# IOT - SUM24
# Faculty of Computers and Data Science
# 02-24-03205 - Field Training
# Module 3 Assignment Report

Ahmed Mohamed Gaber
2203173

Hazem Ahmed Abdelfatah
2203175

Moataz Ali Ramadan
2203177

Momen Wagdy Hamed
2206134

Youssef Salah Mostafa
22010442

August 2024

# Table of Contents

## 1 Circuits

### 1.1 Circuit 1

In this question, we were tasked with connecting the ESP32 to a global server and fetch the time data through NTP (Network Time Protocol), the ESP handles this by using set libraries made for this task.

```
#include <WiFi.h>
#include <WiFiUdp.h>
#include <NTPClient.h>

const char* ssid     = "network_ssid";
const char* password = "network_password";

// Define NTP Client
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);
// Time offset in seconds UTC+1 means
3600 which means 1 hour
const long utcOffsetInSeconds = 7200 + 3600;
void setup() {
  Serial.begin(115200);
  // Connect to Wi-Fi network
  WiFi.begin(ssid, password);
  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    // Wait half a second before retrying
  }

  Serial.println("Connected to WiFi");

  // Initialize NTPClient with a time offset
  timeClient.begin();
  timeClient.setTimeOffset(utcOffsetInSeconds);
}

void loop() {
  timeClient.update();   // Update time
  Serial.println(timeClient.getFormattedTime());
  // Print the time
  delay(1000);
  // Wait for a second before printing again
}
```

Figure 1: Question 1 Code.

This code works by connecting to the WIFI and creating a UDP socket to pass through to the `NTPClient`, then we offset the time by an hour to account for day-light savings, at the end we update the time and print it to the serial monitor. NTP is crucial for distributed systems since it offers them a better way to synchronize their clocks across all devices without having to implement long and tedious algorithms.

## 1.2 Circuit 2

In this question we were asked to implement a web server that offers LED control over on the ESP, where the ESP hosts a web server that serves an `HTML` page that can control the LED light.

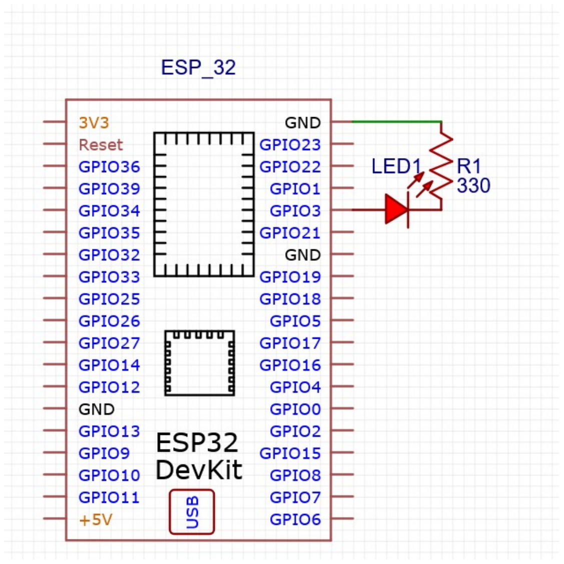

Figure 2: Question 2 Circuit.

