SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University)

FACULTY OF SCIENCE AND HUMANITIES

TIRUCHIRAPPALLI - 621105

DEPARTMENT OF COMPUTER APPLICATIONS

PRACTICAL RECORD

NAME	**:**	
REGISTER NUMBER	**:**	
COURSE	**:**	**MCA**
SEMESTER/YEAR	**:**	**III / II**
SUBJECT CODE	**:**	**PCA20D09J**
SUBJECT NAME	**:**	**INTERNET OF THINGS LAB**

NOVEMBER 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University)

FACULTY OF SCIENCE AND HUMANITIES

TIRUCHIRAPPALLI - 621105

DEPARTMENT OF COMPUTER APPLICATIONS

BONAFIDE CERTIFICATE

This is to certify that the bonafide work done by________________________ Register No:__________________ for the course **"INTERNET OF THINGS LAB-(PCA20D09J) "** at, SRM Institute of Science and Technology, Tiruchirappalli in November 2025.

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

The record is submitted for the University Practical Examination held at SRM Institute of Science and Technology on --------------.

Submitted for the University Practical Examination held at FSH Computer Lab, **SRM Institute of Science and Technology**, Tiruchirappalli.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University)

FACULTY OF SCIENCE AND HUMANITIES

TIRUCHIRAPPALLI - 621105

PCA20D09J – INTERNET OF THINGS LAB

Ex. No. : 1	Define and Explain Eclipse IoT Project
Date: 14-07-2025	

Aim

To study and understand the Eclipse IoT Project, its components, purpose, and importance in building open-source Internet of Things (IoT) solutions.

Introduction

Eclipse IoT refers to a set of open-source projects under the Eclipse Foundation, focused on building an open Internet of Things (IoT) ecosystem. The goal is to provide a comprehensive platform and set of tools for developing, deploying, and managing IoT solutions, from devices to cloud-based applications.

Key Components of Eclipse IoT:

1. Device Connectivity and Communication:

   Eclipse Paho: Provides client implementations of the MQTT (Message Queuing Telemetry Transport) protocol, which enables lightweight and efficient messaging between IoT devices.
   Eclipse Milo: Implements the OPC UA (Open Platform Communications Unified Architecture) standard used in industrial automation and IoT for secure, reliable data exchange.
2. IoT Gateways:

   Eclipse Kura: A Java-based framework for building IoT gateways, supporting device management, network configuration, and multiple communication protocols.
3. IoT Cloud Platforms:

   Eclipse Hono: Offers a uniform API for connecting and managing a large number of IoT devices and forwarding telemetry data to cloud services.
   Eclipse Kapua: A modular IoT cloud platform providing services such as device management, data management, and application development.
4. Development Tools and Frameworks:

   Eclipse Vorto: Enables developers to define IoT information models and auto-generate platform-specific code for interoperability.
   Eclipse Ditto: Provides tools for managing digital twins, representing virtual counterparts of physical IoT devices.
5. Security and Standards:

   Eclipse Leshan: Implements the Lightweight M2M (LwM2M) protocol to ensure secure device management and communication in IoT environments.

Purpose and Importance:

Open Standards: Promotes interoperability and avoids vendor lock-in by adhering to widely accepted standards.

Community-Driven: Enables rapid innovation and adaptation through global developer collaboration.

End-to-End Ecosystem: Provides integrated tools covering all layers — from device communication to cloud-based data analytics.

Result:

The study of the Eclipse IoT Project demonstrates the open-source tools and frameworks under the Eclipse Foundation for developers to build better IoT solutions.

Ex. No. : 2	List and Summarize few Eclipse IoT Projects
Date: 14-07-2025	

Aim

To list and summarize the key open-source projects under the Eclipse IoT initiative and understand their role in building and managing Internet of Things (IoT) solutions.

