# SMART LIGHTING AND HEATING SYSTEM – THIRD DELIVERY

André Filipe Almeida Gonçalves 1210804
Jorge Rafael Novais Moreira 1201458

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Departamento de Engenharia Informática

Instituto Superior de Engenharia do Porto

2024

This report meets the requirements outlined in the Course Unit Description of the Critical Systems Laboratory, in the 1st year of the Master's in Critical Computer Systems Engineering.

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Departamento de Engenharia Informática

Instituto Superior de Engenharia do Porto

9 de janeiro de 2025

# Conteúdo

# 1. INTRODUCTION

This document outlines the comprehensive development process of the selected project for the course *Laboratório de Sistemas Críticos*. It provides a detailed analysis of the problem context, presenting a clear rationale for the selection of components and technologies employed in the project's implementation. Furthermore, it elaborates on the development and analysis of the real-time system. Additionally, the design and functionality of the user interface are thoroughly described, showcasing its role in enhancing usability and system interaction.

## 1.1. Context

In modern buildings like the one housing the CISTER unit, smart management of lighting and heating is essential for promoting environmentally friendly practices, reducing operational costs, and enhancing the well-being of researchers. This project aims to develop an automated climate and lighting control system tailored to the unique characteristics of the CISTER building, particularly focusing on its 3rd floor.

The floor comprises seven individual offices, each equipped with one smart blind and one smart heater, four double-occupancy offices with two independent smart blinds and heaters each, and the director's office, which features three smart blinds and four smart heaters. The building layout is illustrated in the accompanying diagram.

All offices are equipped with smart devices, including blinds represented in green, heaters in red, and ceiling lights in orange, which can be monitored and controlled via wireless sensor networks (WSN) as shown in the Figure 1 .

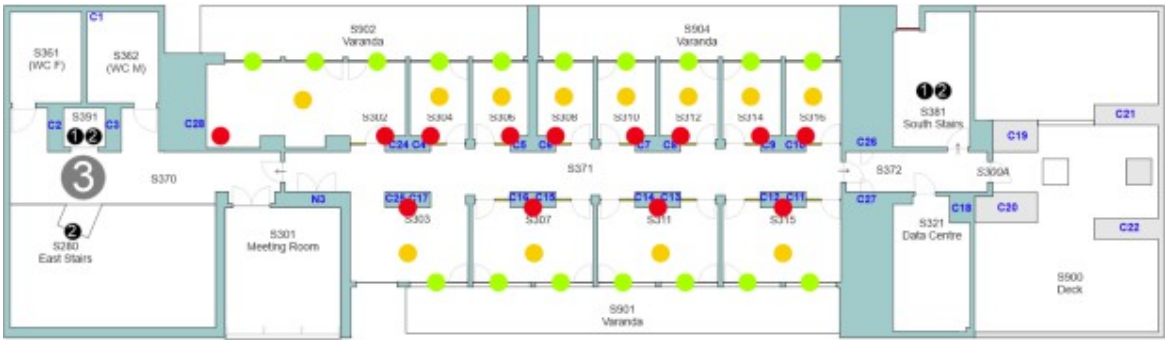


*Figure 1- Floor 3 of CISTER building.*

## 1.2. OBJECTIVES

The focus of this project is to automate the lighting and temperature control on the 3rd floor of the CISTER building, specifically targeting the individual offices due to the variety of office types present on the floor.

The first task involves automating the control of office lighting. Priority is given to natural sunlight, with the blinds automatically adjusting to maximize natural light entry. When natural light is insufficient, artificial lighting will compensate to maintain the desired illumination level.

The second task addresses the control of heating through the smart heater installed in the office.

Additionally, the envisioned automated climate control system will include a user interface that allows office occupants to configure their preferences for lighting and temperature, as well as manually adjust the blinds, lights, and heater. The system will also notify occupants if their preferences cannot be met, potentially indicating an operational issue.

The objective is to design and implement a prototype system that fulfils these requirements. Due to limited access to the building (in line with current government guidelines) and the potential risks of interacting with real infrastructure—such as damaging the blinds, which could result in costly repairs or user inconvenience—the implementation will primarily rely on simulated behaviours. However, integrating physical sensors or actuators, such as representing blind movements with different LED colours, is strongly encouraged. When such actuators are included, their status (enabled/disabled) must be clearly indicated.

# 2. DESIGN

This chapter outlines the understanding of the problem and the design of the solution that will guide the project's development.

## 2.1. PROBLEM OVERVIEW

The project involves managing various actuators based on environmental parameters. Since temperature and light intensity are the key factors, it is essential to use sensors capable of accurately measuring these values.

### 2.1.1. Components Used

| Component | What we would like to use | What we will be using |
|---|---|---|
| Temperature Sensor | DHT11 | DHT11 |
| Luminosity Sensor | TSL2561 | LDR5539 |
| Blinds Mechanism | SG90 | SG90 |
| Heater Mechanism | Heating Plate | LED |
| Smart Lights | LED | LED |

*Table 1- Components*

As mentioned above, there are alternative components we would prefer to use instead of the ones currently chosen. We will now explain the reasons for selecting these components and explain why we would choose some others over the ones initially planned.

**Temperature Sensor:**

We have decided to keep the DHT11 sensor for temperature measurement in our project. The DHT11 operates within a voltage range of 3V to 5.5V, and we will power it using a 3.3V supply. Since the ESP32 operates at 3.3V logic, we will handle the level shifting for the data line to ensure proper communication between the ESP32 and the DHT11 sensor. Despite its limitations in accuracy compared to other sensors like the DHT22, the DHT11 is sufficient for our project’s requirements, providing reasonably accurate temperature readings within the desired range while keeping the project cost-effective.

The DHT11 will be connected to **pin 18** of the ESP32 and is mounted securely on a breadboard for easy prototyping and wiring. Its three pins (VCC, GND, and DATA) are connected as follows:

- **VCC** is connected to the 3.3V power rail of the ESP32.
- **GND** is connected to the ground rail.
- **DATA** is connected to ESP32 **pin 18**.

This setup ensures reliable temperature readings while keeping the design simple and robust.

**Luminosity Sensor:**

The TSL2561 is a digital light sensor capable of measuring illuminance with higher precision and a wider dynamic range, making it suitable for applications requiring accurate light intensity measurement in varying conditions. Despite its advantages, the TSL2561 requires more complex interfacing and is generally more expensive than analogue alternatives.

For this project, we have chosen the LDR5539, a light-dependent resistor (LDR), which, while less precise and lacking the dynamic range of the TSL2561, is both cost-effective and simple to integrate. The LDR5539 provides adequate performance for our requirements by detecting light intensity changes effectively. Additionally, its straightforward analogue output simplifies the circuit design and reduces the need for additional hardware or complex programming, aligning well with our budget constraints and project scope.

The LDR is mounted on the breadboard and forms a voltage divider circuit with a 10kΩ resistor. This configuration allows the ESP32 to measure the changing resistance of the LDR in response to light levels by reading the corresponding voltage change. The centre of the voltage divider is connected to **pin 34**, an analogue input pin on the ESP32.

The connections are as follows:

- One end of the LDR is connected to the **3.3V power rail.**
- The other end of the LDR is connected to one side of the 10kΩ resistor.
- The opposite side of the resistor is connected to **GND.**
- The centre tap of the voltage divider (between the LDR and resistor) is connected to ESP32 **pin 34**.

**Blinds Sensor:**

We have chosen to use the SG90 motor for the blind's mechanism, as it is a low-cost, widely available, and reliable option for small projects. It offers sufficient torque and precision for controlling small blinds, and it is easy to integrate with microcontrollers like Arduino. In our project, we will use the SG90 motor to simulate the blinds mechanism by moving it forward to open the blinds, which will trigger one LED to light up, and moving it backward to close the blinds, which will light up a different LED. This setup will simulate the opening and closing of blinds.

The SG90 servo motor is powered directly from the ESP32, which provides adequate power for this low-torque motor. The connections are as follows:

- **VCC** is connected to the **3.3V power rail.**
- **GND** is connected to the **ground rail**.
- **Control signal** is connected to ESP32 **pin 19.**

**Heater Mechanism:**

We initially planned to use a heating plate to control temperature more directly. However, due to power constraints and the nature of the project, we decided to substitute the heating plate with an LED, which will allow us to simulate the heating process more efficiently without requiring additional power resources. In our simulation, using an LED is simpler because it provides a straightforward way to visually represent the heating process without the complexity of managing actual temperature control. The LED will be turned on to simulate heating, offering a clear and easy-to-implement solution for our needs.

The LED is mounted on the breadboard and connected in series with a 220Ω resistor to limit the current and protect the LED. The ESP32 toggles the LED on or off to simulate the heater being activated or deactivated.
The connections are as follows:

- The positive leg (anode) of the LED is connected to **pin 4** of the ESP32.
- The negative leg (cathode) is connected to one side of the 220Ω resistor.
- The other side of the resistor is connected to **GND**.

**Smart Lights:**

We have chosen an LED because it fits the functional requirements of the smart lights in our project. The LED provides a simple and effective way to simulate the behaviour of a real smart light. When the ambient brightness, measured by the LDR5539, falls below a defined threshold, the LED automatically lights up, mimicking how smart lights respond to low light conditions.

The system dynamically monitors light intensity through the LDR5539 sensor, connected to an analogue input pin of the ESP32. The ESP32 processes the sensor readings and compares them to the predefined threshold. If the measured brightness is lower than the threshold, the ESP32 sends a signal to activate the LED.

The LED is mounted on the breadboard and connected in series with a 220Ω resistor to limit the current and protect the LED. The ESP32 toggles the LED on or off to simulate the heater being activated or deactivated.
The connections are as follows:

- The positive leg (anode) of the LED is connected to **pin 5** of the ESP32.
- The negative leg (cathode) is connected to one side of the 220Ω resistor.
- The other side of the resistor is connected to **GND**.

### 2.1.2. Technologies

| Technologies |
| --- |
| MQTT |
| TCP/IP |
| NODE-RED |
| ARDUINO (ESP32) |

*Table 2 – Technologies*

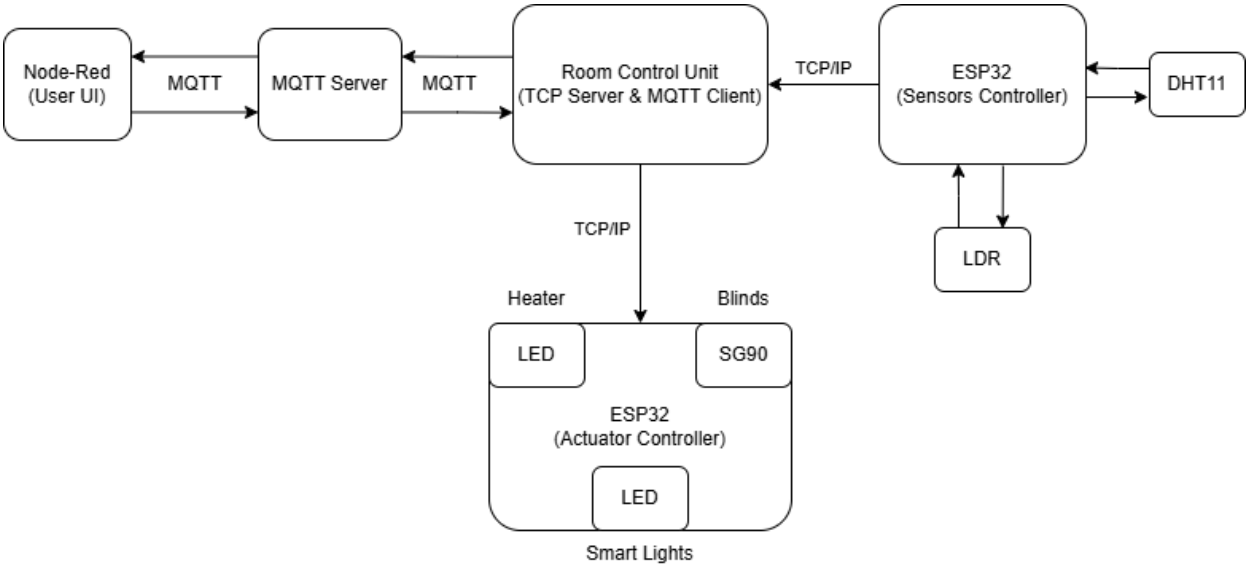


*Figure 2 - Communication scheme.*

**Arduino:**

The system incorporates two Arduino boards, each serving a distinct role to ensure modularity and efficiency. The Sensor Arduino is tasked with collecting data from the DHT11 temperature and humidity sensor and the LDR5539 light-dependent resistor. This Arduino processes the sensor data and transmits it to the TCP server for validation and further handling. Initially, the data is sent in an unstructured format tailored to the project's immediate needs. Although Smart Data Models are not yet implemented, they are planned for future integration to standardize the data format and enhance compatibility with other systems.

The Actuator Arduino, on the other hand, manages the system's actuators, including the SG90 servo motor and LEDs. It receives validated commands from the TCP server and executes actions such as adjusting the blinds mechanism, turning on or off the simulated smart lights, or activating the heating simulation. Both Arduinos connect to the same Wi-Fi network, enabling reliable communication with the TCP server and ensuring smooth operation within the system's architecture.

**MQTT:**

MQTT is planned for use in the future to facilitate communication between the control unit (TCP server) and the user interface, which is built using Node-RED. Once developed, MQTT will allow the control unit to publish sensor data received from the Sensor Arduino to an MQTT broker, making it accessible to Node-RED for real-time visualization and monitoring. Additionally, MQTT will be used to send user commands from Node-RED to the TCP server, which will relay them to the Actuator Arduino. This bi-directional communication will help streamline integration and data flow across the system.