The circuit is pretty straight forward, it consists of just an LED attached to the ESP32 that lights up depending on the code.

```
#include <WiFi.h>
#include <WebServer.h>

// WiFi credentials
const char* ssid = "network_ssid
const char* password = "network_password";

// LED pin
const int ledPin = 13;

// Create a web server on port 80
WebServer server(80);

// Function to handle the root URL
```

```
void handleRoot() {
  // HTML content for the root URL
  String html = "<html><body>";
  html += "<h1>ESP32 LED Control</h1>";
  html += "<p><a href=\"/LED/on\">
  <button>Turn LED On</button></a></p>";
  html += "<p><a href=\"/LED/off\">
  <button>Turn LED Off</button></a></p>";
  html += "</body></html>";
  // Send the HTML content to the client
  server.send(200, "text/html", html);
}

// Function to handle turning the LED on
void handleLedOn() {
  digitalWrite(ledPin, HIGH); // Turn the LED on
  // Send response to the client
  server.send(200, "text/html", "<html><body>
  <h1>LED is ON</h1><p><a href=\"/\">
  <button>Back</button></a></p></body></html>");
}

// Function to handle turning the LED off
void handleLedOff() {
  digitalWrite(ledPin, LOW); // Turn the LED off
  // Send response to the client
  server.send(200, "text/html", "<html><body>
  <h1>LED is OFF</h1><p><a href=\"/\">
  <button>Back</button></a></p></body></html>");
}

void setup() {
  Serial.begin(115200);
  // Start the Serial communication

  pinMode(ledPin, OUTPUT);
  // Set the LED pin as output
  digitalWrite(ledPin, LOW);
  // Initialize the LED as off

  // Connect to WiFi
  WiFi.begin(ssid, password);

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  // Display IP address on the Serial Monitor
  Serial.println("Connected to WiFi");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());

  // Setup the web server routes
  server.on("/", handleRoot);
  // Handle the root URL
  server.on("/LED/on", handleLedOn);
  // Handle turning the LED on
  server.on("/LED/off", handleLedOff);
  // Handle turning the LED off
  server.begin();
  // Start the web server
  Serial.println("HTTP server started");
}

void loop() {
  // Handle client requests
  server.handleClient();
}
```

Figure 3: Question 2 Code.

In this code we start by defining the network's SSID and password, then we create a web server, the `handleRoot` function sends a web-page containing to buttons to turn the LED ON or OFF, then we define 2 paths, one for when the ON button is pressed, and one for when the OFF button is pressed, then we attach them to the server and start the server's operation. In the submission there is another code made using python that interfaces with the ESP32's web server and turns the LED ON and OFF.

## 1.3 Circuit 3

In this question we were asked to create a program on ESP32 that scans nearby networks and display their data including SSID, Encryption type, RSSI, and channel number on the LCD. This question was tricky, first problem was saving the network data to be scrolled through, this can be achieved by storing the data in multiple arrays, each array containing a property of the network, where the index is shared across all arrays. Second problem was converting the encryption type found to strings to be visualized, this was done using a huge switch-case statement. The looping of the screen was tricky since many ways could be implemented, we settled on using an incrementing index that gets passed to a modulus function, similar to how circular queues operate.

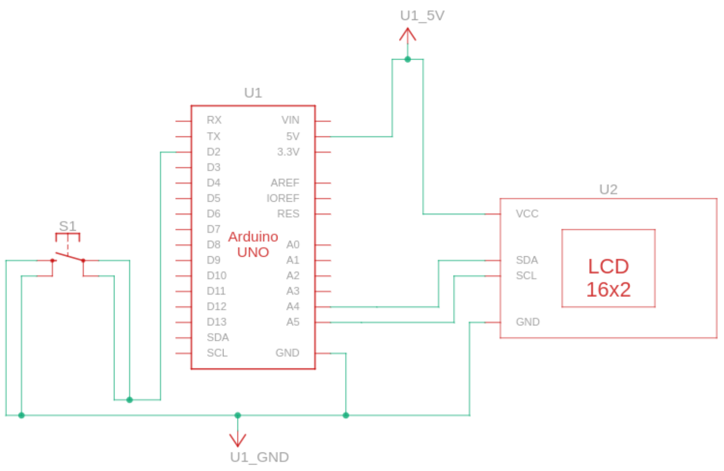

Figure 4: Question 3 Schematic.

