INSTITUTE POLYTECHNI of Cávado and Ave
C

DEGREE IN ENGINEERING SYSTEMS
IT
SYSTEMS SOAKED IN TIME Real

Practical Work

Students (Group 4) :
Francisco Arantes -
23504 Tiago Oliveira -
16622 Luís Ferreira -
23516

Teacher :
Paulo Macedo

December 27th, 2023

Contents

1 Introduction

In a context of increasing digitalization and automation, real-time embedded systems play a key role in transforming conventional homes into smart and efficient spaces. This project aims to develop and integrate a series of real-time embedded systems into a mock-up of a real house, providing a practical and applied perspective of emerging technologies in the field of Smart Housing.

The mockup, faithfully representing the various spaces of a home, will serve as a backdrop for the implementation of four distinct systems, each addressing specific challenges related to comfort, security and home automation. By exploring and integrating these systems, we aim to create a synergy that not only demonstrates the practical application of embedded systems in real time, but also offers a holistic solution to make everyday life more convenient and efficient.

Figure 1: Smart Home

System A - Interior Lighting Control:

The first system aims to optimize indoor lighting based on the intensity of sunlight, thus providing constant illumination and efficient energy management. The use of LEDs and LDR sensors will provide a dynamic response to the environment, adjusting intelligently to the lighting conditions.

System B - Climate Control:

The second system focuses on controlling the ambient temperature via a ventilation system. The fan is triggered automatically when the temperature exceeds 24 degrees Celsius, providing not only thermal comfort, but also visually indicating the status of the system via LEDs and an LCD display.

System C - Security System (Alarm):

The third system introduces a layer of security by detecting intruder movements with a PIR sensor. The alarm, consisting of a flashing light and a characteristic sound signal, reinforces home protection, while a disarm button gives the user control.

System D - Additional Automation for Smart Housing:

The fourth system is an additional component that aims to automate various tasks in the home, such as parking, watering and access control. The integration of this system adds a layer of versatility to the smart home.

In addition, each system will be enriched with an additional requirement of the group's choice, such as the use of interrupts, the creation of graphical interfaces for remote monitoring, multitasking or performance analysis. This project thus seeks to go beyond simple technical implementation, providing a complete vision of the potential of real-time embedded systems in the context of Smart Housing.

2 Requirements Analysis

Requirements analysis, both functional and non-functional, is essential to ensure that embedded systems effectively meet the needs of *Smart Housing* users, establishing operational criteria and performance conditions.

2.1 System A - Interior Lighting Control

Functional requirements:

- **Light Intensity Measurement:** The system must be able to measure the intensity of sunlight via the *LDR* sensor.
- **Dynamic Lighting Adjustment:** Based on the measured intensity, the system should adjust the interior lighting using *LEDs* on five predefined scales.
- **Serial monitoring:** Display the sensor input and *LED* output values on the serial monitor for debugging and monitoring purposes.

Non-functional requirements:

- **Energy** efficiency**:** The system should aim for energy efficiency, ensuring that lighting is adjusted to optimize energy consumption.
- **Real-time** response**:** The system's response to lighting adjustments must be fast and in real time to maintain constant illumination.

2.2 System B - Climate Control

Functional Requirements:

- **Temperature control:** The system should monitor the ambient temperature via a sensor and trigger the cooling fan when the temperature exceeds 24°C.
- **Status Indication:** Use *LEDs* to indicate the status of the fan (cooling or stabilized) and show the status on the *LCD* display.
- ***Display* control :** The brightness of the *LCD display* must be controlled by means of a potentiometer.

Non-functional requirements:

- **Energy Efficiency:** The system should be designed to optimize energy consumption by switching off the fan when it is not needed.
- **User Interface:** Ensure that the information shown on the *LCD* display is clear and easily understandable.

2.3 System C - Security System (Alarm)

Functional Requirements:

- **Intrusion detection:** The system must detect the movement of intruders by means of a *PIR* sensor.
- **Luminous and audible alarm:** Activates a flashing light signal and an audible alarm signal for 10 seconds whenever movement is detected.
- **Alarm disarming:** Integrate a push button to disarm the alarm.

Non-functional requirements:

- **Security:** Ensure system security by allowing disarming only via the designated button.
- **Alarm receptivity:** The alarm must repeat the beep at 5-second intervals until it is disarmed.

2.4 System D

Functional Requirements:

- **Lighting Control:** The system must be able to turn lights on/off in response to commands received by the *IR* sensor.
- **Garage Control:** The system must activate a motor to open the garage when it receives specific commands via *IR*.

Non-functional requirements:

- **Energy Efficiency:** The system should be optimized to minimize energy consumption by turning off lights or motors when they are not in use.
- **Quick Response to *IR* Command:** Ensure that the system responds quickly and accurately to commands received via the *IR* sensor.

2.5 Additional requirements

Figure 2: Interrupts

Use of *interrupts*:

Implement *interrupts* in one of the systems to guarantee a rapid response to specific events.

Figure 3: Monitoring

Graphical Interface for Remote Monitoring:

Create a graphical interface for real-time remote monitoring on one of the systems.

Figure 4: *Multitasking*

Multitasking :

Implement *multitasking*, with *Timer* or *RTOS*, to concurrently manage the execution time of 2 or more components in one of the systems.

Figure 5: Performance Analysis

Performance Analysis:

Carry out a comparative performance analysis between two versions of a system program, taking energy efficiency into account.

3 System specification

3.1 System A - Interior Lighting Control

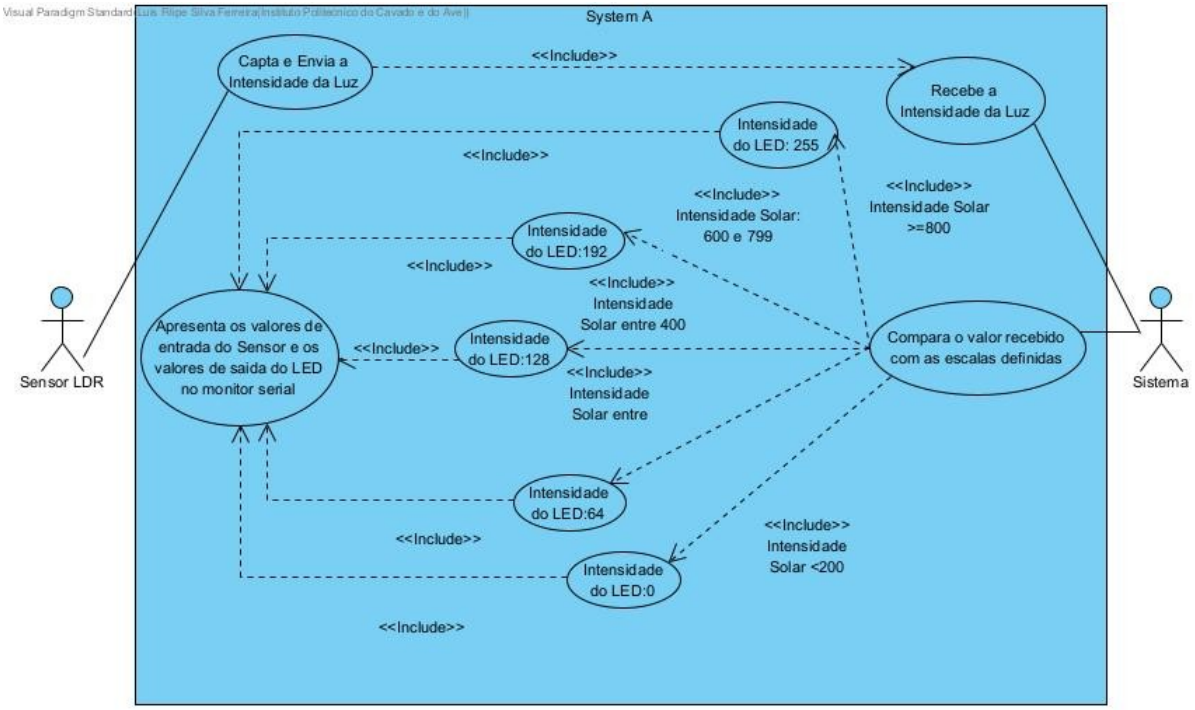

Figure 6: System A Use Case

Hardware System Architecture:

- **LDR sensor:** Measures the intensity of sunlight.
- **LED:** Adjusts the lighting according to the intensity scales.
- **Serial Monitor:** Displays sensor input and LED output values.

