Dedication

Acknowledgement

Abstract

Keywords:

Contents

List of Figures

List of Tables

Glossary of Acronyms

- **CAN** Controller Area Network
-

-

General Introduction

The latest events of the current decade have highlighted the challenges that manufacturers, suppliers, and end customers face during fluctuations in logistics and supply chain processes. Living in a VUCA world—Volatile, Uncertain, Complex, and Ambiguous—requires us to continuously adapt to changes and anticipate future events by preparing our developed environments and scaling our solutions. Simultaneously, it is crucial to maintain high standards that ensure productivity, enhance work safety, and optimize ergonomics.

In this context, the primary objective of intralogistics is to optimize, integrate, automate, and manage internal logistical flows of material and information within distribution centers, warehouses, or manufacturing plants. This subfield focuses on increasing operational efficiency by employing new technologies, such as autonomous robots.

Modernizing industrial environments through intralogistics offers significant potential for companies that adopt and adapt to it. However, convincing potential customers of the efficiency and impact of intralogistics robots presents challenges. These limitations include high training and implementation costs, changes to work routines, and the need for space and process adaptations.

A recent study from CBRE, the world's largest real estate services provider, revealed that European industrial and logistics investments increased by 16% in Q1 of 2024 compared to Q1 of 2023. Despite this, many warehouses are old, repurposed buildings that are unorganized due to the nature of their daily tasks. These brownfield warehouses are expensive to maintain and digitalize but represent ideal grounds for developing and utilizing fully autonomous systems. Unlike AGVs, autonomous vehicles possess the intelligence and capability to plan and execute their plans efficiently. They are designed to adapt to uneven terrains and unorganized working environments given the revolutionary technologies that they hold.

In this context, STILL, a KION group company, has been developing smart intralogistics solutions since its establishment more than a 100 years ago, successfully integrating automation into logistics. STILL offers a wide variety of products that cater to industries ranging from food retail to automotive manufacturing and chemical sectors. Their solutions address various customer challenges, such as reaching high shelves, order picking, palletizing, fleet management, and providing consulting services. Trusted by leading German companies like Siemens, STILL's products and services are renowned for their reliability and efficiency.

The STILL Autonomous Robots department focuses on developing and enhancing smart vehicles. These autonomous robots, with minimal cost-effective input from the warehouse environment, can perceive their surroundings, estimating their positions, efficiently planning future tasks, controlling their movements to reach destinations, executing desired actions, and making corrections if necessary. This focus on smart, autonomous vehicles demonstrates STILL's commitment to pushing the boundaries of intralogistics and automation.

In light of this, this thesis aims to contribute to the process of palletizing by optimizing a local path planning approach applied in the warehouse's stations near the shelves or spots where pallets are located for picking or in free placing areas. The developed approach seeks to plan the near-field path optimally while simultaneously avoiding obstacles.

The objective is to create predictable, repeatable, and explainable vehicle behaviors, demonstrating the autonomous vehicle's ability to generate effective solutions tailored to each specific scenario. By focusing on optimal, pattern-based near-field path planning, this thesis addresses the challenge of navigating complex intralogistics environments, ensuring maximum efficiency and safety in operations. This approach not only enhances the vehicle's performance but also showcases the potential of autonomous technology in transforming modern intralogistics.

This work encloses 4 chapters:

- **Chapter 1** gives a deep insight about the host company's structure, activities and products. Then it dives into the project context and it motivations, the studied problematic, the fundamental aspects of the work, the thesis specifications and, the work methodology.
- **Chapter 2** delves into the state of the art of the work area, then goes through a review of the literature that served as a base of the thesis and gave an overview of the existing solutions. Finally, it presents milestones followed in the course of the thesis work.
- **Chapter 3** explains the development steps of the approach: it presents the mathematical aspect of splines and their implementation in robotic path planning, explains the geometric division of the stations into transition zones, discusses the studied path discrimination approaches, and finally it explores the optimization approaches for the local path planning problem.
- **Chapter 4** explicits the steps it takes to implement the developed approach in the RACK framework, test them in the RACK simulation system, then on the automated vehicle, run different test scenarios and states the obtained results.

Chapter 1
Project scope

Chapter 1

Project scope

Introduction

1 Host company: SmartLab Solutions GmbH

The following section presents SmartLab Solutions GmbH (SLS) as the host company as well as its products.

1.1 General information about SLS and its vision

SLS is a German start-up based in Dresden, Saxony, and was founded in 2021. It aims to revolutionize the way laboratories manage their processes by providing a full service for laboratory digitization through analyzing, planning, and implementation.

SLS plans and builds complete solutions for laboratory digitization and offers comprehensive advice through designing solutions to assist laboratories and enhance their efficiency. [2]

The company is composed of 10 employees during the period in which this internship took place. The organizational chart of the company is detailed in Figure 1.1. [4]

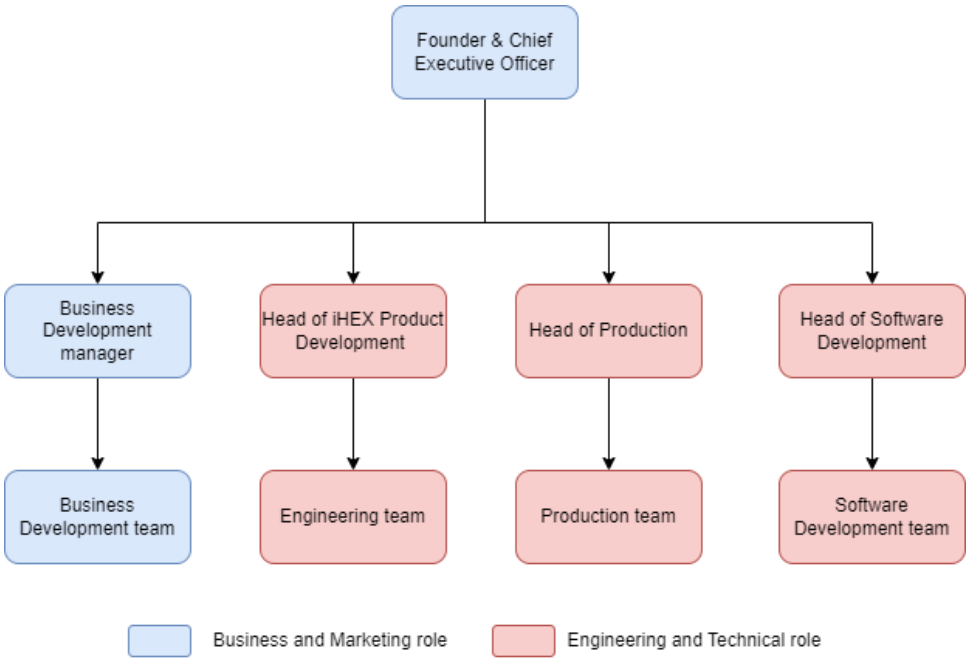


Figure 1.1: SmartLab Solutions GmbH hierarchy

1.2 iHEX system: SmartLab Solutions GmbH main product

Smartlab Solutions GmbH is developing digitization solutions for laboratories. The main activities are the engineering, development and production of customized solutions for customers, as well as developing a generic and modular solution which is based on smart and connected laboratory benches called **iHEX system**.

By attaching hexagon shaped elements - called **iHEX elements**, together, the needed lab devices held and controlled by the iHEX element, will be connected automatically together and everything is close-by, which enormously enhances efficiency since people do not have to make many footsteps to reach devices.

As shown in Figure 1.2 below, connected iHEX elements form **an island**. The island could be composed of:

- Static iHEX elements: These are the foundational components of the iHEX ecosystem, designed to serve as stationary nodes within a laboratory setup. They provide a stable and reliable base for various experiments and processes.
- Mobile iHEX elements: In contrast, mobile iHEX elements introduce a dynamic dimension to the system. They are specially designed components that can be moved and connected to static elements. These mobile units extend the versatility of the iHEX system by facilitating flexible experimentation setups and accommodating changes in laboratory configurations.

The iHEX system solution focuses mainly on ensuring:

- Seamless integration from sample to results: Thanks to connectivity, it assists the user with the data analytic. It also ensures flexibility of the workflow depending on the user needs.
- Workflow automation beyond individual process: Thanks to the fully integration of the process, the workflow is automated and easily controlled.

Financially speaking, the iHEX solution offers cost-efficiency since the customer can start with few modules, and then upgrade if necessary. [3]

Figure 1.2: SLS GmbH - iHEX product

The iHEX system is basically a geometry solution with a high degree of digitization. It allows the user to interface with the different laboratory components, give them orders and receive feedback:

- LED strip control: Each iHEX element is equipped by an LED strip to give the user information about each device and element states.
- Buzzer control: As an additional informative signal, the iHEX system offers the feature of auditorily signalling the user.
- Power switching: Each iHEX element is capable of controlling the electricity to power on or off the lab device.
- Reporting the connection of a new iHEX mobile element: As a feedback, the iHEX system offers the functionality of reporting such events.

The iHEX system assisted by a Berlin-based company product, offers a dash-boarding service (Dashboard - GUI (Graphical User Interface) in Figure 1.3 in the following page) in order to monitor all the data flow inside the laboratory.

Each iHEX element - modeled in a blue hexagon in Figure 1.3, is equipped with a Sub-Controller (SC). All the SCs communicate with a central unit called Main Controller (MC) (Green rectangle in Figure 1.3). The MC communicates with the server which is its intermediate with the Dashboard.

The GUI is able to interface with the hardware part via the server using an MQTT (Message Queuing Telemetry Transpot) interface; all the commands and reports are exchanged via the server. To do so, the GUI only interacts with its one counterpart; the MC, which is responsible for controlling one island.

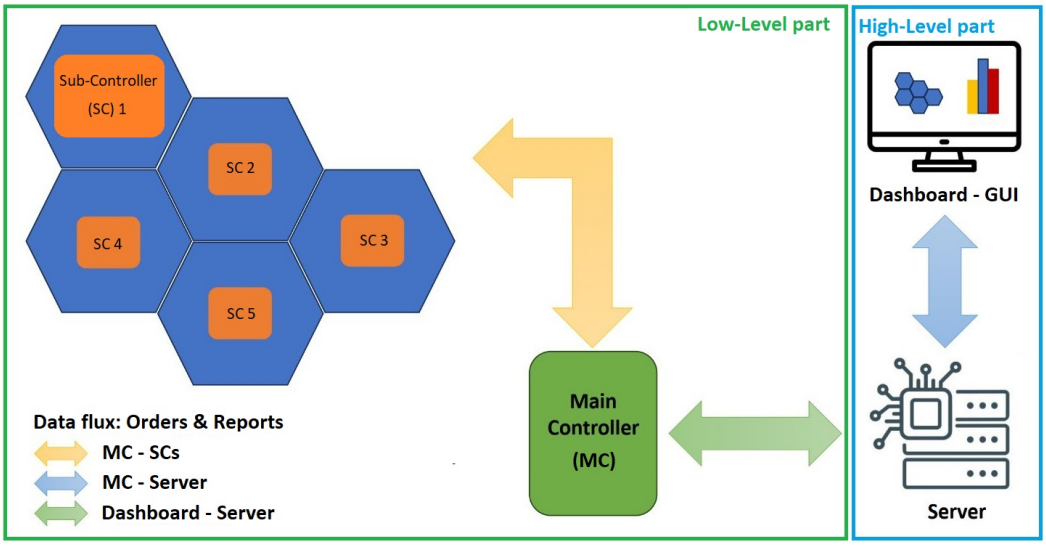


Figure 1.3: iHEX system overview

The high-level part of the project, shown in a green frame in Figure 1.3, is handled by the software development team, whereas the low-level part, shown in a blue frame in Figure 1.3, is handled by the engineering team to which I belong.

2 Project specifications

This project consists of developing, implementing, and testing a low-level communication protocol firmware for the iHEX system to ensure the control of the bench elements.

The following are the different specifications required by the company:

- **Choose the appropriate communication protocol and the different platforms:** This part consists of analyzing the available communication protocols in the market, detailing their characteristics, and choosing the most appropriate one based on the different project constraints.

After making the decision, it is important to choose the platforms to work on within this project. Choosing these platforms implies doing the technico-economical study and decide the platforms based on different criteria.

- **Ensure the communication between the elements**: This part consists of developing the needed firmware for the chosen platforms. These firmware provide the needed APIs (Application Programming Interfaces) for the application development.
- **Control Interface (CI) development**: Based on the functional specifications provided by the company, we have to elaborate the operational specifications to ensure the needed functionalities and develop the application software for all the chosen platforms. The functionalities are communication, LED control, powering control, devices control, feedback reception, (etc.).
- **Hardware development**: After the prototype validation (software and hardware parts), it is essential to proceed to finalize the hardware development. It consists of designing and developing the necessary electronic gadgets and the Printed Circuit Boards (PCBs) that assemble all the electronic modules to ensure the functionalities required by the company.

3 Project management tools

To ensure the successful delivery of embedded software projects, effective project management approaches are necessary.

This part consists of introducing the work methodology, the managerial strategy, and the project management tools used throughout the project.

3.1 Agile methodolgy

We adopted the Agile methodology for the whole project as it has shown its ability to improve efficiency.

This method consists of dividing the project into sprints, each sprint is also divided into cards (elementary tasks). Using this methodology, we aim to increase transparency, flexibility, and enhancement. [5]

Once all the sprints and cards are created and scheduled, the work starts respecting the following rules:

- **Bi-weekly meetings**: Evaluate the sprint and check for global advancement.
- **Frequent interactions between team members**: Good communication is essential to make this step successful.

All the Agile methodology phases are indicated in Figure 1.4.

Figure 1.4: Agile methodology

3.2 Project management software

In order to assist the management job, several software tools are available.

3.2.1 Microsoft tools: Microsoft Teams - Microsoft Outlook

Microsoft Teams is a powerful project management tool that provides a range of features to support the needs of Agile teams. With Teams, team members can communicate in real-time, share files and documents, and collaborate on tasks and projects.

As for Microsoft Outlook, it is a widely-used email client that provides a range of features to support communication and collaboration, including email, calendar, and task management.

3.2.2 YouTrack

YouTrack is a comprehensive project management tool that is designed specifically for Agile teams. The platform provides a range of features to support Agile methodologies:

- **Tickets (cards) tracking**: Tickets are used to assign micro tasks to the Agile team's members and track their signs of progress. Each ticket can have only one status at a time. Depending on its activity and workflow, each company chooses the appropriate set of statuses.
- **sprints tracking**: YouTrack offers the feature of recapitulating the different tickets of the sprint.
- **Backlog management**: The backlog contains all the awaiting tasks with their explanation.

- **Dashboarding**: YouTrack dashboards are a powerful feature that makes the team members focused on their goals. Customizable dashboards are so advantageous as they allow the members to select the information that matters most to them. This feature provides real-time visibility which enables team members to quickly identify issues and take action as needed.

YouTrack offers also the Kanban board which is a lean method used in agile project management. SLS GmbH Kanban board is shown in Figure 1.5 .

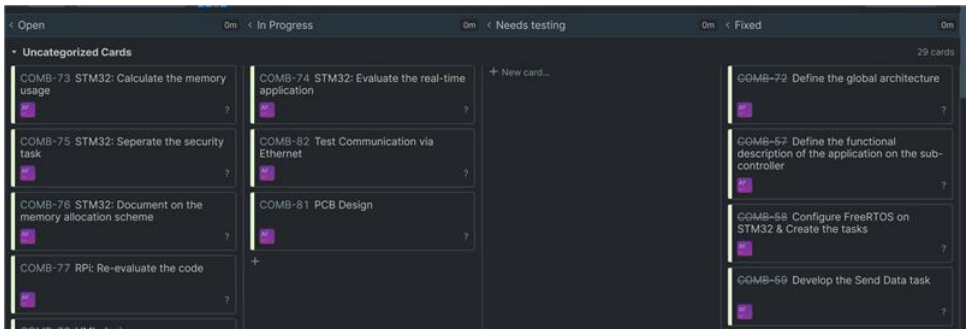


Figure 1.5: SLS GmbH Kanban board

3.2.3 GitHub

GitHub is a popular software development platform that is widely used by Agile teams to manage code repositories and collaborate on projects.

The platform provides a range of features to support project management, including issue tracking and pull requests.

- **Issue tracking**: With this feature, team members can easily report and track bugs and other issues, assign tasks to team members, and monitor progress.
- **Pull request**: This feature allows team members to propose changes to code and collaborate on code reviews before merging changes into the codebase.

Conclusion

Chapter 1 has presented the host company and its vision. Then, it detailed the general context and the objectives of this project, as well as the adopted working methodology. In the subsequent chapter, we will introduce the different milestones we went through to select the appropriate communication protocol and the MC and SC development boards. Then, we will describe the different steps to develop the communication software on both the MC and SC.

Chapter 2
Design and development of the CAN Bus software

Chapter 2

Design and development of the CAN Bus software

Introduction

This chapter focuses on the design and development of the CAN (Controller Area Network) interface for the iHEX system. It starts with discussing the specific constraints and requirements defined by the company's vision, which influence the selection of the communication protocol, the Main Controller (MC) platform, and the Sub-Controller (SC) microcontroller units (MCUs). The chapter then outlines the step-by-step procedure followed in developing the CAN communication software on both the MC and SC platforms. Finally, the test and validation process of the implemented software will be described.

1 Communication protocol selection

The selection of an appropriate communication protocol for the iHEX system is crucial to ensure efficient and reliable communication between the interconnected elements. The target in this part is the yellow data flux in Figure 2.1.