The circuit isn't complicated, it consists of a mircrocontroller connected to an LCD for displaying networks and a push button for scrolling through networks.

```
#include "WiFi.h"
#include <LiquidCrystal_I2C.h>

const int buttonPin = 23;
// GPIO pin connected to the push button
int buttonState = 0;
// Variable to hold the button state
int previousButtonState = HIGH;
// Variable to hold the previous button state
int currentIndex = 0;
// Index to track the current network being
displayed
const int maxNetworks = 20;
// Maximum number of networks to store

// Arrays to store WiFi scan results
String ssid[maxNetworks];
long rssi[maxNetworks];
int channel[maxNetworks];
int encryption[maxNetworks];
int networkCount = 0;
// Number of networks found

LiquidCrystal_I2C lcd(0x27, 16, 2);
// Initialize the LCD with I2C address
0x27 and 16x2 size

void setup() {
  Serial.begin(115200);

  // Initialize the LCD
  lcd.init();
  lcd.backlight();

  // Initialize the button pin as an input
  with an internal pull-up resistor
  pinMode(buttonPin, INPUT_PULLUP);

  // Set WiFi to station mode and disconnect
  from an AP if it was previously connected.
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);

  Serial.println("Setup done");

  // Initial WiFi scan
  scanNetworks();
}

void loop() {
  // Read the state of the push button
  buttonState = digitalRead(buttonPin);

  // Check if the button is pressed
  (LOW state because of pull-up)
  if (buttonState == LOW &&
  previousButtonState == HIGH) {
    Serial.println("Button pressed");

    // Display the next network's information
    displayNetwork(currentIndex);

    // Move to the next network,
    wrap around if necessary
    currentIndex = (currentIndex + 1)
    % networkCount;

    // Debounce the button
    delay(200);
  }

  previousButtonState = buttonState;
}

void scanNetworks() {
  Serial.println("Scanning for networks...");
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Scanning for ");
    lcd.setCursor(0, 1);
    lcd.print("    networks...");

  // Perform the WiFi scan
  networkCount = WiFi.scanNetworks();
  if (networkCount == 0) {
    Serial.println("No networks found");
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("No networks found");
  } else {
    Serial.print(networkCount);
    Serial.println(" networks found");
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("networks found");
    lcd.setCursor(7, 1);
    lcd.print(networkCount);


    // Limit the number of networks to
    maxNetworks
    if (networkCount > maxNetworks) {
      networkCount = maxNetworks;
    }

    // Store the scan results
    for (int i = 0; i < networkCount; ++i) {
      ssid[i] = WiFi.SSID(i);
      rssi[i] = WiFi.RSSI(i);
      channel[i] = WiFi.channel(i);
      encryption[i] = WiFi.encryptionType(i);
    }
  }

  // Delete the scan result to free memory
  WiFi.scanDelete();
}

void displayNetwork(int index) {
  if (index < networkCount) {
    // Display on Serial Monitor
    Serial.print(index + 1);
     // Print network number
```

```
Serial.print("- ");
Serial.print(ssid[index]);
// Print SSID (WiFi name)
Serial.print(" (Ch: ");
Serial.print(channel[index]);
// Print Channel number
Serial.println(")");

Serial.print(" RSSI: ");
Serial.print(rssi[index]);
// Print RSSI value

Serial.print(" | ");
Serial.print("Encryption: ");
String encryptionType;
switch (encryption[index]) {
  case WIFI_AUTH_OPEN:
  encryptionType = "Open"; break;
  case WIFI_AUTH_WEP:
  encryptionType = "WEP"; break;
  case WIFI_AUTH_WPA_PSK:
  encryptionType = "WPA"; break;
  case WIFI_AUTH_WPA2_PSK:
  encryptionType = "WPA2"; break;
  case WIFI_AUTH_WPA_WPA2_PSK:
  encryptionType = "WPA+WPA2"; break;
  case WIFI_AUTH_WPA2_ENTERPRISE:
  encryptionType = "WPA2-EAP"; break;
  case WIFI_AUTH_WPA3_PSK:
  encryptionType = "WPA3"; break;
  case WIFI_AUTH_WPA2_WPA3_PSK:
  encryptionType = "WPA2+WPA3"; break;
  case WIFI_AUTH_WAPI_PSK:
  encryptionType = "WAPI"; break;
  default:
  encryptionType = "Unknown";
}
Serial.println(encryptionType);
Serial.println();
// Blank line for spacing between networks

// Display on LCD
lcd.clear();
lcd.setCursor(0, 0);
lcd.print(ssid[index].substring(0, 8));
 // Print the SSID (trimmed to fit)
lcd.setCursor(9, 0);
lcd.print("Ch:");
```

```
        lcd.print(channel[index]);
        // Print the channel number

        lcd.setCursor(0, 1);
        lcd.print("RSSI:");
        lcd.print(rssi[index]);
        // Print the RSSI value

        lcd.setCursor(9, 1);
        lcd.print(encryptionType.substring(0, 7));
        // Print the encryption type
        (trimmed to fit)
    }
}
```

Figure 5: Question 3 Code.

