Universidade do Minho
Escola de Engenharia

EMBEDDED SYSTEMS
RESEARCH
GROUP

Master's in Industrial Electronics and Computers Engineering

Embedded Systems

Automated Greenhouse

Authors:
Alfredo Rodrigues
João H. Ferreira

Professor:
Adriano Tavares

January 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Automation grows every day in different aspects of our world. Nowadays, as society has advanced in recent years, one area that has faced enormous challenges is agriculture. This is due to the growing demand for food, but in a sustainable and efficient way.

In this context, greenhouses are one of the main forms of cultivation today, because they offer a controlled environment that allows plants to be grown all year round practically regardless of the weather conditions and their changes.

With this in mind, the project would be the design of an automated greenhouse, which emerged as an attempt to optimise traditional agricultural practices. These revolutionary structures go beyond ordinary greenhouses, incorporating various automatic systems to maximise productivity and minimise the waste of resources. This is increasingly important due to the growing challenges related to climate change, increased demand for good food or water scarcity.

1.2 Problem Statement Analysis

The aim of this project is to automate a conventional greenhouse without compromising its correct operation.

Conventional greenhouses have been able to combat some of these issues, including climate variability, but although they are fundamental, they still fall far short, facing challenges such as dependence on intensive manual practices or exposure to human error, which can lead to poor decisions or mismanagement of natural resources.

To this end, the product will use irrigation systems, temperature, humidity and CO2 measurement as well as water reservoirs for better utilisation of resources with less physical effort.

The various parameters of the greenhouse can be controlled remotely. This makes it more independent so that the farmer doesn't have to go there every day.

Chapter 2

System

2.1 System Overview

The system overview provides a visual representation of the interplay among various subsystems, this description deepens our grasp of the product's core.

The greenhouse will automatically adjusts the parameters according to the previously defined on the database.

The user will be able to access the contents of the greenhouse by entering the correct pin on a keypad, he will also have access to control the parameters in real time from an app, seen in figure 1.

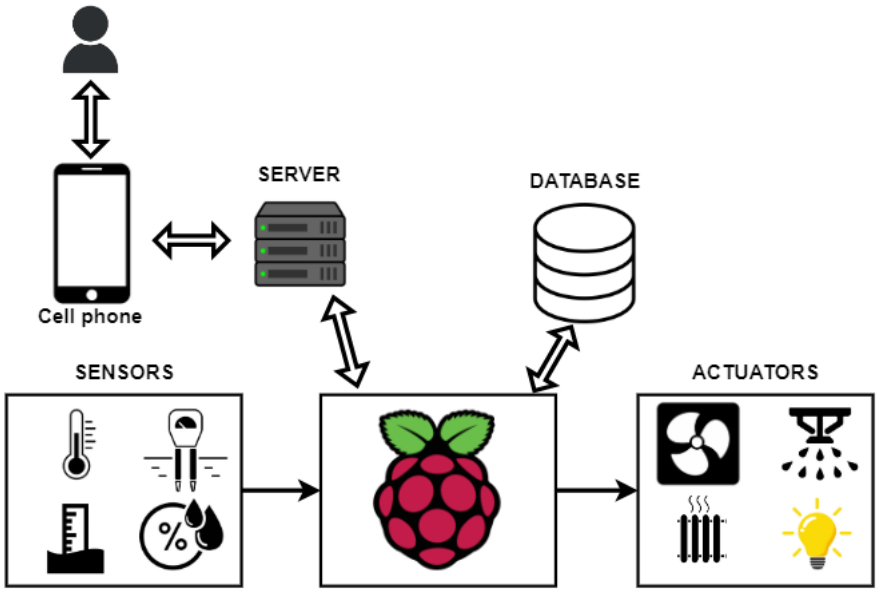

Figure 1: System overview representation

2.2 Requirements & Constraints

2.2.1 Requirements

In our project there are some different specifications we must accomplish. These requirements can be divided into two groups, technical and non-technical.

Functional Requirements

- Control and adjust temperature, humidity, light and CO2 levels
- Monitor water level of the tank
- Control access through a security door
- View metrics remotely through GUI

Non-Functional Requirements

- Reliable
- Secure
- User-Friendly

2.2.2 Constraints

In terms of constraints these can also be divided in the same categories as the requirements.

Technical Constraints

- Use a Raspberry Pi 4B
- Use Linux and Buildroot
- Use the C++ programming language

Non-Technical Constraints

- Limited human resources (2 persons in the group)
- The deadline is the end of the semester
- Limited financial budget

2.3 Market Research

Market research, figure 2, is a very important step when designing a new product. Done well, it allows developers to understand the target audience and increase the likelihood of success. Commitment to greenhouse automation has increased and is expected to continue to grow. Annual growth of around 9.7% is estimated until 2028. The Asian continent has invested the most in this area, but the United States is still the country where this type of product is most common. Although investment in this area is also growing in Europe, it still lags behind Asia and North America.

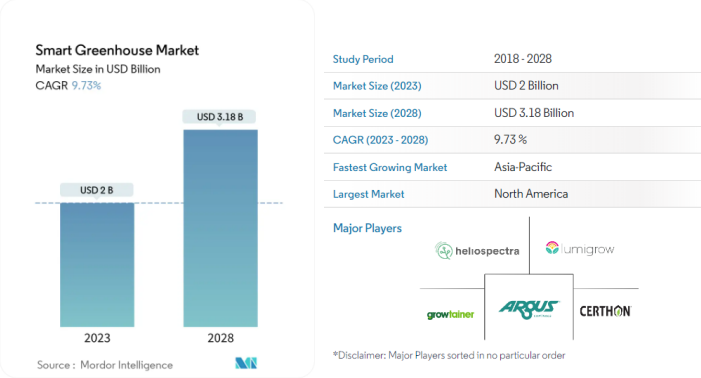

Figure 2: Smart greenhouse market study
[1]

In general, the price of the initial investment and the lack of knowledge on the part of the people working in this field (farmers) means that this market doesn't develop very quickly. The general development of society tends to counteract this.

For example, Bluelab is a company that both converts normal greenhouses into automated greenhouses and designs them from scratch. This is one of the latest projects they've developed, figure 3, where it's possible to control all the parameters of the greenhouse, from the temperature to the pH of the water.

Figure 3: Bluelab automated greenhouse solution
[2]

A small automated greenhouse of 3mx3m can cost around €3,000, so these greenhouses can easily cost thousands of euros, depending on various factors such as the technology and structure used, or even the city chosen to build the greenhouse.

2.4 System Architecture

2.4.1 Hardware Architecture

The hardware architecture is composed of three systems: the main system, sensors, and actuators. The Raspberry Pi 4B serves as the core of the system where all information is processed from both the inputs and the GUI through the database. The input system includes several sensors such as temperature, humidity, light and water level in the tank. The output system is composed of various actuators that control the parameters, as can be seen in the figure 4.

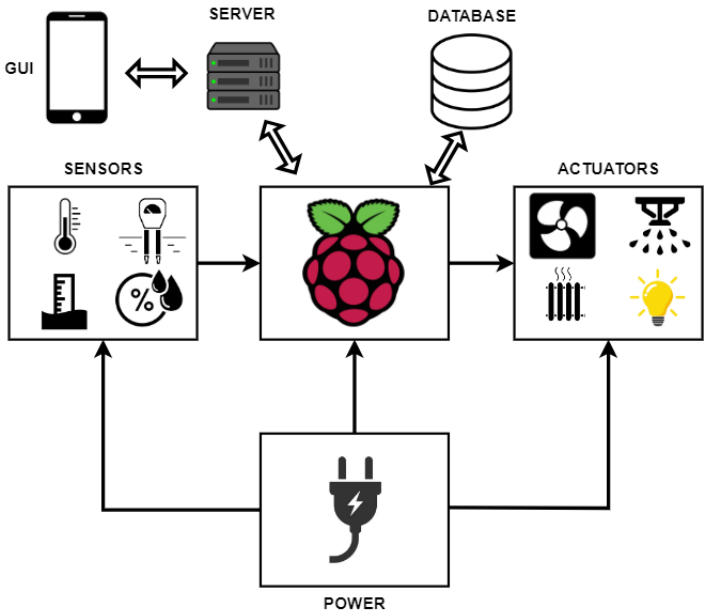

Figure 4: Hardware architecture

2.4.2 Software Architecture

The software system comprises three layers: application, middleware, and custom Linux OS. The lowest layer includes device drivers for the sensors and actuators, while the middle layer is dedicated to data processing and acquisition. Moreover, it uses the Pthreads model enabling the creation and manipulation of threads.

Finally, the GUI and the database for storing system and parameter information are located in the application layer.

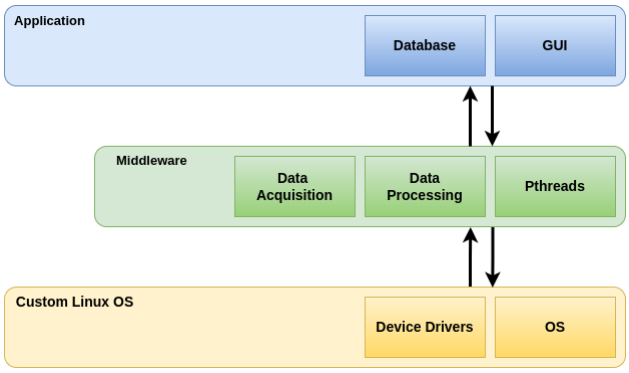

Figure 5: Software architecture

Chapter 3

System Analysis

3.1 Events

In the following table, its possible to view the system events that may occur on both the local and remote systems.

Events	**System Response**	**Trigger**	**Type**
Power On	Initialize the sensors	User	Asynchronous
Read Sensors	Collect data sensors	Timer	Synchronous
Control Parameters	Turn on/off actuators	System	Synchronous
Save Data	Store values in database	System	Synchronous
Import Data	Read from database the set values	Timer	Synchronous
Display Metrics in App	Show the metric in remote system	System	Synchronous
Set Parameters Remotely	Change the reference values	User	Asynchronous
Greenhouse Access	Open the security door	User	Asynchronous
Temperature Threshold	Shutdown heat system + activate cooling system + alert sent to user app	System	Asynchronous
Calculate Average Values	Get values from sensors and calculates average	System	Synchronous
Change Actuators State	Set or Reset actuators depending on the reference previously calculated	System	Asynchronous
Connection Lost	Try to restore connection + after timeout alert is sent to user app	System	Asynchronous

Table 1: Events table

3.2 Use Cases

A use case is a series of actions and events that define the system-user interactions needed to achieve a specific goal. Studying these system behaviours and project requirements is crucial. In the figure 6 its possible to see the use case diagram related to the system of this project.

There will be 5 actors in the system: the GUI, timer, actuators, temperature trigger and system.

The user will interact with the app and after logging in will be able to check the current values of their greenhouse, change the set values of the greenhouse or open the security door. The door is associated with the security system in which the behaviour of the door is similar to the actuators.

The values read from sensors are stored in the system and after sent to database, while the set values are sent the other way around, from the app to the system.

The sensors read the values from the greenhouse, this values are processed and if necessary the parameters are adjusted by the actuators.

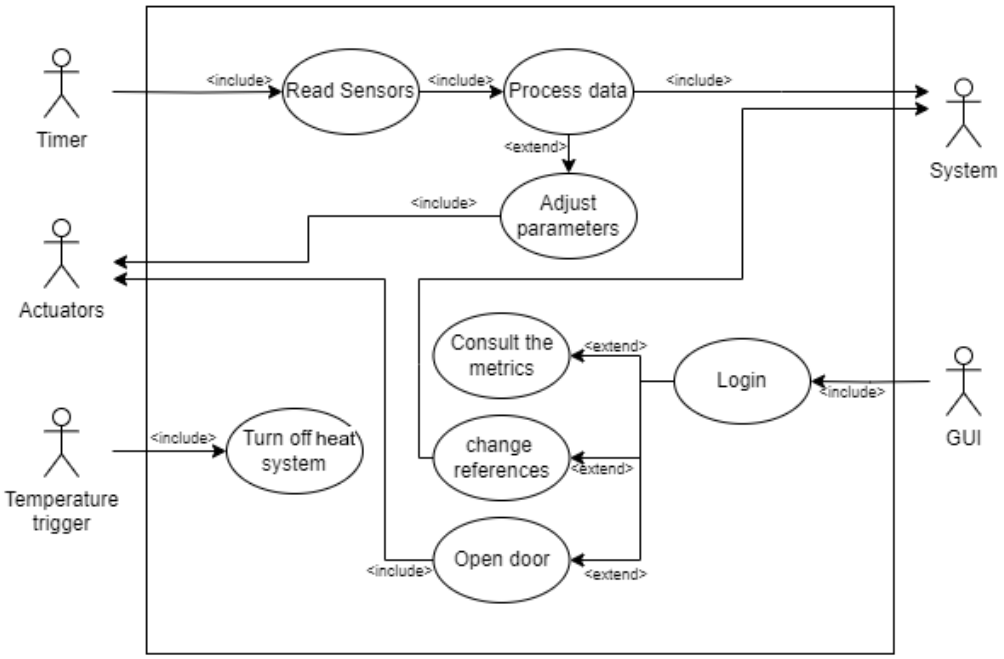

Figure 6: Use case diagram

3.3 State Chart

In the following state machine, figure 7, its possible to see that after switching on, the system enters an initialization phase where the sensors are initialized. The system then enters the IDLE state and waits for an event to occur.

If the sensor trigger is activated, the sensor values are read and sent to the database. Additionally, the actuators are activated, if required, to maintain the parameters according to the given conditions.

If the door is opened by triggering, it can remain open for a predetermined time. In a separate state, the values in the database from the remote control will be accessed and modified if required.

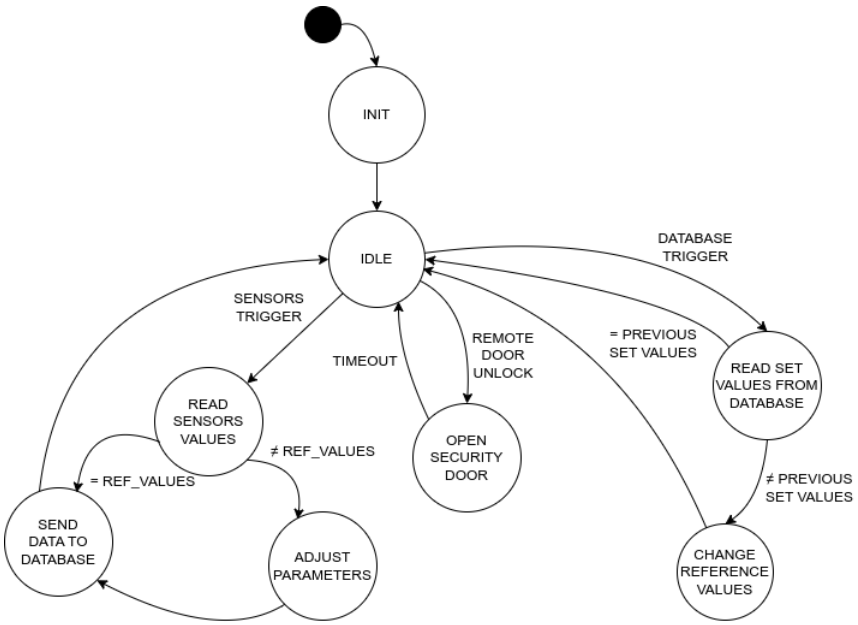

Figure 7: State chart

3.4 Sequence Diagram

Sequence diagrams are used to describe how the actors and the system interact with each other in a temporal sequence.

This sequence diagram, figure 8, shows the process of acquiring data from the sensors and controlling the parameters via the actuators. The database will be responsible for capturing the current greenhouse values and defining new set values if the user so wishes.

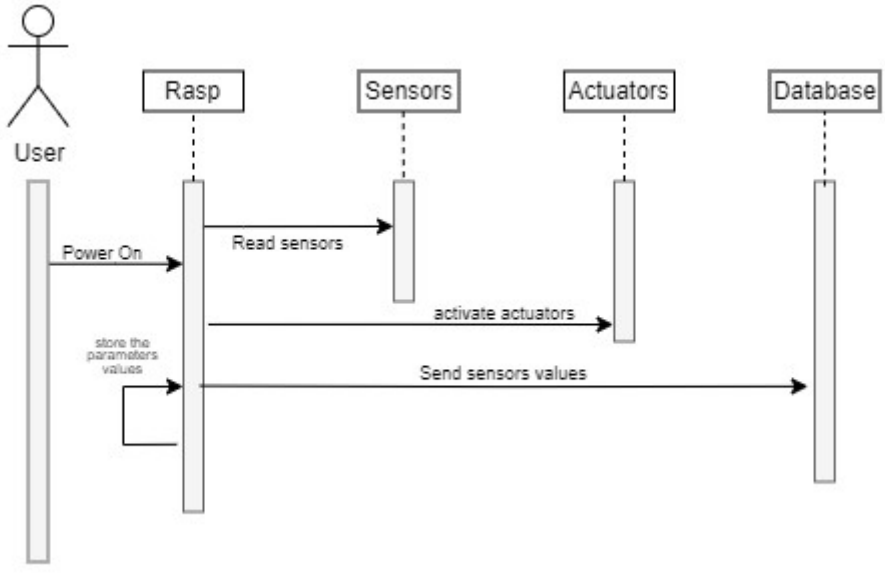

Figure 8: Data acquisition and parameter control sequence diagram

The graphic on figure 9 shows the interaction between the user and the app. After logging in, it will be possible to see the values of the greenhouse in real time, as well as its set values. If the USER wants to open the door, this will be communicated to sysyem security, which will release it. After a while, the door will be closed again.

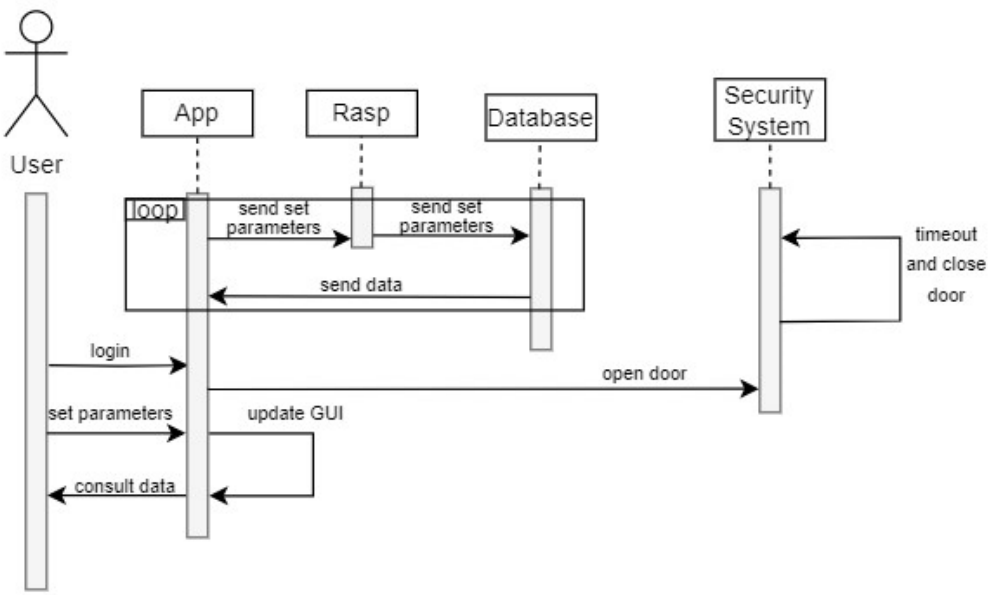

Figure 9: Interaction between user and interface sequence diagram

3.5 Database Diagram

Referring to the greenhouse project, figure 10 below shows a simple entity relationship diagram that depicts what will be stored in the database as well as its components. In this case, the database will have a user associated with a system, since the product to be developed is more intended for single-person use, this user will have a username and password provided by the greenhouse supplier.

As for the system, it will have the predefined values of the greenhouse parameters from the mobile application and the current values will also be stored in the database to be sent and monitored in the app.

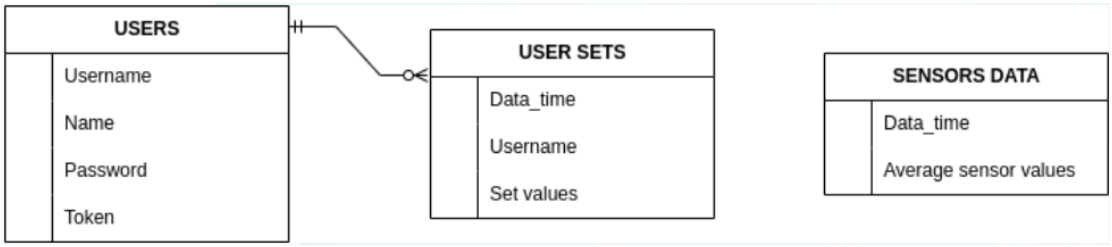

Figure 10: Entity Relationship Diagram

3.6 Technology Stack

The stack technology will begin at the hardware layer, which includes the Raspberry Pi, sensors, and actuators. Following this, there will be a communication layer, drivers for both sensors and actuators, an operating system, and application software for data processing and acquisition.

The top two layers consist of a database where the program's information will be stored and a mobile app for user interaction.

The following figures 11 & 12, is an illustration of this project technologies stacks.

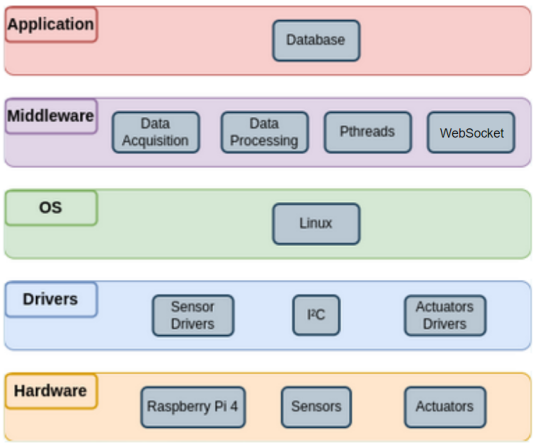

Figure 11: Local Technology Stack

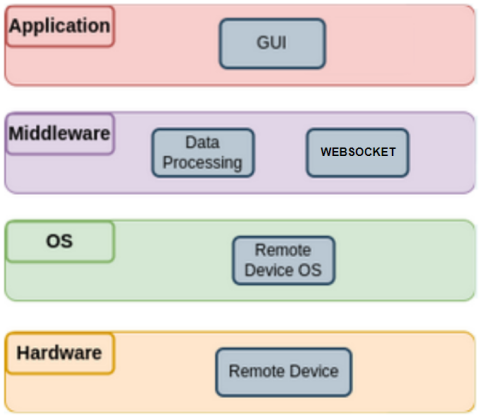

Figure 12: Remote Technology Stack

Chapter 4

Design

In the design phase its determined how the system will meet the requirements based on the constraints defined previously.