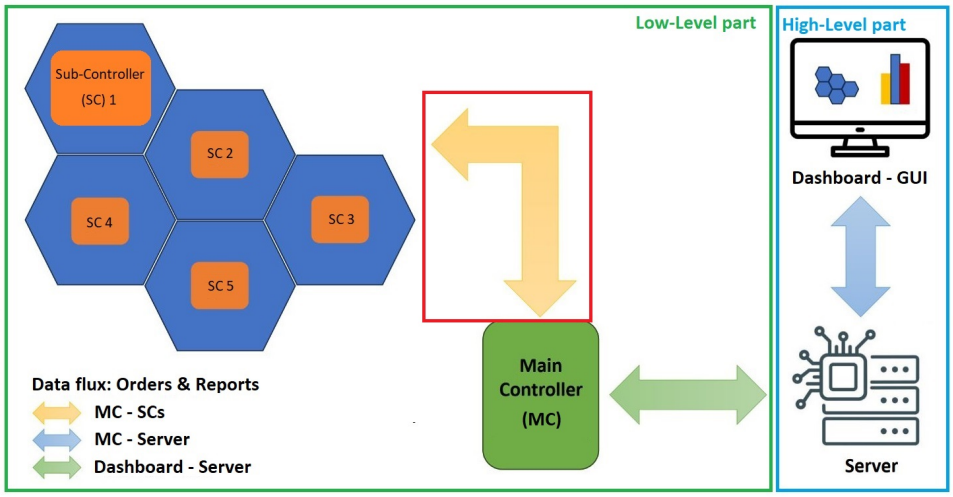


Figure 2.1: Communication protocol selection

1.1 The company requirements

During the protocol selection process, we took several factors and constraints into consideration. The following requirements were identified based on the company's vision and the specific needs of the iHEX system:

- **Minimization of cable length:** The company aims to reduce the cable length required for communication. This requirement is important to optimize the physical wiring and reduce complexity.
- **Scalability and extendability:** The communication protocol should support a scalable and extendable number of communicating nodes. This is essential to accommodate future expansion of the iHEX system, allowing for the addition of new elements without significant wiring modifications. Simplifying the wiring process for both static and new nodes enhances flexibility, reduces installation and configuration time, and enhance the user experience for the user.
- **Robustness in laboratory environment:** The communication protocol should be robust and able to withstand the challenging environment of laboratories. Laboratories are often characterized by high levels of electromagnetic interference and other disturbances. The chosen protocol should be resilient to such environmental factors.
- **Real-time capability:** The protocol should support real-time communication to ensure timely and synchronized interactions between the elements. Real-time capability is essential for accurate control and coordination within the iHEX system.

At this level, the SLS engineering team proposed to choose Ethernet as the communication protocol since it is already used within the iHEX system in some devices. As much more

communication protocols are available in the marked, we decided to compare the Ethernet and the Controller Area Network (CAN).

The following Table 2.1 summarizes, in short, the comparison between CAN and Ethernet.

Criteria	CAN	Ethernet
Minimization of cable length	One pair of twisted cables. Estimation: 10 m of one pair of twisted cable for an island of 10 elements.	n Ethernet cables; n is the number of iHEX elements per island. Estimation: 55 m of Ethernet cable (4 pairs of twisted cables) for an island of 10 elements.
Scalability and extendability	Supports $2^{29}-1$ communicating nodes	Depends on the router ports number
Robustness in severe environment	✓	✓
Real-time capability	✓	✓
Estimated cost for the prototype	40 €	110 €
Availability	✓	✓

Table 2.1: Comparison between CAN and Ethernet

Following the previous comparison, CAN protocol seems more convenient for the application in question.

1.2 Controller Area Network (CAN) specifications

This section will provide an overview of the CAN-Bus protocol, including its principle, topology, electronics, and data frame structure. This overview helps in identifying the CAN advantages that align with the company's vision and the constraints of the project.

1.2.1 CAN-Bus protocol overview

The CAN-Bus protocol is a widely-used communication standard that was originally developed for automotive applications. It has since found, extensive use in various industrial automation systems, making it an ideal choice for the iHEX ecosystem. The protocol is designed to facilitate robust and reliable communication between distributed nodes in a network, even in harsh and noisy environments such as laboratories. [6]

1.2.2 CAN-Bus topology

Figure 2.2 shows that CAN-Bus networks typically employ a bus topology, where all the nodes are connected to a shared communication medium, known as the bus. This topology simplifies the wiring process and enables easy scalability and extendability, which aligns with the requirements of the iHEX system.

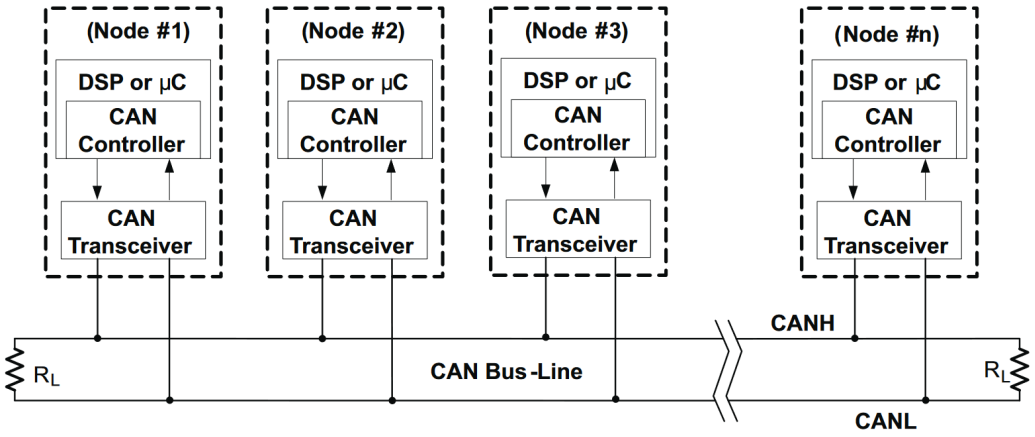


Figure 2.2: CAN-Bus topology [7]

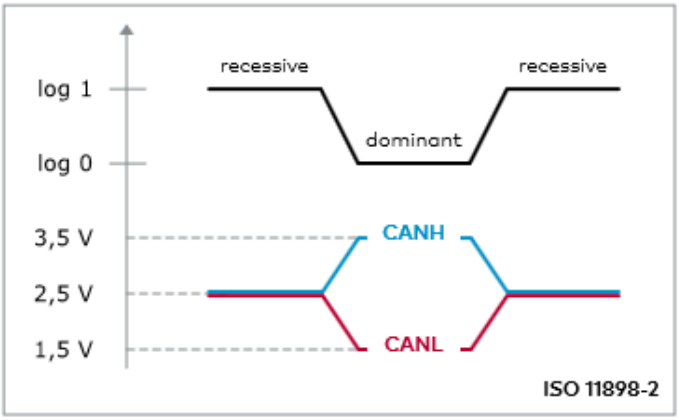
1.2.3 CAN-Bus electronics

The CAN-Bus protocol relies on a specific set of electronic components to ensure reliable communication. Each node in the network consists of an MCU, which interfaces with the CAN controller. The CAN controller, through interfacing with the CAN Transceiver, handles the transmission and reception of messages and manages the arbitration process, allowing multiple nodes to communicate without conflicts. The bus is terminated with resistors (Figure 2.2) at each end to prevent signal reflections and ensure signal integrity. [7]

The CAN-Bus protocol utilizes a differential signaling scheme, where the voltage levels on the bus represent the logical states of the transmitted data. This technique is behind the CAN robustness and accuracy even in severe environments. This signaling method involves transmitting data over twisted-pair cables, where the current flowing in each signal line is equal but opposite in direction. This method results in a field-canceling effect which is the key for the noise cancellation. [7]

As figured in Figure 2.3 in the next page, on the CAN-Bus, the voltage levels represent the logical states of the transmitted data. There are two logical levels to represent the data: dominant and recessive. A dominant level is represented by a logical 0, where the voltage on the CAN-High (CANH) line is higher than the voltage on the CAN-Low (CANL) line. On the other hand, a recessive level is represented by a logical 1, where the voltage on the CANH line is equal to the voltage on the CANL line. During communication, multiple nodes on the CAN-Bus may attempt to transmit simultaneously. This is resolved through arbitration based on the dominant and recessive levels. If two nodes transmit different logic levels, the node that

transmits a dominant level (logical 0) will carry on the transmission, and the one that transmits a recessive level (logical 1) aborts the transmission. [7]



© 2010-2017. Vector Informatik GmbH. All rights reserved. Any distribution or copying is subject to prior written approval by Vector. V2.0

Figure 2.3: CAN logical levels

1.2.4 CAN data frame

As mentioned in Figure 2.4, the data frame in CAN consists of several components that collectively enable the transmission and reception of data. [7]

- **Start-of-Frame (SoF) field**: It serves as a synchronization marker for all nodes on the bus. This field allows the receiving nodes to align themselves with the incoming data.
- **Arbitration field**: It contains the Identifier. The Identifier uniquely identifies the message and determines its priority on the bus. Nodes with higher priority Identifiers have the ability to overwrite lower priority messages during transmission, if the two nodes start the transmission at the same time. Two ID format are possible: standard (11-bit ID) or extended (29-bit ID). The arbitration field also contains the RTR bit which indates whether it is about a data or request CAN message. A data CAN message, contrary to the request one, contains data to be transmitted.
- **Control field**: It contains various control bits that govern the behavior of the data frame. These bits include the Data Length Code (DLC), which indicates the length of the data being transmitted (8 bytes is the maximal allowed length). They include also the IDE bit which indicates whether the ID is standard (logical 0) or extended (logical 1).
- **Data field**: It carries the actual payload or data being transmitted. The length of the Data field is determined by the DLC specified in the Control field.
- **CRC field**: It contains a cyclic redundancy check value. This value allows the receiving node to verify the integrity of the transmitted data and detect any potential errors.

- **Acknowledgement field**: It contains the acknowledgement bit (ACK) which indicates whether the transmitted message is received or not.
- **End-of-Frame (EoF) field**: It marks the end of the frame transmission.

Figure 2.4: CAN data frame

The data frame structure in CAN-Bus ensures efficient and reliable communication by providing synchronization, arbitration, control, data, error detection, acknowledgement, and termination mechanisms.

1.2.5 Other advantages of CAN

Two other advantages of CAN are notable. One is its built-in **acknowledgement** system, which guarantees reliable transmission of information. And the other is its built-in checking mechanism based on **Cyclic Redundancy Check** (CRC), which guarantees reliability. [7]

By selecting the CAN protocol, the iHEX system can benefit from its robustness, real-time capability, and potential for future scalability. This choice aligns with the company's requirements and ensures reliable and efficient communication within the system.

2 Selection of the Main Controller (MC) development board

Selecting the MC development board means selecting the hardware part of the MC. The target in this part is the green part in Figure 2.5.

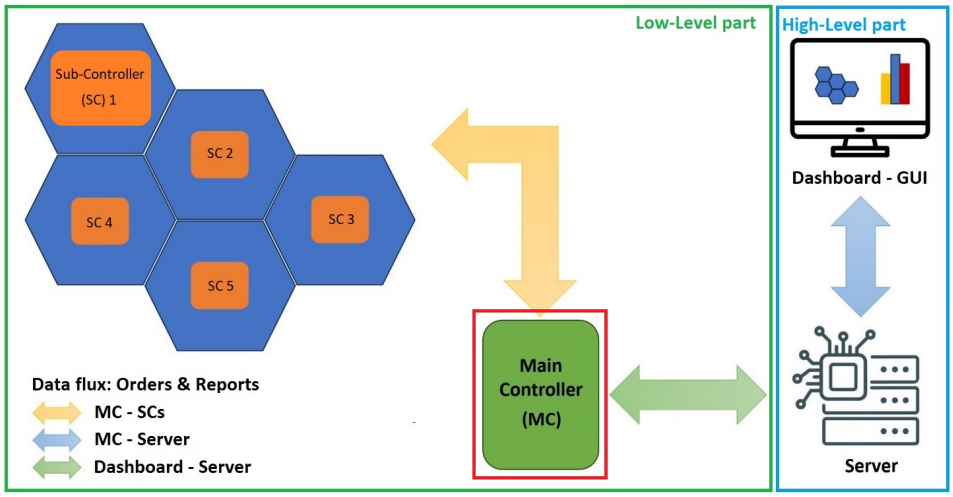


Figure 2.5: MC development board selection

Raspberry Pi is used for other purposes in the same project. It was chosen thanks to its suitability for meeting the project's requirements. The Raspberry Pi offers several advantages and capabilities that align with the needs of the project.

- It provides robust support for multi-threading, allowing for efficient parallel execution of tasks. This is crucial for handling concurrent processes and optimizing the performance of the MC.
- It does not have native support for CAN, but with the use of external modules and appropriate software libraries, CAN communication can be effectively implemented on the Raspberry Pi platform.
- It offers connectivity options to connect to an MQTT server via TCP, either through Ethernet or Wi-Fi. This enables the MC to publish and subscribe to MQTT topics, facilitating real-time data exchange and integration with other systems or devices.
- Another advantage of using the Raspberry Pi as the MC is its capability to write log files for debugging purposes. By logging relevant information, errors, and events, the MC can assist in troubleshooting and identifying issues during the development and testing phases.

The Raspberry Pi is characterized with more technical characteristics [8]:

- 8GB of Random Access Memory (RAM).
- Up to 256GB of SD-Card memory (Flash memory).
- 1.8GHz Cortex-A72 CPU (Central Processing Unit).

The Raspberry Pi serves as a suitable platform for the MC in this project. Its support for multi-threading, CAN communication, MQTT connectivity, and logging capabilities align well with the project's requirements, providing a reliable and efficient foundation for the development and operation of the MC in the iHEX system.

3 Selection of the Sub-Controller (SC) development board

Selecting the SC development board means selecting the hardware part of the SC. The target in this part is the orange part in Figure 2.6.

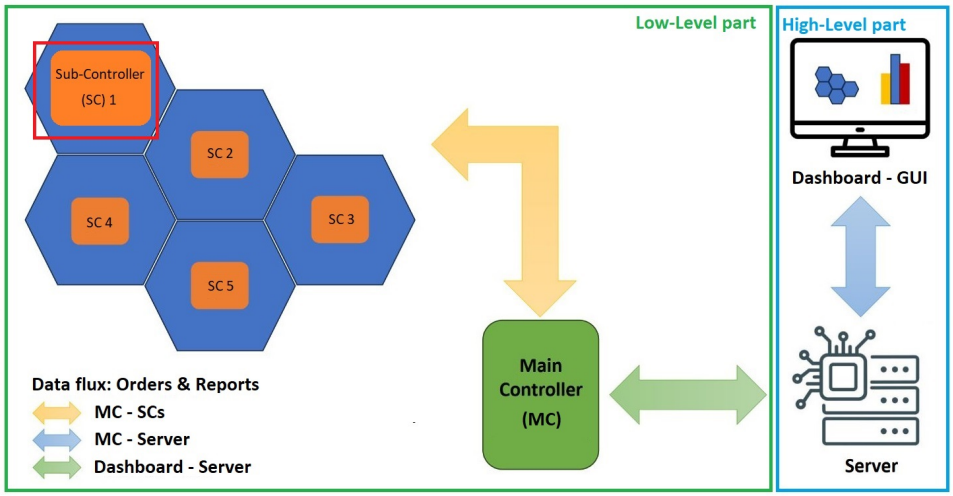


Figure 2.6: SC development board selection

To fulfill the requirements of the project, the STM32F303RETx MCU (MicroController Unit), installed on the NUCLEO-F303RE evaluation board [10], was chosen as the ideal option. The STM32F303RETx is a powerful MCU of STMicroelectronics (ST) that satisfies the project's needs.

3.1 Characteristics of the MCU: Microcontroller Unit

The STM32F303RETx microcontroller chosen for the project is characterized by the following performances [9]:

- CPU: The microcontroller is built with a highly-efficient ARM Cortex-M4 core.
- 72 MHz CPU Clock: The MCU operates at a maximum CPU clock frequency of 72 MHz. This clock speed determines the processing capability of the MCU and affects the execution speed of the program instructions.

- 128 KB RAM Memory: The MCU is equipped with 128 KB of Random Access Memory (RAM). Sufficient RAM allows for the efficient handling of variables, data structures, and temporary storage requirements.
- 512 KB Flash Memory: The MCU has 512 KB of Flash memory. The size of the Flash memory determines the amount of program code that can be stored on the MCU.
- Peripherals:
  - CAN controller: The STM32F303RETx integrates a CAN controller as an embed communcation peripheral, enabling seamless communication using the CAN protocol. As discussed before, this peripheral has to interface with a CAN transceiver to be able to be connected to the CAN bus.
  - General purpose Timers: The STM32F303RETx provides 6 general purpose timers that are capable of generating over than 20 PWM signals for controlling various components or systems.
- Sufficient IO pins: The microcontroller offers a substantial number of IO pins, ensuring compatibility with various peripherals and facilitating the connection of external devices, which is crucial during the prototyping phase.

The combination of a 72 MHz CPU clock, 128 KB RAM memory, and 512 KB Flash memory provides a balance between processing speed, data storage capacity, and program code size. These specifications of a general purpose MCU are suitable for a range of real-time applications, including CAN communication, PWM generation, and other tasks required by the project.

3.2 The supplier company

The decision to choose the STM32F303RETx microcontroller and NUCLEO-F303RE evaluation board was also influenced by the excellent support provided by ST. The company offers a comprehensive suite of development tools and resources that greatly facilitate the development process. Some of the key support features include:

- **ST software tools:** ST offers a series of software tools that support the embedded software engineer in developing embedded applications.
  - STM32CubeIDE: A powerful integrated development environment (IDE) based on Eclipse. [11]
  - STM32CubeMX: A graphical tool that enables easy configuration and initialization of the microcontroller peripherals and generates the necessary initialization code, easing the setup process. [12]
  - STM32CubeMonitor: A monitoring and visualization tool that allows real-time monitoring of the MCU's variables during runtime. [13]

  - STM32CubeProgrammer: A programming tool that provides various programming and debugging features, allowing easy firmware updates and debugging of the microcontroller.[14]

- **HAL library:** ST provides the Hardware Abstraction Layer (HAL) library, which offers a standardized and easy-to-use API for accessing the MCU's peripherals and functions.

Additionally, the NUCLEO boards offered by ST are specifically designed for proof of concepts and evaluation, making them well-suited for development and testing purposes.

Furthermore, ST provides extensive documentation for the STM32 MCUs and NUCLEO boards, including reference manuals, user manuals, and datasheets. These resources offer detailed information about the MCU's architecture, features, pinouts, and programming guidelines, aiding developers in utilizing the MCU to its full potential.

The combination of a robust development ecosystem, including software tools, support libraries, and documentation resources, enhances the ease of development, accelerates the learning curve, and ensures reliable support for the chosen STM32 MCU and NUCLEO evaluation board.

4 Development of the CAN communication software

The project specifications make us suggest the following architecture for the MC and SC.