At present, MQTT is still in the development phase, and data transmission is handled using basic unstructured communication between the components. Smart Data Models will later be incorporated to ensure standardized data formatting for MQTT communication.

**TCP/IP:**

TCP/IP is the primary protocol used for communication between system components, ensuring reliable and connection-oriented data exchange. The Sensor Arduino sends sensor readings to the TCP server over TCP/IP, where the data is validated and processed. The server also sends commands to the Actuator Arduino through the same protocol, ensuring that actuator actions are carried out without any loss of data. This reliable communication protocol is essential for maintaining the integrity of the system.

**NODE-RED:**

While Smart Data Models are not yet implemented, they are an important planned feature for the system. These models will provide standardized data formats for sensor readings and actuator commands, ensuring consistent communication across components. Once integrated, Smart Data Models will improve the system's compatibility with third-party IoT platforms and streamline data management. The Sensor Arduino will format its sensor data according to these models before transmitting it to the server, and user commands from Node-RED will also be structured accordingly. This future enhancement will ensure the scalability and interoperability of the system.

This combination of technologies ensures that the project is built on a solid foundation, with immediate functionality in place and a roadmap for future improvements as seen in the Figure 2.

**Smart Data Model:**

## 2.2. Problem Domain Design

In this section, it will be discussed the problem domain design created to answer the problem posed in this assignment.

### 2.2.1. Black Box

First, we identified the Stakeholder Needs, the requirements that the stakeholders have for the assignment. The identified Stakeholder Needs were:

| | id | text |
|---|---|---|
| User Needs | SN-1 | |
| Automated Lighting Control | SN-1.1 | When natural sunlight is available, the system shall adjust blinds to allow the required amount of light.<br>If natural sunlight is insufficient, the system shall activate artificial lighting to compensate. |
| Natural Light Adjustment | SN-1.1.1 | When natural sunlight is available, the system shall adjust blinds to allow the required amount of light. |
| Artificial Light Compensation | SN-1.1.2 | When natural sunlight is insufficient, the system shall activate artificial lighting to compensate. |
| Automated Heating Control | SN-1.2 | The system shall control the heating in the office using smart heaters. |
| User Interface | SN-1.3 | The system shall provide a user interface that allows office occupants to configure their preferences for lighting and temperature, and manually change the status of blinds, lights, and heaters. |
| Hazard Mitigation | SN-1.4 | The system shall include measures to prevent or mitigate failures by identifying malfunctioning components, providing temporary fixes, and providing long-term fixes. |
| Notification System | SN-1.5 | If the occupant's preferences are not being met, the system shall notify the occupant and indicate a potential problem with the system. |
| WSN Network | SN-1.6 | Where possible, the system shall utilize Wireless Sensor Networks (WSNs) for communication and control. |

*Figure 3 Stakeholder Needs*

The next step was creating the system context, where we represented the system needed and the interaction between these systems.

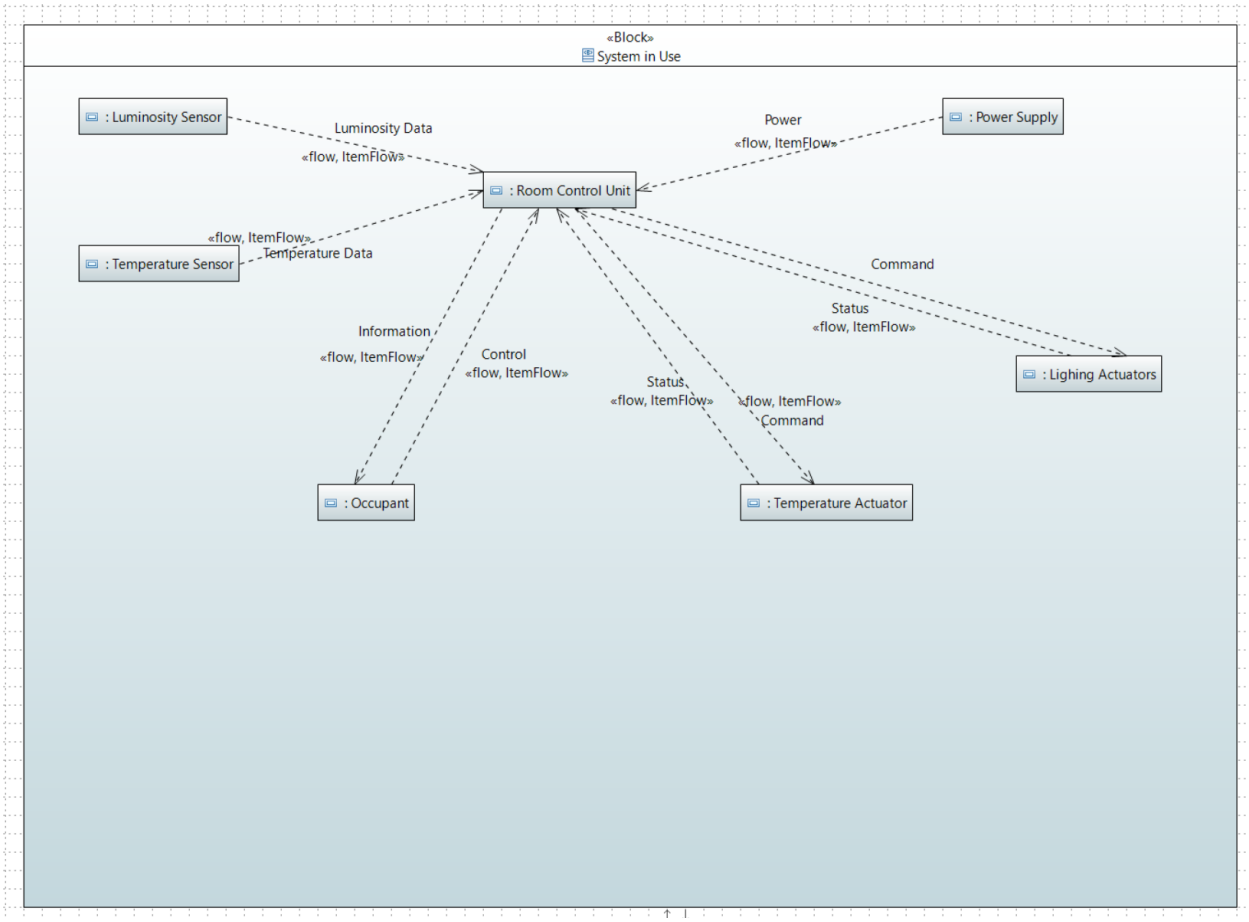


*Figure 4 System Context*

Following the creation of the system context, were created a use case diagram for the use cases that we found.

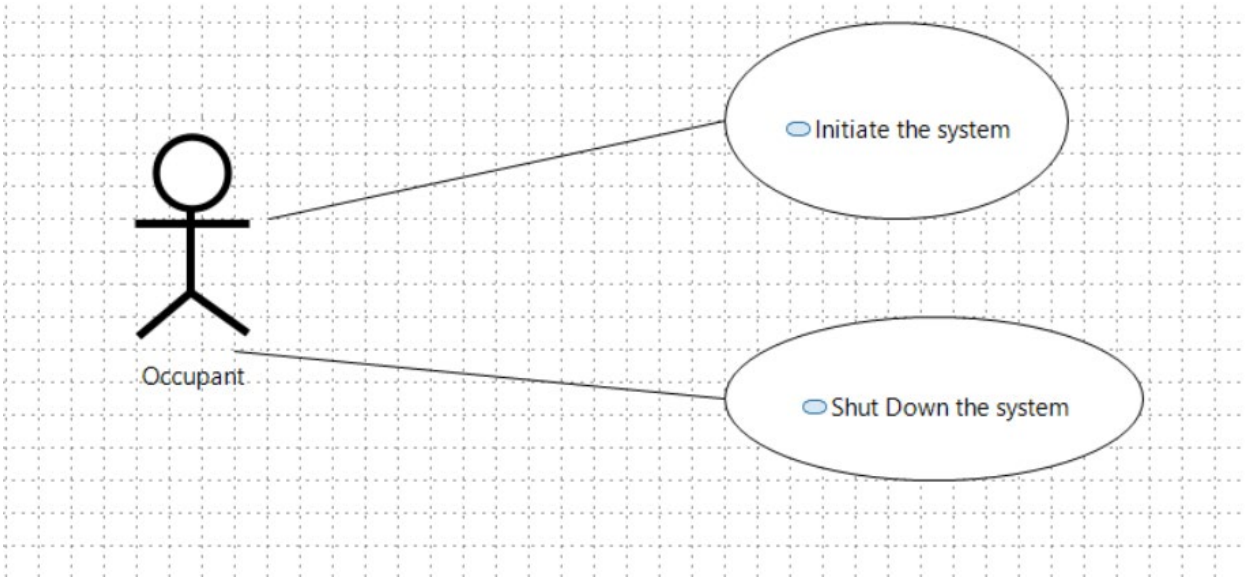


*Figure 5 Use Cases*

Using the use cases that we found, we created an activity diagram for these use cases, showing the flow of the systems during execution, and further explaining what happens between these systems.

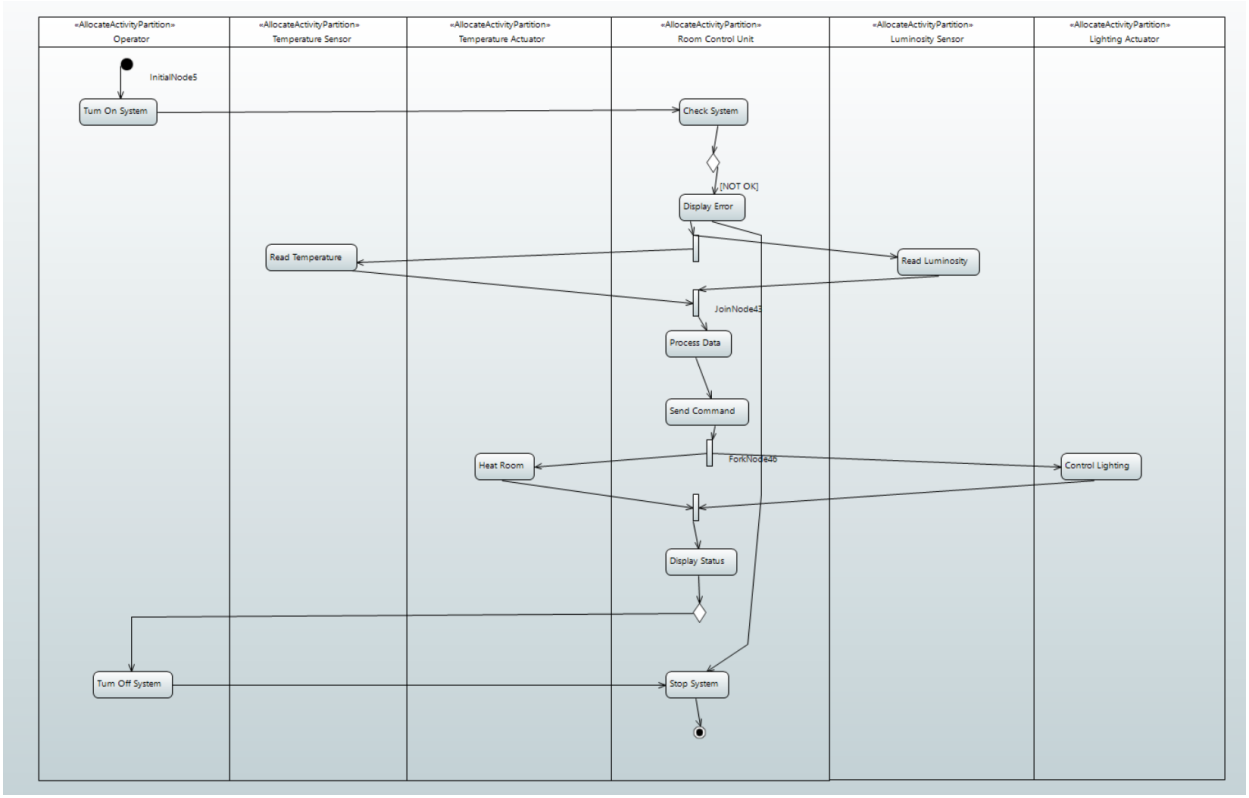


*Figure 6 Activity Diagram*

To finish the black box design, the measures of effectiveness, the metric that ensure the system is working correctly, are created.

We measure the power needed for the system to work and the response time between the system receiving the data and sending the proper command.

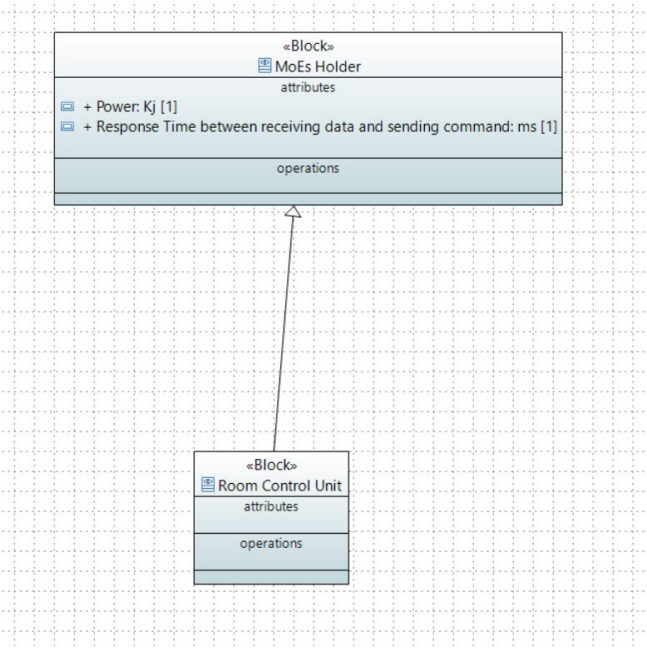


*Figure 7 Measures of Effectiveness*

### 2.2.2. White Box

The first thing done for the white box model was figure out what were the conceptual systems that existed. The following figure is the representation of these conceptual systems.