List and Summary of Eclipse IoT Projects:

1. Eclipse Mosquitto:
   An open-source MQTT broker implementing a lightweight messaging protocol ideal for small sensors, mobile devices, and high-latency networks.
2. Eclipse Kura:
   A gateway framework providing core services for Java developers, including device communication, data management, and remote configuration.
3. Eclipse Paho:
   A collection of MQTT client libraries for multiple programming languages (Java, C, Python, JavaScript, etc.), enabling machine-to-machine (M2M) communication.
4. Eclipse Hono:
   Offers uniform APIs for connecting IoT devices to a backend infrastructure, enabling data ingestion, device management, and command execution.
5. Eclipse Ditto:
   A digital twin management framework that allows virtual representation of physical IoT devices, facilitating real-time synchronization and communication.
6. Eclipse Vorto:
   A tool used to define IoT device information models, helping to standardize device descriptions and interoperability across platforms.
7. Eclipse IoT Packages:
   A collection of IoT libraries and tools such as Eclipse Concierge, Eclipse Californium (CoAP protocol), and others that support IoT application development.
8. Eclipse Kapua:
   A modular IoT cloud platform designed for managing and integrating edge devices, offering features for gateway management and data analytics.

9. Eclipse Mita:

   A domain-specific programming language created to simplify IoT application development and reduce coding complexity.

10. Eclipse Unide:

    Focuses on Industry 4.0 data standardization by providing open formats and services for production process data exchange.

11. Eclipse Whiskers:

    A software platform for edge computing, enabling low-latency data processing near IoT devices.

12. Eclipse Leshan:

    Implements the Lightweight M2M (LwM2M) protocol in Java for secure device management and communication.

13. Eclipse Agail:

    A lightweight messaging protocol for IoT, similar to MQTT, but optimized for diverse environments and network conditions.

14. Eclipse Fog05:

    A fog computing platform that orchestrates compute, storage, and networking resources across cloud and edge layers, ensuring efficient resource utilization.

15. Eclipse 4diac:

    An industrial automation framework based on the IEC 61499 standard, supporting distributed control systems and IIoT applications.

16. Eclipse Oniro:

    A project aimed at developing a secure, open-source IoT operating system, enabling interoperability across connected devices.

17. Eclipse Cyclone DDS:

    An implementation of the DDS (Data Distribution Service) protocol for real-time systems, ensuring low latency and high reliability in domains like autonomous vehicles and industrial automation.

Result

The study successfully listed and summarized various Eclipse IoT Projects, demonstrating robust ecosystem for developing IoT projects.

Ex. No. : 3	Smart Lighting
Date: 21-07-2025	

Aim

To design and implement a Smart Lighting System using an Arduino microcontroller.

Algorithm

Step 1. Start the Arduino program.

Step 2. Define LED pins, Set pins 13, 12, and 11 as output using the pinMode() function.

Step 3. In the loop() function:

a. Turn ON LED connected to pin 13, wait for 2 seconds, then turn it OFF.

b. Turn ON LED connected to pin 12, wait for 2 seconds, then turn it OFF.

c. Turn ON LED connected to pin 11, wait for 2 seconds, then turn it OFF.

Step 4. The loop repeats indefinitely, creating a continuous sequential lighting effect.

Step 5. Observe the sequence to confirm that lights operate automatically and cyclically.

Source Code

SKETCH.IO:

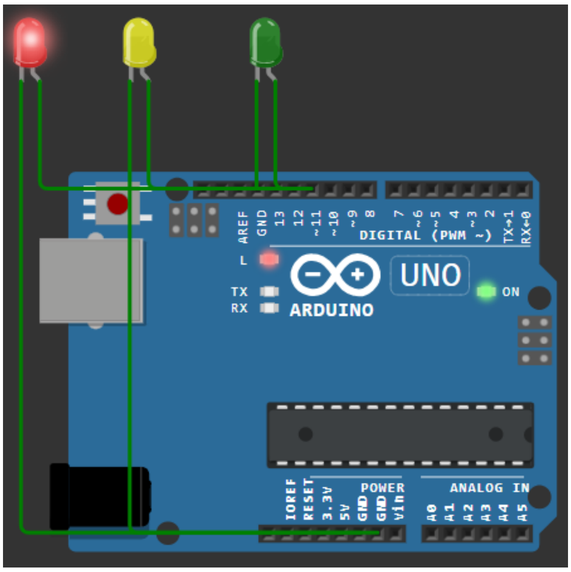
```
void setup() {
  // put your setup code here, to run once:
  pinMode(13, OUTPUT);
  pinMode(12,OUTPUT);
  pinMode(11, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(13, HIGH);
  delay(2000);
  digitalWrite(13, LOW);
  digitalWrite(12, HIGH);
  delay(2000);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  delay(2000);
  digitalWrite(11, LOW);
}
```

