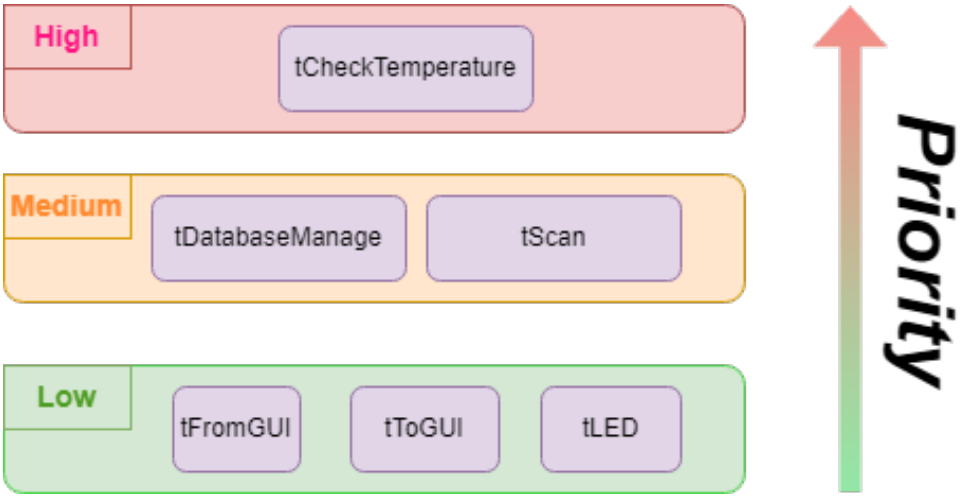

Figure 3.37: Thread Priorities

3.4.3 Thread Synchronization

Thread Synchronization is essential to ensure a reliable system. There are several types of mechanisms to synchronize threads, the ones described below were the ones used in our project:

Table 3.4: Mutex

Name	Description
mtDatabaseManage	Ensure that the tDatabaseManage thread sends or receives the data to or from the database atomically
mtToGUI	Ensure that the tToGUI thread sends the correct data to the display atomically.
mtFromGUI	Ensure that the tFromGUI thread receives the correct command and data from the user input.

Message Queue

The various message queues used in this system will ensure a safe communication among the different processes and the Main process. The operations described are all in relation to the Main Thread, whereas, in the other threads, the operations are inverse.

Table 3.5: Message Queue

Name	Description
msgToDatabase	Message Queue used by tDatabaseManage to send data to the dDatabase thread (write operation).
msgFromDatabase	Message Queue used by tDatabaseManage to receive data from the dDatabase thread (read operation).
mtFromGUI	Ensure that the tFromGUI thread receives the correct command and data from the user input.
msgFromGUI	Message Queue to receive data from tSendCommand to the tFromGUI thread (read operation).
msgToLED	Message Queue to send data from the tCheckTemperature thread to the tLED thread

Condition Variables

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. This system will have five condition variables to wake up the respective thread when necessary.

Table 3.6: Condition Variable

Name	Description
condToGUI	Condition variable used to wake up the tToGUI thread, allowing the Main thread to send data to the GUI.
condFromGUI	Condition variable used to wake up the tFromGUI thread, allowing the Main thread to receive data from the GUI.
condDatabaseManage	Condition variable used to wake up the tDatabaseManage thread, allowing the Main thread to send or receive data to or from the database.
condLED	Condition variable to wake up the tLED thread, allowing the LED to blink to acknowledge the success of the scan.
condScan	Condition variable to wake up the tScan thread, enabling the product scanning.
condReadTemp	Condition variable to wake up the thread tCheckTemperature to read the system temperature.

3.5 Class Diagrams

Class diagrams are crucial as they visually represent the structure and relationships within our system, simplifying complex systems providing better organization. They help detect errors early and provide a framework for coding. In this chapter we will provide a detailed description about each of this class definitions. In the figure 3.38, there is representation of the relationships between all the classes in our system.

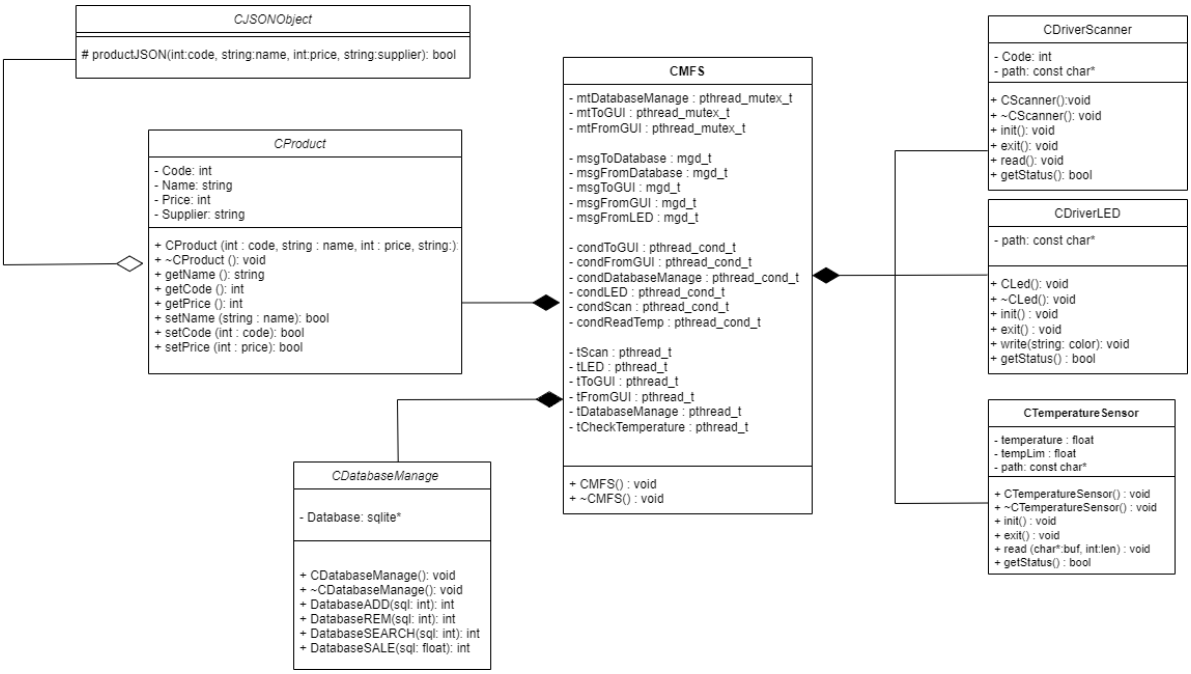

Figure 3.38: Classes Diagram

Database Class

This class is responsible for managing the database. The class will have some public attributes to manage the current stock and two private ones, which are the path for the database directory and the respective database pointer.

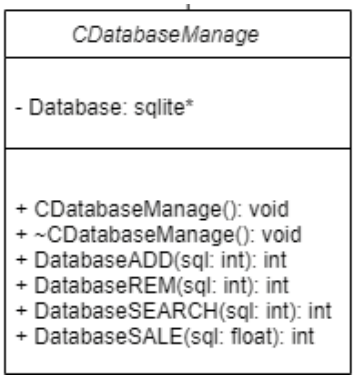

Figure 3.39: Database Class

LED Device Driver Class

This class represents the RGB LED device driver that will be implemented.

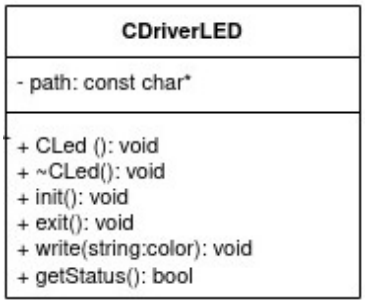

Figure 3.40: LED Class

Scanner Device Driver Class

In the figure 3.41 is represented the class of the scanner device driver.

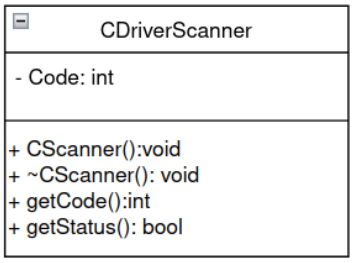

Figure 3.41: Scanner Class

3.6 Flow Charts

Flow charts are crucial in the realm of software development, serving as a fundamental tool to design, plan and comprehend algorithms and programs. They act as a roadmap, illustrating the sequence of processes and highlighting crucial decision-making points within the software's structure. Below there are some flow charts depicting some threads and a description of their functionality:

Main Thread Flowchart

The main thread is responsible for the system start up and shutdown. This thread assumes a pivotal role in the system by maintaining the system's stability and reliability throughout its lifecycle.

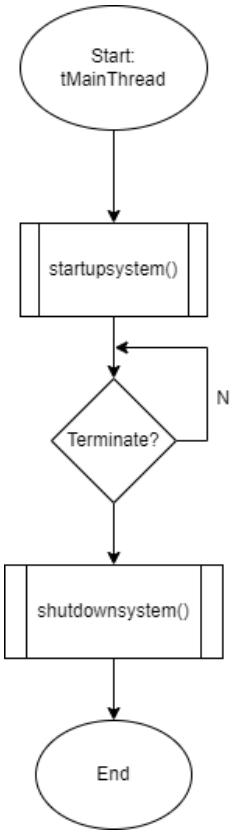

Figure 3.42: Main Thread

LED Thread Flowchart

The tLED thread can be awakened by the respective condition variable from the tScan thread, ensuring a synchronized control and coordination between these threads. This allows the thread to active promptly based on a specific condition dictated by the tScan thread. If the scan is successful, the LED will blink green; otherwise, it will blink red.

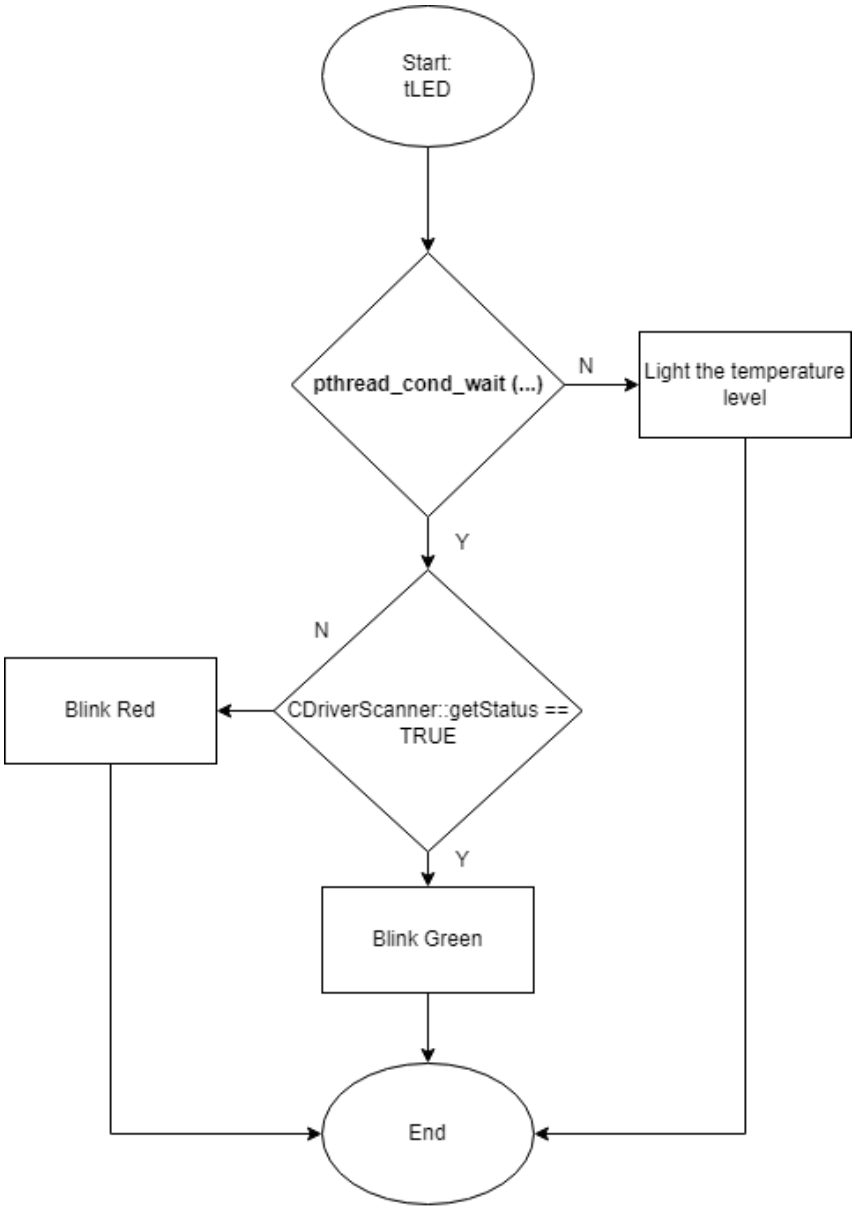

Figure 3.43: LED Thread

Temperature Check Thread

The tCheckTemperature thread is responsible for reading the temperature value from the system. This thread will run intermittently, which necessitates the use of the condition variable which will be set when the system signal "SIGALRM" it set.

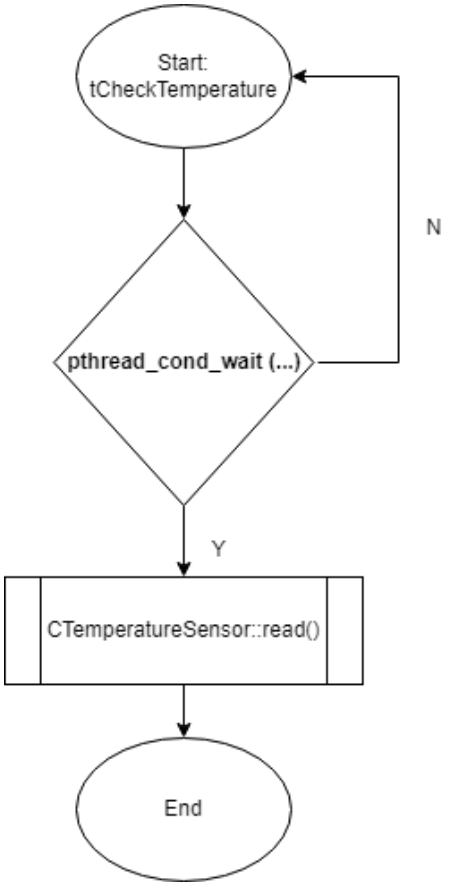

Figure 3.44: CheckTemperature Thread

Database Manage Thread

This thread will be responsible for the communication with the deamon process. It will be awakened by the respective condition variable ("condDatabaseManage") and will write to or reading from the message queues with the deamon process. To ensure a secure communication, a mutex will be used to load data into the message queues atomically.

