OBJECTIVE

The main goal of this experiment is to show how to set up an IoT system using ESP32 microcontrollers, the Arduino IoT Cloud, and different sensors and actuators. The activities will cover:

1. Describing how the ESP32 microcontroller communicates with the Arduino IoT Cloud for sending data and receiving control commands.
2. Demonstrating how to build a practical IoT system for monitoring a farm, particularly focusing on soil moisture sensors and controlling a water pump.
3. Providing a detailed block diagram that outlines the system's architecture and components.
4. Creating a program that sends and receives data via Wi-Fi to and from the cloud, along with integrating LED indicators for visual feedback.
5. Setting up a local ESP32 server with button switches and LED widgets for easy remote control and monitoring.
6. Implementing a gas sensor monitoring system in a kitchen that sends data to the cloud and stores it in Google Sheets.
7. Writing programs to pull data from the cloud and trigger an alarm if gas levels go above a safe limit (50 ppm).

By addressing these points, the report aims to give a clear and practical guide on building IoT systems with ESP32 microcontrollers and Arduino IoT Cloud, covering everything from collecting sensor data to controlling devices and monitoring remotely.

COMPONENTS

The necessary hardware and software components to solve all the problems of this experiment are provided below. Explanations of these components will be given accordingly while solving a specific problem.

Hardware list:

1. ESP32 Microcontroller
2. Soil Moisture Sensor
3. Relay Module
4. LCD monitor
5. Power Supply
6. Jumper Wires
7. Green and Red LEDs
8. Resistors
9. Wi-Fi Router
10. Breadboard

11. USB Cable
12. Push Button
13. Gas Sensor MQ-5
14. Buzzer

Software list:

1. Arduino IDE
2. WiFi Library
3. Arduino IoT Cloud Library
4. Arduino Connection Handler Library
5. Wire Library
6. LiquidCrystal_I2C Library
7. ESPAsyncWebServer Library
8. AsyncTCP Library
9. ESP32 Board Support Package (BSP) for Arduino
10. Google Sheets
11. Google Sheets API
12. Google Apps Script

THEORY

The primary hardware component for this entire experiment is the ESP32. The ESP32 is a versatile and powerful microcontroller, perfect for embedded projects that need Wi-Fi and Bluetooth connectivity [1]. Created by Espressif Systems, it builds on the success of the earlier ESP8266, offering more processing power, better connectivity, and greater flexibility. Here’s an overview:

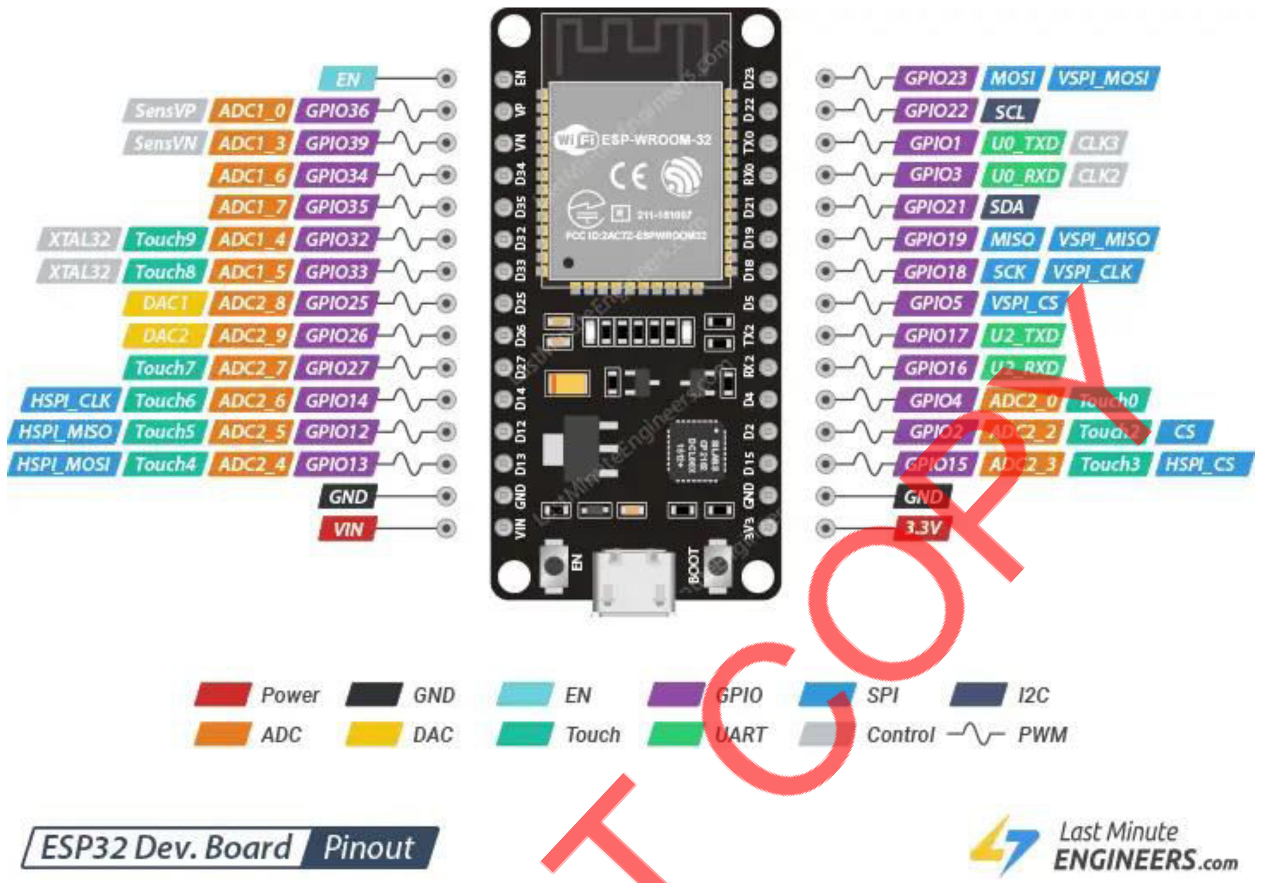

Fig 1: ESP32 Pinout [1]

a) Architecture:

The ESP32 is powered by a dual-core Xtensa LX6 microprocessor, which can run at speeds up to 240 MHz. It's loaded with a variety of peripherals like digital and analog interfaces, GPIO pins, SPI, I2C, UART, ADC, DAC, and more, allowing it to easily connect with sensors, actuators, and other devices.

b) Wireless Connectivity:

Wi-Fi: The ESP32 supports 802.11 b/g/n Wi-Fi, making it easy to connect to local networks for internet access and communication.