Software System Architecture:

- **Sensor reading:** Captures the light intensity measured by the LDR sensor.
- **Control Logic:** Based on the intensity scales, it determines the lighting setting.
  1. ≥ 800 LED 255;
  2. ≥ 600 and < 800 LED 192;
  3. ≥ 400 and < 600 LED 128;
  4. ≥ 200 and < 400 LED 64;
  5. < 200 LED 0.
- **LED control:** Activates the LED according to the scales set.
- **Monitoring:** Displays sensor and LED values on the serial monitor.

3.2 System B - Climate Control

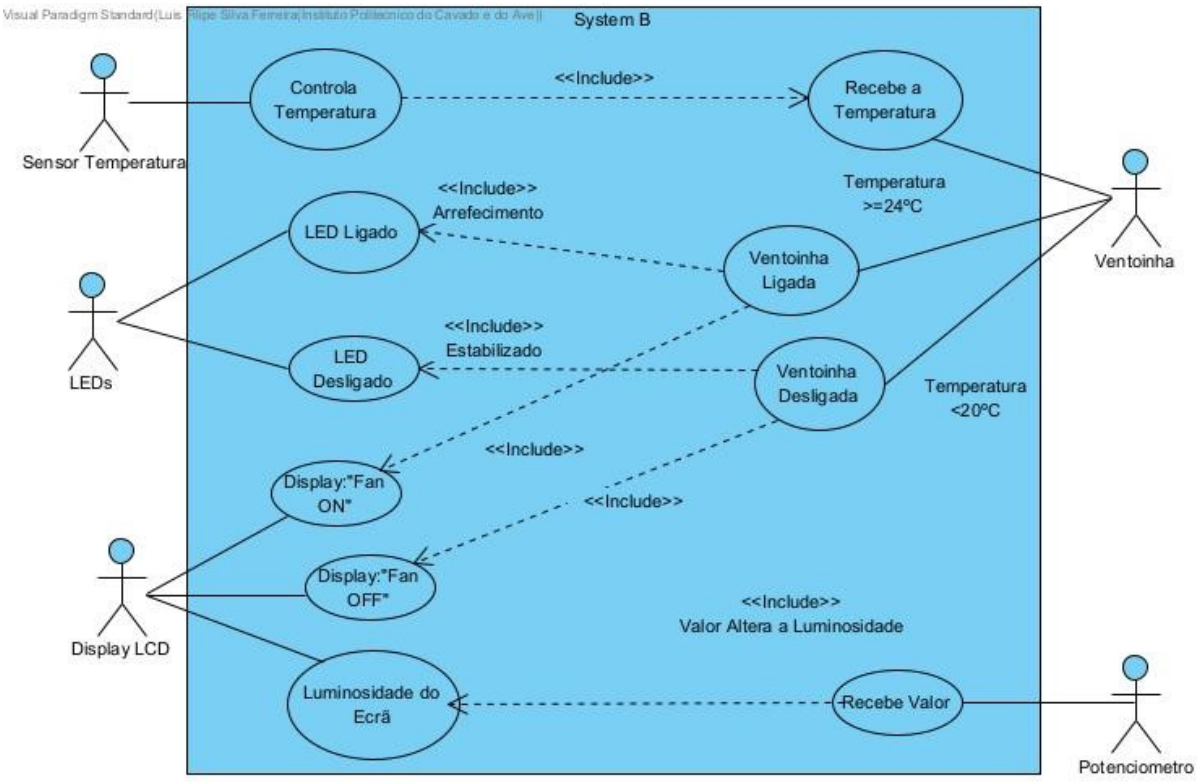

Figure 7: System B Use Case

Hardware System Architecture:

- **Temperature Sensor:** Measures the ambient temperature.
- **Fan:** Controls cooling based on temperature.
- **LEDs and LCD display:** Indicate system status and temperature.
- **Potentiometer:** Controls the brightness of the display.

Software System Architecture:

- **Sensor reading:** Captures temperature values.
- **Control Logic:** Decides when to turn the fan on or off.
  1. ≥ 24° C Switch on fan;
  2. < 20° C Switch off fan.
- **LED control and display:** Indicates status and displays temperature.
  1. When the fan is on, the red LED turns on and the green LED turns off;
  2. When the fan is off, the green LED turns on and the red LED turns off;
  3. The 1ª line of the LCD shows the fan status ("Fan ON/ Fan OFF");
  4. The 2ª line of the LCD shows the temperature ("Temp: xxxºC").
- **Potentiometer control:** Adjusts the brightness of the display.

3.3 System C - Security System (Alarm)

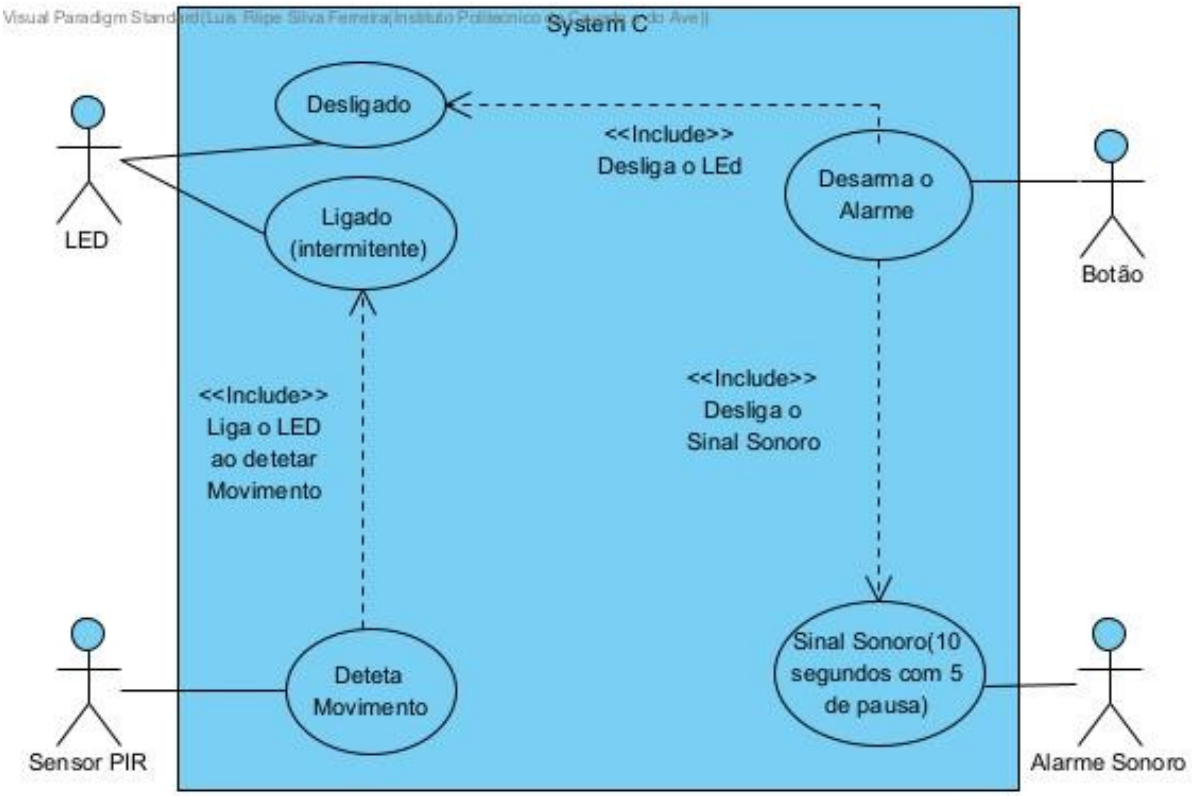

Figure 8: System C Use Case

Hardware System Architecture:

- **PIR sensor:** Detection of intrusive movements.
- **LED:** Flashing light signal.
- **Buzzer:** Alarm tone.
- **Push button:** To disarm the alarm.
- **Serial Monitor:** System monitoring.