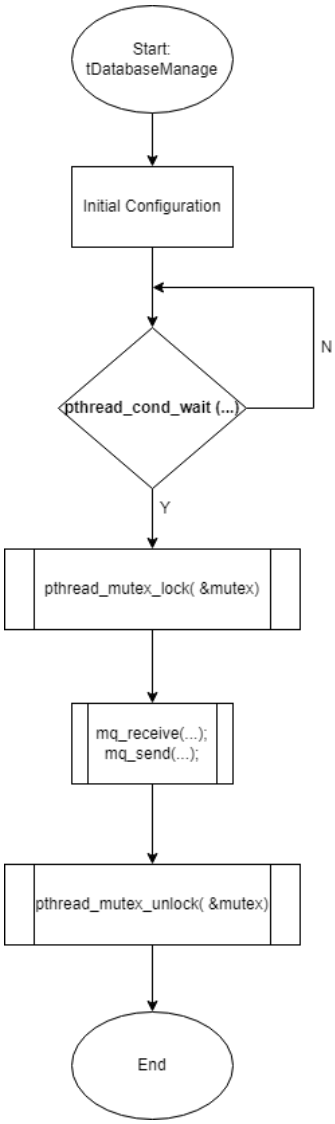

Figure 3.45: Database Manage Thread

Daemon Process Flowchart

This thread in the daemon process will be responsible for the direct communication with the database. It must check for changes in the respective message queues and proceed to execute the command sent by the tDabaseManage thread in the Main process.

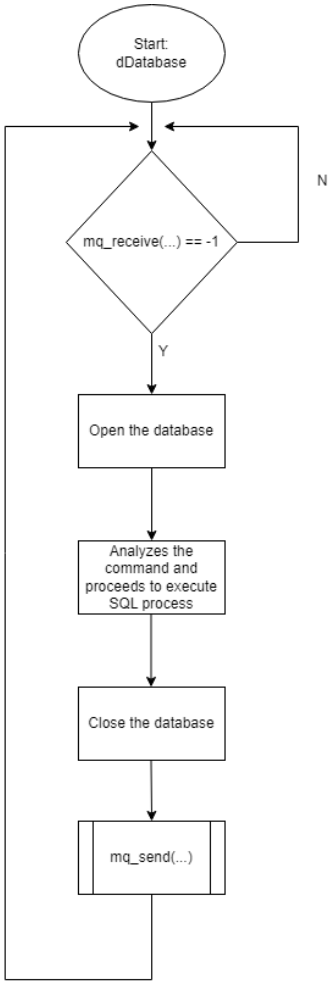

Figure 3.46: Database Daemon Thread

Startup and Shutdown Sequences:

Defining these sequences is necessary to ensure that the system starts up and shuts down smoothly and without any unexpected errors. These two sequences represent the system's behavior upon startup and shutdown.

Startup Flowchart

During the startup sequence, the system initializes all the components and their respective drivers, simultaneously displaying the appropriate interface on the display.

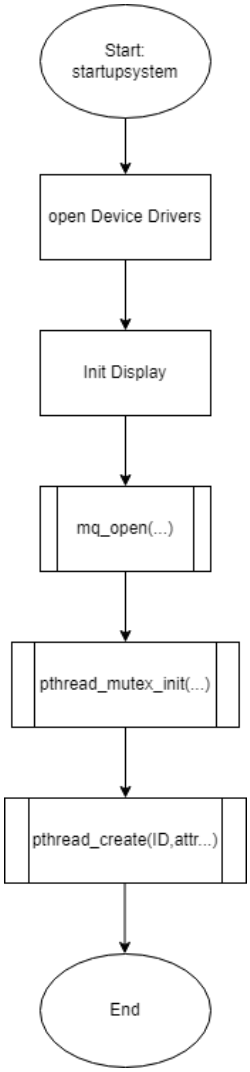

Figure 3.47: Startup FlowChart

Shutdown Flowchart

In the shutdown sequence, the system should notify the user through the LED, followed by the eventual deactivating of the display. It must also to kill all the threads and destroy all the message queues, mutexes and condition variables created before.

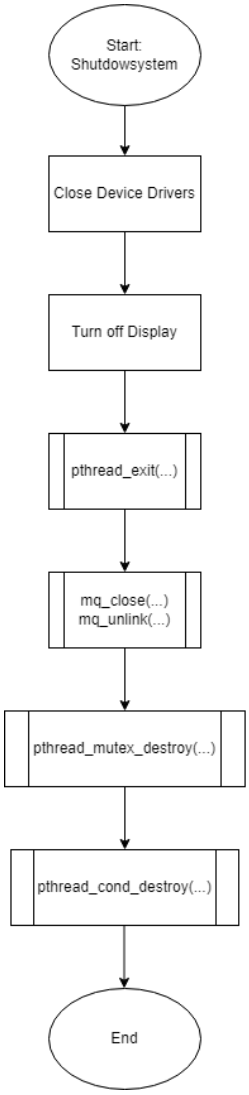

Figure 3.48: Shutdown FlowChart

3.7 Test Cases

Test Cases work as safety checks for embedded systems. They are crucial to confirm the proper functioning of the system components, both hardware and software. This preemptive approach significantly reduces the risk of errors and helps in ensuring the seamless operation of the system in the long run. We will incorporate another form of testing through our Tiebreak group. With these varied verifications processes, we can effectively monitor our project's progress and gain different perspectives about difficulties that we might encounter.

In the barcode scanner test case, represented in the Table 3.7, we expect to receive a buzzer sound(already

integrated in the barcode scan module) when a scan is made, as well as to read the correct product code.

Table 3.7: Barcode Scanner Test Case

Test Name	Expected Result	Real Result
Barcode Scanner	Buzzer noise	
Product Code	Read the correct product code	

The LED test cases are represented in the table 3.8.

Table 3.8: LED Test Cases

Test Name	Expected Result	Real Result
LED Acknowledgment	The LED blinks GREEN if success or RED if not	
LED Temperature	The LED changes its color based on the temperature value	

The temperature sensor test cases are represented in Table 3.9. As it is obsvious, it is expected that the sensor is capable of reading the correct temperature value.

Table 3.9: Temperature Sensor Test Case

Test Name	Expected Result	Real Result
Temperature Sensor	Read the correct temperature value	

The Table 3.10 represents the power supply test case, in which it is expected that the module supplies the correct power to the system.

Table 3.10: Power Supply Test Case

Test Name	Expected Result	Real Result
Power Supply	Power Supply ensuring precise power delivery	

The Table 3.11 represents the touch screen test cases.

Table 3.11: GUI Test Cases

Test Name	Expected Result	Real Result
Display Menus	Display presenting the correct menus	
User Login with PIN	Logged in successfully	
User Input Command	Process the user command and performs it	
Notifications	Informs the user about the success of the commands	

The Table 3.12 represents the test cases to the Database.

Table 3.12: Database Test Cases

Test Name	Expected Result	Real Result
Create Database	Database created successfully	
Load from Database	Read information from the Database	
Save into Database	Save information to the Database	
Delete from Database	Delete information from the Database	

The Table 3.13 represents the system shutdown test case, which occurs when the system overheats. This is also a type of error handling for this system.

Table 3.13: System Shutdown Test Case (Error Handling)

Test Name	Expected Result	Real Result
System shutdown	The system shuts down if the temperature exceeds a certain limit	

3.8 Dry run test

Dry run tests are commonly used to simulate the execution of a program or process without executing it. This type of simulation helps in understanding the flow of the program or identifying potential issues. This process will allow us to preemptively identify any potential issues and ensure that the system functions as intended before deployment.

The following table represents a dry run test of our system and what to expect in terms of the behavior of the different components.

Table 3.14: Dry run test

Line	Battery	Display	Scanned	Temperature Limit	RGB LED	Update Database	System Shutdown
1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0
3	1	1	1	0	1	1	0
4	1	1	0	1	1	0	1

3.9 Errors Handling Specification

Our system, like any other embedded system, will operate under conditions where anything can happen. Because of that, it is crucial to define error-handling routines here in the design phase.

User Interface: In the event of errors, we can notify the user through both the GUI and the simple RGB LED. For example, if the scan was unsuccessful, the user is informed and prompted for another scan.

System Shutdown: If the measured temperature exceeds a predetermined threshold, the system can be manually shutdown but if not, it must shutdown automatically to prevent any damage to the system and the user.

Wi-Fi Connection Lost: If the system loses the wi-fi connection, it will automatically attempt to reconnect. After several attempts, if unsuccessful, the system will display a message to the user.

Chapter 4

Implementation

4.1 Classes

4.1.1 Interfaces

Graphical User Interface

In the figure 4.1 it is possible to see the final implementation of our graphical user interface, with the three main windows, the Stock check, the stock management and the sales register and also the information window, which displays the systems temperature.

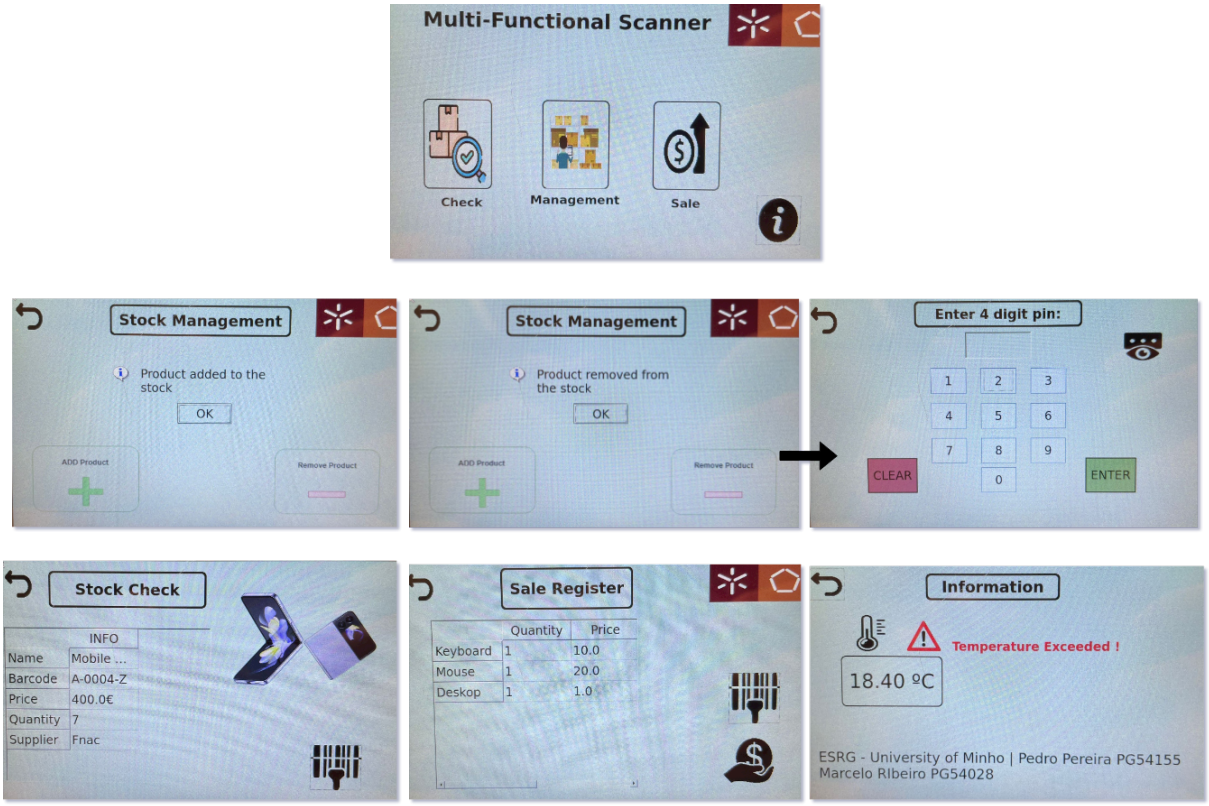

Figure 4.1: Graphical User Interface

Scanner Class

Description: The scanner's class represents a simple interface to interact with a barcode scanner module through a serial port. This class provides public methods for initialization, exiting, reading the data from the scanner, and retrieving the scanned barcode. It encapsulates the necessary configurations and operations required to communicate with the device. It includes necessary headers for serial communication, and the file specifies constants such as the serial port path `SERIAL_PORT_PATH`, the size of the scan command `SCAN_COMMAND_SIZE`, the buffer size `BUFF_SIZE`, and the size of the barcode code `CODE_SIZE`.

The figure 4.2 represents the interface of this class.

Usage

1. Create an instance of Scanner.
2. Initialize the scanner using the `init()` method.
3. Wake up the scanner module and retrive scanned data using the `readScanner()` method.
4. Obtain the scanned barcode in various formats using `getCode()` and `getString()` methods.
5. Ensure proper cleanup by calling exit().

```
#ifndef SCANNER_H
#define SCANNER_H

#include <termios.h>
#include <stdint.h>
#include <string>

constexpr const char* SERIAL_PORT_PATH="/dev/ttyAMA0";
constexpr int SCAN_COMMAND_SIZE=9;
constexpr int BUFF_SIZE=256;
constexpr int CODE_SIZE=8;

class Scanner {
public:
    Scanner();
    ~Scanner();
    void init();
    int exit();
    //Scanner Operations
    void readScanner();
    const uint8_t* getCode() const;
    std::string getString() const;

private:
    struct termios g_tty;
    int fd;
    uint8_t code[CODE_SIZE];
    //HEAD1 HEAD1 TYPES WRITEOPP ADDRESS ADDRESS DATA NOCRC NOCRC
    uint8_t scan[SCAN_COMMAND_SIZE]={0x7E,0x00,0x08,0x01,0x00,0x02,0x01,0xAB,0xCD};
    int open_config_port();
};
#endif
```

Figure 4.2: Interface of Scanner Class

MLX90614 Class

Description: The MLX90614's class provides an interface to interact with a MLX90614 infrared temperature sensor through the I2C protocol. The file specifies constants such as the I2C address `MLX90614_I2CADDR`, device path `I2C_DEV_PATH`, and a command for reading the thermopile sensor `command`. This class encapsulates the necessary I2C management, including configurations and operations specific to the MLX90614 sensor.

The figure 4.3 represents the interface of this class.