Bluetooth: It also includes Bluetooth Low Energy (BLE), enabling it to communicate with smartphones, tablets, and other Bluetooth devices.

c) Memory:

The ESP32 usually comes with built-in flash memory for storing programs and SRAM for data storage and execution. It has plenty of memory capacity to store firmware, data, and support dynamic program execution.

d) Peripheral Interfaces:

GPIO: The ESP32 offers many General Purpose Input/Output (GPIO) pins that can be used for various functions like digital input/output, handling interrupts, PWM, and more.

Serial Interfaces: It supports several serial communication protocols like SPI, I2C, and UART, allowing easy integration with a wide range of external devices.

Analog Inputs: The ESP32 includes ADC pins for reading analog sensor inputs and DAC pins for generating analog signals.

e) Development Environment:

SDKs: Espressif provides comprehensive SDKs and tools for programming the ESP32 in languages like C/C++ and Python.

Arduino IDE: The ESP32 is well-supported by the Arduino community, making it easy to create and deploy projects using the familiar Arduino IDE.

f) Power Management:

The ESP32 has advanced power management features, including various sleep modes and low-power operation, making it ideal for battery-powered projects that need long battery life.

g) Security Features:

It comes with strong security measures like secure boot, flash encryption, and cryptographic accelerators to ensure data integrity and device authentication in connected applications.

h) Applications:

IoT Devices: The ESP32 is widely used in Internet-of-Things (IoT) projects, powering devices like smart home gadgets, environmental sensors, and wearable tech.

Industrial Automation: It's also used in industrial automation, offering connectivity and control for monitoring and managing equipment remotely.

Consumer Electronics: The ESP32 finds its way into various consumer electronics, including smartwatches, fitness trackers, and home automation systems.

For programming purposes, the Arduino Integrated Development Environment (IDE) will be used as primary software. It is essential software for programming Arduino boards, which are popular in many electronics projects. The Arduino IDE makes it easy to write, compile, and upload code to an Arduino board. It's designed to be simple enough for beginners to use, yet powerful enough for more experienced users.

Two Key functions of Arduino IDE are explained below:

```
void setup() {
   // runs once when the program starts
}

void loop() {
   // repeats over and over
}
```

These two functions form the backbone of any Arduino program. The *setup()* function runs just once at the beginning, when the program starts. It's used for tasks that need to be done upfront, like setting pin modes or starting serial communication.

Once *setup()* has done its job, the *loop()* function takes over. This function runs continuously, repeating if the Arduino is on or until the program stops. The *loop()* function is where the main action happens, such as reading inputs from sensors, controlling outputs like LEDs or motors, and handling calculations.

PROBLEM 1

Explain briefly how ESP32 is communicating via Arduino IoT Cloud. Suppose you are observing the moisture in an agricultural farm using a sensor. You have to send the sensor data to a cloud. Based on the sensor data, the cloud would send a signal to the microcontroller used in the farm to turn on the motor pump. Explain how you can build such a system using the concept we have seen in this experiment. Use a proper block diagram to draw your system.

Hardware Components

1. ESP32 Microcontroller:

The ESP32 is a versatile microcontroller with built-in Wi-Fi and Bluetooth capabilities. It serves as the central controller for this project. Its role includes connecting to the Wi-Fi network, interfacing with the Arduino IoT Cloud, reading data from the soil moisture sensor, and controlling the motor pump through a relay. The ESP32 processes the incoming and outgoing data, ensuring the system operates as intended.

2. Soil Moisture Sensor:

The soil moisture sensor measures the amount of moisture present in the soil. It typically consists of two probes that are inserted into the soil. The sensor provides an analog output that correlates with the moisture level. This analog signal is read by the ESP32 and used to determine the soil's moisture content. Higher moisture levels result in a higher analog reading, and vice versa [2].

3. Relay Module:

The relay module acts as a switch to control the motor pump. It can handle higher currents and voltages than the ESP32's GPIO pins can manage directly. The relay module has a control input that receives signals from the ESP32. When the relay is activated (i.e., the control input is set to HIGH), it closes the circuit and allows current to flow through the motor pump, turning it on. When deactivated (i.e., the control input is set to LOW), the circuit is open, and the motor pump remains off [1].

4. Liquid Crystal Display (LCD):

The LCD used is of the LiquidCrystal_I2C type, which communicates with the ESP32 via the I2C protocol. This type of LCD is equipped with an I2C interface to simplify wiring and reduce the number of pins required for communication. The LCD displays information such as the soil moisture level and the status of the motor pump. The display is managed through a specific library that controls the text and cursor positions on the screen [3].

5. Power Supply:

The power supply provides the necessary electrical power to all components in the system. For the ESP32 and the relay module, a stable 5V power source is typically used. The soil moisture sensor may also require a 5V supply, depending on its specifications. Ensuring that the power supply delivers the correct voltage and current is crucial for the reliable operation of the entire system.

6. Connecting Wires:

Connecting wires link all the components to the ESP32. These wires are used to transmit signals between the ESP32, the soil moisture sensor, the relay module, and the LCD. The wires must be securely connected to ensure reliable communication and control within the system.

Integrating the Components:

Connecting the Soil Moisture Sensor:

The sensor's analog output pin is connected to a GPIO pin on the ESP32. This pin reads the analog voltage provided by the sensor, which correlates with the soil's moisture level.

Connecting the Relay Module:

The control input of the relay module is connected to another GPIO pin on the ESP32. This setup allows the ESP32 to turn the relay on or off, thereby controlling the motor pump.

Connecting the LCD:

The LCD is connected to the ESP32 using the I2C interface. The SDA (data line) and SCL (clock line) pins of the LCD are connected to the corresponding I2C pins on the ESP32. This configuration allows the ESP32 to send data to the LCD for display.

Powering the Components:

Each component is connected to the appropriate power source. The ESP32 and relay module typically use a 5V power supply, while the soil moisture sensor may also use 5V. The LCD, using I2C, is also powered by the same source.

Software Components:

1. Arduino IDE:

The Arduino Integrated Development Environment (IDE) is the primary software used for writing, compiling, and uploading code to the ESP32 microcontroller. It provides an editor for writing the code, a compiler for checking and building the code, and a tool to upload the code to the ESP32. The Arduino IDE also includes serial monitoring capabilities to view debug information and data from the ESP32.

2. WiFi Library:

The *WiFi.h* library is used to manage Wi-Fi connectivity on the ESP32. It provides functions to connect to Wi-Fi networks, check connection status, and handle network-related operations. This library is crucial for enabling the ESP32 to communicate with the Arduino IoT Cloud over the internet.

3. Arduino IoT Cloud Library:

The *ArduinoIoTCloud.h* library is used to connect the ESP32 to the Arduino IoT Cloud. It facilitates the synchronization of device data with the cloud, allowing for remote monitoring and control. This library manages cloud properties and ensures that data from the ESP32, such as soil moisture readings and motor control signals, are properly sent to and received from the cloud [4].