## 1.4 Circuit 4

In this question we were tasked with creating a system using an ESP32 that sends IR data to a Node-RED server that performs a calculation and returns the servo angle. After reading the distance from the IR sensor, we can communicate the results using HiveMQ using MQTT communication protocol and utilize Node-RED for processing and controlling the program flow.

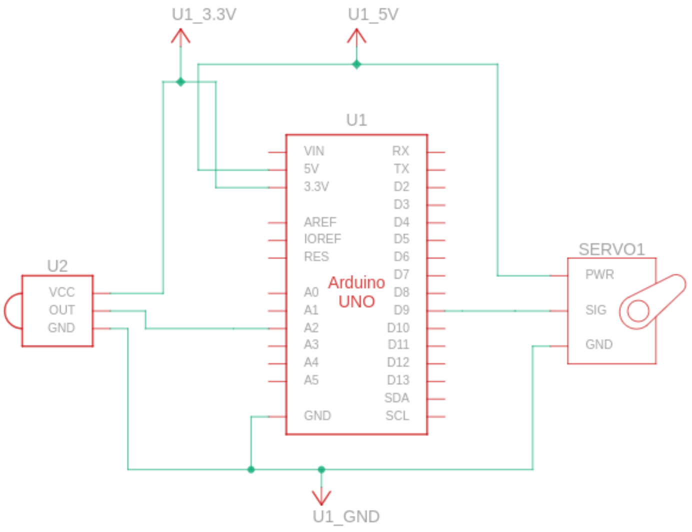

Figure 6: Question 4 Schematic.

```
// necessary libraries
```

```
#include <WiFiClientSecure.h>
// Library for secure Wi-Fi connection
#include <esp_crt_bundle.h>
// ESP32 certificate bundle
#include <WiFi.h>
// Library for Wi-Fi functions
#include <PubSubClient.h>
// library for MQTT client implementation
using HiveMQ
#include <ESP32Servo.h>
// library for servo control on ESP32

// Wi-Fi credentials
const char* ssid = "network_ssid";
// Wi-Fi network SSID
const char* password = "network_password";
// Wi-Fi network password

// MQTT broker configuration
const char* mqtt_broker = "hivemq_broker_url";
// MQTT broker
const char* mqtt_username = "mqtt_username";
// MQTT username
const char* mqtt_password = "mqtt_password";
// MQTT password
const char* mqttServer = "hivemq_alt_broker_url";
// MQTT alternative address
const int mqttPort = 8883;
// MQTT port

// MQTT topics
const char* IR_sensor_topic = "IR/distance";
// topic for publishing the IR sensor's
distance readings
const char* servo_topic = "servo/angle";
// topic to subscribe for determining the
servo's angle

// instance for Wi-Fi client, MQTT client and
servo motor
WiFiClientSecure espClient;
// secure Wi-Fi client instance
PubSubClient client(espClient);
// MQTT client instance using the Wi-Fi client
Servo servo;
// servo motor instance

// pin configuration
```

```
int servo_pin = 33;
// setting GPIO pin for servo at 33
int IR_sensor_pin = 32;
// setting GPIO pin for IR sensor at 32

// declaring the callback function
for handling MQTT messages
void callback(char* topic, byte* payload,
unsigned int length);

void setup() {

// declaring serial communication
Serial.begin(115200);
// setting serial communication at baud 115200
delay(10);

// connecting to Wi-Fi
WiFi.begin(ssid, password);
// begin Wi-Fi connection
while (WiFi.status() != WL_CONNECTED)
{  // if it is still not connected
  delay(500);
  Serial.print(".");
  // print "." to indicate its status as
  connecting
}
// setting Wi-Fi client into insecure
espClient.setInsecure();
// setting insecure for SSL
Serial.println("Connected.");
// confirming the success of the connection

// setting MQTT configuration
client.setServer(mqtt_broker, mqttPort);
// setting MQTT client server
client.setCallback(callback);
// setting MQTT client callback function

// connecting to MQTT broker
while (!client.connected()) {
  Serial.print("Connecting to MQTT Broker...");
  // printing that we will attempt to connect
  to MQTT broker
  String client_id = "esp32-client-" +
  String(WiFi.macAddress());
  // create client ID
  if (client.connect(client_id.c_str(),
  mqtt_username, mqtt_password)) {
  // attempt connection
    Serial.println("Connected to MQTT Broker");
    // connection complete
    client.subscribe("servo/angle");
    // subscribing to servo angle topic
  } else {
    Serial.print("Failed with state ");
    // connection failed
    Serial.print(client.state());
    // print failure state
    delay(2000);
  }
}

// setting pin configuration
pinMode(IR_sensor_pin, INPUT);
// setting IR sensor's pin as input
servo.attach(servo_pin);
// attaching servo to its pin
servo.write(0);
// setting servo's initial position
delay(1000);
}

void loop() {

// maintaining MQTT connection
client.loop();

// reading and publishing the IR sensor's
readings
int distance = analogRead(IR_sensor_pin);
// read analog value for distance from IR sensor
Serial.println(distance);
// printing read distance on serial monitor
client.publish(IR_sensor_topic,
String(distance).c_str());
// publishing distance to MQTT IR sensor topic
delay(1000);
}

// setting callback function for handling MQTT
messages
void callback(char* topic, byte* payload,
unsigned int length) {
  String messageTemp;
  // declaring temporary string
```

```
    for(int i=0; i<length; i+=1) {
    // looping through recieved payload
      messageTemp += (char)payload[i];
      // turning each character to string message
    }
    if (String(topic) == "servo/angle") {
    // if the message is for the servo angle topic
      int position = (payload[0] == '0') ? 0 : 180;
      // get the servo's position to either 0 or 180
      based on the recieved message
      servo.write(position);
      // apply the recieved position
      }
}
```