4.1 Analysis Review

In the previous chapter, a brief analysis was conducted on the developing product. Initially, requirements and constraints of the system were defined after conducting a thorough study.

4.1.1 Entity Relationship Diagram

In relation to the analysis phase, the entity relationship diagram was redone, with much more specification on the variables and attributes that the database will store, these being the different parameters of the greenhouse such as avgTemperature, avgHumidity, setLuminosity, among others.

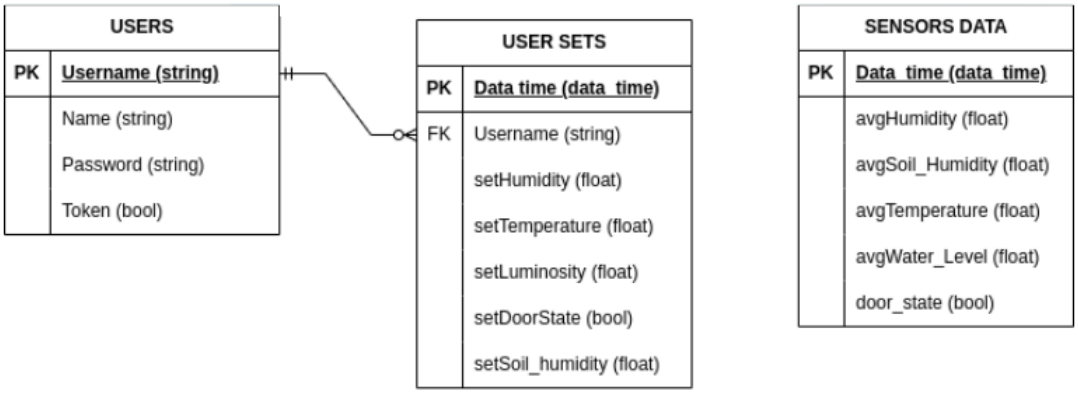

Figure 13: Entity Relationship Diagram

4.2 Hardware Specification

4.2.1 Development Board

The board used to develop this project will be the Raspberry Pi 4B, figure 14, which is one of the constraints of the project. This board includes a 64-bit quad-core ARM BCM2711 processor, as well as the following features.

Technical Specifications:

- 2GB LPDDR4-3200 SDRAM
- Raspberry Pi standard 40 pin GPIO header
- 2 USB 3.0 ports and 2 USB 2.0 ports
- 2 micro-HDMI ports
- 1 display port (2-lane MIPI DSI)
- 1 camera port (2-lane MIPI CSI)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- 1 jack 3,5mm port (4-pole stereo audio and composite video port)

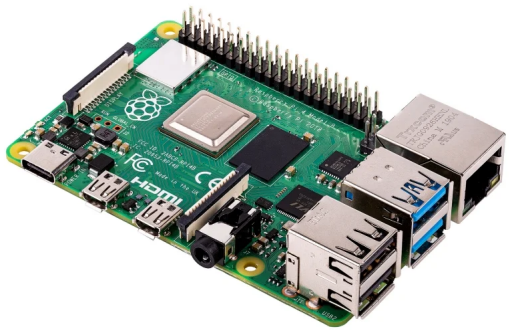

Figure 14: Raspberry Pi 4B
[3]

The board features 40 GPIO pins, visible in the figure 15, and enables connection to different external peripherals and technologies including UART, I2C, or SPI.

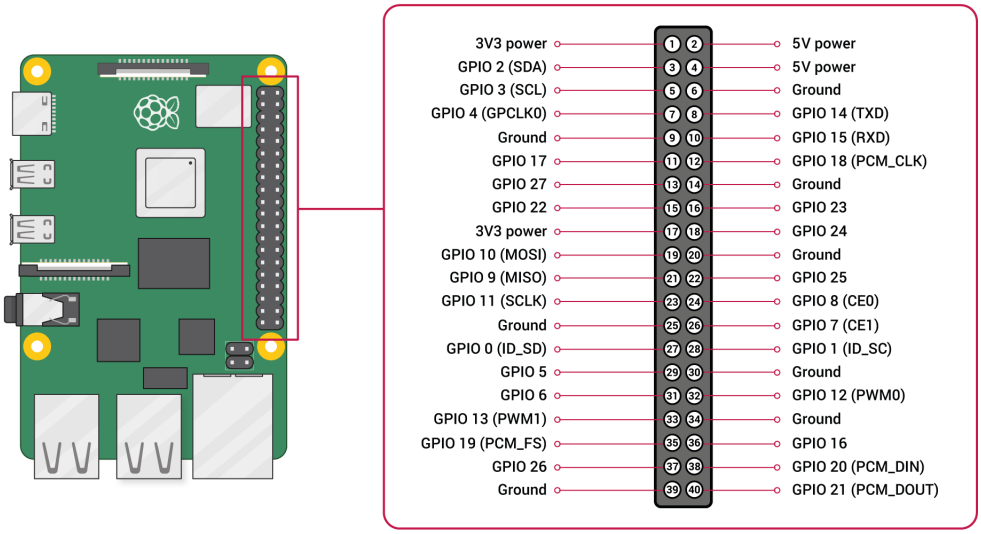

Figure 15: Raspberry Pi 4B Pinout
[4]

4.2.2 SD Card

SD memory card will be used to store the generated embedded Linux OS image. Kingston Canvas 32 GB, figure 16, was the chosen model with a capacity of 32 GB is anticipated to be sufficient to operate the system.

Figure 16: Kingston Canvas 32 GB SD Card
[5]

4.2.3 Sensors

The main aim of this project will be to control the various parameters of the greenhouse and to do this measurements is needed, which is why the automated greenhouse project will use various sensors.

Temperature and Humidity Sensor

Air temperature and humidity are critical parameters to monitor in a greenhouse. To obtain these values digitally, the Si2071 sensor will be used which communicates via I^2C.

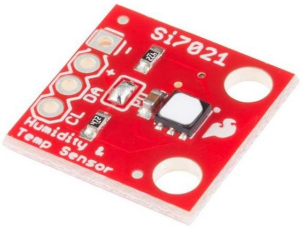

Figure 17: Temperature & Humidity Sensor
[6]

Technical Specifications:

- I^2C Interface
- Working Voltage: 1.9-3V
- Precision Relative Humidity Sensor: ± 3% RH (max), 0–80% RH
- High Accuracy Temperature Sensor: ±0.4 °C (max), –10 to 85 °C
- Low Power Consumption:150µA active current and 60ηA standby current

Pin Configuration

Si7021	**Board Pin**
VCC	3.3V
GND	GND
SDA	I^2C SDA Pin
SCL	I^2C SCL Pin

Table 2: Si7021 Interface

Test Cases

Test the behaviour of the Si7021 Temperature and Humidity sensor based on the input temperature and air-humidity levels.

Input	**Expected Output**	**Real Output**
Read Temperature	Value approximate to actual temperature	Read temperature was approximately as expected
Read Air-Humidity	Value approximate to actual humidity	Read humidity was approximately as expected

Table 3: Temperature and Air-Humidity Sensor Test cases

Soil Moisture Sensor

To determine whether the soil requires watering, it is necessary to acquire the percentage of soil moisture and assess this reading to calculate the required amount of water for optimal moisture content.

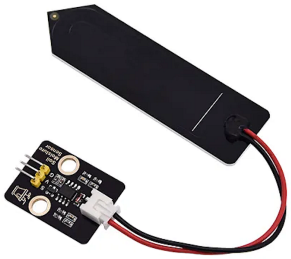

Figure 18: KS0510 Soil Moisture Sensor
[7]

Technical Specifications:

- Input Voltage: 3.3 - 5.5V
- Output Voltage: 0 - 3V
- Current: 5mA
- Dimensions:
  - Signal end: 31.6mm x 23.7mm
  - Detection end: 23.8mm x 83mm
- Operating temperature range: 1 - 50°C

Pin Configuration

KS0510	**Board Pin**
V	3.3V
G	GND
S	A0(ADC-module)

Table 4: KS0510 Interface

Test Cases

Test the behaviour of the KS0510 soil-humidity sensor based on the soil moisture measured.

Input	**Expected Output**	**Real Output**
Read Soil Moisture	Value approximate to actual soil humidity	Humidity soil reading changes depending on the water deposited

Table 5: Soil-Humidity Sensor Test cases

Water-Level Sensor

In order to read the value of the water level, the Grove Water Level Sensor will be used to measure the water level, thus obtaining the percentage of water in the tank.

Figure 19: Grove Water Level Sensor
[8]

Technical Specifications:

- Input Voltage: 3.3 - 5.5V
- Temperature Range: -40 °C - 105 °C
- Weight: 9.8g
- Precision: ± 5mm
- I^2C Interface
- Dimensions: 20mm x 133mm

Pin Configuration

Water Level Sensor	**Board Pin**
VCC	3.3V
GND	GND
SDA	I^2C SDA Pin
SCL	I^2C SCL Pin

Table 6: Water Level Sensor Interface

Test Cases

Test the behaviour of the Grove Water Level Sensor based on the tank.

Input	**Expected Output**	**Real Output**
Read water level	Value approximate to actual level	Sensor output was the right water level

Table 7: Grove Water Level Sensor Test cases

Light Sensor

In order to obtain the light levels, a Light-Dependent Resistor(LDR) based circuit will be used to evaluate the light level, which will serve as an indicator to switch the system's grow lights on or off.

Figure 20: Light Dependent Resistor(LDR)
[9]

Technical Specifications:

- Light resistance(10Lux): 10 - 20KΩ
- Dark resistance: 1MΩ
- Max voltage: 150V
- Max power: 100mW

Pin Configuration

LDR	**Board Pin**
Pin 1	A1(ADC-module)
Pin 2	GND

Table 8: LDR Interface

Test Cases

Test the behaviour of the LDR based on the input luminosity.

Input	**Expected Output**	**Real Output**
Read Luminosity	Value approximate to actual luminosity	Voltage values varies depending on the light

Table 9: LDR Test cases

4.2.4 Actuators

To regulate the greenhouse parameters, actuators must be used to manipulate the processed data and change the current parameters to meet the desired values.

Water Pump

In order to supply the greenhouse with water, an R385 DC Diaphragm Water Pump, figure 21, will be used, which can be used with water up to temperatures of around 80°C.

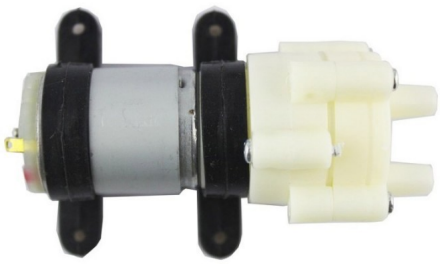

Figure 21: R385 Water Pump
[10]

Technical Specifications:

- Voltage: 12V
- Current: 0.5-0.7A
- Max slope(Head): 3m
- Water flow(Traffic): 1.5-2L/min
- Temperature threshold: 80°C
- Work life: up to 2500h

Pin Configuration

Water Pump	**Relay Pin**
VCC	Relay1
GND	GND

Table 10: R385 Interface

Test Cases
Test the behaviour of the R385 water pump based on the input voltage.

Input	Expected Output	Real Output
12V	Water Pump On	Water Flowing through the pump
0V	Water Pump Off	Water not flowing through the pump

Table 11: Water Pump Test cases

Electric Fan

Fans will circulate air in the greenhouse, which can be useful in effectively reducing the temperature.

Figure 22: Electric Fan
[11]

Technical Specifications:

- Working Voltage: 6-13.8V
- Nominal Current: 0,074A
- Power Consumption: 0.89W
- Fan Speed: 2600rpm ±10%
- Dimensions: 80 x 80 x 20 mm
- Noise Level: 30dB(A)
- Weight: 66g

Pin Configuration

Electric Fan	Relay Pin
VCC	Relay2
GND	GND

Table 12: Electric Fan Interface

Test Cases
Test the behaviour of the electric fan based on the input voltage.

Input	Expected Output	Real Output
12V	Fan Working	The fan worked properly
0V	Fan Shutdown	The fan stopped

Table 13: Fan Test cases

Heating Resistor

In order to control the temperature of the greenhouse, warming it up when necessary is essential. For this purpose, the chosen heating element illustrated in the figure 23 will be used.

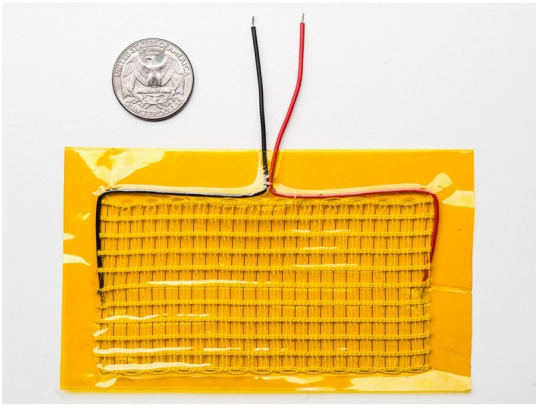

Figure 23: Heating Resistor
[12]

Technical Specifications:

- Working Voltage: 5-12V
- Dimensions: 110.65mm / 4.35” x 70.45mm / 2.77” x 1.54mm / 0.06”
- Wire Length: 35.8mm / 1.4”
- Weight: 2.53g
- Insulation: Polyimide film (’Kapton’)
- Fabric made of Polyester filament and micro stainless steel fiber

Pin Configuration

Heating Resistor	Relay Pin
VCC	Relay3
GND	GND

Table 14: Heating Resistor Interface

Test Cases

Test the behaviour of the heating resistor based on the input voltage.

Input	Expected Output	Real Output
12V	Heating Turned On	The Heating Resistor started to heat up
0V	Heating Turned Off	The Heating Resistor didn't heat up

Table 15: Heating Resistor Test cases

Lighting

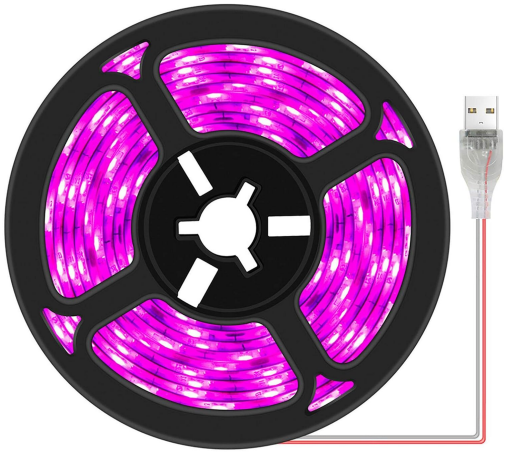

Figure 24: Grow Lights
[13]

Technical Specifications:

- Working Voltage: 5V
- Dimensions: 16 x 13 x 1 cm
- Water Resistant

Pin Configuration

Grow Lights	Relay Pin
VCC	Relay4
GND	GND

Table 16: Lighting Interface

Test Cases
Test the behaviour of the Grow Lights based on the input voltage.

Input	Expected Output	Real Output
12V	Lights Turned On	The lights came on
0V	Lights Turned Off	The lights stayed off

Table 17: Grow Lights Test Cases

Water Sprinkler(Humidifier)

In addition to a second R385 water pump, water nebulisers will be used to induce humidity in the greenhouse. Regarding pin configurations and test cases, it is identical to the R385 water pump. The only distinction is the intended function of the sprinkler for air-humidity insertion.

Figure 25: Water Sprinkler
[14]

Electromagnet

To secure the greenhouse, an access door controlled by an electromagnet will be installed which, after receiving a signal from the app, will grant or deny access to the interior of the greenhouse, thus protecting the user's crops.

Figure 26: Electromagnet
[15]

Technical Specifications:

- Rated Voltage: 5V
- Rated Power: 1W
- Holding Force: 3kg
- Wire Lenght: 28cm

Pin Configuration

Electromagnet	**Relay Pin**
VCC	Relay5
GND	GND

Table 18: Electromagnet Interface

Test Cases
Test the behaviour of the Electromagnet based on the input voltage.

Input	**Expected Output**	**Real Output**
12V	Magnetism On	A magnet was drawn
0V	Magnetism Off	No reaction from the magnet

Table 19: Electromagnet Test Cases

Relay Module

In order to control the different actuators and supply the necessary current and voltage, a module of 8 5V-relays will be used.

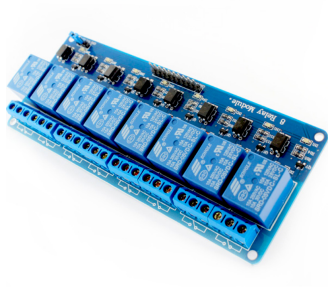

Figure 27: 8-channel 5V Relay Module
[16]

Technical Specifications:

- Driver Current: 15-20mA
- Equipped with high-current relay: AC250V 10A - DC30V 10A
- Indication LED for Relay output status
- Size: 13.80 x 5.50 x 1.70cm

Test Cases
Test the behaviour of the Relay Module based on the input control voltage.

Input	**Expected Output**	**Real Output**
IN1 High	Current flow on normally-open Pin1	The relay docked and there is current flowing
IN2 High	Current flow on normally-open Pin2	The relay docked and there is current flowing
IN3 High	Current flow on normally-open Pin3	The relay docked and there is current flowing
IN4 High	Current flow on normally-open Pin4	The relay docked and there is current flowing
IN5 High	Current flow on normally-open Pin5	The relay docked and there is current flowing
IN6 High	No current flow on normally-closed Pin6	The relay disengaged and current stopped flowing

Table 20: Relay Module Test Cases

Pin Configuration

Relay Pin	Board Pin	Output Pinout
VCC	5V	-
IN1	GPIO23	-
IN2	GPIO6	-
IN3	GPIO13	-
IN4	GPIO5	-
IN5	GPIO19	-
IN6	GPIO25	-
NO1	-	Water Pump
NO2	-	Electric Fan
NO3	-	Heating Resistor
NO4	-	Grow Lights
NO5	-	Water Sprinkler
NC6	-	Electromagnet
GND	GND	GND

Table 21: Relay Module Interface

4.2.5 Power Supply

In order to power the entire circuit, it was decided to separate it into two control sources, one for powering the Raspberry Pi and the other for the actuators, as they will need more current.

Raspberry Power Supply

Figure 28: Raspberry Power Supply
[17]

Technical Specifications:

- Output Voltage: 5.1V
- Maximum power: 15.3 W
- Voltage range: 100 - 240 VAC(rated) / 96 - 264 VAC(operating)

Actuators Power Supply

Figure 29: Actuators Power Supply
[18]

Technical Specifications:

- Output Voltage: 12V
- Nominal Power: 60W
- Weight: 0.165 kg
- Dimensions: 36x79x110 mm

4.2.6 Hardware Connection

The hardware connections can be divided into two groups, inputs and outputs. The inputs use an ADC module that uses the I^2C protocol to communicate with the raspberry. This ADC is responsible for converting the analogue values of the S0510 soil moisture sensor and the LDR that evaluates the intensity of the light incident on the greenhouse. The Si7021 temperature and humidity sensor will also be connected to the same I^2C pins (GPIO 2/3) as the ADC module, as it will be necessary to specify the different addresses for data interpretation. The Grove water level sensor will also be used to measure the water level in the tank.

The system's outputs will all be connected to a relay module that will control the activation of devices such as fans, heating resistors, electromagnets, etc. as well as supply the required power for this devices.

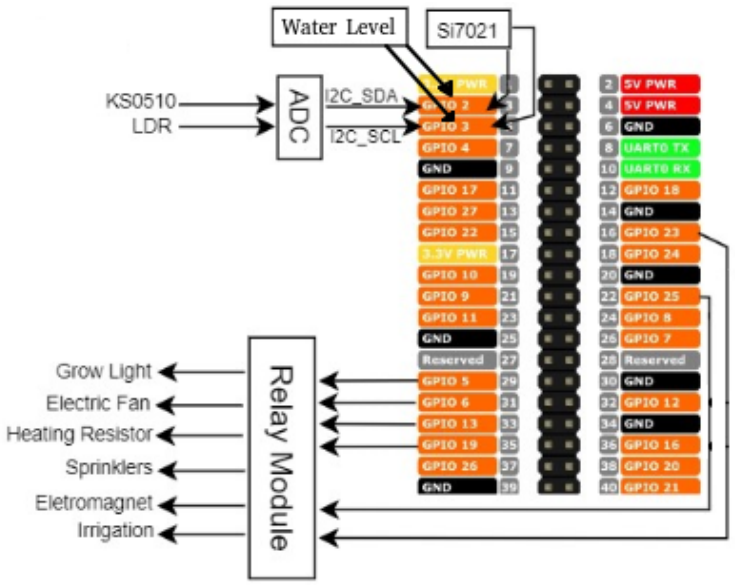

Figure 30: Hardware Connection

Actuator	Connection	Pinout
Fan	Relay	GPIO6
Light	Relay	GPIO5
Heating Resistor	Relay	GPIO13
Sprinklers	Relay	GPIO19
Electromagnetic Door	Relay	GPIO25
Irrigation System	Relay	GPIO23

Table 22: Actuators Hardware Connection

Sensor	Connection	Pinout
—	VCC	Pin2
—	GND	Pin9
Si7021	I²C(SCL)	GPIO3(I²C1 SCL)
Si7021	I²C(SDA)	GPIO2(I²C1 SDA)
LDR	—	ADC(A0)
Water Level	I²C(SDA)	GPIO2(I²C1 SDA)
Water Level	I²C(SCL)	GPIO2(I²C1 SCL)
KS0510	—	ADC(A1)
ADC	LDR	A0
ADC	KS0510	A1
ADC	ADDR	GPIO26
ADC	I²C(SCL)	GPIO3(I²C1 SCL)
ADC	I²C(SDA)	GPIO2(I²C1 SDA)

Table 23: Sensors Hardware Connection

4.3 Tools and COTS

4.3.1 Tools

- **GitHub** - Will be used for version control and collaborative development, enabling both group members to track changes to the project's code.