4. Arduino Connection Handler Library:

The *Arduino_ConnectionHandler.h* library is responsible for managing the connection between the ESP32 and the Arduino IoT Cloud. It handles the establishment, maintenance, and termination of the cloud connection, ensuring that the ESP32 remains connected to the cloud and can communicate data effectively.

5. Wire Library:

The *Wire.h* library is used for I2C communication. It allows the ESP32 to communicate with the LCD using the I2C protocol. This library simplifies the process of sending and receiving data over the I2C bus, which is necessary for managing the LCD display.

6. LiquidCrystal_I2C Library:

The *LiquidCrystal_I2C.h* library is used to interface with the LCD over I2C. It provides functions for initializing the LCD, controlling its display, and managing its backlight. This library makes it easier to interact with the LCD, enabling the ESP32 to display soil moisture levels and motor pump status.

7. Arduino IoT Cloud Code:

The code written for the Arduino IoT Cloud defines how the ESP32 interacts with the cloud. It includes the following functions:

Setup Function: It initializes hardware components, connects to Wi-Fi, starts the cloud connection, and sets up properties for cloud communication.

Loop Function: It continuously updates the cloud with sensor readings, displays data on the LCD, and controls the motor pump based on cloud data.

Callback Functions: Defines actions to be taken when cloud properties change, such as turning the motor pump on or off based on the *motorControl* variable.

Integration of Software Components

Connecting to Wi-Fi:

The *WiFi.h* library's functions are used to connect the ESP32 to the Wi-Fi network using the SSID and password. This connection enables the ESP32 to communicate with the Arduino IoT Cloud.

Communicating with the Arduino IoT Cloud:

The *ArduinoIoTCloud.h* and *Arduino_ConnectionHandler.h* libraries work together to handle communication between the ESP32 and the Arduino IoT Cloud. These libraries manage data synchronization and ensure that changes in cloud properties are reflected on the ESP32 and vice versa.

Handling I2C Communication with the LCD:

The *Wire.h* and *LiquidCrystal_I2C.h* libraries are used for I2C communication with the LCD. The *Wire.h* library manages the I2C bus, while the *LiquidCrystal_I2C.h* library provides functions to control the LCD display.

Updating and Displaying Data:

The code written in the Arduino IDE uses the included libraries to perform tasks such as reading sensor data, updating cloud properties, and displaying information on the LCD. The *setup()* function initializes the system, while the *loop()* function performs periodic tasks such as reading sensor data and updating the display.

Procedure:

1. Setting up the Development Environment:

The first step involves installing the Arduino IDE on the computer. This integrated development environment allows for writing, compiling, and uploading code to the ESP32. Once installed, it is necessary to ensure that the required libraries are added. Specifically, the *WiFi*, *ArduinoIoTCloud*, *Arduino_ConnectionHandler*, *Wire*, and *LiquidCrystal_I2C* libraries need to be included. These libraries can be installed through the Library Manager in the Arduino IDE.

2. Hardware Assembling:

The ESP32 microcontroller, soil moisture sensor, relay module, and LCD need to be set up. The soil moisture sensor is connected to an analog input pin on the ESP32. The relay module's control input is connected to a digital output pin on the ESP32. The LCD is connected using the I2C interface. Proper wiring is essential to ensure correct data transmission and control. Additionally, the power supply should be checked to provide the correct voltage to all components.

3. Coding:

```
#include <WiFi.h>
#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

const char SSID[] = "the_pianist";
const char PASS[] = "2002";

const int sensorPin = 34;
const int relayPin = 5;

int soilMoisture;
bool motorControl = false;

LiquidCrystal_I2C lcd(0x27, 16, 2);

void setup() {
  Serial.begin(9600);
  lcd.begin();
  lcd.backlight();

  WiFi.begin(SSID, PASS);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("Connected to WiFi");

  ArduinoCloud.begin(ArduinoIoTPreferredConnection);

  ArduinoCloud.addProperty(soilMoisture, READ, 1 * SECONDS, NULL);
     ArduinoCloud.addProperty(motorControl,    READWRITE,    ON_CHANGE,
onMotorControlChange);

  pinMode(sensorPin, INPUT);
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, LOW);
}

void loop() {
  ArduinoCloud.update();
  int moistureLevel = analogRead(sensorPin);
  Serial.print("Moisture Level: ");
  Serial.println(moistureLevel);

  soilMoisture = moistureLevel;

  lcd.setCursor(0, 0);
  lcd.print("Moisture: ");
  lcd.print(moistureLevel);
  lcd.setCursor(0, 1);
  lcd.print(motorControl ? "Pump: ON" : "Pump: OFF");

  if (motorControl) {
    digitalWrite(relayPin, HIGH);
    Serial.println("Motor Pump ON");
  } else {
    digitalWrite(relayPin, LOW);
    Serial.println("Motor Pump OFF");
  }

  delay(2000);
}

void onMotorControlChange() {
  if (motorControl) {
    digitalWrite(relayPin, HIGH);
  } else {
    digitalWrite(relayPin, LOW);
  }
}
```

This code includes essential functionalities for connecting to Wi-Fi, interacting with the Arduino IoT Cloud, reading sensor data, displaying information on an LCD, and controlling a motor pump. Each section is explained to clarify its purpose and connection to other parts of the code.

Library Inclusions: Essential libraries are included at the beginning of the code. These libraries manage Wi-Fi connectivity, cloud communication, I2C communication, and LCD display functionality.

Variable Declarations: Variables for sensor readings, motor control, and hardware pin definitions are declared. These variables are used to store and manage data throughout the program.

Setup Function: This function initializes hardware components, connects to Wi-Fi, and sets up the Arduino IoT Cloud connection. It also configures properties for cloud synchronization and sets initial pin states.

Loop Function: The loop function continuously updates cloud properties with sensor readings, updates the LCD display, and controls the relay based on cloud data. This function operates in a recurring loop, ensuring that the system remains responsive to changes.

Callback Function: A callback function is defined to handle changes in cloud properties. This function is triggered whenever the motor control property is updated in the cloud, allowing for real-time control of the motor pump.

4. Testing the setup:

After uploading the code, the system needs to be tested to ensure that it operates as expected. This involves checking the Wi-Fi connection status, verifying that sensor readings are accurate, ensuring that the LCD displays the correct information, and confirming that the relay properly controls the motor pump based on cloud data. Any issues encountered during testing should be addressed by reviewing the code and hardware connections.