Usage

1. Create an instance of MLX9014.
2. Initialize the sensor using the `init()` method.
3. Read temperature data using `readTemperature()`.
4. Obtain the temperature value using `getTemperature()` method.
5. Ensure proper cleanup by calling exit().

```
#ifndef MLX90614_H
#define MLX90614_H

#include <linux/i2c-dev.h>
#include <linux/i2c.h>

constexpr int MLX90614_I2CADDR=0x5A;
constexpr const char* I2C_DEV_PATH="/dev/i2c-1";
constexpr char command=0x06; // command 0x06 is reading thermopile sensor

class MLX90614{
public:
    MLX90614();
    ~MLX90614();

    int init();
    int exit();
    void readTemperature(); //setter
    double getTemperature() const; //getter

private:
    typedef union i2c_smbus_data i2c_data;
    i2c_data data;
    int fd;
    double temperature=0;

    int openI2C();
    int setSlaveAddress();
    int enableCheckSum();
    int readData(i2c_smbus_data& data, char command);
};
#endif
```

Figure 4.3: Interface of MLX90614 Class

rgbDevice Class

Description: The rgbDevice class provides an interface to control the RGB LED through GPIO pins. The interface of this class is presented in the figure 4.4, showing all the public methods and private members of this class. It also includes all the necessary constants for the device name, lcass name and device path. The GPIO pins for the red, green and blue pins of the RGB LED are also defined as constants. The class provides public methods for initialization, exiting, and writing the respective color. Internally, it holds a file descriptor and a pointer to the GPIO registers.

The figure 4.4 represents the interface of this class.

Usage

1. Create an instance of rgbDEvice.
2. Initialize the RGB LED using the `init()` method.
3. Set the color using the `writeColor()` method.
4. Ensure proper cleanup by calling `exit()`.

```
#ifndef RGBDRIVER_H
#define RGBDRIVER_H

#include <termios.h>
#include <stdint.h>
#include <string>
#include <fcntl.h>
#include <unistd.h>
#include <cstdio>
#include <cerrno>

constexpr const char* DEVICE_NAME= "rgbDevice";
constexpr const char* CLASS_NAME= "rgbledClass";
constexpr const char* LED_DEVICE_PATH= "/dev/rgbDevice";

//led pins
constexpr int LedGpioPinR = 17;
constexpr int LedGpioPinG = 27;
constexpr int LedGpioPinB = 22;

class rgbDevice {
    public:
        rgbDevice();
        ~rgbDevice();

        int init();
        int exit();
        void writeColor(int color);

    private:
        int LEDfd;
        struct GpioRegisters *s_pGpioRegisters;
};

#endif
```

Figure 4.4: Interface of rgbDevice Class

CProduct Class

Description: The CProduct class encapsulates the representation and functionality of a product within a inventory system. Objects of this class are designed to store information about a product, including its name, barcode, price, supplier, quantity on hand, quantity purchased, and the number of times it has been scanned. This class provides three different constructors, to allow the creation of `CProduct` objects based on various attributes. The class provides methods to modify and to retrieve these attributes, enabling dynamic management of product information.

The `CProduct` class also includes comparison operators, allowing a straightforward comparison between product objects.

```
#ifndef __PRODUCT_
#define __PRODUCT_

#include <iostream>

using namespace std;

class CProduct
{
    public:
    CProduct();
    CProduct( string Name, string Barcode, string Price,string Suplier); //The objects are created in Products.json
    CProduct( string Name, string Barcode, string Price,int Quantity, string Suplier);
    CProduct( string Name, size_t Buyed_Qty);
    ~CProduct();

    void setName(string NewName);
    void setBarcode(string NewBarcode);
    void setPrice(string NewPrice);
    void IncQuantity();
    void DecQuantity();
    void setSuplier(string NewSuplier);
    void setBuyed_Qty(size_t Buyed_Qty);
    void incBuyed_Qty();
    void setScansAmount(int Amount);
    void incScansAmount();

    string getName();
    string getBarcode();
    string getPrice();
    string getSuplier();
    size_t getQuantity();
    size_t getBuyed_Qty();
    int getScansAmount();

    bool operator==(const CProduct &other)const;

    private:
    string Name, Barcode, Suplier, Price;
    size_t Quantity, Buyed_Qty, Scanned_counter;
};

#endif
```

Figure 4.5: Interface of CProduct Class

CDatabaseManage Class

Description: The `CDatabaseManage` class serves as the central hub for managing product data, using a combination of JSON files and an embedded SQLite database. This class is designed to easily operate the database, such as initializing, adding and removing products, searching for products by barcode, and conducting sales transactions. This class includes essential libraries, including JSONCpp to handle JSON files, SQLite for database management, and libcurl for network-related operations.

```
#ifndef __DATABASE_
#define __DATABASE_

#include <iostream>
#include <string>
#include <fstream>
#include <ctime>
#include <sstream>
#include <unistd.h>

#include "CProduct.h"
#include "jsoncpp-master/include/json/value.h"
#include "jsoncpp-master/include/json/json.h"
#include "SQLITE/sqlite3.h"
#include "curl/include/curl/curl.h"

class CDatabaseManage
{
public:
    CDatabaseManage();
    CDatabaseManage(const char* DB_Path, sqlite3* db);
    ~CDatabaseManage();

    vector<CProduct> InitDB(string JsonFilePath,sqlite3* db);
    void addOperation(string Scanned_barcode, vector<CProduct> &products,sqlite3* db,string JsonFilePath);
    void remOperation(string Scanned_barcode, vector<CProduct> &products,sqlite3* db,string JsonFilePath);
    CProduct searchOperation(string Barcode,sqlite3* db);
    void saleOperation(vector<CProduct> &Bill);

private:
    const char* DB_Path;
    sqlite3* db;
};

#endif
```

Figure 4.6: Interface of CDatabaseManage Class

MFS Class

Description: The `MFS` class is the main class of our system. In C++ object-oriented programming, a composition is a special type of association between classes that means a more robust relationship than a standard association. This type of relationship is often described as a "has-a" relationship, indicating that one class possesses an instance of another class as a member.

Applying this concept to our system, the `MFS` class acts as the central unit, encapsulating diverse functionalities through a composition relationship with other classes. Also, following the principles of the singleton design pattern, the `MFS` class ensures the existence of a sole instance, promoting global accessibility.

In figure 4.7, it is possible to see the included libraries and dependencies and also the constant expressions, such as the temperature threshold, LED modes, and file paths for the SQLite database and a JSON file.

```
1  #ifndef MFS_H
2  #define MFS_H
3
4  #include "/home/pedro/Desktop/Pthreads/Scanner/Scanner.h"
5  #include "/home/pedro/Desktop/Pthreads/mlx90614/mlx90614.h"
6  #include "/home/pedro/Desktop/sqliteNovo/sqlite/SQLITE/sqlite.h"
7  #include "sqlite3.h"
8  #include "/home/pedro/Desktop/sqliteNovo/sqlite/CDatabaseManage.h"
9  #include "/home/pedro/Desktop/sqliteNovo/sqlite/CProduct.h"
10 #include "/home/pedro/Desktop/driverRGBLED/rgbDriver.h"
11 #include "mfswrapper.h"
12
13 #include <unistd.h>
14 #include <pthread.h>
15 #include <mqueue.h>
16 #include <ostream>
17 #include <iostream>
18 #include <fstream>
19 #include <vector>
20 #include <signal.h>
21
22 #include <QDialog>
23 #include <QTimer>
24 #include <QEventLoop>
25 #include <QMessageBox>
26 #include <QObject>
27 #include <QPushButton>
28 #include <QMetaObject>
29
30 /* constexpr VS define. Const is handled by the compiler, defines are for preprocessing.
31    Used constexpr because it has type checking, and define does not. */
32
33 // Temperature threshold
34 constexpr int TEMP_TRESHOLD=18;
35
36 // Constants for LED modes
37 constexpr int OFF=0;
38 constexpr int ON=1;
39 constexpr int RED=2;
40 constexpr int GREEN=3;
41 constexpr int BLUE=4;
42
43 // File Paths
44 constexpr const char* pathDB = "/etc/basedados/BaseDeDados";
45 constexpr const char* pathJSON = "/etc/basedados/Products.json";
46
```

Figure 4.7: Interface of MFS Class - 1

In figure 4.8 is represented the public methods of the `MFS` class. The singleton design, as mentioned earlier, assures the existence of only one instance of `MFS` class. This design pattern is implemented through the use of a static member function, `getInstance()`, which provides access to the single instance. Proper lifetime management is crucial in this context and is ensured by disabling the copy constructor and copy assignment operation.

It is also depicated the device operations,

```
class MFS
{
public:
    /*-------------SINGLETON------------*/
    static MFS& getInstance();
    MFS(MFS const&) = delete;
    void operator=(MFS const&) = delete;
    ~MFS();

    void startupSystem();   //startup sequence
    void shutdownSystem();  //shutdown sequence

    /*-------------DEVICE OPERATIONS------------*/
    void performScannerRead();
    void performSensorRead();
    double getSensorValue() const;
    std::string getCodeString() const;
    void writeColor(int value);
    /*-------------------------------------*/

    CProduct getFoundProduct(){return FoundProd;}
    void setFoundProduct(CProduct prod){FoundProd = prod;}

    bool isDBsearchComplete(){return DBSearchComplete;}
    void Reset_isDBsearchComplete(){DBSearchComplete = false;}

    void PurshaseCompleted_signal();

    void IncBuyedQtyVector(CProduct product);
    void ResetQtyVector();
    /*------------- CONDITION VARIABLE ------------*/
    void signaltoGUIfromStockCheck();
    void signalScan();
    void signalHandler();

    /*------------- WRAPPER ------------*/
    int getMLXStatus();

    /*---------- DB OPERATION ---------*/
    static void receive_Value(int value);
```

Figure 4.8: Interface of MFS Class - Public Members

The figures 4.9 and 4.10 represent the private members and methods of the `MFS` class. Because of the singleton design pattern, the constructor of this class must be a private member.

```
87
88  private:
89      MFS();
90
91      int DBoperation=0;    //stores the value from the signal
92      int LedMode=0;        //Can be representing either the temperature or the scan ack
93      int mlxStatus=0;
94      int previousLED=0;
95
96      const char* DB_Path = pathDB;
97      const char* JsonFilePath = pathJSON;
98      sqlite3* db;
99
100     //***objetos de classes - COMPOSICAO ****
101     Scanner scanner;
102     MLX90614 TempSensor;
103     rgbDevice led;
104
105     CProduct Product;
106     CProduct FoundProd;      //Necessary to acess the search product in others source files
107     CDatabaseManage DBM;
108     vector<CProduct> ProductsVector;
109     bool DBSearchComplete;
110
111     bool signalFromStockcheck;
112     bool signalFromCheckTemp;
113     bool signalFromScan;
114     bool ledModeChanged;
115
116     //-------------- START UP SYSTEM -----------
117     void SetupThread(int prio,pthread_attr_t *pthread_attr,struct sched_param *pthread_param);
118     void createThreads();
119     void initializeAttributes();
120     void initializeDevices();
121     void waitforThreads();
122     void insertDrivers();
123
124
125     //-------------- SHUT DOWN SYSTEM -----------
126     void mutexDestroy();
127     void condDestroy();
128     void destroyAttributes();
129     void deinitializeDevices();
130     void signalThreads();
131     void removeDrivers();
132
133     int thread_policy;
```

Figure 4.9: Interface of MFS Class - Private Members

Regarding thread management, the class also includes attributes and parameters defined for the threads, along with unique threads IDs. The condition variables and mutexes are also private members, as well as, the static methods serving as entry points for the threads.

```
137
138     //ATTRIBUTES
139     pthread_attr_t thread_attr,
140         Highprio_RR_attr,
141         Mediumprio_RR_attr,
142         Lowprio_RR_attr;
143
144     struct sched_param thread_param,
145         Highprio_RR_param,
146         Mediumprio_RR_param,
147         Lowprio_RR_param;
148
149     //THREADS IDS
150     pthread_t tScan_id, tDBM_id, tLED_id, tCheckTemp_id, tToGUI_id;  //5
151
152     //CONDITION VARIABLE
153     pthread_cond_t condToGUI, condLED, condScan, condReadTemp, condDatabaseManage, condDatabaseSearchComplete_ = PTHREAD_COND_INITIALIZER;
154
155     //MUTEXES
156     pthread_mutex_t condMutex, mtScan, mtLED, mtDatabaseManage, mtTemperature, mtToGUI = PTHREAD_MUTEX_INITIALIZER;
157
158     // Thread entry functions
159     static void* tcheckTemperature(void* arg);
160     static void* tdatabaseManage(void* arg);
161     static void* tscan(void* arg);
162     static void* tfromGUI(void* arg);
163     static void* ttoGUI(void* arg);
164     static void* tLED(void* arg);
165 };
166 #endif
```

Figure 4.10: Interface of MFS Class - Private Members

mfsWrapper Class

The `mfsWrapper` class is a QObject-based wrapper that uses the signal and slot mechanisms of the Qt framework. This class became necessary due to a limitation in the original `MFS` class, which did not inherit from QObject. The wrapper class allows the use of Qt's features, such as signals and slots.

Similar to the original `MFS` class, the `mfsWrapper` class also adopts the singleton design pattern. Public methods are available for emitting the signals, which are defined as class members. The signals will act as notifiers, informing other components about specific events or changes.

In the figure 4.11, the interface of this wrapper class is depicted.

```
1   #ifndef MFSWRAPPER_H
2   #define MFSWRAPPER_H
3
4   #include <QObject>
5
6   class mfsWrapper : public QObject
7   {
8       Q_OBJECT
9   public:
10      static mfsWrapper& getInstance();
11      void emitUpdateSignal(double value); //for the temperature sensor
12      void emitScanSignal(std::string code);  //for the scanner's code
13      void emitWarningSignal(int value);  //for the i2c warnings
14      void emitTableUpdateSale(); //for the register sale table
15      void emitAddSignal(int value);   //for the stock management
16      void emitRemoveSignal(int value); //for the stock management
17
18  signals:
19      void updateValueSignal(double value);
20      void updateScanSignal(std::string code);
21      void updateWarningSignal(int value);
22      void updateTableSale();
23      void addSignal(int value);
24      void removeSignal(int value);
25
26  private:
27      mfsWrapper(QObject *parent = nullptr);
28  };
29
30  #endif // MFSWRAPPER_H
31
```

Figure 4.11: Interface of mfsWrapper Class

4.1.2 Implementation

Scanner Class

In figure 4.12, it is possible to see the default constructor and destructor of the class, as well as the initialization function `init()`, along the `exit()` method. The `init()` method calls a private method of this class, while the `exit()` method exits and cleans up resources by closing the same file.