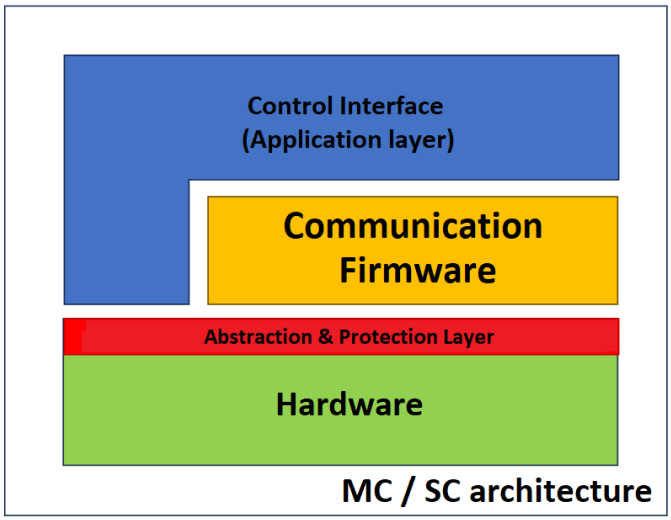


Figure 2.7: MC/SC architecture

Figure 2.7 in the previous page shows that both the MC and SC are mainly composed of 4 parts:

- Hardware: It presents the necessary hardware for the application. It is the lowest layer. In general, it is composed of the processing unit, the peripheral, memory, and other electronic components.
- Abstraction & protection layer (APL): It is a software layer. This layer is commonly used not only to facilitate the interface with the hardware, but also to protect against wrong usage or destructive usage.
- Communication firmware: The software that handles the communication with the iHEX parts. In interacts with the hardware layer via the APL and abstracts the communication process for the software layer on the top by providing an API.
- Control Interface (CI): The software that makes the whole system provide a set of functionalities to the user. It is the highest layer. It uses the API methods provided by the communication firmware layer to communicate and also interacts with the hardware layer to control the physical parts of the iHEX system.

Developing the CAN communication software means developing the communication firmware the yellow part in Figure 2.7.

The following part involves the low-level part of this project. It consists of developing the software abstraction layer that enable the user to interface properly with the CAN transceivers via Raspberry Pi and STM32.

4.1 MC: Development of the CAN communication software

The development of the CAN communication firmware for the Raspberry Pi involves configuring the Raspberry Pi to support CAN communication and implementing the necessary software components, and then, developing the abstraction layer based on the need of the user.

4.1.1 MC: Hardware setup:

This part presents the different steps to go through in order to make the Raspberry Pi support CAN communication. As this development board does not embed any CAN controller, it is required to use a CAN transceiver with another communication protocol interface. One common method is to use the "RS485 CAN HAT" board in Figure 2.8: An electronic circuit, based on MCP2515 chip, that, sticked on the top of the IO pins, makes the Raspberry Pi communicate via CAN-Bus protocol. [15]

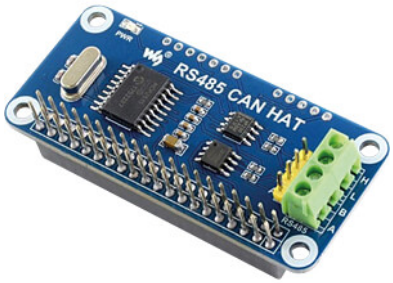


Figure 2.8: RS485 CAN HAT board

The MCP2515 is an integrated circuit (IC) produced by Microchip Technology. It is a standalone CAN controller that provides an SPI interface between a microcontroller or other host device and a CAN bus. [16]

Figure 2.9: MCP2515 as a black-box

The following section presents the necessary wiring to establish the physical connection between the Raspberry Pi and the MCP2515 module. It consists mainly of connecting the power supply and the SPI pins as it is mentioned in Table 2.2.

MCP2515 Pins	RPi Pins
VCC	5v (2)
GND	GND (39)
MOSI	GPIO10 (19)
MISO	GPIO9 (21)
SCK	GPIO11 (23)
INT	GPIO25 (32)
CS	GPIO8 (24)

Table 2.2: Wiring Configuration for RS485 CAN HAT to Raspberry Pi

4.1.2 MC: Software setup

This section summarizes the different steps to configure SPI interface on Raspberry Pi, install the can-utils library, and then setup the CAN interface.

Configuring the SPI interface returns to adding configuration lines to the "/boot/config.txt" file:

1. Edit the configuration file by running the command:

```
sudo nano /boot/config.txt
```

Listing 2.1: Open the config.txt file in read/write mode

2. Add the following lines to the file to enable SPI interface, configure the MCP2515 module for the can0 interface specifying the oscillator frequency and the interrupt pin, and overlay the necessary SPI driver for the Raspberry Pi.

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
```

Listing 2.2: Configuration lines of SPI configuration

3. Install the `can-utils` library by executing:

```
sudo apt-get install can-utils
```

Listing 2.3: Install can-utils library

`can-utils` is a developed library by "Volkswagen Group Electronic Research". It is used in this project to test the CAN configuration on the Raspberry Pi.

4. Reboot the Raspberry Pi using the command:

```
sudo reboot
```

Listing 2.4: Reboot the Raspberry Pi

5. Set up the CAN interface with the desired bitrate by running

```
sudo ip link set can0 up type can bitrate 500000
```

Listing 2.5: Configuration of the CAN interface

6. Check the result of the configuration.

```
sudo ifconfig
```

Listing 2.6: Check all the available network interface

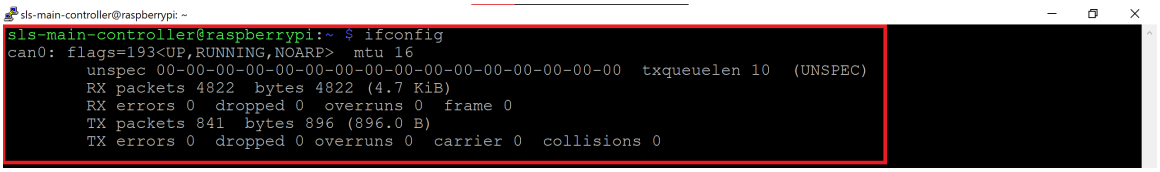


Figure 2.10: MC: Result of successful CAN configuration

Figure 2.10 shows that the MC is successfully configured to interface with the CAN transceiver and `can0` is set as a network interface. At this level, the Raspberry Pi is ready to send and receive CAN messages using the commands `cansend` and `candump`:

- **cansend**: A terminal command that is part of `can-utils` library. It allows the user to send CAN messages on the bus.

```
sls-main-controller@raspberrypi:~ $ cansend can0 123#2580
```

Figure 2.11: MC: Send CAN message on the bus

As figured in Figure 2.11, the sent message has an identifier equal to `0x123` and a payload equal to `0x2580`.

- `cansdump`: A terminal command that is part of `can-utils` library. It allows the user to listen to CAN messages on the bus.

```
sls-main-controller@raspberrypi:~ $ candump can0
  can0       123   [2]  25 80
```

Figure 2.12: MC: Listen to CAN messages on the bus

As figured in Figure 2.12, a new message is detected with the same identifier and payload of the sent message.

The software and hardware setup is successfully established. The project is ready to move on to developing the MC communication firmware.

4.1.3 MC: Design and development of the abstraction layer

The abstraction layer refers to a software component or module that provides a simplified interface and hides the underlying complexity of the system. It allows higher-level components to interact with lower-level components without needing to understand the intricacies of the implementation.

Figure 2.13 below presents the MC communication software layers architecture. The MCP2515 IC is programmed through writing in its registers via SPI commands. The SPI commands are exchanged with the IC in the Low-level firmware. On the top, a software abstraction layer is developed to provide the user with the needed API methods to interface correctly with the CAN transceiver.

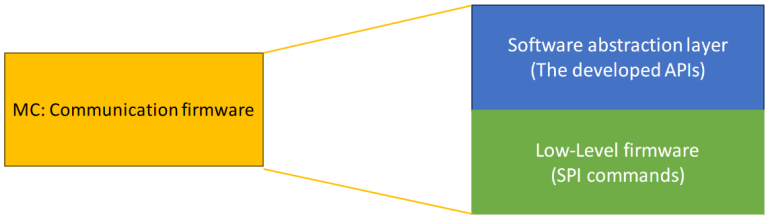


Figure 2.13: MC: Communication firmware architecture

Developing the software abstraction layer starts with defining the API methods that will be exposed for higher-level components to use.

The abstraction layer is mainly composed of two classes:

- **CAN_Header** class: It is implemented to make the MC able to send and receive CAN messages.
- **CAN_FilterManager** class: It is implemented to make the CAN transceiver filter the received CAN messages.

The UML design in Figure 2.14 represents the two classes, their attributes and methods, as well as the relationship between them.

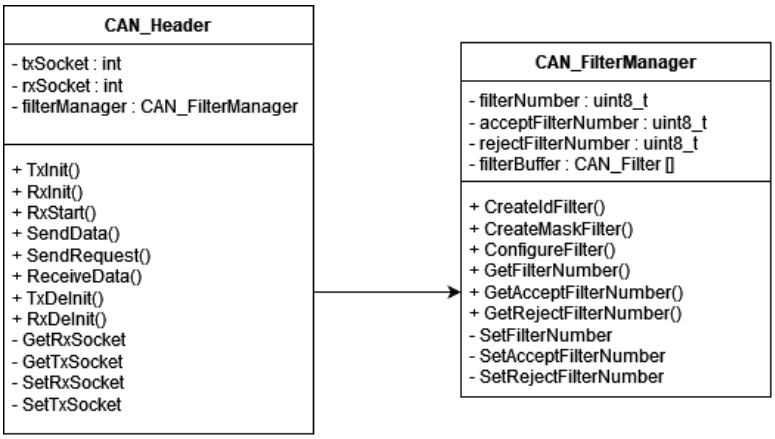


Figure 2.14: MC: CAN software UML design

In this context, the following are the headers of the `CAN_Header` class (API):

- `TxInit`: This method allows the user to initialize and properly configure the transmission module. It creates the socket responsible for handling the transmission process. It takes no parameter.
- `RxInit`: This method allows the user to initialize and properly configure the reception module. It creates the socket responsible for handling the reception process. It takes no parameter.

- `RxStart`: After initializing the reception module, this method allows the user to start the reception process. It takes no parameter.
- `SendData`: After initializing the transmission module, this method allows the user to send CAN data messages. It takes 3 parameters:
  - **id**: The id of the sender node. It is a 32-bit unsigned integer.
  - **dlc**: The length of the data to be transmitted. As the maximal value of dlc is 8 (bytes). It is an 8-bit unsigned integer.
  - **txFrame**: The address of the array that contains the data to be transmitted. It is a pointer on 8-bit unsigned integer.

  This method uses the `CAN_iSendFrame` function which is a upgraded version of `cansend` of the library `can-utils`.
- `SendRequest`: This method allows the user to send CAN request messages. It takes 1 parameter:
  - **id**: The id of the sender node. It is a 32-bit unsigned integer.

  This method also uses the `CAN_iSendFrame` function.
- `ReceiveData`: This method allows the user to check if there are any new received CAN messages. It takes 1 parameter:
  - **rxFrame**: If a new message is received, the message fields are stored in the rxFrame parameter. It is a pointer on `canfd_frame` (cf. Listing **??**).

  This method uses the `CAN_iReceiveFrame` function which is a upgraded version of `candump` of the library `can-utils`.
- `RxDeInit`: This method allows the user to safely and correctly disable the reception module.
- `TxDeInit`: This method allows the user to safely and correctly disable the transmission module.

All the previous methods are developed to abstract and facilitate the transmission and reception process.

The reception process requires the configuration of reception filter. In the following we describe the methods headers of the `CAN_Filter` class (API):

- `CreateIdFilter`: This method allows the user to create ID filter that makes the CAN transceiver whether accept or reject a message with the ID in question. It takes 3 parameters:

  - **id**: The id of the sender node. It is a 32-bit unsigned integer.
  - **rtr**: It specifies whether it is about a data message, a remote one, or it does not matter. It is a `CAN_Rtr_ten` (cf. Listing **??**).
  - **type**: It specifies whether the filter is an `accept` filter or a `reject` one. It is a `CAN_FilterType_ten` (cf. Listing **??**).

- `CreateMaskFilter`: This method allows the user to create mask filter that makes the CAN transceiver whether accept or reject a message that succeeds the filtering process accroding to the mask filtering rules. It takes 4 parameters:
  - **id**: The id of the sender node. It is a 32-bit unsigned integer.
  - **mask**: The mask of the sender node. It is a 32-bit unsigned integer.
  - **rtr**: It specifies whether it is about a data message, a remote one, or it does not matter. It is a `CAN_Rtr_ten` (cf. Listing **??**).
  - **type**: It specifies whether the filter is an `accept` filter or a `reject` one. It is a `CAN_FilterType_ten` (cf. Listing **??**).

- `ConfigureFilters`: This method allows the user, after creating all the filters, to configure the CAN transceiver. It takes 1 parameter:
  - **can**: The `CAN_Header` object that handles the transmission and reception processes.

All the previous methods of both CAN_Header and CAN_Filter return a `CAN_Status_ten` value, (cf. Listing **??**) to indicate whether the operation is successful or not.

4.1.4 MC: Development of the Low-Level (LL) firmware

In the next paragraphs the lower software layer (LL firmware) is detailed. As mentioned before, `CAN_stSendData` and `CAN_stSendRequest` methods use the `CAN_iSendFrame` function, which is which is an upgraded version of `cansend` of the library `can-utils`. [17]

`CAN_iSendFrame` takes 2 parameters:

- **frame**: contains all the fields to elaborate a CAN message. It is a pointer on `canfd_frame` (cf. Listing **??**).
- **s**: the transmission socket. It is an integer.

`CAN_stSendData` and `CAN_stSendRequest` elaborate the `frame` parameter to feed it to `CAN_iSendFrame`. They assign the length to `len` and the data frame to `data` and elaborates the `can_id`.

The `can_id` is a 32-bit unsigned integer as it is explained in Table 2.3.

EFF	RTR	ERR	ID
31	30	29	[28:0]

Table 2.3: can_id format

Bits 28:0 **ID**: the identifier of the node.

Bit 29 **ERR**: the error bit.
0 by default.
It is set to 1 whenever a transmission error occurs.

Bit 30 **RTR**: the RTR bit.
Set/ Reset by software during the configuration.
0: Data frame.
1: Remote frame.

Bit 31 **EFF**: the IDE bit.
Set/ Reset by software during the configuration.
0: Standard ID.
1: Extended ID.

Basically the `can_id` is equal to the identifier.

Depending on the frame type (data or remote frame) and the identifier type (standard or extended), the **forcing and masking rules** are applied to set or reset the corresponding bits in `can_id`.

Also, `CAN_stReceiveData` method uses the `CAN_iReceiveFrame`, which is a patched version of `candump` of the library `can-utils`. `CAN_iReceiveFrame` takes 2 parameters:

- **frame**: presents the object where to store all the received CAN message fields. It is a pointer on `canfd_frame` (cf. Listing **??**).
- **s**: the reception socket. It is an integer.

Filtering process is done by hardware. The MCP2515 is equipped by embed configurable filters that filter the received messages on the CAN bus. If the message succeed the filtering process, then the reception is reported to the Raspberry Pi. Otherwise, it is discared.

In this project, a filter is basically a `CAN_filter` object which has `filter` and `type` as attributes. `filter` is `can_filter` (cf. Listing **??**) and `type` is `CAN_FilterType_ten` (cf. Listing **??**).

Creating filters using `CreateIdFilter` or `CreateMaskFilter` returns to elaborating the two parameters: `can_id` and `can_mask`, the elements of `filter`.

The masking rules that the CAN transceiver applies are the following:

- *The `can_id` bits are associated with `mask_id` bits specifying which bits of the identifier are marked as "must match" (set bit in the `can_mask`) or as "don't care" (reset bit in the `can_mask`).*
- *A "must match" bit has to match the corresponding bit in the `can_id`.*
- *A "don't care" bit does not matter whether it matches the corresponding bit in `can_id` or not.*
- *An identifier succeeds the filtering process when all its "must match" bits match their corresponding bits.*
- *An identifier fails the filtering process when at least one of its "must match" bits does not match its corresponding bit.*

Rule 2.1: CAN masking rules

As `can_id` and `can_mask` have exactly the same formats as in Table 2.3, then the filtering process can also be applied for the **EFF** and **RTR** bits.

Same as the identifier creation, the **forcing and masking rules** are applied to set or reset the corresponding bits in `can_id` and `can_mask`.

`CreateIdFilter` allows the user to create a filter that only accepts or rejects a message with a specific identifier.

`CreateMaskFilter` allows the user to customize the filter to accept or reject a set of identifiers. By applying the masking rules in Rule 2.1, the user specifies which set of identifiers to accept or reject.

At this level, an example should be very useful to further explain the concept.

Lets assume that a **mask** filter is configured with the following parameters:

MASK: `0x20000111 = 0b 0010 0000 0000 0000 0000 0001 0001 0001`
ID: `0xA0000431 = 0b 1010 0000 0000 0000 0000 0100 0011 0001`

In this example, based on the `mask`, the bits 0,4,8 and 29 are `"must match"` bits. The others are `"don't care"` ones. In combination with the `id`, the succeeded IDs have 1 in the bit 0, 0 in the bit 4, 0 in the bit 8, and 1 in the bit 29.

Let's be two messages `message1` with `id1` and `message2` with `id2`.

`id1 = 0x21353011 = 0b 0010 0001 0011 0101 0011 0000 0001 0001`
`id2 = 0x21353110 = 0b 0010 0001 0011 0101 0011 0001 0001 0000`

`message1` succeeds the filtering process because all the `"must match"` bit match the bits in **ID**, whearas `message2` fails it because at least one of the `"must match"` bits does not match the bits in **ID** (bits 0 and 8 in this example).

Depending on the filter type, the successful messages - `message1` in the example, are whether accepted or rejected.

4.1.5 MC: Test and validation of the developed APIs:

In order to ensure the functionality and reliability of the communication APIs on the MC platform (Raspberry Pi), thorough testing and validation were conducted. The testing process involved the development of two essential functions—one dedicated to testing message trans-

mission and the other focused on testing message reception. These functions served as integral components of the testing suite, enabling a comprehensive evaluation of the MC's ability to communicate with other nodes through the CAN bus.

The code in Listing **??** presents the transmission testing function. The sent message is well received by the counterpart, and the following is the result on the MC terminal. It shows that the transmission is successful and the testing program terminates correctly. The methods used for to transmit CAN messages are validated.

