كلية العلوم التطبيقية - أيت ملول
ⵜⴰⵙⴳⴷⴰⵍⵜ ⵏ ⵜⵎⴰⵙⵙⴰⵏⵉⵏ ⵜⵉⵙⵏⴰⵙⵏⵉⵏ - ⴰⵢⵜ ⵎⵍⵍⵓⵍ
FACULTE DES SCIENCES APPLIQUEES – AIT MELLOUL

SESN/M1/2023-2024

Advancement Report

Master : Embedded Systems Digital Services

Smart IoT-Based Operator : RC Car WIFI Control System with Real time sensors data and Camera feed

Realized by :

BAARAR Mohamed

Supervised by :

Pr, Aimad Karkouch

Module :

Iot

Table of Contents

Table des figures

Summary

This project involves designing and implementing an IoT-based remote-controlled (RC) car using ESP32 microcontrollers, WiFi connectivity, and an Xbox 360 controller. The system leverages MQTT for communication, integrates sensor-based environmental adaptations for speed control, and includes a camera module for real-time video streaming. This only reflects the current advancement on the project so far, and the full vision is detailed and highlighted in the ”Future development and plans” section.

Chapitre 1

Conception

1 Introduction

This project aims to create an IoT-based RC car using ESP32 microcontrollers, WiFi connectivity, and an Xbox 360 controller for user input. The system leverages MQTT for communication and features sensor-based environmental adaptation for speed control and an integrated camera module for real-time video streaming.

2 System Architecture

The system architecture is divided into four main components :

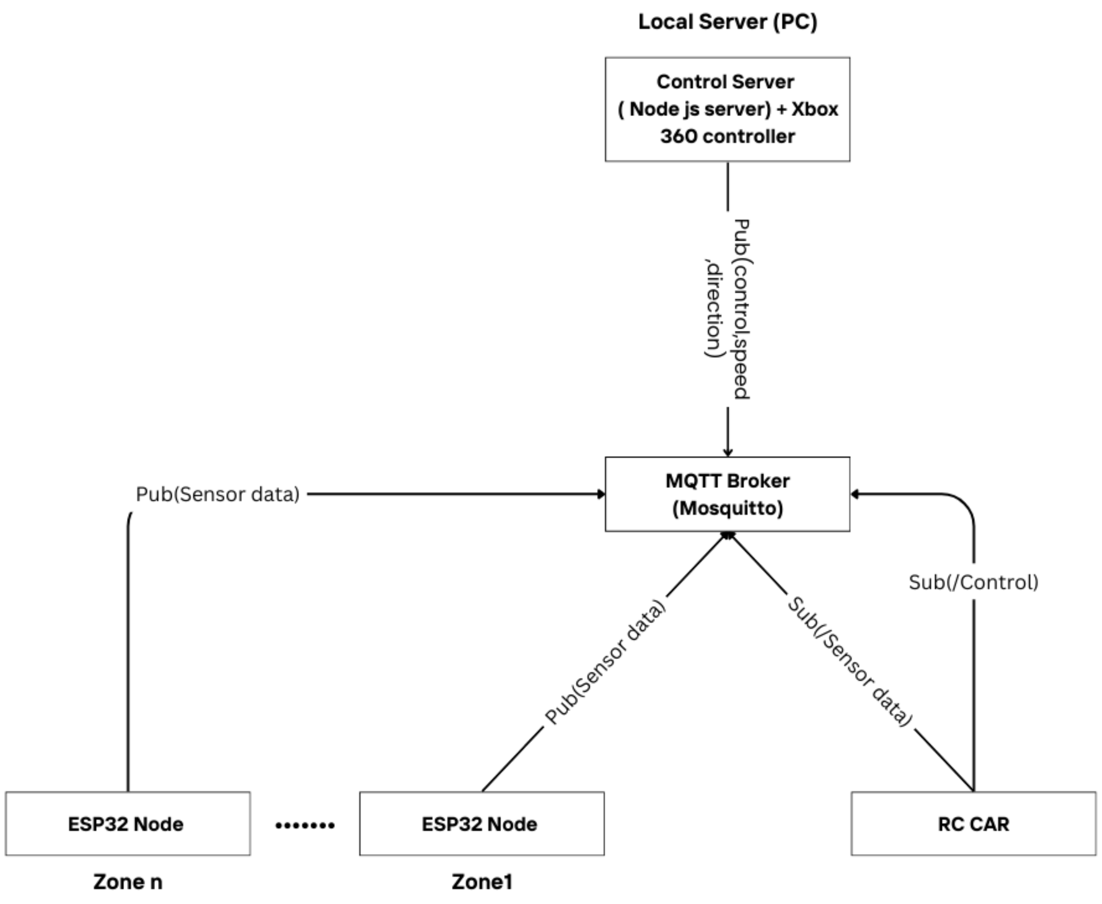
1. **Sensor Node (ESP32)** : Measures temperature and humidity and classifies weather conditions.

2. **Control Module (ESP32)** : Receives control commands from the MQTT broker and drives the RC car's motors.

3. **Camera Module (ESP32)** : Streams video from the car.

4. **Control Server (PC)** : Interfaces with an Xbox 360 controller to send control commands via MQTT.

2.1 Block Diagram



3 Concepts and Technologies

3.1 ESP32 Microcontroller

ESP32 is a low-cost, low-power microcontroller with integrated Wi-Fi and dual-mode Bluetooth. It is widely used in IoT applications due to its rich feature set, including :

- WiFi connectivity
- Bluetooth
- Multiple GPIO pins
- Built-in support for various sensors and peripherals

3.2 MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight messaging protocol designed for low-bandwidth, high-latency networks. It uses a publish/subscribe model, making it suitable for IoT applications where devices communicate intermittently.

Key Components :

- **Broker** : Central server that receives messages from clients and routes them to subscribers.
- **Client** : Device or application that publishes messages to topics or subscribes to topics to receive messages.

3.3 Sensors and Actuators

- **DHT11 Sensor** : Measures temperature and humidity.
- **Motor Driver** : Controls the motors of the RC car based on speed and direction commands.
- **I2C Displays** : Display informations such as speed, weather ...

3.4 Video Streaming

An ESP32 camera module captures and streams video, providing real-time visual feedback to the user.

3.5 Xbox Controller

An Xbox 360 controller connected to a PC provides an intuitive interface for controlling the RC car.

4 Protocols and Communication

4.1 WiFi Communication

WiFi is used for :

- Connecting ESP32 modules to the local network.
- Enabling communication between ESP32 modules and the MQTT broker.

4.2 MQTT Protocol

MQTT is used for :

- Publishing sensor data from the sensor module to the MQTT broker.
- Publishing control data (speed, direction) from the control server to the MQTT broker.
- Subscribing to control commands from the control server to the control module.
- Subscribing to sensor data updates by the control module for speed and direction adjustments.

Setup :

- **Broker** : Mosquitto broker running on a local machine (IP : `localhost`, Port : `8883`).
- **Clients** : ESP32 modules and the control server (PC) act as MQTT clients.

4.3 SSL/TLS Security

Secure communication between the MQTT broker and clients is ensured using SSL/TLS.

Certificates :

- **CA Certificate** : Used by clients to verify the broker's identity.
- **Client Certificate** : Used by the broker to verify the client's identity (not used in this basic setup).