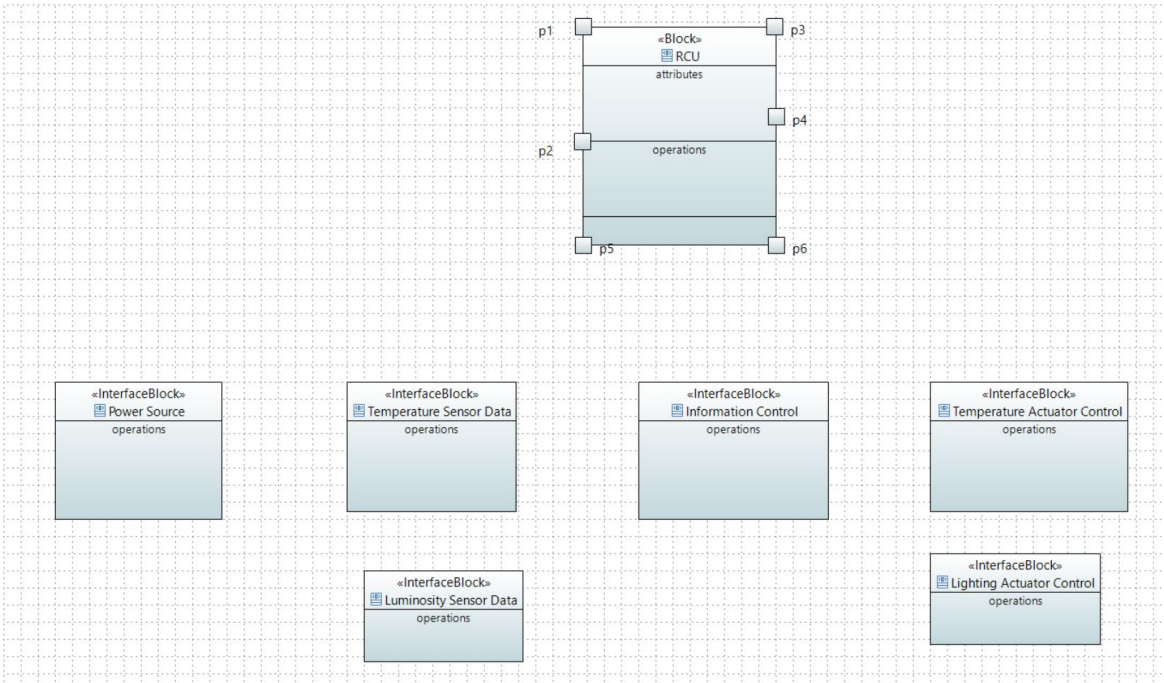


*Figure 8 Conceptual Systems*

The next step of the modelling process was to represent the conceptual subsystems envisioned for the room control system and the interaction between these conceptual subsystems and with the outside systems.

To figure out what were the subsystems needed for the system, we did a functional analysis of said system, being the main functionalities of the system reading the data from the sensors, processing said data, sending commands to the temperature and lighting actuators, and interacting with the user.

So, we used these functionalities to create each subsystem.

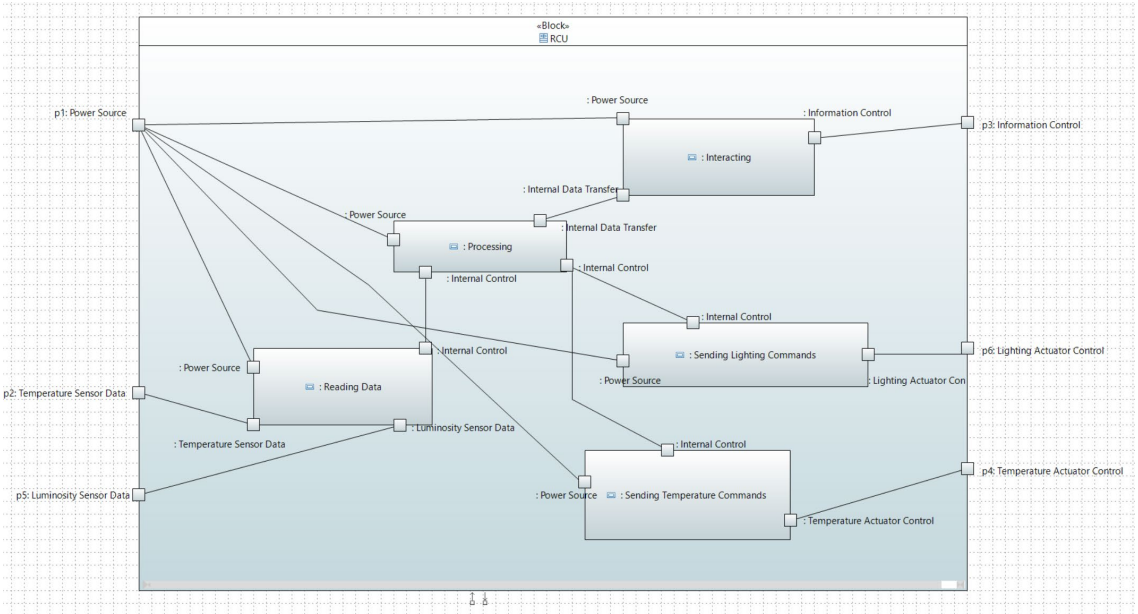


*Figure 9 Conceptual Subsystems*

Like in the black box model we created an activity diagram to represent the flow of execution of these conceptual subsystems.

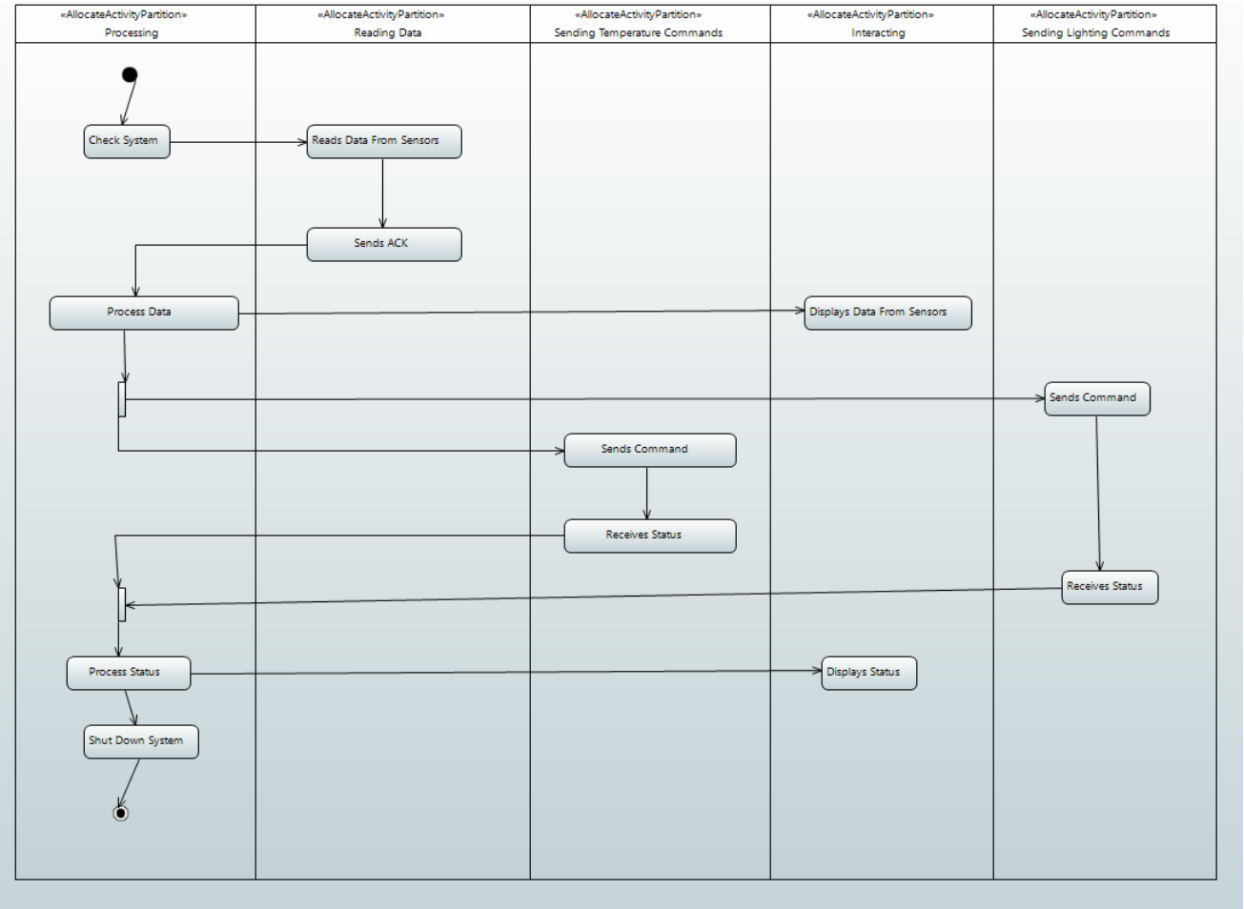


*Figure 10 Conceptual Subsystems Activity Diagram*

Lastly, we represented the measure of effectiveness used by these subsystems.

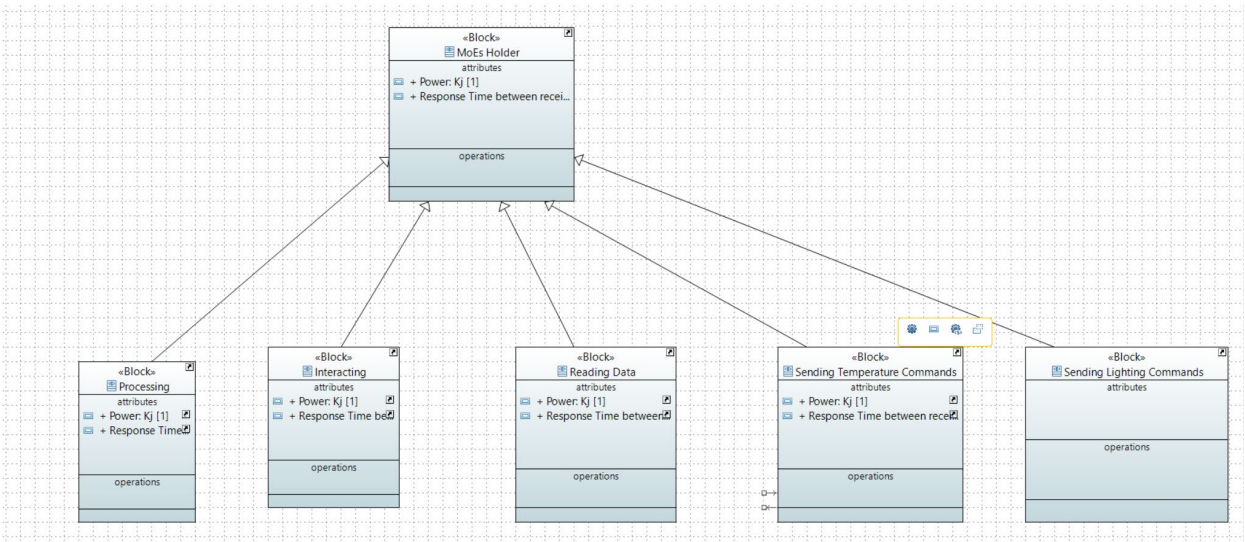


*Figure 11 Measures of Effectiveness for the Conceptual Systems*

### 2.2.3. Traceability

The last part of the problem domain is the traceability between the Stakeholder Needs and the systems that are represented in the system context, explaining what each system does in relation to the Stakeholder Needs, and subsystems representing also the relation between these

subsystems and the Stakeholder Needs.

| | | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|
| | | Automated Lighting Control | Automated Heating Control | User Interface | Hazard Mitigation | Notification System | WSN Network |
| 0 | System in Use | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | Property1 : Room Control Unit | ☑ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 3 | Property2 : Luminosity Sensor | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |
| 4 | Property3 : Lighing Actuators | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |
| 5 | Property4 : Power Supply | ☑ | ☑ | ☑ | ☑ | ☑ | ☑ |
| 6 | Property5 : Occupant | ☐ | ☐ | ☑ | ☐ | ☑ | ☐ |
| 7 | Property6 : Temperature Sensor | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 8 | Property7 : Temperature Actua... | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 9 | Sending Temperature Commands | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 10 | Reading Data | ☑ | ☑ | ☐ | ☑ | ☐ | ☐ |
| 11 | Interacting | ☐ | ☐ | ☑ | ☐ | ☑ | ☐ |
| 12 | Processing | ☑ | ☑ | ☐ | ☑ | ☐ | ☑ |
| 13 | Sending Lighting Commands | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |

*Figure 12 Traceability*

## 2.3. Solution Domain Model

In this section, it will be discussed the solution domain design created to answer the problem posed in this assignment.

### 2.3.1. System Requirements

The system requirements are directly correlated to the Stakeholder Needs represented in the problem domain section. These requirements are the minimum applications that the system needs to function correctly as envisioned by the stakeholders.

| | | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|
| | | Natural Light Adjustment | Artificial Light Compensation | Heating Control | Preference Configuration | Preference Notification | Failure Prevention | Temporary Fix | Wireless Sensor Network |
| 0 | Subsystem Requir... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | User Interface | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 3 | Feedback Display | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☑ |
| 4 | User Input Vali... | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☐ |
| 5 | Interface Availa... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |
| 6 | Error Notification | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☑ |
| 7 | Real-time Inter... | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |

*Figure 13 System Requirements*

### 2.3.2. High Level Architecture

After the requirements, the high-level architecture was modelled, where all the subsystems that describe the system, room control, are captured. The subsystems are as follows, the data reading system, processing system, command system and user interacting system.