Block Diagram:

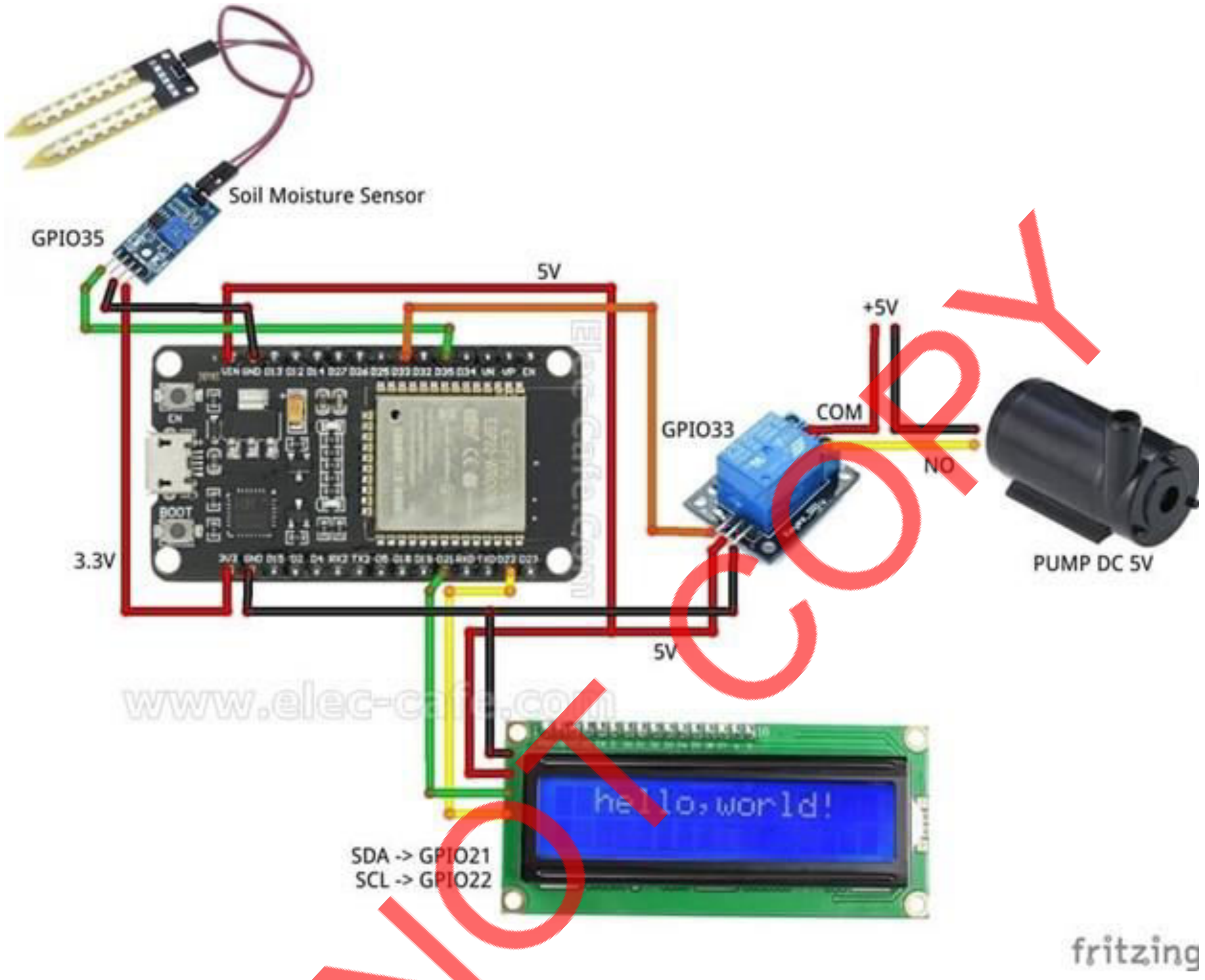

Fig 2: block diagram for Problem 1 [5]

PROBLEM 2

Write a program to send “Hello!” and “CSE 4326!” using Wi-Fi to the cloud and then read the data from the cloud. A green and red led light is connected to another ESP32. If “Hello!” is sent, then a green led would turn on and if “CSE 4326!” is sent then the red led would turn on.

Hardware Implementation:

A detailed hardware integration for the problem is described as follows:

1. ESP32 Modules: The two ESP32 microcontrollers are essential components. One ESP32 will be responsible for sending messages ("Hello!" and "CSE 4326!") to the cloud via Wi-Fi, while the other ESP32 will receive these messages and control the LEDs accordingly. The first ESP32 will require a proper power supply, typically through a USB cable connected to a computer or external power source. Similarly, the second ESP32 will also need power through a USB connection. Both ESP32s will be configured for Wi-Fi communication.

2. Green and Red LEDs: The two LEDs will be connected to the second ESP32. The green LED will light up when the message “Hello!” is received from the cloud, while the red LED will light up when the message “CSE 4326!” is received. Each LED will be connected to a digital GPIO pin on the ESP32. The appropriate pins will be set as outputs in the code. Resistors will be connected in series with each LED to prevent excessive current flow that could damage the LEDs.

3. Resistors: Each LED will require a resistor in series to limit the current. A typical value for these resistors might be 220 ohms, which ensures that the LEDs receive a safe amount of current when the GPIO pin outputs a HIGH signal.

4. Wi-Fi Router: The ESP32 modules will connect to the cloud through a Wi-Fi network provided by the router. Both ESP32s must be connected to the same network, allowing the cloud to serve as the intermediary for message exchange. The Wi-Fi credentials will be configured within the code.

5. Jumper Wires: These will be used to establish electrical connections between the ESP32s, the LEDs, and the resistors. The jumper wires will be inserted into the breadboard to complete the circuits and link the components together without soldering.

6. Breadboard: The breadboard will serve as a non-permanent platform for building the circuit. It will allow the LEDs, resistors, and ESP32s to be connected securely, enabling quick and easy adjustments during testing and development.

7. USB Cable: The USB cables will provide power to the ESP32 modules and also facilitate the uploading of code from the computer to the ESP32. Both ESP32s will be connected to a computer using USB cables during the programming phase.

Software Implementation:

A detailed description of the software involved in the system and their integration is described as follows:

1. Arduino IDE:

The Arduino IDE will be used to write, compile, and upload the code to both ESP32 modules. It provides a user-friendly platform where the code will be written, leveraging the necessary libraries for cloud communication and GPIO control. The Arduino IDE also offers debugging capabilities through the Serial Monitor, allowing the user to view real-time data such as Wi-Fi connection status, cloud messages, and LED control signals.

2. Arduino IoT Cloud:

The Arduino IoT Cloud will be an essential platform for managing communication between the two ESP32 modules. Through this cloud service, data will be sent and received from one ESP32 to the other. The first ESP32 will send the messages ("Hello!" and "CSE 4326!") to the cloud, and the second ESP32 will retrieve this data. The IoT Cloud provides built-in functionality to handle device-to-cloud communication, simplifying the process of managing data flow between multiple devices. The cloud platform will store and synchronize the data, allowing for efficient control of the LEDs on the second ESP32.