5 Security Considerations

5.1 Network Security

- **MQTT Security** : Use SSL/TLS to encrypt communication between clients and the broker, preventing eavesdropping and man-in-the-middle attacks.

5.2 Authentication and Authorization

- **MQTT Authentication** : Clients authenticate with the broker using a username and password.
- **Authorization** : Configure the broker to allow only authorized clients to publish or subscribe to specific topics.

6 Implementation Details

6.1 Sensor Module

- **ESP32 Setup** : Connect the DHT11 sensor and OLED display to the ESP32.
- **WiFi Connection** : Connect to the local WiFi network.
- **MQTT Client** : Publish sensor data to the broker at regular intervals.

6.2 Control Module

- **ESP32 Setup** : Connect motor driver and OLED display to the ESP32.
- **MQTT Client** : Subscribe to control commands and sensor data updates.
- **Motor Control** : Adjust motor speed and direction based on received commands and data feed from the sensor nodes (speed limiter).

6.3 Camera Module

- **ESP32 Camera** : Capture and stream video to a specified endpoint.

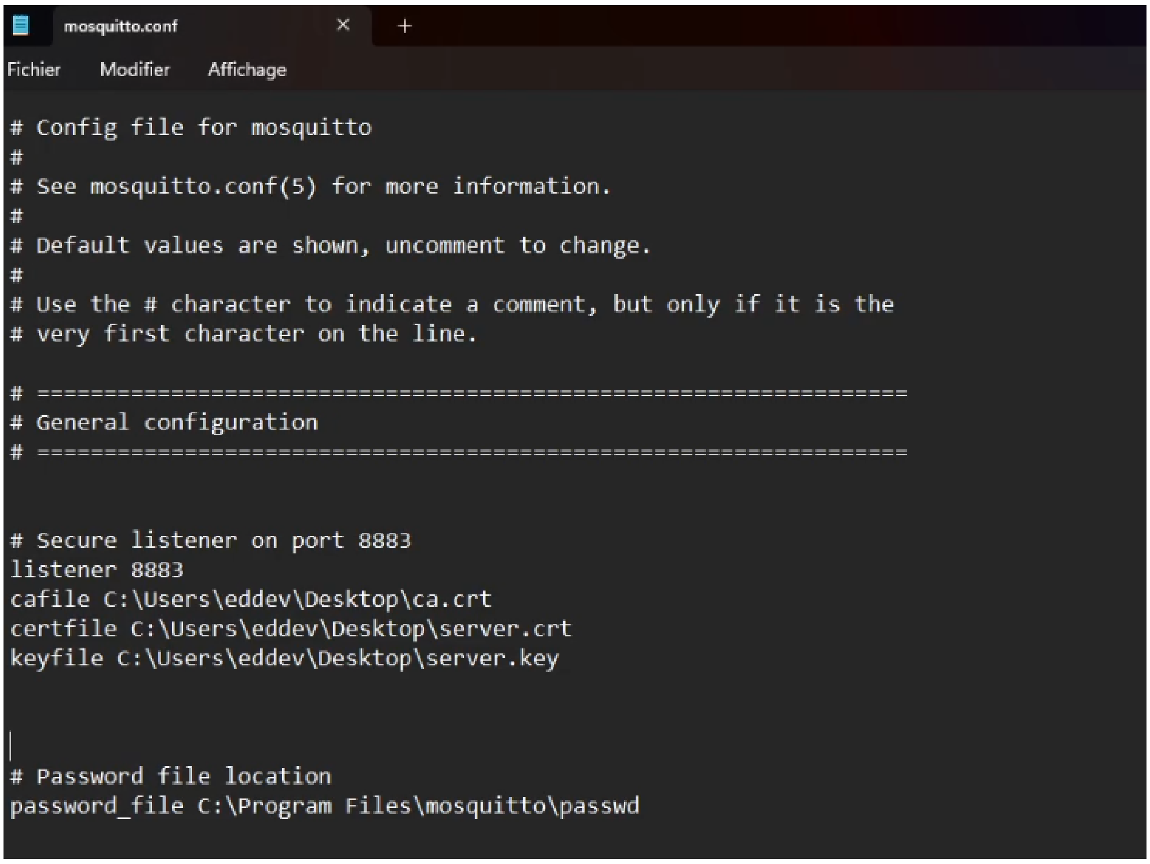
6.4 Control Server (PC)

- **Xbox Controller Interface** : Use Node.js and `node-hid` to read inputs from the Xbox controller.
- **MQTT Client** : Publish control commands based on user input to the broker.

Chapitre 2

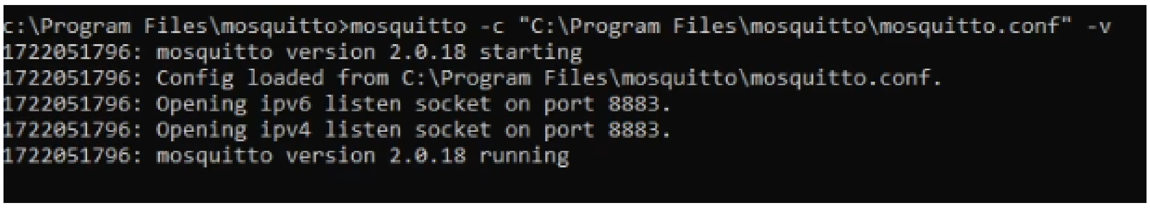
Realisation

1 mosquitto Broker setup



```
# Config file for mosquitto
#
# See mosquitto.conf(5) for more information.
#
# Default values are shown, uncomment to change.
#
# Use the # character to indicate a comment, but only if it is the
# very first character on the line.

# =================================================================
# General configuration
# =================================================================


# Secure listener on port 8883
listener 8883
cafile C:\Users\eddev\Desktop\ca.crt
certfile C:\Users\eddev\Desktop\server.crt
keyfile C:\Users\eddev\Desktop\server.key



# Password file location
password_file C:\Program Files\mosquitto\passwd
```

Fig. 2.1 : broker config file

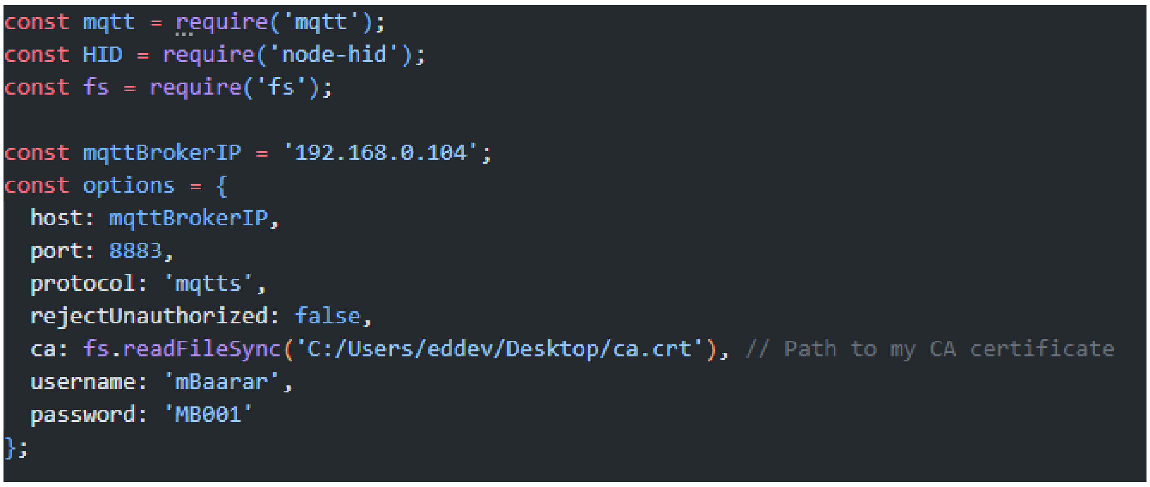


```
c:\Program Files\mosquitto>mosquitto -c "C:\Program Files\mosquitto\mosquitto.conf" -v
1722051796: mosquitto version 2.0.18 starting
1722051796: Config loaded from C:\Program Files\mosquitto\mosquitto.conf.
1722051796: Opening ipv6 listen socket on port 8883.
1722051796: Opening ipv4 listen socket on port 8883.
1722051796: mosquitto version 2.0.18 running
```