Figure 7: Question 4 Code.

In this code the ESP32 manages the publishing and subscription of topics and Node-RED does the processing in between and controls the program flow.

Broker:

- HiveMQ

Topics:

- IR/distance: Topic for publishing distance data from the IR sensor.
- servo/angle: Topic for controlling the angle of the servo motor through subscription.

In this setup, we begin by importing the essential libraries required for the operation of the ESP32, including Wi-Fi and MQTT communication, as well as the necessary sensor and actuator control. Following this, we define the Wi-Fi credentials and configure the MQTT broker details. Next, we specify the MQTT topics for data exchange and proceed with instance initialization, where we create instances for the Wi-Fi client, MQTT client, and the servo motor. We then configure the pins associated with the IR sensor and the servo.

In the `setup` function, we initiate serial communication and establish a connection to the WIFI network. After successfully connecting to WIFI, we configure the MQTT client to connect to the HiveMQ broker and subscribe to the relevant topics.

In the `loop` function, we ensure the MQTT connection remains active, continuously read data from the IR sensor, and publish the distance readings to the designated topic.

The `callback` function is responsible for handling incoming MQTT messages. It processes the messages to control the servo motor, adjusting its position based on the received data.

In Node-RED, the flow consists of three primary stages: acquisition, processing, and actuation. The flow begins with the MQTT Broker Node, which connects to the HiveMQ broker and handles the broker settings. The MQTT Input Node is used for acquisition, receiving data from the IR sensor through the publishing topic. The Function Node handles the processing, where it analyzes the distance readings, checks them against a predefined threshold of 2000, and determines the appropriate angle for the servo motor. Finally, the MQTT Output Node takes care of actuation by publishing the calculated servo angle, which is then used to control the motor based on the subscribed topic.

## 1.5 Circuit 5

In this question we were tasked with creating an MQTT system that reads data from multiple sensors. This system uses a microphone to take voice commands to operate.