The `open_config_port()` method opens and configures the serial port with specified attributes. This is done using the `termios` functions, that is provided to control asynchronous communication ports. By the use of the `termios` structure it is possible to configure the communication port. This method first opens the specified serial port file using the `open()` system call, and upon success, proceeds to configure the

```
/**
 * @brief Default constructor for the Scanner class.
 */
Scanner::Scanner(){}

/**
 * @brief Default destructor for the Scanner class.
 */
Scanner::~Scanner(){}

/**
 * @brief Initializes the scanner by opening the corresponding serial port file.
 * @return 0 on success, -1 on failure.
 */
void Scanner::init(){
    if(open_config_port()!=0){
        perror("Error opening file...\n");
        return;
    }
}

/**
 * @brief Exits and cleans up resources by closing the serial port file.
 * @return 0 on success, -1 on failure.
 */
int Scanner::exit(){
    if(close(fd)!=0){
        perror ("Error closing device ... \n");
        return -1;
    }
    return 0;
}
```

Figure 4.12: Implementation of the Scanner Class - 1

port attributes. The termios structure `g_tty` is employed to costumize the communication parameters. The `cfsetispeed()` and `cfsetospeed()` functions set the input and output baud rates stored in the `termios` structure, respectively, in this case, both configure to B9600. The minimum character read and wait time are also defined, through `g_tty.c_cc[VMIN]` and `g_tty.c_cc[VTIME]`.

Furthermore, the function adjusts the control flags (`c_cflag`) to specify settings such as stop bits, hardware flow control, and enabling the receiver. The changes are then applied to the serial port using the `tcsetattr()` function with the `TCSANOW` argument (to apply the changes immediately).

```
47  /**
48   * @brief Opens and configures the serial port.
49   * @return 0 on success, -1 on failure.
50   */
51  int Scanner::open_config_port() {
52      fd = open(SERIAL_PORT_PATH, O_RDWR | O_NOCTTY);
53      if(fd < 0) {
54          printf("Could not open file %s...%d\r\n",SERIAL_PORT_PATH,fd);
55          return fd;
56      }
57      //Configure Serial Port
58      if(tcgetattr(fd, &g_tty)) {
59          perror("Something went wrong while getting port attributes...\r\n");
60          close(fd);
61          return -1;
62      }
63      //configure the input and output baudrate
64      cfsetispeed(&g_tty,B9600);
65      cfsetospeed(&g_tty,B9600);
66      //raw mode
67      cfmakeraw(&g_tty);
68
69      //minimum character read and wait time
70      g_tty.c_cc[VMIN] = 0;
71      g_tty.c_cc[VTIME] = 60;
72
73      //stop bits, no rts/cts, ignore CD signal ,enable receive.
74      g_tty.c_cflag &= ~CSTOPB;
75      g_tty.c_cflag &= ~CRTSCTS;
76      g_tty.c_cflag |= CLOCAL | CREAD;
77      //apply the changes to the serial port
78      if(tcsetattr(fd,TCSANOW,&g_tty) < 0) {
79          perror("Something went wrong while setting port attributes...\r\n");
80          close(fd);
81          return -1;
82      }
83      return 0;
84  }
```

Figure 4.13: Implementation of the Scanner Class - 2

The `readScanner()` method initiates communication with the scanner module by writing the scan command provided by the manufacturer. It then reads the scanner's response to the command, including a 256-byte buffer containing the actual data read by the scanner. The retreived code is then stored in the private member variable `code()`. Finally, the `getCode()` and `getString()` methods return the same private member but in different data types.

```
86  /**
87   * @brief Reads data from the scanner, including the scan command and actual code.
88   */
89  void Scanner::readScanner() {
90      uint8_t buff[BUFF_SIZE];
91      //write the scan command
92      if (write(fd, scan, SCAN_COMMAND_SIZE) < 0) {
93          perror("Error writing to UART1");
94          return;
95      }
96      //read the scanner's response
97      memset(buff, 0, BUFF_SIZE);
98      if (read(fd, buff, BUFF_SIZE) < 0){
99          perror("Error reading from UART2");
100         return;
101     }
102     // read the 256 byte buffer
103     memset(buff, 0, BUFF_SIZE);
104     if (read(fd, buff, BUFF_SIZE) < 0){
105         perror("Error reading from UART3");
106         return;
107     }
108     //get the actual code
109     for (int i = 0; i < CODE_SIZE; ++i) {
110         code[i] = buff[i];
111     }
112 }
113
114 /**
115  * @brief Getter function for the scanner code.
116  * @return Pointer to constant data representing the scanner code.
117  */
118 const uint8_t* Scanner::getCode() const{
119     return code;
120 }
121
122 /**
123  * @brief Getter function for the scanner code as a string.
124  * @return String representation of the scanner code.
125  */
126 std::string Scanner::getString() const{
127     std::string barcode((char*)code, CODE_SIZE);
128     return barcode;
129 }
```

Figure 4.14: Implementation of the Scanner Class - 3

MLX90614 Class

The implementation of the `MLX90614` Class is represented by the figures 4.15, 4.16 and 4.17.

In figure 4.15, it is possible to see the default constructor and destructor of the class, as well as the initialization function `init()`, along with the cleanup function, which is the `exit()` method.

```
13  /**
14   * @brief Default constructor for the MLX90614 class.
15   */
16  MLX90614::MLX90614(){}
17
18  /**
19   * @brief Default destructor for the MLX90614 class.
20   */
21  MLX90614::~MLX90614(){}
22
23  /**
24   * @brief Initializes the MLX90614 sensor by opening the I2C interface, setting the slave address, and enabling checksum.
25   * @return 0 on success, non-zero on failure.
26   */
27  int MLX90614::init(){
28      if(openI2C()!=0){
29          perror("Error opening I2C...\n");
30          return 1;
31      }
32      if(setSlaveAddress()!=0){
33          perror("Error defining slave address...\n");
34          return 2;
35      }
36      if(enableCheckSum()!=0){
37          perror("Error enabling checksum...\n");
38          return 3;
39      }
40      return 0;
41  }
42
43  /**
44   * @brief Cleans up resources and closes the I2C interface.
45   * @return 0 on success, -1 on failure.
46   */
47  int MLX90614::exit(){
48      if(close(fd)!=0){
49          fprintf(stderr, "Failed to close I2C interface %s Error: %s\n", I2C_DEV_PATH, strerror(errno));
50          return -1;
51      }
52      return 0;
53  }
```

Figure 4.15: Implementation of the MLX90614 Class - 1

The `openI2C()`, `setSlaveAddress()` and `enableCheckSum()` methods provide the proper configuration for the communication with this sensor using the I2C protocol. Through the use of the `ioctl()` system call, it is possible to manipulate the underlying device parameters, ensuring effective and reliable communication with the sensor.

```
55  /**
56   * @brief Opens the I2C interface.
57   * @return 0 on success, -1 on failure.
58   */
59  int MLX90614::openI2C(){
60      fd = open(I2C_DEV_PATH, O_RDWR); // open i2c bus
61      if (fd < 0) {
62          fprintf(stderr, "Failed to open I2C interface %s Error: %s\n", I2C_DEV_PATH, strerror(errno));
63          return -1;
64      }
65      return 0;
66  }
67
68  /**
69   * @brief Sets the slave address for the MLX90614 sensor.
70   * @return 0 on success, -1 on failure.
71   */
72  int MLX90614::setSlaveAddress(){
73      if (ioctl(fd, I2C_SLAVE, MLX90614_I2CADDR) != 0) {
74          fprintf(stderr, "Failed to select I2C slave device! Error: %s\n", strerror(errno));
75          return -1;
76      }
77      return 0;
78  }
79
80  /**
81   * @brief Enables checksum for the MLX90614 sensor.
82   * @return 0 on success, -1 on failure.
83   */
84  int MLX90614::enableCheckSum(){
85      if (ioctl(fd, I2C_PEC, 1) != 0) {
86          fprintf(stderr, "Failed to enable SMBus packet error checking, error: %s\n", strerror(errno));
87          return -1;
88      }
89      return 0;
90  }
```

Figure 4.16: Implementation of the MLX90614 Class - 2

Finally, the implementation of `readData()`, `readTemperature()` and `getTemperature()` methods are presented, showcasing the core functionalities responsible for reading sensor data and retrieving temperature information. The `readData()` method uses the `SMBus` level transations (which is a subset from the I2C protocol). The `readTemperature()` reads the temperature from the thermopile sensor, performing the necessary conversion to Celsius degrees. Lastly, the `getTemperature()` method returns the value of the `temperature` variable, which is a private attribute of this class.

```
92  /**
93   * @brief Reads data from the MLX90614 sensor using I2C_SMBUS transaction.
94   * @param data Reference to i2c_smbus_data structure.
95   * @param command I2C command to read data.
96   * @return 0 on success, -1 on failure.
97   */
98  int MLX90614::readData(i2c_smbus_data& data, char command){
99      // Build request structure
100     struct i2c_smbus_ioctl_data sdat = {
101         .read_write = I2C_SMBUS_READ,
102         .command = command,
103         .size = I2C_SMBUS_WORD_DATA,
104         .data = &data
105     };
106
107     // Perform the actual request
108     if (ioctl(fd, I2C_SMBUS, &sdat) < 0) {
109         return -1;
110     }
111     return 0;
112 }
113
114 /**
115  * @brief Reads the temperature from the MLX90614 sensor.
116  */
117 void MLX90614::readTemperature(){
118     if (readData(data, command)!=0) {
119         fprintf(stderr, "Failed to perform I2C_SMBUS transaction, error: %s\n", strerror(errno));
120         return;
121     }
122     temperature = (double) data.word;
123     temperature = (temperature * 0.02) - 0.01;
124     temperature = temperature - 278.15;
125 }
126
127 /**
128  * @brief Gets the temperature from the MLX90614 sensor.
129  * @return The temperature in Celsius.
130  */
131 double MLX90614::getTemperature() const{
132     return temperature;
133 }
```

Figure 4.17: Implementation of the MLX90614 Class - 3

rgbDevice Class

In figure 4.18 it is represented the implementation of the `rgbDevice` class. The `init()` method initializes the device by opening the associated file in write-only mode, and the `exit()` method cleans up resources by closing the file. The core functionality is in the `writeColor()` method, which writes a color value to the RGB LED device. It takes an integer color value as input, converts it to an ASCII value, and writes it to the device file. Obviously, the file descriptor is expected to be open before calling this method.

```cpp
#include "/home/pedro/Desktop/driverRGBLED/rgbDriver.h"

/**
 * @brief Default constructor for the rgbDevice class.
 */
rgbDevice::rgbDevice(){}

/**
 * @brief Default destructor for the rgbDevice class.
 */
rgbDevice::~rgbDevice(){}

/**
 * @brief Initializes the RGB LED device by opening the corresponding device file in write-only mode.
 * @return 0 on success, -1 on failure.
 */
int rgbDevice::init(){
    LEDfd=open(LED_DEVICE_PATH, O_WRONLY);
    if(LEDfd<0){
        perror("Failed to open LED Device !\n");
        return -1;
    }
    return 0;
}

/**
 * @brief Exits and cleans up resources by closing the RGB LED device file.
 * @return 0 on success, -1 on failure.
 */
int rgbDevice::exit(){
    if(close(LEDfd)!=0){
        perror("Failed to close LED Device !\n");
        return -1;
    }
    return 0;
}

/**
 * @brief Writes a color value to the RGB LED device.
 * @param color Integer color value.
 */
void rgbDevice::writeColor(int color){
    char buffer[2];
    // Convert to ASCII value
    buffer[0] = static_cast<char>(color + '0');
    buffer[1] = '\0';
    if(LEDfd<0){
        perror("Device not open !\n");
        return;
    }
    write(LEDfd,buffer,1);
}
```

Figure 4.18: Implementation of the rgbDevice Class

CProduct Class

The provided code defines the member functions for the `CProduct` class, including various constructors for the class objects, destructor, and various setter and getter methods.

```
#include <iostream>
#include "CProduct.h"

CProduct::CProduct(){}
CProduct::CProduct(string Name, string Barcode, string Price,string Suplier): Buyed_Qty(0),Scanned_counter(0)
{
    this->Name = Name;
    this->Barcode = Barcode;
    this->Price = Price;
    this->Suplier = Suplier;
}
CProduct::CProduct(string Name, string Barcode, string Price,int Quantity, string Suplier): Buyed_Qty(0),Scanned_counter(0)
{
    this->Name = Name;
    this->Barcode = Barcode;
    this->Price = Price;
    this->Quantity = Quantity;
    this->Suplier = Suplier;
}

CProduct::CProduct( string Name, size_t Buyed_Qty): Buyed_Qty(0),Scanned_counter(0)
{
    this->Name = Name;
    this->Buyed_Qty = Buyed_Qty;
}

CProduct::~CProduct()
{
    this->Name = "";
    this->Barcode = "";
    this->Price = "";
    this->Quantity = 0;
    this->Suplier = "";
}

void CProduct::setName(string NewName)       {this->Name = NewName;}
void CProduct::setPrice(string NewPrice)     {this->Price = NewPrice;}
void CProduct::setBarcode(string NewBarcode) {this->Barcode = NewBarcode;}
void CProduct::IncQuantity()                 {this->Quantity += 1;}
void CProduct::DecQuantity()                 {this->Quantity -= 1;}
void CProduct::setSuplier(string NewSuplier) {this->Suplier = NewSuplier;}
void CProduct::setBuyed_Qty(size_t Buyed_Qty){this->Buyed_Qty = Buyed_Qty;}
void CProduct::incBuyed_Qty()                {this->Buyed_Qty +=1;}
void CProduct::setScansAmount(int Amount)    {this->Scanned_counter = Amount;}
void CProduct::incScansAmount(){this->Scanned_counter +=1; }

string CProduct::getName()        {return Name;}
string CProduct::getBarcode()     {return Barcode;}
string CProduct::getPrice()       {return Price;}
string CProduct::getSuplier()     {return Suplier;}
size_t CProduct::getQuantity()    {return Quantity;}
size_t CProduct::getBuyed_Qty()   {return Buyed_Qty;};
int CProduct::getScansAmount() {return Scanned_counter;}
```

Figure 4.19: Implementation of the CProduct Class

The `CProduct` class provides a comparison operation, implemented as a member method. This operator facilitates the comparison between two instances of the `CProduct` class for equality, taking into account several attributes. The method returns a boolean value indicating whether all these attributes are equal between the two objects.

```
bool CProduct::operator==(const CProduct &other)const
{
    return (Name == other.Name && Barcode == other.Barcode && Price == other.Price && Quantity == other.Quantity && Suplier == other.Suplier);

}
```

Figure 4.20: Implementation of the CProduct Class

CDatabaseManage Class

The figures 4.21 to 4.28, provide the implementation of the methods of the `CDatabaseManage` class.

```
#include "CDatabaseManage.h"
using namespace std;

/**
 * @brief Default constructor for CDatabaseManage.
 */
CDatabaseManage::CDatabaseManage(){}

/**
 * @brief Parameterized constructor for CDatabaseManage.
 *
 * @param DB_Path The path to the SQLite database.
 * @param db The SQLite database.
 */
CDatabaseManage::CDatabaseManage(const char* DB_Path,sqlite3* db)
{
    this->DB_Path = DB_Path;
    this->db = db;
}

/**
 * @brief Destructor for CDatabaseManage.
 */
CDatabaseManage::~CDatabaseManage(){ sqlite3_close(db); }