FIG. 2.2 : broker start

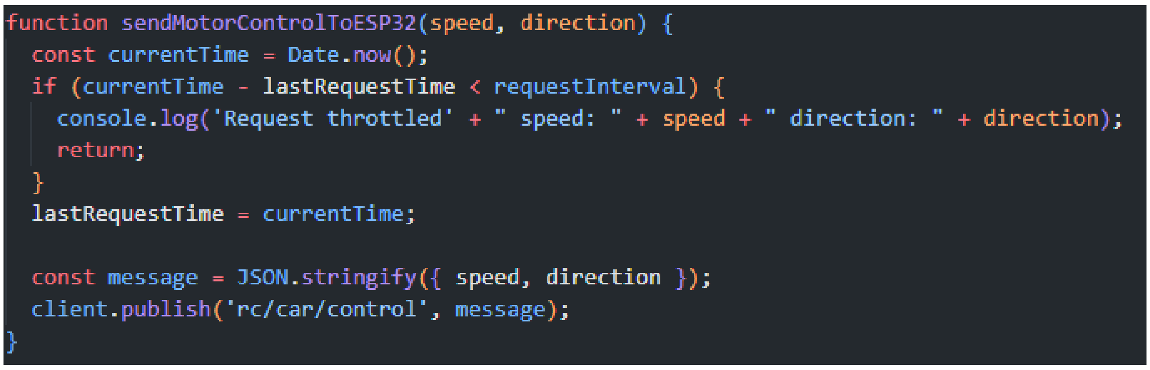
2 Local server

This nodejs Localserver is used to capture input from the Connected Controller and send control data to the car, publishing commands (speed, direction) to the topic ”rc/car/control” respecting a timed interval.



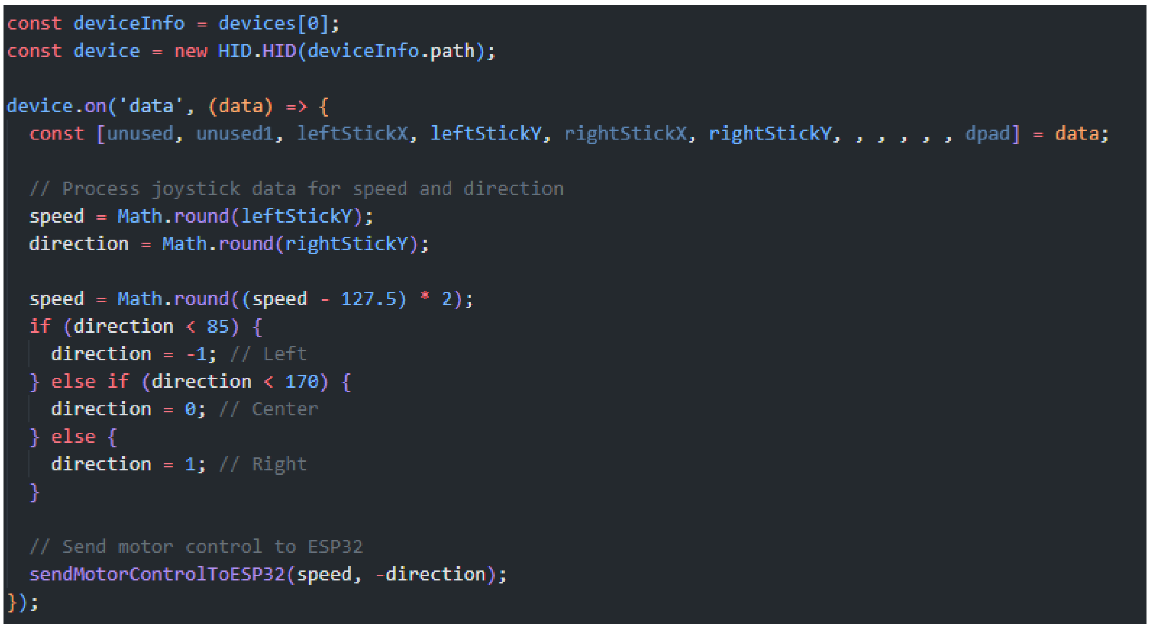
```
const mqtt = require('mqtt');
const HID = require('node-hid');
const fs = require('fs');

const mqttBrokerIP = '192.168.0.104';
const options = {
  host: mqttBrokerIP,
  port: 8883,
  protocol: 'mqtts',
  rejectUnauthorized: false,
  ca: fs.readFileSync('C:/Users/eddev/Desktop/ca.crt'), // Path to my CA certificate
  username: 'mBaarar',
  password: 'MB001'
};
```

FIG. 2.3 : Connecting to MOsquitto Broker WITH SSL/TLS



```
function sendMotorControlToESP32(speed, direction) {
  const currentTime = Date.now();
  if (currentTime - lastRequestTime < requestInterval) {
    console.log('Request throttled' + " speed: " + speed + " direction: " + direction);
    return;
  }
  lastRequestTime = currentTime;

  const message = JSON.stringify({ speed, direction });
  client.publish('rc/car/control', message);
}
```

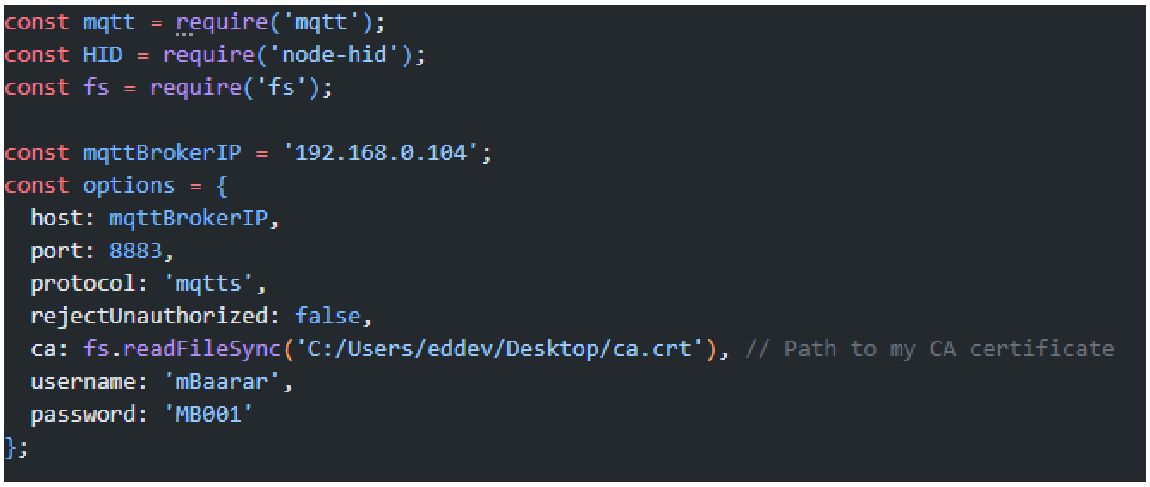
FIG. 2.4 : Sending control data



```
const deviceInfo = devices[0];
const device = new HID.HID(deviceInfo.path);

device.on('data', (data) => {
  const [unused, unused1, leftStickX, leftStickY, rightStickX, rightStickY, , , , , , dpad] = data;

  // Process joystick data for speed and direction
  speed = Math.round(leftStickY);
  direction = Math.round(rightStickY);

  speed = Math.round((speed - 127.5) * 2);
  if (direction < 85) {
    direction = -1; // Left
  } else if (direction < 170) {
    direction = 0; // Center
  } else {
    direction = 1; // Right
  }

  // Send motor control to ESP32
  sendMotorControlToESP32(speed, -direction);
});
```