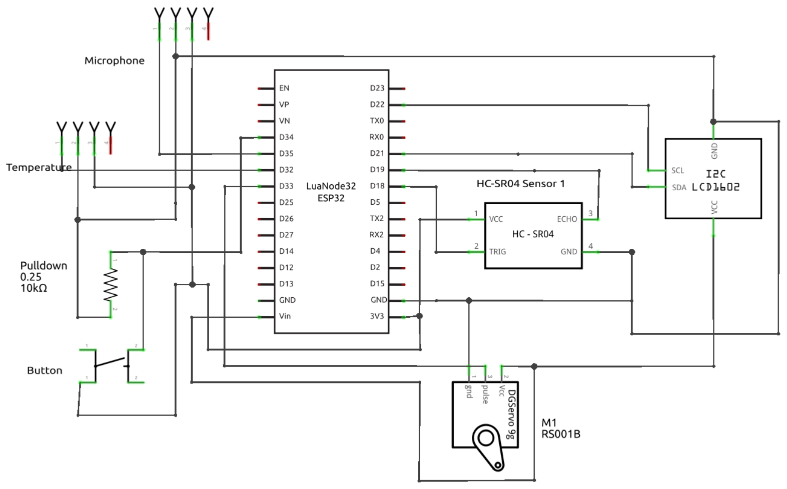

Figure 8: Question 5 schematic.

```
#include <NewPing.h>
// Library for ultrasonic sensor
#include <LiquidCrystal_I2C.h>
// Library for I2C LCD display
#include <ESP32Servo.h>
// Library for controlling servos on ESP32
#include <WiFiClientSecure.h>
// Library for secure Wi-Fi connection
#include <esp_crt_bundle.h>
```

```
// ESP32 certificate bundle
#include <WiFi.h>
// Library for Wi-Fi functions
#include <PubSubClient.h>
// Library for MQTT communication

// Pin Definitions
#define trig_pin 19
// Ultrasonic sensor trigger pin
#define echo_pin 18
// Ultrasonic sensor echo pin
#define button_pin 34
// Button input pin
#define mic_pin 35
// Microphone input pin

// Wi-Fi credentials
const char* ssid = "network_ssid";
// Wi-Fi SSID
const char* password = "network_pass";
// Wi-Fi password

// MQTT Broker details
const char* mqtt_broker = "hivemq_broker_url";
const char* mqtt_username = "mqtt_username";
// MQTT username
const char* mqtt_password = "mqtt_password";
// MQTT password
const int mqtt_port = 8883;
// MQTT port for secure connection
const char* topic_publish = "esp32/send";
// MQTT topic for publishing messages
const char* topic_distance = "esp32/distance";
// MQTT topic for distance sensor data
const char* running_topic = "esp32/distance";
// MQTT topic for distance sensor data
const char* topic_mic = "esp32/mic";
// MQTT topic for microphone data
const char* topic_temp = "esp32/temp";
// MQTT topic for temperature readings
const char* topic_subscribe = "esp32/receive";
// MQTT topic for receiving messages

// Variables for timing and control
long mil = 0;
// Current time in milliseconds
long lastmil = 0;
// Last recorded time in milliseconds
```

```
int pos = 0;
// Position of the servo motor
int i = 0;
// Index for the mic data being sent
bool moveto180 = true;
// Flag to control servo direction
bool mic_read = false;
// Flag to control microphone data reading
long lastmic = 0;
// Last time microphone data was read
long micstart = 0;
// Start time for microphone data reading

// Variables to store sensor readings
float currentDistance = 0;
// Current distance reading
int currentTemp = 0;
// Current temperature reading

// Initialize Wi-Fi and MQTT client objects
WiFiClientSecure espClient;
// Secure Wi-Fi client
PubSubClient client(espClient);
// MQTT client using secure Wi-Fi client
Servo serv;
// Servo motor control object
LiquidCrystal_I2C lcd(0x27,16,2);
// LCD display object (I2C address:
0x27, 16x2 characters)
NewPing sensor(trig_pin, echo_pin, 40);
// Ultrasonic sensor object (max distance: 40cm)
#define LED_PIN 2  // LED pin (GPIO 2)

// Callback function to handle messages received on
subscribed topics
void callback(char* topic, byte* payload,
unsigned int length) {

    // Convert payload to a string
    String message;
    for (int i = 0; i < length; i++) {
        message += (char)payload[i];
    }

    // Check the content of the message and display
    the appropriate data
    if (message == "0") {
        running_topic = topic_temp;
    } else if (message == "1") {
        running_topic = topic_distance;
    } else if (String(topic) == running_topic) {
        // Display the message on the LCD (if it
        doesn't match "0" or "1")
        lcd.clear();
        lcd.home();
        lcd.print("Message: ");
        lcd.setCursor(0, 1);
        lcd.print(message);
    }
}


void setup() {
    // Initialize Serial Monitor
    Serial.begin(115200);

    // Set pin modes
    pinMode(32, INPUT);
    // Set GPIO 35 as input (analog pin for
    temperature sensor)
    pinMode(LED_PIN, OUTPUT);
    // Set LED pin as output
    digitalWrite(LED_PIN, LOW);
    // Initially turn off the LED

    // Initialize Servo
    serv.attach(33);
    // Attach servo to GPIO 33
    serv.write(0);
    // Set initial servo position to 0 degrees

    // Initialize LCD
    lcd.init();