Software System Architecture:

- **PIR sensor reading:** Detects intrusive movements.
- **Alarm Logic:** Activates the LED and buzzer in response to movement.
  1. Activates flashing red LED;
  2. 10-second audible signal;
  3. Repeat this beep at 5-second intervals.
- **Button control:** Allows the alarm to be disarmed using a *FALLING* type *Interrupt*.
- **Monitoring:** Displays information on the serial monitor.

3.4 System D - Command Control System

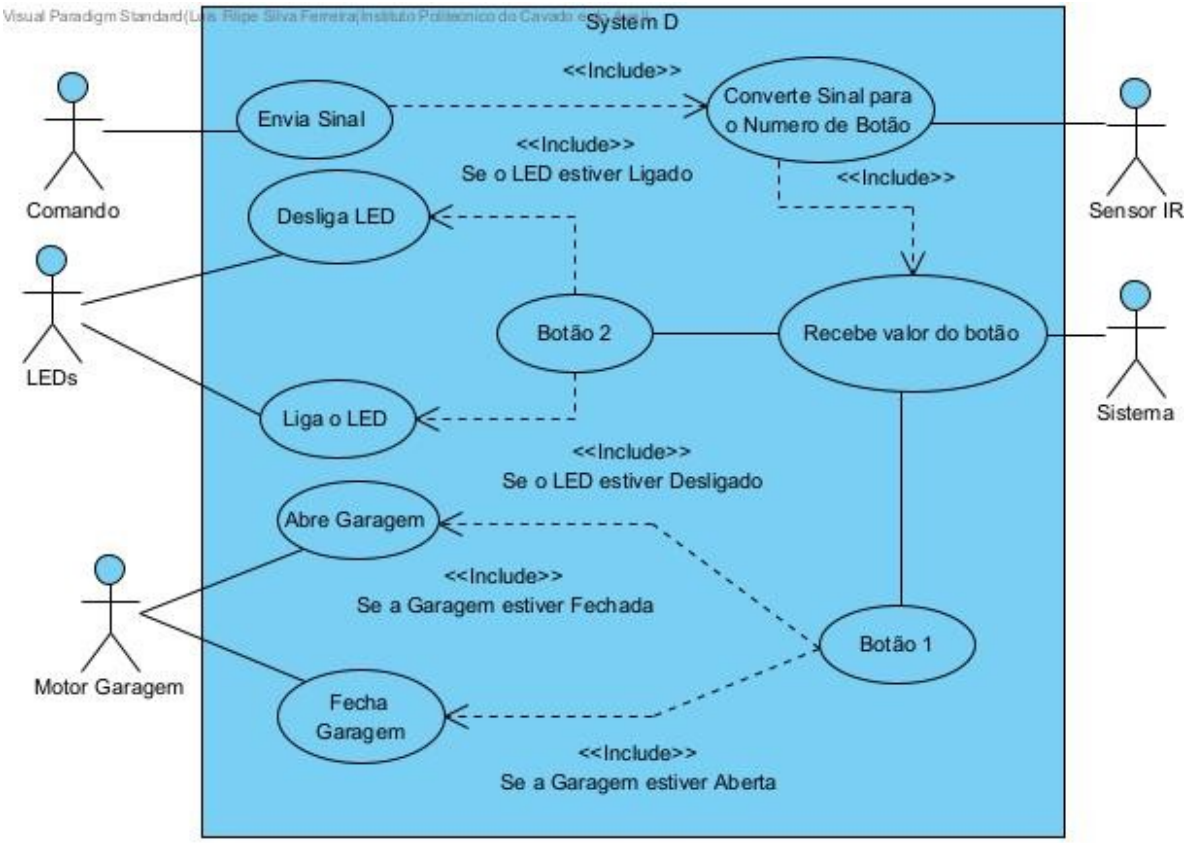

Figure 9: System D Use Case

Hardware System Architecture:

- **IR sensor:** Receive infrared commands.
- **LEDs:** Light signal.
- **Motor:** Electronic motor to open and close the garage in response to IR...
- **Push button:** To disarm the alarm.
- **Serial Monitor:** System monitoring.

Software System Architecture:

- **Reading the IR Sensor:** Interpreting incoming commands.
- **Control Logic:** Analyze incoming IR commands and determine corresponding actions.

1. **Button 1:** Controls an LED in room 3;
2. **Button 2:** Controls an LED in room 4;
3. **Button 3:** Controls the garage motor.

4 Design model

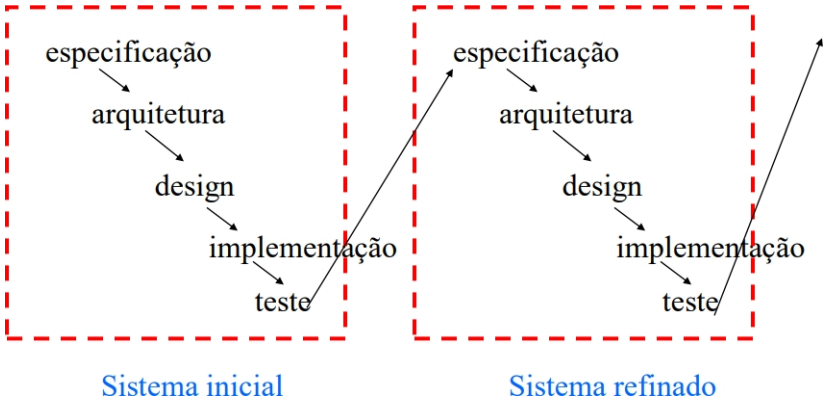

Figure 10: Successive Refinement Model

The process of integrating the four independent systems was conducted gradually, following the principles of the Successive Refinement Model. This model consists of an iterative approach made up of five distinct phases: *specification*, *architecture*, *design*, *implementation* and *testing*. In each iteration, a new system was developed and then improved with the integration of the subsequent system, a method that promoted the continuous evolution of the whole, as illustrated in Figure 10.

Figure 11: Specification

Specification:

Each system was initially specified, outlining the specific requirements and functionalities. Individual systems were designed based on particular needs, such as lighting, climate control, security and automation.

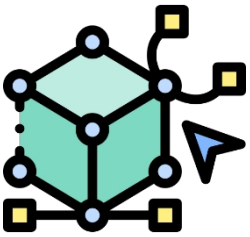

Figure 12: Architecture

Architecture:

Once specified, the systems were designed taking into account the overall structure and interrelationship between their components. A modular architecture was designed to facilitate future integration and guarantee the cohesion of the whole.

Figure 13: Design

Design:

The design of each system was based on the defined architecture. Specific components, such as LEDs, sensors and actuators, were designed to fulfill their individual functions, while common elements were identified to promote interoperability.

Figure 14: Implementation

Implementation:

Each system was implemented according to the established design. The implementation took into account specific requirements and the independent functionality of each system, ensuring that each one could operate autonomously.

Figure 15: Test

Test:

After the individual implementation of each system, exhaustive tests were carried out to validate functionality and identify possible improvements. This phase ensured that each system was operating correctly before the integration stage.

The systems were integrated progressively, starting with the combination of two systems and progressing to the complete integration of all four systems. Each integration stage was followed by a refinement phase, in which the resulting system was improved and optimized with the inclusion of the next system.

Benefits of the Iterative Integration Approach:

Figure 16: Incremental development

Incremental development:

Integration was conducted in gradual steps, allowing for continuous improvements and adjustments as new systems were incorporated.

Figure 17: Continuous Validation

Continuous Validation:

Regular tests were carried out during each integration phase, ensuring that each integrated system maintained its functionality and contributed to the whole in an efficient manner.

Figure 18: Flexibility and Adaptability

Flexibility and Adaptability:

The iterative approach allowed constant adaptation to specific requirements, enabling a dynamic response to emerging needs.

By following this methodology, a harmonious integration of the four systems was achieved, resulting in a cohesive, efficient whole capable of meeting the diverse demands of a Smart Housing environment.

5 Building the system

Before assembling each system on the model, we began by using the *TinkerCad* platform.