Fig. 2.5 : Sending control data

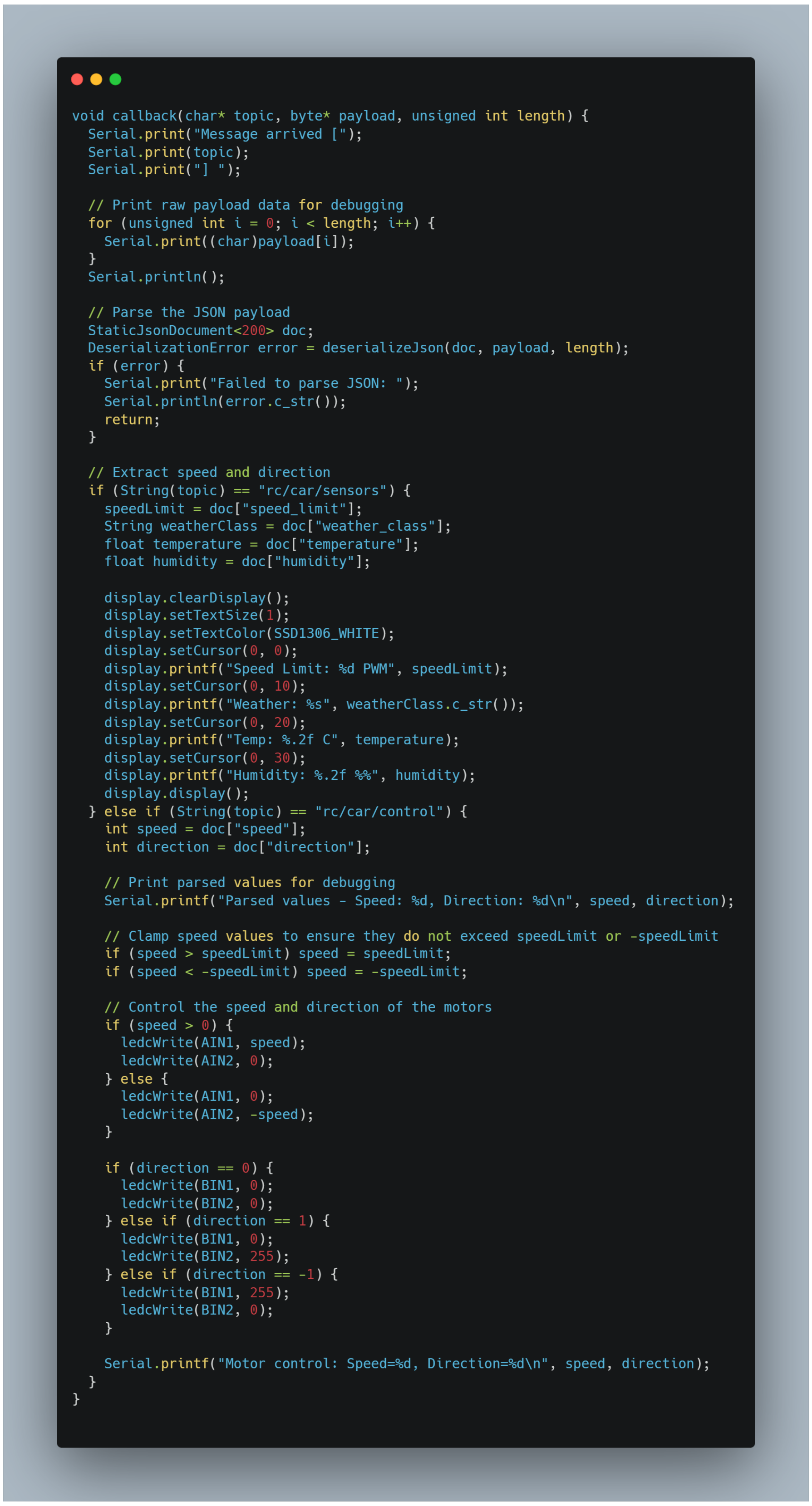
3 Rc car control block



```
const mqtt = require('mqtt');
const HID = require('node-hid');
const fs = require('fs');

const mqttBrokerIP = '192.168.0.104';
const options = {
  host: mqttBrokerIP,
  port: 8883,
  protocol: 'mqtts',
  rejectUnauthorized: false,
  ca: fs.readFileSync('C:/Users/eddev/Desktop/ca.crt'), // Path to my CA certificate
  username: 'mBaarar',
  password: 'MB001'
};
```

Fig. 2.6 : Connecting to broker using ca certificate and user/password

Callback function to get speed and direction and make sure it doesn't pass speed limit.