In comparison with the conceptual subsystem in the problem domain, the command system is a junction of the two subsystems created for sending commands for each type of actuators.

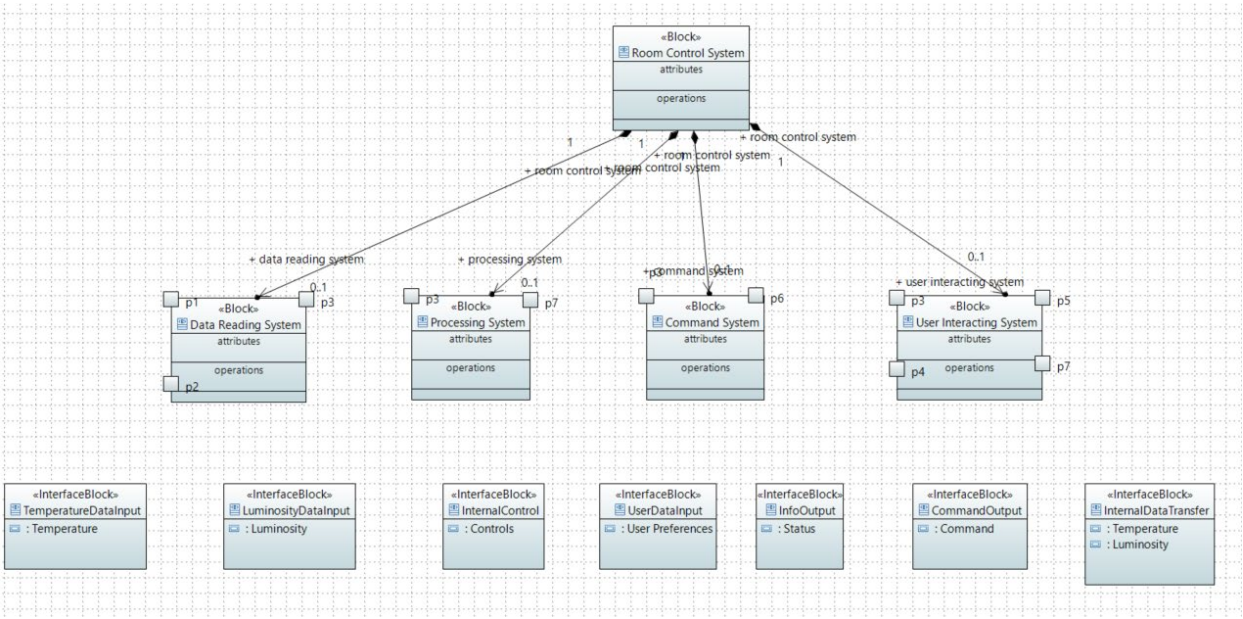


*Figure 14 High Level Architecture*

### 2.3.3. Subsystem Requirements

The subsystem requirements specify the requirements for the individual subsystems within the larger system. These requirements outline what is necessary for a part of the system to function properly and integrate seamlessly with the rest of the system.

As we have four subsystems, we will have one table for each subsystem.

| | id | text |
|---|---|---|
| Subsystem Requ... | SSR.1 | |
| Data Processing | SSR-1.1 | The subsystem shall process data from sensors, user inputs, and external systems to control the office environment. |
| Decision-making | SSR-1.2 | The subsystem shall make decisions based on predefined rules and occupant preferences to optimize office conditions. |
| Data Integrity | SSR-1.3 | The subsystem shall validate incoming data to ensure it is free from corruption or malicious interference. |
| Fault Isolation | SSR-1.4 | If a fault occurs in the processing subsystem, it shall isolate the fault to prevent impact on other subsystems. |
| Safe State | SSR-1.5 | If the subsystem fails, it shall transition the system into a safe state to avoid harm to occupants or equipment. |
| Error Handling | SSR-1.6 | The subsystem shall detect errors during operation and log them for diagnosis while attempting to recover automatically. |

*Figure 15 Processing System Requirements*

The following table represents the subsystem requirements for the data reading system.

| | id | text |
|---|---|---|
| Subsystem Requ... | SSR-1 | |
| Sensor Data C... | SSR-1.1 | The subsystem shall collect data from all connected sensors, including lighting, temperature, and occupancy sensors. |
| Data Accuracy | SSR-1.2 | The subsystem shall ensure that sensor data is accurate to within the tolerances specified by the sensor manufacturer. |
| Real-time Data | SSR-1.3 | The subsystem shall read data from all sensors at intervals of no more than one second. |
| Sensor Validati... | SSR-1.4 | The subsystem shall validate sensor readings to detect anomalies that may indicate sensor malfunction or tampering. |
| Data Overload ... | SSR-1.5 | The subsystem shall implement safeguards to prevent data overload from impairing system performance. |
| Safe Failure | SSR-1.6 | If a sensor fails to provide data, the read data subsystem shall notify the processing subsystem and use the last known good value as a fallback. |
| Error Logging | SSR-1.7 | The subsystem shall log errors, including missing or out-of-range sensor data, for further analysis. |

*Figure 16 Data Reading System Requirements*

The last tables represent, respectively, the command system requirements and the user interacting system requirements.

| | id | text |
|---|---|---|
| Subsystem Requ... | SSR-1 | |
| Command Tra... | SSR-1.1 | The subsystem shall transmit commands to the actuators, including blinds, lights, and heaters, based on instructions from the processing subsystem. |
| Real-time Com... | SSR-1.2 | The subsystem shall ensure that all commands are transmitted to the actuators within 100 milliseconds of receiving instructions from the processing subsystem. |
| Command Con... | SSR-1.3 | The command subsystem shall confirm the successful execution of commands by receiving status updates from actuators and sending these confirmations to the processing subsystem. |
| Failure Notifica... | SSR-1.4 | If an actuator does not respond to a command within a specified time frame, the command subsystem shall notify the processing subsystem of the failure. |
| Fallback State | SSR-1.5 | If an actuator fails to execute a command, the command subsystem shall revert the actuator to its last known safe state. |

*Figure 17 Command System Requirements*

| | id | text |
|---|---|---|
| Subsystem Requ... | SSR-1 | |
| User Interface | SSR-1.1 | The subsystem shall provide a user interface that allows occupants to configure preferences for lighting, temperature, and blinds. |
| Feedback Disp... | SSR-1.2 | The subsystem shall display feedback on the current system status, including lighting, temperature, and actuator conditions. |
| User Input Vali... | SSR-1.3 | The subsystem shall validate user inputs to ensure they are within acceptable operational ranges before transmitting them to other subsystems. |
| Interface Avail... | SSR-1.4 | The human interacting subsystem shall ensure the user interface remains accessible even during partial system failures. |
| Error Notification | SSR-1.5 | The subsystem shall notify occupants of errors, such as actuator malfunctions or unmet preferences, and provide suggested actions. |
| Real-time Inte... | SSR-1.6 | The subsystem shall process user commands and provide feedback within one second of interaction. |

*Figure 18 User Interacting System Requirements*

### 2.3.4. Traceability

Like it was done in the previous subsection, traceability was done for each set of requirements of each subsystem with the system requirements, ensuring that each subsystem requirement can be traced to one or more system requirements.

The following matrices were created to help visualize the traceability between each set of subsystem requirements and the system requirements.

| | | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|
| | | Natural Light Adjustment | Artificial Light Compensation | Heating Control | Preference Configuration | Preference Notification | Failure Prevention | Temporary Fix | Wireless Sensor Network |
| 0 | Subsystem Requir... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | Data Processing | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 3 | Decision-making | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 4 | Data Integrity | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |
| 5 | Fault Isolation | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |
| 6 | Safe State | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ |
| 7 | Error Handling | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ |

*Figure 19 Processing Subsystem Requirements to System Requirements*

| | | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|
| | | Natural Light Adjustment | Artificial Light Compensation | Heating Control | Preference Configuration | Preference Notification | Failure Prevention | Temporary Fix | Wireless Senso |
| 0 | Subsystem Requir... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | Sensor Data Col... | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 3 | Data Accuracy | ☐ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 4 | Real-time Data | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |
| 5 | Sensor Validation | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ |
| 6 | Data Overload P... | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ |
| 7 | Safe Failure | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☑ | ☐ |
| 8 | Error Logging | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ |

*Figure 20 Data Reading Requirements to System Requirements*

| | | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|
| | | Natural Light Adjustment | Artificial Light Compensation | Heating Control | Preference Configuration | Preference Notification | Failure Prevention | Temporary Fix | Wireless Senso |
| 0 | Subsystem Requir... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | Command Trans... | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |
| 3 | Real-time Com... | ☑ | ☑ | ☑ | ☐ | ☐ | ☐ | ☐ | ☑ |
| 4 | Command Confi... | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☑ |
| 5 | Failure Notificat... | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ |
| 6 | Fallback State | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |

*Figure 21 Command Requirements to System Requirements*

| | | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|
| | | Natural Light Adjustment | Artificial Light Compensation | Heating Control | Preference Configuration | Preference Notification | Failure Prevention | Temporary Fix | Wireless Sensor Network |
| 0 | Subsystem Requir... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| 2 | User Interface | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |
| 3 | Feedback Display | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☑ |
| 4 | User Input Vali... | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☐ |
| 5 | Interface Availa... | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |
| 6 | Error Notification | ☐ | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ | ☑ |
| 7 | Real-time Inter... | ☐ | ☐ | ☐ | ☑ | ☐ | ☐ | ☐ | ☑ |

*Figure 22 User Interacting Requirements to System Requirements*

## 2.4. Safety and Reliability Analysis

In terms of safety and reliability there were some additions that had in mind the creation of a system which was both safe and reliable.

Not only the addition of requirements like hazard mitigation in the Stakeholder Needs, or the

fallback state in the command subsystem requirements allow the system to be more reliable, limiting the fails and its damages, but also the addition of checks when starting the system allows the better execution of said system.

With the addition of error handling, we allow the user to, in real time, check the system and its execution, and in case of error the occupant can promptly verify the system and correct any errors or failures in said system.

# 3. Development

This section outlines the development process of the project, which integrates sensor readings, actuator execution, server communication, and low-level assembly code to perform led control.

## 3.1. ESP32 Sensor Reading

The ESP32 collects data from various sensors, including a DHT11 temperature sensor and an LDR (Light Dependent Resistor) and then sends the data retrieved from the sensors to the server TCP as seen in Figure 23.

```
// Read sensors & Measure sensor reading time
  unsigned long startSensorRead = micros();
  float temperature = dht.readTemperature();
  int lightLevel = analogRead(LDRPIN);
  unsigned long sensorReadDuration = micros() - startSensorRead;

  if (isnan(temperature)) {
    Serial.println("Failed to read from DHT sensor");
    delay(2000);
    return;
  }

  // 2. Measure data sending time
  unsigned long startDataSend = micros();
  String data = String(temperature) + "," + String(lightLevel) + "\n";
  client.print(data);
  unsigned long dataSendDuration = micros() - startDataSend;

  // Print timing information
  Serial.print("Sensor Read Time: ");
  Serial.print(sensorReadDuration);
  Serial.println(" microseconds");

  Serial.print("Data Send Time: ");
  Serial.print(dataSendDuration);
  Serial.println(" microseconds\n\n");
```

*Figure 23 – Sensors*

The ESP32 continuously reads the temperature and light levels from the sensors and sends them to the server via TCP. The data is sent in a comma-separated format, and the server can then process and act on the information.

## 3.2. ESP32 Actuator Execution

The ESP32 controls actuators based on the commands received from the server. For example, it can turn on or off an LED or control a motor depending on what the server commands. This part of the system is shown below, where the server reads the sensor values, analyses the data, and sends the correct commands to the ESP32. Below is an example of the esp32-side logic executing commands:

```
// Wait for server's response
  unsigned long startCommandReceive = micros();
  if (client.available()) {
    String commands = client.readString();
    unsigned long commandReceiveDuration = micros() - startCommandReceive;

    Serial.print("Command Receive Time: ");
    Serial.print(commandReceiveDuration);
    Serial.println(" microseconds\n\n");

    // Parse and execute commands
    int index = 0;
    while ((index = commands.indexOf('\n')) != -1) {
      unsigned long startCommandExecution = micros();

      String command = commands.substring(0, index);
      command.trim();

      if (command == "HEATER_ON") {
        digitalWrite(HEATERPIN, HIGH);
      } else if (command == "HEATER_OFF") {
        digitalWrite(HEATERPIN, LOW);
      } else if (command == "LIGHT_ON") {
        lightOnAssembly();
      } else if (command == "LIGHT_OFF") {
        lightOffAssembly();
      } else if (command == "BLINDS_OPEN") {
        for (pos = 0; pos <= 180; pos++) {
          blindsServo.write(pos);
          delay(15);
        }
      } else if (command == "BLINDS_CLOSE") {
        for (pos = 180; pos >= 0; pos--) {
          blindsServo.write(pos);
          delay(15);
        }
      }
```

*Figure 24- Actuators*

## 3.3. Server TCP

The server handles the sensor data received from the ESP32, ensuring its validity, generating the corresponding commands, and sending them back to the ESP32 to control the actuators. Upon receiving the data, the server first validates its format and extracts the temperature and light-dependent resistor (LDR) values. If the data is valid, the server generates commands based on predefined conditions, as shown in the table below. For example, it adjusts the blinds, lights, and heater according to the LDR value and temperature. These commands are sent back to ESP32 if there are any changes in state, ensuring smooth communication.