Figure 31: GitHub logo

- **Draw.io** - Used to create various diagrams, such as overview diagrams and flowcharts, or to show the connections between project parts.

Figure 32: Draw.io logo

- **Overleaf** - Will be used to create and edit the report throughout the process. It has the advantage of being a cloud-based service and is very useful when it comes to co-operative work, templates and document formatting.

Figure 33: Overleaf logo

- **Visual Studio Code** - VS Code will be used due to its versatility and user-friendly interface, as well as its ability to easily connect to GitHub and the wide range of extensions available. This makes it a potent tool for developing and managing software components.

Figure 34: Visual Studio Code logo

- **Buildroot** - Will be used for configuration and generation of the Raspberry Pi kernel image.

Figure 35: Buildroot logo

- **I²C** - The Inter-Integrated Circuit (I2C) protocol is widely used to link numerous devices on a shared bus. It will be used with the Raspberry Pi 4 development board for communication with sensors and other peripherals.

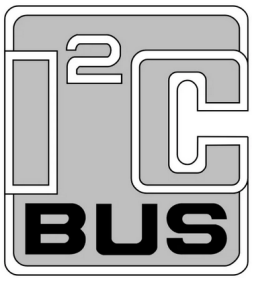

Figure 36: I²C Logo

4.3.2 COTS

- **POSIX Threads** - Set of standard API's (Application Programming Interface) on Unix and Unix-like systems that enables creation and manipulation of concurrent threads. These threads can execute different parts of the code in parallel, enhancing performance.

Figure 37: Pthreads Logo

- **SQLite** - Database management system used to store the parameters, user and system information.

Figure 38: SQLite Logo

- **Node.js** - JavaScript runtime environment used with the express framework for developing a server between remote and local system.

Figure 39: Node.js Logo

4.4 Software Specification

4.4.1 Main System

The figure 40 below shows our system as a whole and its interactions. On the left is the remote system, where the APP represents the interface for interaction with the user and the server acts as an intermediary between the user and the local system.

As for the local system on the right, it will consist of two processes, one that will be the main process and a second that will interface with the server. This way, as the values are read by the sensors, they will be stored in the database and, when necessary, they will make the necessary route to the user interface, being sent from the first to the second via a message queue and then sent to the server via a WebSocket, so the user will be able to access the greenhouse's values in real time.

When the user chooses new parameters for their greenhouse, they will take the opposite route to the one described above, first they will be stored on the server, where they will be sent to the local system, more precisely to the "Communication System", via a WebSocket. Once this process has received the data, it will send it to the main process via a message queue so that it can be stored in the database to be used when necessary, such as when starting up the system.

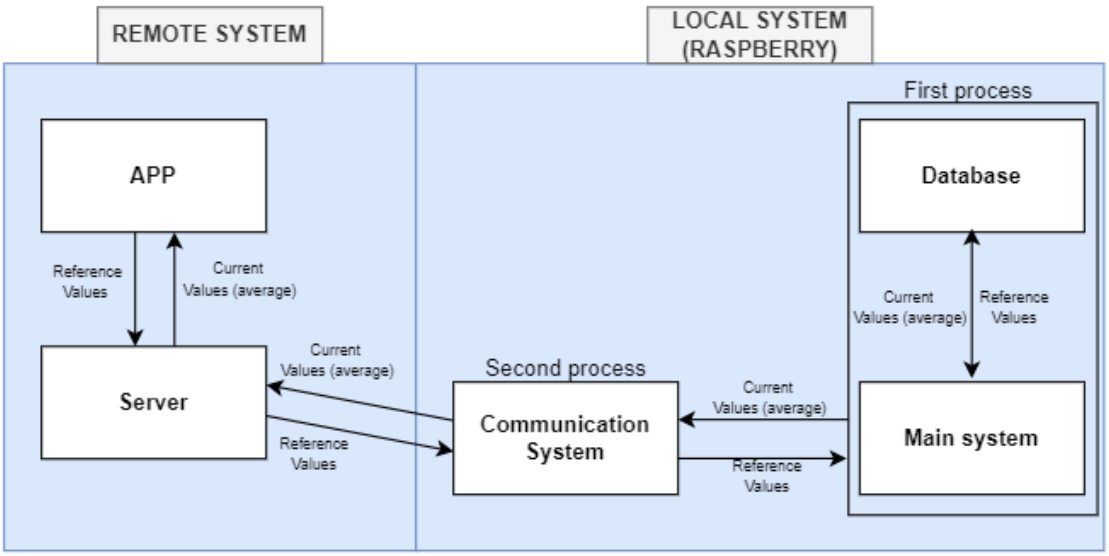

Figure 40: Main System Overview

4.4.2 Local Software System

Class Diagrams

Actuators Class

A class has been designed to oversee the actuators, as they perform identical functions. There will be functions to set the actuator's state and to manage the exit door, in addition to the constructor and destructor. Instances will be generated for each actuator in the greenhouse. These will be instantiated with a name and a state as parameters.

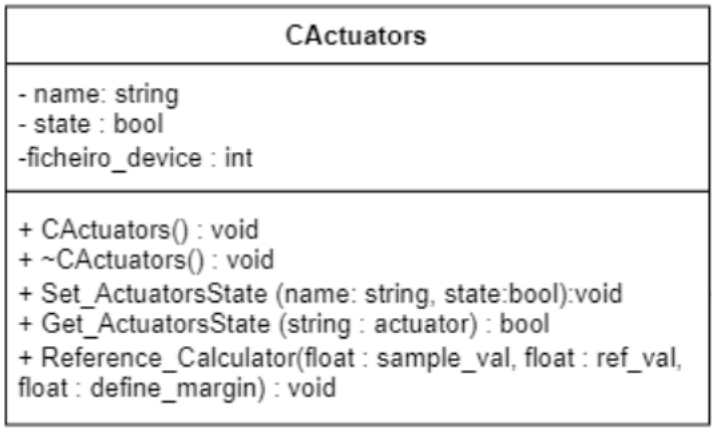

Figure 41: Actuators Class

Sensors Class

Each sensor is assigned to a class according to its unique method of operation. The variables that each sensor measures will correspond to the variables of its of its assigned class. In addition to the constructor and destructor, each class will also have functions will allow reading and manipulation of the sensors.

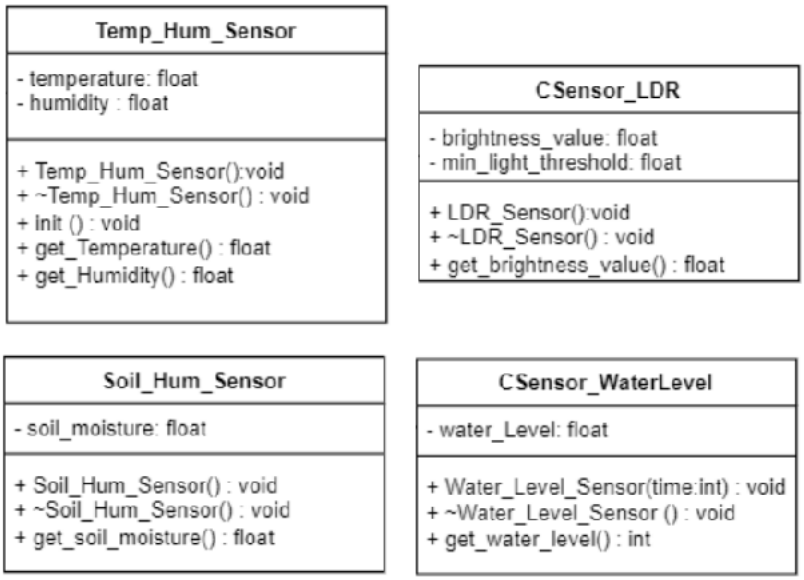

Figure 42: Sensors Classes

Database Class

This class connects the local and remote systems. In addition to the constructor and destructor, there will be other functions such as "InsertData()" which will store average values read from the sensors. StoreParameters()" will be used to store the desired parameters for the greenhouse from the app. There will also be functions to return both the user's credentials and to send the current values to the application. The attribute "The Databasepath" indicates the path of the file or the location of the directory where the database file is stored, and is used to access and manage the data in the database.

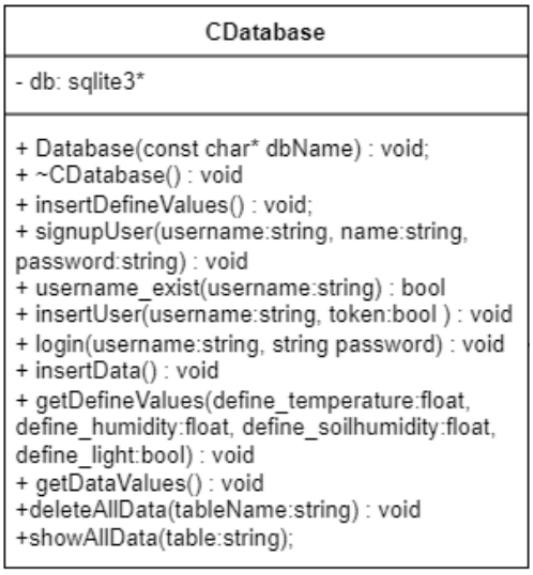

Figure 43: Database Class

Local System Class

Contains all required variables for the program. The "Data processing()" function will act as an intermediary between the application and the database. It will store the greenhouse parameters and later store them in the database.

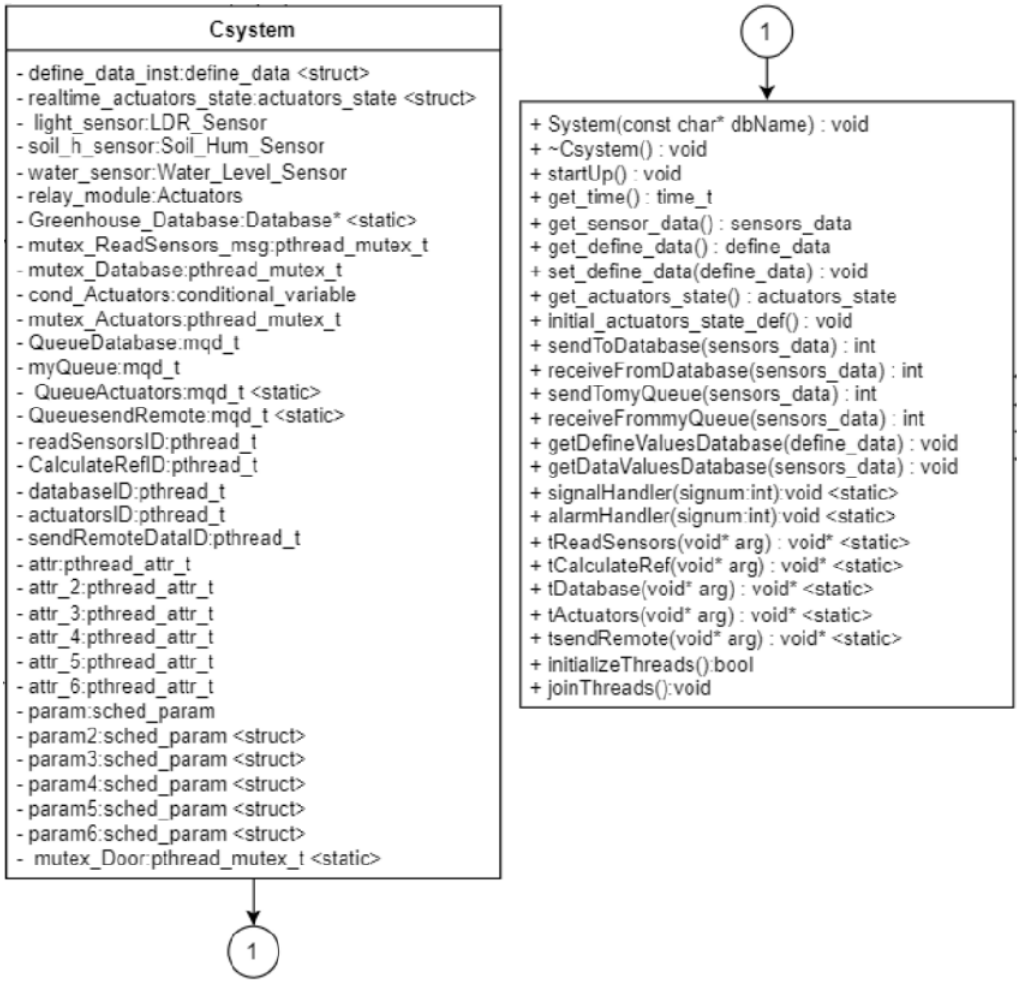

Figure 44: Local System Class

Overall Class Diagram

The figure 45 below shows all the classes and their interactions within the system. The sensor and actuator classes are integrated into the MainSystem class through composition, which means that the MainSystem class contains objects of the sensor and actuator classes as its members. This composition-based approach enables the MainSystem to efficiently obtain and manipulate sensor data and communicate with the actuators.

The MainSystem class and the Database class have an aggregation relationship because the Database is used by the MainSystem for storing data, but it can remain independent of the MainSystem. This grants greater flexibility in the system's structure. The relationship between the user class and the database follows the same logic as between the main system and the sensor class.

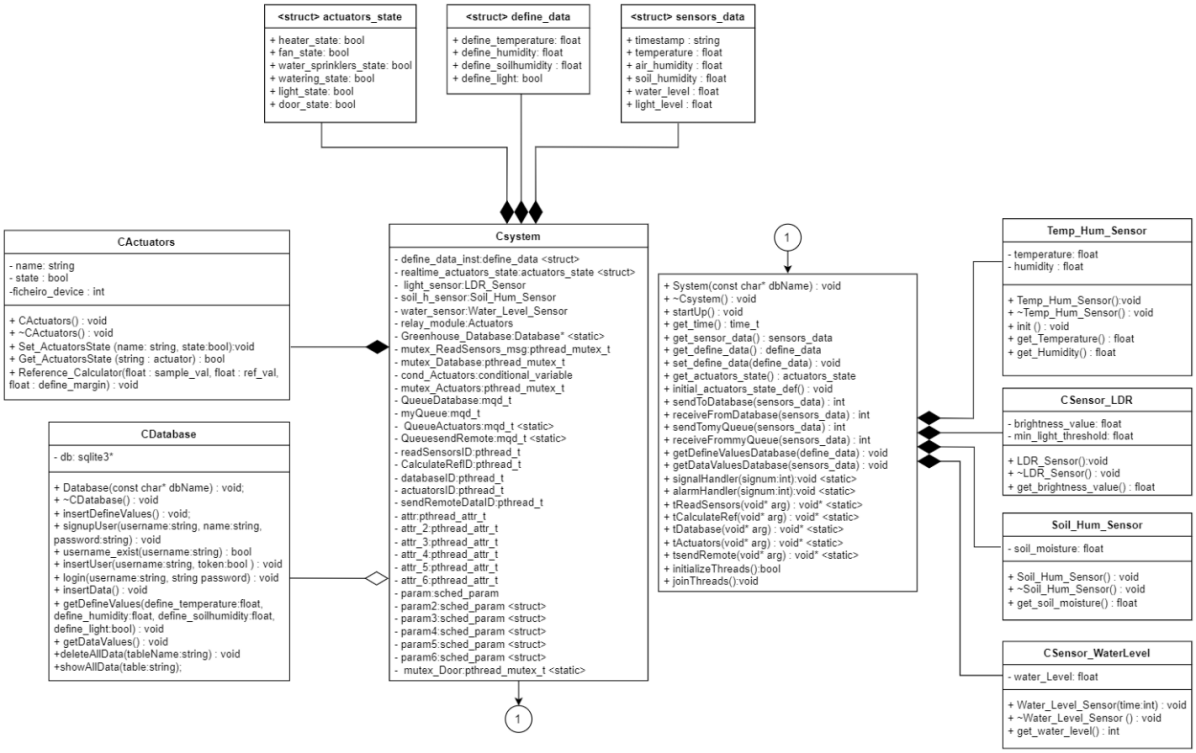

Figure 45: System Class Diagram

Thread Overview

The main process is composed by the following threads:

- tReadSensors: the thread involves reading the sensor and storing the value in a variable
- tActuators: activate the necessary actuators
- tDatabase: it will control the input and output of information from the database
- tCalculateRef: calculate the difference the wanted values and the current values of the greenhouse
- tSendRemoteData: responsible for sending data to the second process (Communication Process)

The second process or communication process is composed by:

- tReceiveFromLocal: this thread will be responsible for getting the data sent from the local system via message queue and forward it to the server via WebSocket
- tReceiveFromServer: it will receive data from the server via WebSocket, process it and send the filtered data to the local system

Start-up/Shutdown Thread:
During the start-up process, the program will initialize the necessary components such as drivers, sensors, actuators, and synchronization objects. Additionally, all threads will be created. In the shutdown process, all threads, as well as the drivers and synchronization objects, will be destroyed.

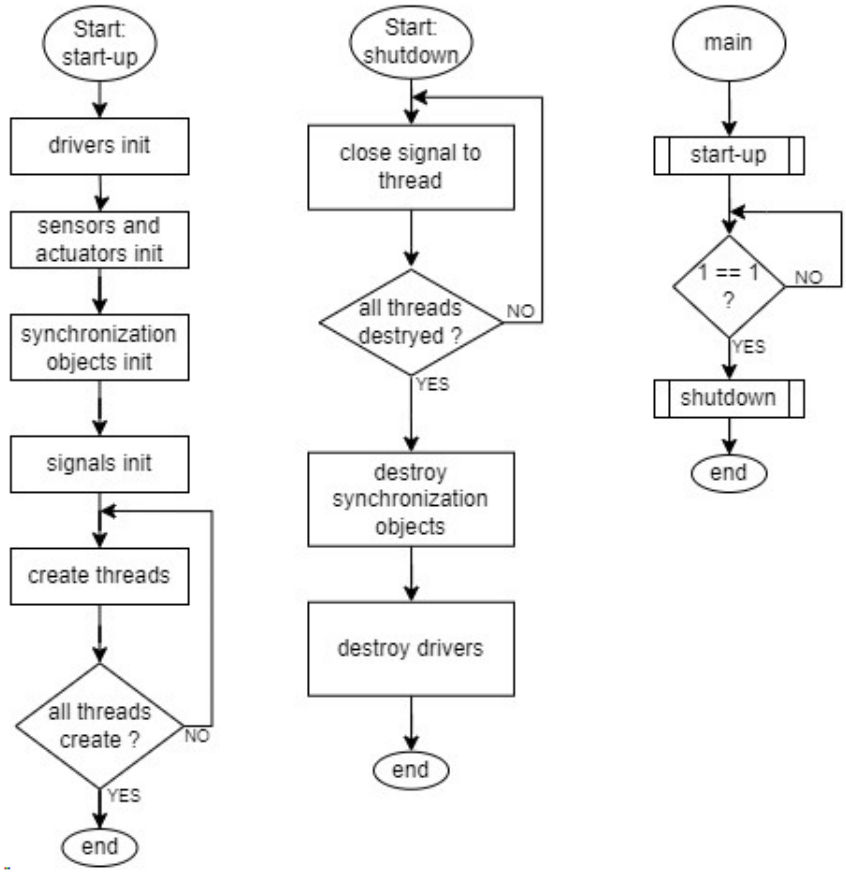

Figure 46: Main and Start-up/Shutdown Flowchart

Read Sensors Thread:
In this thread, the sensor values shall be stored before being dispatched to a message queue for storage in the database.

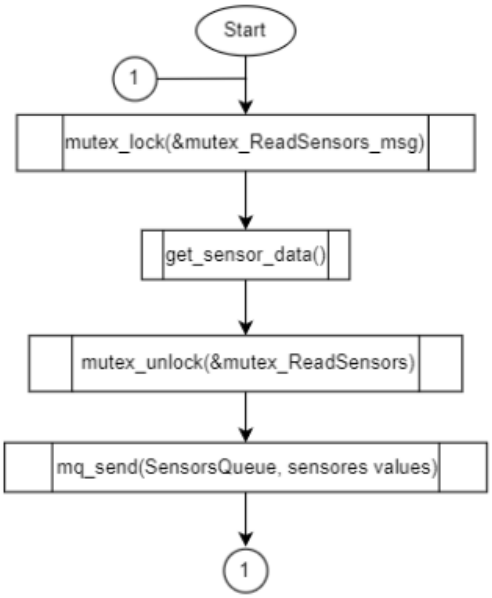

Figure 47: tReadSensors Flowchart

Actuators Thread:
This thread will control the actuators. It will first await the conditional variable, followed by reading values from the message queue. The bias will be calculated, and based on the result, the state of the actuators will change if necessary.

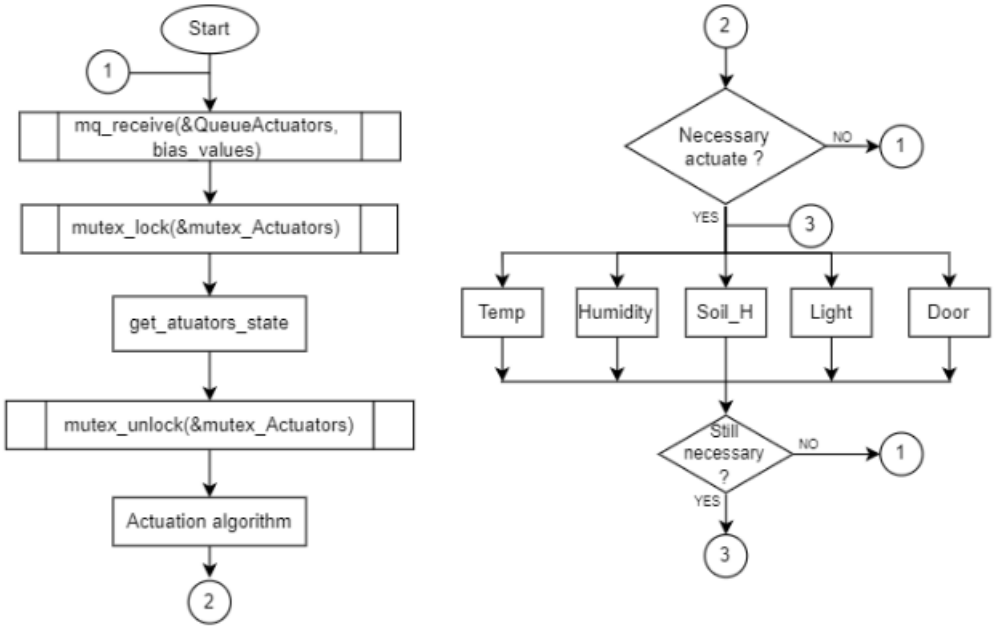

Figure 48: tActuators Flowchart

Calculate Reference Thread:
This thread is responsible for calculating the average values. Firstly, it reads sensor values from the message queue and stores them in an array. If the array becomes full, the average value is calculated and sent to the database. However, if the array is not full, the thread remains inactive and waits for new values from the sensors.