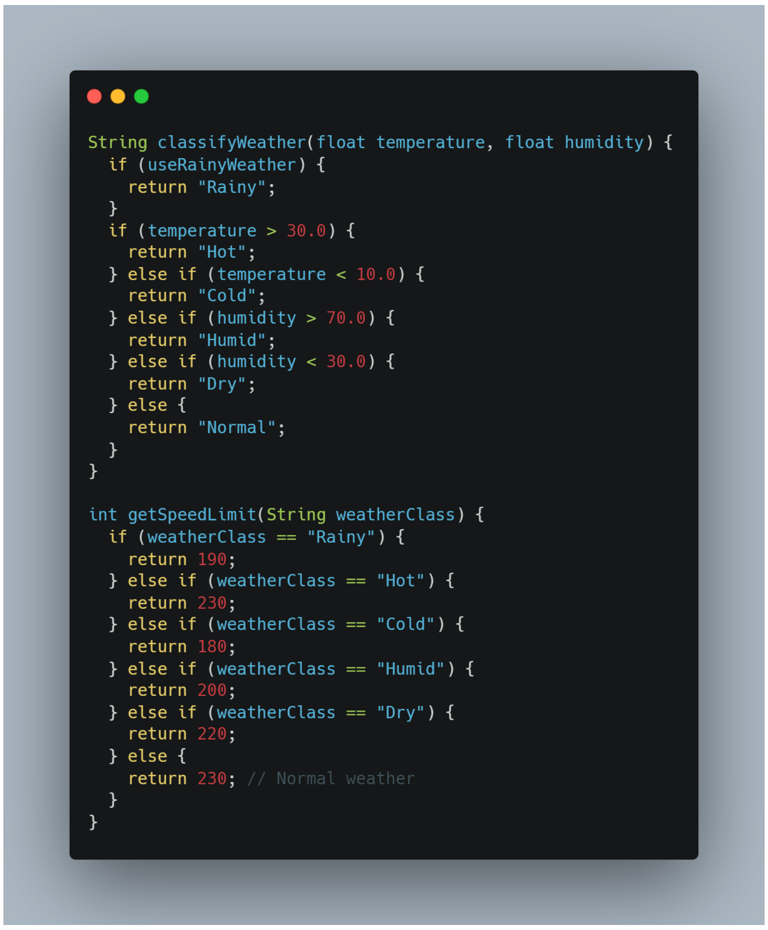


```
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");

  // Print raw payload data for debugging
  for (unsigned int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();

  // Parse the JSON payload
  StaticJsonDocument<200> doc;
  DeserializationError error = deserializeJson(doc, payload, length);
  if (error) {
    Serial.print("Failed to parse JSON: ");
    Serial.println(error.c_str());
    return;
  }

  // Extract speed and direction
  if (String(topic) == "rc/car/sensors") {
    speedLimit = doc["speed_limit"];
    String weatherClass = doc["weather_class"];
    float temperature = doc["temperature"];
    float humidity = doc["humidity"];

    display.clearDisplay();
    display.setTextSize(1);
    display.setTextColor(SSD1306_WHITE);
    display.setCursor(0, 0);
    display.printf("Speed Limit: %d PWM", speedLimit);
    display.setCursor(0, 10);
    display.printf("Weather: %s", weatherClass.c_str());
    display.setCursor(0, 20);
    display.printf("Temp: %.2f C", temperature);
    display.setCursor(0, 30);
    display.printf("Humidity: %.2f %%", humidity);
    display.display();
  } else if (String(topic) == "rc/car/control") {
    int speed = doc["speed"];
    int direction = doc["direction"];

    // Print parsed values for debugging
    Serial.printf("Parsed values - Speed: %d, Direction: %d\n", speed, direction);

    // Clamp speed values to ensure they do not exceed speedLimit or -speedLimit
    if (speed > speedLimit) speed = speedLimit;
    if (speed < -speedLimit) speed = -speedLimit;

    // Control the speed and direction of the motors
    if (speed > 0) {
      ledcWrite(AIN1, speed);
      ledcWrite(AIN2, 0);
    } else {
      ledcWrite(AIN1, 0);
      ledcWrite(AIN2, -speed);
    }

    if (direction == 0) {
      ledcWrite(BIN1, 0);
      ledcWrite(BIN2, 0);
    } else if (direction == 1) {
      ledcWrite(BIN1, 0);
      ledcWrite(BIN2, 255);
    } else if (direction == -1) {
      ledcWrite(BIN1, 255);
      ledcWrite(BIN2, 0);
    }

    Serial.printf("Motor control: Speed=%d, Direction=%d\n", speed, direction);
  }
}
```

Fig. 2.7 : Callback function rccar

4 Sensor Node

we do the same thing to connect to the broker, and in these blocks we define weather classes and based on that we define speedlimit and send it (pub) to the topic ”/rc/car/sensors”



```
String classifyWeather(float temperature, float humidity) {
  if (useRainyWeather) {
    return "Rainy";
  }
  if (temperature > 30.0) {
    return "Hot";
  } else if (temperature < 10.0) {
    return "Cold";
  } else if (humidity > 70.0) {
    return "Humid";
  } else if (humidity < 30.0) {
    return "Dry";
  } else {
    return "Normal";
  }
}

int getSpeedLimit(String weatherClass) {
  if (weatherClass == "Rainy") {
    return 190;
  } else if (weatherClass == "Hot") {
    return 230;
  } else if (weatherClass == "Cold") {
    return 180;
  } else if (weatherClass == "Humid") {
    return 200;
  } else if (weatherClass == "Dry") {
    return 220;
  } else {
    return 230; // Normal weather
  }
}
```