Furthermore, the server tracks the time taken for different tasks, such as receiving the data, validating it, generating commands, and sending them to the client. These timing metrics assist in performance monitoring and troubleshooting. The server continues processing data and commands in a loop until the client disconnects.

| Condition | Blinds | Lights | Heater |
|---|---|---|---|
| Light < 2000 | Open (**BLINDS_OPEN**) | On (**LIGHT_ON**) | Based on Temp |
| 2000 <= Light <3000 | Open (**BLINDS_OPEN**) | Off (**LIGHT_OFF**) | Based on Temp |
| Light <=3000 | Close (**BLINDS_CLOSE**) | Off (**LIGHT_OFF**) | Based on Temp |
| Temp < 20ºC | No Change | No Change | On (**HEATER_ON**) |
| Temp >= 20ºC | No Change | No Change | Off (**HEATER_OFF**) |

*Table 3- System Behaviour*

## 3.4. Assembly

To implement this project, it was necessary to include a piece of code developed in Assembly. After evaluating the requirements, it was decided to implement two Assembly functions to control the state of an **LED**, which represents one of the actuators in the system. These functions directly manipulate the ESP32's GPIO registers, ensuring efficient control of the hardware.

The first function, **lightOnAssembly**, turns on the **LED** by setting **GPIO5** to high (on), while the second function, **lightOffAssembly**, turns off the **LED** by setting **GPIO5** to low (off). Here's the code for both functions:

```
// Inline assembly function to turn on light
void lightOnAssembly() {
  asm volatile (
    "movi a2, 0x3FF44000\n"
    "movi a3, (1 << 5)\n"    /
    "l32i a4, a2, 0x4\n"
    "or a4, a4, a3\n"
    "s32i a4, a2, 0x4\n"
    ::: "a2", "a3", "a4", "memory"
  );
}

// Inline assembly function to turn off light
void lightOffAssembly() {
  asm volatile (
    "movi a2, 0x3FF44000\n"
    "movi a3, ~(1 << 5)\n"
    "l32i a4, a2, 0x4\n"
    "and a4, a4, a3\n"
    "s32i a4, a2, 0x4\n"
    ::: "a2", "a3", "a4", "memory"
  );
}
```

*Figure 25 - Assembly Code*

The Assembly code begins by loading the base address of the GPIO control registers into a register, which provides the starting point for accessing the GPIO registers on the ESP32. Then, a bitmask is created for GPIO5, targeting pin 5 on the ESP32. The current state of the GPIO output register is read, and depending on the desired action, the corresponding bit for GPIO5 is set or cleared. Specifically, in the **lightOnAssembly** function, a bitwise OR operation is performed to set GPIO5 to high (turning on the LED), and in the **lightOffAssembly** function, a bitwise AND operation is used to clear the bit and turn off the LED. Finally, the updated value of the GPIO output register is stored, applying the change to the GPIO pin.

## 3.5. Real Time Scheduling

To address our real-time scheduling requirements, we began by analysing the problem and dividing it into distinct subsystems: the Room Control Unit, the Arduino for sensors, and the

Arduino for actuators, as illustrated in Figure 2.

To maximize the efficient use of computing time, we researched various scheduling algorithms to determine which would best suit our needs. We concluded that pre-emptive Fixed Priority Scheduling (FPS) is the most appropriate approach because it ensures tasks are executed in a specific order, which aligns with our system's requirements.

**Room Control Unit:**

The Room Control Unit manages the following tasks:

- **Verify_Data_Sensors**: This periodic task continuously receives data from the sensors. The data collected by this task serves as input for other system operations.
- **Generate_Commands**: This aperiodic task executes only when an action is required for the actuators. For instance, if the temperature exceeds the predefined threshold, this task will send a command to turn off the heater (LED). Additionally, a sporadic server will need to be implemented to handle command generation and transmission, ensuring a given bandwidth is allocated for efficient communication with the actuators.
- **MQTT_Data**: This periodic task still to be implemented sends data to the interface via MQTT at regular intervals. It relies on shared resources, such as temperature and luminosity data, which must first be updated by the **Verify_Data_Sensors** task. To ensure **MQTT_Data** uses the most recent data, a mutex will be implemented to manage access to these shared resources, preventing it from transmitting outdated information.
- **Sensor_Verification**: This periodic task will be implemented to periodically verify the proper functioning of the sensors. For example, the DHT11 sensor will read the temperature and compare it with the previously recorded value. If the temperature shows a significant variation, it may indicate a malfunction in the sensor. Similarly, if the sensor stops providing readings altogether, it may also be broken, prompting the need for a technician to inspect and verify the component.

**Arduino (Sensors):**

This Arduino is responsible for managing the following tasks:

- **Read_Data_Sensors:** This periodic task continuously reads data from its respective components. For example, the LDR reads luminosity, and the DHT11 reads temperature. The collected data is then stored in variables for further use.
- **Send_Data_Sensors:** This periodic task continuously sends the sensor data to the Room Control Unit. To ensure that the data sent is up-to-date and consistent, a mutex will be

implemented, preventing outdated or unordered data from being transmitted.

**Arduino (Actuators):**

This Arduino is responsible for managing the following tasks:

- **Receive_Commands**: this Aperiodic task will only be executed when the Room Control Unit sends commands.

**Task Sets:**

| | C(ms) | D(ms) | T(ms) | Priority |
|---|---|---|---|---|
| **Verify_Data_Sensors** | 10 | 35 | 35 | 4 |
| **MQTT_Data** | 10 | 70 | 70 | 2 |
| **Sensor_Verification** | 15 | 100 | 100 | 1 |

The server will have priority 3 to be able to send the command as soon as possible. In order for the task set to schedulable, and at the same time the server be able to execute when needed with the right priority, we gave the server a bandwidth of 15%. So, for this sporadic server we gave it a period of 40, and a capacity of 6.

| | C(ms) | D(ms) | T(ms) | Priority |
|---|---|---|---|---|
| **Read_Data_Sensors** | 10 | 20 | 20 | 2 |
| **Send_Data_Sensors** | 7 | 30 | 30 | 1 |

# 4. Updates

In this section, we will outline the recent updates and improvements made to the system. These changes are aimed at improving the overall report and providing more clarity on some of the topics previously addressed. We will describe the modifications to the scheduling approach and task management, add scheduling and concurrency analysis, explain the thought behind the decisions made and finally give more context on updated code.

| | |
|---|---|
| Technologies | We updated the ARDUINO/MQTT and added SmartDataModels |
| Esp32 Sensors and Actuators | Updated Code + Threads |
| Server | Updated Code + Threads |
| Real time scheduling | Choices explanation and schedulability analysis and blocked time analysis |
| Interface | Added the interface |
| Problem Domain Model | Updates to the black box and white box |

## 4.1. TECHNOLOGIES

### 4.1.1. Arduino

The system incorporates two Arduino boards, each serving a distinct role to ensure modularity and efficiency, although for the sake of the simulation we will only be using a single ESP32 board. The Sensor Arduino is tasked with collecting data from the DHT11 temperature and humidity sensor and the LDR5539 light-dependent resistor. This Arduino processes the sensor data and transmits it to the TCP server for validation and further handling. Unlike the initial stages, where data was transmitted in an unstructured format, the system now integrates Smart Data Models. This implementation standardizes the data format, enhancing compatibility with other systems and aligning the project with best practices for data exchange in IoT environments.

The Actuator Arduino, on the other hand, manages the system’s actuators, including the SG90 servo motor and LEDs. It receives validated commands from the TCP server and executes actions such as adjusting the blinds mechanism, turning on or off the simulated smart lights, or activating the heating simulation. Both Arduinos connect to the same Wi-Fi network, enabling reliable communication with the TCP server and ensuring seamless operation within the system’s architecture. The adoption of Smart Data Models further strengthens the system's robustness, ensuring scalability and interoperability for future enhancements.

### 4.1.2. Mqtt

MQTT is used to facilitate communication between the control unit (TCP server) and the user interface, which is built using Node-RED. Once developed, MQTT will allow the control unit to publish sensor data received from the Sensor Arduino to an MQTT broker, making it accessible to Node-RED for real-time visualization and monitoring. Additionally, MQTT will be used to send user commands from Node-RED to the TCP server, which will relay them to the Actuator Arduino. This bi-directional communication will help streamline integration and data flow across the system.

At present, MQTT is still in the development phase, and data transmission is handled using basic unstructured communication between the components. Smart Data Models will later be incorporated to ensure standardized data formatting for MQTT communication.

### 4.1.3. Smart Data Models

To standardize the payloads received from the server, we applied a smart data model to the data sent to the user interface. The payload is composed of the mandatory fields, such as an ID, the date of creation, and the origin of the payload. Additionally, it includes the necessary fields to send relevant information to the user interface. These fields are the **temperature field**, which returns the temperature of the room, and the **illuminance field**, which returns the luminosity of the same room.

```
snprintf(payload, 512,
         "{ \"id\": \"%s_%ld\", \"dateObserved\": \"%s\", \"location\": \"%s\", \"temperature\": \"%s\", \"illuminance\": \"%s\" }",
         location, currentTime, timeString, location, tempStr, humidStr);

return payload;
```

*Figure 26 Smart data model*

## 4.2. Esp32 (Real Time Scheduling)

To effectively handle sensor readings and data transmission, the ESP32 uses a FreeRTOS-based approach, and two periodic tasks were created, each one running on a single core of the ESP32:

- **Sensor Reading**
- **Data Transmission Task**

Both tasks are assigned specific priorities, with the sensor reading task having a higher priority (2) to ensure timely and accurate data collection. The tasks are pinned to specific cores of the ESP32's dual-core processor to optimize performance and reduce context-switching overhead.

Specifically, the sensor reading task runs on Core 1 (secondary core), while the data transmission task is executed on Core 0 (primary core). By running both tasks on the same CPU, the system avoids potential synchronization issues and ensures deterministic behaviour.

The following images showcases how the two tasks are being scheduled and on what core the task is being executed.

```
// Create tasks
xTaskCreatePinnedToCore(
    readDataTask,               // Task function
    "ReadDataTask",             // Name
    4096,                       // Stack size
    NULL,                       // Parameters
    READ_DATA_PRIORITY,         // Priority
    NULL,                       // Task handle
    1                           // Core ID (1 = secondary core)
);

xTaskCreatePinnedToCore(
    sendDataTask,               // Task function
    "SendDataTask",             // Name
    4096,                       // Stack size
    NULL,                       // Parameters
    SEND_DATA_PRIORITY,         // Priority
    NULL,                       // Task handle
    0                           // Core ID (0 = primary core)
);
```

*Figure 27 - Esp32 Tasks Scheduling*

### 4.2.1. Esp32 Sensors

The ESP32 collects data from various sensors, including a DHT11 temperature sensor and an LDR (Light Dependent Resistor) and then sends the data retrieved from the sensors to the server TCP as seen in Figure 23.

```
// Read Data Sensors Task (Periodic, Priority 2)
void readDataTask(void *parameter) {
    TickType_t xLastWakeTime = xTaskGetTickCount();
    const TickType_t xPeriod = pdMS_TO_TICKS(READ_DATA_PERIOD);

    while (1) {
        // Record start time
        unsigned long startTime = micros();

        // Read sensors
        float temp = dht.readTemperature();
        int light = analogRead(LDRPIN);

        if (!isnan(temp)) {
            // Update shared data with mutex protection
            xSemaphoreTake(sensorMutex, portMAX_DELAY);
            sensorData.temperature = temp;
            sensorData.light_level = light;
            xSemaphoreGive(sensorMutex);

            Serial.printf("Read - Temperature: %.1f°C, Light: %d\n", temp, light);
        }

        // Calculate execution time
        unsigned long executionTime = micros() - startTime;
        Serial.printf("Read Task Execution Time: %lu microseconds\n", executionTime);

        // Wait for next period
        vTaskDelayUntil(&xLastWakeTime, xPeriod);
    }
}
```

*Figure 28 - Read Data Task*

The ESP32 continuously reads the temperature and light levels from the sensors and sends them to the server via TCP. The data is sent in a comma-separated format, and the server can then process and act on the information.

The task retrieves the latest sensor readings (temperature and light levels) from a shared SensorData structure. To ensure thread safety, a mutex is used while accessing this shared data.

### 4.2.2. Esp 32 Actuators

The ESP32 transmits sensor data to a TCP server through a dedicated FreeRTOS task designed specifically for this purpose. This task is executed periodically with a set interval of 30 milliseconds, ensuring a consistent flow of data to the server. The data, which consists of temperature and light readings, is retrieved from a shared SensorData structure. To maintain data integrity and avoid concurrency issues, a mutex is used to safely access this shared resource, preventing simultaneous modifications by other tasks.

Once the data is retrieved, it is formatted as a comma-separated string, such as 25.5,300\n, and

sent to the server using the established TCP connection. This format was chosen for its simplicity and compatibility with server-side parsing logic. After transmitting the sensor data, the task checks for any incoming commands from the server. These commands are processed by the handleCommands function that control the connected actuators as seen in the Figure 29. For instance, a HEATER_ON command activates the heater, while BLINDS_OPEN adjusts the servo motor to open the blinds. Commands are read and handled non-blockingly, ensuring the task does not stall due to server delays.

If the TCP connection to the server is lost, the task quickly identifies the disconnection and attempts to reconnect automatically. This reconnection mechanism is essential to ensure the system remains functional despite temporary network interruptions. The task's periodic execution is controlled by the FreeRTOS **vTaskDelayUntil** function, which allows for accurate timing and efficient use of the CPU.