TinkerCad is a free *web* platform for *3D design*, electronics and coding, where you can build and simulate the execution of a huge variety of circuits. 2

5.1 System A

The components needed to build this system were essentially a *breadboard*, 1 *LED*, 1 220 *Ohm* resistor, 1 10K *Ohm* resistor, a light sensor, the *Arduino*, a USB type A-B cable to connect the *Arduino* to the computer and electrical cables to connect the *Arduino to* the *breadboard*.

In short, depending on the sunlight intensity value captured by the light sensor, the *LED* intensity is adjusted according to 5 scales and returned to the serial monitor r the sensor input and *LED* output value.

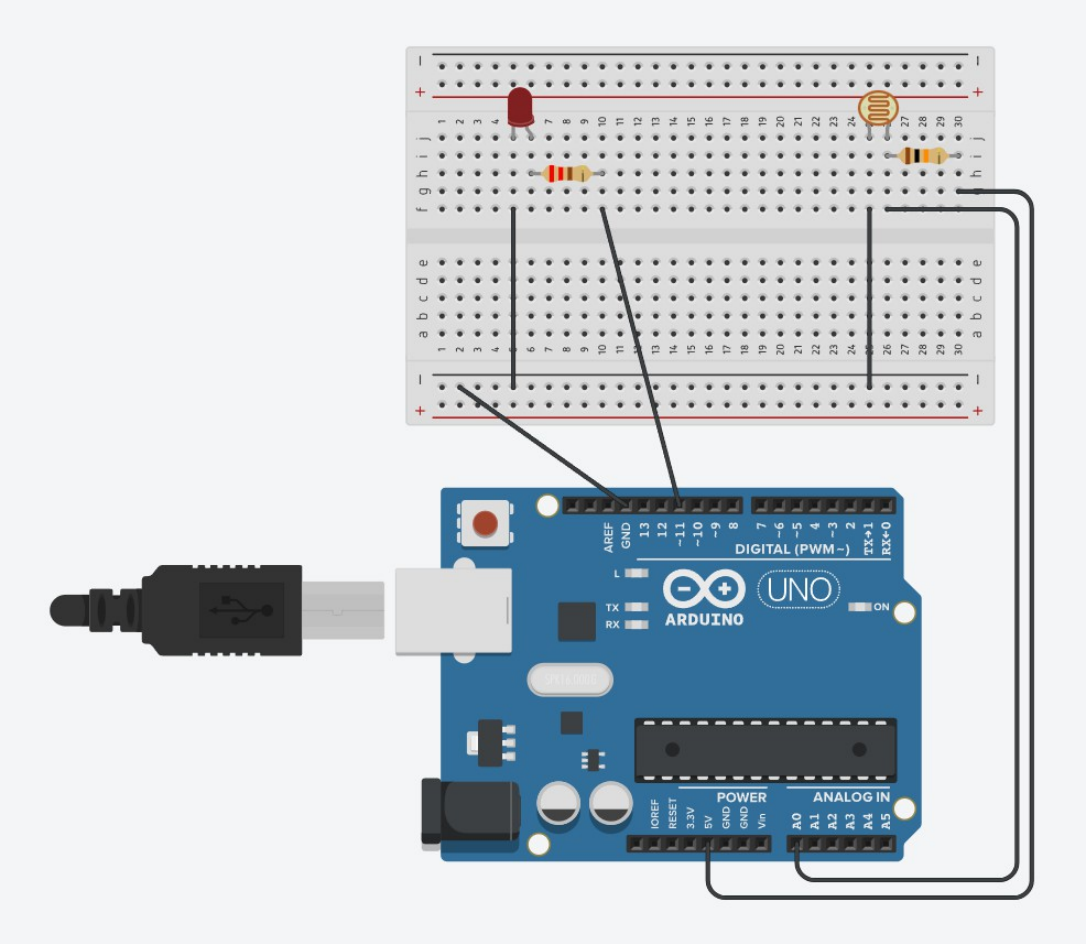

Figure 19: System A circuit

5.2 System B

The components used to assemble this system were a *breadboard*, 1 red *LED* for when the fan is on, 1 green *LED* for when the fan is off, 3 220 *Ohms* resistors, a temperature sensor, 1 potentiometer, 1 *LCD display*, 1 fan, the *Arduino*, a USB type A-B cable to connect the Arduino to the computer and electrical cables for connections between the *Arduino* and the *breadboard*.

For assembly on the model, we used a temperature and humidity sensor to enrich the system.

Essentially, the *Arduino* receives the values from the temperature and humidity sensor, when the temperature exceeds 24 degrees *Celsius* the fan is switched on along with the red *LED*, when the temperature drops to 20 degrees the fan is switched off and the red *LED* turns off and the green *LED* turns on. The temperature and humidity values are shown on the *LCD display* along with the fan status (*ON/OFF*). The potentiometer controls the brightness of the *LCD display*.

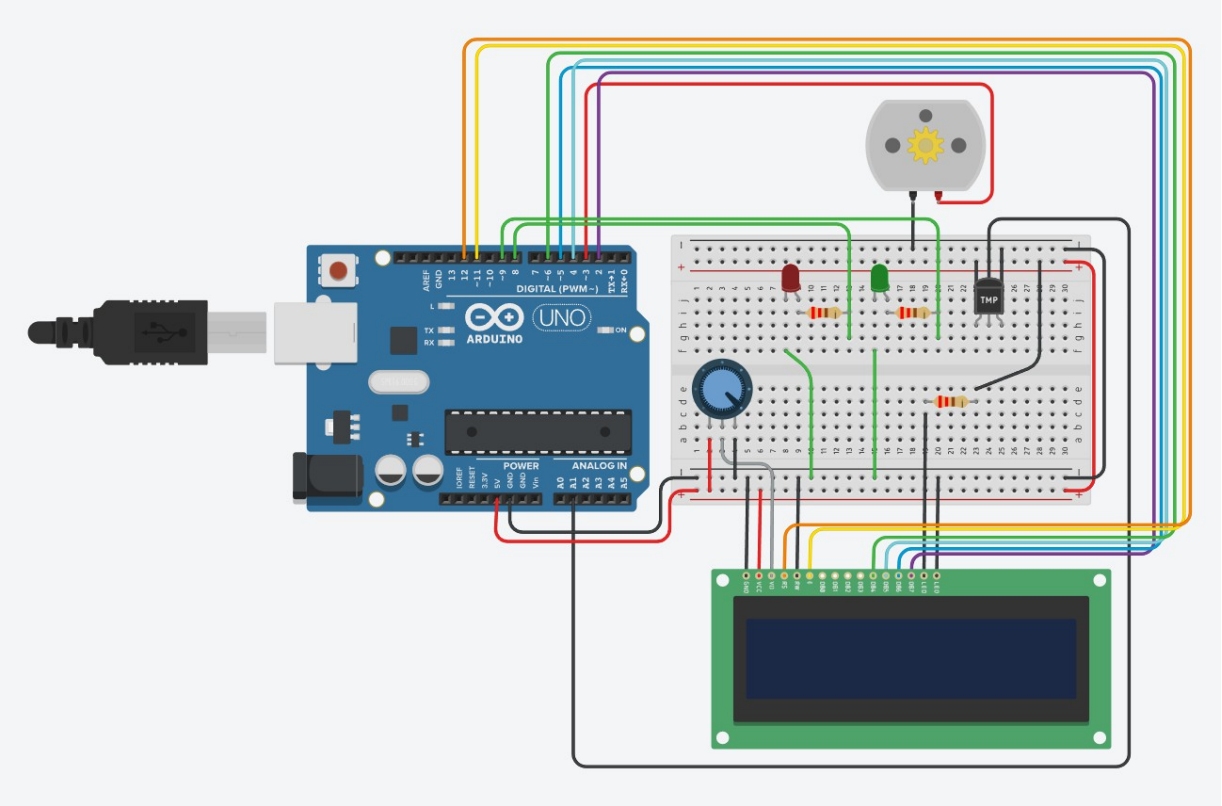

Figure 20: System B circuit

5.3 System C

The components used to assemble this system were a *breadboard*, 1 red *LED* to turn on intermittently when movement is detected, 1, 2 re- sistance of 220 *Ohms*, motion sensor, 1 *buzzer* to serve as an alert, 1 button to turn off the alarm, the *Arduino*, a USB type A-B cable to connect the Arduino to the computer and electrical cables for connections between the *Arduino* and the *breadboard*.