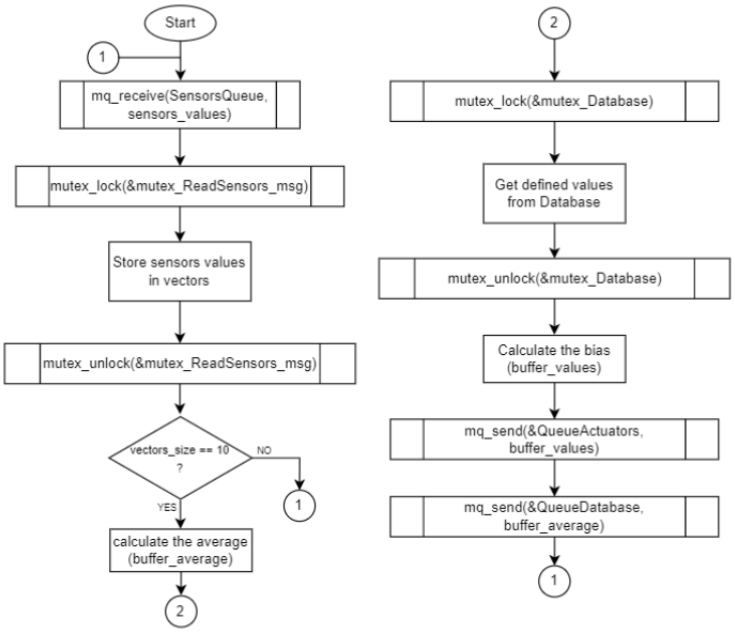

Figure 49: tCalculateRef Flowchart

Database Thread:
This thread is responsible for storing program data in the database. Initially, it will check if the message queues storing sensor values are empty. If they are, it will then examine the message queues that store references. Any queues containing values will be saved.

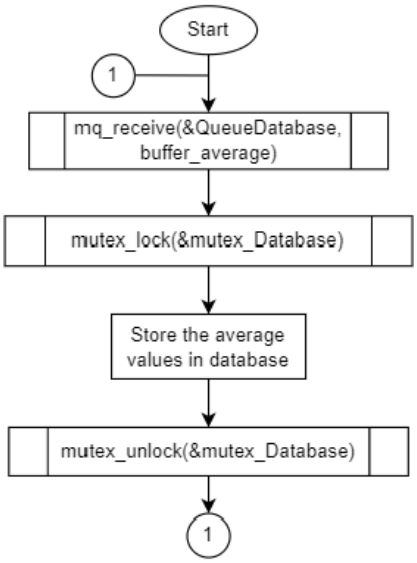

Figure 50: tDatabase Flowchart

Send Remote Data Thread:

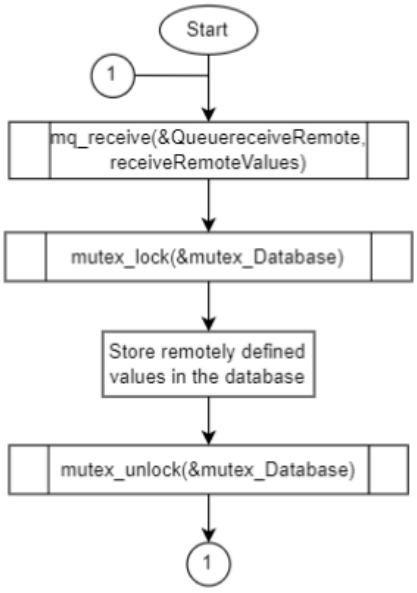

Figure 51: tSendRemoteData Flowchart

Task Interaction:

Image 52 below illustrates the relationship between the different threads in the program and the communication between each other. When the sensors are read, the values are stored in a message queue, after which an average value is calculated for the parameters. These parameters will be stored in the database and used to check whether or not the actuators need to be activated by comparing the average values with the reference values chosen by the user.

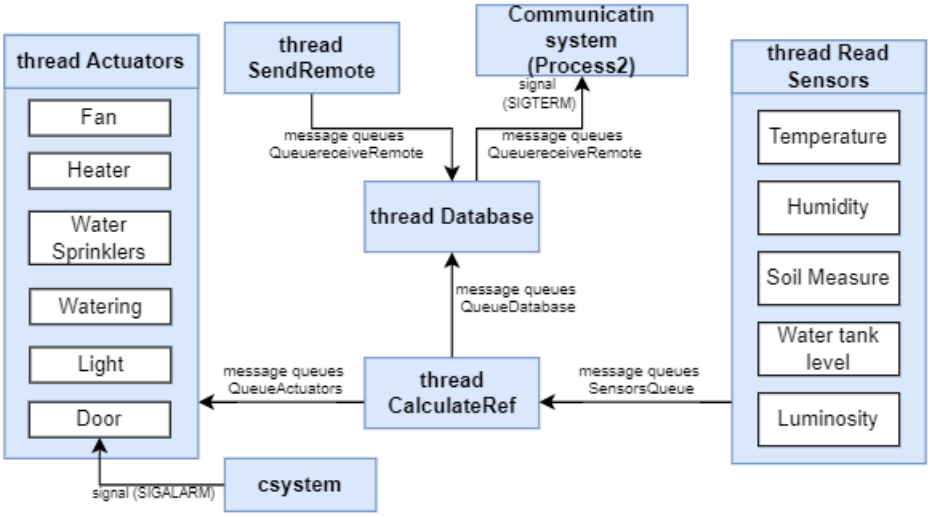

Figure 52: Task Interactions

Thread Priority

The priority of the threads is crucial to optimizing the system's performance.

The sensor and actuator threads were given the highest priority, as they are responsible for collecting data and changing the actuator's state in real time, which are high-priority tasks.

The next thread calculates the polarization, which is important for keeping an up-to-date record of the greenhouse variables and ensuring that the system works with the latest values.

The database will have an intermediate thread, between 'calculateRef' and the thread that sends the data to the remote system.

Finally, the thread with the lowest priority will be the one responsible for sending the data to the remote system, as there is greater tolerance and margin in the sending times.

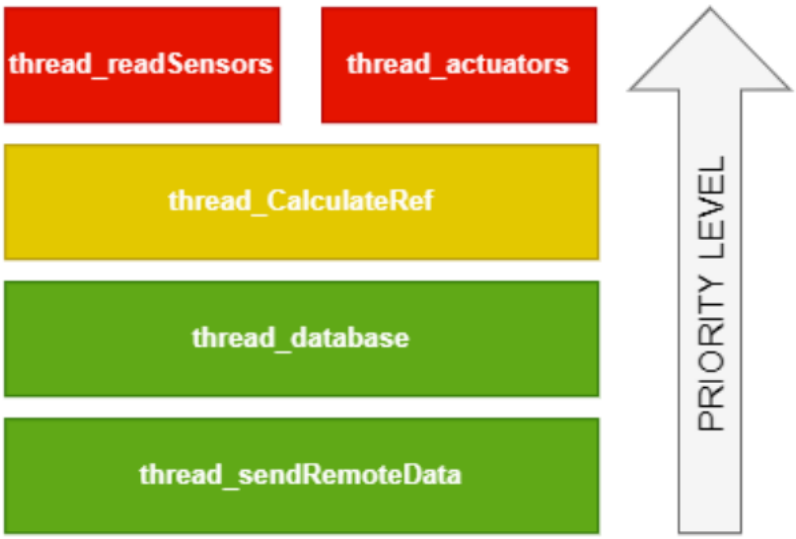

Figure 53: Thread Priority

Task Synchronization

Message Queues:
A message queue is a list of messages chained together that is used to communicate between processes or threads.
Three message queues will be used to change information between threads and process:

- SensorsQueue: It will store the values read from the sensors and send them to the 'calculateRef' thread, where the average value and the bias to the defined parameters will be calculated.
- QueueDatabase: This message queue will be used to retrieve the average values calculated by the 'calculateRef' thread and store them in the database.
- QueueActuators: After calculating the bias, the message queues will be used to send the results to the actuators thread.
- QueuesendRemote: This queue will retrieve the most recent values stored in the database and transmit them to the remote system.
- QueuereceiveRemote: Receives the intended parameters to the greenhouse from the remote system.

Mutexes:
Mutexes are tools used to control concurrency and prevent threads from interfering with each other in a harmful manner. They are created in an unblocked state and become blocked when a task acquires them. Once released, they return to an unblocked state.

This project is going to use the following Mutexes:

- mutex_ReadSensors_msg: To ensure the security of the data obtained from sensors and stored in a struct for future use.
- mutex_Actuators: Like the sensor mutex, this serves to protect access to the real-time states of the actuators, which are stored in a struct.
- mutex_Database: To prevent race conditions when accessing the database, simultaneous read and write movements must be protected

4.4.3 Remote Software System

GUI

Upon accessing the application, the initial step is to input the user's login credentials. In the subsequent window, the user is offered a choice between four environment options. The first three, Tropical, Arid and Regular, already contain preset configurations which become visible to the user upon selection. The fourth option, labelled 'Custom', gives the user ability to adjust factors such as temperature, humidity, ground level, and presence or absence of light to their needs. Selecting "OK" will choose the environment and set the corresponding parameters for the greenhouse if these are between the accepted values. The "View data" option displays the current values for temperature, humidity, soil moisture, water tank level, and light presence.

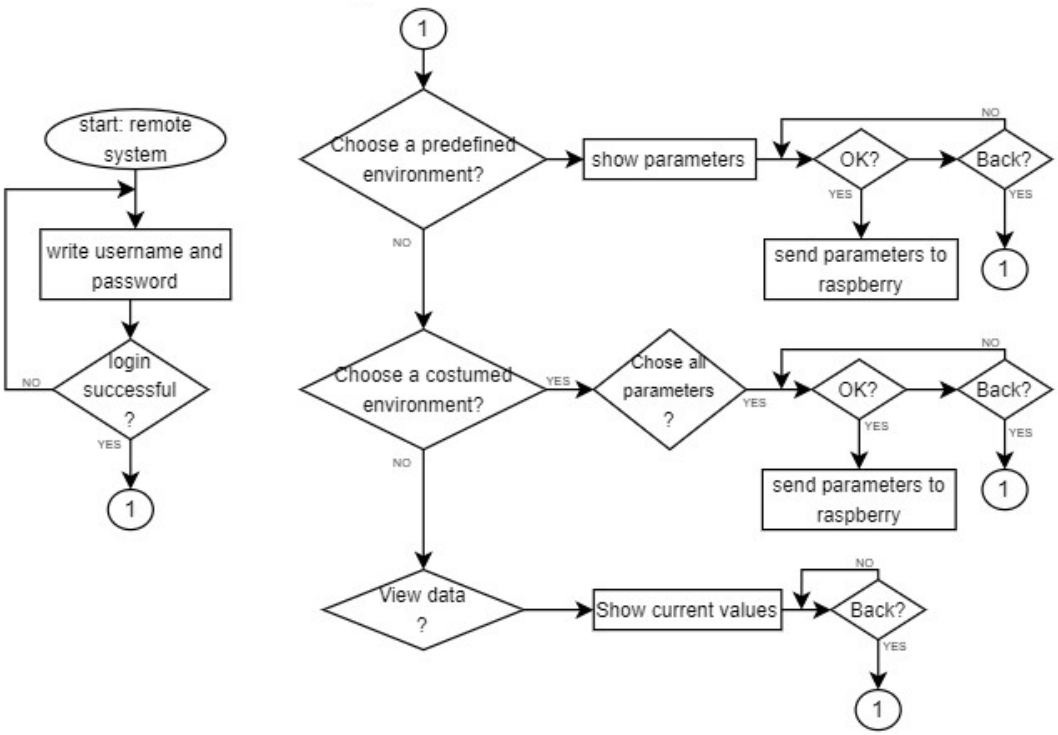

Figure 54: Remote System Flowchart

GUI Services & Layout

The GUI for this project will provide some services such as authentication by login, setting different parameters such as temperature, humidity, among others. These parameters can be predefined or customisable according to the user's preference. It will also have the ability to view the various parameters in real-time as well as a record of these different values.

On this first page, illustrated by Figure 55, it will be possible to log in to the User App, the credentials will be provided by the supplier, this solution was based on models similar to routers and internet providers where the credentials are also provided by them.

In the Figure 56 below, its possible to see the main page of the user app where the user can define the parameters for the greenhouse and see a recent record of these metrics. The user will also be able to choose from 3 different predefined environments such as tropical, arid and regular. It is also possible for the user to define their own parameters in the custom section. These parameters can only be defined within a certain range and the user will be responsible for them, with a warning appearing on the GUI alerting them or even forbidding the selection of values outside the allowed range.

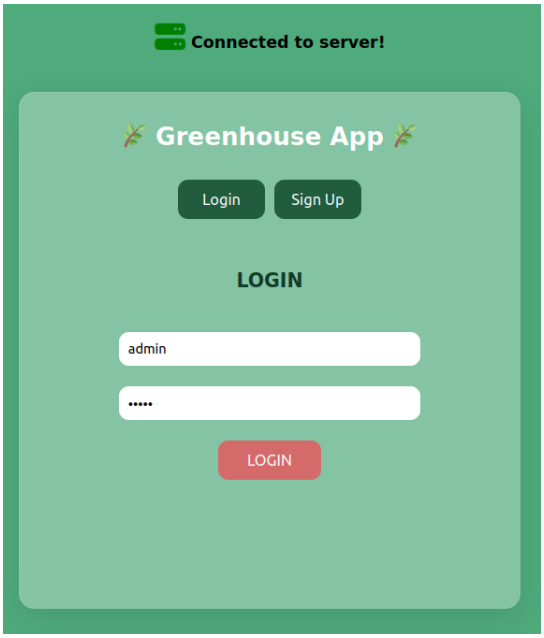

Figure 55: GUI Login Page

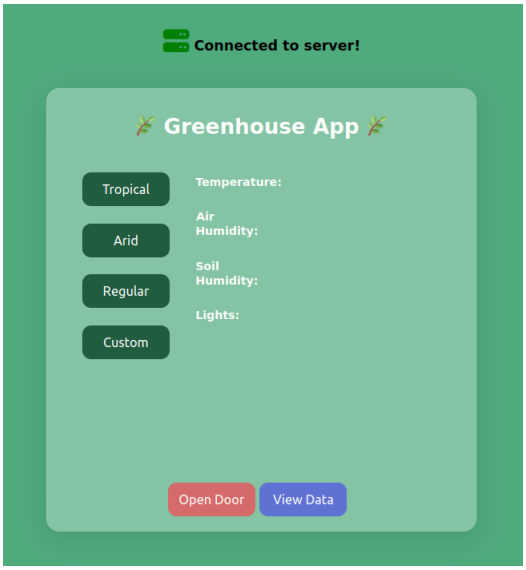

Figure 56: GUI Main page

In the following Figure 57 below, you can see the greenhouse configuration using the "Regular" environment, which is the default preset environment in which it is balanced in all parameters from temperature to soil humidity with the option of having the light switched on or off after reaching the minimum sunlight level.

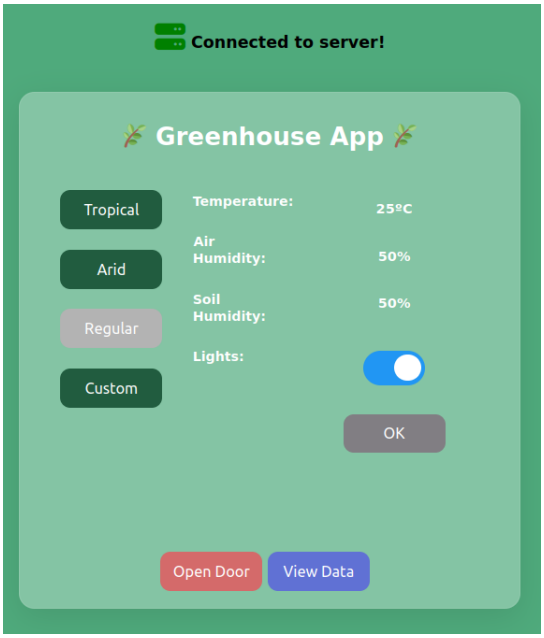

Figure 57: Regular Environment Selection

Figure 58 shows the customised environment section, in which the user can create their own environment with the desired parameter values within the allowed range, as well as the possibility of having the lights switched on or off after the minimum level of sunlight has been reached. Taking into account the allowed range of values, the user is alerted with a warning of the responsibility of changing the parameters in which the crops can be affected, as well as forbidding selection of values outside the allowed range.

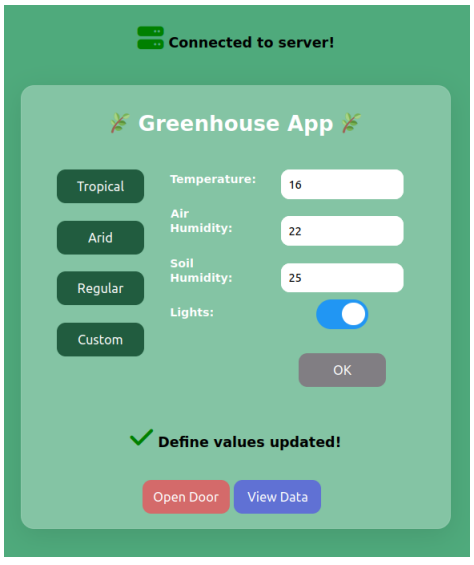

Figure 58: Custom Environment Selection

Finally, you can see in figures 62 & 63, the area that allows the user to see the record of previous values in which recent samples of the parameters values stored in the database.

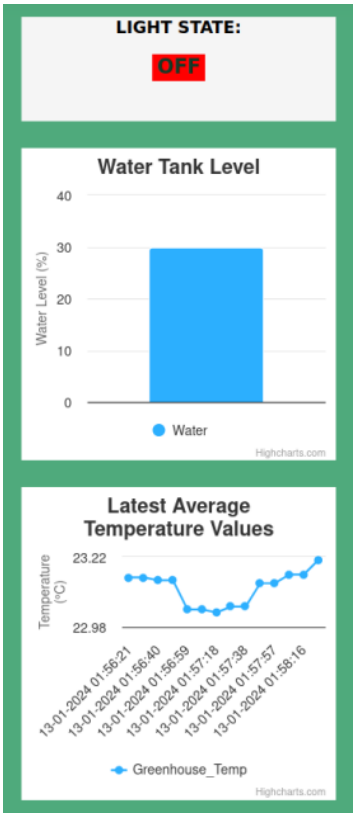

Figure 59: View Data Section 1

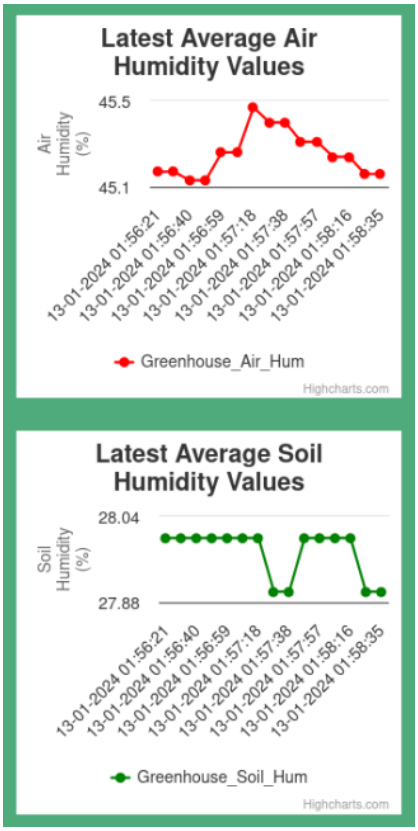

Figure 60: View Data Section 2

4.4.4 Test Cases

It is important to know if the system will work as it was expected and to see that will be done various tests. The tests are divided into three categories: hardware tests, which will test each peripheral individually (described next to the hardware components), local and remote test cases which each system functions and the attempt to put them all together will be tested.

Local System Test Cases:

Test	**Expected Result**	**Real Result**
Read temperature	Read approx. temperature value	The expected
Read humidity	Read approx. humidity value	The expected
Read soil moisture	Read approx. soil moisture value	The expected
Read the water tank level	Read water level of the tank	The expected
Read the light level	Read if light is sufficient	The expected
Send message to queue	Message written correctly	The expected
Read message from queue	Message read correctly	The expected

Table 24: Local System Test Cases

Remote System Test Cases:

Test	Expected Result	Real Result
Login into account	Retrieve login with credentials from database and grant/deny access	The expected
Choose environment	Be able to send data from the app environment section to raspberry	The expected
Create new environment	Be able to create user's own environment with desired parameters	The expected
View data	Successfully show the current values of the greenhouse with the data alocated in the database	The expected

Table 25: Remote System Test Cases

Database Test Cases:

Test	Expected Result	Real Result
Create table	Successfully create a table	The expected
Write data into the table	Successfully write into the table	The expected
Read data from the table	Successfully read from table	The expected
Open database	Successfully open database	The expected
Close database	Successfully close database	The expected
Delete database	Successfully delete database	The expected
Delete data from the table	Successfully delete data from table	The expected

Table 26: Database Test Cases

Dry Run

In order to analyse the behaviour of the system's core, a dry run was carried out with the inputs sensed in the greenhouse and the outputs that would be activated depending on the reaction to the system's inputs, such as the temperature above and below the defined margin, the light level below the preset level and the unlocking of the security door, among others.