```

Figure 4.21: Implementation of the CDatabaseManage Class - Constructors and Destructor

This function preprares an SQL query to find a product in the database based on its barcode and returns a `CProduct` object representing the found product.

```
/**
 * @brief Finds a product in the database based on its barcode.
 *
 * @param Barcode The barcode of the product to find.
 * @param db The SQLite database.
 * @return The found CProduct object or an empty CProduct if not found.
 */
static CProduct DB_Product_find(string Barcode,sqlite3* db)
{
    string sqlQuery = "SELECT * FROM productsDB WHERE Barcode = '" + Barcode + "'";
    sqlite3_stmt *stmt;

    int rc = sqlite3_prepare_v2(db, sqlQuery.c_str(), -1, &stmt, nullptr);
    if (rc != SQLITE_OK)    //Error Handle
    {
        cerr << "Error preparing SQL statement: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        return CProduct("", "", "", 0, "");
    }

    rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW)   //Product Found
    {
        // Fetch the product details directly using sqlite3_column_* functions
        string name     = (const char *)sqlite3_column_text(stmt, 0);
        string price    = (const char *)sqlite3_column_text(stmt, 2);
        int quantity = sqlite3_column_int(stmt, 3);
        string supplier = (const char *)sqlite3_column_text(stmt, 4);

        sqlite3_finalize(stmt);
        return CProduct(name, Barcode, price, quantity, supplier);
    }

    cerr << "Product not found in the database." << std::endl;
    sqlite3_finalize(stmt);
    return CProduct("", "", "", 0, "");
}
```

Figure 4.22: Implementation of the CDatabaseManage Class - Find product in database function

This method reads data from the JSON file and initializes the database with the product information. It returns a vector of `CProduct` objects. All the actions involving the database were executed with a prepared statement to avoid database leaks.

```
65  /**
66   * @brief Initializes the database using data from a JSON file.
67   *
68   * @param JsonFilePath The path to the JSON file.
69   * @param db The SQLite database.
70   * @return A vector of CProduct objects initialized from the JSON file.
71   */
72  vector<CProduct> CDatabaseManage::InitDB(string JsonFilePath,sqlite3* db) {
73
74      vector<CProduct> products;
75      char*err;
76      ifstream jsonFile(JsonFilePath);
77      Json::Value JsonProduct;
78      Json::Reader reader;
79      reader.parse(jsonFile,JsonProduct);
80
81      for (const Json::Value&JsonINDEX : JsonProduct) {
82          string name = JsonINDEX["Name"].asString();
83          string barcode = JsonINDEX["Barcode"].asString();
84          string price = JsonINDEX["Price"].asString();
85
86          string suplier = JsonINDEX["Suplier"].asString();
87
88          int quantity = 0;           //Sets a default value so if the product doesnt exist the quantity will be 0
89
90
91          //--------Quantity fetch from the DataBase------
92          string fetch_qnty = "SELECT Quantity FROM productsDB WHERE Barcode = '" + barcode + "';";
93          sqlite3_stmt* stmt;
94          int rc = sqlite3_prepare_v2(db, fetch_qnty.c_str(), -1, &stmt, NULL);
95
96          if (rc == SQLITE_OK) {
97              if (sqlite3_step(stmt) == SQLITE_ROW) {
98                  // Extract the quantity value from the result set
99                  quantity = sqlite3_column_int(stmt, 0);
100             }
101
102             sqlite3_finalize(stmt);
103         } else {
104             cerr << "Error preparing SQL statement: " << sqlite3_errmsg(db) << endl;
105         }
106         //---------------------------------------------
```

If the product read from JSON doesn't exist in the database, it will be inserted in it by preparing a SQL statement and binding its parameters to the statement. Using statements instead of executing directly, assures a better performance and security in our system and database.

```
111         if (DB_Product_find(barcode, db) == CProduct("", "", "", 0, "")) {
112             string write_cmd = "INSERT INTO productsDB (Name, Barcode, Price, Quantity, Suplier) VALUES (?, ?, ?, ?, ?)";
113             sqlite3_stmt* stmt;
114
115             int rc = sqlite3_prepare_v2(db, write_cmd.c_str(), -1, &stmt, nullptr);
116             if (rc != SQLITE_OK) {
117                 cerr << "Error preparing DB insert statement: " << sqlite3_errmsg(db) << endl;
118
119             }
120
121             // Bind parameters to the statement
122             sqlite3_bind_text(stmt, 1, name.c_str(), -1, SQLITE_STATIC);
123             sqlite3_bind_text(stmt, 2, barcode.c_str(), -1, SQLITE_STATIC);
124             sqlite3_bind_text(stmt, 3, price.c_str(), -1, SQLITE_STATIC);
125             sqlite3_bind_int(stmt, 4, quantity);
126             sqlite3_bind_text(stmt, 5, suplier.c_str(), -1, SQLITE_STATIC);
127
128             // Execute the prepared statement
129             rc = sqlite3_step(stmt);
130             if (rc != SQLITE_DONE) {
131                 cerr << "Error inserting into DB: " << sqlite3_errmsg(db) << endl;
132             }
133
134             // Finalize the statement
135             sqlite3_finalize(stmt);
136         }
```

In order to a manager to be able to update the price (by updating the JSON file), the price is the only variable that will be read from JSON and written into the database even if the product already exists.

```
137         else
138         {
139             // Product exists, update the Price
140             string update_cmd = "UPDATE productsDB SET Price = ? WHERE Barcode = ?";
141             sqlite3_stmt* stmt;
142
143             int rc = sqlite3_prepare_v2(db, update_cmd.c_str(), -1, &stmt, nullptr);
144             if (rc != SQLITE_OK) {
145                 cerr << "Error preparing DB update statement: " << sqlite3_errmsg(db) << endl;
146                 return products; // Assuming you want to exit the function on an error
147             }
148
149             // Bind parameters to the statement
150             sqlite3_bind_text(stmt, 1, price.c_str(), -1, SQLITE_STATIC);
151             sqlite3_bind_text(stmt, 2, barcode.c_str(), -1, SQLITE_STATIC);
152
153             // Execute the prepared statement
154             rc = sqlite3_step(stmt);
155             if (rc != SQLITE_DONE) {
156                 cerr << "Error updating product in DB: " << sqlite3_errmsg(db) << endl;
157             }
158
159             // Finalize the statement
160             sqlite3_finalize(stmt);
161         }
162     }
163     return products;        //Returns the full object vector
164 }
165
```

Figure 4.23: Implementation of the CDatabaseManage Class - Initialization of database 2

The `addOperation()` function searches in the `CProducts` vector for the product we want to add and if it founds it, its quantity on stock is incremented internally in the system and also in the database. The add operation only adds quantity, it doesn't add new products. To do so, the user must change the JSON file.

```
167 /**
168  * @brief Adds a product from the database and updates its quantity.
169  *
170  * @param Scanned_barcode The barcode of the scanned product.
171  * @param products The vector of CProduct objects.
172  * @param db The SQLite database.
173  */
174 void CDatabaseManage::addOperation(string Scanned_barcode, vector<CProduct> &products, sqlite3* db)
175 {
176     for (CProduct &product : products) {
177         if (DB_Product_find(Scanned_barcode, db).getBarcode() == product.getBarcode()) {
178             product.IncQuantity();
179
180             // Using prepared statement to update quantity
181             string updateQtyQuery = "UPDATE productsDB SET Quantity = ? WHERE Barcode = ?";
182             sqlite3_stmt* stmt;
183
184             int rc = sqlite3_prepare_v2(db, updateQtyQuery.c_str(), -1, &stmt, nullptr);
185             if (rc != SQLITE_OK) {
186                 cerr << "Error preparing DB update statement: " << sqlite3_errmsg(db) << endl;
187                 return;
188             }
189
190             // Bind parameters to the statement
191             sqlite3_bind_int(stmt, 1, product.getQuantity());
192             sqlite3_bind_text(stmt, 2, product.getBarcode().c_str(), -1, SQLITE_STATIC);
193
194             // Execute the prepared statement
195             rc = sqlite3_step(stmt);
196             if (rc != SQLITE_DONE) {
197                 cerr << "Error updating DB product quantity: " << sqlite3_errmsg(db) << endl;
198             }
199
200             // Finalize the statement
201             sqlite3_finalize(stmt);
202         }
203     }
204 }
205
206
```

Figure 4.24: Implementation of the CDatabaseManage Class - Add product to database method

The `remOperation()` does the same as the `addOperation()` method but instead of incrementing it decrements the quantity of the scanned product.

```
/**
 * @brief Removes a product from the database and updates its quantity.
 *
 * @param Scanned_barcode The barcode of the scanned product.
 * @param products The vector of CProduct objects.
 * @param db The SQLite database.
 */
void CDatabaseManage::remOperation(string Scanned_barcode, vector<CProduct> &products, sqlite3* db)
{
    for (CProduct &product : products) {
        if (DB_Product_find(Scanned_barcode, db).getBarcode() == product.getBarcode()) {
            if (product.getQuantity() > 0) product.DecQuantity();

            // Using prepared statement to update quantity
            string updateQtyQuery = "UPDATE productsDB SET Quantity = ? WHERE Barcode = ?";
            sqlite3_stmt* stmt;

            int rc = sqlite3_prepare_v2(db, updateQtyQuery.c_str(), -1, &stmt, nullptr);
            if (rc != SQLITE_OK) {
                cerr << "Error preparing DB update statement: " << sqlite3_errmsg(db) << endl;
                return;
            }

            // Bind parameters to the statement
            sqlite3_bind_int(stmt, 1, product.getQuantity());
            sqlite3_bind_text(stmt, 2, product.getBarcode().c_str(), -1, SQLITE_STATIC);

            // Execute the prepared statement
            rc = sqlite3_step(stmt);
            if (rc != SQLITE_DONE) {
                cerr << "Error updating DB product quantity: " << sqlite3_errmsg(db) << endl;
            }

            // Finalize the statement
            sqlite3_finalize(stmt);
        }
    }
}
```

Figure 4.25: Implementation of the CDatabaseManage Class - Remove product to database method

The search method uses the `DB_Product_find()` function and returns the found product.

```
/**
 * @brief Searches for a product in the database based on its barcode.
 *
 * @param Barcode The barcode of the product to search.
 * @param db The SQLite database.
 * @return The found CProduct object or an empty CProduct if not found.
 */
CProduct CDatabaseManage::searchOperation(string Barcode,sqlite3* db)
{
    return DB_Product_find (Barcode,db);
}
```

Figure 4.26: Implementation of the CDatabaseManage Class - Search for a product in the database method

And finally, the last method on our `CDatabaseManage` Class, the `saleOperation()` method, is used to update the `ThingSpeak` graphics with the purchased quantity of each product.

After receiving the products list, it fetches the products that were bought and places their quantity on a array(followed by a comma ”,”) in order to read it when formatting the final .csv file (Comma-separated values). To send information to the website we have to perform a HTTP POST request (using the .csv file). Because `ThingSpeak` only receives information every 15 seconds, we must send one request per sale, due to having multiple products in the same sale we cannot send multiple requests instantly nor in a 15 seconds interval. After searching all the products, if any product was bought, it will then start formatting

the rest of the file which must have the following format example:

```
"write_api_key=XXXXXXX&time_format=absolute&updates=20180614T12:12:220500,1,,3,0.4"
```

Expecting the API key that references our TS account, the exact time that the request is made and the values of the respective graphics/fields. As you can see on figure 4.27, when something is bought we start by calculating the time (using `CTime`) from the current year to the current second.

```
193  /**
194   * @brief Conducts a sale operation and updates relevant information.
195   *
196   * @param products_list The vector of CProduct objects representing products in the sale.
197   */
198  void CDatabaseManage::saleOperation(vector<CProduct> &products_list)
199  {
200      string FieldValues[products_list.size()];
201      size_t LastProduct = products_list.size() - 1;
202      int Flag_Buyed;
203
204      //PURCHASE CHECKING...
205      for (int x = 0; x <= LastProduct; x++)
206      {
207          if (products_list[x].getBuyed_Qty() > 0)       //If the product was bought
208          {
209              products_list[x].setScansAmount(0);         //Resets the num of scanns for the next purchase start with 0 scans in each product
210              Flag_Buyed = 1;
211              FieldValues[x] = to_string(products_list[x].getBuyed_Qty()) + ",";
212              cout << FieldValues[x] << endl;
213              products_list[x].setBuyed_Qty(0);           //resets the buyed quantity for future purchases
214          }
215          else FieldValues[x] = ",,";
216      }
217      // Because the last product doesn't have a comma next to him
218      if (products_list[LastProduct].getBuyed_Qty() > 0) //If the product was bought
219      {
220          products_list[LastProduct].setScansAmount(0);   //Resets the num of scanns for the next purchase start with 0 scans in each product
221          Flag_Buyed = 1;
222          FieldValues[LastProduct] = to_string(products_list[LastProduct].getBuyed_Qty());
223          cout << FieldValues[LastProduct] << endl;
224          products_list[LastProduct].setBuyed_Qty(0);     //resets the buyed quantity for future purchases
225      }
226      else FieldValues[LastProduct] = "";
227
228      //In case of purchase it will update the website grafics
229      if (Flag_Buyed)
230      {
231          Flag_Buyed = 0;
232
233          //DATE AND TIME  FETCHING...
234          //time_t now = time(0);
235          //tm* localtime = gmtime(&now);
236
237          time_t now = time(0);
238          tm* localTime = localtime(&now);
239          int year = localTime->tm_year + 1900;
240          int month = localTime->tm_mon + 1;
241
242          //string DateTime = to_string(year)+"-"+to_string(month)+"-"+ to_string(localtime->tm_mday)+"T"+to_string(localtime->tm_hour) +":"+to_string(localtime->tm_min)+":"+to_string(localtime->tm_sec);
243          string DateTime = to_string(year) + "-" + to_string(month) + "-" + to_string(localTime->tm_mday) + "T" +
244                            to_string(localTime->tm_hour) + ":" + to_string(localTime->tm_min) + ":" + to_string(localTime->tm_sec);
245
```

Figure 4.27: Implementation of the CDatabaseManage Class - Sale for a product in the database method

Next up we start creating/writing to the .csv file, starting with the API key, the Date&Time and then the previously created array. After that, we just need to read the file's content and store it in a string variable; initialize cURL; set the URL that contains our channel number and the .csv file name. After, specify the data that will be sent in the POST request and finally perform the request and clean up cURL. This way we are able to send multiple information into the the website instantly.