Essentially, the *Arduino* receives the values from the motion sensor, and as soon as motion is detected, the *LED* turns on intermittently and the *buzzer* emits a sound signal lasting 10 seconds at 5-second intervals; after the button is pressed, the *LED* turns off and the *buzzer* stops emitting sound.

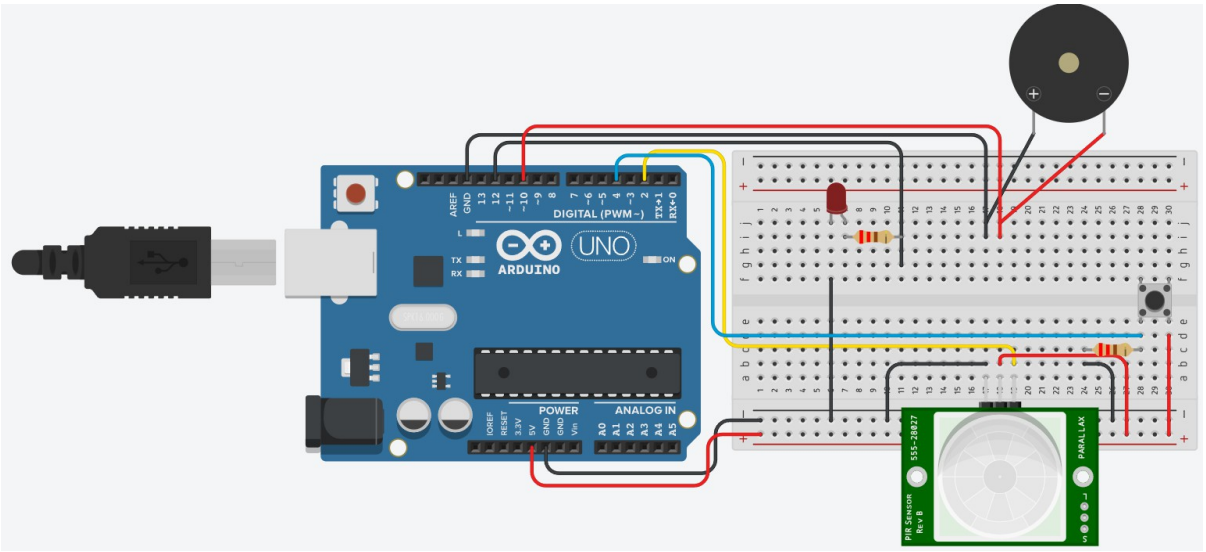

Figure 21: System circuit C

5.4 System D

The components used to assemble this system were a *breadboard*, 1 *servo* motor, 1 infrared sensor, 1 *LED*, 1 *220Ohms* resistor, 1 remote control, the *Arduino*, a USB type A-B cable to connect the Arduino to the computer and electrical cables to connect the *Arduino* to the *breadboard*.

Essentially, the infrared sensor is constantly listening for a signal from the control unit, after receiving the signal from button 1 (optional in the implementation) it rotates the *servo* motor 90 degrees anticlockwise, when it once again receives the signal from button 1 it rotates the *servo* motor clockwise.

The same is produced for the *LED*, which is switched on and off as the signal from button 2 is received.

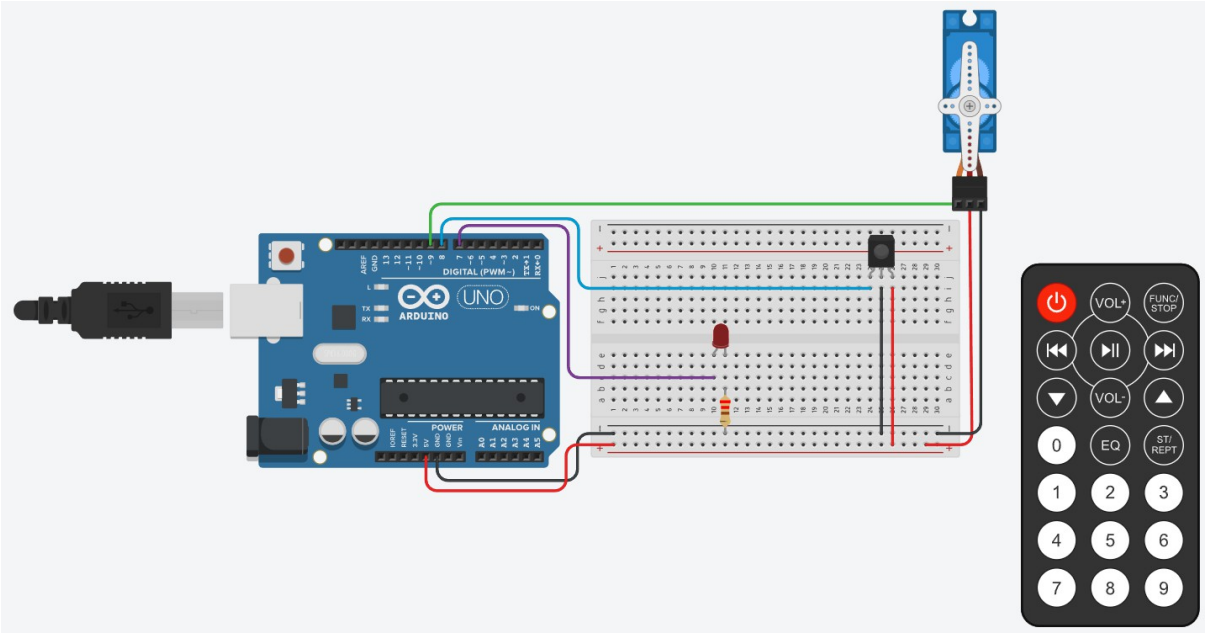

Figure 22: System D circuit

6 Coding

6.1 Code System A

```
void TaskSystemA(void *pvParameters) {
  // Avoid compiler warning about unused parameter
  (void)pvParameters;

  // Variables to store light sensor and LED
  values int val_light = 0, val = 0;

  // Infinite loop for continuous task execution
  while (1)
  {
    // Read analog value from the light
    sensor val_light = analogRead(Light);

    // Map the light sensor value to LED brightness
    if (val_light >= 800) val = 255;
    else if (val_light >= 600) val = 192;
    else if (val_light >= 400) val = 128;
    else if (val_light >= 200) val = 64;
    else val = 0;

    // Print light sensor and LED values to Serial Monitor
    Serial.print("Light Sensor: ");
    Serial.print(val_light);
    Serial.print(" Light Value:
    ");
    Serial.println(val);

    // Adjust LED brightness based on light sensor
    value if (statesLed[0] != 0) analogWrite(Room3,
    val);
    if (statesLed[1] != 0) analogWrite(Room4, val);

    // Delay to control the update rate of the
    task vTaskDelay(pdMS_TO_TICKS(100));
  }
}
```

Listing 1: SystemA.ino

The code presents the implementation of an indoor lighting control system using an Arduino, a light sensor (LDR), and LEDs. The concept behind this system is to dynamically adapt the lighting of the interior space based on the intensity of sunlight. The LDR sensor takes readings of the ambient light intensity, and the algo-rhythm maps these readings to different LED brightness levels, creating five scales

different. In this way, as the sunlight varies, the interior lighting is adjusted to ensure constant brightness and promote greater energy efficiency.

The continuously running code (infinite loop) reads the analog value of the light sensor, determines the corresponding brightness level and adjusts the intensity of the LEDs accordingly. The sensor's input values and the LEDs' output are displayed on the serial monitor for monitoring and debugging.

Essentially, this system represents a practical approach to creating an intelligently lit environment, adapting effectively to ambient light conditions while providing efficient energy management.