3. ArduinoIoTCloud Library:

This library will facilitate communication between the ESP32 modules and the Arduino IoT Cloud. It provides functions to define properties (such as the messages being sent) and handle changes in those properties. By integrating this library into the code, the ESP32s can send data (e.g., "Hello!" or "CSE 4326!") to the cloud and receive updates on the second ESP32 for controlling the LEDs based on the received message.

4. Wi-Fi Library:

The Wi-Fi library will enable both ESP32 modules to connect to the local Wi-Fi network. It provides the functions needed to initialize the Wi-Fi connection, handle authentication (SSID and password), and check the connection status. Once connected, the ESP32 modules can interact with the cloud through the Wi-Fi network. The Wi-Fi library handles all low-level networking processes, making it easier for the ESP32s to communicate with the Arduino IoT Cloud.

5. Arduino_ConnectionHandler Library:

This library will be used to manage the cloud connection, ensuring that the ESP32 modules maintain an active link with the Arduino IoT Cloud. It simplifies the process of setting up the connection between the devices and the cloud, handling potential disconnections or connection drops. This library ensures that data is transmitted and received reliably between the ESP32s and the cloud.

CODE:

```
#include <WiFi.h>
#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

const char SSID[] = "the_pianist";
const char PASS[] = "2002";

const int greenLEDPin = 12;
const int redLEDPin = 13;

String receivedMessage = "";
const String message1 = "Hello!";
const String message2 = "CSE 4326!";
```

```
void setup() {
  Serial.begin(9600);

  pinMode(greenLEDPin, OUTPUT);
  pinMode(redLEDPin, OUTPUT);
  digitalWrite(greenLEDPin, LOW);
  digitalWrite(redLEDPin, LOW);

  WiFi.begin(SSID, PASS);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("WiFi connected");

  ArduinoCloud.begin(ArduinoIoTPreferredConnection);

    ArduinoCloud.addProperty(receivedMessage,   READWRITE,   ON_CHANGE,
onMessageReceived);
}

void loop() {
  ArduinoCloud.update();
}

void onMessageReceived() {
  Serial.println("Message received: " + receivedMessage);

  if (receivedMessage == message1) {
    digitalWrite(greenLEDPin, HIGH);
    digitalWrite(redLEDPin, LOW);
  } else if (receivedMessage == message2) {
    digitalWrite(greenLEDPin, LOW);
    digitalWrite(redLEDPin, HIGH);
  } else {
    digitalWrite(greenLEDPin, LOW);
    digitalWrite(redLEDPin, LOW);
  }
}
```

Procedure:

1. Setting Up the Development Environment:

The Arduino IDE is installed and set up on the development machine. After installation, the necessary libraries, including *WiFi.h*, *ArduinoIoTCloud.h*, and *LiquidCrystal_I2C.h* are installed. These libraries are required for the ESP32 to connect to Wi-Fi, interact with the Arduino IoT Cloud, and handle the LCD display. Installation is typically done through the Arduino Library Manager.

2. Connecting the Hardware Components:

The ESP32 microcontroller is connected to two LEDs: a green LED connected to GPIO pin 12, and a red LED connected to GPIO pin 13. The LEDs are connected to appropriate resistors to prevent excessive current flow. Power is supplied to the ESP32, and it is connected to the computer through USB for both power and communication.

3. Connecting to the Wi-Fi Network:

The code includes the configuration of Wi-Fi credentials (SSID and password) to establish the connection. The variables *SSID* and *PASS* are set to the actual network credentials.

```
const char SSID[] = "the_pianist";
const char PASS[] = "2002";
```

The *setup()* function contains the process to initiate the Wi-Fi connection using *WiFi.begin()* with the SSID and password. A loop is repeated until the ESP32 successfully connects to the network, and a confirmation message is printed to the serial monitor.

```


WiFi.begin(SSID, PASS);
```

```
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
  }
  Serial.println("WiFi connected");
```

4. Initializing Cloud Connection:

The Arduino IoT Cloud connection is configured in the *setup()* function after establishing the Wi-Fi connection. The *ArduinoCloud.begin()* function is called to initiate the cloud connection.

```
  ArduinoCloud.begin(ArduinoIoTPreferredConnection);
```

A cloud property receivedMessage is defined to store and sync messages received from the cloud. This property is added with READWRITE permissions, allowing both reading and writing to the cloud.

```
ArduinoCloud.addProperty(receivedMessage,    READWRITE,    ON_CHANGE,
onMessageReceived);
```

5. Handling LED Control Based on Cloud Message:

LEDs are connected to GPIO pins, with the green LED on pin 12 and the red LED on pin 13. In the *setup()* function, these pins are initialized as outputs.

```
   pinMode(greenLEDPin, OUTPUT);

  pinMode(redLEDPin, OUTPUT);
  digitalWrite(greenLEDPin, LOW);
  digitalWrite(redLEDPin, LOW);
```

6. Sending and Receiving Data from the Cloud:

In the *loop()* function, *ArduinoCloud.update()* is called to keep the cloud properties synchronized and ensure that the *receivedMessage* property is updated with new data.

```
  ArduinoCloud.update();
```

7. Processing Cloud Data and Controlling LEDs:

When a message is received from the cloud, the *onMessageReceived()* function is called. This function checks if the received message is "Hello!" or "CSE 4326!". Based on the message, it controls the corresponding LED. If "Hello!" is received, the green LED is turned on, and the red LED is turned off. If “CSE 4326!" is received, the red LED is turned on, and the green LED is turned off.

```
    void onMessageReceived() {
        if (receivedMessage == message1) {
            digitalWrite(greenLEDPin, HIGH);
            digitalWrite(redLEDPin, LOW);
        } else if (receivedMessage == message2) {
            digitalWrite(greenLEDPin, LOW);
            digitalWrite(redLEDPin, HIGH);
        } else {
            digitalWrite(greenLEDPin, LOW);
            digitalWrite(redLEDPin, LOW);
        }
    }
```

8. Testing the Setup:

After uploading the code to the ESP32, the system is tested by sending the messages "Hello!" and "CSE 4326!" from the cloud dashboard or another cloud interface. Observing the LEDs should confirm that they turn on as expected based on the sent message.

PROBLEM 3

Create a local ESP32 server on which two Widgets are shown, one is a button switch which can be pressed to turn ON or OFF. The other widget is an LED image/symbol which shows the status of the LED i.e., when the light in a room is ON, the LED button turns ON else OFF.

Hardware Implementation

Push Button:

The push button is a simple electrical component used to send a signal to the ESP32 when pressed. It will be connected to a GPIO pin on the ESP32. Pressing the button will toggle the LED's state and send a corresponding request to the local server running on the ESP32. A pull-down resistor is used to ensure that the button's input reads LOW when not pressed, and HIGH when pressed.