The data is updated, shortly after the request is made and the sales charts are very illustrative as can be seen in figure 4.29.

```
247
248         //WRITE TO THE .CSV FILE WITH THE UPDATED PURSHASE INFROMATION
249         ofstream outputFile("/etc/basedados/bulk_update.csv");
250         if (!outputFile.is_open())
251         {
252             cerr << "Erro ao abrir o arquivo CSV para escrita." << endl;
253         }
254         else
255         {
256             outputFile << "write_api_key=LDR330TRY4KZZOK7&time_format=absolute&updates=" + DateTime +",";   //Writes first the inicial infromation and then the Grafic values
257             //cout << "WRITE TO THE .CSV FILE WITH THE UPDATED PURSHASE INFROMATION" << endl;
258             for (int i = 0; i <= LastProduct; i++)
259             {
260                 outputFile << FieldValues[i];
261             }
262         }
263         outputFile.close();                                                                    //closes the the file on write mode
264
265         //READS THE UPDATED INFROMATION FROM THE .CSV FILE
266         ifstream file("/etc/basedados/bulk_update.csv");
267         if (!file.is_open())
268         {
269             cerr << "Error opennig CSV file." << endl;
270         }
271         else
272         {
273             //cout << "READS THE UPDATED INFROMATION FROM THE .CSV FILE" << endl;
274             //Reads from the .csv file into the "file_contents" string
275             stringstream buffer;
276             buffer << file.rdbuf();
277             string file_contents = buffer.str();
278             file.close();                       //closes the read mode csv
279
280             CURL *cURL;
281             CURLcode res;
282
283             cURL = curl_easy_init();            //cURL initialization
284             if(cURL) {
285
286                 //Sets the URL with the desiared channel
287                 curl_easy_setopt(cURL, CURLOPT_URL, "http://api.thingspeak.com/channels/2349110/bulk_update.csv");
288
289                 //Sets the Data the will be sent in the request
290                 curl_easy_setopt(cURL, CURLOPT_POSTFIELDS, file_contents.c_str());      //Needs to be a Char* because cURL is a C language library
291
292                 //Executes the request
293                 res = curl_easy_perform(cURL);
294
295                 //Performing Error check
296                 if(res != CURLE_OK) cout << "curl_easy_perform() failed: " << curl_easy_strerror(res) << endl;
297
298                 curl_easy_cleanup(cURL);
299             }
300             curl_global_cleanup();
301         }
302     }
303 }
```

Figure 4.28: Implementation of the CDatabaseManage Class - Search for a product in the database method

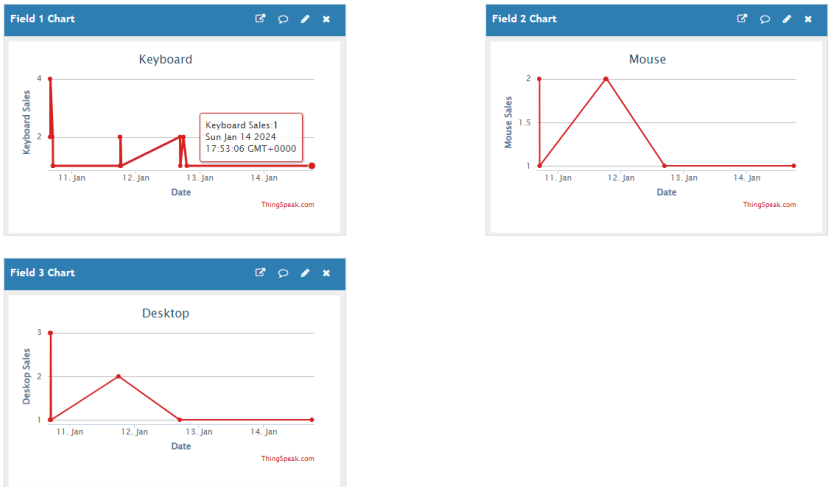

Figure 4.29: Sale's Charts - ThingSpeak

mfsWrapper Class

The figure 4.30 provides the respective code for the implementation of the `mfsWrapper` Class. The implementation of this class is quite simple; the public methods only emit the respective signals with the correct parameters. These signals will be connected to slots in another UI class, providing a reliable communication between different components of the application.

```
1  #include "mfswrapper.h"
2
3  mfsWrapper::mfsWrapper(QObject *parent) : QObject(parent) {}
4
5  mfsWrapper& mfsWrapper::getInstance() {
6      static mfsWrapper instance;
7      return instance;
8  }
9
10 void mfsWrapper::emitUpdateSignal(double value) {
11     emit updateValueSignal(value);
12 }
13
14 void mfsWrapper::emitScanSignal(std::string code){
15     emit updateScanSignal(code);
16 }
17
18 void mfsWrapper::emitWarningSignal(int value){
19     emit updateWarningSignal(value);
20 }
21
22 void mfsWrapper::emitTableUpdateSale(){
23     emit updateTableSale();
24 }
25
26 void mfsWrapper::emitAddSignal(int value){
27     emit addSignal(value);
28 }
29
30 void mfsWrapper::emitRemoveSignal(int value){
31     emit removeSignal(value);
32 }
33
```

Figure 4.30: Implementation of class mfsWrapper

MFS Class

The presented code outlines the functionality of the primary class, MFS, within this system. Acting as the central element, this class oversees a range of tasks involving temperature sensing, scanning, LED manipulation, and database management.

Firstly, this class has some constant expressions representing the Linux shell commands. These commands are intended for loading and unloading the necessary kernel modules in this Linux system.

```
5  constexpr const char* systemCallRGB = "insmod /etc/rgbDriver.ko";
6  constexpr const char* systemI2CDEV = "insmod /lib/modules/5.10.92-v7l/kernel/drivers/i2c/i2c-dev.ko";
7  constexpr const char* systemI2CBCM2835 = "insmod /lib/modules/5.10.92-v7l/kernel/drivers/i2c/busses/i2c-bcm2835.ko";
8  constexpr const char* systemRemRGB = "rmmod rgbDriver";
9  constexpr const char* systemRemI2CDEV = "rmmod i2c-dev";
10
```

Figure 4.31: Implementation MFS Class - Constant Expressions

The main purpose of the first two methods is to facilitate interaction with the core functionalities of the `MFS` class, primarily for unlocking the threads from the UI.

It is also possible to observe the implementation of the method for retrieving the status of the temperature sensor, as well as the `signalHandler()` function to handle the Linux signal `SIGALRM`, which is responsible to trigger the temperature check. Lastly, the method `receive_value(int value)` to manage the database operation.

The figure 4.32 visually represents the implementation of these `MFS` class methods.

The three methods represented in 4.33 contribute to the functionality related to product purchases and quantity management. Together, these methods enhance the capabilities of the `MFS` class in managing product-related data, providing essential functionality for purchase and quantity tracking.

```
/*--------------- FUNCTIONS FOR WRAPPER ------------*/
/**
 * @brief Signals GUI from stock check.
 */
void MFS::signaltoGUIfromStockCheck() {
    pthread_mutex_lock(&condMutex);
    signalFromStockcheck=true;
    pthread_cond_signal(&condToGUI);
    pthread_mutex_unlock(&condMutex);
}

/**
 * @brief Signals for initiating a scan.
 */

void MFS::signalScan(){
    pthread_mutex_lock(&condMutex);
    pthread_cond_signal(&condScan);
    LedMode = BLUE;
    ledModeChanged=true;
    pthread_cond_signal(&condLED);
    pthread_mutex_unlock(&condMutex);
}

/**
 * @brief Gets the status of the MLX sensor.
 * @return Status of the MLX sensor.
 */
int MFS::getMLXStatus(){
    return mlxStatus;
}

/*----------------------SIGNAL----------------------*/
/**
 * @brief Signal handler function.
 */
void MFS::signalHandler(){
    pthread_mutex_lock(&condMutex);
    pthread_cond_signal(&condReadTemp);
    pthread_mutex_unlock(&condMutex);
    alarm(2);
}
/*--------------------------------------------------*/

/*----------------------DATABASE OPERATION----------------------*/
/**
 * @brief Receives a value for database operation.
 * @param value The value for database operation.
 */
void MFS::receive_Value(int value)
{
    MFS& mfs = MFS::getInstance(); //cria uma referência à instância única da classe MFS usando o padrão Singleton, permitin
    mfs.DBoperation = value;
}
```

Figure 4.32: Implementation MFS Class - Public Methods

```
/**
 * @brief Signals that purchase is completed.
 */
void MFS::PurshaseCompleted_signal()
{
    pthread_mutex_lock(&mtDatabaseManage);
    pthread_cond_signal(&condDatabaseManage);  // sinalize a condição para acordar a thread
    pthread_mutex_unlock(&mtDatabaseManage);
}

/**
 * @brief Increments the quantity of a product in the vector.
 * @param product The product to increment quantity.
 */
void MFS::IncBuyedQtyVector(CProduct product)
{
    for (int i = 0; i < ProductsVector.size(); ++i)
    {
        if (ProductsVector[i].getBarcode() == product.getBarcode())
        {
            ProductsVector[i].incBuyed_Qty();
            break;
        }
    }
}

/**
 * @brief Resets the quantity of all products in the vector.
 */
void MFS::ResetQtyVector()
{
    for (int i = 0; i < ProductsVector.size(); ++i)
    {
        ProductsVector[i].setBuyed_Qty(0);
    }
}
/*--------------------------------------------------*/
```

Figure 4.33: Implementation MFS Class - Public Methods

The `MFS` class is designed to implement the Singleton design pattern, ensuring that only one instance of the class exists throughout the system's lifetime. The `MFS()` constructor is explicitly defined as private, preventing external instantiation of the class. The `getInstance()` method acts as the gateway to access the single instance of the `MFS` class. This method creates the instance on the first call and subsequently returns a reference to it.

```
/**
 * @brief Constructor for the MFS class.
 */
MFS::MFS(){}

/**
 * @brief Destructor for the MFS class.
 */
MFS::~MFS(){}

/*----------------------------singleton--------------------*/
/**
 * @brief Gets the instance of the MFS class.
 * @return Reference to the instance of the MFS class.
 */
MFS& MFS::getInstance() {
    static MFS instance;  // A instância é criada apenas na primeira chamada
    return instance;
}
```

Figure 4.34: Implementation MFS Class - Public Methods

Moving on to device operations, these methods are designed to encapsulate interactions with specific devices that are part of the `MFS` class composition. The names of these methods provide a clear explanation of their respective functionalities.

```
/*---------DEVICE OPERATIONS---------*/
/**
 * @brief Performs a scanner read operation.
 */
void MFS::performScannerRead(){
    scanner.readScanner();
}

/**
 * @brief Performs a sensor read operation.
 */
void MFS::performSensorRead(){
    TempSensor.readTemperature();
}

/**
 * @brief Gets the temperature value from the sensor.
 * @return The temperature value.
 */
double MFS::getSensorValue() const{
    return TempSensor.getTemperature();
}

/**
 * @brief Gets the scanner code as a string.
 * @return The scanner code as a string.
 */
string MFS::getCodeString() const{
    return scanner.getString();
}

/**
 * @brief Writes a color value to the RGB LED device.
 * @param value The color value to write.
 */
void MFS::writeColor(int value){
    led.writeColor(value);
}
/*-----------------------------------------------*/
```

Figure 4.35: Implementation MFS Class - Public Methods

Moving on to operational aspects, the following methods of the `MFS` class manage thread creation, kernel driver insertion, device initialization and thread synchronization.

First the `createThreads()` method initiates the new threads in the calling process. Following that, `inserDrivers()` is responsible for inserting vital kernel drivers into the Linux system. Next, the `initializeDevices()` handles the initialization of the devices. Lastly, the `waitforThreads()` ensures a synchronized mechanisms, temporarily pausing the program's progression until all concurrently running threads finish their assigned task.

```
/*-----------------------------------------------*/

/** ------- @brief Startup System Sequence -------  **/

/**
 * @brief Creates threads for various operations.
 */
void MFS::createThreads(){
    //pthread_create (&thread_id,     &thread_attr,        thread_routine, NULL);
    //HIGH PRIO
    pthread_create (&tCheckTemp_id, &Highprio_RR_attr,  tcheckTemperature, NULL);

    //MEDIUM PRIO
    pthread_create (&tScan_id,      &Mediumprio_RR_attr, tscan, NULL);
    pthread_create (&tDBM_id,       &Mediumprio_RR_attr, tdatabaseManage, this); //this is necessary so we can pass an obje

    //LOW PRIO
    pthread_create (&tToGUI_id,     &Lowprio_RR_attr,    ttoGUI, NULL);
    pthread_create (&tLED_id,       &Lowprio_RR_attr,    tLED, NULL);
}

/**
 * @brief Inserts necessary kernel drivers.
 */
void MFS::insertDrivers(){
    system(systemCallRGB);
    system(systemI2CDEV);
    //system(systemI2CBCM2835);
}

/**
 * @brief Initializes devices such as the RGB LED, temperature sensor, and scanner.
 */
void MFS::initializeDevices(){
    led.init();
    led.writeColor(ON);  //todos a 1
    mlxStatus=TempSensor.init();
    scanner.init();
}

/**
 * @brief Waits for threads to finish.
 */
void MFS::waitforThreads(){
    //pthread_join (thread_id, NULL);
    pthread_join (tCheckTemp_id, NULL);
    pthread_join (tScan_id, NULL);
    pthread_join (tDBM_id, NULL);
    pthread_join (tToGUI_id, NULL);
    pthread_join (tLED_id, NULL);
}
```

Figure 4.36: Implementation MFS Class - Public Methods

The `initializeAttributes()` method plays a pivotal role in configuring and setting up thread attributes. This method uses the `POSIX thread API` (similar to the `createThreads()` and `waitforThreads()` methods), employing four distinct attribute objects: `thread_attr`, `Highprio_RR_attr`, `Mediumprio_RR_attr`, and `Lowprio_RR_attr`. Setting up the thread policies and parameters is accomplished using the `SetupThread()` function, illustrated in Figure 4.37. Moving on to the `startupSystem()`, this method composes the initialization sequence. The sequence of the startup process can be easily interpreted from Figure 4.38.