6.2 Code System B

```
float hum, temp; // Global variables to store humidity and temperature

void TaskSystemB(void *pvParameters) {
  // Avoid compiler warning about unused parameter
  (void)pvParameters;

  // Flag to track whether cooling is active
  bool isCooling = false;

  // Infinite loop for continuous task execution while (1) {
    // Read humidity and temperature from the DHT sensor
    hum = dht.readHumidity();
    temp = dht.readTemperature();

    // Update LCD with temperature and humidity values lcd.setCursor(5, 1);
    lcd.print(int(temp));
    lcd.setCursor(13, 1);
    lcd.print(int(hum));

    // Cooling control based on temperature if (!isCooling && temp > 24) {
      digitalWrite(Room1R, HIGH);
      digitalWrite(Room1G, LOW);
      digitalWrite(AC, HIGH);
      lcd.setCursor(0, 0);
      lcd.print("Fan ON "); isCooling = true;
    } else if (isCooling && temp < 20) {
      digitalWrite(Room1R, LOW);
      digitalWrite(Room1G, HIGH); digitalWrite(AC, LOW);
      lcd.setCursor(0, 0);
      lcd.print("Fan OFF");
      isCooling = false;
    }

    // Delay to control the update rate of the task
    vTaskDelay(pdMS_TO_TICKS(100));
  }
}
```

```
44 void requestEvent() {
```

```
  // Respond to I2C master request with temperature and
   humidity values
46 Wire.write((uint8_t* )&temp,
  sizeof(float)); Wire.write((uint8_t*
48 )&hum, sizeof(float));
}
```

Listing 2: SystemB.ino

The code presents the implementation of a climate control system that uses a temperature and humidity sensor (DHT sensor), a fan, LEDs and an LCD display. The system regulates the room temperature by activating the fan when the temperature exceeds 24 degrees Celsius and deactivating it when the temperature is below 20 degrees Celsius.

The LCD display shows relevant information, such as the fan status ("Fan ON" or "Fan OFF") on the 1st line and the current temperature on the 2nd line. In addition, two LEDs visually indicate the status of the room: the red LED turns on when the fan is running, while the green LED indicates that the room is stabilized.

The brightness of the LCD display is controlled via a potentiometer, allowing the visibility to be adjusted as required. In addition, the code responds to requests from an I2C master, providing the temperature and humidity values on demand.

In short, this system aims to create a climate-controlled and efficient environment, adapting dynamically to temperature conditions and providing visual feedback via LEDs and the LCD display.

6.3 Code System C

```
// Function to play a tone on the buzzer
void playTone(int frequency) {
  unsigned int period = 1000000 / frequency;
  unsigned int halfPeriod = period / 2;
  unsigned int cycles =           *
  unsigned     i;                   frequency12
               5 / 1000;
  // Generate the specified tone
  for (i = 0; i < cycles; ++i) {
    digitalWrite(Buzzer, HIGH);
    delayMicroseconds(halfPeriod);
    digitalWrite(Buzzer, LOW);
    delayMicroseconds(halfPeriod);
  }
}
```

Listing 3: SystemC.ino

This code implements a function called playTone which is responsible for generating a sound on a buzzer based on a specified frequency.

```
bool isActive = false, found = false;

void TaskSystemC(void *pvParameters) {
  // Avoid compiler warning about unused parameter
  (void)pvParameters;

  unsigned long time;
  bool isToAwait =
  false;
  byte ledState = HIGH;

  // Infinite loop for continuous task
  execution while (1) {
    // Update LCD with isActive state
    lcd.setCursor(8, 0);
    lcd.print(isActive);

    // If not active, delay and continue to the next
    iteration if (!isActive) {
      digitalWrite(Buzzer, LOW);
      digitalWrite(Room2, LOW);
      vTaskDelay(pdMS_TO_TICKS(100));
      continue;
    }

    // PIR sensor detected motion
    if (digitalRead(PIR) && !found) {
      time =
      millis(); found
      = true;
      isToAwait = false;
    }

    // Execute actions when motion is detected
    if (found && !isToAwait)
      { playTone(1000);
      digitalWrite(Room2, ledState);
      ledState = !ledState;

      // Check if the elapsed time is greater than 10 seconds
      if (millis() - time > 10000)
        { digitalWrite(Buzzer,
        LOW);
        ledState = LOW;
        digitalWrite(Room2, LOW);
        isToAwait =
        true; time =
        millis();
```

```
45              }
            }
```

```
47
      // Wait for 5 seconds after the action and reset
49    if (isToAwait && millis() - time > 5000) {
        isToAwait = false;
51      time = millis();
      }
53
      // Delay to control the update rate of the task
55    vTaskDelay(pdMS_TO_TICKS(100));
    }
57 }
```

Listing 4: SystemC.ino

This code represents the implementation of a security system that uses a PIR (Passive Infrared) motion sensor. When the system is active (isActive is true) and the PIR sensor detects movement, various alarm actions are triggered. These actions include activating an audible signal (Buzzer), switching the state of an LED (Room2) between on and off to create a red flashing effect, and measuring the duration of these actions.

The system waits 5 seconds after each alarm trigger before it can be activated again, ensuring intervals between consecutive events. The time elapsed from the start to the end of each iteration of the cycle is recorded and printed on the serial terminal for performance monitoring.

This security system provides visual and audible alerts when movement is detected by the PIR sensor, and it can be deactivated by pressing a button, although the specific implementation of the button is not included in the code supplied.

```
void interruptHandler() {
  // Toggle the isActive state and reset variables
  isActive = !isActive;
  found = false;
  delay(10); // Small delay to debounce the button
  digitalWrite(Buzzer, LOW);
  digitalWrite(Room2, LOW);
}
```

Listing 5: SystemC.ino

This code implements a function called interruptHandler which is responsible for stopping the Buzzer and deactivating the alarm when the button is pressed.

6.4 System code D

```
// Return -1, if supplied code does not map to a key.
int mapCodeToButton(unsigned long code) {
  // Check for codes from this specific remote
  if ((code &              == 0x0000FF00) {
  0x0000FFFF))
    // No longer need the lower 16 bits. Shift the code by 16
    // to make the rest easier.
    code >>= 16;
    // Check that the value and inverse bytes are complementary
   .
    if (((code >> 8) ^ (code & 0x00FF)) == 0x00FF) {
      return code & 0xFF; // Extract the lower 8 bits as the
   button code
    }
  }
  return -1; // Return -1 if the code doesn't match
   the expected format
}

int readInfrared() {
  int result = -1;
  // Check if we've received a new code
  if (IrReceiver.decode()) {
    // Get the infrared code
    unsigned long code = IrReceiver.decodedIRData.
   decodedRawData;
    // Map it to a specific button on the remote
    result = mapCodeToButton(code);
    // Enable receiving of the next value
    IrReceiver.resume();
  }
  return result;
}
```

Listing 6: SystemD.ino

This code implements a function called *readInfrared* which is responsible for stopping receiving the infrared value of the command when it is pressed, and also implements a function called *mapCodeToButton* which is responsible for decoding the hexadecimal value of the command into a value that is easier to read.

```
void openGate() {
  for (int i = 5; i < 170; i++) {
    myservo.write(i);
    delay(30);
  }
}

void closeGate() {
  for (int i = 170; i > 5; i--) {
    myservo.write(i);
    delay(30);
  }
}
```

Listing 7: SystemD.ino

This code implements two functions called *openGate* and *closeGate* which are responsible for opening and closing the gate.

```
void TaskSystemD(void* pvParameters)
{
  (void)pvParameters;
  unsigned long time;
  int button = -1;
  bool isOpen =
  false;

  while (1)
  {
    button = readInfrared();

    switch (button) {
      case 12: // Button
      1
        // Toggle LED state for Room3
        statesLed[0] = !statesLed[0];
        digitalWrite(Room3, LOW);
        break;
      case 24: // Button 2
        // Toggle LED state for Room4
        statesLed[1] = !statesLed[1];
        digitalWrite(Room4, LOW);
        break;
      case 94: // Button 3
        // Toggle gate state (open/close)
        if (isOpen) closeGate();
        else openGate();
        isOpen = !isOpen;
        break;
```

```
29          default:
              // No action for other buttons
31            break;
          }
33
          vTaskDelay(pdMS_TO_TICKS(100));
35      }
}
```

Listing 8: SystemD.ino

The *TaskSystemD* function plays a crucial role in a larger system that involves controlling LEDs and a gate via an infrared remote control. In each iteration of the cycle, the code checks for new code coming from the co-controller. infrared remote control using the *readInfrared* function.