Below is an example of the esp32-side logic executing sending data and waiting for commands:

```
// Send Data Sensors Task (Periodic, Priority 1)
void sendDataTask(void *parameter) {
    TickType_t xLastWakeTime = xTaskGetTickCount();
    const TickType_t xPeriod = pdMS_TO_TICKS(SEND_DATA_PERIOD);

    while (1) {
        // Record start time
        unsigned long startTime = micros();

        if (client.connected()) {
            // Get sensor data with mutex protection
            xSemaphoreTake(sensorMutex, portMAX_DELAY);
            float temp = sensorData.temperature;
            int light = sensorData.light_level;
            xSemaphoreGive(sensorMutex);

            // Send data to server
            String data = String(temp) + "," + String(light) + "\n";
            client.print(data);
            Serial.printf("Sent: %s", data.c_str());

            // Check for and handle incoming commands
            while (client.available()) {
                String command = client.readStringUntil('\n') + "\n";
                handleCommand(command);
            }
        } else {
            // Attempt to reconnect
            Serial.println("Reconnecting to server...");
            client.connect(server_ip, server_port);
        }

        // Calculate execution time
        unsigned long executionTime = micros() - startTime;
        Serial.printf("Send Task Execution Time: %lu microseconds\n", executionTime);

        // Wait for next period
        vTaskDelayUntil(&xLastWakeTime, xPeriod);
    }
}
```

*Figure 29 - Send Data Task*

```
85  // Function to handle commands from server
86  void handleCommand(String command) {
87      if (command == "HEATER_ON\n") {
88          digitalWrite(HEATERPIN, HIGH);
89      } else if (command == "HEATER_OFF\n") {
90          digitalWrite(HEATERPIN, LOW);
91      } else if (command == "LIGHT_ON\n") {
92          //digitalWrite(LIGHTPIN, HIGH);
93          lightOnAssembly();
94      } else if (command == "LIGHT_OFF\n") {
95          //digitalWrite(LIGHTPIN, LOW);
96          lightOffAssembly();
97      } else if (command == "BLINDS_OPEN\n") {
98          for (int pos = 0; pos <= 180; pos++) {
99              blindsServo.write(pos);
100             vTaskDelay(pdMS_TO_TICKS(15));
101         }
102     } else if (command == "BLINDS_CLOSE\n") {
103         for (int pos = 180; pos >= 0; pos--) {
104             blindsServo.write(pos);
105             vTaskDelay(pdMS_TO_TICKS(15));
106         }
107     }
108 }
```

*Figure 30 – handleCommand*

## 4.3. Server (Room Control Unit)

The server handles the sensor data received from the ESP32, ensuring its validity, generating the corresponding commands, and sending them back to the ESP32 to control the actuators. Upon receiving the data, the server first validates its format and extracts the temperature and light-dependent resistor (LDR) values. If the data is valid, the server generates commands based on predefined conditions, as shown in the table below, although these predefined conditions can be changed in the user interface. For example, it adjusts the blinds, lights, and heater according to the LDR value and temperature. These commands are sent back to ESP32 if there are any changes in state, ensuring smooth communication.

Furthermore, the server tracks the time taken for different tasks, such as receiving the data, validating it, generating commands, and sending them to the client. These timing metrics assist in performance monitoring and troubleshooting. The server continues processing data and commands in a loop until the client disconnects.

| Condition | Blinds | Lights | Heater |
|---|---|---|---|
| Light < 2000 | Open (**BLINDS_OPEN**) | On (**LIGHT_ON**) | Based on Temp |
| 2000 <= Light <3000 | Open (**BLINDS_OPEN**) | Off (**LIGHT_OFF**) | Based on Temp |
| Light <=3000 | Close (**BLINDS_CLOSE**) | Off (**LIGHT_OFF**) | Based on Temp |
| Temp < 20ºC | No Change | No Change | On (**HEATER_ON**) |
| Temp >= 20ºC | No Change | No Change | Off (**HEATER_OFF**) |

### 4.3.1. Room Control Unit (Real Time Scheduling)

In the server, each task that occurs such as receiving data, validating it, generating commands, and sending them back to the ESP32, triggers the creation of a new thread. Each thread is assigned specific characteristics to ensure proper scheduling and timely execution.

In the server, each distinct task triggers the creation of a new thread, enabling concurrent processing of different tasks to improve responsiveness and efficiency. Each thread is assigned a priority level, which determines its execution order relative to other threads, with higher-priority tasks such as data validation or sending commands being prioritized over less critical tasks as seen below.

```
1
2       pthread_create(&verify_thread, NULL, verify_data_task, NULL);
3       pthread_create(&server_thread, NULL, command_server_task, NULL);
4       pthread_create(&mqtt_thread, NULL, mqtt_task, NULL);
5       pthread_create(&preference_detection_thread, NULL, preference_detection_task, NULL);
6       pthread_create(&sensor_verification_thread, NULL, sensor_verification_task, NULL);
7
8       set_thread_priority(verify_thread, VERIFY_DATA_PRIORITY);
9       set_thread_priority(server_thread, SERVER_PRIORITY);
10      set_thread_priority(mqtt_thread, MQTT_PRIORITY);
11      set_thread_priority(preference_detection_thread, PREFERENCE_DETECTION_PRIORITY);
12      set_thread_priority(sensor_verification_thread, SENSOR_VERIFICATION_PRIORITY);
13
14      pthread_join(verify_thread, NULL);
15      pthread_join(server_thread, NULL);
16      pthread_join(mqtt_thread, NULL);
17      pthread_join(preference_detection_thread, NULL);
18      pthread_join(sensor_verification_thread, NULL);
19
```

*Figure 31 - Threads and defined priority.*

The execution time of each thread is measured to ensure that tasks are completed within the expected time frame, and each thread has a defined deadline by which it must complete its task as seen below.

```
1  // real time scheduling
2  #define VERIFY_DATA_PERIOD 35
3  #define VERIFY_DATA_EXECUTION 10
4  #define SERVER_PERIOD 40
5  #define SERVER_CAPACITY 6
6  #define MQTT_PERIOD 70
7  #define MQTT_EXECUTION 10
8  #define SENSOR_VERIFICATION_PERIOD 100
9  #define SENSOR_VERIFICATION_EXECUTION 15
10
11 // priorities
12 #define VERIFY_DATA_PRIORITY 5
13 #define SERVER_PRIORITY 4
14 #define MQTT_PRIORITY 3
15 #define PREFERENCE_DETECTION_PRIORITY 2
16 #define SENSOR_VERIFICATION_PRIORITY 1
17 void set_thread_priority(pthread_t thread, int
   priority)
18 {
19     struct sched_param param;
20     param.sched_priority = priority;
21     pthread_setschedparam(thread, SCHED_FIFO, &param);
22 }
```

*Figure 32- Execution Time, deadlines and setPriority function*

If a thread fails to meet its deadline, corrective actions, such as logging a warning or retrying the operation, may be taken to prevent delays. To manage these threads, the server employs a scheduling algorithm based on priorities or deadlines, ensuring that critical operations are handled promptly.

The **Verify Data Task** validates the received sensor data. If the data is valid, the server continues processing it, otherwise, an error is logged and every time the shared resource is accessed the mutex blocks it from being accessed by any other task that may be running. Below is the code snippet for the verify Data task:

```c
void *verify_data_task(void *arg)
{
    struct timespec next_period;
    clock_gettime(CLOCK_MONOTONIC, &next_period);

    while (1)
    {
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        char buffer[BUFFER_SIZE] = {0};
        int bytes_read = recv(shared_data.client_sock, buffer, BUFFER_SIZE - 1, 0);

        if (bytes_read > 0)
        {
            float temp;
            int light;
            if (sscanf(buffer, "%f,%d", &temp, &light) == 2)
            {
                pthread_mutex_lock(&shared_data.data_mutex);
                shared_data.previous_temperature = shared_data.temperature;
                shared_data.temperature = temp;
                shared_data.light_level = light;
                pthread_mutex_unlock(&shared_data.data_mutex);

                printf("Received - Temperature: %.1f°C, Light: %d\n", temp, light);
            }
        }

        clock_gettime(CLOCK_MONOTONIC, &end);
        long elapsed_us = (end.tv_sec - start.tv_sec) * 1000000 +
                          (end.tv_nsec - start.tv_nsec) / 1000;
        if (elapsed_us < VERIFY_DATA_EXECUTION * 1000)
        {
            usleep(VERIFY_DATA_EXECUTION * 1000 - elapsed_us);
        }

        add_ms_to_timespec(&next_period, VERIFY_DATA_PERIOD);
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next_period, NULL);
    }
    return NULL;
}
```

*Figure 33 - Verify Data Task*

The **Handle Command Server Task** generates commands based on the validated data and every time the shared resource is accessed the mutex blocks it from being accessed by any other task that may be running. For example, it controls the heater and blinds based on the temperature and LDR values. The generated commands are then sent to the ESP32. Below is the code snippet for the handle Command Server Task:

```c
void *command_server_task(void *arg)
{
    struct timespec next_period;
    clock_gettime(CLOCK_MONOTONIC, &next_period);

    while (1)
    {
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        pthread_mutex_lock(&shared_data.data_mutex);
        float temp = shared_data.temperature;
        int light = shared_data.light_level;
        pthread_mutex_unlock(&shared_data.data_mutex);

        pthread_mutex_lock(&system_state.state_mutex);


         // Lights control
        if (light < shared_data.light_preference && system_state.lights_state != 1 && system_state.blinds_state == 0)
        {
            send(shared_data.client_sock, "LIGHT_ON\n", strlen("LIGHT_ON\n"), 0);
            printf("Sent command: LIGHT_ON\n");
            system_state.lights_state = 1;
            sendAlertToMQTT("/comcs/g12012/alerts", "Lights are on!");
        }
        else if (light >= shared_data.light_preference && system_state.lights_state != 0 && system_state.blinds_state == 0)
        {
            send(shared_data.client_sock, "LIGHT_OFF\n", strlen("LIGHT_OFF\n"), 0);
            printf("Sent command: LIGHT_OFF\n");
            system_state.lights_state = 0;
            sendAlertToMQTT("/comcs/g12012/alerts", "Lights are off!");
        }

        // Blinds control
        if (light < shared_data.light_preference  && system_state.blinds_state != 0)
        {
            send(shared_data.client_sock, "BLINDS_OPEN\n", strlen("BLINDS_OPEN\n"), 0);
            printf("Sent command: BLINDS_OPEN\n");
            system_state.blinds_state = 0;
            sendAlertToMQTT("/comcs/g12012/alerts", "Blinds are opening!");
        }
        else if (light >= shared_data.light_preference  && system_state.blinds_state != 1)
        {
            send(shared_data.client_sock, "BLINDS_CLOSE\n", strlen("BLINDS_CLOSE\n"), 0);
            printf("Sent command: BLINDS_CLOSE\n");
            system_state.blinds_state = 1;
            sendAlertToMQTT("/comcs/g12012/alerts", "Blinds are closing!");
        }



        // Heater control
        if (temp < shared_data.temp_preference && system_state.heater_state != 1)
        {
            send(shared_data.client_sock, "HEATER_ON\n", strlen("HEATER_ON\n"), 0);
            printf("Sent command: HEATER_ON\n");
            system_state.heater_state = 1;
            sendAlertToMQTT("/comcs/g12012/alerts", "Heater is on!");
        }
        else if (temp >= shared_data.temp_preference && system_state.heater_state != 0)
        {
            send(shared_data.client_sock, "HEATER_OFF\n", strlen("HEATER_OFF\n"), 0);
            printf("Sent command: HEATER_OFF\n");
            system_state.heater_state = 0;
            sendAlertToMQTT("/comcs/g12012/alerts", "Heater is off!");
        }

        pthread_mutex_unlock(&system_state.state_mutex);

        clock_gettime(CLOCK_MONOTONIC, &end);
        long elapsed_us = (end.tv_sec - start.tv_sec) * 1000000 +
                          (end.tv_nsec - start.tv_nsec) / 1000;
        if (elapsed_us < SERVER_CAPACITY * 1000)
        {
            usleep(SERVER_CAPACITY * 1000 - elapsed_us);
        }

        add_ms_to_timespec(&next_period, SERVER_PERIOD);
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next_period, NULL);
    }
    return NULL;
}
```

*Figure 34 - Generate Commands*

The **MQTT Data Task** is responsible for transmitting the sensor data read from the ESP32 to the user interface via MQTT. This enables real-time interaction with the room, allowing the user to monitor sensor readings and control the environment remotely and every time the shared resource is accessed the mutex blocks it from being accessed by any other task that may be running. Below is the code snippet for the MQTT Data Task:

```c
void *mqtt_task(void *arg)
{
    (void)arg; // Suppress unused parameter warning
    struct timespec next_period;
    clock_gettime(CLOCK_MONOTONIC, &next_period);


    while (1)
    {
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        pthread_mutex_lock(&shared_data.data_mutex);
        float temp = shared_data.temperature;
        int light = shared_data.light_level;
        pthread_mutex_unlock(&shared_data.data_mutex);

        char *payload = toSmartData(temp, light);
        if (payload != NULL)
        {
            sendAlertToMQTT("/comcs/g12012/data", payload);
            free(payload);
        }

        clock_gettime(CLOCK_MONOTONIC, &end);
        long elapsed_us = (end.tv_sec - start.tv_sec) * 1000000 +
                          (end.tv_nsec - start.tv_nsec) / 1000;
        if (elapsed_us < MQTT_EXECUTION * 1000)
        {
            usleep(MQTT_EXECUTION * 1000 - elapsed_us);
        }

        add_ms_to_timespec(&next_period, MQTT_PERIOD);
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next_period, NULL);

    }
    return NULL;
}
```

*Figure 35- Mqtt_data*

The **Sensor_Verification_Task** is responsible ensuring that the sensors are working in perfect conditions by monitoring the readings and if there's a significant disparity in values between two consecutive reads it sends an alert for the user to check the sensors because they may be malfunctioning. This provides extra safety regarding our system integrity and ensuring the quality of the product and every time the shared resource is accessed the mutex blocks it from

being accessed by any other task that may be running. Below is the code snippet for the Sensor Verification Task:

```c
void sensor_verification_task(void *arg)
{
    (void)arg; // Suppress unused parameter warning
    struct timespec next_period;
    clock_gettime(CLOCK_MONOTONIC, &next_period);

    while (1)
    {
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        // verify if the current and the previous temperatures have significant difference
        pthread_mutex_lock(&shared_data.data_mutex);

        float temp = shared_data.temperature;
        float previous_temp = shared_data.previous_temperature;

        // if the temperature difference is greater than 1.5 degrees, send an alert

        if ((temp - previous_temp) > 1.5 || (previous_temp - temp) > 1.5)
        {
            sendAlertToMQTT("/comcs/g12012/alerts",
"Sensor is experiencing high temperature disparity! Something may be wrong with the sensor.");
        }

        pthread_mutex_unlock(&shared_data.data_mutex);

        clock_gettime(CLOCK_MONOTONIC, &end);
        long elapsed_us = (end.tv_sec - start.tv_sec) * 1000000 +
                          (end.tv_nsec - start.tv_nsec) / 1000;

        if (elapsed_us < SENSOR_VERIFICATION_EXECUTION * 1000)
        {
            usleep(SENSOR_VERIFICATION_EXECUTION * 1000 - elapsed_us);
        }

        add_ms_to_timespec(&next_period, SENSOR_VERIFICATION_PERIOD);
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next_period, NULL);

    }
    return NULL;
}
```

*Figure 36 - Sensor Verification Task*

As it's possible to see below on the left we can see the room control unit receiving data from the actuators that are sending data as seen on the right side of the image.