FIG. 2.8 : Logic block Sensor node

Chapitre 3

Demo

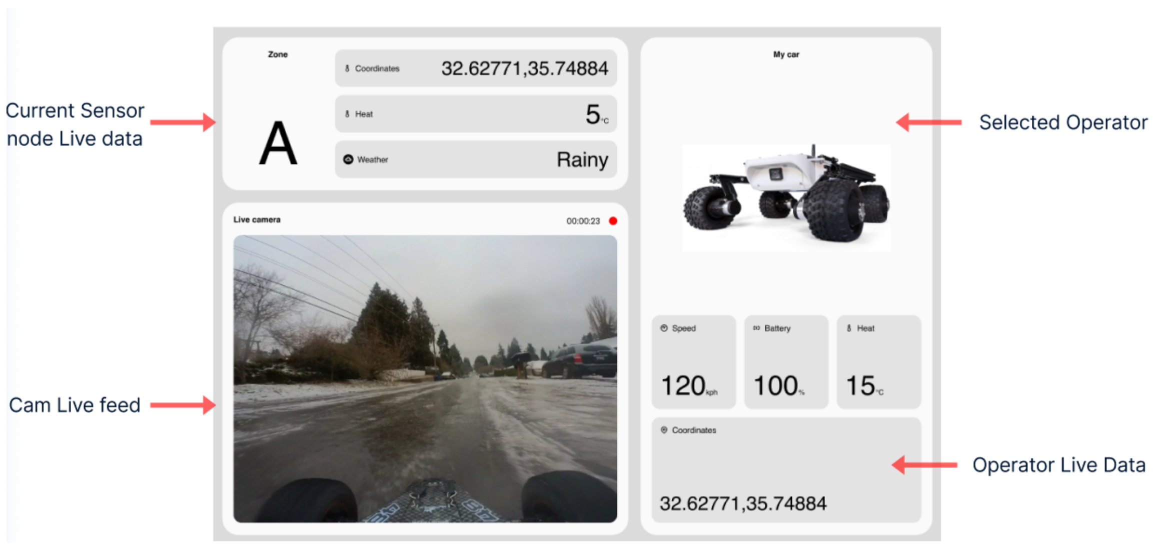


FIG. 3.1 : User Interface

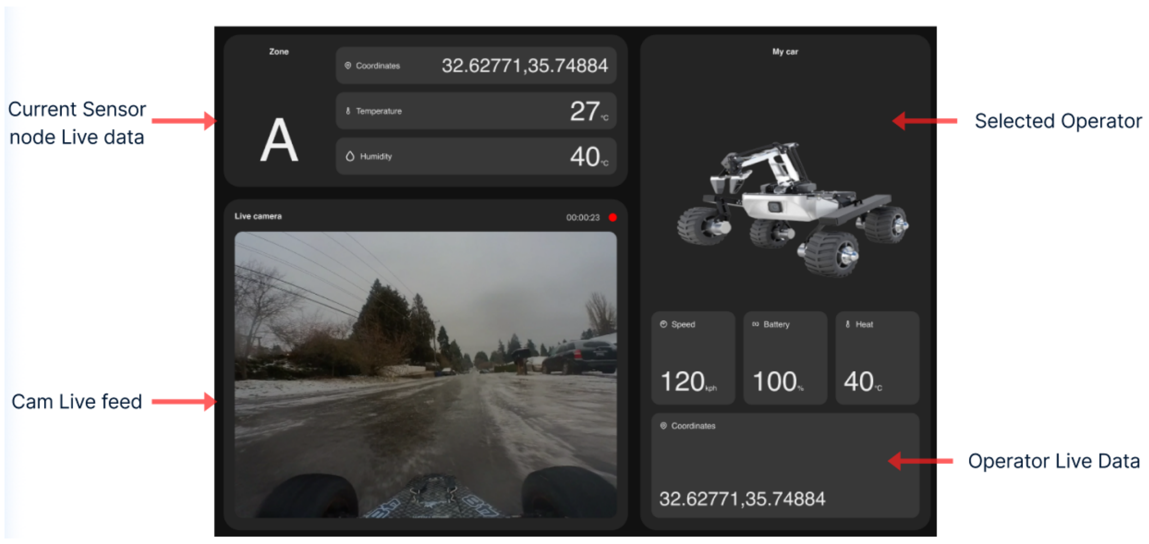


FIG. 3.2 : User Interface Dark

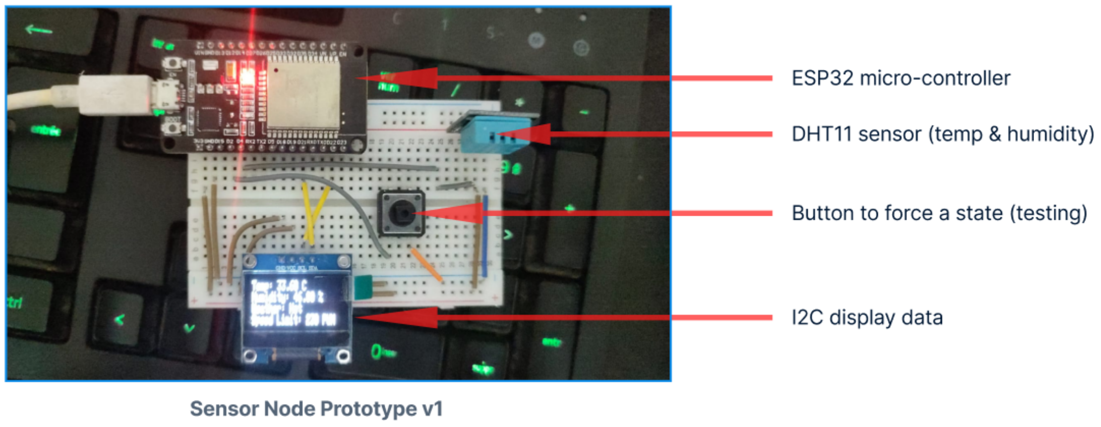

ESP32 micro-controller
DHT11 sensor (temp & humidity)
Button to force a state (testing)
I2C display data
Sensor Node Prototype v1

FIG. 3.3 : Sensor Node

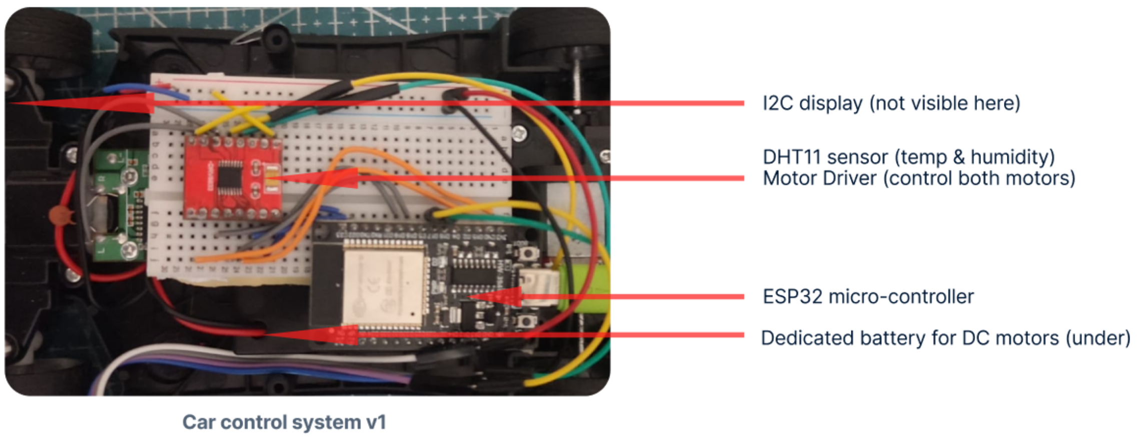

I2C display (not visible here)
DHT11 sensor (temp & humidity)
Motor Driver (control both motors)
ESP32 micro-controller
Dedicated battery for DC motors (under)
Car control system v1