```
sls-main-controller@raspberrypi:~ $ ./CAN_Test
Transmission OK
TX terminates correctly
```

Figure 2.15: MC: Successful transmission test

Additionally, the code in Listing **??** serves in testing the reception ability. The following is the result on the MC terminal. It shows that the test program received a new CAN message with the correct ID and data. The testing program terminates correctly. The methods used to receive CAN messages are validated.

```
sls-main-controller@raspberrypi:~ $ ./CAN_Test
New message is received
ID:      123
Len:     2
data[0]: 25
data[1]: 80
RX terminates correctly
```

Figure 2.16: MC: Successful reception test

By conducting these thorough tests on both message transmission and reception, the reliability and effectiveness of the MC's communication APIs were evaluated. Any identified issues or anomalies were addressed to ensure seamless communication between the MC and other elements on the same bus.

4.2 SC: Development of the CAN communication software

The development of the CAN interface API for the STM32 microcontroller involved a systematic approach to enable seamless communication over the CAN bus network. The process encompassed various stages, including hardware setup, software configuration, identification and development of essential methods, and thorough testing.

4.2.1 SC: Hardware setup

This part presents the different needed steps to succeed the hardware configuration.

The STM32F303RE has an embed CAN controller. It a peripheral called bxCAN, and it support several features. To be able to connect to the CAN bus, transmit and receive CAN messages, the MCU needs to connect to a CAN transceiver that plays the role of converting the digital signals from the microcontroller into the appropriate voltage levels required for communication over the CAN bus, and vice versa.

In this project, the MCP2551 is used as the CAN transceiver. [18]

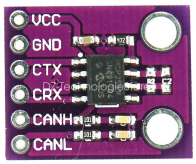


Figure 2.17: MCP2551 board

The MCP2551 is an IC produced by Microchip Technology.

Figure 2.18: MCP2551 as a black-box

The following section presents the necessary wiring to establish the physical connection between the NUCLEO-F303RE and the MCP2551 module.

MCP2551 Pins	NUCLEO Pins
VCC	5v
GND	GND
CTX	CAN_TX
CRX	CAN_RX

Table 2.4: Wiring Configuration for MCP2551 to NUCLEO-F303RE board

The TX and RX pins are internally mapped in the MCU. Depending on the used CAN peripheral, two pins are reserved for this purpose.

Only one bxCAN is available in the STM32F303RE. Based on the MCU datasheet - Pinout and pins description section, Table 14, only one combination of two pins is available. CAN_TX and CAN_RX are mapped respectevely to PA12 and PA11.

4.2.2 SC: Software setup

The software setup involves utilizing the STM32CubeIDE tools, specifically the graphical interface (.ioc file).

The CAN peripheral is configured through this interface, enabling the necessary settings for achieving the desired baudrate of 500,000 bits/s.

The baudrate is the inverse of the bit timing. The bit timing is obtained by configuring various registers and values.

The following values are configured to achieve the desired bit timing:

- Prescaler for Time Quantum
- Time Quanta in Bit Segment 1
- Time Quanta in Bit Segment 2
- Resynchronization Jump Width

The baudrate is obtained applying the following formula: [19]

$$Baudrate = \frac{f}{PSC * (SJW + Seg1 + Seg2)} (1)$$

where:

- Baudrate: The baudrate of the CAN bus communication. It is in bit per second.
- PSC: The value of the prescaler. It is used to divide the clock frequency of the CAN peripheral.
- f: The clock frequency of the CAN peripheral.
- SJW: The number of ReSynchronization Jump Width quanta. This value determines the number of time quanta used for synchronization during the bit synchronization phase. It allows the synchronization between transmitter and receiver in the presence of bit errors. It is equal to 1.
- Seg1: The number of the segment 1 quanta. It includes the PROP_SEG and PHASE_SEG1 of the CAN standard.
- Seg2: The number of the segment 2 quanta. It defines the location of the transmit point. It represents the PHASE_SEG2 of the CAN standard.

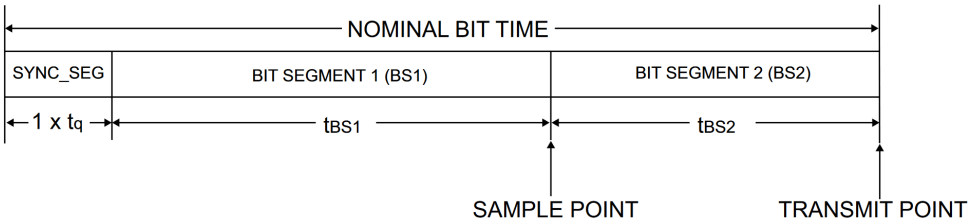


Figure 2.19: Bit timing [19]

According to the STM32F303xD/E datasheet block diagram, the bxCAN peripheral is connected to `APB1` bus. In this project, the CAN baudrate used is 500Kbits/s and the APB1 bus

frequency is set to its maximum value: 36MHz. Applying the forumula (1), the following must be a suitable combination.

PSC	SJW	Seg1	Seg2
9	1	6	1

Table 2.5: SC: CAN configuration parameters

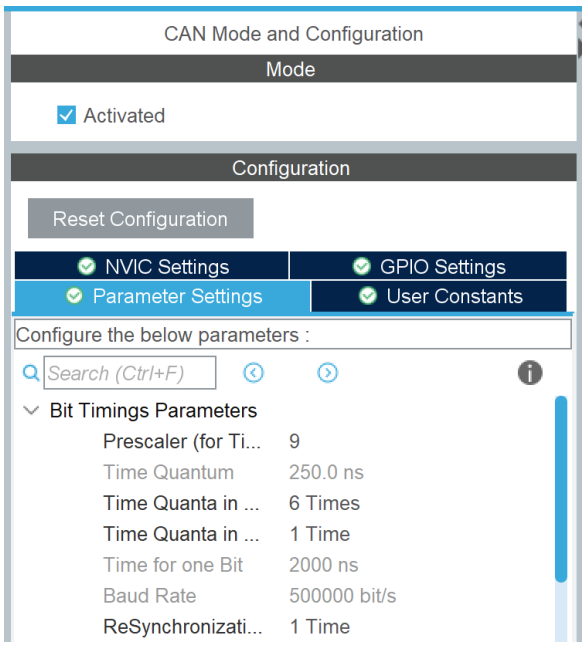


Figure 2.20: SC: CAN configuration

To test the setup, ST provides an embed mode of the bxCAN peripheral called `Loop-back` mode [20]. It consists of internally short-circuiting the TX and RX pin so the transmitted message is directly captured by the reception unit of the same peripheral. A test software was developed. Figure 2.21 below presents the result.

Expression	Type	Value
rxBuffer	uint8_t [8]	[8]
rxBuffer[0]	uint8_t	0x25
rxBuffer[1]	uint8_t	0x80
rxBuffer[2]	uint8_t	0x0
rxBuffer[3]	uint8_t	0x0
rxBuffer[4]	uint8_t	0x0
rxBuffer[5]	uint8_t	0x0
rxBuffer[6]	uint8_t	0x0
rxBuffer[7]	uint8_t	0x0
rxHeader	CAN_RxHeaderTypeDef	{...}
StdId	uint32_t	0x123
ExtId	uint32_t	0x0
IDE	uint32_t	0x0
RTR	uint32_t	0x0
DLC	uint32_t	0x2
Timestamp	uint32_t	0x0
FilterMatchIndex	uint32_t	0x0

Figure 2.21: SC: Successful CAN configuration test

The result shows that a new CAN message with the correct ID, data length and data is received. The hardware and software setup is validated. The system is ready to exchange CAN messages.

4.2.3 SC: Design and development of the abstraction layer

Figure 2.22 presents the SC communication software layers architecture. It is composed of two main layers. The lower is HAL: Hardware Abstraction Layer of STMicroelectronics. The higher is the software abstraction layer which is developed to provide the user with the needed API methods to interface correctly with the CAN transceiver; that being said, the software abstraction layer uses HAL.

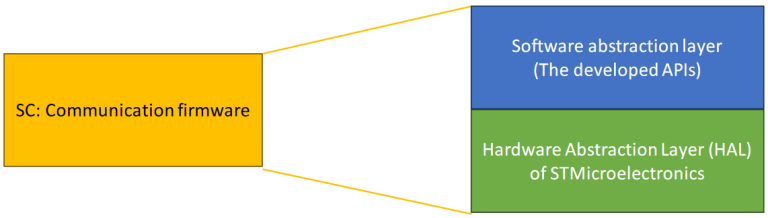


Figure 2.22: SC: Communication firmware architecture

Same as in the MC, the procedure starts with defining the methods that will be provided to higher-level components to use.

The UML design in Figure 2.23 presents the three modules developed to provide the needed API.

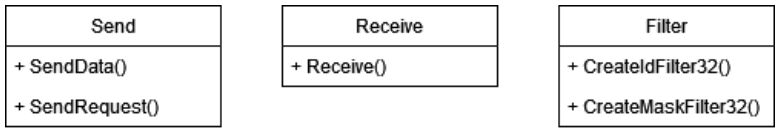


Figure 2.23: SC: CAN software UML design

In this context, five main methods are developed:

- `CAN_stSendData`: This method allows the user to send CAN data messages. (`SendData` in Figure 2.23) It takes 1 parameter:
  - **txMessage**: The CAN message to be sent. It is a pointer on `CAN_Message_tst` (cf. Listing **??**).
- `CAN_stSendRequest`: This method allows the user to send CAN request messages. (`SendRequest` in Figure 2.23). It takes 1 parameter:
  - **txMessage**: The CAN message to be sent. It is a pointer on `CAN_Message_tst` (cf. Listing **??**).
- `CAN_stReceive`: This method allows the user to check for and handle the received messages. (`Receive` in Figure 2.23). It takes 2 parameters:
  - **fifo**: The FIFO list to check. It is a 32-bit unsigned integer and could be whether `CAN_RX_FIFO_0` or `FIFO_RX_FIFO_1` (cf. Listing **??**).
  - **rxMessage**: The received CAN message. It is a pointer on `CAN_Message_tst` (cf. Listing **??**).
- `CAN_stCreateIdFilter32`: This method allows the user to create an ID filter that makes the bxCAN peripheral only accepts the messages with the ID in question. This method could configure one or two IDs the filter. (`CreateIdFilter32` in Figure 2.23). It takes 4 parameters:
  - **id1**: The first identifier. It is a 32-bit unsigned integer.
  - **id2**: The second identifier. It is a 32-bit unsigned integer.
  - **FilterFIFOAssignment**: The FIFO list to which is assigned the filter in question. - After succeeding the filtering process, the received message is stored in the FIFO list in order to be treated. It is a 32-bit unsigned integer and could be whether `CAN_RX_FIFO_0` or `FIFO_RX_FIFO_1` (cf. Listing **??**).
  - **FilterBank**: The bank to which belongs the filter in question. It is a 32-bit unsigned integer and could be one of the 28 values (cf. Listing **??**).

- `CAN_stCreateMaskFilter32`: This method allows the user to create an ID filter that makes the bxCAN peripheral only accepts the messages with ID succeeding the mask filtering process according to the same rule in Rule 2.1. (`CreateMaskFilter32` in Figure 2.23). This method takes 4 parameters:
  - **id**: The identifier. It is a 32-bit unsigned integer.
  - **mask**: The mask. It is a 32-bit unsigned integer.
  - **FilterFIFOAssignment**: The FIFO list to which is assigned the filter in question. - After succeeding the filtering process, the received message is stored in the FIFO list in order to be treated. It is a 32-bit unsigned integer and could be whether `CAN_RX_FIFO_0` or `FIFO_RX_FIFO_1` (cf. Listing **??**).
  - **FilterBank**: The bank to which belongs the filter in question. It is a 32-bit unsigned integer and could be one of the 28 values (cf. Listing **??**).

The filtering process, is not exactly the same as in the MC. Here, in the STM32 environment, there is no freedom to choose the type of the filter; all the filters are acceptance ones. That mean, each message succeeding the filtering process is accepted and assigned to one of the two hardware FIFO lists. The masking rules in are still valid.

Filter banks are embed pass-band filter to decompose the received signal.

All the methods return a `CAN_Status_ten` value (cf. Listing **??**) to indicate whether the operation is successful or not. They share the same return type as the MC methods.

The software layer developed for this project operates at a higher level than the HAL provided by STM32. It uses the methods provided by the HAL to inerface with the underlying hardware and extends its functionality by implementing additional methods specific to the project requirements.

4.2.4 SC: Test and validation of the developed API

To ensure the reliability of the developed APIs, a test process should be held in order to validate the efficiency of the developed methods of the SC.

The code in Listing **??** presents the transmission test function. The sent message is well received by the counterpart and the following is the result captured by the logic analyzer.

Figure 2.24: SC: Result of successful transmission methods test

The result shows that the logic analyzer captured a CAN message on the bus. The ID (`0x123`), DLC (2)and data (`0x2580`) fields are recognized and successfully verified.

Moreover, the code in Listing **??** tests the reception. The following is the result on the debugger view of the IDE.

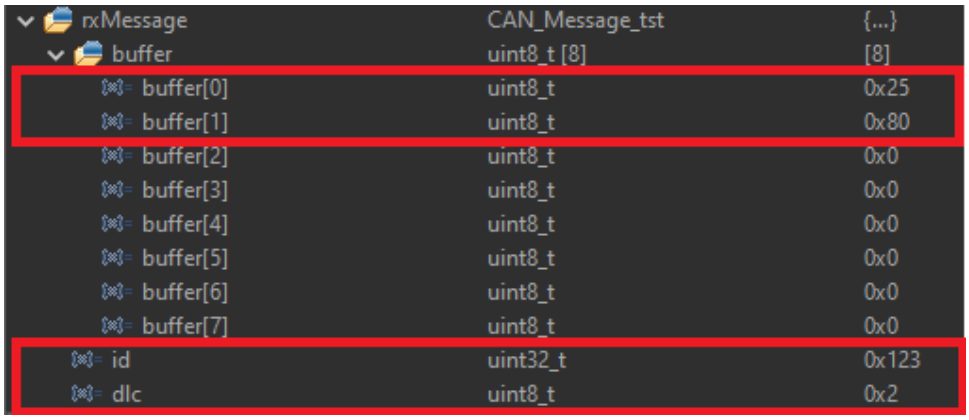


Figure 2.25: SC: Result of successful reception methods test

The result in Figure 2.25 shows that the MCU received a new CAN message on the bus. The ID (`0x123`), DLC (`2`)and data (`0x2580`) fields are recognized and successfully verified.

Conclusion

In chapter 2 we justified the selection of CAN-Bus as the communication protocol and the choice of Raspberry Pi and STM32 as respectively, the MC and SC development boards. We also detailed the design and development of the CAN-Bus firmware for the iHEX system, encompassing both the MC platform (Raspberry Pi) and the SC platform (STM32). The provided APIs for message transmission and reception were thoroughly tested and validated, ensuring reliable and efficient communication within the iHEX system. The output of this chapter is a necessary part for the development of the Control Interface (CI) in chapter 3.

Chapter 3
Design and development of the Control Interface

Chapter 3

Design and development of the Control Interface

Introduction

In this chapter, we delve into the heart of the iHEX system's functionality – the Control Interface (CI) software which forms the cornerstone of communication and control, enabling seamless interaction between the MC and the SCs. This chapter presents the design, development, and test of the CI on both the MC and SC and the integration of both of them.

1 Control Interface (CI) on the MC

1.1 Multithreaded architecture:

In the design of the CI on the MC, a multithreaded architecture was adopted to meet the dynamic and concurrent demands of communication, command processing, monitoring, and logging within the iHEX system. The use of multithreading was essential in ensuring a responsive and efficient operation of the control software.

1.1.1 Multithreading need

A fundamental requirement for the Control Interface is its ability to handle multiple tasks simultaneously. The MC must efficiently **manage communication with the server**, **manage communication with the SCs via CAN Bus**, **continuously monitor the health of the CAN network**, and **maintain a log of system activities**. This multifaceted demand necessitates a multithreaded approach to prevent bottlenecks and ensure timely execution.

1.1.2 The <pthread.h> library

To implement the multithreaded architecture, the <pthread.h> library (a default library of the Raspberry Pi OS), which provides the POSIX threads API, was leveraged. POSIX threads (pthreads) enable the creation and management of multiple threads within a single

process. This library offers the necessary tools for thread **creation**, **synchronization**, and **communication**. [21]

1.1.3 Design overview

Before diving into the specifics of each thread, the diagram in Figure 3.1 was created to visualize the architecture. This design served as a blueprint for structuring the threads, their interactions, and data flow.

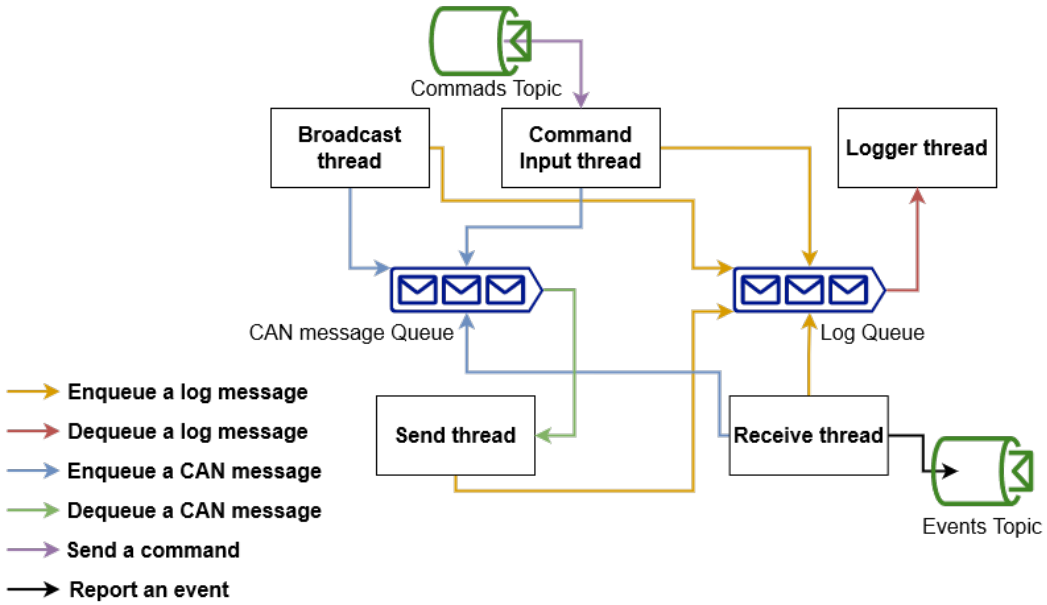


Figure 3.1: MC: Design of the CI software architecture

As deduced from the design in Figure 3.1, the MC CI interfaces with the server via the topics (Command Topic and Events Topic) on the one side. On the other side, it interfaces with the iHEX elements via CAN-Bus communication. It is also equipped with a logging service.

1.1.4 Threads, message queues and data flow

The process is basically composed of the following threads:

- Send Thread: Responsible for transmitting CAN messages. The Send Thread retrieves messages from the shared message queue "CAN message Queue". Messages in the queue are added by other threads, reflecting the ongoing communication demands of the system. This thread uses the transmission methods of the developed API in Chapter 2 (cf. 4.1 MC: Development of the CAN communication software).

- Receive Thread: This thread continuously checks for incoming CAN messages. Upon receipt, the Receive Thread decode the message content, extracting command data and relevant parameters. The extracted data is subsequently forwarded for processing. This thread uses the reception and filtering method of developed API in Chapter 2 (cf. 4.1 MC: Development of the CAN communication software).

- Command Input Thread: Focused on MQTT-based control commands from the server, the Command Input Thread listens for incoming messages. Upon receiving a command, it processes the command data and executes the corresponding routine. The incoming messages are in JSON format (cf. example in Listing **??**).

- Broadcast Thread: Responsible for network health verification, the Broadcast Thread periodically sends broadcast CAN messages. These messages serve as connectivity checks, ensuring the stability of network connections.

- Logger Thread: Tasked with recording system activities, the Logger Thread receives log messages from all the other threads in the "Log Queue". It then organizes and writes these messages into a log file, ensuring a comprehensive record of system events. The log messages are in a JSON format (cf. Listing **??**).

In the `main()` function, the set of threads is created. The threads are initialized and provided with their respective execution paths, ensuring parallel execution.

After their creation, threads are monitored and managed. They are cancelled and joined to ensure a controlled and synchronized termination of execution. Additionally, the `main()` function handles cleanup tasks, ensuring the safe and orderly conclusion of the program.

In this case, we can deduce that the CI process is composed of six threads: the five threads responsible for the CI's functionalities and the main one.

As mentioned in Figure 3.1, two queues are created to manage the communication between the threads:

- CAN message queue: Serving as a central communication hub, the CAN message Queue facilitates data exchange between the Send, Receive, Broadcast, and Command Input Threads. The Send Thread dequeues CAN messages, which are enqueued and processed by the Receive, Command Input, and Broadcast Threads.

- Log Queue: Send, Receive, Command Input, and Broadcast threads report results, errors and information messages to the Logger thread so the log file is updated. The Log messages are queued to the Log Queue by all the mentioned threads and are dequeued by the Logger thread.

1.2 MQTT integration

The integration of the Message Queuing Telemetry Transport (MQTT) protocol within the Control Interface on the Main Controller (MC) plays a pivotal role in establishing seamless and efficient communication throughout the iHEX system. This section highlights the multifaceted implementation of MQTT, encompassing the reception of MQTT messages by the Command Input thread, the publication of messages by Receive threads.

1.2.1 Importance and advantages of MQTT integration

MQTT's integration serves as a linchpin, enabling cohesive communication within the iHEX system. By providing a versatile and standardized protocol, MQTT fosters the exchange of control commands and system information, elevating the efficiency and responsiveness of the CI. [22]

MQTT's integration presents several advantages:

- Efficient and Lightweight: MQTT's minimal overhead ensures efficient data transmission, making it particularly well-suited for resource-constrained environments.
- Real-time Responsiveness: The publish-subscribe architecture facilitates real-time command transmission and prompt system responses.
- Asynchronous Operation: MQTT's decoupled publish-subscribe model allows for asynchronous communication, enhancing the overall efficiency of the system.
- Reliable Message Delivery: MQTT employs quality of service levels to ensure reliable message delivery, crucial for maintaining the integrity of system interactions.

The used server in this project has `"192.168.1.101"` as IP address and `"1883"` as a port.

Critical to the realization of MQTT capabilities within the Control Interface is the employment of the `paho.mqtt` library (cf. [23] and [24]). This versatile library, available in multiple programming languages including C, C++, and Python, simplifies the complexities of MQTT communication tasks. It presents a robust set of functionalities for message publication, subscription, and comprehensive MQTT management.

1.2.2 Subscribing to "Commands Topic" (CT)

The MQTT implementation extends beyond the reception of messages by the Command Input thread. The Command Input thread, pivotal to enabling command transmission from the server to the MC, subscribes to the CT. This subscription mechanism ensures that incoming control commands, encoded as MQTT messages and encapsulated in JSON format (cf. Listing **??**), are effectively received by the thread. Upon receipt, the Command Input thread processes the commands and channels them to the relevant task queues for execution.

In order to accomplish this, Commands Input thread subscribes to CT and defines a callback to be executed once a new message is published as figured in Listing 3.1.