The image displayed shows data more specifically the temperature and light level read by the sensors and every time the user changes the preferences (light or temperature) it sends a message to the server.

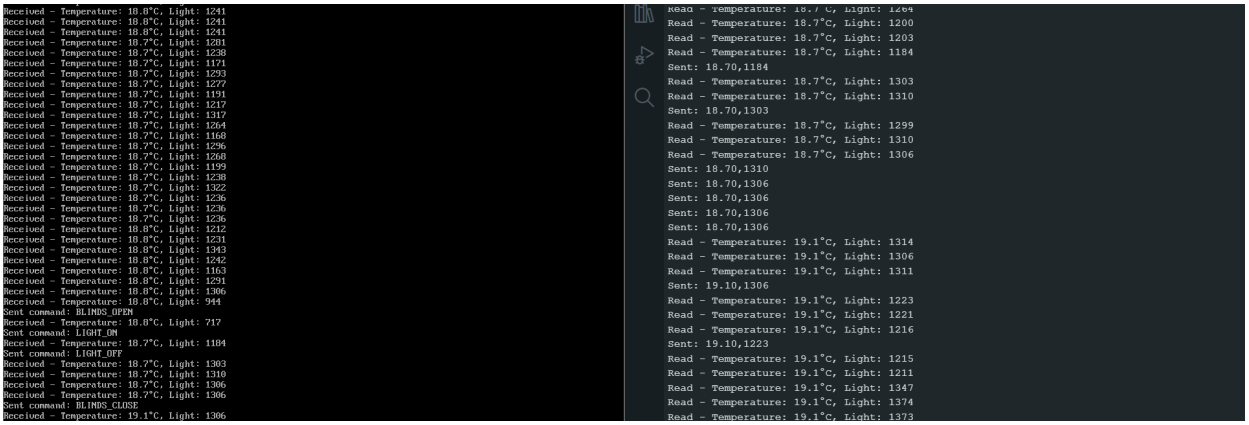


*Figure 37 - Server and ESP32 running*

## 4.4. Interface

The interface is an essential part of the real time system. It enables the user to verify the conditions of the room, while setting its preferences for said conditions and receiving updates on the status of the heater, blinds and lights of said room. In conjunction with notifications of status, the interface also receives warnings, to ensure that the user is aware of any failures that may occur during the system's execution.

For both the temperature and luminosity of the room, there will be two gauges showing the conditions of said room. A slide widget will be the way for the user to set its preference for light that he wants, and a numeric widget for the temperature user preferences.

A small notification will appear in the top right corner of the screen, when changes are made, like turning on the heater, or when errors happen in the system.

As can be seen in Figure 37, after the user sets its preferences, said preferences will be sent, through mqtt to the server, which in turn will use said preferences to manage the system.

The following figure shows the user interface.

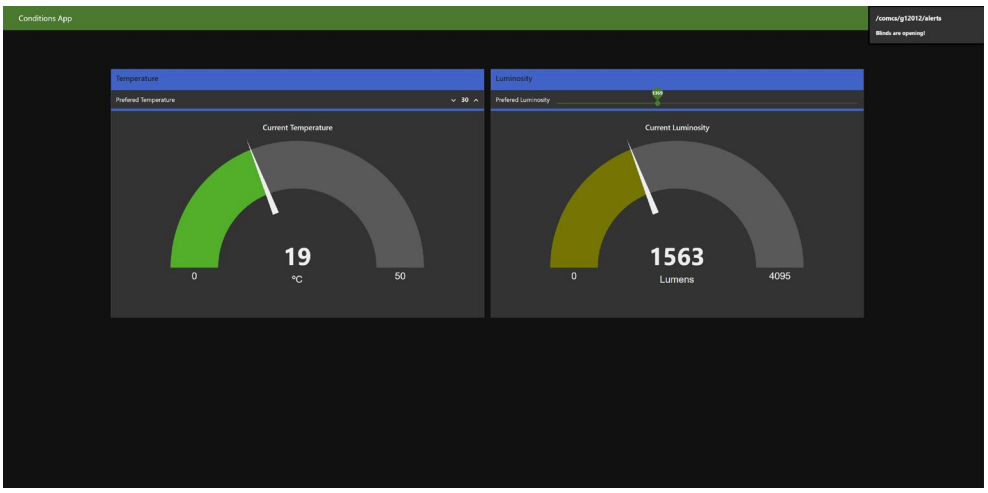


*Figure 38 User Interface*

## 4.5. Real Time Scheduling Analysis:

In the previous iteration it was not clear why we chose to use FPS and a sporadic server so, we concluded that pre-emptive Fixed Priority Scheduling (FPS) is the most appropriate approach because it ensures tasks are executed in a specific order, which aligns with our system's requirements. FPS was chosen due to its ability to prioritize critical tasks, provide deterministic execution, and allow high-priority tasks to pre-empt lower-priority ones, ensuring timely responses to events. Additionally, FPS's simplicity and efficiency make it suitable for our system, as it guarantees that important tasks, such as sensor data collection and actuator commands, are executed in a predictable and reliable manner without delays or missed deadlines.

Additionally, a sporadic server was implemented to handle command generation and transmission, ensuring a given bandwidth is allocated for efficient communication with the actuators. We chose a sporadic server because it allows actuator commands to be sent only when needed, ensuring that the system doesn't become overloaded. Unlike other methods, such as an aperiodic server, which could use up unnecessary resources, the sporadic server helps us manage communication more efficiently. It makes sure that the commands are sent at the right time, without causing delays or wasting bandwidth.

### 4.5.1. ESP32 Schedulability analysis

In order to understand if our taskset is able of being used in our system we have to analyse if the CPU can handle it before actually proceeding for the development phase.
Firstly, we must calculate the CPU utilization of our taskset:

$$U = \frac{10}{20} + \frac{7}{30} = 0.73$$

Next, we have to calculate the max amount of utilization that the CPU can handle for a taskset with 2 tasks:

$$0.73 \leq 2(2^{\frac{1}{2}} - 1)$$

$$0.73 \leq 0.83$$

As it was verified above the CPU utilization of the ESP32 is 0.73 that is below the threshold

permitted wish is 0.83, therefore the ESP32 is schedulable.

Since the CPU is schedulable, we will now proceed to calculate the worst-case response time (WCT):

$$R_2^0 = 7\ ms$$

$$R_2^1 = 7 + \left\lceil \frac{7}{20} \right\rceil \times 10$$

$$R_2^1 = 17\ ms$$

$$R_2^2 = 7 + \left\lceil \frac{17}{20} \right\rceil \times 10$$

$$R_2^2 = 17\ ms$$

Since $R_2^1 = R_2^2 = 17$, we can conclude that the worst-case response time of our taskset is 17 ms.

Duo to having sharing recourses we must calculate the maximum blockage time to ensure that all of tasks can run smoothly without any of them to overlap each other by using the equation below:

$$B_i = \begin{cases} 0 \\ max\{use(k,i) \times C_k\} \end{cases}$$

For the task with higher priority (read_data_sensors) the shared resource is taking 6 ms of execution time, while the task with least priority (send_data_sensors) is taking 3 ms of execution time.

$$B_1 = 0$$

$$B_2 = max\{use(A,2) \times 6\}$$

$$B_2 = \max\{1 \times 6\}$$

$$B_2 = 6\ ms$$

Therefore, the WCBT (worst case blocking time) is 6 ms.

### 4.5.2. Server Schedulability analysis

In order to understand if our taskset is able of being used in our system we have to analyse if the CPU can handle it before actually proceeding for the development phase.
Firstly, we must calculate the CPU utilization of our taskset:

$$U = \frac{10}{35} + \frac{10}{70} + \frac{15}{100} + \frac{6}{40} = 0.73$$

Next, we have to calculate the max amount of utilization that the CPU can handle for a taskset with 2 tasks:

$$0.73 \leq 4(2^{\frac{1}{4}} - 1)$$

$$0.73 \leq 0.757$$

As it was verified above the CPU utilization of the ESP32 is 0.73 that is below the threshold permitted wish is $0.757$, therefore the ESP32 is schedulable

Since the CPU is schedulable, we will now proceed to calculate the worst-case response time (WCT):

$$R_4^0 = 15\ ms$$

$$R_4^1 = 15 + \left\lceil \frac{15}{35} \right\rceil \times 10 + \left\lceil \frac{15}{70} \right\rceil \times 10 + \left\lceil \frac{15}{40} \right\rceil \times 6$$

$$R_4^1 = 41\ ms$$

$$R_4^2 = 15 + \left\lceil \frac{41}{35} \right\rceil \times 10 + \left\lceil \frac{41}{70} \right\rceil \times 10 + \left\lceil \frac{41}{40} \right\rceil \times 6$$

$$R_4^2 = 57\ ms$$

$$R_4^3 = 15 + \left\lceil \frac{57}{35} \right\rceil \times 10 + \left\lceil \frac{57}{70} \right\rceil \times 10 + \left\lceil \frac{57}{40} \right\rceil \times 6$$

$$R_4^3 = 57\ ms$$

Since $R_4^2 = R_4^3 = 57$, we can conclude that the worst-case response time of our taskset is 57 ms.

Duo to having sharing recourses we must calculate the maximum blockage time to ensure that all of tasks can run smoothly without any of them to overlap each other by using the equation below:

$$B_i = \begin{cases} 0 \\ max\{use(k,i) \times C_k\} \end{cases}$$

For the task with higher priority (Verify_data_sensors) the shared resource is taking 7 ms of execution time, while the sporadic server is executing it uses the shared resource for 2 ms, the task with priority below the server (MQTT_data) uses shared resource for 4 ms and finally, the task with least priority (Sensor_verification) uses the shared resource for 3 ms.

$$B_1 = 0$$

$$B_2 = max\{use(A,2) \times 4\}$$

$$B_2 = \max\{1 \times 4\}$$

$$B_2 = 4\ ms$$

$$B_3 = max\{use(A,3) \times 2\ \}$$

$$B_3 = \max\{1 \times 2\}$$

$$B_3 = 2\ ms$$

$$B_4 = max\{use(A,4) \times\ 7\}$$

$$B_4 = \max\{1 \times 7\}$$

$$B_4 = 7\ ms$$

$$WCBT = B_1 +\ B_2 + B_3 + B_4 = 13\ ms$$

Therefore, the WCBT (worst case blocking time) is 13 ms.

## 4.6. Problem Domain Model

Some updates were made to the problem domain model to expand on the different actuators that compose the previous lighting actuator.