    // Initialize LCD
    lcd.backlight();
    // Turn on LCD backlight

    // Connect to Wi-Fi
    Serial.print("Connecting to Wi-Fi");
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConnected to Wi-Fi");
    // Set up secure Wi-Fi client
    (insecure for this example)
    espClient.setInsecure();

    // Connect to MQTT Broker
    client.setServer(mqtt_broker, mqtt_port);
    client.setCallback(callback);
    // Set the MQTT callback function

    while (!client.connected()) {
        Serial.print("Connecting to MQTT Broker
        ...");
        String client_id = "esp32-client-" +
        String(WiFi.macAddress());
        if (client.connect(client_id.c_str(),
        mqtt_username, mqtt_password)) {
            Serial.println("Connected to
           MQTT Broker");
            client.subscribe(topic_subscribe,1);
            // Subscribe with QoS level 1
            client.subscribe(topic_temp,1);
            // Subscribe with QoS level 1
            client.subscribe(topic_distance,1);
            // Subscribe with QoS level 1
        } else {
            Serial.print("Failed with state ");
            Serial.print(client.state());
            delay(2000);
        }
    }

    // Publish an initial message to the MQTT
    topic with QoS level 1
    client.publish(topic_publish, "Hello
    from ESP32",true);
    // Set QoS level 1 and retain flag

    // Record the current time
    mil = millis();
    lastmil = mil;
}

void loop() {
    // Keep the MQTT client connected and process
    incoming messages
    client.loop();

    // Update the current time
```

```
mil = millis();
Serial.println(analogRead(34));

// Check if button is pressed and start reading
microphone data
if (!mic_read && analogRead(34) > 2000) {
    client.publish(topic_mic,
    String(analogRead(35)).c_str());
    // Set QoS level 1
    lastmic = mil;
    mic_read = true;
    // Start microphone data reading
    micstart = mil;
    // Record start time of microphone
    data reading
}

// Continue reading microphone data every
10ms for 4 seconds
if (mic_read && mil - lastmic > 10) {
    client.publish(topic_mic,
    String(analogRead(35)).c_str());
    // Set QoS level 1
    lastmic = mil;
    if (mil - micstart > 4000) {
        client.publish(topic_mic, "Done");
        // Set QoS level 1
        mic_read = false;
        // Stop microphone data reading
        after 4 seconds
    }
}

// Control servo movement and publish sensor
data every 10
0ms
if (mil - lastmil > 100) {
    // Move servo motor to the next position
    if (moveto180) {
        pos += 10;
        if (pos == 180) moveto180 = false;
        // Change direction at 180 degrees
    } else {
        pos -= 10;
        if (pos == 0) moveto180 = true;
        // Change direction at 0 degrees
    }
```

```
                // Read temperature from analog pin
                (connected to a temperature sensor)
                int temp = analogRead(32);
                currentTemp = map(4095 - temp, 0, 4095, -55,
                125);
                // Map the analog reading to temperature range

                // Measure distance using ultrasonic sensor
                unsigned int uS = sensor.ping();
                currentDistance = uS / US_ROUNDTRIP_CM;
                // Convert time to distance in cm

                // Publish temperature and distance data
                to MQTT topics with QoS level 1
                client.publish(topic_temp,
                String(currentTemp).c_str());
                // Set QoS level 1
                client.publish(topic_distance,
                String(currentDistance).c_str());
                // Set QoS level 1

                // Update servo position
                serv.write(pos);

                // Record the last time the loop was
                executed
                lastmil = mil;
            }
        }
```

Figure 9: Question 5 Code.