FIG. 3.4 : Car Control system

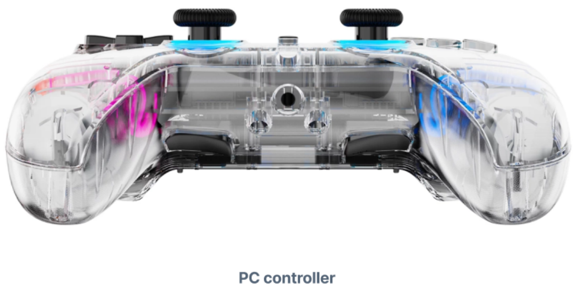


FIG. 3.5 : Controller

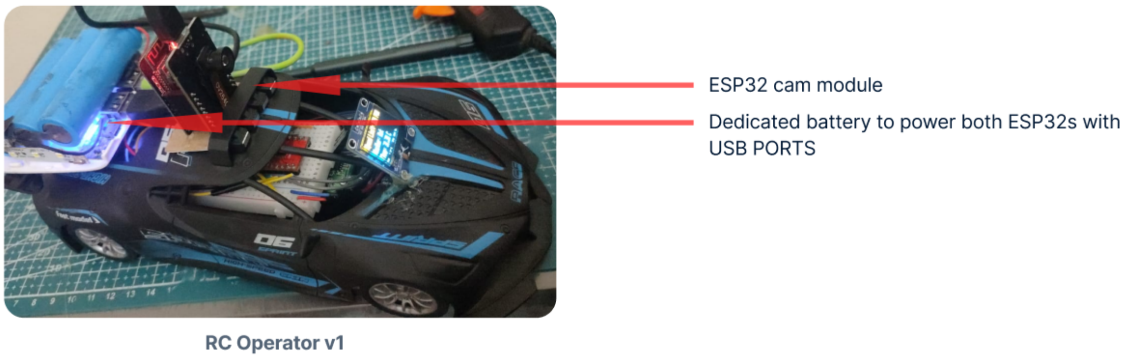


Fig. 3.6 : Operator v1

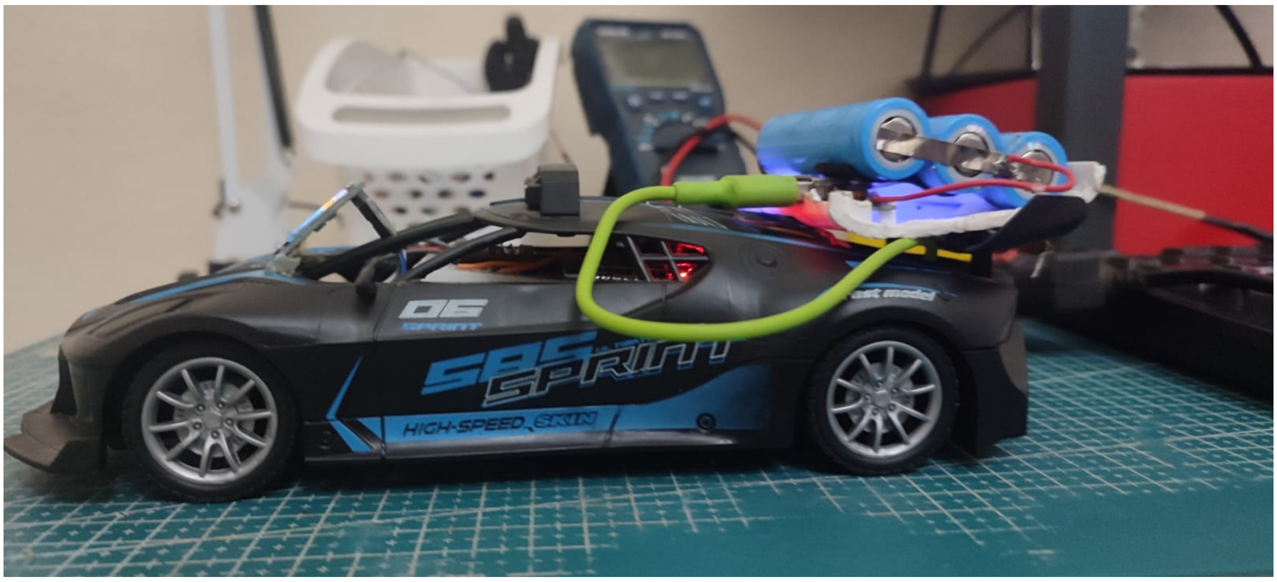


Fig. 3.7 : RC CAR Side image

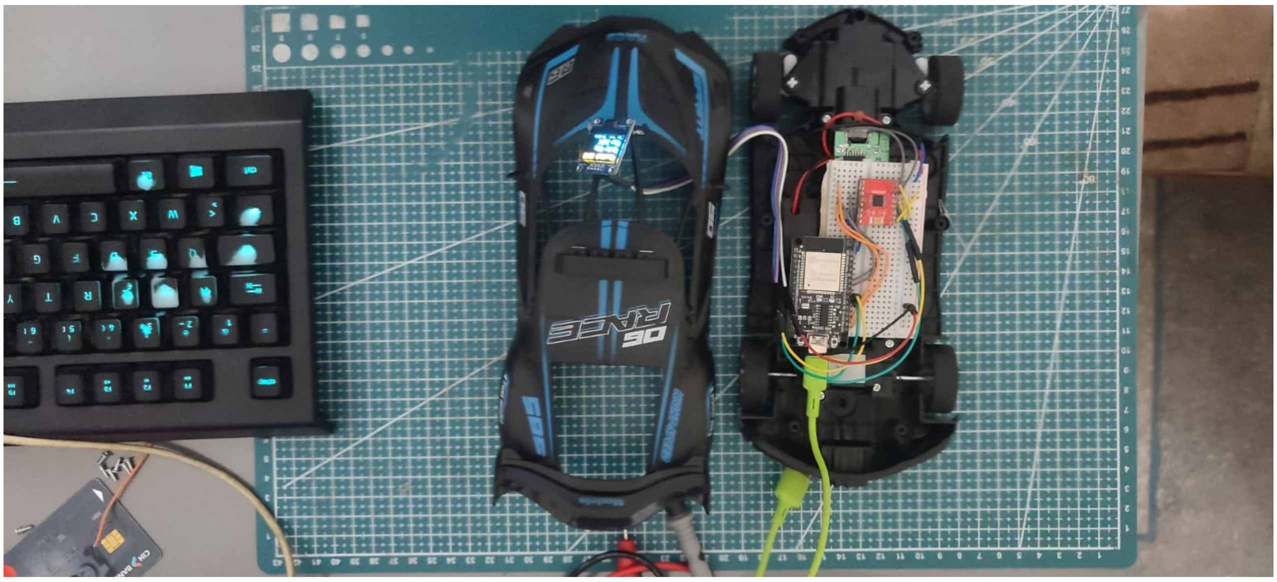


Fig. 3.8 : RC CAR TOP image

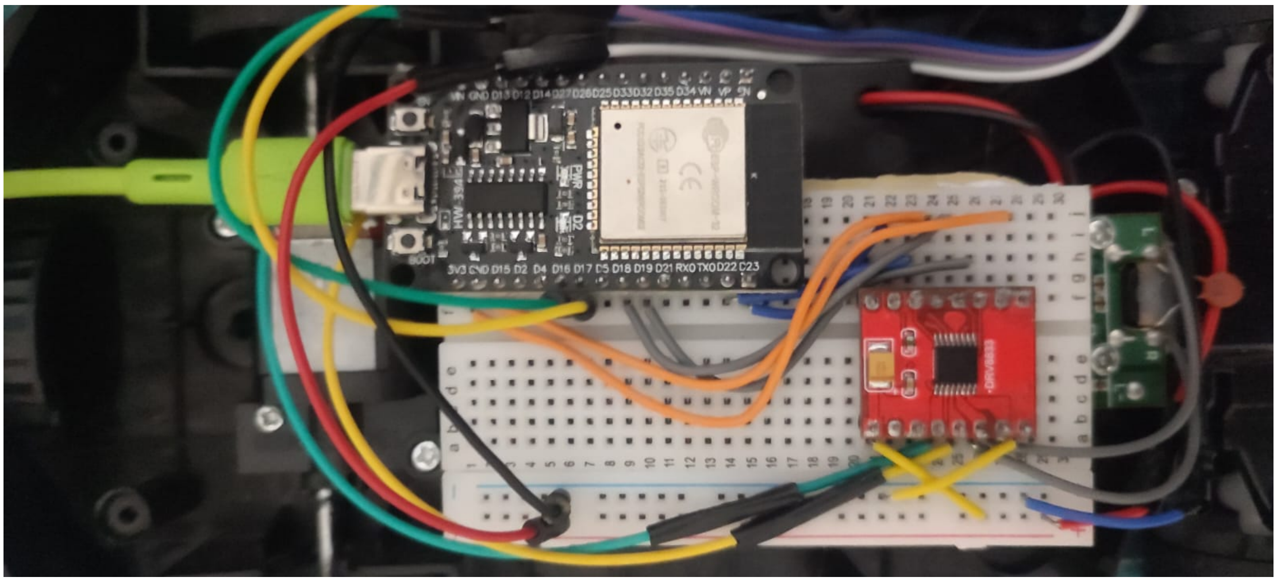


Fig. 3.9 : RC CAR circuit 1

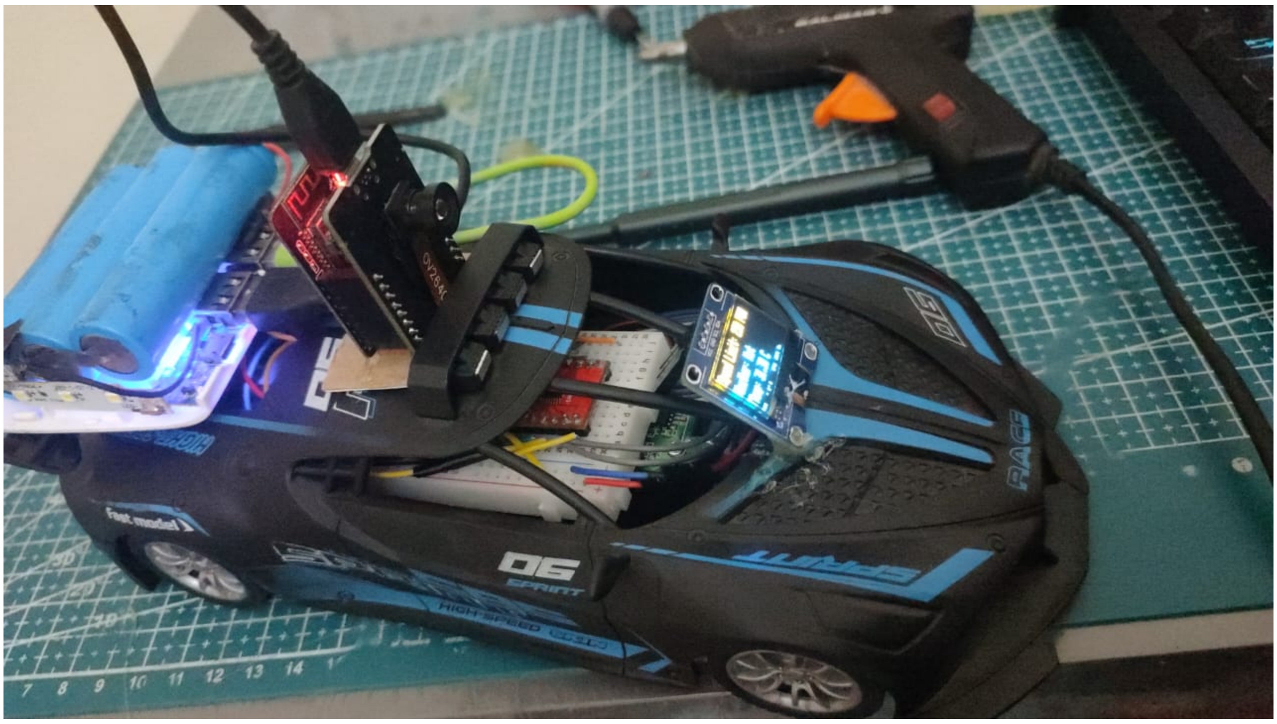
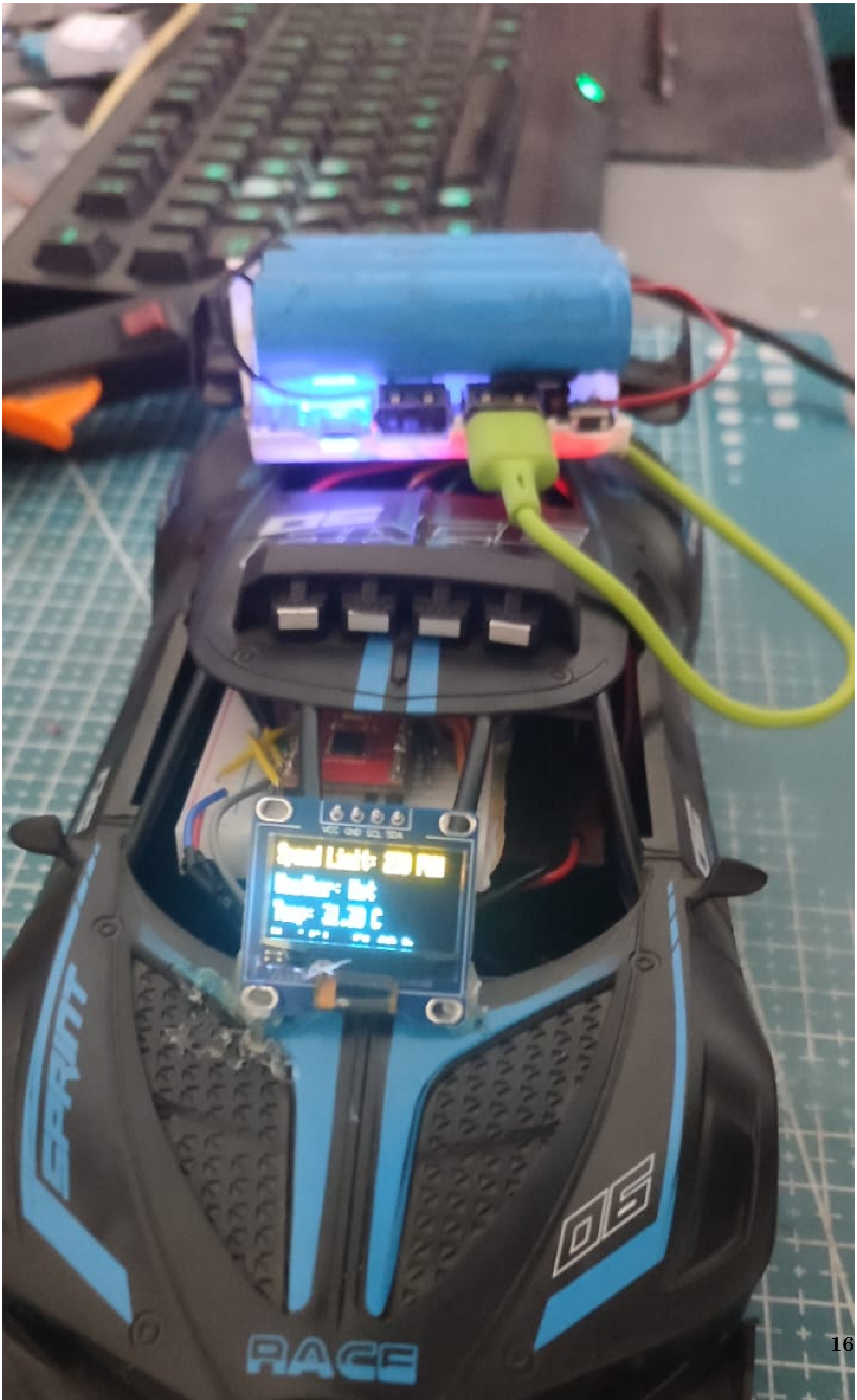


Fig. 3.10 : RC CAR + cam


VCC GND SCL SDA
Speed Limit:
Weather: Hot
Temp:
SPRINT
05
RACE

Chapitre 4

Conclusion

1 Conclusion

This IoT-based RC car control system demonstrates the integration of various technologies and protocols to achieve a sophisticated remote control solution. The project showcases real-time control, sensor-based environmental adaptation, and video streaming capabilities, providing a comprehensive and interactive user experience.

2 Future developement and plans

- **SOS feature** : add a functionality to each node in this IOT solution to send S.O.S (distress signal), and make the car capture it and Track its location dynamically to investigate further. (main objective)
- **Historical Data and Logs** : Register Historical Data and Logs of the different components.
- **Migrate to cloud** : migrating from local machine to cloud services and architecture.
- **User Interface** : Develop a Friendly User Interface to make it easier on the operators of the car.
- **Obstacle Detection** : Integrate additional sensors for obstacle detection and avoidance, also use localserver as a processing unit to process the camera feed and send commands accordinly such as object detection and auto-tracking of objects with the car.

- **Automated Navigation** : Implement algorithms for autonomous navigation.
- **Enhanced Security** : Implement advanced authentication and authorization mechanisms.
- **Extended Functionality** : Add more control features and improve user interface on the control server.