Temperature below margin ?	Temperature above margin ?	Humidity below margin ?	Humidity above margin ?	Soil Humidity below margin ?	Light level below set value ?	Remote door unlock ?	Water pump	Electric Fan	Heating resistor	Lights	Water Sprinkler	Eletromagnet
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	1	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	1	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	1	1	1	1	0	0	1	0	1
0	1	0	1	0	0	0	0	1	0	0	0	0

Figure 61: System Core Dry Run

Error Handling

In terms of error handling, the following prevention and safety methods have been defined for the correct operation of the greenhouse.

- Temperature above MAX VALUE (35 °C):
  - Alert is sent to the User App
  - Heating resistor is deactivated
  - Electric fan and water sprinklers turn on for cooling effects
- Connection Lost:
  - Try to restore connection with the server
  - Alert sent to User App after disconnecting

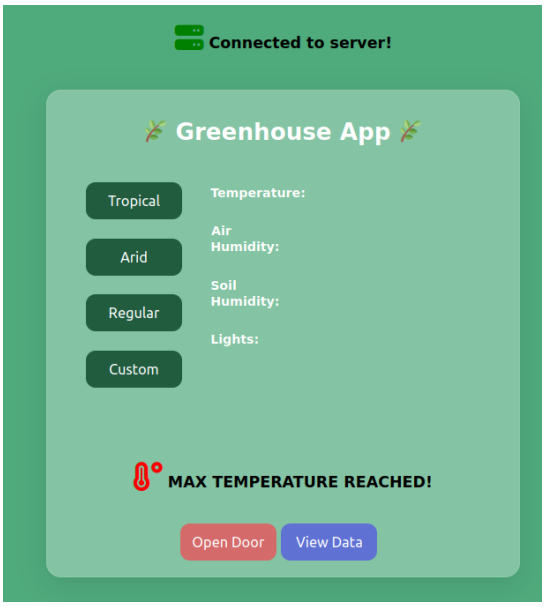

Figure 62: Max Temperature Warning

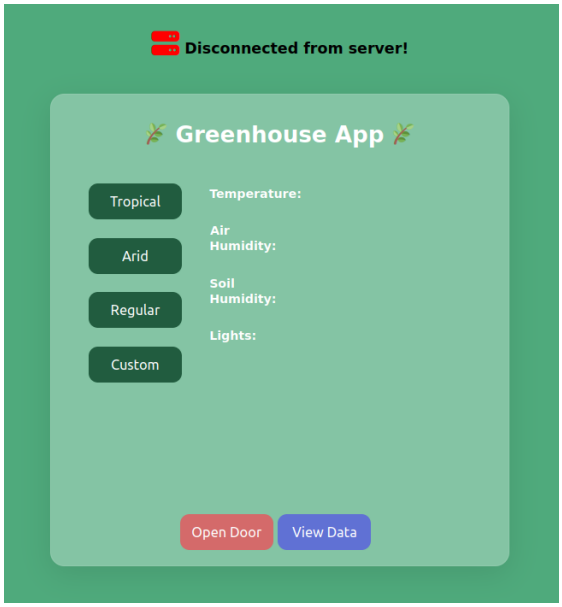

Figure 63: Connection Lost Warning

4.5 Theoretical Concepts

4.5.1 Kernel

In an embedded system, the "kernel" is the central element of the operating system designed to run on devices with limited resources, such as microcontrollers and embedded microprocessors. It manages hardware resources like memory, processors, input/output and communication, and is the basis for running applications and services on the embedded system, as shown in figure 64 below. Embedded kernels have a pivotal role in task coordination, resource allocation, and efficient and dependable system performance, often in real-time scenarios. Various kernel types exist in embedded systems, such as monolithic and microkernel kernels, with each demonstrating distinct characteristics and offering unique trade-offs.

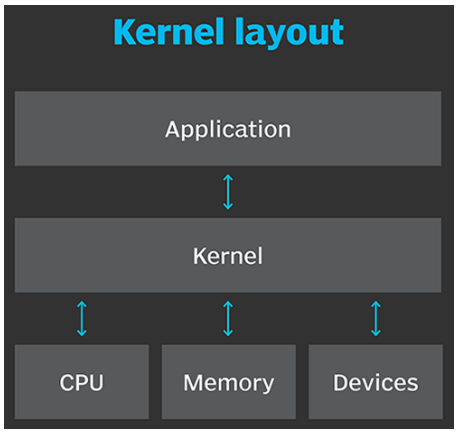

Figure 64: Kernel Layout
[19]

4.5.2 Process vs Thread

Threads and processes are two key concepts in the execution of tasks in computer systems. Processes are independent units with their own memory, whereas threads are lightweight execution units that share the memory of the process. Processes are more isolated and have a higher resource overhead, while threads are more efficient and suitable for tasks that require parallelism and direct communication. The choice between them depends on the specific requirements of the task, balancing isolation and efficiency.

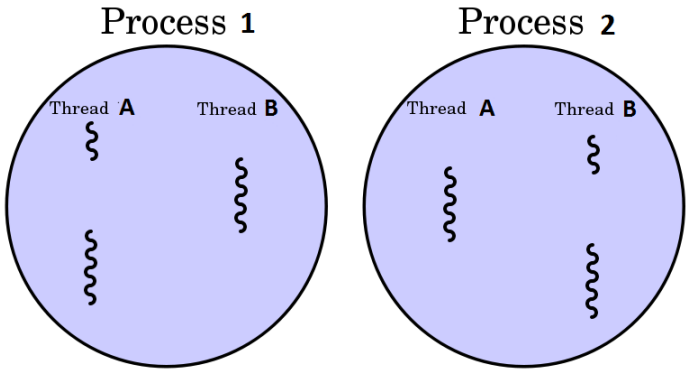

Figure 65: Process and Thread Correlation

4.5.3 Task Synchronization

Mutex

A mutex is one of the synchronization mechanisms available and is used to ensure that critical resources are accessed by only one thread at a time. It is essential for preventing cases in which multiple pieces of code try to access a shared resource simultaneously, which can lead to unexpected behavior and errors. This way, when a thread gets a mutex, other tasks that also try to access the resource are blocked until the mutex is released. This ensures that only one thread has access to the resource at any given time, thus avoiding conflicts and guaranteeing data consistency. In addition, mutexes can also be used to protect critical sections of code, where data-sensitive operations are performed. Mutexes are therefore an essential tool for ensuring that systems work reliably and predictably.

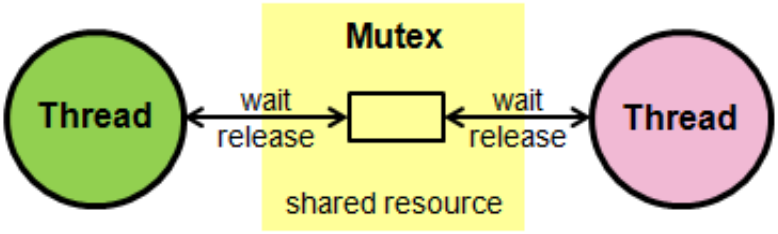

Figure 66: Mutex Management
[20]

Signals

In Linux systems, signals are essential tools for process communication and control. They work by sending asynchronous notifications to processes or threads. This allows the operating system to report important events or request the execution of specific actions. There are various signals available, each of which serves a distinct purpose. For example, SIGINT is used to interrupt a process, while SIGKILL immediately ends it. Signals are crucial for managing processes, facilitating communication between applications, and ensuring system stability. However, they must be handled with care to prevent synchronization issues and achieve proper stability.

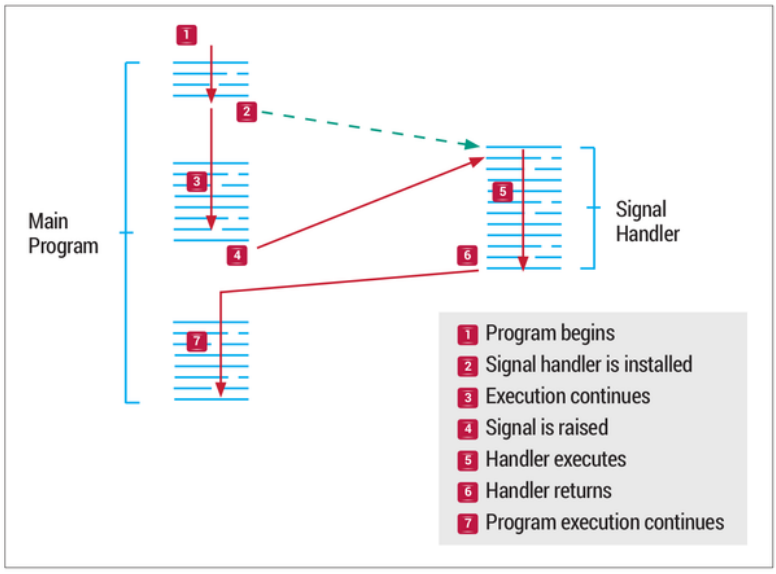

Figure 67: Linux Signals Demonstration
[21]

Condition Variable

A condition variable is a synchronization mechanism that allows tasks to wait for a certain event or condition before continuing their execution. This is useful, since in many cases it is necessary for a task to wait for a specific condition to be met before proceeding. When a task wants to wait for a condition, it releases a mutex that controls access to the shared resource and waits until it is notified so that it can continue its execution.

Message Queue

A message queue is a data structure used in computer systems for inter-process or inter-thread communication. It enables programs to send asynchronous messages to each other, offering an effective and scalable method of exchanging data and synchronizing operations. The messages are queued and read by the other processes in the sequential order in which they were received. Message queues are vital for interprocess communication, enabling applications to work together and efficiently coordinate tasks, without depending on direct communication or memory sharing, which may result in concurrency issues.

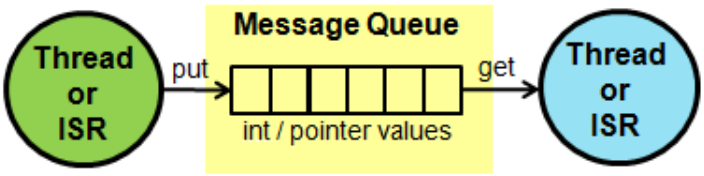

Figure 68: Message Queue Ilustration
[22]

4.5.4 Device Drivers

Device drivers are essential components of a computer's operating system, responsible for enabling effective communication between the hardware and the software application. Device drivers can categorized into two main classes: character device drivers and block device drivers. Character device drivers are designed to handle I/O operations on devices in a character-by-character and are used for a vast variety of components. Block device drivers, on the other hand, are intended for devices that store and retrieve data in fixed-size blocks or sectors, compared to character device drivers, allow the operating system to work with data in more structured and efficient ways.

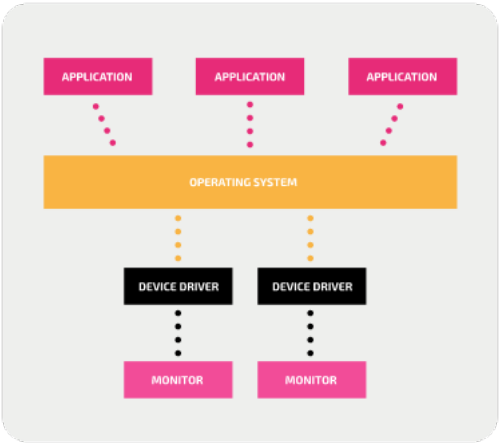

Figure 69: Device Driver Interaction

4.5.5 I²C

I²C (Inter-Integrated Circuit) is a serial communication bus developed by Philips, primarily used for connecting low-speed peripherals. This protocol enables concurrent connection of several target devices on a communication bus through two wires, data line (SDA) and a serial clock line (SCL), and also simplifies and helps multiple controllers that exchange commands and data. Data communication is delivered in byte packets, featuring an individual address for each target device. In the figure 70 below, its possible to see one I²C master multiple slave mode.

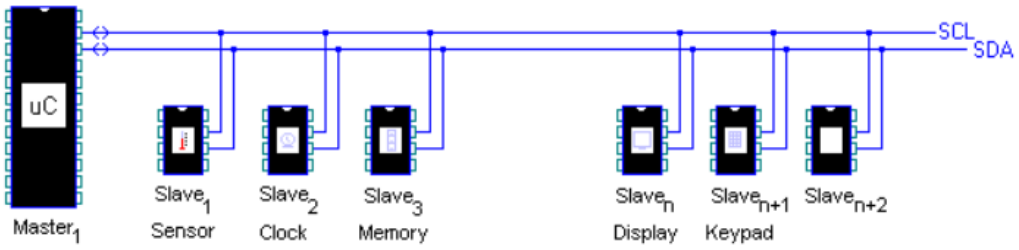

Figure 70: I²C Bus
[23]

I²C communication is initiated by the controller device using an I²C START condition. If the bus is available, an I²C controller claims it for communication by sending an I²C START signal. To do so, the controller device first lowers the SDA line and then lowers the SCL line. This sequence signifies that the controller device is taking control of the I²C. The communication bus forces other controller devices on it to hold their communication.

After the controller device has finished communicating, the SCL signal rises to a high level and subsequently, the SDA signal also rises to a high level. This signifies the occurrence of an I²C STOP condition, which frees up the bus for other controllers to communicate on or for the same controller to initiate communication with another device.

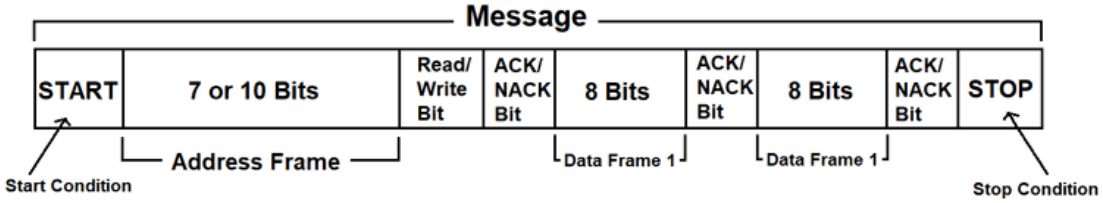

Figure 71: I²C Frame Format

The I²C protocol is split into frames. At the start of communication, the controller device sends an address frame after a START. One or more data frames follow the address frame, with each containing a single byte. Each frame includes an acknowledge bit to inform the controller that the target or controller device has received communication.

Chapter 5

Implementation

5.1 Buildroot Configuration

The Buildroot tool will be configured according to the specific requirements of the system. Once the necessary packages have been chosen, the "make" command is used in the Buildroot directory to apply and incorporate the modifications made. This process provides a flexible configuration, allowing you to adapt the compilation environment according to your project's needs.

5.1.1 SQLite Configuration

SQLite, a C language library, houses the main SQL functions and will play a crucial role in the project. Its main function will be to manage the storage of essential information, such as the data of authorized users (username and password) to access the greenhouse, as well as recording the current values coming from the sensors and the desired parameters for controlling the greenhouse.

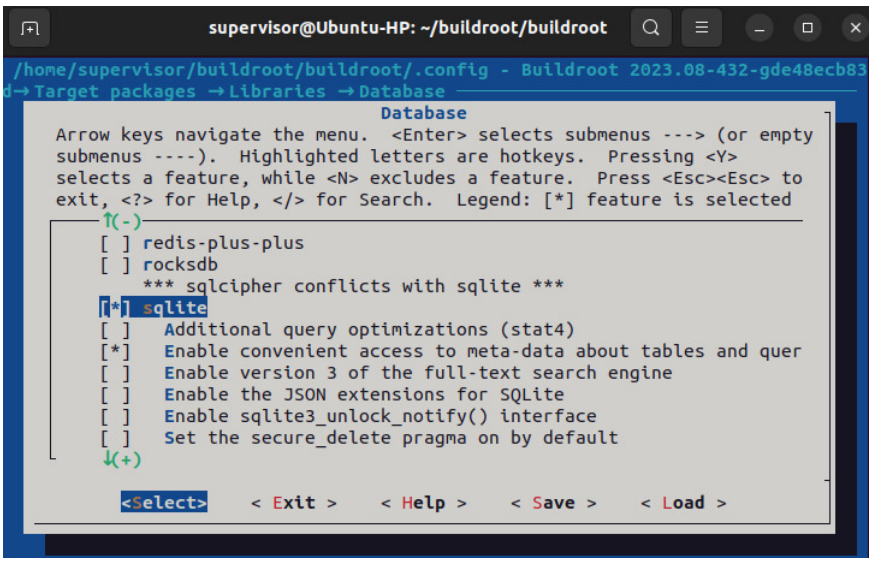

Figure 72: Buildroot configuration of SQLite

5.1.2 Time Data Configuration

Adding the "ntp" and "ntpd" packages to Buildroot makes it possible to synchronize the system clock. "ntp" facilitates time synchronization, while "ntpd" maintains this synchronization continuously, guaranteeing time accuracy on the device.

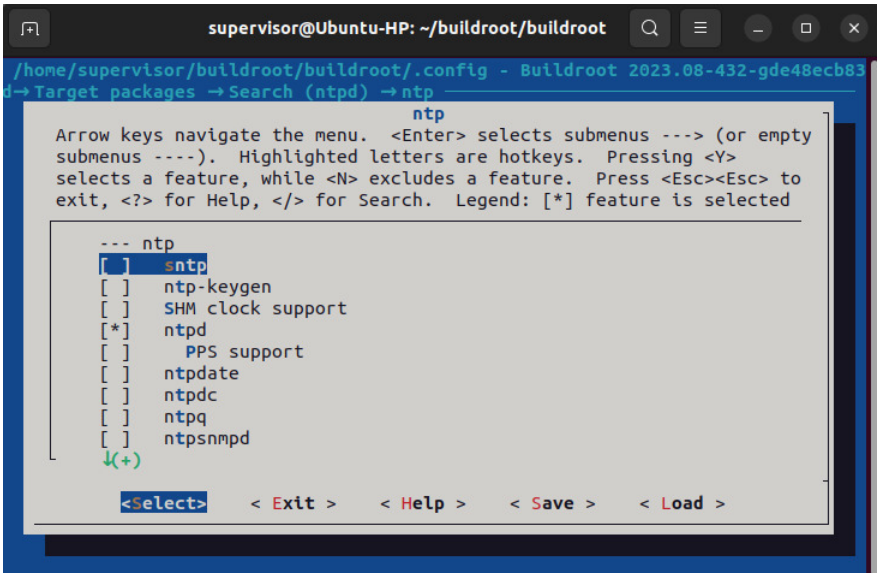

Figure 73: Buildroot configuration for Time Data

5.1.3 Communication Configuration

The inclusion of the "libcpprestsdk" library in Buildroot adds essential communication functionality to the project. This library was used to develop asynchronous communication and HTTP connections to our server, allowing the end user to interact with the local system.

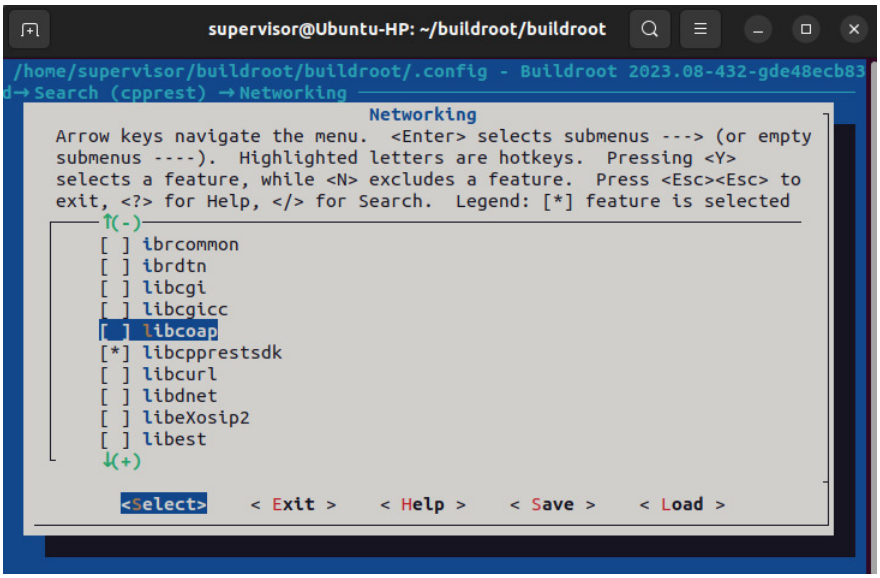

Figure 74: Buildroot configuration for WebSockets

5.2 Hardware

5.2.1 Protoboard Design & Circuitry

Sensors Board 1

The purpose of this protoboard is to connect the temperature and air-humidity sensor, ADC module, and water level sensor to the intermediate protoboard, which connects to the Raspberry Pi 4 pinout.