LED:

The LED is connected to another GPIO pin on the ESP32. A current-limiting resistor is placed in series with the LED to protect it from excessive current. The LED will indicate the state of the light in the room based on the button's status**.**

Resistors:

The pull-down resistor is connected between the GPIO pin and ground for the push button. The current-limiting resistor is placed in series with the LED to ensure proper current flow and prevent damage.

Software Implementation

ESPAsyncWebServer Library:

This library is integrated into the Arduino IDE code to create and manage the web server. It enables the ESP32 to serve web pages and handle user interactions, such as pressing buttons to turn the LED on or off. The web server code uses this library to update the status of the LED based on the button's state [6].

AsyncTCP Library:

The AsyncTCP library is included as a dependency for the ESPAsyncWebServer library. It ensures that the ESP32 can handle asynchronous network communications, allowing the web server to operate smoothly and respond to user actions without delays.

ESP32 Board Support Package (BSP) for Arduino:

The BSP is installed in the Arduino IDE to enable programming the ESP32. It includes the necessary core libraries and functions for interacting with the hardware, such as handling GPIO pins and managing Wi-Fi connections. This support package ensures that the code written in the Arduino IDE can be correctly compiled and uploaded to the ESP32.

CODE:

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>

const char* ssid = "the_pianist";
const char* password = "2002";

const int ledPin = 2;

AsyncWebServer server(80);

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("Connected to WiFi");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());

  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    String html = "<html><body>";
    html += "<h1>ESP32 LED Control</h1>";
    html += "<button onclick=\"toggleLED()\">Toggle LED</button>";
    html += "<br/><br/>";
    html += "<img src=\"/led_status\" id=\"ledStatus\" />";
    html += "<script>";
    html += "function toggleLED() {";
    html += "  fetch('/toggle_led');";
    html += "  setTimeout(updateLEDStatus, 500);";
    html += "}";
    html += "function updateLEDStatus() {";
          html    +=    "     fetch('/led_status').then(response    =>
response.text()).then(status => {";
    html += "    document.getElementById('ledStatus').src = status ===
'ON' ? '/led_on.png' : '/led_off.png';";
    html += "  });";
    html += "}";
    html += "updateLEDStatus();";
    html += "</script>";
    html += "</body></html>";
    request->send(200, "text/html", html);
  });

     server.on("/toggle_led",    HTTP_GET,    [](AsyncWebServerRequest
*request){
    int ledState = digitalRead(ledPin);
    digitalWrite(ledPin, !ledState);
    request->send(200, "text/plain", "LED Toggled");
  });

     server.on("/led_status",    HTTP_GET,    [](AsyncWebServerRequest
*request){
    int ledState = digitalRead(ledPin);
    request->send(200, "text/plain", ledState == HIGH ? "ON" : "OFF");
  });

  server.begin();
}
```

```
void loop() {
      //codes are unnecessary as the server is handling requests asnychronously
}
```

Procedure:

1. Hardware assembling:

The ESP32 microcontroller and an LED connected to GPIO 2 must be set up. The LED is connected to its anode to GPIO 2 and the cathode to a ground pin on the ESP32. Proper wiring is crucial to ensure correct LED operation. The power supply should be checked to provide the correct voltage to the ESP32 and connected components.

2. Libraries and Functions:

Library Inclusions: Essential libraries are included at the beginning of the code. These libraries manage Wi-Fi connectivity and web server functionality.

Variable Declarations: Variables for the Wi-Fi credentials and LED control are declared. These variables manage the connection and LED state.

Setup() Function: This function initializes serial communication for debugging, sets the LED pin mode, connects to the Wi-Fi network, and starts the web server. It also configures the web server endpoints to handle HTTP requests for toggling the LED and updating its status.

Loop() Function: The *loop()* function remains empty as all the necessary tasks are handled by the web server and Wi-Fi connection, which operate asynchronously.

3. Finding the ESP32 IP Address:

Once the ESP32 is connected to the Wi-Fi network, the IP address assigned to the ESP32 is printed to the Serial Monitor. This IP address is required to access the local server hosted on the ESP32.

4. Accessing the Webpage:

The IP address of the ESP32 is entered into a web browser on a computer or mobile device. This action loads the local server's webpage, which includes a button for toggling the LED and an image showing the LED status.

5. Testing the Setup:

On the webpage, the button can be pressed to toggle the LED on or off. The LED status image updates based on the current state of the LED. Observations are made to ensure that the LED responds correctly to button presses and that the status image accurately reflects the LED's state.

CODE Explaination:

1. Libraries:

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
```

2. Variables:

```
const char* ssid = "the_pianist";
const char* password = "2002";
```

These variables define the SSID and password for the Wi-Fi network that the ESP32 will connect to. They must match the network credentials of the Wi-Fi router.

```
const int ledPin = 2;
```

This variable defines the GPIO pin (pin 2) to which an LED is connected. Pin 2 is often used as a built-in LED pin on ESP32 boards.

3. Server Initialization:

```
AsyncWebServer server(80);
```

This creates an instance of the *AsyncWebServer* object on port 80, which is the default HTTP port. The web server will listen for incoming HTTP requests on this port.

4. Setup Function:

```
void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
```

The *setup()* function starts the serial communication with a baud rate of 9600 for debugging purposes. The *ledPin* is set as an output so that the LED can be controlled.

5. Connecting to Wi-Fi:

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
Serial.println("Connected to WiFi");
Serial.print("IP Address: ");
Serial.println(WiFi.localIP());
```

This segment attempts to connect the ESP32 to the Wi-Fi network specified by the *ssid* and *password*. It continuously checks the connection status, printing dots in the serial monitor until a connection is established. Once connected, the IP address of the ESP32 is printed.

6. Handling Web Requests:

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
```

This defines the route for the root URL. When a GET request is made to this URL, the server responds with an HTML page containing a button to toggle the LED and an image that reflects the current LED status.

The HTML structure includes following JavaScript functions:

Toggling the LED: When the button is clicked, the *toggleLED()* function sends a request to *toggle_led* to change the LED state.

Updating the LED status: The *updateLEDStatus()* function fetches the current status from *led_status* and updates the image to reflect whether the LED is on or off.

Toggle LED:

```
server.on("/toggle_led", HTTP_GET, [](AsyncWebServerRequest *request){
  int ledState = digitalRead(ledPin);
  digitalWrite(ledPin, !ledState);
  request->send(200, "text/plain", "LED Toggled");
});
```

This handler listens for GET requests to the *toggle_led* route. When the route is accessed, it reads the current state of the LED, toggles it (i.e., if it's on, it turns off, and vice versa), and responds with a text message "LED Toggled."