```
// Callback: iHEX_SubscriberCallback object
iHEX_SubscriberCallback callback;
// Subscriber client creation
mqtt::async_client SubscriberClient(ADDRESS, SUBSCRIBER_CLIENT_ID);
// Subscriber connection options
mqtt::connect_options SubscriberConnOpts;
SubscriberClient.set_callback(callback); // assign the callback
// make the client send a keep-alive message every 20s
SubscriberConnOpts.set_keep_alive_interval(20);
// Clean other previous sessions
SubscriberConnOpts.set_clean_session(true);
// Connect and wait for connection
SubscriberClient.connect(SubscriberConnOpts)->wait();
// Subscribe and wait for subscription
SubscriberClient.subscribe(COMMANDS_TOPIC, QOS)->wait();
```

Listing 3.1: Setup MQTT subscriber: Commands Input thread

Where:

- `ADDRESS` is `"192.168.1.101:1883"`.

- `SUBSCRIBER_CLIENT_ID` is `"iHEX_MainControllerSubscriber"`.
- `COMMANDS_TOPIC` is `"CommandsTopic"`.
- `QOS` is the Quality Of Service and it is `1`.

1.2.3 Publishing to "Events Topic" (ET)

Receive thread, attuned to the content of incoming CAN messages, occasionally publish MQTT messages to the server via the ET. The messages are encapsulated in JSON format (cf. Listing **??**). In order to accomplish this, Receive thread publishes the messages as detailed in Listing 3.2.