Circuit Layout

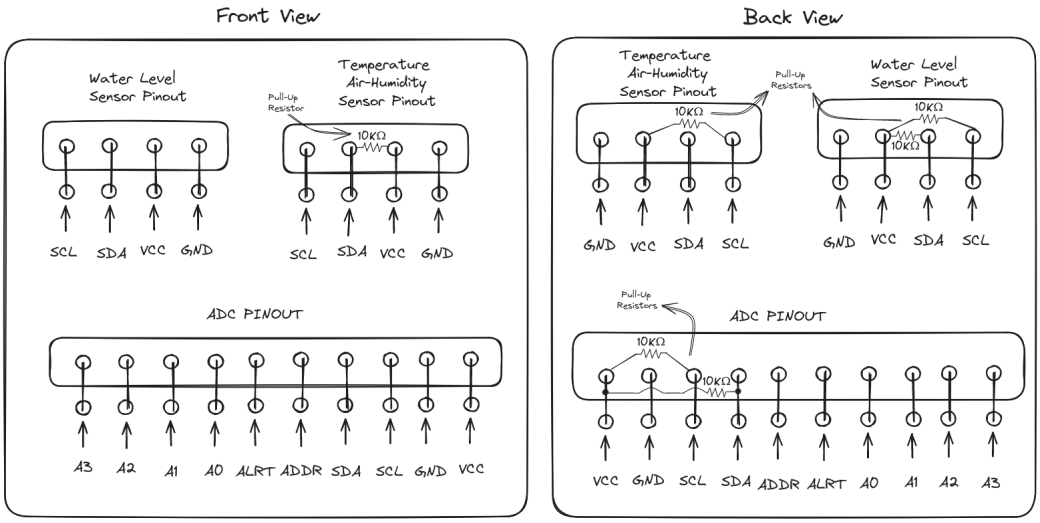

Figure 75: Sensors Board 1 - Circuit Layout

Physical Layout

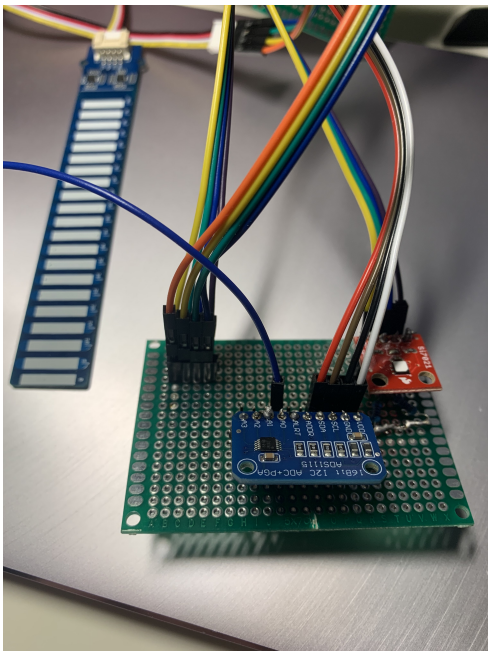

Figure 76: Sensors Board 1 Front View

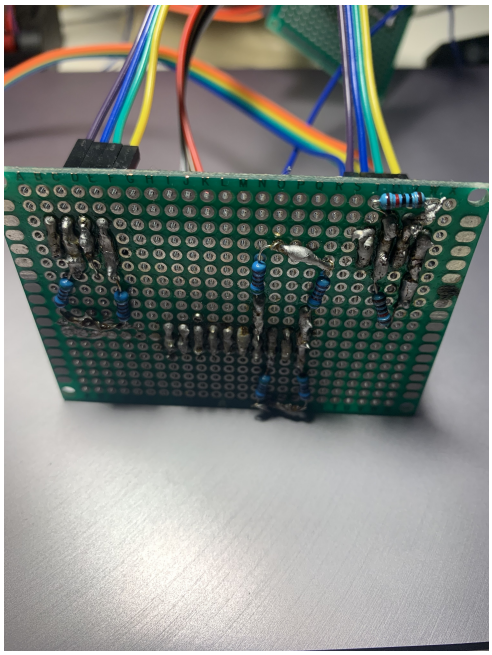

Figure 77: Sensors Board 1 Back View

Sensors Board 2

The protoboard shown in the figures below, fig.79 & fig.80, was designed to house the analogue sensors, i.e. the circuit that detects the intensity of the light, made up of an LDR and a resistor, and the soil moisture sensor.

Circuit Layout

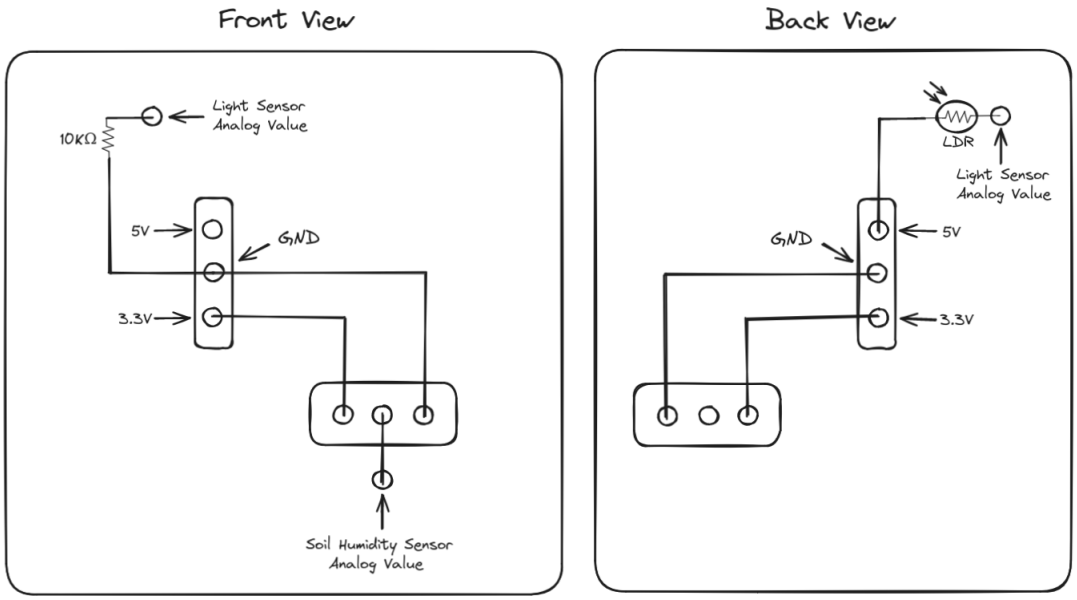

Figure 78: Sensors Board 2 - Circuit Layout

Physical Layout

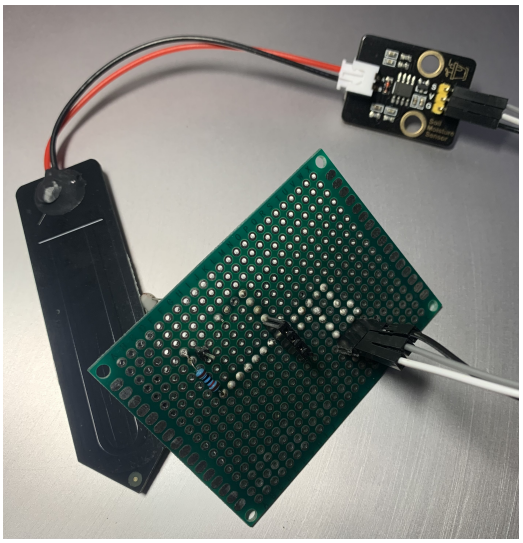

Figure 79: Sensors Board 2
Front View

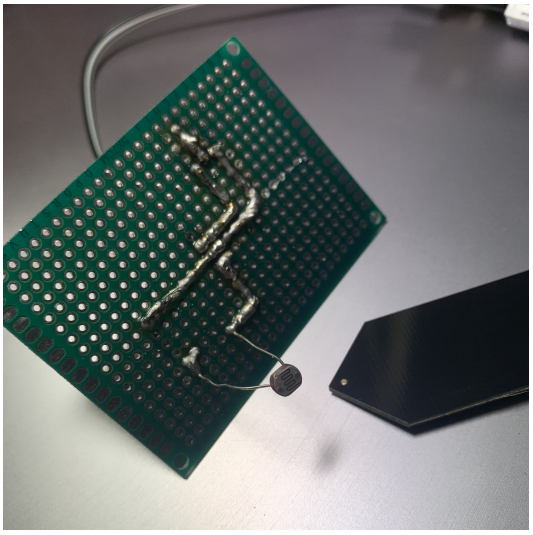

Figure 80: Sensors Board 2
Back View

Connections Board

This protoboard was designed to act as a intermediary between the raspberry pinout and the sensors and actuators of the local system.

Circuit Layout

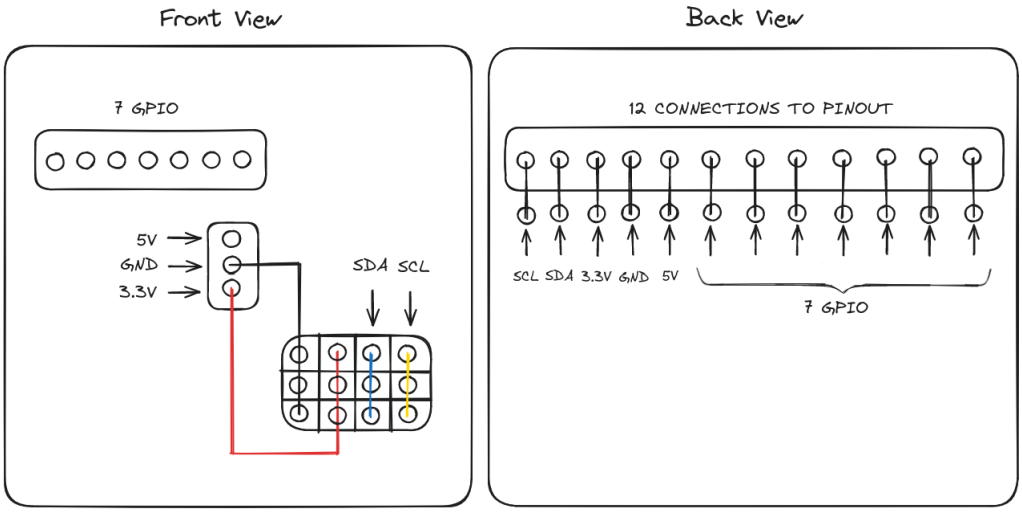

Figure 81: Connections Board - Circuit Layout

Physical Layout

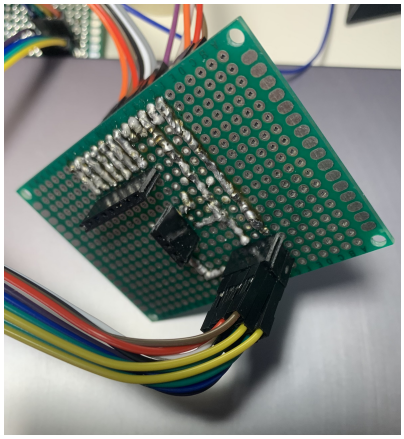

Figure 82: Connections Board Front View

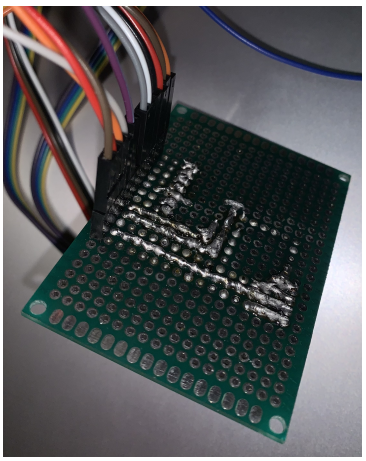

Figure 83: Connections Board Back View

5.3 Local System

5.3.1 Classes

Actuators

The purpose of this class is to configure and manipulate the state of the actuators and calculate the bias between the values obtained and the desired values for the greenhouse.

```
class Actuators{
    private:
        string name;
        bool state;
        int ficheiro;

    public:
        Actuators();    //construtor
        ~Actuators();   //destrutor
        void set_Actuators_State(string, bool); //altera estado dos atuadores
        int reference_calculator(float, float, float);  //calcula o desvio entre os valores na estufa e os valores pretendidos
        void initialize_Actuators(); //inicializa todos os atuadores (como desligados)
};

#endif
```

Listing 5.1: cActuators Interface

Sensor Classes

Temperature & Air-Humidity Sensor:

This class is associated with the configuration and reading of the temperature and humidity sensor. It will then measure the current values in the greenhouse about these parameters, for which a sensor has been used that takes both readings.

```
//classe sensor de temperatura e humidade
class Temp_Hum_Sensor{
    private:
        float temperature, humidity;
    public:
        Temp_Hum_Sensor();
        ~Temp_Hum_Sensor();
        void init();
        float get_Temperature();  //devolve o valor de temperatura lido pelo sensor
        float get_Humidity(); //devolve o valor de humidade
};
```

Listing 5.2: cTemp_Hum_Sensor Interface

Light Sensor:

This class will be responsible for measuring the intensity of the light incident on the greenhouse, using an LDR.

```
//classe do sensor de luz
class LDR_Sensor{
    private:
        float brightness_value, min_light_threshold;
    public:
        LDR_Sensor();
        ~LDR_Sensor();
        float get_brightness_value();  //devolve o valor lido pelo
    sensor
};
```

Listing 5.3: cLDR_Sensor Interface

Soil Humidity Sensor:

Since nothing survives in a greenhouse without water, this class is responsible for reading the soil moisture level and configuring the associated sensor.

```
//classe do sensor de humidade do solo
class Soil_Hum_Sensor{
    private:
        float soil_moisture;
    public:
        Soil_Hum_Sensor();
        ~Soil_Hum_Sensor();
        float get_soil_moisture();    //devolve o valor lido pelo
   sensor
};
```

Listing 5.4: cSoil_Hum_Sensor Interface

Water Level Sensor:

This class will be responsible for checking the water level in the tank associated with the greenhouse and configuring the associated sensor.

```
//classe do sensor que mede o nivel de agua no reservatorio
class Water_Level_Sensor{
    private:
        float water_Level;
    public:
        Water_Level_Sensor();
        ~Water_Level_Sensor();
        int get_water_level();      //devolve o valor lido pelo
   sensor
};
```

Listing 5.5: cWater_Level_Sensor Interface

Database

The class Database takes care of making the connection with the local database using the sqlite wrapper functions.

This class contains the functionality needed to manipulate the different tables associated with the database, store and return the current values present in the greenhouse as well as store and return the desired parameters for it and also store

the data (username and password) of all the users who will have access to the application.

```
class Database{

public:
    Database(const char* dbName);       //construtor -> abre
    Database e ve se conexao foi conseguida
    ~Database();                        //destrutor -> fehca a
    Database

    void insertData(string, float, float, float, float, bool);  //
    inserir dados lidos pelos sensores
    void insertDefineValues(float, float, float, bool);         //
    inserir valores pretendidos na estufa
    void deleteAllData(string);
    void showAllData(string);
    void getDefineValues(float&, float&, float&, bool&);
    void getDataValues(string&, float&, float&, float&, float&,
    float&);
    bool login(string, string);

private:
    sqlite3* db;
};
```

Listing 5.6: cDatabase Interface

System

The System class plays the main role in the system, incorporating both sensors and actuators, highlighting the compositional relationship previously described. In addition, it includes a pointer to the database, highlighting the aggregation relationship between these classes.

Furthermore, the class contains the variables needed to implement the multi-threading system, including the threads implemented, their configuration variables, message queues used and essential mutexes to ensure the proper functioning of the system. Three structs have been implemented to facilitate access to specific sets of variables in a simple and practical way.

As for the public members, in addition to the constructor and destructor, the System class includes the start-up function and various auxiliary functions used in the different threads. These functions deal with receiving signals, reading the current time and encapsulating some functions implemented in the other classes, promoting a modular and easy-to-maintain approach.

5.3.2 System Startup

The program's initialization phase is a logical and designed process. Then, in the class constructor, two I2C modules, namely 'i2c-dev' and 'i2c-bcm2838', are loaded using system commands.

Firstly, the system class's required objects, namely the sensors and actuators, are created. Additionally, the 'gpio_irq' module is loaded using the 'insmod' command. Subsequently, the code acquires the system's creation moment and executes the database initialization.

To carry out this stage, instantiate an object of the 'Database' class and associate it with the 'Greenhouse_Database' pointer of the class.

The 'StartUp' function executes the initialization sequence, which includes initializing mutexes and creating message queues for inter-component communication. If an error occurs during this process, the code will eliminate previously allocated resources to ensure consistency and safe execution.

After initializing the control and communication structures, the function configures the signals used for different events, such as 'SIGUSR1', 'SIGALARM', and 'SIGTERM'. A timer is created and associated with one of the signals to carry out specific operations at set intervals.

The final stage involves initializing the threads with specific attributes. If thread creation fails, an error message is displayed.

Finally, the function 'joinThreads()' waits for the threads to complete.

```
System::System(const char* dbName) {

  system("modprobe i2c-dev");
  system("modprobe i2c-bcm2835");

  const char *comando_gpio_irq = "insmod gpio_irq.ko";
  int resultado_gpio_irq = std::system(comando_gpio_irq);

  // Verificar se a execucao foi bem-sucedida
  if (resultado_gpio_irq == 0) {
    // O comando foi executado com sucesso
    cout << "Modulo do driver carregado com sucesso.\n";
  } else {
    //comando falhou
    aviso("Falha ao carregar o modulo do driver");
  }
  cout << "***************" << endl;
  cout << "System Created " << endl ;

  Greenhouse_Database = new Database(dbName);  //inicia database

  //abre device driver para GPIO17 com interrupt
  int fd = open(DEVICE_PATH, O_RDWR);
  if (fd == -1) {
    erro("Falha ao abrir o dispositivo");
    delete Greenhouse_Database;
    close(fd);
  }

  // Configurar o PID usando a IOCTL
  if (ioctl(fd, REGIST_PID, 0) == -1) {
    erro("Falha ao configurar PID usando IOCTL");
    delete Greenhouse_Database;
    close(fd);
  }
  close(fd);

  startUp();

  pthread_mutex_lock(&mutex_Database);
  define_data storageData;
  getDefineValuesDatabase(define_data_inst);
  pthread_mutex_unlock(&mutex_Database);

  initial_actuators_state_def();
  signal(SIGUSR1, &System::signalHandler);
```

```
47      signal(SIGALRM, &System::alarmHandler);
48      signal(SIGTERM, &System::signalHandler);
49
50      timer_t timerid;
51      struct sigevent sev;
52      struct itimerspec its;
53
54      // Configuracao da estrutura sigevent para notificar via sinal
55      memset(&sev, 0, sizeof(sev));
56      sev.sigev_notify = SIGEV_SIGNAL;
57      sev.sigev_signo = SIGTERM;
58
59      // Cria um timer
60      if (timer_create(CLOCK_REALTIME, &sev, &timerid) == -1) {
61        perror("timer_create");
62        exit(EXIT_FAILURE);
63      }
64
65      // Configuracao do timer para disparar a cada 15 segundos
66      its.it_value.tv_sec = 15;
67      its.it_value.tv_nsec = 0;
68      its.it_interval.tv_sec = 15; // Intervalo tambem configurado
     para 15 segundos
69      its.it_interval.tv_nsec = 0;
70
71      // Configurar o timer
72      if (timer_settime(timerid, 0, &its, NULL) == -1) {
73        perror("timer_settime");
74        exit(EXIT_FAILURE);
75      }
76      initializeThreads();
77      joinThreads();
78    }
```

Listing 5.7: System Constructor

5.3.3 Processes

Within the system, there are two distinct processes that operate together to ensure correct functionality. The first process, called the "Main process", takes on the critical responsibility of executing the main code, which includes reading sensors, calculating averages, controlling actuators, and storing them in the database. The second process, "Communication process", plays a key role in facilitating interaction between the Main Process and external communications, including communication with the remote server.

Communication between these two processes is established via message queues. This communication mechanism offers an efficient and asynchronous solution for exchanging data between processes.

There will be 2 message queues that will exchange data unidirectionally between the two processes. The Communication Process receives data from the remote server and places it in an internal message queue, so that the Main Process can access it and store it in the database. The Main Process deposits information in the message queue and the Communication Process is responsible for retrieving this data and transmitting it to the remote server.

In this case, the current values of the greenhouse will be taken from the database of the "main process" to the remote system, while the values defined by the user in the remote system will travel the reverse route, going from the server to the "communication process" and from there to the main process where

they will be stored in the database.

This distributed architecture provides an efficient separation of concerns, allowing the Core Process to concentrate on the system's essential operations, while the Communication Process handles data exchange and external connection.

5.3.4 Threads

tReadSensors

This thread will be responsible for reading the sensor values and sending them to another thread via message queue. The sensors are read individually but stored in a struct for easier access and management.

```
void *System::tReadSensors(void *arg){

    sensors_data system_data;
    System *local_sys = static_cast<System*>(arg);

    while(1) {
        cout << "entrou thread sensores" << endl;

        pthread_mutex_lock(&local_sys->mutex_ReadSensors_msg);
            system_data = local_sys->get_sensor_data();
            cout << "horas: " << system_data.timestamp << endl;
        pthread_mutex_unlock(&local_sys->mutex_ReadSensors_msg);

       int result = local_sys->sendTomyQueue(system_data);
            cout << "resultado do envio: " << result << endl <<
   flush;

       if (result == -1) {
        perror("mq_send");
       } else {
        cout << "Message sent successfully!" << endl << flush; ;
       }
        sleep(1);  // 1s delay
    }
}
```

Listing 5.8: Read Sensors Thread

tCalculateRef

This thread will receive the values read by the sensors via a struct and store them in a vector. When the vector reaches 10 units, its average will be calculated and sent to the database thread, also using a message queue. The mutex used is due to the fact that access to the database is also carried out in other threads.

At first, the last desired values will be taken from the database, with the average values of the sensors and with the now updated desired values, the difference between them will be calculated, considering a certain hysteresis margin. This difference will then be sent to the actuators thread.