LED Status:

```
server.on("/led_status", HTTP_GET, [](AsyncWebServerRequest *request){
  int ledState = digitalRead(ledPin);
  request->send(200, "text/plain", ledState == HIGH ? "ON" : "OFF");
});
```

This route provides the current status of the LED (either "ON" or "OFF") by reading the value from *ledPin*.

7. Server Begin:

```
server.begin();
```

This starts the asynchronous web server, allowing it to handle incoming requests.

8. Loop Function:

```
void loop() {
      //codes are unnecessary as the server is handling requests
asynchronously
}
```

Since the web server is asynchronous, no code is needed in the *loop()* function. The server will handle incoming requests in the background**.**

PROBLEM 4

A gas sensor is connected in your kitchen to measure the gas in ppm. The sensor is connected to ESP32 to send the data in the cloud. Now, write the necessary programs to save the gas data from the cloud in a google sheet or excel file. Then, read from the Excel file and turn on a buzzer if the gas value crosses 50 ppm.

Hardwares

1. ESP32 Microcontroller

2. Gas Sensor MQ-5:

The MQ-5 gas sensor is a widely used module for detecting various gases like LPG (liquefied petroleum gas), natural gas, methane, and other combustible gases [7]. Here's a breakdown of how the MQ-5 sensor works:

a) **Sensing Principle:** The MQ-5 sensor works on the principle of metal oxide semiconductor (MOS) conductivity. It has a sensing element made of tin dioxide (SnO_2). When the target gas is present, it reacts chemically with the tin dioxide, changing its conductivity.

b) **Heating Element:** The sensor includes a small built-in heater that warms the sensing element to an optimal temperature, usually between 200-400°C. This heating process enhances the sensor's sensitivity and speeds up its response time.

c) **Resistance Change:** When the target gas is detected, the resistance of the tin dioxide changes. This resistance change is proportional to the amount of gas in the air—the more gas there is, the bigger the change in resistance.

d) **Load Resistor:** To measure the change in resistance, the sensor is connected in series with a load resistor, forming a voltage divider circuit. The output voltage from this circuit is then measured and can be linked to the gas concentration after calibration.

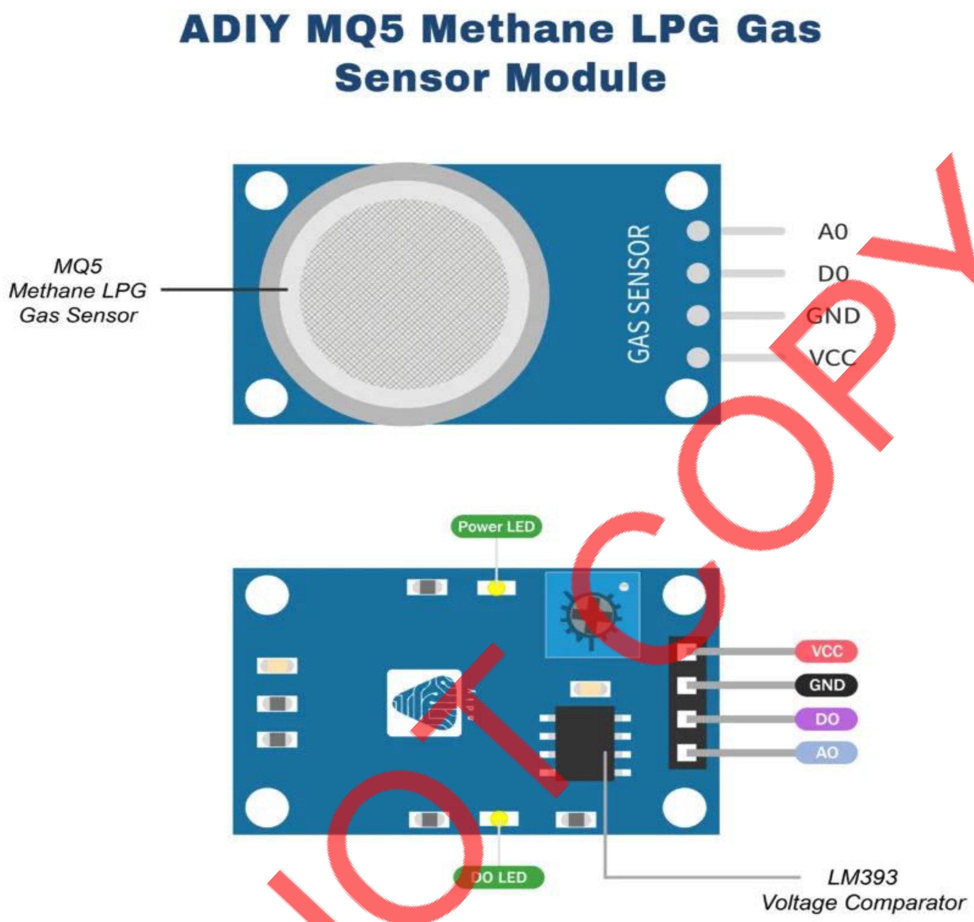

Fig 3: MQ-5 sensor Diagram [7]

e) **Interface Circuitry:** The output voltage from the MQ-5 sensor needs to be processed by additional circuits before it can be connected to an Arduino or other microcontrollers. This processing might include amplification, filtering, and analog-to-digital conversion, depending on the project's needs.

f) **Calibration:** The MQ-5 sensor needs to be calibrated to ensure accurate gas readings. This involves exposing the sensor to known concentrations of the target gas and adjusting its output accordingly. Calibration is usually done when the sensor is first set up.

g) **Response and Recovery Time:** The sensor typically responds to gas in tens of seconds to a few minutes, depending on factors like gas concentration and environmental conditions. Recovery time—the time it takes for the sensor to return to its normal state after detecting gas—is also important.

h) **Applications:** The MQ-5 gas sensor is commonly used in gas leak detection systems, industrial safety setups, and various DIY projects that involve gas sensing. It can be easily integrated with platforms like Arduino, Raspberry Pi, or other microcontrollers for data collection and processing.

3. Buzzer

The buzzer is a small audio signaling device that emits a sound when activated. It will be triggered by the ESP32 when the gas concentration exceeds the defined threshold of 50 ppm, acting as an alarm.

4. Breadboard and Jumper Wires

5. Power Supply (USB cable or battery)

Hardware Integration

The MQ-5 gas sensor will be connected to the ESP32's analog input pin to read the gas concentration. The ESP32 will then process this data and send it to the cloud for storage. A buzzer will be connected to one of the ESP32's digital output pins to sound an alert when the gas levels exceed 50 ppm. The ESP32 will communicate with the cloud using its built-in Wi-Fi, allowing the data to be saved into Google Sheets via cloud APIs.