```
// Publisher client creation
mqtt::async_client PublisherClient(ADDRESS, PUBLISHER_CLIENT_ID);
// Publisher connection options
mqtt::connect_options PublisherConnOpts;
// Connect and wait for connection
PublisherClient.connect(PublisherConnOpts)->wait();
// make the client send a keep-alive message every 20s
PublisherConnOpts.set_keep_alive_interval(20);
// Clean other previous sessions
PublisherConnOpts.set_clean_session(true);
// Prepare the message
pubMsg_ptr=mqtt::make_message(EVENTS_TOPIC, mqttPubMsg_s, QOS,
    false);
// Publish the message
PublisherClient.publish(pubMsg_ptr)->wait();
```

Listing 3.2: Publish MQTT messages

Where:

- `ADDRESS` is `"192.168.1.101:1883"`.
- `PUSBLISHER_CLIENT_ID` is `"iHEX_MainControllerPublisher"`.
- `EVENTS_TOPIC` is `"EventsTopic"`.
- `QOS` is the Quality Of Service and it is `1`.

1.3 JSON integration

The MQTT messages received by the Command Input thread and those published by the Receive thread, and also the Log messages trated by the Logging thread are encoded in JSON format. JSON (JavaScript Object Notation) is a lightweight data interchange format that offers a human-readable and standardized way to represent data [25].

The JSON structure enables efficient parsing and handling of message content, promoting clear communication between different components of the iHEX system. `json.hpp` is used as a C++ library. (cf. [26]).

2 Control Interface (CI) on the SCs

The CI on the SCs constitutes the operational heart of the iHEX system's low-level components. This section provides an in-depth exploration of the CI on the SCs, focusing on the real-time task execution, communication and synchronization strategies, and the direct interaction with hardware components.

2.1 FreeRTOS middelware

To facilitate the real-time execution and coordination of tasks within the iHEX system, the Control Interface (CI) on the SC was developed with the support of FreeRTOS. Serving as a free and reliable middleware, FreeRTOS offers a robust platform for managing concurrent tasks, ensuring efficient resource allocation, and maintaining deterministic behavior. [27]

2.2 Real-Time task execution

The CI on the SCs thrives within the real-time realm, propelled by its meticulous execution of specific tasks. These tasks are meticulously designed to ensure swift responses to control commands and real-time physical inputs.

The design in Figure 3.2 presents an overview of the SC Control Interface software architecture.

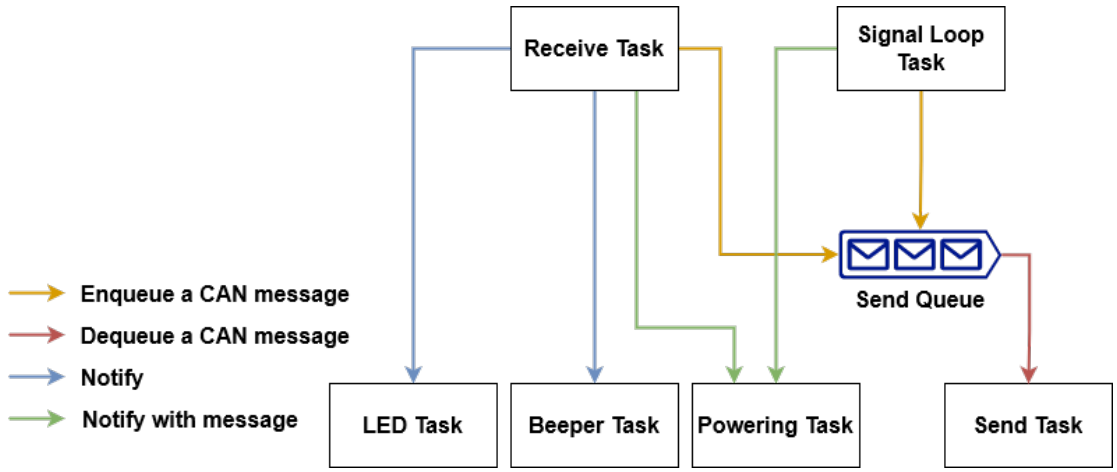


Figure 3.2: SC: Design of the CI software architecture

The following tasks collectively orchestrate the dynamic operation of the SCs:

- **Send task:** The Send Task undertakes the transmission of essential CAN messages, fostering communication with the MC and fellow SCs. These messages encapsulate crucial updates on status, acknowledgments, and responses to received commands, playing a pivotal role in the iHEX system's seamless operation. This task uses the transmission methods of the developed API in Chapter 2. (Section 4.2 SC: Development of the CAN communication software)

- **Receive task:** The Receive Task, devoted to the reception and interpretation of incoming CAN messages, decodes the intricacies of the received data. It identifies command instructions, extracts essential parameters, and passes on this information for subsequent actions, ensuring a cohesive response to commands issued by the MC. This task uses the reception and filtering methods of the developed APIs in Chapter 2 (Section 4.2 SC: Development of the CAN communication software)

- **Signal Loop task:** A constant monitor of an electric signal, the Signal Loop Task perpetually assesses the robustness of physical connections within the iHEX system. This task safeguards communication pathways and validates the integrity of connections, forming a foundational layer of the iHEX system's reliability.

- **LED task:** The LED Task, meticulously aligned with hardware components, facilitates the visualization of system states and operational conditions. Through precise control of LEDs, this task contributes to diagnostics and ongoing monitoring, providing tangible insights into system behavior.

- **Beeper task:** Auditory feedback is managed by the Beeper Task, which orchestrates the patterned beeps that convey system alerts and user notifications. By skillfully manipulating sound signals, this task adds an additional layer of interaction and communication within the iHEX environment.

- **Powering task:** The Electricity Task holds sway over the activation and deactivation of electrical components. By exercising control over power sources, this task ensures safe, controlled, and optimized electrical operations, enhancing the safety and reliability of the iHEX system.

As mentioned in Figure 3.2 above, one queue is created to manage the communication between task. Send queue contains the CAN messages enqueued by "Receive" and "Signal Loop" tasks. "Send Task" dequeues the CAN messages and send them on the CAN Bus.

Not only queues are used to ensure the communication and synchronization between tasks, notifications are also. Notifications (yellow arrows in Figure 3.2) allow a task to send data to another one ensuring the same efficiency and consuming less memory than queues.

2.3 Hardware interaction

Direct interaction with hardware components is another pivotal role undertaken by the Control Interface on the SCs. By governing the LEDs, beepers, electrical components, and the loop signal the interface ensures that control functionalities are realized seamlessly. This interaction streamlines the translation of high-level commands into tangible, actionable changes within the iHEX system's physical components. It also makes the SC able to get feedback on the physical connections within iHEX system.

2.3.1 LED management

The interface governs the RGB (Red-Green-Blue) LEDs integrated into the iHEX system, providing a visual representation of system states and conditions. The LED Task oversees the controlled illumination and switching of LEDs, offering real-time insights and diagnostics.

Figure 3.3 presents the control schema that allows the SC to switch on and off the LEDs and control the colour and the visual effect.

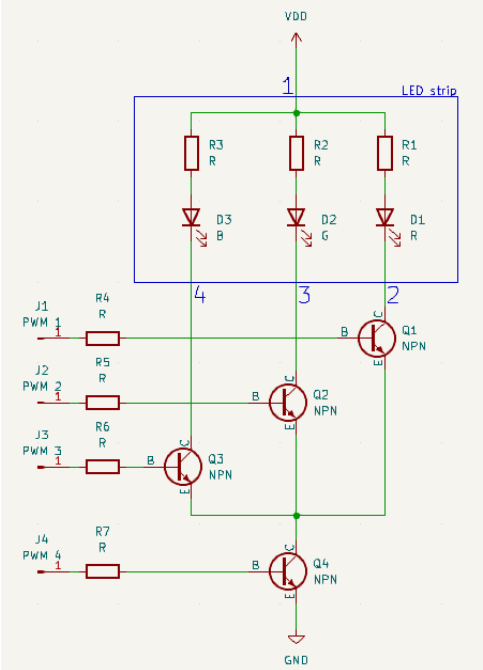


Figure 3.3: LED control schema

The LEDs are controlled with PWM signals (PWM1, PWM2, PWM3, and PWM4) on the MOSFETs Gates:

- `PWM1`, `PWM2`, and `PWM3` respectively control R, G, and B.
- `PWM4` controls the blinking effect required by the company.

PWM signals are gnerated by a timer embed in the SC (STM32 MCU).
For that purpose, the timer's channels are configured in their PWM generation mode.
As known, a PWM signal is characterized by its **frequency** and its **duty cycle**.
The frequency f is obtained by applying the formula (2) [28]:

$$f = \frac{f_{clk}}{(1 + PSC) * (1 + ARR)} \quad (2)$$

where:

- f_{clk} is the frequency of the bus to which the timer is connected.
- PSC is the Prescaker register.
- ARR is the AutoReload Register.

In mode 1 (cf. [28]), the duty cycle δ is obtained by applying the formula (3):

$$\delta = \frac{CCR}{ARR} \quad (3)$$

where:

- CCR is the Capture-Compare Register.

Rule 3.1: PWM generation

To generate `PWM1`, `PWM2` and `PWM3` signals, 3 channels of Timer1 are used in its PWM generation Mode 1 to control the LEDs applying the configuration (4):

$$f_{clk} = 72MHz, \quad PSC = 1, \quad and \quad ARR = 35999 \quad (4)$$

It results in $f = 1000Hz$ and the duty cycle is obtained by modifying CCR applying the formula (3).

To generate `PWM4` signal, Channel1 of Timer3 is used in its PWM generation Mode 1 applying the configuration (5):

$$f_{clk} = 72MHz \quad and \quad PSC = 35999 \quad (5)$$

In that case, f and the duty cycle are obtained by respectively modifying ARR and CCR applying the formula (2) and (3).

2.3.2 Buzzer control

Auditory feedback is a significant aspect of the iHEX system's communication. The Beeper Task manages beeping patterns, translating system alerts and notifications into audible signals that resonate with users.

Figure 3.4 presents the control schema that allows the SC switch on and off and blink the buzzer.

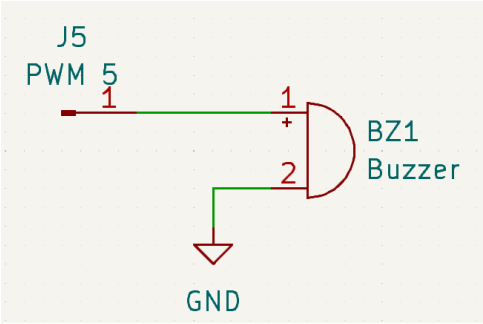


Figure 3.4: Buzzer control schema

In order to accomplish this, Channel2 of Timer15 is used in its PWM generation Mode 1 applying the configuration (5).

In that case, f and the duty cycle are obtained by respectively modifying ARR and CCR applying the formula (2) and (3).

2.3.3 Electricity control

The Electricity Task extends beyond the simple activation and deactivation of electrical components; it also encompasses the provision and prevention of electricity to and from mobile elements integrated within the system. Given the high power nature of electricity, this process is meticulously controlled through the utilization of relays as shown in Figure 3.5.

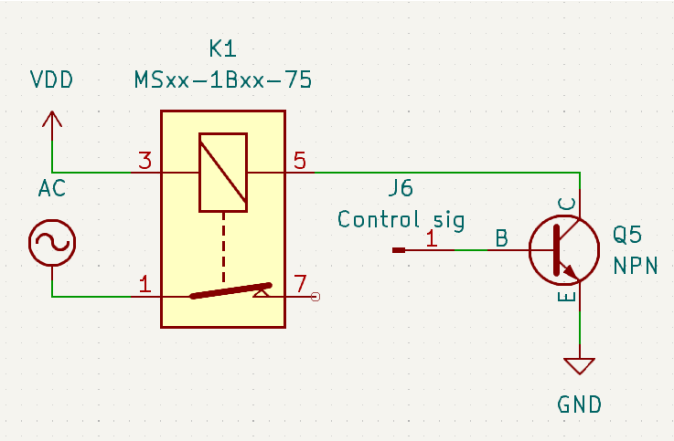


Figure 3.5: Electriciy control schema

If `Control sig` is `VCC`, then the coil of the relay is active. It applies an attractive force on the contact so the circuit is opened.

If `Control sig` is `GND`, the the coil of the relay is not active. The contact is released and the circuit is closed.

2.3.4 Signal Loop monitoring

A fundamental pillar of the Control Interface's hardware interaction is the continuous monitoring of the signal loop—a vital electric signal designed exclusively for the dynamic iHEX

mobile elements. The Signal Loop Task, a dedicated module within the Control Interface, takes on the responsibility of incessantly scrutinizing this signal's integrity.

The signal loop serves as an indispensable mechanism for detecting and responding to the connection and disconnection of iHEX mobile elements—a dynamic addition to the static installed system.

The Signal Loop Task steadfastly tracks the signal loop's presence and strength exclusively for mobile elements. When a mobile element establishes a connection, the task promptly identifies the robust signal, triggering the inclusion of the element within the iHEX ecosystem. Similarly, the task is primed to detect any weakening or absence of the signal, which signifies the disconnection of a mobile element.

Figure 3.6 represents the simple schema of the Signal Loop module.

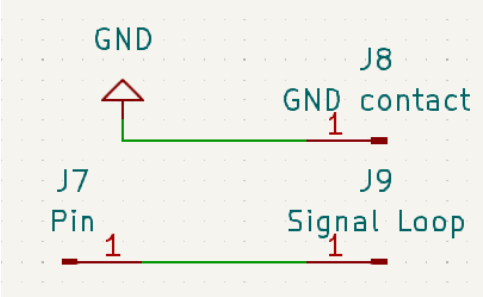


Figure 3.6: Signal Loop schema

The following paragraph presents a brief summary of the hardware setup:

- GPIO (General Purpose Input Outpu) Configuration: A dedicated GPIO pin is configured as a pull-up input. In this state, the GPIO pin is held at a high voltage level (logic '1') through a pull-up resistor connected internally to a positive voltage source (Vcc).
- Signal Loop Contact: A Signal Loop contact is established as part of the mobile element. This contact interacts with the dedicated GPIO pin.
- Ground signal: When an iHEX mobile element establishes a connection, the ground signal and the Signal Loop are short-circuited. This short-circuit effectively forces the ground signal to the GPIO pin.

When the mobile element is connected and the ground signal is short-circuited with the Signal Loop, the GPIO pin is effectively pulled down to a low voltage level (logic '0'). This change in voltage level is detected by the SC's Control Interface, which recognizes the connection of the mobile element.

Conversely, when the mobile element is disconnected, the ground signal and the Signal Loop are disconnected. As a result, the GPIO pin returns to its default high voltage level (logic '1'), indicating the absence of a connection.

3 Integration and interoperability

The successful operation of the iHEX system hinges on the seamless integration and interoperability of its diverse components. This section delves into the complex network of connections and interactions that allow the MC and SCs to harmoniously function as a unified whole.

3.1 Integration of MC and SCs

The integration of the MC and SCs serves as the foundation for the iHEX system's holistic operation. The MC, acting as the central coordinator, establishes communication links with each SC. Through the Control Interface, the MC sends and receives command messages, orchestrates control functionalities, and interprets system states. This integration ensures that the SCs respond promptly and accurately to commands issued by the MC, fostering a symbiotic relationship between control and execution.

3.2 Bidirectional communication channels

The iHEX system's integration is orchestrated through a comprehensive array of communication channels, each designed to facilitate effective information exchange and robust coordination. These channels not only enable the MC to control the SCs but also foster a continuous feedback loop that enhances the system's responsiveness.

3.2.1 Channel one: Command propagation

The first channel, as shown in Figure 3.7, involves the MC's command dissemination to the SCs. It commences with the reception of a command on the "Commands Topic" of the MQTT server. The command is encapsulated in an MQTT message in JSON format. The MC interprets this command, extracting its specifics and intent. Subsequently, the MC translates the command into a suitable CAN message, which is then sent onto the CAN bus to reach the SCs. This channel exemplifies the MC's role as a central coordinator, converting high-level instructions into actionable commands for the SCs.

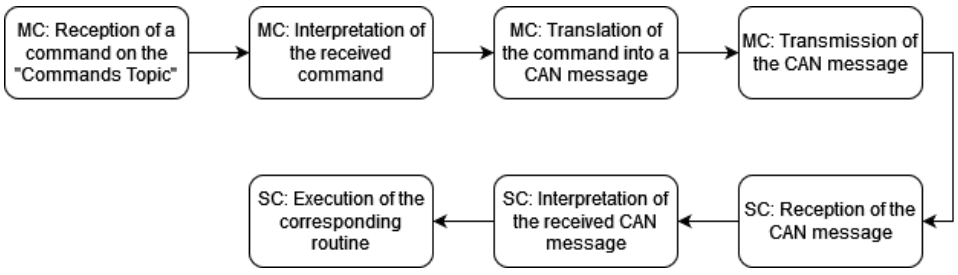


Figure 3.7: Channel one different steps

Example:
A command received through the MQTT server instructs the iHEX system to light on the LEDs in Red (cf. Listing **??**). The MC processes this command, converts it into a CAN message (cf. Listing **??**), and dispatches it to the SC responsible for the designated module's control.

3.2.2 Channel two: Feedback and reporting

The second channel, as shown in Figure 3.8, encompasses the SCs' transmission of feedback and information to the MC. An SC detects events, gathers information, or generates real-time feedback that holds relevance to the system's operational status. This information is encapsulated in CAN messages, which the SC sends onto the CAN bus. Upon reception, the MC interprets the CAN messages, extracting vital details. The MC then channels the extracted information into an MQTT message in JSON format. This message is subsequently published to the ET of the MQTT server, facilitating remote monitoring and comprehensive system awareness.

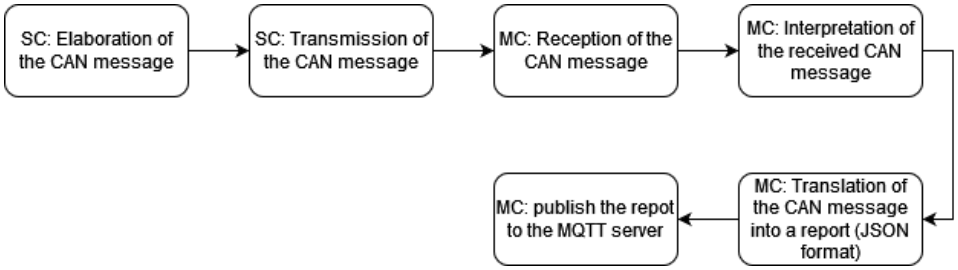


Figure 3.8: Channel two different steps

Example:
An SC sends a CAN message containing `"connection lost"` (cf. Listing **??**). The MC deciphers the CAN message and generates the corresponding MQTT message (cf. Listing **??**). This MQTT message is published to the "Events Topic," enabling remote users to stay informed about the laboratory's conditions.