The chosen method of not using the value read by the sensor, but rather the average of a set of samples, contributes to greater stability and increased precision, as well as reducing point errors.

```
float calculateAverage(const vector<float>& buffer) {

    if (buffer.empty()) {
        return 0.0;  // Retorna 0 se o vetor estiver vazio para
   evitar divisao por zero
```

```
    }

    float sum = 0.0;
    for (size_t i = 0; i < buffer.size(); ++i) {
        sum += buffer[i];
    }
    return (sum / buffer.size());
}
```

Listing 5.9: Calculate Average Function

tActuators

This thread, after receiving the current bias in the greenhouse parameters, the difference between the desired value and the current value considering a hysteresis margin, will be responsible for managing the state of the actuators. The previous state of the actuators is checked so as not to generate repeatability in the state of some actuators, but, when necessary, their state will be changed.

```
void *System::tActuators(void *arg) {

    int received_data [sizeof(int) * 6];
    System *local_sys = static_cast<System*>(arg);

    while (1) {
        cout << "entrou thread Atuadores" << endl;

        ssize_t bytesRead;// = local_sys->receiveFromActuators(
    received_data);

        if (mq_receive(QueueActuators, reinterpret_cast<char*>(
    received_data), sizeof(int) * 6, 0) == -1) {
            perror("mq_receive");
            bytesRead = -1;
            cout << "resultado da rececao na atuadores: " <<
    bytesRead << endl;
        } else {
            bytesRead = 0;
            cout << "resultado da rececao na atuadores: " <<
    bytesRead << endl;
        }

        if (bytesRead == 0) {  // Verificar se a mensagem tem o
    tamanho correto

            pthread_mutex_lock(&local_sys->mutex_Actuators);

                int temp = received_data[0];
                int hum = received_data[1];
                int soilhum = received_data[2];
                int light = received_data[3];
                int waterlevel = received_data[4];
                int door = received_data[5];

                actuators_state prev_actuators_state;
                prev_actuators_state = local_sys->
    realtime_actuators_state;

            pthread_mutex_unlock(&local_sys->mutex_Actuators);
```

Listing 5.10: Actuators Thread Extract

All the actuators are managed in the same way, except for the greenhouse door. After the button that allows the greenhouse to open is pressed, a SIGNAL will be sent, in which it will send the corresponding function the need to change the status of the door.

```
void System::signalHandler(int signum) {

    if(signum == SIGUSR1){

        int vetorDoor [6] = {0, 0, 0, 0, 0, 1};
        //Abertura da porta
        if (mq_send(QueueActuators, reinterpret_cast<const char*>(
    vetorDoor), sizeof(int) * 6, 0) == -1) {
            perror("mq_send");
        }

        cout << "Recebeu o sinal SIGUSR1" << std::endl;

        //Porta mantida aberta durante 5s
        alarm(5);
```

Listing 5.11: Signal Handler Extract - Open Security Door

tDatabase

After the CalculateRef thread has calculated the average of the samples read by the sensors, these will be received via a message queue in the Database thread. Then lock the database mutex, since more than one thread accesses the database, the data will store the current values of the greenhouse and after that the mutex will be released.

```
void *System::tDatabase(void *arg){

    System *local_sys = static_cast<System*>(arg);
    sensors_data receivedDatabase_data;

    while (1) {
        std::cout << "entrou thread Database" << endl;

        int status = local_sys->receiveFromDatabase(
    receivedDatabase_data);

        if (status == 0) {

            // Imprimir os valores
            cout << "Received data" << endl;
            cout << "hours: " << receivedDatabase_data.timestamp
    << endl;
            cout << "Temperature: " << receivedDatabase_data.
    temperature << endl;
            cout << "Air Humidity: " << receivedDatabase_data.
    air_humidity << endl;
            cout << "Soil Humidity: " << receivedDatabase_data.
    soil_humidity << endl;
            cout << "Water Level: " << receivedDatabase_data.
    water_level << endl;
            cout << "Light Level: " << receivedDatabase_data.
    light_level << endl;

            bool light_state;
            if(receivedDatabase_data.light_level < LIGHT_DEFINE &&
     local_sys->define_data_inst.define_light == 1){
```

```
                light_state = true;
            } else {
                light_state = false;
            }

            string time = receivedDatabase_data.timestamp;

            float temperature = receivedDatabase_data.temperature;
            float humidity_air = receivedDatabase_data.
    air_humidity;
            float soil_humidity = receivedDatabase_data.
    soil_humidity;
            float water_level = receivedDatabase_data.water_level;

            pthread_mutex_lock(&local_sys->mutex_Database);
                local_sys->Greenhouse_Database->insertData(time,
    temperature, humidity_air, soil_humidity, water_level,
    light_state);
            pthread_mutex_unlock(&local_sys->mutex_Database);

        } else {
            cout << "Invalid message size or no data available" <<
     endl;
        }
        usleep(100000);  // Aguarda por 100ms
    }
}
```

Listing 5.12: Database Thread

tSendRemote

This thread will be responsible for receiving the desired parameters from the second process, previously sent by the server. After receiving the values, already filtered by the second process, and after locking the database mutex, the values will be stored in the respective table. They will be useful in the future when the bias between the current values and the desired values is calculated.

```
void *System::tsendRemote(void *arg){

    System *local_sys = static_cast<System*>(arg);
    float receiveBuffer[4];

    while(1){
        cout << "Entrou na thread que vai receber os dados!!!" <<
    endl;

        ssize_t bytesRead;

        if (mq_receive(local_sys->QueuereceiveRemote,
    reinterpret_cast<char*>(receiveBuffer), sizeof(float)* 4, 0) ==
     -1) {
            perror("mq_receive");
            bytesRead = -1;
            cout << "Data recieved on Process 1: " << bytesRead <<
     endl;
        } else {
            bytesRead = 0;
            cout << "Data recieved on Process 1: " << bytesRead <<
     endl;

            bool light_def = false;
```

```
20
21              if(receiveBuffer[3] == 1){
22                  light_def = true;
23              }
24
25              pthread_mutex_lock(&local_sys->mutex_Database);
26                  local_sys->Greenhouse_Database->insertDefineValues
    (receiveBuffer[0],receiveBuffer[1],receiveBuffer[2], light_def)
    ;
27                  local_sys->define_data_inst.define_temperature =
    receiveBuffer[0];
28                  local_sys->define_data_inst.define_humidity =
    receiveBuffer[1];
29                  local_sys->define_data_inst.define_soilhumidity =
    receiveBuffer[2];
30                  local_sys->define_data_inst.define_light =
    light_def;
31              pthread_mutex_unlock(&local_sys->mutex_Database);
32
33
34              cout << "Received Data: ";
35              for (int i = 0; i < 4; ++i) {
36                  cout << receiveBuffer[i] << " ";
37              }
38              cout << std::endl;
39          }
40      }
41  }
```

Listing 5.13: Receive From Remote Thread

tReceiveFromLocal

This thread will receive data from the main process via a WebSocket and convert it to JSON before sending it to the server. The values sent to the server will correspond to the current values.

```
1  void* tReceiveFromServer(void* thread_ID){
2
3    while(1){
4
5      client.receive().then([](websocket_incoming_message msg) {
6        return msg.extract_string();
7      }).then([](string content) {
8
9        // Converter a string wide para string normal (UTF-8)
10       string utf8_content(content.begin(), content.end());
11
12       //process data -> parsing -> switch das actions -> enviar
    para local via msg_queue
13       processData(utf8_content);
14
15
16       ssize_t bytesRead = mq_send(QueuereceiveRemote,
    reinterpret_cast<char*>(define_buffer), sizeof(float)* 4, 0);
17
18       if (bytesRead == -1) {
19         std::cerr << "Error receiving data in Process 2" << std::
    endl;
20         perror("mq_receive");
21       } else {
22         cout << "Data received in process 2!" << endl;
23       }
```

```
24
25        cout << "Received Message: " << utf8_content << std::endl;
26      }).wait();
27    }
28
29    return nullptr;
30 }
```

Listing 5.14: Receive From Server Thread

tReceiveFromServer

This thread is responsible for receiving data from the server using a blocking function. The received data is then parsed using JSON and converted into a string.

The parsed data is then sent via a WebSocket to the main system and stored in the database.

The received values correspond to the user's desired values for the greenhouse.

```
1  void* tReceiveFromLocal(void* thread_ID){
2
3    float sensor_data_buffer[11];
4
5    while(1){
6
7      //Receive data from rasp
8      ssize_t bytesRead = mq_receive(QueuesendRemote,
       reinterpret_cast<char*>(sensor_data_buffer), sizeof(float)* 11,
       0);
9
10     if (bytesRead == -1) {
11       std::cerr << "Error receiving data in Process 2" << std::
       endl;
12       perror("mq_receive");
13     } else {
14       cout << "Data received in process 2!" << endl;
15     }
16
17     std::tm tm = {};
18
19     tm.tm_mday = static_cast<int>(sensor_data_buffer[0]);  // Dia
20     tm.tm_mon  = static_cast<int>(sensor_data_buffer[1])- 1;  //
       Mes
21     tm.tm_year = static_cast<int>(sensor_data_buffer[2])- 1900;
       // Ano
22     tm.tm_hour = static_cast<int>(sensor_data_buffer[3]);  // Hora
23     tm.tm_min  = static_cast<int>(sensor_data_buffer[4]);  //
       Minuto
24     tm.tm_sec  = static_cast<int>(sensor_data_buffer[5]);  //
       Segundo
25
26     std::ostringstream oss;
27     oss << std::put_time(&tm, "%d-%m-%Y %H:%M:%S");
28
29
30     string timestamp = oss.str();
31
32     string temperature = to_string(sensor_data_buffer[6]);
33     string air_humidity = to_string(sensor_data_buffer[7]);
34     string soil_humidity = to_string(sensor_data_buffer[8]);
35     string water_level = to_string(sensor_data_buffer[9]);
36
37     string light = "OFF" ;
```

```
38
39      int a = sensor_data_buffer[10];
40
41      if(a == 1){
42        light = "ON";
43      } else {
44        light = "OFF";
45      }
46
47      for(int i=0; i<5; i++){
48        cout << " buffer -> " << sensor_data_buffer[i] << endl;
49      }
50
51      //Send data to server
52      try {
53        // JSON msg to send
54        web::json::value json_msg;
55
56        json_msg[U("action")]  = web::json::value::string(U("
    update_data"));
57
58        json_msg[U("temp")]    = web::json::value::string(timestamp)
    ;
59        json_msg[U("a_hum")]   = web::json::value::string(
    temperature);
60        json_msg[U("s_hum")]   = web::json::value::string(
    air_humidity);
61        json_msg[U("light")]   = web::json::value::string(
    soil_humidity);
62        json_msg[U("water")]   = web::json::value::string(
    water_level);
63        json_msg[U("time")]    = web::json::value::string(light);
64
65        utility::string_t dataJSON = json_msg.serialize();
66
67        // Send msg to server
68        websocket_outgoing_message out_msg;
69        out_msg.set_utf8_message(dataJSON);
70        client.send(out_msg).wait();
71
72        cout << "Mensagem enviada com sucesso para o servidor." <<
    endl;
73
74      } catch (const web::websockets::client::websocket_exception& e
    ) {
75        cerr << "Erro ao enviar mensagem: " << e.what() << endl;
76      } catch (const std::exception& e) {
77        cerr << "Erro desconhecido: " << e.what() << endl;
78      }
79
80      //get messages via message queue
81      sleep(5);
82    }
83
84    return nullptr;
85  }
```

Listing 5.15: Receive From Server Thread

5.3.5 Device Drivers

Device drivers play a crucial role in facilitating communication between hardware and user space code. They provide standardized calls invoked in user space, which are then mapped to specific devices. This makes it possible to reference the desired driver for communication after each operation.

The kernel module can be seamlessly integrated into the Linux kernel using the insmod command and subsequently removed using rmmod. Modules run in kernel space, while applications run in user space.

In kernel space, Linux offers a set of functions that interact directly with the hardware and allow the transfer of data between kernel space and user space. Therefore, each device driver will implement only the necessary functions for its respective functionality.

GPIO Input With Interruption

In the context of the GPIO with interrupt, during initialization (init), the function called in "insmod", the desired GPIO is configured as input. It is then assigned the number of an interrupt that will be generated when a rising transition occurs. Subsequently, the driver is configured and registered in the system.

When the module is unloaded (exit), the resources allocated to the device driver are released. In the code provided, the OPEN and RELEASE functions have been implemented mainly for visual reasons.

During an upward transition, the gpio_irq_handler function is triggered, generating an interruption. At this point, a SIGUSR1 signal is sent to a specific process via the irq_handler_t function. To determine which process should receive the signal, the ioctl function is used. This function stores the PID of the main process when a certain condition is met.

This device driver will be used in the scenario where the door is opened when someone clicks a button. In this context, the system receives a signal that triggers the door to open. After receiving this signal, another signal is generated for a period of 5 seconds. At the end of these 5 seconds, the door is automatically closed.

```
1  static irq_handler_t gpio_irq_handler(unsigned int irq, void *
       dev_id, struct pt_regs *regs) {
2      pr_info("gpio_irq: Interrupcao foi acionada e ISR foi chamada
       !\n");
3
4      if (user_task) {
5          pr_info("gpio_irq: Enviando sinal para o processo do user\
       n");
6          send_sig(SIGUSR1, user_task, 0);
7      } else {
8          pr_err("gpio_irq: Processo do user nao encontrado\n");
9      }
10
11     return (irq_handler_t)IRQ_HANDLED;
12 }
13
14 static long device_ioctl(struct file *file, unsigned int cmd,
       unsigned long arg) {
15     switch (cmd) {
16         case REGIST_PID:
17             user_task = get_current();
18             printk("GPIO_PID: %d\n", user_task->pid);
19             break;
20
```

```
21          default:
22              return -EINVAL; // Comando IOCTL invalido
23      }
24
25      return SUCCESS;
26  }
27
28  static int device_open(struct inode *inode, struct file *file) {
29      pr_info("gpio_irq: Device opened\n");
30      return 0;
31  }
32
33  static int device_release(struct inode *inode, struct file *file)
        {
34      pr_info("gpio_irq: Device closed\n");
35      return 0;
36  }
37
38  static struct file_operations device_fops = {
39      .owner = THIS_MODULE,
40      .unlocked_ioctl = device_ioctl,
41      .release = device_release,
42      .open = device_open,
43  };
44
45  static int __init gpio_irq_init(void) {
46      pr_info("gpio_irq: Modulo a carregar... ");
47
48      if (gpio_request(17, "rpi-gpio-17")) {
49          pr_err("Erro!\nNao e possivel alocar GPIO 17\n");
50          return -1;
51      }
52
53      if (gpio_direction_input(17)) {
54          pr_err("Erro!\nNao e possivel configurar GPIO 17 como
        entrada!\n");
55          gpio_free(17);
56          return -1;
57      }
58
59      irq_number = gpio_to_irq(17);
60
61      if (request_irq(irq_number, (irq_handler_t)gpio_irq_handler,
        IRQF_TRIGGER_RISING, "meu_gpio_irq", NULL) != 0) {
62          pr_err("Erro!\nNao e possivel solicitar a interrupcao num:
         %d\n", irq_number);
63          gpio_free(17);
64          return -1;
65      }
66
67      pr_info("Concluido!\n");
68      pr_info("GPIO 17 esta mapeado para a interrupcao num: %d\n",
        irq_number);
69
70      // Inicializar a estrutura cdev
71      cdev_init(&c_dev, &device_fops);
72
73      // Alocar numero de dispositivo dinamicamente
74      if (alloc_chrdev_region(&device_number, 0, 1, DEVICE_NAME)) {
75          pr_err("Erro!\nNao e possivel alocar numero de dispositivo
        \n");
76          free_irq(irq_number, NULL);
77          gpio_free(17);
```

```
78          return -1;
79      }
80
81      // Adicionar cdev ao sistema
82      if (cdev_add(&c_dev, device_number, 1)) {
83          pr_err("Erro!\nNao e possivel adicionar cdev ao sistema\n"
    );
84          unregister_chrdev_region(device_number, 1);
85          free_irq(irq_number, NULL);
86          gpio_free(17);
87          return -1;
88      }
89
90      // Criar a classe do dispositivo
91      if (IS_ERR(device_class = class_create(THIS_MODULE, "
    gpio_irq_class"))) {
92          pr_err("Erro!\nNao e possivel criar a classe do
    dispositivo\n");
93          unregister_chrdev_region(device_number, 1);
94          free_irq(irq_number, NULL);
95          gpio_free(17);
96          return -1;
97      }
98
99      // Criar o dispositivo no /dev
100     if (IS_ERR(device_object = device_create(device_class, NULL,
    device_number, NULL, DEVICE_NAME))) {
101         pr_err("Erro!\nNao e possivel criar o dispositivo no /dev\
    n");
102         class_destroy(device_class);
103         unregister_chrdev_region(device_number, 1);
104         free_irq(irq_number, NULL);
105         gpio_free(17);
106         return -1;
107     }
108
109     pr_info("Numero de dispositivo alocado dinamicamente: %d\n",
    MAJOR(device_number));
110
111     return 0;
112 }
113
114 static void __exit gpio_irq_exit(void) {
115     pr_info("gpio_irq: A descarregar o modulo... ");
116     cdev_del(&c_dev);
117     unregister_chrdev_region(device_number, 1);
118     free_irq(irq_number, NULL);
119     gpio_free(17);
120     class_destroy(device_class);
121     device_destroy(device_class, device_number);
122 }
123
124 module_init(gpio_irq_init);
125 module_exit(gpio_irq_exit);
```

Listing 5.16: GPIO_IRQ_Handler Device Driver

Relay Module - 6 GPIO Output

The device driver for controlling the actuators, which uses a relay module, has been implemented to handle the uniform actuation and control of all the actuators, configured as GPIO outputs. In the initialization process (init), a device class is created, a device is associated with this class, and it is added to the system. Subsequently, the pins are configured as GPIO outputs.

In the exit function, as a matter of good practice, the pins used are configured as GPIO inputs. The device is then removed and destroyed, along with the associated class.

In the context of the functionality implemented, only the write function was developed. On receiving data from the user, the function checks its validity. It then configures the GPIO associated with the first character according to the desired state indicated by the second character.

```
int main(void)
{
    int fd = open("/dev/rele0", O_WRONLY);

    if (fd < 0)
    {
        perror("Erro ao abrir o dispositivo");
        exit(EXIT_FAILURE);
    }

    int releNumber = 4; // Numero do rele a controlar (1, 2, 3, ...)
    char comando[3];    // Formato do comando: "XY" onde X e o numero do rele e Y e o estado (0 ou 1)

    for (int count = 0; count < 5; ++count)
    {
        // Ligar o rele
        sprintf(comando, "%d1", releNumber);
        write(fd, comando, strlen(comando));

        printf("Rele %d ligado!\n", releNumber);
        sleep(2); // Esperar por 2 segundos

        // Desligar o rele
        sprintf(comando, "%d0", releNumber);
        write(fd, comando, strlen(comando));

        printf("Rele %d desligado!\n", releNumber);
        sleep(2); // Esperar por 2 segundos
    }

    close(fd);
    return 0;
}
```

Listing 5.17: Relay Module Device Driver

5.4 Remote System

5.4.1 Server

To develop the remote system and enable communication and interconnection with the local system, Node.js was used with the express framework to create a server.

This server allows interaction between the user's remote device (such as a mobile phone or computer connected to the network) and the local system, in this case the greenhouse, as well as providing web pages to give the user a user-friendly interface for operating the system in question. To enable bidirectional communication in real time, we used WebSockets via libraries provided by the express framework and buildroot.

During all the product's testing and production, the server was hosted on a laptop connected to the local network, so users who wanted to interact with the local system had to be connected to the same network.

In a real-life scenario, this server is able to communicate to anywhere in the world, as long as it has an Internet connection. However, due to time and resource constraints, as well as limited knowledge in the area of networks and telecommunications, particularly with regard to port forwarding techniques, firewall configurations and SSL certificates, the server will remain running on localhost for the demonstration.

Node.js + Express + WebSockets

The combination of Node.js, Express framework, and WebSockets was chosen because real-time programming was a main requirement for the project.

This combination offers a powerful solution for building servers that support bidirectional communication in real time and making dynamic web pages available. Node.js enables JavaScript execution on the server side, while Express simplifies web application creation.

The use of WebSockets, through either the Socket.IO or ws library, enables real-time communication between the server and clients. WebSockets provide a continuous connection between the server and the client, beginning with the initial 'Handshake', which allows for efficient and immediate exchange of messages in both directions until the connection is closed on one side.

Server Configuration

```
1  const express = require('express');
2  const http = require("http");
3  const WebSocket = require("ws");
4
5  const app = express();
6  const server = http.createServer(app);
7  const wss = new WebSocket.Server({ server });
8
9  const APP_PORT = process.env.PORT || 5000;
10 const APP_URL = process.env.URL || `http://localhost:${APP_PORT
       }`;
11
12 const { readFile } = require('fs').promises;
13
14 const ACTIONS = {
15     LOGIN:          "login",
```

```
    SIGN_UP:         "sign_up",
    UPDATE_DEFINES: "update_def",
    OPEN_DOOR:       "open_door",
    UPDATE_DATA:     "update_data",
    AUTHENTICATION: "authentication",
    CONNECTION:      "connection",
};

app.use(express.static('public'));

//GET root
app.get('/', async(request, response) => {
    response.send(await readFile('./public/index.html', 'utf-8'));
});


//GET homepage
app.get('/home', async(request, response) => {
    response.send(await readFile('./public/home.html', 'utf-8'));
});


//Server listening on APP_PORT
server.listen(APP_PORT, () => console.log(`App Available on ${APP_URL}`));

let clients_html = [];
let clients_rasp = [];

//Connection to Web Socket Server
wss.on("connection", (ws) => {
        console.log("Client connected to server!");

    ws.on("close", () => {
        console.log("Client disconnected from server!");
    })

    ws.on("message", handleIncomingMsg.bind(null, ws));
});
```

Listing 5.18: Server Configuration

Events

Node.js has an event-driven architecture, with handles designed to manage interactions with the server. These handles work for both user-side web page interactions and local system interactions, which in this case involves the Raspberry Pi.

The approach taken was to analyse the use cases of the application and the tools that would be made available to the end user, such as visualising greenhouse data in real time, defining metrics, among others.

The main handle (handleIncomingMsg) will process the messages coming from both sides and, depending on the action described in the JSON packet exchanged, will analyse the request and redirect it to the specific handle for the intended action.

```
function handleConnection(ws, msg, type){
    if(type == "html"){
        clients_html.push(ws);
        console.log("HTML client connected!");
    } else if(type == "rasp"){
        clients_rasp.push(ws);
        console.log("RASP system connected!");
    } else {
        console.log("Client undefined!");
    }
}

function handleUpdateDefines(temp_def, a_hum_def, s_hum_def,
    light_define){

    const temp = temp_def;
    const air_hum = a_hum_def;
    const soil_hum = s_hum_def;
    const light = light_define;


    console.log("***** Define Values *****");
    console.log("  Temperature   -> ", temp);
    console.log("  Air Humidity  -> ", air_hum);
    console.log("  Soil Humidity -> ", soil_hum);
    console.log("  Light         -> ", light);
    console.log("************************");

    const dataJSON = JSON.stringify({
        action: ACTIONS.UPDATE_DEFINES,
        temperature: temp,
        air_humidity: air_hum,
        soil_humidity: soil_hum,
        light_def: light});

    clients_rasp.forEach((client) => {
        client.send(dataJSON);
    });
};

function handleUpdateData(timestamp, temp, a_hum, s_hum,
    water_level, light){

    console.log("***** Update Data *****");
    console.log("  Time -> ", timestamp);
    console.log("  Temperature   -> ", temp);
    console.log("  Air Humidity  -> ", a_hum);
    console.log("  Soil Humidity -> ", s_hum);
    console.log("  Light         -> ", light);
    console.log("  Water         -> ", water_level);
    console.log("************************");

    const dataJSON = JSON.stringify({
        action: ACTIONS.UPDATE_DATA,
        time: timestamp,
        temperature: temp,
        air_humidity: a_hum,
        soil_humidity: s_hum,
        light_sens: light,
        water: water_level,
        });

    clients_html.forEach((client) => {
```

```
62            client.send(dataJSON);
63        });
64    }
65    });
```

Listing 5.19: Event Handles Extract

Figures 84 and 85 illustrate the interaction between the Local system, Server, and Client. The Local system sends real-time sensor information in JSON packets via websockets. The node.js server intercepts the message and processes it using the incoming message handle. Finally, the server sends the data to all remote clients connected to the web app via websockets.