Software Components:

1. Arduino IDE

2. ESP32 Core for Arduino

The ESP32 core provides compatibility for programming ESP32 devices using the Arduino IDE. This core includes functions that enable Wi-Fi communication, control of GPIO pins, and other essential ESP32 features.

3. Google Sheets API

The Google Sheets API will be used to store and retrieve gas sensor data from the cloud. This will allow remote access to the collected data and the ability to trigger alerts based on real-time readings.

4. Arduino Libraries

WiFi.h: This library allows the ESP32 to connect to the Wi-Fi network for cloud communication.

HTTPClient.h: The HTTP Client library is required to send HTTP requests from the ESP32 to the cloud API to store the data.

MQUnifiedsensor.h: This library will provide functions specific to the MQ series of gas sensors to process the data correctly [9].

5. Google Apps Script

To automate data processing and detection of gas levels crossing the threshold, a script using Google Apps Script will be used. This script will monitor gas values and trigger the logic required to send a command to the ESP32 for turning on the buzzer.

6. Serial Monitor

The serial monitor within the Arduino IDE will be used to display real-time gas sensor readings for debugging purposes during the development and testing phase.

Software Integration

The Arduino IDE will be used to write and upload code to the ESP32, with the ESP32 core providing essential libraries for communication and control. The MQUnifiedsensor.h library will allow the ESP32 to read accurate values from the MQ-5 sensor. The WiFi.h and HTTPClient.h libraries will manage the connection to the Wi-Fi network and the communication with cloud services for storing data.

The Google Sheets API will be integrated to store the gas sensor data in real-time. It will be used to read the stored data periodically and check if the gas level exceeds 50 ppm. If this threshold is crossed, a command will be sent back to the ESP32 to activate the buzzer, using the same Wi-Fi connection for seamless control.

Lastly, the Serial Monitor in the Arduino IDE will assist in debugging and real-time monitoring of gas levels, helping to ensure that the data flow and control logic work as intended.

Procedure:

The following code will be run on the Arduino IDE:

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <MQUnifiedsensor.h>

const char* ssid = "the_pianist";
const char* password = "2002";

String                              googleScriptURL                              =
"https://script.google.com/macros/s/AKfycby9w6_d9ExampleIDghjkl90XYZ/e
xec";

#define BUZZER_PIN 4
#define MQ5_PIN 34

#define Board "ESP-32"
#define Voltage_Resolution 5
#define ADC_Bit_Resolution 10
#define RatioMQ5CleanAir 6.5


MQUnifiedsensor   MQ5(Board,   Voltage_Resolution,   ADC_Bit_Resolution,
MQ5_PIN, RatioMQ5CleanAir);


void setup() {
  Serial.begin(9600);
  pinMode(BUZZER_PIN, OUTPUT);
  digitalWrite(BUZZER_PIN, LOW);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");
  MQ5.setRegressionMethod(1);
  MQ5.init();
}



// sending gas data to Google Sheets
void sendDataToCloud(float gasPPM) {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    // building URL with gas data
    String url = googleScriptURL + "?gas=" + String(gasPPM);
    http.begin(url);
    int httpResponseCode = http.GET();
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Data sent to cloud: " + response);
    } else {
      Serial.println("Error sending data to cloud");
    }
    http.end();
  } else {
    Serial.println("WiFi not connected");
  }
}

// reading gas levels and control buzzer
void loop() {
  MQ5.update();
  float gasPPM = MQ5.readSensor(); // reading gas concentration
  Serial.print("Gas concentration: ");
  Serial.println(gasPPM);

  sendDataToCloud(gasPPM);

  // buzzer logic
  if (gasPPM > 50) {
    digitalWrite(BUZZER_PIN, HIGH);
    Serial.println("Buzzer ON - Gas level exceeded 50 ppm");
  } else {
    digitalWrite(BUZZER_PIN, LOW);
    Serial.println("Buzzer OFF");
  }
  delay(2000);
}
```

In addition to the code that runs on the Arduino IDE for the ESP32, several other components are required to handle cloud communication and interaction with Google Sheets. These components ensure that the data from the gas sensor is stored in Google Sheets and monitored to trigger a buzzer when needed.

Google Apps Script for receiving and storing data in Google Sheets

A Google Apps Script is needed to accept HTTP POST requests from the ESP32 and store the gas data in a Google Sheet. This script functions as a webhook endpoint where the ESP32 sends its data [8].

Google Apps Script:

```
function doPost(e) {
var gasValue = e.parameter.gas;
var sheet =
SpreadsheetApp.openById('https://docs.google.com/spreadsheets/d/3fLMnO
pQRsTuVwXyZ1234567aBcD8EfGhIjKLM9NoPqRsTuvWxYz/edit').getActiveSheet()
;
  var timestamp = new Date();
  sheet.appendRow([timestamp, gasValue]);
        return       ContentService.createTextOutput("Data       stored
successfully");
}
```

Once the script is created, it is deployed as a web app. Permissions are set to allow access from anyone (including anonymous users), enabling the ESP32 to send data. The web app URL generated will be used in the ESP32 code for data transmission.

Google Sheets for Data Storage

A Google Sheet is used to store the gas sensor data received from the ESP32. Each time the data is sent, the Google Apps Script appends the gas value and timestamp to a new row in the sheet [8].

Random Values from the Google Sheet:

```
     Timestamp                        Gas Value (ppm)
2024-09-04 10:15:30                        35.7
2024-09-04 10:18:40                        51.2
```

The gas sensor data is stored in real-time, with each new measurement being added automatically.

DISCUSSION

The solution to the given four problems came through a lot of trial and error. The major challenges involved scripting, testing various setups, and finding proper libraries etc. During several dry runs, quite a few calculations seemed tricky or misleading. Mastering the Arduino IoT cloud library functions and implementing them with necessary coding and analytical skills was the most difficult part of this experiment. However, with the help of a lot of tutorials provided by our honorable teacher and some key resources mentioned in the reference section, the problems were eventually taken care of with utmost care and precision.

REFERENCES

1. **https://lastminuteengineers.com/esp32-vs-esp8266-comparison/**
2. **https://circuitdigest.com/microcontroller-projects/interfacing-soil-moisture-sensor-with-arduino-uno**
3. **https://lastminuteengineers.com/i2c-lcd-arduino-tutorial/**
4. **https://www.youtube.com/watch?v=qQGM5oBKAZc**
5. **https://randomnerdtutorials.com/guide-for-soil-moisture-sensor-yl-69-or-hl-69-with-the-arduino/**
6. **https://www.youtube.com/watch?v=aUSwEkJCIAA**
7. **https://www.youtube.com/watch?v=eJwyH-SizVU&t=10s**
8. **https://www.youtube.com/watch?v=u7TYu61l0t4**
9. **https://github.com/miguel5612/MQSensorsLib/blob/master/src/MQUnifiedsensor.h**