Circuit connections

For three LEDs connected externally:

LED Pin	Arduino Pin	Connection
LED1 Anode (A)	13	Connect via resistor (220Ω) to pin 13
LED1 Cathode (C)	GND	Connect to GND
LED2 Anode (A)	12	Connect via resistor to pin 12
LED2 Cathode (C)	GND	Connect to GND
LED3 Anode (A)	11	Connect via resistor to pin 11
LED3 Cathode (C)	GND	Connect to GND

Output



Result

The Arduino successfully controlled three LEDs, demonstrating a basic smart lighting automation system

Ex. No. : 04 (a)	Architecture of IoT
Date: 21-07-2025	

Aim

To study and sketch the architecture of the Internet of Things (IoT) and understand the functions of each layer.

Architecture of IoT

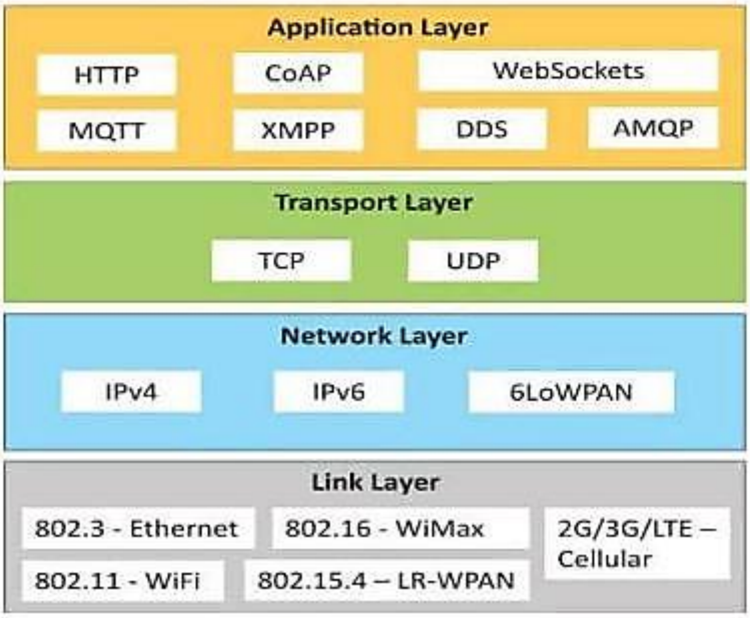


Fig. 1: Architecture of IoT

Description of Layers:

1. Application Layer

   The topmost layer that provides end-user services and IoT applications. It interprets the processed data and delivers meaningful insights through dashboards or control interfaces.
   Examples: Smart home control apps, healthcare monitoring, industrial automation dashboards.

2. Transport Layer

   This layer ensures reliable and efficient communication between IoT devices and cloud servers. It manages end-to-end data transfer using standard internet protocols.
   Common Protocols: TCP (Transmission Control Protocol), UDP (User Datagram Protocol), MQTT, CoAP.

3. Network Layer

   The Network Layer is responsible for routing data packets from source devices to destination servers over the internet. It provides unique addressing using IP/IPv6 and supports low-power wireless networking.
   Technologies: 6LoWPAN, RPL, IP, ICMP.

4. Link (Perception) Layer

   This is the bottom layer, enabling physical connectivity and data sensing. It connects sensors, actuators, and embedded devices to the IoT network. It also handles local communication and medium access control (MAC).
   Examples: Wi-Fi, Bluetooth, Zigbee, RFID, LoRa, NFC.

Result:

Thus, the IoT architecture was demonstrated with seamless data communication and control across different layers in IoT environments.

Ex. No. : 4 (b)	Smart Object API Gateway Service
Date: 21-07-2025	

Aim

To demonstrate smart object API gateway enabling communication between IoT devices and cloud-based services through standardized APIs.

Algorithm

Step 1. Connect sensors or LEDs to an Arduino board representing smart devices.

Step 2. Write a program to collect sensor data (or control LEDs) and send it via serial or network interface.

Step 3. Use a simulated API Gateway (e.g., via Node.js, Python Flask, or MQTT broker) to receive data from Arduino.