When it detects a new code, the mapCodeToButton function is used to associate that code with a specific button on the remote control. The task then performs specific actions based on the button pressed.

- Button 1 (Code 12): Toggle the status of the LED associated with Room3. If it's on, it's off; if it's off, it's on.
- Button 2 (Code 24): Toggle the status of the LED associated with Room4 in the same way as Button 1.
- Button 3 (Code 94): Switches the status of the gate between open and closed. If the gate is open, it will be closed, and vice versa. The current state of the gate is kept in a variable called isOpen.

This task is essential for the system's interaction with the infra-red remote control, allowing the user to control the lighting (LEDs) and the status of the gate remotely.

7 Tests/Results

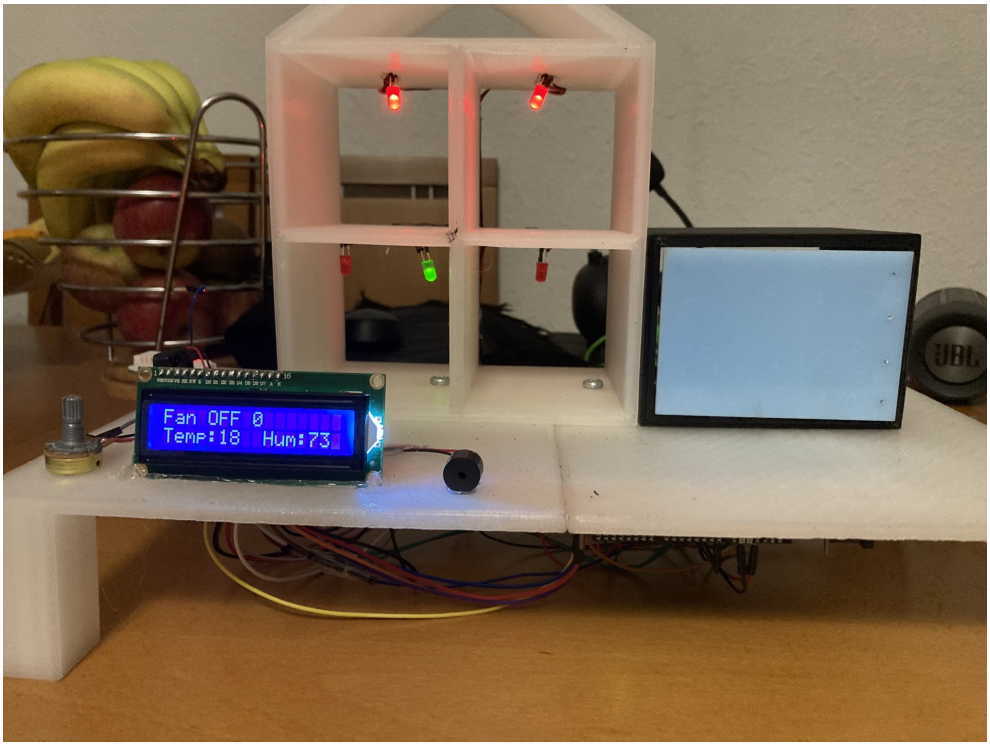

Figure 23: House

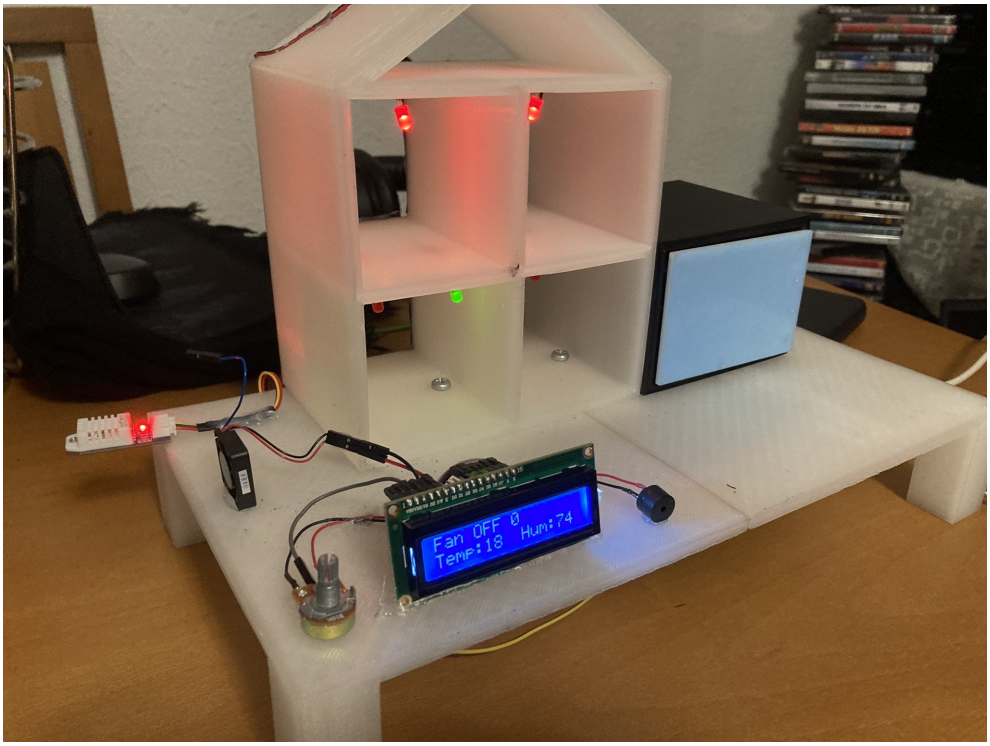

Figure 24: House

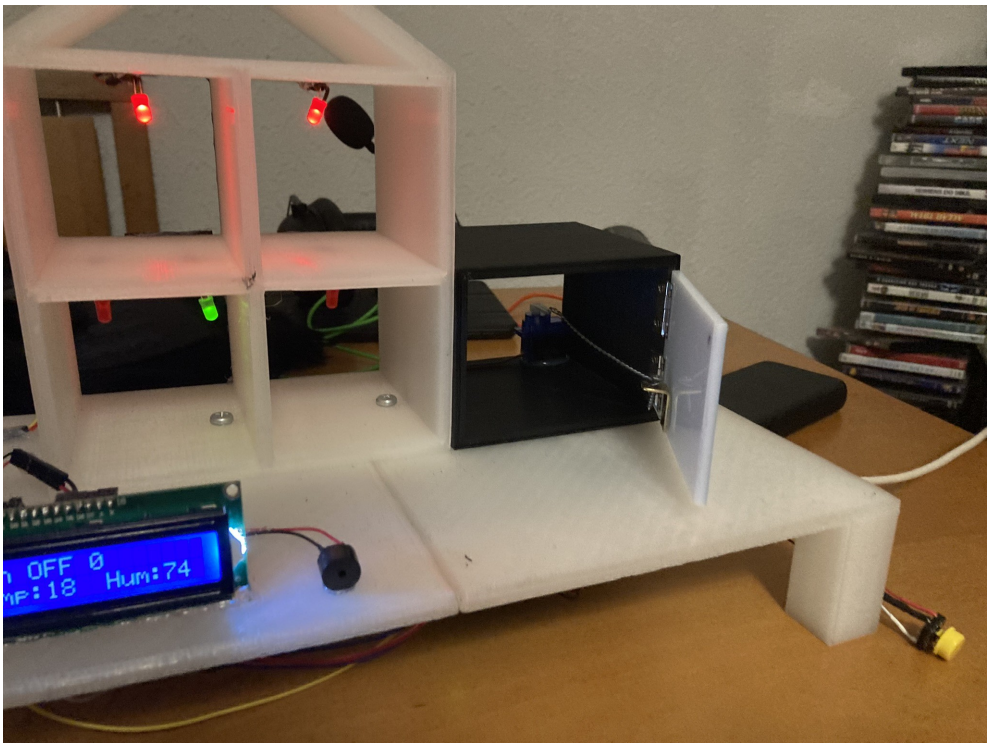

Figure 25: House

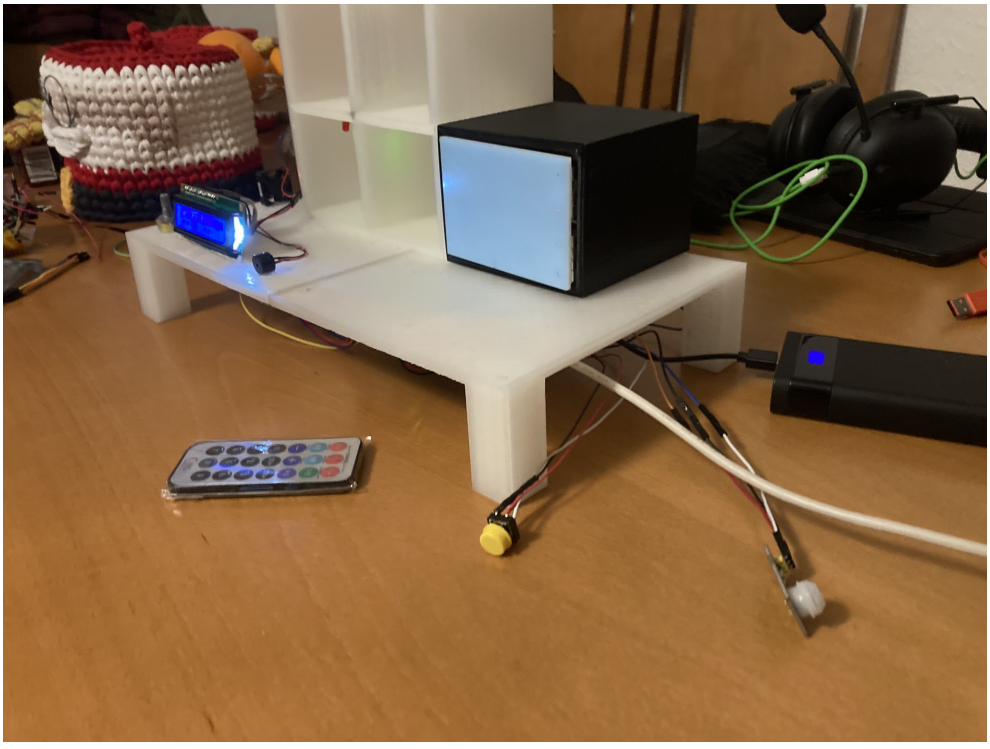

Figure 26: House

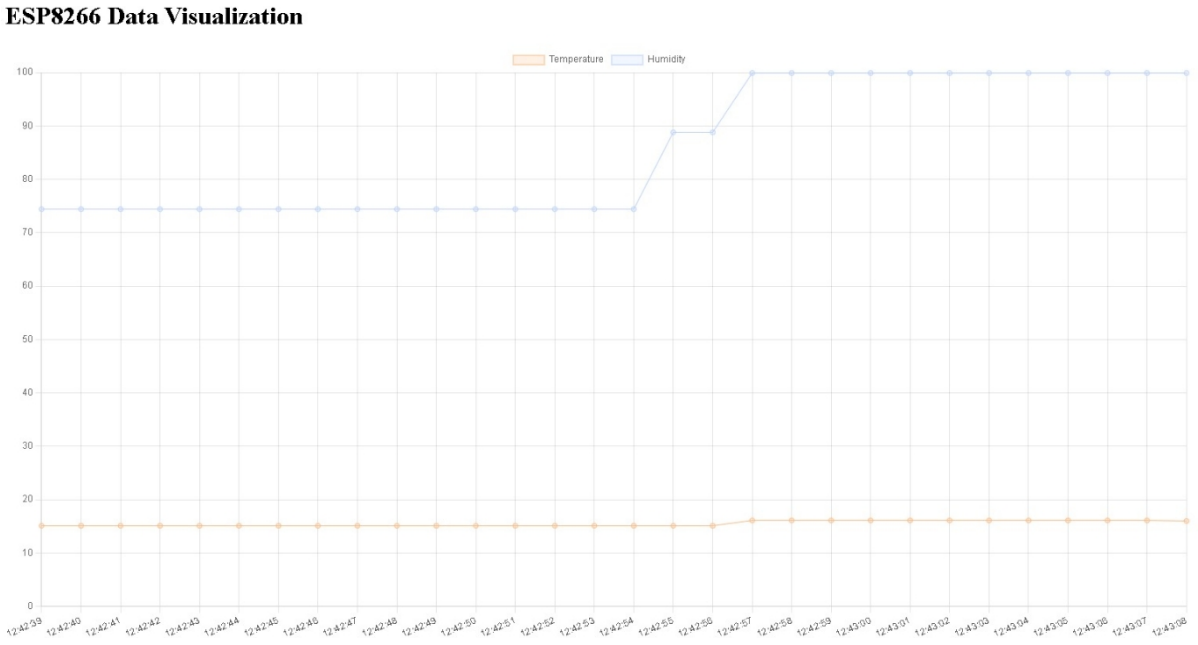

Figure 27: Temperature demonstration website

7.1 Non-optimized code System C

```
void TaskSystemCWorst(void* pvParameters) {
  // Avoid compiler warning about unused parameter
  (void)pvParameters;

  long time;
  bool isToAwait = false;
  byte ledState = HIGH;

  // Infinite loop for continuous task execution
  while (1) {
    // Update LCD with isActive state
    lcd.setCursor(8, 0);
    lcd.print(isActive);

    if (isActive) {
      // PIR sensor detected motion
      if (digitalRead(PIR) == HIGH) {
        if (!found) {
          time = millis();
          found = true;
          isToAwait = false;
        }
      }
      // Execute actions when motion is detected
      if (found) {
        if (!isToAwait) {
          playTone(1000);
          digitalWrite(Room2, ledState);
          ledState = !ledState;

          // Check if the elapsed time is greater than 10
   seconds
          if (millis() - time >= 10000) {
            digitalWrite(Buzzer, LOW);
            ledState = LOW;
            digitalWrite(Room2, LOW);
            isToAwait = true;
            time = millis();
          }
        }
      }

      // Wait for 5 seconds after the action and reset