In this code we first import necessary libraries for the operation of each sensor and actuator, then we start defining the pins for them, after that we define the network data and the MQTT broker details, we then create a WIFI client, Servo, Ultrasonic sensor, LCD, and Public subscription client.

After that we define the callback function, which works depending on the given topic, in the classifier side we have 2 set classes, 0 for Temperature and 1 for Distance. We start by setting up the devices, attaching all the needed sensors and connecting to WIFI and MQTT server and subscribing to the selected topics.

During operation we check for the button pin, if it is active we start sending microphone data for 4 seconds in intervals of 10ms, at the end we send `"Done"` as a termination sequence. The timer for the servo, temperature, and ultrasonic are all 100ms, where we move the servo with the attached ultrasonic head, then we send the data to the topics.

On the server side, we have a running server listening for the mic data, when the termination sequence is found the data is then padded or cut to 260 entries and converted to a numpy array and standardized. After which the data undergoes conversion from time domain to frequency domain using the FFT (Fast Fourier Transform) algorithm which generated complex numbers, we get the magnitude of the complex vectors which is a real value, the real values are symmetric since the audio data was real. We only use half the data to train the network, taking advantage of the symmetry, which is a user defined, custom trained, network.

The network used is a simple ANN initialized using the He initialization to prevent the dead neurons that are often created by linear acting functions such as *ReLU* and *tanh*.

$$W \sim \mathcal{N}\left(0, \frac{2}{n_l}\right)$$

where:

- $W$ represents the weights of the layer.
- $\mathcal{N}\left(0, \frac{2}{n_l}\right)$ is a normal distribution with a mean of 0 and a variance of $\frac{2}{n_l}$.
- $n_l$ is the number of input units in the layer.

The model is a 4 layer network, it consists of 130 neurons in the first layer, then 230 neurons in the second layer, then 70 neurons in the third, and finally 2 in the last. Leading to a parameter count of approximately 70k.

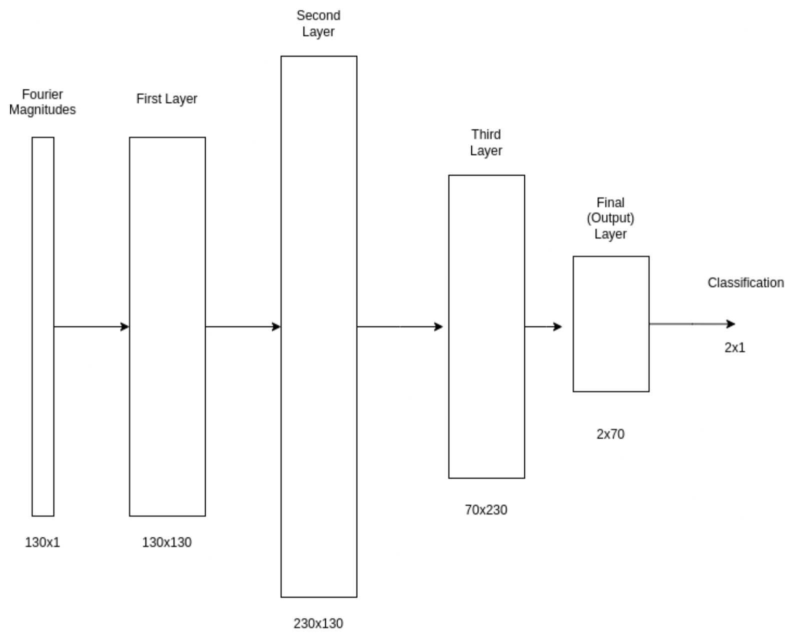

Figure 10: Speech Network Diagram.