3.3 Detecting and integrating new iHEX mobile elements

One of the pivotal scenarios highlighting the integration prowess of the iHEX system is the detection and integration of new iHEX mobile elements. This process, as showin in Figure 3.9 exemplifies the orchestration of control, communication, and synchronization across MC and SCs.

3.3.1 Signal Loop task: Detecting new connection

The process commences within the SCs, specifically through the vigilance of the Signal Loop Task. This task constantly monitors the signal loop, a specialized electric signal unique to iHEX mobile elements. Upon detecting a new connection—an indication of a mobile element's integration—the Signal Loop Task springs into action.

3.3.2 Reporting to the MC

As the Signal Loop Task identifies a new connection, it promptly formulates a CAN message signaling this event. This message is dispatched onto the CAN bus, reaching the MC with the crucial information about the new iHEX mobile element's inclusion.

3.3.3 MC response

The MC, now informed about the new mobile element, meticulously orchestrates the subsequent steps. It initiates the process by locking the `"newMemberMutex"`, ensuring the exclusive handling of new connection reports and preventing potential overlaps.

3.3.4 Confirmation and activation

Following mutex locking, the MC promptly responds to the SC. It sends a confirmation CAN message, informing the SC that the new connection has been duly acknowledged and accepted by the MC. The SC, armed with the MC's confirmation, proceeds to feed the mobile element with electricity—an imperative step for its activation.

3.3.5 Handshake and recognition

As the new mobile element becomes active, it responds with a hand-shake signal, a succinct yet significant gesture signifying its successful integration. This hand-shake serves as a recognition beacon, alerting the MC that the new element is fully operational and a seamless part of the iHEX ecosystem.

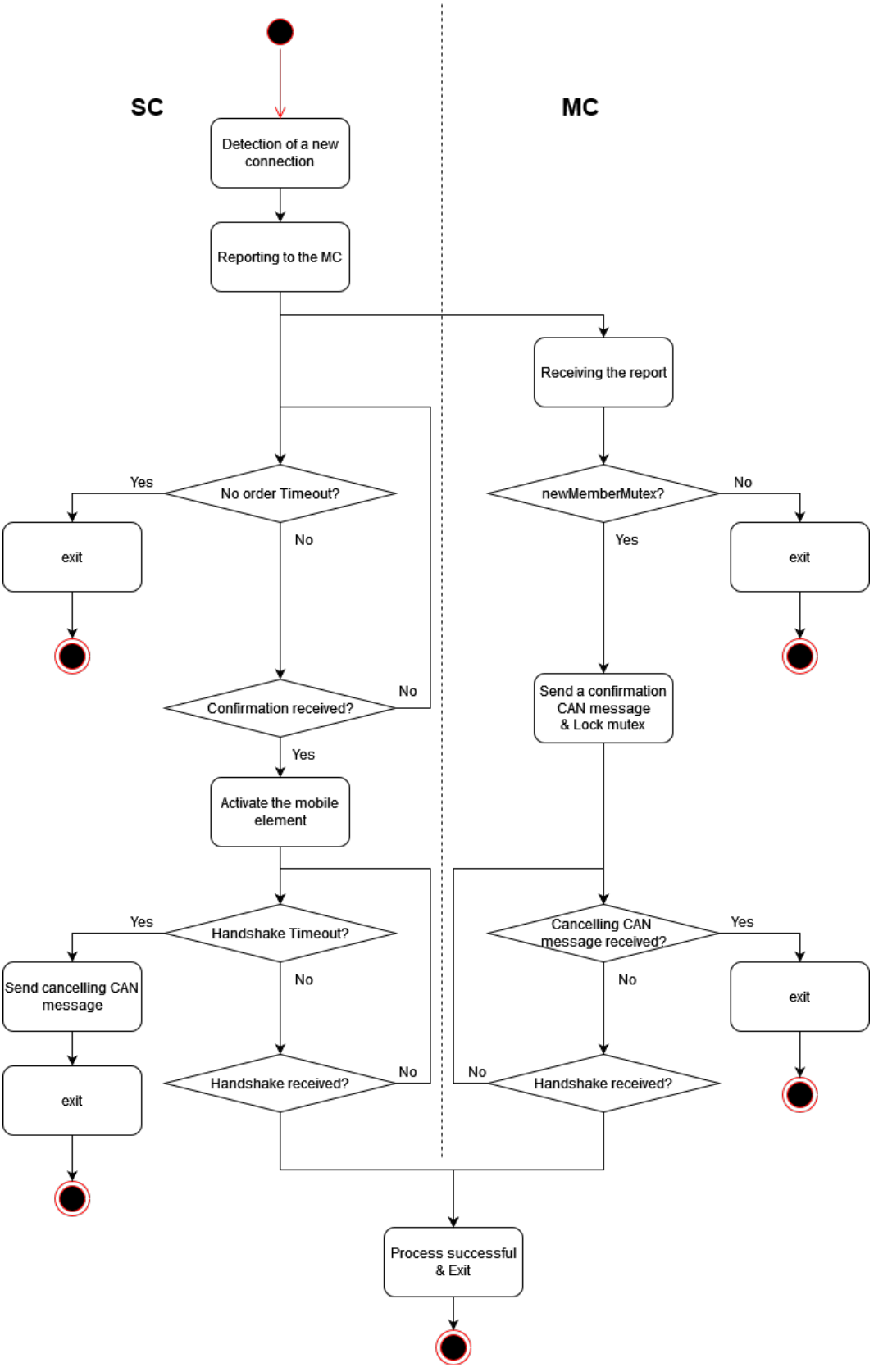


Figure 3.9: New connection detection routine

4 Testing the iHEX system

An essential aspect of system development involves rigorous testing to validate its operational capabilities and ensure the reliability of its intricate mechanisms. The iHEX system is no exception, and its comprehensive testing regime encapsulates the evaluation of its communication channels and the dynamic integration of new iHEX mobile elements.

4.1 Channel one test: Command propagation

To validate the effectiveness of the first communication channel, a controlled test was conducted. A command was initiated from the MC through the CT of the MQTT server. The command (cf. Listing **??**), in JSON format, directed the LED module to light on in red. As expected, the MC interpreted the command, translated it into a CAN message, and dispatched it to the pertinent SC. The SC promptly executed the command, resulting in the successful lightening on of the LED in the desired colour as shown in Figure 3.10.

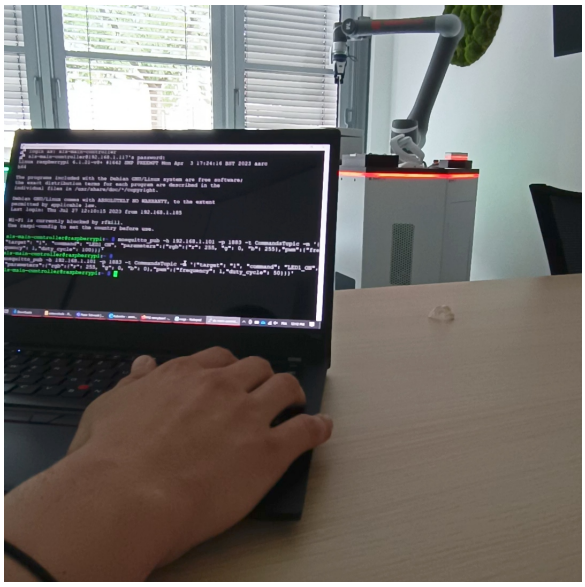


Figure 3.10: Validation of the command propagation

4.2 Channel two test: Feedback and reporting

The second communication channel underwent scrutiny through an evaluation of feedback reporting. An SC, responsible for monitoring environmental parameters, generated a CAN message reporting the connection lost with a mobile element, as shown in Figure 3.11 and 3.12. The MC intercepted the CAN message, deciphered its contents, and seamlessly transformed the information into an MQTT message in JSON format. This MQTT message was promptly published to the ET.

Figure 3.11: Detach of a mobile iHEX element

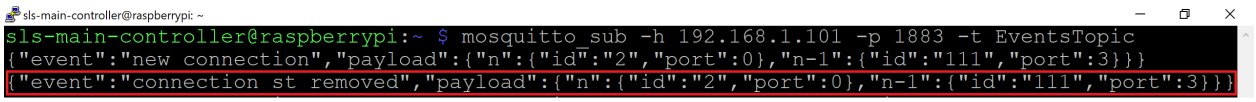


Figure 3.12: Reporting the lost connection on "Events Topic"

In Figure 3.12, a JSON message published on ET is displayed. The message has the following attributes:

- `"event"`: `"connection removed"`. It indicates that a removed connection is detected.
- `"payload"`: It precises the details of the reported event:
  - `"n"`: It reports the lost element details.
    - `"id"`: the id of the lost element (2 in Figure 3.12).
    - `"port"`: 0
  - `"n-1"`: It reports the static element to which the lost element was connected.
    - `"id"`: the id of the considered element (0x111 in Figure 3.12).
    - `"port"`: the exact port through which the lost element was connected (3 in Figure 3.12).

4.3 New connection integration test

The process of integrating a new iHEX mobile element was meticulously assessed. A new mobile element was connected as shown in Figure 3.13, triggering the Signal Loop Task's detection mechanism. The SC reported the connection to the MC through a CAN message, which, in turn, sent a confirmation and waited for a hand-shake signal. The mobile element became active and sent the hand-shake. As the MC publishes the reporting message to `"Events Topic"` as shown in Figure 3.14, the entire process confirmed successful integration.

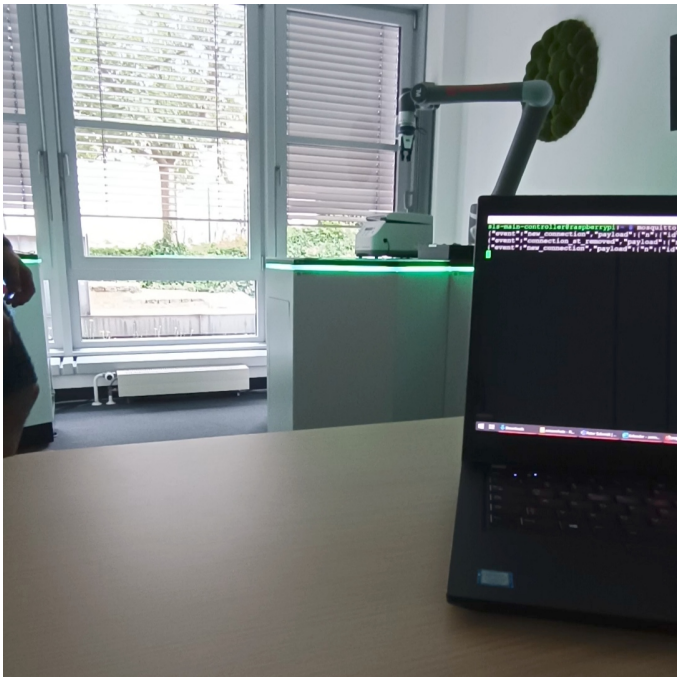


Figure 3.13: A mobile iHEX element connected an island

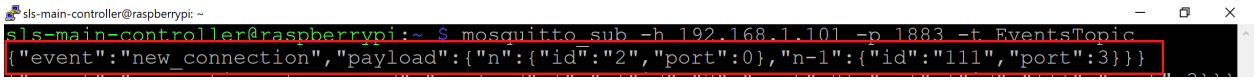


Figure 3.14: Reporting the established connection on "Events Topic"

In Figure 3.14, a JSON message published on ET is displayed. The message has the following attributes:

- `"event": "new connection"`. It indicates that a new connection is detected.
- `"payload"`: It precises the details of the reported event:
  - `"n"`: It reports the new connected element element details.
    - `"id"`: the id of the new connected element (2 in Figure 3.14).
    - `"port"`: 0
  - `"n-1"`: It reports the static element to which the new connected element is connected.
    - `"id"`: the id of the considered element (0x111 in Figure 3.12).
    - `"port"`: the exact port through which the new connected element is connected (3 in Figure 3.12).

5 Identifier configuration module

In our project, each SC plays a unique role, identified by its specific ID. However, managing these IDs in the software used to be a cumbersome process, involving code changes, recompilation, and re-flashing each time an ID needed modification.

To streamline this, we developed a configurable ID module that eliminates the need for code alterations and recompilation when changing an SC's ID. With this module, we first flash a generic code onto the STM32 microcontroller, simplifying the deployment process. Subsequently, we engage the configuration phase, which allows us to set and modify the IDs as needed.

This innovation presents several advanyages:

- It reduces the time and effort required for configuration.
- It enhances the flexibility and adaptability of our system, making it easier to manage unique IDs across multiple SCs.

The technical implementation of the configuration module centers on utilizing UART communication on the STM32F303RE MCU. During startup, the system awaits configuration instructions in a specific format, communicated via a UART message consisting of 2 bytes. Table 3.1 details the configuration message format:

4 bits	12 bits
0x41	ID in HEX format

Table 3.1: UART configuration message format

where `0x41` refers to the ASCII code of `'A'`.

To ensure this communication, UART2, embedded within the STM32F303RE MCU, is employed. It is configured to establish the interface with the MCU via the ST-Link port, which is soldered on the NUCLEO-F303RE board.

This UART-based configuration approach streamlines the process of assigning unique IDs to each SC without necessitating changes to the code, thereby enhancing configurability and flexibility.

6 Software development and deployment process

6.1 Software development and deployment process on the Raspberry Pi

A notable aspect of the software development process is the utilization of cross-compilation. To streamline efficiency, we developed the software intended for the Raspberry Pi on our Windows machines. Subsequently, we employed the `aarch64-linux-gnu` toolchain for cross-compilation [29].

The cross-compilation is adopted for its numerous advantages [30]:

- Development flexibility: Developers can work on their preferred development environment, typically a more powerful desktop or laptop, which can significantly speed up development compared to working directly on the Raspberry Pi.
- Reduced build time: Compiling large codebases on the Raspberry Pi itself can be time-consuming, whereas cross compilation allows developers to compile the code much faster on a more powerful host machine.
- Debugging Tools: Developers can use advanced debugging tools available on their host machine, making it easier to identify and fix issues in the code.
- Portability: Cross-compiled binaries can be easily deployed to multiple Raspberry Pi devices with the same architecture, simplifying the deployment process.
- Resource Efficiency: Cross-compiling doesn't consume Raspberry Pi resources, leaving them available for running and testing the compiled software.
- Compatibility: It ensures that the compiled software is compatible with the target Raspberry Pi's architecture, reducing the risk of compatibility issues.

The outcome of this process, known as `iHEX_MC.exe`, is then securely transferred to the Raspberry Pi via SSH for execution. This approach ensures a smooth and efficient deployment of our control interface software onto the Raspberry Pi.

6.2 Software development and deployment process on STM32 MCU

When working on the STM32 microcontroller, our software development process is equally efficient. Leveraging STM32CubeIDE [11], which is provided by STMicroelectronics and based

on Eclipse, we craft and fine-tune our software on our Windows machines. This IDE not only facilitates the coding process but also supports cross-compilation.

Once our software is polished and ready for deployment, it is cross-compiled and then seamlessly flashed into the flash memory of the STM32F303RE MCU using the ST-Link debugger. Additionally, STM32CubeIDE provides debugging capabilities, which prove invaluable when fine-tuning our code and ensuring optimal performance.

For monitoring the flash memory registers when needed, we employ STM32CubeProgrammer [14], which rounds out our comprehensive software development toolkit.

Conclusion

Chapter 3 explored how the Control Interface operates within the iHEX system, involving both the MC and SCs. The MC manages commands, while SCs handle real-time tasks and hardware control.

We also discussed how MC and SCs communicate, emphasizing their collaboration in the iHEX system. We highlighted how new mobile elements are integrated, showcasing synchronization and testing.

Looking ahead, Chapter 4 will delve into the design and development of crucial PCBs, the building blocks of our system.

Chapter 4
Design and development of Printed Circuit Boards

Chapter 4

Design and development of Printed Circuit Boards

Introduction

In Chapter 4, we dive into the essential process of creating the electronic foundation of the iHEX system—Printed Circuit Boards (PCBs). These PCBs serve as the crucial framework where all the electronic parts of the iHEX system come together, ensuring seamless communication and control. This chapter explores the design, development, and roles of four specific PCB types: Main PCB, IO PCB, LED PCB, and Dock PCB.

1 Architecture

Based on the specifications and the company needs, we designed the global hardware architecture. It is mainly composed of 4 designed PCBs:

- Main PCB is the motherboard that brings the main electronic components of the SC together. Each SC is composed of one Main PCB that is connected directly to an IO PCB.
- IO (Input-Output) PCB mainly ensures connectivity and interaction with the other SCs. It is connected to the Main PCB on one side. On the other side, it is connected whether to IO PCBs of other static SCs or Dock PCBs.
- Dock PCB ensures the interaction and control of the iHEX mobile element newly connected. If it exists, whether in a static or mobile element, it is connected to the SC through the IO PCB.
- LED PCB controls the LED strip. Each LED PCB is part of a SC. It is plugged-in directly on the Main PCB.

The following Figure 4.1 explains graphically the different connections between the PCBs within the suggested architecture:

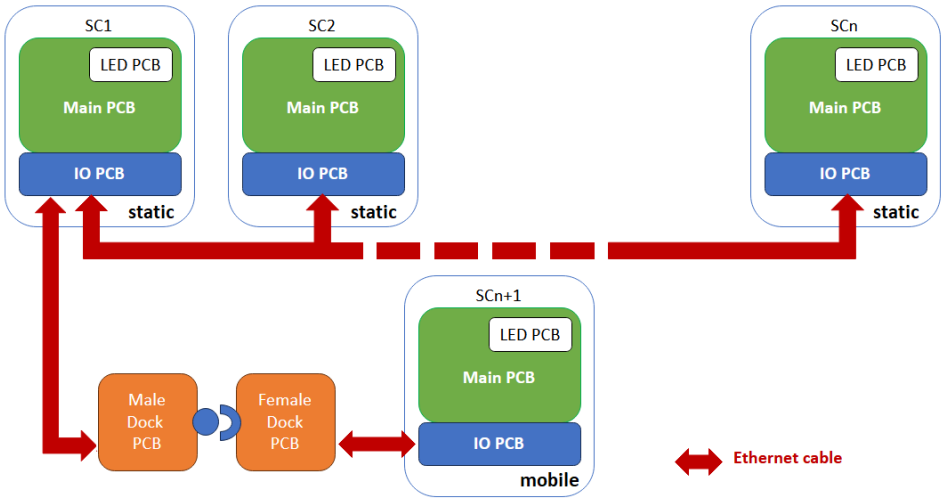


Figure 4.1: The different connections between the PCBs

In Figure 4.1 `SC1`, `SC2`, and `SCn` are static elements. They are part of the same island. They are connected to each other thanks to their IO PCB. `SCn+1` is a mobile element.

`SC1` is equipped by a Dock that offers the possibility of connecting a new mobile element. The Male Dock PCB is connected to `SC1` via its IO PCB. Same for the Female Dock PCB, it is connected to SCn+1 via its IO PCB.

To design the needed PCBs, we use the Computer Aidied Design (CAD) software KiCad [31].

2 | Main PCB: Nucleus of the iHEX hardware

As the central part of the SC, the Main PCB shown in Figure 4.2, basically assembles the different electronic components that define its functionality. This section details the design, integration, and functionality of the Main PCB—where the NUCLEO-F303RE microcontroller, CAN transceiver, power supply, LED controllers, and more converge to drive the iHEX system's SC capabilities.

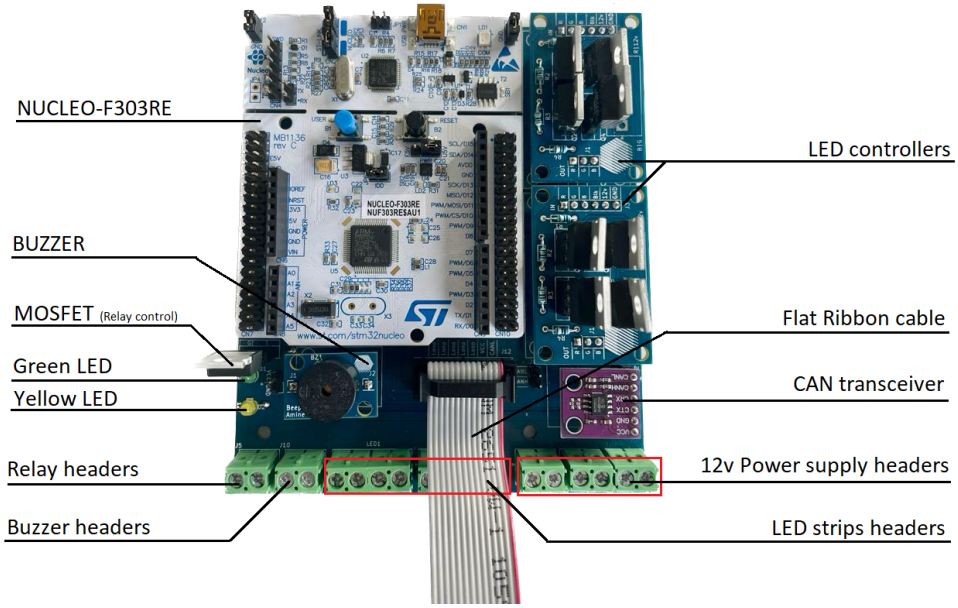


Figure 4.2: Main PCB overview

2.1 Electronic components

The following are the components that are part of the Main PCB:

- **NUCLEO-F303RE MCU:** The NUCLEO-F303RE microcontroller serves as the system's brain, responsible for processing instructions and coordinating actions across the entire iHEX network. Its integration was facilitated by a meticulously downloaded package from SnapEDA, ensuring precise footprints and compatibility.
- **CAN transceiver:** The Main PCB integrates a custom-designed CAN transceiver footprint, enabling smooth communication between various elements of the iHEX system. This CAN transceiver bridges the gap between the CAN bus network and the microcontroller, allowing for reliable data transmission and reception.
- **Header connectors:** A matrix of header connectors emerges as the physical interface between the Main PCB and the iHEX system's external elements. The needed headers are summarized in Table 4.2 below:

Quantity	Component	Role
3	2-Position headers (12V-GND)	Facilitate power supply connections, enabling the efficient distribution of 12V power and ground signals.
2	4-Position headers (LED Control)	Govern the RGB LED strips' behavior, providing ground and 12V connections for dynamic illumination.
1	2-Position header (External Buzzer Control)	Enable external control over the system's auditory feedback via a simple signal.
1	2-Position header (Relay Control)	Regulate electricity to the device atop the iHEX element, enhancing control capabilities.