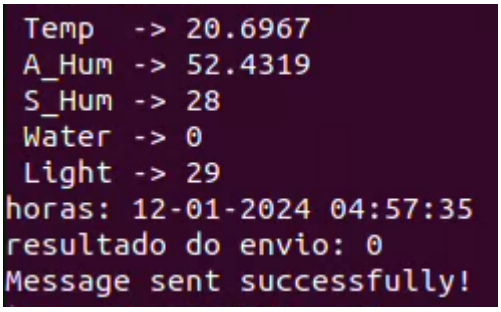

Figure 84: Local System Interaction With Client Through Server

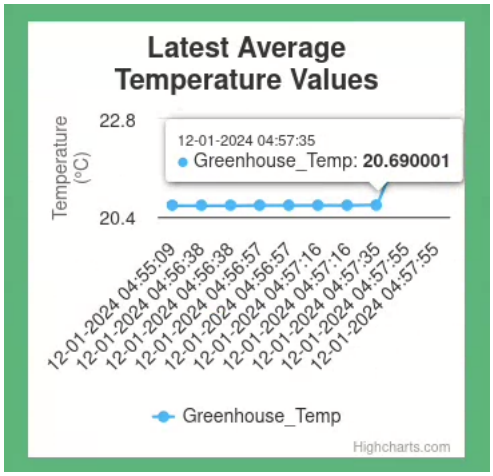

Figure 85: Client Receiving Data Through Server

Figure 86 below shows how the client interacts with the local system through the server via its GUI. In this example, the user is setting the parameters for the greenhouse operation, including a temperature of 21°, air humidity of 45%, soil humidity of 55%, and the light turning on when it reaches the threshold defined in the code.

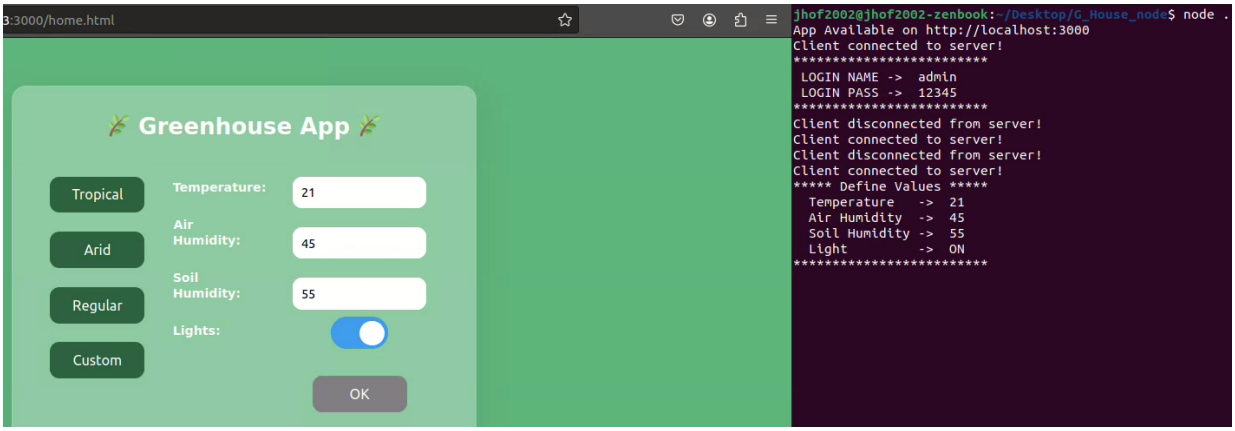

Figure 86: Client-Server Interaction

5.4.2 Remote App

The application's aesthetic design was first created as a static HTML and CSS layout. Once the desired layout was achieved, we added simple JavaScript functions for button interactions, image display, and page navigation. The GUI displays temperature and humidity graphs drawn using the Highcharts API.

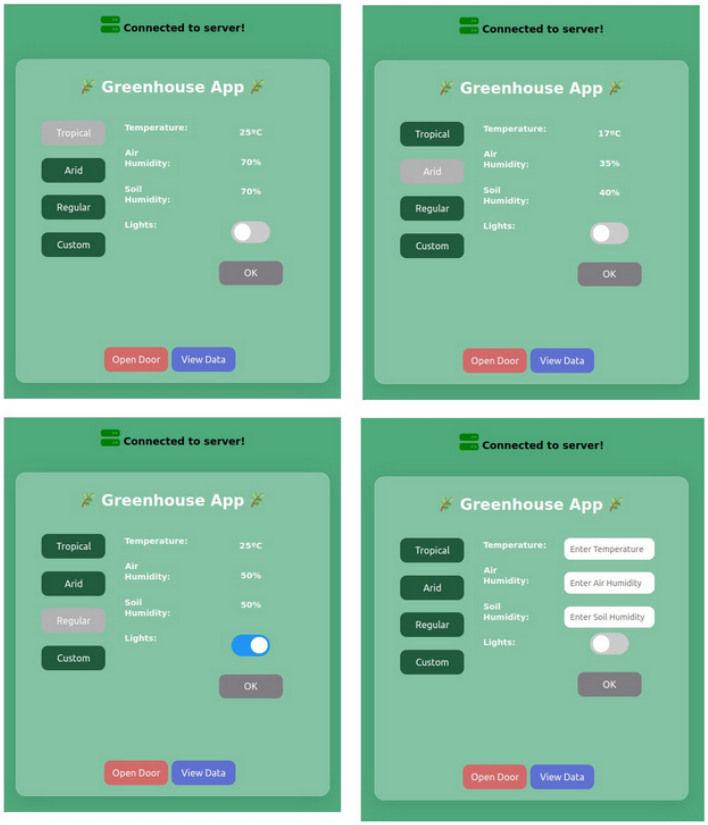

Figure 87: Environments Selection Layout

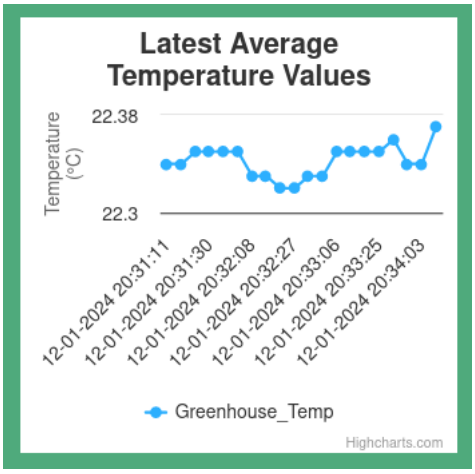

Figure 88: Temperature Graph

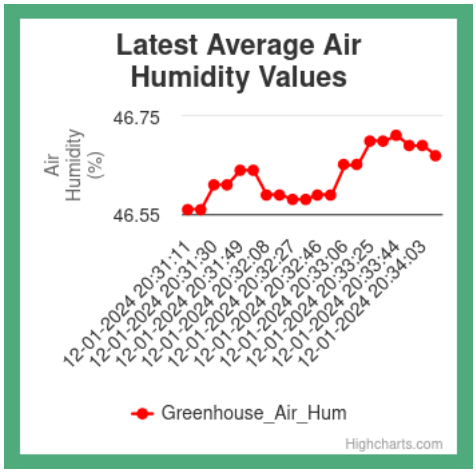

Figure 89: Air-Humidity Graph

Next, bidirectional real-time communication with the server was implemented using the ws library.

This library offers functions for managing events like open, message, and close. These functions enable dynamic transformation of the HTML, CSS, and JS application.

```
function connectWebSocket(){
  socket = new WebSocket(WS_URL);

  socket.addEventListener("open", handleSocketOpen);
  socket.addEventListener("error", handleSocketError);
  socket.addEventListener("message", handleSocketMsg);
  socket.addEventListener("close", handleSocketClose);
}

function handleSocketOpen(){
  //CONNECTED SYMBOL
  block_div('connected');
  hide_div('disconnected');

  console.log("WebSocket connected!");

  const dataJSON = JSON.stringify({
    action: ACTIONS.CONNECTION,
    type: "html"});

  socket.send(dataJSON);
}

function handleSocketError(error){
  console.error("WebSocket Error: ", error);
}

function handleSocketMsg(event){
  const data = JSON.parse(event.data);
  const action = data.action;

  switch(action){
    case ACTIONS.AUTHENTICATION:
    handleClientAuth(data.name);
    break;

    case ACTIONS.UPDATE_DATA:
    handleUpdateData(data.time, data.temperature,
   data.air_humidity, data.soil_humidity, data.light_sens,
   data.water);
    break;

    default:
    console.warn("Action Unknowned! -> ", action);
    break;
  }
}

function handleSocketClose(){

  //DISCONNECTED SYMBOL
  block_div('disconnected');
  hide_div('connected');


  console.log("WebSocket closed! Trying to reconnect in 5s...")
   ;
  setTimeout(connectWebSocket(), 5000);
}
```

Listing 5.20: WebSockets Handle Functions

The same principle of how the server works was applied to the web app, in which certain user actions or messages coming from the server can be filtered by handles and functions to perform the desired behavior.

```
function handleUpdateData(timestamp, sensor_temp, sensor_a_hum,
    sensor_s_hum, sensor_light, sensor_water){

  const time = timestamp;
  const temperature = parseFloat(sensor_temp.trim());
  const air_hum = parseFloat(sensor_a_hum);
  const soil_hum = parseFloat(sensor_s_hum);
  const water = parseFloat(sensor_water);

  //Update sensors arrays
  temp_data.data.push(temperature);
  temp_data.categories.push(time);

  a_hum_data.data.push(air_hum);
  a_hum_data.categories.push(time);

  s_hum_data.data.push(soil_hum);
  s_hum_data.categories.push(time);

  water_graph.push(water);

  //shift after reaching max_value
  if(water_graph.length > 1){
    water_graph.shift();
  }

  if(temp_data.data.length > MAX_SAMPLES){

    temp_data.data.shift();
    temp_data.categories.shift();

    a_hum_data.data.shift();
    a_hum_data.categories.shift();

    s_hum_data.data.shift();
    s_hum_data.categories.shift();
  }

  //LIGHTS
  var div1 = document.getElementById('on_signal');
  var div2 = document.getElementById('off_signal');

  if (sensor_light === "OFF") {
    div2.style.display = "block";
    div1.style.display = "none";
  } else {
    div2.style.display = "none";
    div1.style.display = "block";
  }


  //Error handling -> MAX TEMP
  if(temperature >= MAX_TEMP){
    console.log("WARNING MAX_TEMP REACHED!");
    block_div('max_temp');
  } else {
    hide_div('max_temp');
  }

```

```
59    //update data on GUI
60    viewData();
61 }
```

Listing 5.21: Update Sensor Data Function

Chapter 6

Conclusions & Future Work

In the next stage of the development of the automated greenhouse, the main objective would be to make the user experience even more engaging and practical. To do this, a signup system would be implemented in our application, allowing users to have more personalized control. The signup would be carried out in a system where different professionals would have different priorities.

Only employees with the highest priority would be able to add other employees and access certain forbidden parts of the application. In addition, users would be able to open the greenhouse door directly from the application. This feature will not only simplify access, but will also offer users greater convenience.

A chat feature that would allow all greenhouse employees to communicate with each other would also be something to consider in order to improve the experience and comfort for those working in the greenhouse. This would allow employees to exchange ideas and necessary tasks with each other via the application itself.

It's important to note that these improvements already started being developed. Both the door functionality and the signup functionality have already started to be integrated. The problems in implementing these features require more advanced functionality and greater security.

The project is therefore open to further innovations in order to improve the performance and efficiency as well as the experience of those working with the greenhouse.

The main aim of the project was to design and build an automated greenhouse that would be able to store and manage its data and also allow its users to access and set the different parameters in real time.

Throughout this project, we faced a number of significant challenges that allowed us to expand our knowledge in various areas, from database management to communication between systems. This work therefore required not only technical skills, but also creative thinking to overcome unexpected obstacles.

However, among the many difficulties faced, the integration between the local system and the remote system stood out as the main frontier to be overcome. By overcoming the difficulties in linking the local and remote systems, we not only won a technical victory but also gained knowledge in this area.

Throughout the development of this program, we were faced with the constant need to make crucial decisions and trade-offs that directly shaped the course of the project. Each choice made represented a delicate balance between different considerations, such as performance, simplicity, and resource efficiency, and so the ability to make informed choices and understand the trade-offs involved proved crucial to the program's success.

In conclusion, this challenging project not only tested our skills, but also provided a valuable learning opportunity. Overcoming the difficulties encountered not

only strengthened our knowledge base, but also equipped us with the necessary skills to tackle future complex projects.

Chapter 7

Task Division: Gantt Diagram

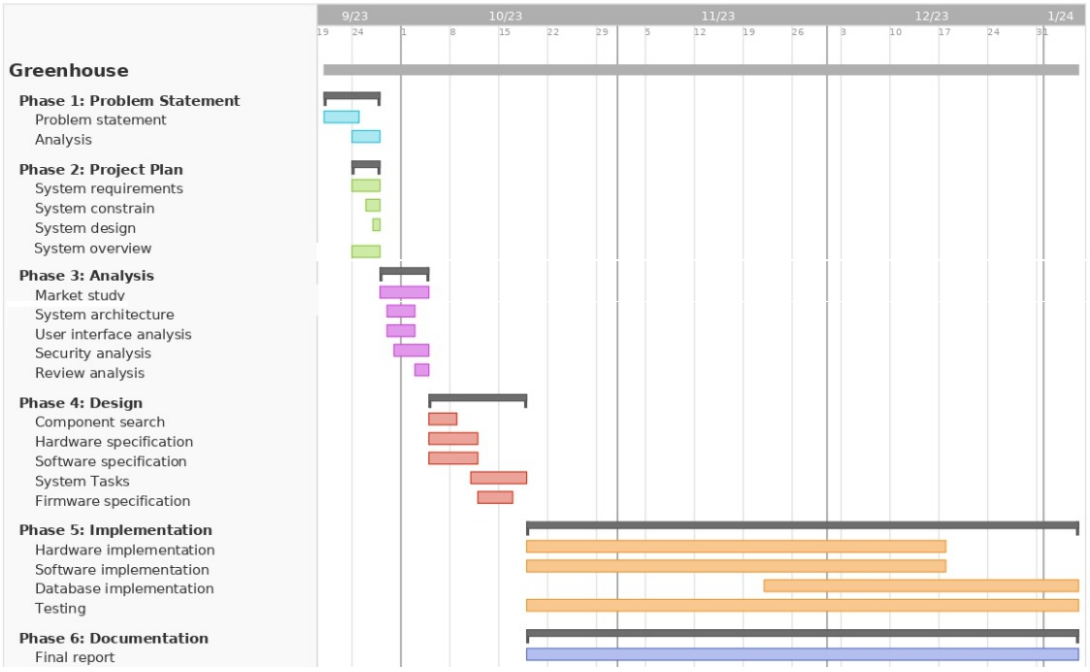

Figure 90: Gantt Diagram

Bibliography

[1] Mordor Intelligence. *Smart Greenhouse Market Size*. URL: https://www.mordorintelligence.com/industry-reports/smart-greeenhouse-market/market-size. (accessed: 01.10.2023).

[2] Bluelab. *Automated Greenhouse*. URL: https://bluelab.com/new_zealand/autogrow/shop-growing-environments/automated-greenhouse. (accessed: 01.10.2023).

[3] PC componentes. *Raspberry Pi 4B*. URL: https://www.pccomponentes.pt/raspberry-pi-4-modelo-b-2gb. (accessed: 03.10.2023).

[4] Raspberry Pi. *Raspberry Pi Documentation*. URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html. (accessed: 03.10.2023).

[5] Aquário Eletrónica. *Canvas Select Plus 32gb Microsdxc Uhs-I U3*. URL: https://www.aquario.pt/pt/product/kingston-technology-memoria-flash-32gb-canvas-select-plus-microsd-sin-sdcs2-32gbsp?gclid=Cj0KCQjw1aOpBhCOARIsACXYv-ca70GUDh5wRT_O-VRIxCDjFrUx0prNccABPIO0vrKnwdbmvqYXVXQaApG5EALw_wcB. (accessed: 03.10.2023).

[6] ElectroFun. *Sensor de Humidade e Temperatura Si7021 - SparkFun*. URL: https://www.electrofun.pt/sensores-arduino/sensor-humidade-temperatura-si7021-sparkfun?utm_campaign=efshopping&utm_source=google&utm_medium=cpc&utm_source=google&utm_medium=shopping&utm_campaign=roas&gclid=CjwKCAjwvrOpBhBdEiwAR58-3PrVFKE4sNk1_rmZf4bYzYQiAMH4sE_kJSQkSh4XunEgdfBkeDxiIhoCGjQQAvD_BwE. (accessed: 03.10.2023).

[7] Botn'Roll. *Capacitive Soil Humidity Sensor Module For Arduino - Keyestudio KS0510*. URL: https://www.botnroll.com/en/temperature/4771-capacitive-soil-humidity-sensor-module-for-arduino-keyestudio-ks0510.html. (accessed: 03.10.2023).

[8] Mauser. *SEEED Módulo Grove - Water Level Sensor*. URL: https://mauser.pt/catalog/product_info.php?products_id=096-8380&utm_source=google&utm_medium=cpc&utm_campaign=shopping_catalog_pt_011&utm_content=feed&gad_source=1&gclid=Cj0KCQjwy4KqBhD0ARIsAEbCt6hH8tEwLsWbTsAZHyDlEcCvcZOM81FqY9ZBBfY7B3Q4Wqqa0JzzHgoaAl9mEALw_wcB. (accessed: 03.10.2023).

[9] Botn'Roll. *LDR 5528 VT90N2 5MM - 10K 20K OHM*. URL: https://www.botnroll.com/en/luz-imagem/3617-vt90n2-ldr-1k-5mm.html. (accessed: 03.10.2023).

[10] Bot'nRoll. *Bomba de Água R385 6 12Vdc 0.5 0.7A*. URL: https://www.botnroll.com/pt/motores-dc/3164-bomba-de-gua-r385-6-12vdc-0-5-0-7a.html. (accessed: 04.10.2023).

[11] PT Robotics. *DC Brushless Fan 80x80x20mm 12V 74mA Sunon*. URL: https://www.ptrobotics.com/ventoinhas/4544-dc-brushless-fan-80x80x20mm-12v-74ma-sunon.html. (accessed: 04.10.2023).

[12] Bot'nRoll. *Resistência de Aquecimento 5W - 10cm x 5cm*. URL: https://www.botnroll.com/pt/resistencias/2000-resistencia-de-aquecimento-5w-10cm-x-5cm-.html. (accessed: 04.10.2023).

[13] AUTUUCKEE. *AUTUUCKEE LED Planta Grow Light, 2835SMD impermeável Full Spectrum LED*. URL: https://www.amazon.es/AUTUUCKEE-impermeable-espectro-invernadero-hidrop%C3%B3nica/dp/B0962S3JNW/ref=sr_1_3?crid=KSTDLWP3MZXE&keywords=full%2Bspectrum%2Bled&qid=1697490193&sprefix=full%2Bsp%2Caps%2C106&sr=8-3&th=1. (accessed: 04.10.2023).

[14] QWORK. *QWORK 50 unidades aspersores de irrigação, aspersores terraço, bicos nebulizadores de água com conector de tubo de água para irrigação e refrigeração*. URL: https://www.amazon.es/-/pt/dp/B0B8J38SXT/ref=sr_1_14?crid=3O8ZO0S6YOW0K&keywords=nebuliza%C3%A7%C3%A3o+aspersor&qid=1697504973&s=lawn-garden&sprefix=nebuliza%C3%A7%C3%A3o+aspersor%2Clawngarden%2C94&sr=1-14. (accessed: 04.10.2023).

[15] Botn'Roll. *Electroíman 5V 3Kg força retenção*. URL: https://www.botnroll.com/pt/solenoides/4361-electro-man-5v-3kg-for-a-reten-o.html. (accessed: 04.10.2023).

[16] Botn'Roll. *Módulo 8 Relés 5V Opto-Isolados*. URL: https://www.botnroll.com/en/digital/3394-8-channel-5v-relay-module-opto-isolated.html. (accessed: 04.10.2023).

[17] elektor. *Official EU Power Supply for Raspberry Pi 4 (black)*. URL: https://www.elektor.com/official-eu-power-supply-for-raspberry-pi-4-black. (accessed: 04.10.2023).

[18] Mauser. *Fonte de alimentação industrial 12VDC 5A 60W - Orno OR-ZL-1633*. URL: https://mauser.pt/catalog/product_info.php?products_id=035-0873&utm_source=google&utm_medium=cpc&utm_campaign=shopping_catalog_pt_011&utm_content=feed&gclid=Cj0KCQjw4bipBhCyARIsAFsieCxe6NAL7LYKM-LVaa0RuzkbJUF6dhxhKhXKTQs5L5XQ75_QUoNrHiwaAlHxEALw_wcB. (accessed: 04.10.2023).

[19] Techtarget. *What is a Kernel? Types of Kernels*. URL: https://www.techtarget.com/searchdatacenter/definition/kernel. (accessed: 24.10.2023).

[20] Keil. *CMSIS-RTOS Mutex*. URL: https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__MutexMgmt.html. (accessed: 24.10.2023).

[21] Devopedia. *Linux Signals*. URL: https://devopedia.org/linux-signals. (accessed: 24.10.2023).

[22] Keil. *CMSIS-RTOS Message Queue*. URL: https://www.keil.com/pack/doc/cmsis/RTOS/html/group__CMSIS__RTOS__Message.html. (accessed: 24.10.2023).

[23] Philips Semiconductors. *Philips Semiconductors I^2C Handbook*. 2004.