Our system operates as a remote device and is equipped with a battery holder to ensure uninterrupted funcionality. Unlike systems with a predefined shutdown sequence, our design deviates from this conven-

```
/**
 * @brief Sets up thread attributes.
 * @param prio Priority value for the thread.
 * @param pthread_attr Thread attributes.
 * @param pthread_param Thread parameters.
 */
void MFS::SetupThread(int prio,pthread_attr_t *pthread_attr,struct sched_param *pthread_param){
    //Assigns priority value to sched_param structure
    pthread_param->sched_priority = prio;

    // Sets the priority variable of the attribut
    pthread_attr_setschedparam (pthread_attr, pthread_param);

    // This thread attributs will be set manualy and not inherited from the calling thread
    //pthread_attr_setinheritsched (pthread_attr, PTHREAD_EXPLICIT_SCHED);

    // Sets the schedule policy for round robin
    pthread_attr_setschedpolicy (pthread_attr, SCHED_RR);
}
```

Figure 4.37: Implementation MFS Class - Setup Threads Method

```
/**
 * @brief Initializes thread attributes.
 */
void MFS::initializeAttributes(){
    //PTHREAD INITIALIZATIONS
    // Step 1: initialize attribute object
    pthread_attr_init (&thread_attr);
    pthread_attr_getschedpolicy (&thread_attr, &thread_policy);
    pthread_attr_getschedparam (&thread_attr, &thread_param);

    //Initialize attribute object
    pthread_attr_init (&Highprio_RR_attr);
    pthread_attr_getschedpolicy (&Highprio_RR_attr, &thread_policy);
    pthread_attr_getschedparam (&Highprio_RR_attr, &Highprio_RR_param);

    pthread_attr_init (&Mediumprio_RR_attr);
    pthread_attr_getschedpolicy (&Mediumprio_RR_attr, &thread_policy);
    pthread_attr_getschedparam (&Mediumprio_RR_attr, &Mediumprio_RR_param);

    pthread_attr_init (&Lowprio_RR_attr);
    pthread_attr_getschedpolicy (&Lowprio_RR_attr, &thread_policy);
    pthread_attr_getschedparam (&Lowprio_RR_attr, &Lowprio_RR_param);

    //Setup attributes and parameters
    SetupThread(25,&Highprio_RR_attr,&Highprio_RR_param);
    SetupThread(10,&Mediumprio_RR_attr,&Mediumprio_RR_param);
    SetupThread(5,&Lowprio_RR_attr,&Lowprio_RR_param);
}


/**
 * @brief Startup procedure for the system.
 */
void MFS::startupSystem(){
    insertDrivers();
    sqlite3_open(DB_Path, &db);
    ProductsVector = DBM.InitDB(JsonFilePath,db);
    initializeDevices();
    initializeAttributes();
    createThreads();
    waitforThreads();
}
```

Figure 4.38: Implementation MFS Class - Startup Sequence

tional approach. This decision is based from the need for manual control over the power state. To power down the system, users are required to manually turn it off.

Moving to the implementation of the thread functions, let's first begin with the `tcheckTemperature` thread, which is the highest priority thread in our system. This thread is crucial for monitoring the system's temperature in real-time. This thread function waits for a condition variable, which is triggered each three seconds by Linux's signal `SIGALRM`. When the signal is received, the function locks the associated mutex to prevent conflicts with other threads and reads the temperature value. Within the critical section, the `LedMode` is updated and it checks if its between the temperature thresholds. Lastly, the thread also signals the GUI about the temperature update for real-time updates.

```
342 /**
343  * @brief Thread function for checking temperature.
344  * @param arg Argument for the thread function.
345  * @return Thread exit status.
346  */
347 void* MFS::tcheckTemperature(void* arg){
348     MFS& inst = MFS::getInstance();
349     static int currentLED=0;
350     for(;;)
351     {
352         pthread_mutex_lock(&inst.condMutex);
353         pthread_cond_wait(&inst.condReadTemp, &inst.condMutex);
354         pthread_mutex_unlock(&inst.condMutex);
355         /*-----------------------Changes the ledmode in the critical zone--------------------*/
356         pthread_mutex_lock(&inst.mtTemperature);
357         cout << "Temp Thread" << endl;
358         inst.performSensorRead();
359         pthread_mutex_unlock(&inst.mtTemperature);
360
361         /*-----------------------CHECK FOR TEMPERATURE THRESHOLDS----------------------*/
362
363         if((MFS::getInstance().getSensorValue()) >= TEMP_TRESHOLD)
364         {
365             currentLED=RED;
366             mfsWrapper::getInstance().emitWarningSignal(1);
367             sleep(3);
368             exit(1);
369         }
370         else if((MFS::getInstance().getSensorValue()) < TEMP_TRESHOLD)
371         {
372             currentLED=GREEN;
373             mfsWrapper::getInstance().emitWarningSignal(0);
374         }
375         /*-----------------------------------------------------------------------------------*/
376         // Update LED mode if changed
377         if (currentLED != inst.previousLED){
378             pthread_mutex_lock(&inst.mtLED);
379             inst.LedMode = currentLED;
380             inst.ledModeChanged = true;
381             pthread_cond_signal(&MFS::getInstance().condLED);
382             pthread_mutex_unlock(&inst.mtLED);
383         }
384
385         /*----------------------SIGNAL TO GUI--------------------*/
386         pthread_mutex_lock(&inst.condMutex);
387         inst.signalFromCheckTemp=true;
388         pthread_cond_signal(&inst.condToGUI);
389         pthread_mutex_unlock(&inst.condMutex);
390         /*--------- ------------------------------------------*/
391     }
392     pthread_exit(NULL);
393 }
```

Figure 4.39: Implementation MFS Class - tcheckTemperature thread function

This thread is responsible of managing the system's scanning operations, providing essential functioanlity for real-time scanning processes. Again, the thread function initiates after receiving the correct condition variable, which is active when a scanning operation is requested. The `mtScan` is then locked and the scanning process begins.

After completing the scanning opearation, the function signals other threads such as 4.43 and `tDatabaseManage`.

```
397 /**
398  * @brief Thread function for checking temperature.
399  * @param arg Argument for the thread function.
400  * @return Thread exit status.
401  */
402 void* MFS::tscan(void* arg){
403     MFS& inst = MFS::getInstance();
404     for(;;)
405     {
406         /*-----------------------Scan Thread locking----------------------*/
407         pthread_mutex_lock(&inst.condMutex);
408         pthread_cond_wait(&inst.condScan, &inst.condMutex);
409         pthread_mutex_unlock(&inst.condMutex);
410
411         /*-----------------------PERFORM THE READING-------------------------*/
412         pthread_mutex_lock(&inst.mtScan);
413         inst.performScannerRead();
414         pthread_mutex_unlock(&inst.mtScan);
415
416         /*------tToGUI and tDatabaseManage Unlock and LedMode selection--------*/
417         pthread_mutex_lock(&inst.condMutex);
418         pthread_cond_signal(&inst.condToGUI);
419         inst.signalFromScan=true;
420         pthread_cond_signal(&inst.condDatabaseManage);
421         pthread_mutex_unlock(&inst.condMutex);
422     }
423     pthread_exit(NULL);
424 }
425
```

Figure 4.40: Implementation MFS Class - tscan thread function

The `tLED` thread function controls the RGB LED based on diverse system conditions. Upon a change in the `LedMode`, the thread enters a critical section. Within this section, the time elapsed since the last color change is calculated to ensure a smooth transition between the respective RGB colors. If the minimum change interval of two seconds has transpired, the LED can change its color.

```
427 /**
428  * @brief Thread function for controlling the RGB LED based on system conditions.
429  *
430  * This thread monitors changes in the LED mode and updates the LED color accordingly.
431  * It waits for a signal indicating a change in LED mode, and enforces a minimum change
432  * interval to avoid rapid LED color changes.
433  *
434  * @param arg Unused parameter (required for thread function signature).
435  */
436 void* MFS::tLED(void* arg){
437     MFS& inst = MFS::getInstance();
438     constexpr int MIN_CHANGE_INTERVAL = 2;  // 2 seconds between each change
439     auto lastChangeTime = std::chrono::steady_clock::now(); // auto for detect data type
440     for(;;)
441     {
442         pthread_mutex_lock(&inst.condMutex);
443         while(inst.ledModeChanged==false){
444             pthread_cond_wait(&inst.condLED, &inst.condMutex);
445         }
446         pthread_mutex_unlock(&inst.condMutex);
447         inst.ledModeChanged=false;
448         auto currentTime = std::chrono::steady_clock::now(); //gets the current time
449         //calculates the elaspsed time in seconds
450         auto elapsedSeconds = std::chrono::duration_cast<std::chrono::seconds>(currentTime - lastChangeTime).count();
451         //checks if the time has elaspsed (2 seconds)
452         if (elapsedSeconds >= MIN_CHANGE_INTERVAL) {
453             cout << "tLED Thread" << endl;
454             pthread_mutex_lock(&inst.mtLED);
455             switch(inst.LedMode){
456             case(OFF):
457                 inst.led.writeColor(OFF); //red for critical temp
458                 inst.previousLED=OFF;
459                 break;
460             case(RED):
461                 inst.led.writeColor(RED); //red for critical temp
462                 inst.previousLED=RED;
463                 break;
464             case(GREEN):
465                 inst.led.writeColor(GREEN); //green for ideal tempearture
466                 inst.previousLED=GREEN;
467                 break;
468             case(BLUE):
469                 inst.led.writeColor(BLUE); //Blink Led for Scan Acknowledgement
470                 inst.previousLED=BLUE;
471                 break;
472             default:
473                 inst.led.writeColor(OFF); //led off
474                 inst.previousLED=OFF;
475                 break;
476             }
477         pthread_mutex_unlock(&inst.mtLED);
478         lastChangeTime = currentTime;  //updates the last change time
479         }
480     }
481     pthread_exit(NULL);
482 }
```

Figure 4.41: Implementation MFS Class - tLED thread function

This thread function is responsible for managing database operations within this system. The functions includes a switch statement to handle different types of database operations, such as adding, removing, searching, or recording sales. Each case executes the appropriate operation. Additionally, the function emits signals to notify the user about the outcome of the database operation.

```
/**
 * @brief Thread function for managing database operations.
 *
 * This thread waits for a signal to perform a database operation and executes the
 * corresponding action based on the operation code.
 *
 * @param arg Unused parameter (required for thread function signature).
 */
void* MFS::tdatabaseManage(void* arg){
    MFS& inst = getInstance();
    int DataBaseOp;
    int errreceive;
    unsigned int sender;
    string barcode;
    for(;;)
    {
        /*---------------------Thread locking---------------------*/
        pthread_mutex_lock(&inst.mtDatabaseManage);
        pthread_cond_wait(&inst.condDatabaseManage, &inst.mtDatabaseManage);
        pthread_mutex_unlock(&inst.mtDatabaseManage);
        /*----------------------------------------------------------*/
        cout << "Database Thread" << endl;
        barcode = inst.getCodeString();
        DataBaseOp = inst.DBoperation;
        /*-------------Database Operation Selection---------------*/
        switch(DataBaseOp)
        {
        case 1:
            if(barcode[0]=='A'){
                inst.DBM.addOperation(barcode, inst.ProductsVector,inst.db);   //Add to
                mfsWrapper::getInstance().emitTableUpdateSale();
                mfsWrapper::getInstance().emitAddSignal(1);  //sucess
            }
            else mfsWrapper::getInstance().emitAddSignal(0);  //failed
            break;
        case 2:
            if(barcode[0]=='A'){
                mfsWrapper::getInstance().emitRemoveSignal(1); //sucess
                inst.DBM.remOperation(barcode, inst.ProductsVector,inst.db);   //Remove from database
            }
            else  mfsWrapper::getInstance().emitRemoveSignal(0); //failed
            break;
        case 3:
            if(barcode[0]=='A'){
                //Search in database and set the foundProduct variable to the product searched/found on DB
                inst.setFoundProduct(inst.DBM.searchOperation(barcode, inst.db) );
                inst.DBSearchComplete = true;  // Set the flag to indicate search completion
                mfsWrapper::getInstance().emitTableUpdateSale();
            }
            else {
                inst.DBSearchComplete = false;
            }
            break;
        case 4:
            //Register a sale in Database
            inst.DBM.saleOperation(inst.ProductsVector);
            break;
        default:
            break;
        }
    }
    pthread_exit(NULL);
}
```

Figure 4.42: Implementation MFS Class - tDatabaseManage thread function

The `tToGUI` thread function is essential to ensure easy communication with the graphical user interface. This thread waits for signals related to stock checking, scanning, or temperature updates. Upon receiving these signals, the thread responds accordingly. If a signal indicates data from a scanning operation (`signalFromScan`), it triggers the emission of the corresponding signal to the GUI. Similarly, in the case of a temperature checking signal (`signalFromCheckTemp`), the thread emits an update signal containing the respective temperature level.

This thread enables the graphical representation of system states, providing users with a clear and dynamic view of ongoing processes.

```
/**
 * @brief Thread function for communication with the GUI.
 *
 * This thread waits for signals related to stock checking, scanning, or temperature
 * updates. It emits corresponding signals to the GUI based on the received signals.
 *
 * @param arg Unused parameter (required for thread function signature).
 */
void* MFS::ttoGUI(void* arg){
    MFS& inst = MFS::getInstance();
    for(;;){
        /*----------------------Thread locking from STOCKCHECK---------------------*/
        pthread_mutex_lock(&inst.condMutex);
        pthread_cond_wait(&inst.condToGUI, &inst.condMutex);
        pthread_mutex_unlock(&inst.condMutex);
        /*-------------------------------------------------------------------*/

        /*----------------------STOCK CHECK------------------------------*/
        if(inst.signalFromScan){
            inst.signalFromScan=false;
            mfsWrapper::getInstance().emitScanSignal(MFS::getInstance().getCodeString());
        }
        else if (inst.signalFromCheckTemp){
            inst.signalFromCheckTemp=false;
            //inst.emitUpdateSignal(MFS::getInstance().getSensorValue());
            mfsWrapper::getInstance().emitUpdateSignal(MFS::getInstance().getSensorValue());
        }
    }
    pthread_exit(NULL);
}
```

Figure 4.43: Implementation MFS Class - tToGUI thread function

4.2 Device Driver

One of the constraints of this project was to implement one device driver. The device driver implemented was to control the GPIO pins associated with the RGB LED pins. To simplify, the device driver implemented acts as a translator between the hardware device and the program or operating system that uses it. This device driver operates as "character device". A character device handles I/O operations as stream of bytes, making it suitable for devices that can be read from or written to. In this case, the GPIO pins, which control the RGB LED, are manipulated through the character device interface. In figure 4.47, it is possible to see the `alloc_chrdev_region()` and `cdev_ivaluablenit()` functions, which are characteristic of character device drivers.

```
#include <linux/cdev.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/init.h>
#include <linux/module.h>
#include <asm/io.h>
#include <linux/timer.h>
#include <linux/device.h>
#include <linux/err.h>
#include <linux/mm.h>
#include <linux/io.h>
//#include <mach/platform.h>

#include "utils.h"

#define DEVICE_NAME "rgbDevice"
#define CLASS_NAME "rgbledClass"

MODULE_LICENSE("GPL");

/* Device variables */
static struct class* ledDevice_class = NULL;
static struct device* ledDevice_device = NULL;
static dev_t ledDevice_majorminor;
static struct cdev c_dev;  // Character device structure

static struct class *s_pDeviceClass;
static struct device *s_pDeviceObject;
struct GpioRegisters *s_pGpioRegisters;

//led pins
static const int LedGpioPinR = 17;
static const int LedGpioPinG = 27;
static const int LedGpioPinB = 22;

```

Figure 4.44: Implementation of Character Device Driver 1

In the figure 4.45 it is presented a snippet of the Write callback operation for the character device, controlling the RGB LED based on the data written by the user-space program to the character device file associated with the module.