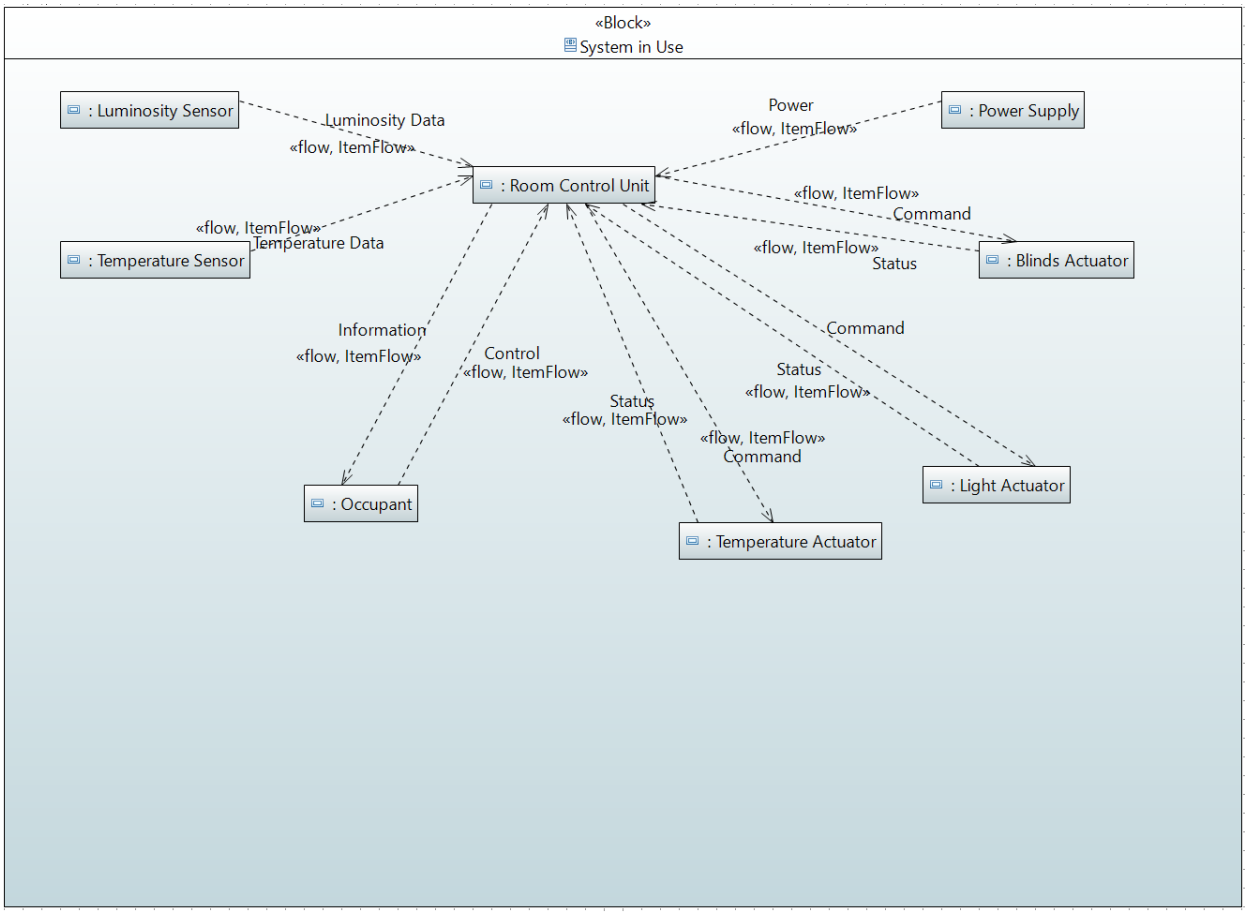


*Figure 39 System Context*

With the specification of the lighting actuators, being the light actuator, controlling the lights, and a blind actuator, that controls the blinds, some changes were needed in the activity diagram.

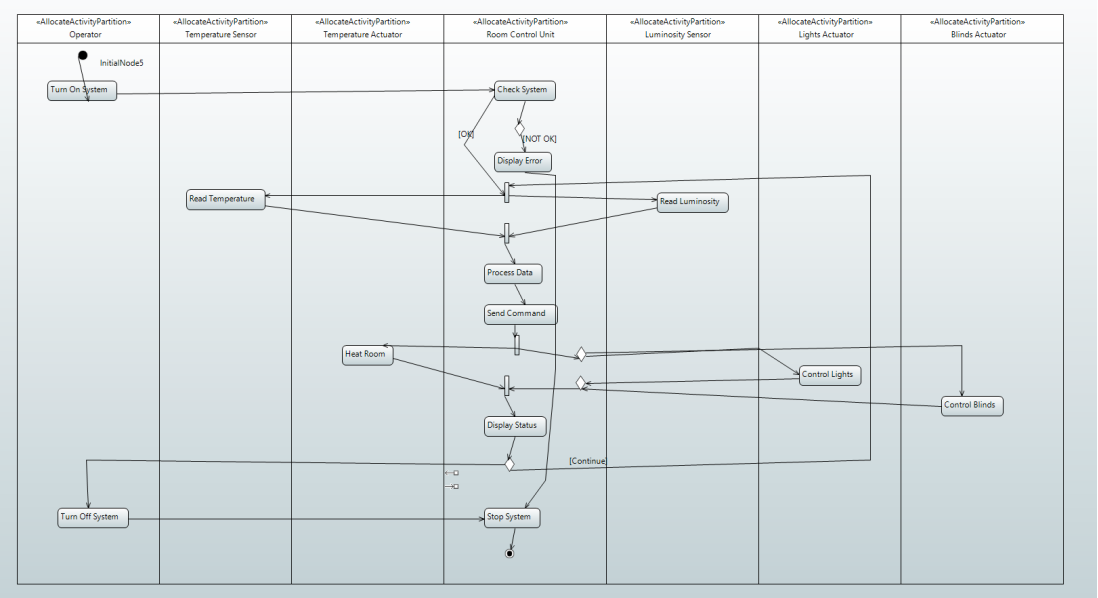


*Figure 40 Activity Diagram*

It was also needed to specify which SoI the use cases in existence belonged to.

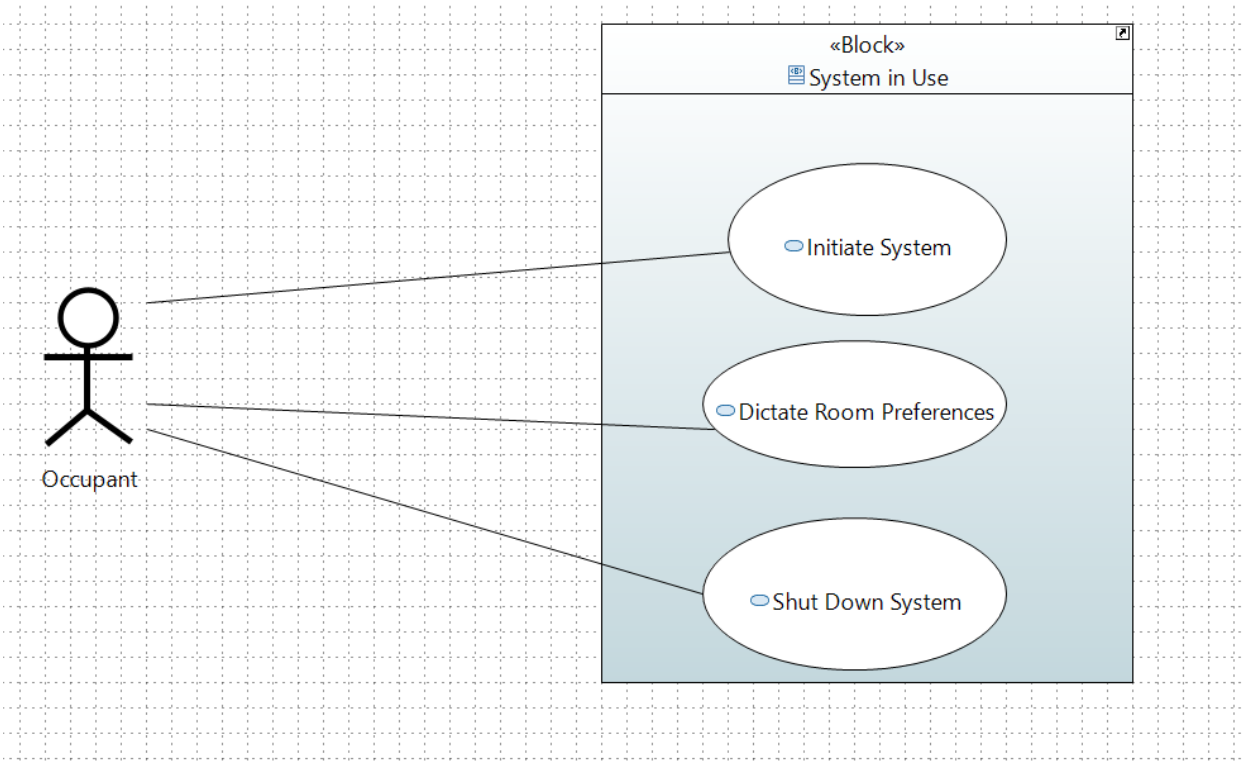


*Figure 41 Use Cases*

With the changes made previously, there was also the necessity to make some changes to the conceptual systems with the addition of an input for both the light actuator and blinds actuator.

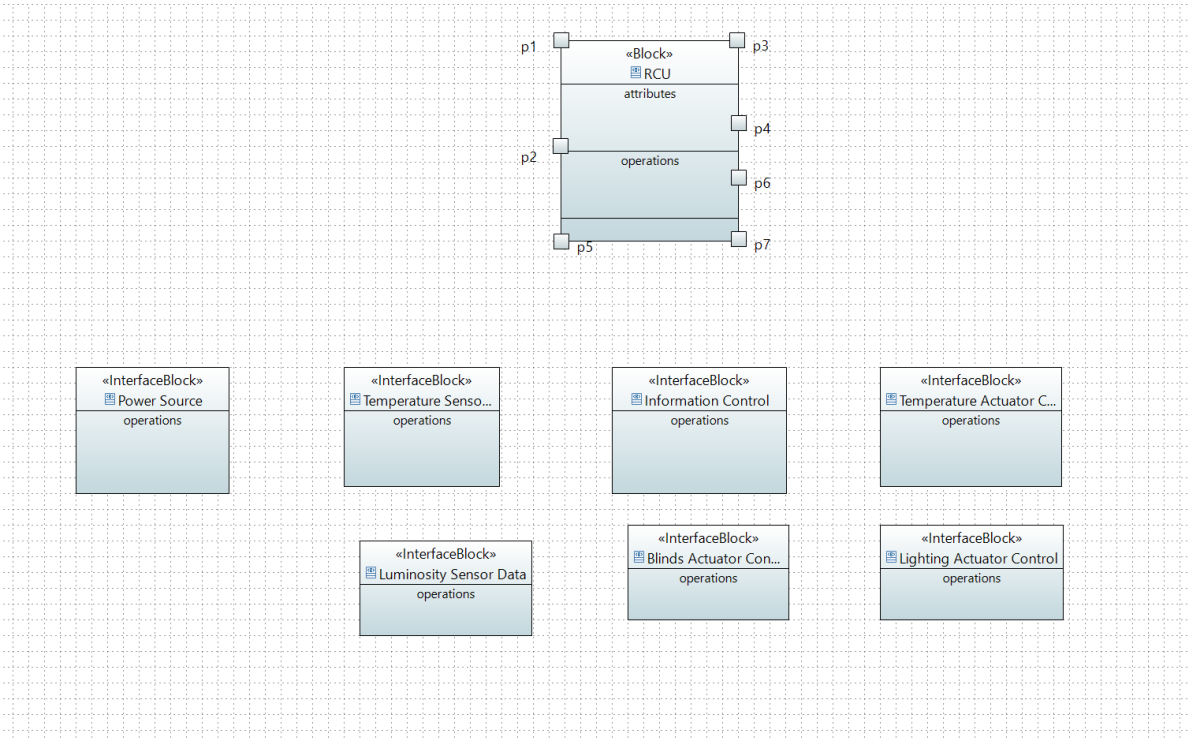


*Figure 42 Conceptual system and its inputs and outputs*

And finally, the interactions between the conceptual subsystems and the conceptual system changed, with the addition of the new actuators.

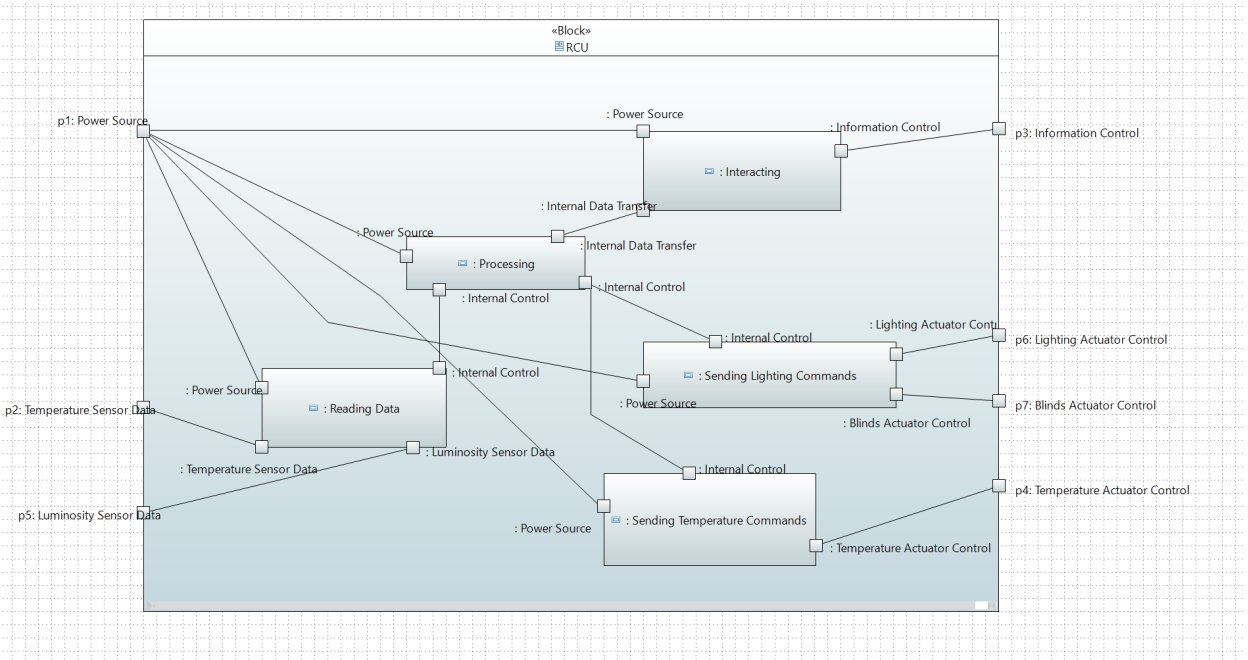


*Figure 43 Conceptual Subsystems and Interactions*