      if (isToAwait) {
        if (millis() - time >= 5000) {
```

```
            isToAwait = false;
            time = millis();
          }
        }
      }
      else {
        digitalWrite(Buzzer, LOW);
        digitalWrite(Room2, LOW);
      }

      // Delay to control the update rate of the task
      vTaskDelay(pdMS_TO_TICKS(100));
    }
}
```

Listing 9: SystemC.ino

Using the code for System C was implemented in another way to test the efficiency of well-done code against badly-done code, and as you can see, this code is not only more difficult, it is also not optimized.

8 Conclusion

Throughout the development of this project, each group worked on the design and implementation of embedded systems to control lighting, climate control, security and additional tasks for a smart home. The initial proposal included the harmonious integration of these systems, culminating in a complete and functional solution.

The final stage of the project consisted of the successful integration of all the systems developed into a single cohesive environment. The fusion of these functionalities provided a fully operational smart home.

However, throughout the development process, we were faced with challenges and limitations. Resource constraints, both in terms of hardware and software, were faced. The complexity of integrating multiple systems also proved to be a critical point, requiring a careful approach to ensure the effectiveness and stability of the unified system.

Despite the challenges, the end result demonstrates the project's success in implementing embedded systems for Smart Housing. Each group made a significant contribution to creating a functional and energy-efficient smart home.

This project has provided valuable practical experience, highlighting the importance of integrating systems to achieve innovative and functional solutions. The difficulties faced along the way serve as learning opportunities, contributing to technical growth and improving systems engineering skills.

9 Bibliography

1. Project Github: https://github.com/TinoMeister/SETR;
2. TinkerCad platform: https://www.tinkercad.com/;
3. Download Arduino IDE: https://www.arduino.cc/en/software;
4. Arduino libraries: https://www.arduino.cc/libraries/