In figure 4.46 it is possible to see the implementations of the Read, Close, Open Callbacks, within the File Operations.

```c
// Write callback for the character device
ssize_t led_device_write(struct file *pfile, const char __user *pbuff, size_t len, loff_t *off) {
    struct GpioRegisters *pdev;

    pr_alert("%s: called (%u)\n",__FUNCTION__,len);

    if(unlikely(pfile->private_data == NULL))
        return -EFAULT;

    pdev = (struct GpioRegisters *)pfile->private_data;

    // Parse the input and control LED accordingly
    if (pbuff[0]=='0'){         // OFF
        SetGPIOOutputValue(pdev, LedGpioPinR, 0);
        SetGPIOOutputValue(pdev, LedGpioPinG, 0);
        SetGPIOOutputValue(pdev, LedGpioPinB, 0);
        printk("LED OFF\n");
    }
    else if(pbuff[0]=='1'){     // ON
        SetGPIOOutputValue(pdev, LedGpioPinR, 1);
        SetGPIOOutputValue(pdev, LedGpioPinG, 1);
        SetGPIOOutputValue(pdev, LedGpioPinB, 1);
        printk("LED ON!\n");
    }
    else if(pbuff[0]=='2'){     // RED
        SetGPIOOutputValue(pdev, LedGpioPinR, 1);
        SetGPIOOutputValue(pdev, LedGpioPinG, 0);
        SetGPIOOutputValue(pdev, LedGpioPinB, 0);
    }
    else if(pbuff[0]=='3'){     // GREEN
        SetGPIOOutputValue(pdev, LedGpioPinG, 1);
        SetGPIOOutputValue(pdev, LedGpioPinB, 0);
        SetGPIOOutputValue(pdev, LedGpioPinR, 0);
    }
    else if(pbuff[0]=='4'){     // BLUE
        SetGPIOOutputValue(pdev, LedGpioPinB, 1);
        SetGPIOOutputValue(pdev, LedGpioPinR, 0);
        SetGPIOOutputValue(pdev, LedGpioPinG, 0);
    }
    return len;
}
```

Figure 4.45: Implementation of Character Device Driver 2

```c
// Read callback for the character device
ssize_t led_device_read(struct file *pfile, char __user *p_buff,size_t len, loff_t *poffset){
    pr_alert("%s: called (%u)\n",__FUNCTION__,len);
    return 0;
}

// Close callback for the character device
int led_device_close(struct inode *p_inode, struct file * pfile){
    pr_alert("%s: called\n",__FUNCTION__);
    pfile->private_data = NULL;
    return 0;
}

// Open callback for the character device
int led_device_open(struct inode* p_indode, struct file *p_file){
    pr_alert("%s: called\n",__FUNCTION__);
    p_file->private_data = (struct GpioRegisters *) s_pGpioRegisters;
    return 0;
}

// File operations for the character device
static struct file_operations ledDevice_fops = {
    .owner = THIS_MODULE,
    .write = led_device_write,
    .read = led_device_read,
    .release = led_device_close,
    .open = led_device_open,
};
```

Figure 4.46: Implementation of Character Device Driver 3

The figure 4.47 presents the initialization function for the module upon loading. This function is responsible to initialize the character device, create a device class, and set up the GPIO pins. It also logs important information, such as the allocated character device region and the mapped virtual address of the GPIO registers.

Lastly, in figure 4.48 it is represented the cleanup function of this module. The function initiates the operation by logging a message, indicating the execution of the cleanup process. Following this, it proceeds to reset the GPIO pin functions, setting all the GPIO pins to input mode. With the use of the *iounmap()*, the address space is returned to the kernel, effectively preventing memory leaks. The *cdev_del(&c_dev)* removes the character device from the system.

```c
// Initialization function for the module
static int __init ledModule_init(void) {
    int ret;
    struct device *dev_ret;

    pr_alert("%s: called\n",__FUNCTION__);

    // Allocate character device region
    if ((ret = alloc_chrdev_region(&ledDevice_majorminor, 0, 1, DEVICE_NAME)) < 0) {
        return ret;
    }
    pr_info("Major=%d Minor=%d \n", MAJOR(ledDevice_majorminor), MINOR(ledDevice_majorminor));

    // Create device class
    if (IS_ERR(ledDevice_class = class_create(THIS_MODULE, CLASS_NAME))) {
        unregister_chrdev_region(ledDevice_majorminor, 1);
        return PTR_ERR(ledDevice_class);
    }

    // Create device
    if (IS_ERR(dev_ret = device_create(ledDevice_class, NULL, ledDevice_majorminor, NULL, DEVICE_NAME))) {
        class_destroy(ledDevice_class);
        unregister_chrdev_region(ledDevice_majorminor, 1);
        return PTR_ERR(dev_ret);
    }

    // Initialize character device
    cdev_init(&c_dev, &ledDevice_fops);
    c_dev.owner = THIS_MODULE;
    if ((ret = cdev_add(&c_dev, ledDevice_majorminor, 1)) < 0) {
        printk(KERN_NOTICE "Error %d adding device", ret);
        device_destroy(ledDevice_class, ledDevice_majorminor);
        class_destroy(ledDevice_class);
        unregister_chrdev_region(ledDevice_majorminor, 1);
        return ret;
    }

    // Map GPIO registers to virtual address
    s_pGpioRegisters = (struct GpioRegisters *)ioremap(GPIO_BASE, sizeof(struct GpioRegisters));

    pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pGpioRegisters);

    // Set GPIO pin functions
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinR, 1); // Output
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinG, 1); // Output
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinB, 1); // Output
    printk("Device Added !\n");

    return 0;
}
```

Figure 4.47: Implementation of Character Device Driver 4

The associated device and device class are then dismantled. The *device_destroy(ledDevice_class, ledDevice_majorminor)* function erases the specific device associated with the character device, while *class_destroy(ledDevice_class)* destroys the device class structure. Lastly, the resources associated with the character device are released by calling *unregister_chrdev_region(ledDevice_majorminor, 1)*, which triggers the removal of the allocated character device region.

```c
// Cleanup function for the module
static void __exit ledModule_exit(void) {
    pr_alert("%s: called\n",__FUNCTION__);

    // Reset GPIO pin functions
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinR, 0); // Input
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinG, 0); // Input
    SetGPIOFunction(s_pGpioRegisters, LedGpioPinB, 0); // Input

    // Unmap GPIO registers
    iounmap(s_pGpioRegisters);

    // Remove character device
    cdev_del(&c_dev);

    // Destroy device and class
    device_destroy(ledDevice_class, ledDevice_majorminor);
    class_destroy(ledDevice_class);

    // Unregister character device region
    unregister_chrdev_region(ledDevice_majorminor, 1);
}

module_init(ledModule_init);
module_exit(ledModule_exit);
```

Figure 4.48: Implementation of Character Device Driver 5

4.3 Buildroot

In order to generate the correct and efficient embedded Linux system, this was accomplished using Buildroot software. Below, some important packages for the correct functionality will be provided. It's worth noting that not all packages will be demonstrated, as it would be too extensive.

4.3.1 Packages

Qt5

Integrating the Qt5 package in Buildroot is essential for projects employing the Qt framework. Qt facilitates the development of cross-platform applications with user-friendly graphical interfaces.

```
--- Qt5
[ ]   qt53d
-*-   qt5base
()      Custom configuration options
()      Config file
[*]     Compile and install examples (with code)
[ ]     concurrent module
[ ]     MySQL Plugin
[ ]     PostgreSQL Plugin
        SQLite 3 support (No sqlite support)  --->
-*-     gui module
-*-       widgets module
-*-       OpenGL support
            OpenGL API (OpenGL ES 2.0+)  --->
-*-         opengl module
[*]       linuxfb support
          *** directfb backend available if directfb is enabled ***
          *** X.org XCB backend available if X.org is enabled ***
[*]       eglfs support
(linuxfb) Default graphical platform
[*]       fontconfig support
[*]       harfbuzz support
[*]       GIF support
[*]       JPEG support
[*]       PNG support
[*]     syslog support
[*]     DBus module
[*]     Enable ICU support
[*]     Enable Tslib support
[*]   qt5charts
[*]   qt5coap
      *** qt5connectivity needs neard and/or bluez5_utils ***
-*-   qt5declarative
-*-     quick module
[ ]   qt5doc (includes quick demos)
[ ]   qt5enginio
[*]   qt5graphicaleffects
[*]   qt5imageformats
[ ]   qt5knx
[ ]   qt5location
[ ]   qt5lottie
↓(+)
```

Figure 4.49: Qt5 Buildroot Package

JSON

Handling JSON data requires the use of libraries essential for parsing, manipulating and generating JSON-formatted information withing embedded Linux systems. By adding these packages to the image, it is then possible to manage JSON data, which is crucial in the context of our project.

```
[ ] benejson
[*] cJSON
[ ]   cJSON utils
-*- expat
[ ] jansson
[ ] jose
[ ] jsmn
[ ] json-c
[ ] json-for-modern-cpp
[ ] json-glib
[*] jsoncpp
[ ] libbson
[ ] libfastjson
[*] libjson
[ ] roxml
[ ] libucl
[ ] libxml2
[ ] libxml++
[ ] libxmlrpc
[ ] libxslt
-*- libyaml
[ ] Mini-XML
[ ] pugixml
[ ] rapidjson
[ ] rapidxml
[ ] raptor
[ ] serd
[ ] sord
[ ] tinyxml
[ ] tinyxml2
[ ] valijson
[ ] xerces-c++
[ ] xml-security-c
[ ] yajl
[ ] yaml-cpp
```

Figure 4.50: JSON Libraries

libcurl

`libcurl` is a powerful and versatile library designed to perform network-related operations, making it a valuable tool in scenarios where data needs to be transmitted securely and efficiently over the internet. In our system, the use of `libcurl` becomes essential to record sales into a website due to its capabilities in handling HTTP requests and facilitating communication with web servers.

sqlite

Adding the `sqlite` library to the Buildroot image was also essential to ensure efficient stock management in our project. `sqlite` embedded nature simplifies our architecture by eliminating the need for external database servers.

```
[*] libcurl
[*]   curl binary
[*]   verbose strings
[*]   proxy support
[*]   cookies support
[*]   enable extra protocols and features
```

Figure 4.51: libcurl Library

```
[*] sqlite
[ ]   Additional query optimizations (stat4)
[ ]   Enable convenient access to meta-data about tables and queries
[ ]   Enable version 3 of the full-text search engine
[*]   Enable the JSON extensions for SQLite
[ ]   Enable sqlite3_unlock_notify() interface
[ ]   Set the secure_delete pragma on by default
[ ]   Disable fsync
```

Figure 4.52: sqlite Library

As mentioned earlier, not all the packages are detailed here, and many others have been employed to ensure the availability of all necessary tools for this project.

4.4 Final structure

In the following figures its represented our final structure with the all the components connected.

Figure 4.53: Front_view

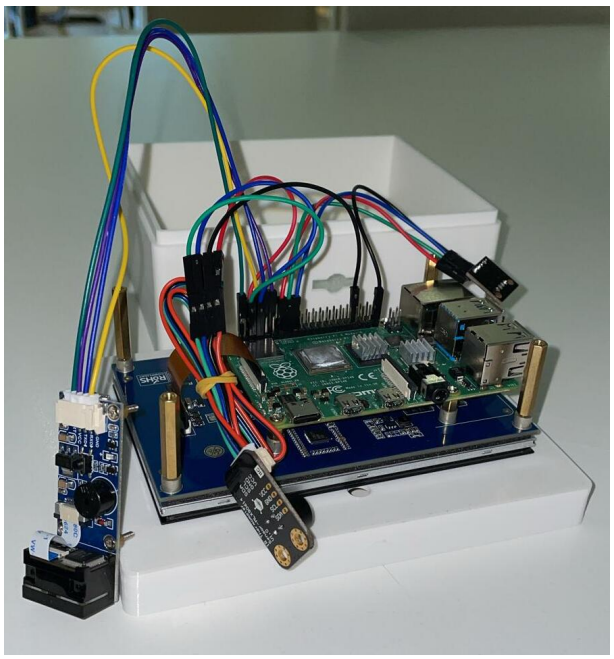

Figure 4.54: Inside_view

4.5 Conclusion

The Multi-Functional Scanner project has successfully achieved its objectives, having complied with the restrictions of having a customized Linux image generated by buildroot and deployed to a raspberry Pi, using PThreads as the API to our mulitthread system and also using C++ as our Oriented Object Programming language . Finally the scanner is now able to add new products with a JSON file as a catalog, search any existing product on the database using SQLite3, add or remove product quantities from it, register a purchases and also create charts with all the sales information. A RGB LED driver was also successfully implemented to represent the systems temperature, which is read from a temperature sensor that has been integrated.

4.6 Future Work

For future work, several areas can be explored to enhance the capabilities and robustness of the MFS. One potential focus could be the integration of advanced machine learning algorithms for product recognition, enabling the system to automatically identify and catalog new products without the need for a manual JSON file.

An embedded database system has space for improvement. The exploration of cloud integration, allows for remote access and data storage, fostering scalability and flexibility. Furthermore, considering the dynamic nature of markets, implementing real-time updates and notifications based on inventory changes and sales trends could provide valuable insights for efficient inventory management.

Continuous refinement of the Graphical User Interface(GUI), coupled with rigorous testing for scalability and reliability, will contribute to the long-term success and adaptability of the MultiFunctional Scanner system.

4.7 Acknowledgments

Last but no least, a special thanks to Professors, Dr.Nuno Cardoso and Dr.Adriano Tavares for providing the necessary resources, support and motivation throughout the project!

Bibliography

[1] Programação de Microcontroladores (MCS-51) by Adriano T., Carlos L., Carlos S., Jorge C., Paulo C., 2009.

[2] The Linux Kernel Documentation - https://docs.kernel.org/driver-api/driver-model/driver.html

[3] Michael Kerrisk - https://man7.org/index.html

[4] (C KISS) - Keep it simple and straightforward with C by Adriano T., Sandro P., 2017.

[5] PThreads Programming: A POSIX Standard for Better Multiprocessing, 1st Edition

[6] Zebra Mobile Computer

[7] Tera Digital Mobile Computer

[8] Raspberry Pi Model 4B Specifications

[9] Barcode Reader User Manual

[10] Touch Screen User Manual

[11] Battery Holder

[12] Temperature Sensor MLX90614 Datasheet

[13] Raspberry Pi Model 4B Pinout