2	6-Position female headers (LED driver control)	Manage LED driver control signals, harmonizing the visual experience.
2	3-Position female header (LED driver control)	
1	6-Position female header (CAN transceiver)	Establish a seamless connection between the CAN transceiver and the network.
1	14-Position connector (Flat Ribbon Cable)	Link the Main PCB with the IO PCB, ensuring CAN communication, and signal loop and relay control signal transmission.
2	1-Position female header (Internal Buzzer control)	Enable internal control over the system's auditory feedback.
1	3-Position female header (MOSFET)	Facilitate control over the MOSFET that governs electricity supply to the system's top device.
1	2-Position male header (CANH and CANL for Troubleshooting and Visualization)	Provide direct access for troubleshooting and visualization of CAN signals.
1	2-Position male header (VCC and Ground for undefined purposes)	Offer flexible connection points for future expansion.

Table 4.1: Electronic components in the Main PCB

- **Indicator LEDs:** The two indicator LEDs in Table 4.2 contribute to the Main PCB's intuitive user interface:

Quantity	Component	Role
1	Green LED (VCC - 3V3 Indicator)	Illuminates to signify the presence of a 3.3V supply.
1	Yellow LED (VDD - 12V Indicator)	Lights up to indicate the availability of a 12V supply.

Table 4.2: Main PCB LED indicators

2.2 Schematic, integration, and PCB design

The Main PCB's construction encompassed the schematic design, footprint assignment, and PCB layout. The schematic diagram, as shown in Figure **??**, interwove necessary wiring, outlining the connections between components. Each component's footprint was assigned to ensure a perfect fit and proper connectivity. The resulting PCB design in Figure 4.3, encapsulates the harmonious fusion of components, underscoring the NUCLEO-F303RE's principal role of controling the iHEX system.

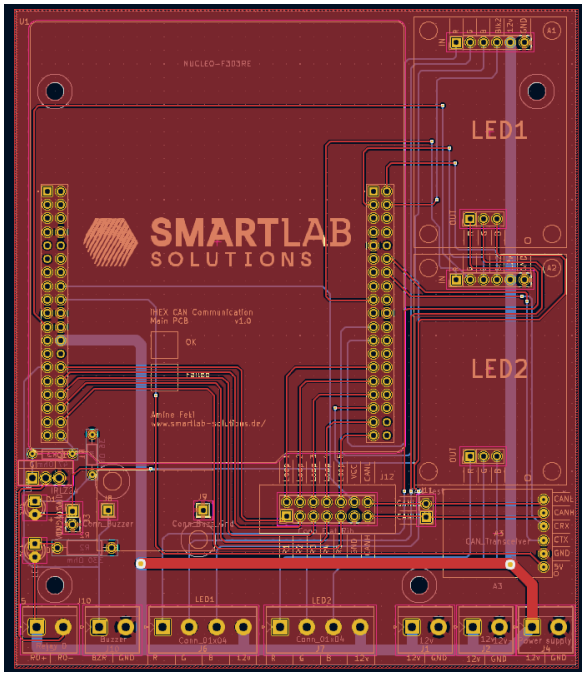


Figure 4.3: Design of the Main PCB

3 IO PCB: Connectivity and control

The IO PCB, as shown in Figure 4.4, serves as the interface that empowers the connected SC to connect, communicate, and control the other members of the island (MC and other SCs).

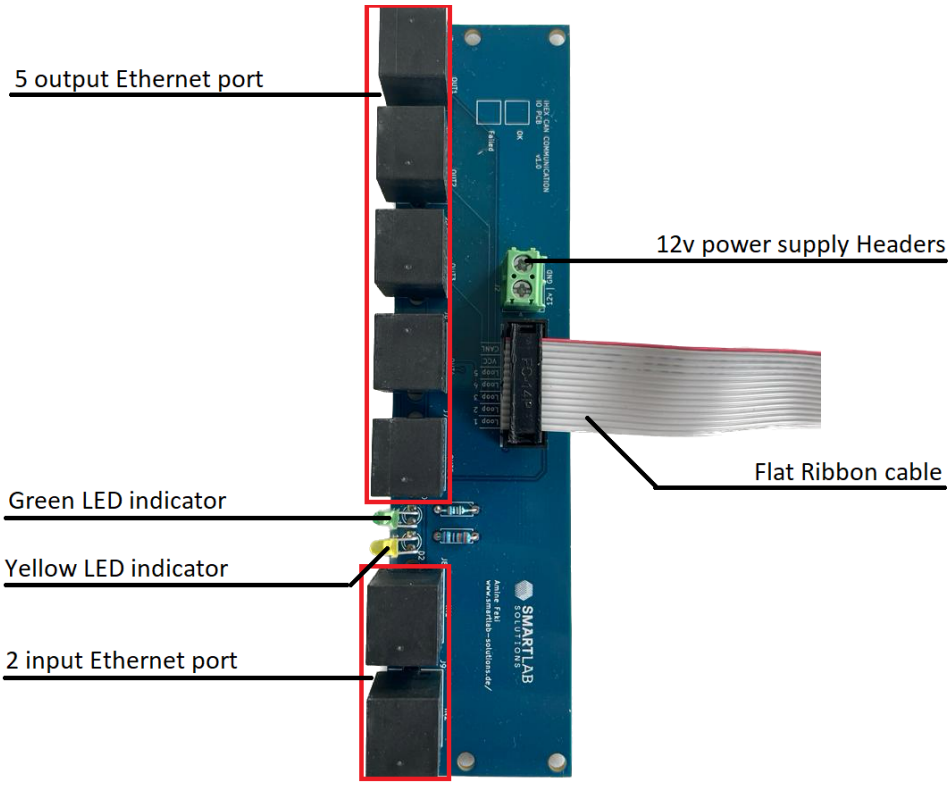


Figure 4.4: IO PCB overview

3.1 Ethernet ports:

At the main part of the IO PCB the Ethernet ports play a distinctive role in facilitating communication and connectivity:

- 2 Input Ethernet Ports: Establish a vital link between the current iHEX element and the CAN bus via CANH and CANL signals. These ports serve for communication between the SC and the other parts of the island (MC and other SCs).
- 5 Output Ethernet Ports: These ports accommodate both static and mobile iHEX elements' unique needs:
  - Static iHEX Element Output: For static elements, these ports offer CANH and CANL terminals to facilitate the connection of subsequent static iHEX elements to the CAN bus.
  - Mobile iHEX Element Output (Connected to Dock): When connected to a mobile iHEX element via the Dock, these ports become the interface of connectivity and control:
    * CANH and CANL Terminals: Transmit CAN communication signals, allowing integration into the iHEX network.
    * Signal Loop Detection: Receive the signal loop value (3.3V or GND) from the dock, detecting connection or disconnection of mobile elements.

* 12V Power Supply: Provide a regulated 12V power supply to the relay for electricity control.
* Relay Trigger: Receive a 3.3V signal to trigger the relay, enabling precise control over electricity supply.

3.2 Header connectors

The IO PCB features essential header connectors (Table 4.3) that facilitate communication and integration with the Main PCB and other system elements:

Quantity	Component	Role
1	14-Position Header (Flat Ribbon Cable)	Form a vital connection to the Main PCB, transmitting the signal loop information, receiving relay control signals, and facilitating CAN bus connectivity via CANH and CANL terminals.
1	2-Position Header (12V Power Supply)	Provide a standardized 12V power supply to system elements, ensuring consistent and reliable operation.

Table 4.3: IO PCB Header

3.3 Indicator LEDs

Two indicator LEDs (Table 4.4) provide instant visual feedback on the IO PCB's status:

Quantity	Component	Role
1	Green LED (3.3V Indicator)	Lights up to signal the presence of a 3.3V supply, confirming the availability of essential power.
1	Yellow LED (12V Indicator)	Provide a standardized 12V power supply to system elements, ensuring consistent and reliable operation.

Table 4.4: IO PCB LED indicators

3.4 Schematic integration, and PCB design

We brought the IO PCB to life through creating a schematic design (Figure **??**), assigning footprint, and developing the PCB layout (Figure 4.5). The schematic diagram outlines wiring connections and interactions between components, ensuring the different functionalities. The assignment of each component's footprint guarantees proper fitment and effective integration. The PCB in Figure 4.5 presents the final result.

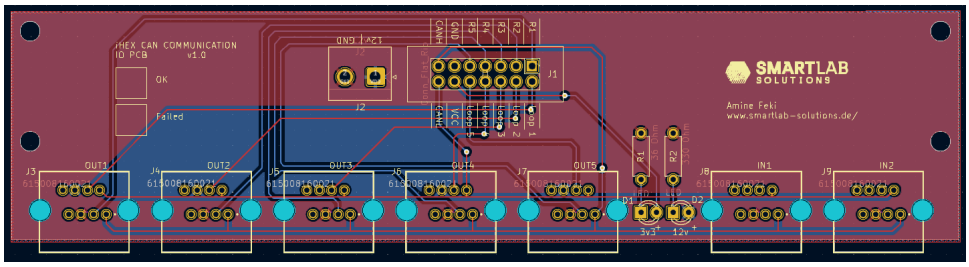


Figure 4.5: Design of the IO PCB

4 Dock PCB: Connecting and powering iHEX mobile elements

Dedicated to the task of connecting mobile elements to static ones, the Dock PCB (Figure 4.6) stands as a bridge that enables the integration between these elements. This section details the design and integration of the Dock PCB, taking in consideration the Male Dock and Female Dock compositions.

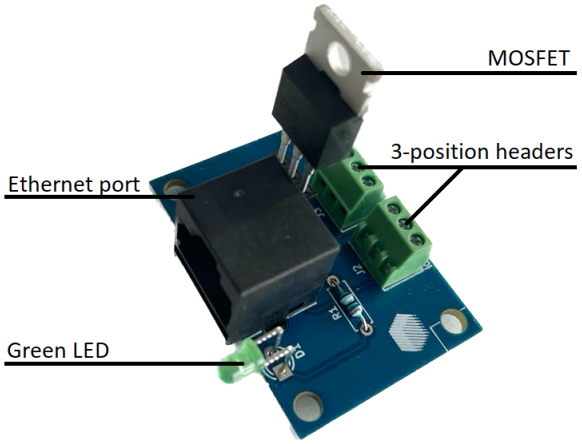


Figure 4.6: Dock PCB overview

4.1 Male Dock

The Male Dock presents the PCB that is integrated in the Dock unit if it exists. In junction with the Female Dock, it contributes to the integration of the iHEX mobile elements. It is mainly composed of:

- Ethernet port: It is used for connecting the Dock to the SC via the IO PCB Output port. It contributes to the establishment of a bidirectional communication between mobile element and the other elements of the island.
- MOSFET: It triggers the relay that governs the power supply of the mobile element.

- Green LED (Relay Control Visualizer): It offers instant visual feedback, illuminating when the relay control signal is active, and extinguishing when inactive. This LED enables visual debugging.
- 2 * 3-Position headers: These connectors provide essential connectivity:
  - CANH and CANL Terminals: Establish CAN communication channels, ensuring fluid integration into the iHEX network.
  - Signal Loop detection: Receive the signal loop information, indicating the connection status of a new mobile element.
  - Relay control signal: Enable the power supply activation and deactivation of mobile elements.
  - Common Ground: Ensure common ground for all the iHEX system.

4.2 Female Dock

The Female Dock presents the PCB that is integrated in the Dock unit if it exists. In junction with the Male Dock, it contributes to the integration of the iHEX mobile elements. It is mainly composed of:

- Ethernet port: It establishes a direct connection to the IO PCB's Input Ethernet port.
- 1 * 3-Position Header: This connector forms the foundation of essential connections:
  - CANH and CANL terminals: Create a vital link for CAN communication, enabling data exchange with the iHEX network.
  - Common Ground: Ensure common ground for all the iHEX system.

4.3 Schematic, integration, and PCB Design

We brought the Dock PCB to life through creating a schematic design (Figure **??**), assigning footprint, and developing the PCB layout (Figure 4.7). The PCB in Figure 4.7 presents the final result.

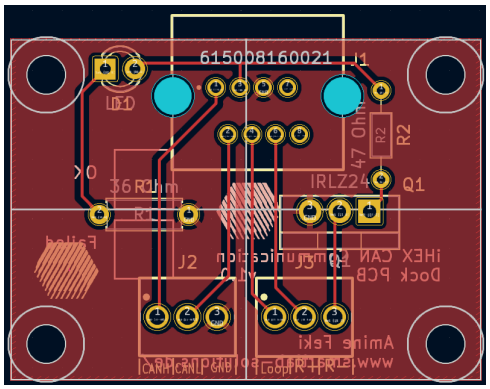


Figure 4.7: Design of the Dock PCB

5 LED PCB: LED strips control

Dedicated to visual signalling the user, the LED PCB (Figure 4.8) controlled by the SC MCU, controls the LED strips of the iHEX element. This section unveils the design and integration of the LED PCB.

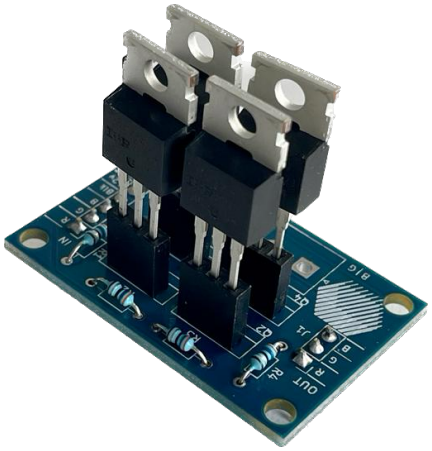


Figure 4.8: LED PCB overview

5.1 RGB color control

The LED PCB is mainly composed of four sophisticated MOSFETs:

- 3 MOSFETs for RGB color control: These dynamic components govern the intensity of each primary color—Red, Green, and Blue. That is achieved by modulating the voltage across the LEDs using PWM signals.
- 1 MOSFET for blinking control: A singular MOSFET assumes the role of controlling the blinking frequency and duty cycle of the control PWM signal.

5.2 Input and Output connections

The LED PCB takes LV (Low-Voltage) control signal (3v3 - GND) as input and delivers MV (Medium-Voltage) control signal (12v - GND). Two connectors make the LED PCB interact with the external electronic components:

- 1 * 6-Position input connector: This input hub encapsulates the RGB color and blinking control:
  - R, G, B, Blk (Blinking) PWM inputs: CMOS PWM signals that control the the colors and the blinking frequency and duty cycle.
  - 12V and GND inputs: Provide the necessary power to illuminate the LEDs.
- 1 * 3-Position output connector: MV PWM signals. It is the amplified image of the CMOS control signal received by the LED PCB.

5.3 Schematic integration, and PCB design

The LED PCB's design process starts with a crafted schematic diagram (Figure 3.3). To each electronic component and depending on the used reference, we assign the corresponding footprint. Finally, we design the PCB (Figure 4.9) which is the final step before printing the final board.

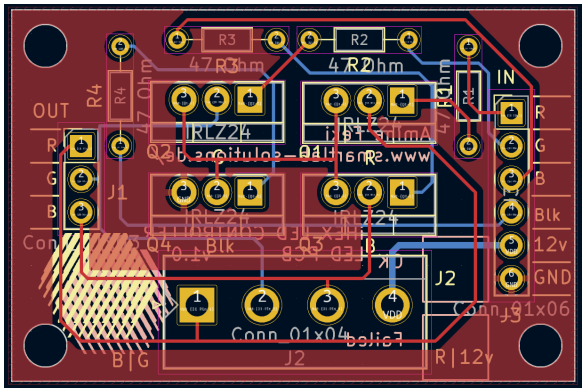


Figure 4.9: Design of LED PCB

6 Production, test and validation

The design and development efforts found realization through the production of the PCBs. The production phase was entrusted to "Pcbcart" [32], a leading Chinese company renowned for its precision and quality. To do so, we generated the necessary Gerbers (.gbr) and Drill Files (.drl) for the fabrication: routes and drills. Upon receiving the fabricated boards, the electronic components which are delivered by trusted suppliers such as Digikey [33], Mouser [34], and Würth Electronik [35] are soldered on the PCBs.

Assembling PCBs, electronic elements, and electrical components lead to obtaining the final SC (Figure 4.10) and Dock boxes, laying the groundwork for comprehensive testing and validation.

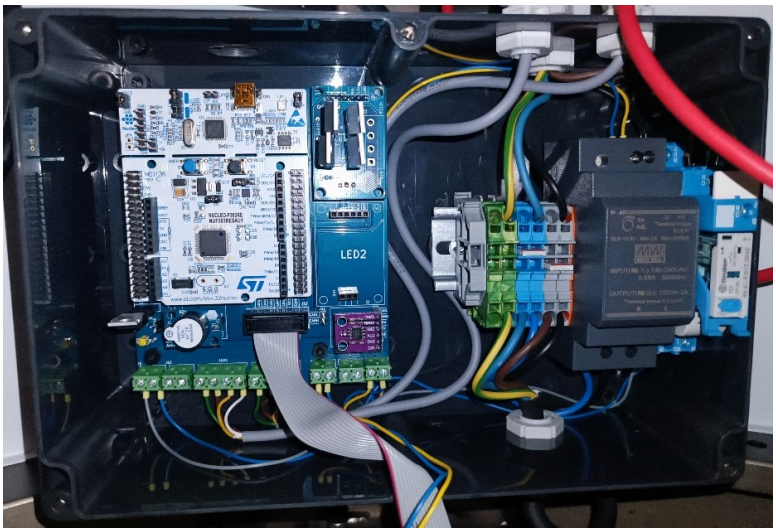


Figure 4.10: SC wiring

In the company's showroom, the island is mainly composed of six iHEX elements and one mobike element equipped each of a SC (Figure 4.11). Two static elements are equipped each with a Dock (Figure 4.12) allowing the connection of the mobile element.

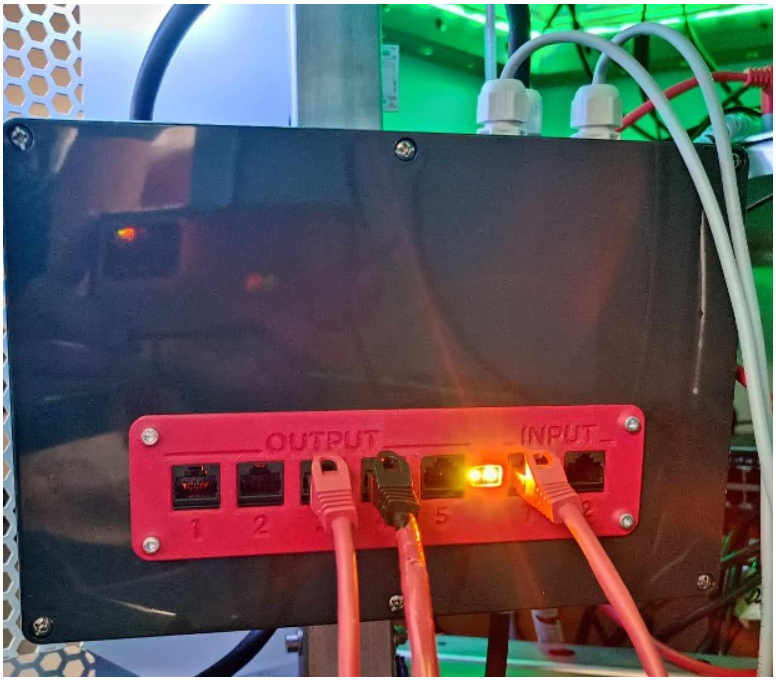


Figure 4.11: SC final box

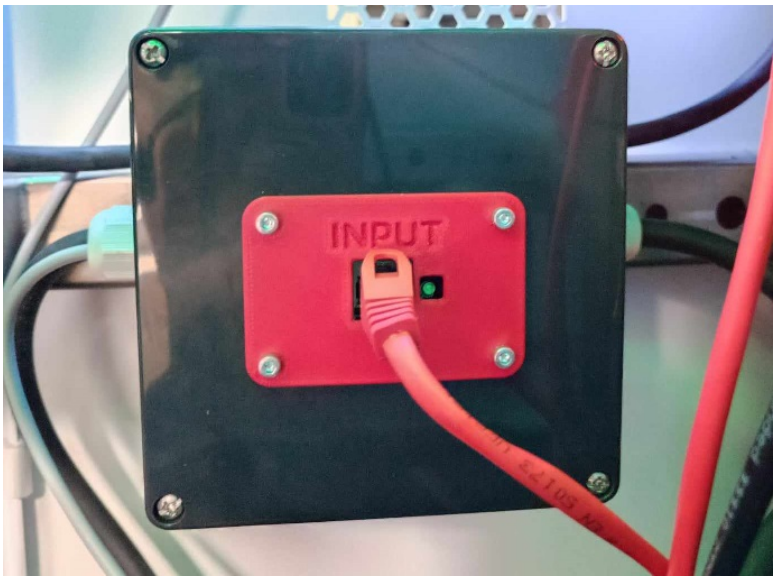


Figure 4.12: Dock final box

The finalized PCBs and the developed CI software are ready to be tested. We make the necessary wiring, bring everything together and start the test and validation process. During this phase, we test the communication across both channels and the iHEX elements ability to correctly respond to the MQTT commands, notably the LED strips and Buzzers control as well as powering on and off the electrical devices. We also have to validate the mobile element connection on one Dock through the reception of a significant MQTT message.

Succeeding the testing phase makes us not only validate our engineering job of design and development of the PCBs, but also make sure of the integrity of the software and hardware final product.

Conclusion

Chapter 4 took us through the process of making important PCBs. We designed these boards, and then turned the designs into real boards with the help of a partner company. After putting the parts on the boards and connecting everything, we tested them to make sure they work well and adapted to the software in real situations. This process shows how software and hardware come together to make the iHEX system work smoothly.

General conclusion

Bibliography

[1] Wikipedia. URL: https://en.wikipedia.org/wiki/Digitization

[2] SmartLab Solutions GmbH website. URL: https://www.smartlab-solutions.de/

[3] SmartLab Solutions GmbH Introduction - Präsentation.

[4] Qualitätsmanagementhandbuch der SmartLab Solutions GmbH.

[5] Attlassian website. URL: https://www.atlassian.com/agile

[6] Wikipedia. URL: https://en.wikipedia.org/wiki/CAN_bus

[7] Texas Instruments article. URL: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1693476287427

[8] Raspberry Pi website. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[9] STM32F303RE datasheet. URL: https://www.st.com/resource/en/datasheet/stm32f303re.pdf

[10] NUCLEO-F303RE User Manual. URL: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

[11] STM32CubeIDE website. URL: https://www.st.com/en/development-tools/stm32cubeide.html

[12] STM32CubeMX website. URL: https://www.st.com/en/development-tools/stm32cubemx.html

[13] STM32CubeMonitor website. URL: https://www.st.com/en/development-tools/stm32cubemonitor.html

[14] STM32CubeProgrammer website. URL: https://www.st.com/en/development-tools/stm32cubemonitor.html

[15] Waveshare website. URL: https://www.waveshare.com/wiki/RS485_CAN_HAT

[16] MCP2515 datasheet. URL: https://www.waveshare.com/w/upload/8/83/MCP2515.pdf

[17] can-utils GitHub repository. URL: https://www.waveshare.com/w/upload/8/83/MCP2515.pdf

[18] MCP2551 Micochip website. URL: https://www.microchip.com/en-us/product/mcp2551#document-table

[19] STM32F303RE reference manual, 31 Controller area network (bxCAN), bxCAN functional description, Bit timing, Figure 397, p. 1027. URL: https://shorturl.at/puD67

[20] STM32F303RE reference manual, 31 Controller area network (bxCAN), Test mode, Loop back mode, p. 1016. URL: https://shorturl.at/puD67

[21] Official Documentation: POSIX pthread.h Header File Specification. URL: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

[22] Wikipedia. URL: https://en.wikipedia.org/wiki/MQTT

[23] paho.mqtt.c library. URL: https://github.com/eclipse/paho.mqtt.c

[24] paho.mqtt.cpp library. URL: https://github.com/eclipse/paho.mqtt.cpp

[25] Wikipedia. URL: https://en.wikipedia.org/wiki/JSON

[26] nlohmann/json library. URL: https://github.com/nlohmann/json

[27] FreeRTOS website. URL: https://www.freertos.org/index.html

[28] STM32F303RE reference manual, 21 General-purpose timers (TIM2, TIM3, TIM4) and 23 General-purpose timers (TIM15, TIM16, TIM17). URL: https://shorturl.at/puD67

[29] SysGCC website, aarch64-linux-gnu toolchain. URL: https://gnutoolchains.com/raspberry64/

[30] Wikipedia. URL: https://en.wikipedia.org/wiki/Cross_compiler

[31] KiCad official website. URL: https://www.kicad.org/

[32] Pcbcart website. URL: https://www.pcbcart.com/

[33] Digikey website. URL: https://www.digikey.de/

[34] Mouser website. URL: https://www.mouser.de/

[35] Würth Elektronik website. URL: https://www.we-online.com

Annexes