Step 4. Format the received data as JSON and expose it as a REST API endpoint.

Step 5. Verify communication by observing transmitted data or API responses on the serial monitor or browser.

Source Code

Arduino Smart Object Simulation

```
int ledPin = 13; // Smart object (LED)
int sensorValue = 0; // Simulated sensor data

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

void loop() {
  // Simulate sensor reading
  sensorValue = analogRead(A0);

  // Send sensor data to API gateway
  Serial.print("Sensor Value: ");
  Serial.println(sensorValue);

  // Simple condition to simulate control from API
  if (sensorValue > 500) {
    digitalWrite(ledPin, HIGH); // Turn ON LED
```

```
    Serial.println("LED ON (Command from Gateway)");
  } else {
    digitalWrite(ledPin, LOW); // Turn OFF LED
    Serial.println("LED OFF (Command from Gateway)");
  }

  delay(2000); // Wait for 2 seconds
}
```

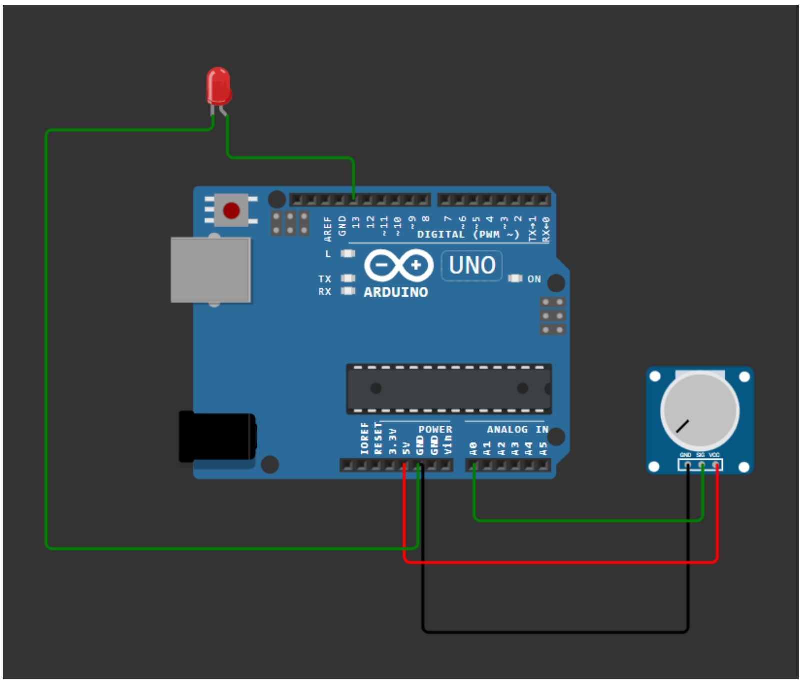
Gateway Simulation (Python)

```
from flask import Flask, jsonify, request

app = Flask(__name__)

# Example endpoint to receive data
@app.route('/sensor', methods=['POST'])
def receive_data():
    data = request.get_json()
    print("Received data:", data)
    return jsonify({"status": "success", "message": "Data received"}), 200

# Example endpoint to send command to device
@app.route('/control', methods=['GET'])
def control_device():
    command = {"LED": "ON"}
    return jsonify(command)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

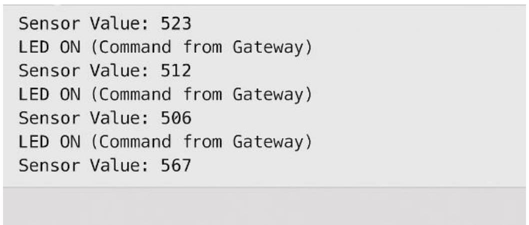
Circuit Connection

Component	**Arduino Pin**	**Connection Details**
Potentiometer Pin 1	5V	Connect to Arduino 5V
Potentiometer Pin 2	A0	Connect to Arduino Analog pin A0
Potentiometer Pin 3	GND	Connect to Arduino GND
LED (built-in)	Pin 13	Already connected inside Arduino board

Device Setup



Output



```
Sensor Value: 523
LED ON (Command from Gateway)
Sensor Value: 512
LED ON (Command from Gateway)
Sensor Value: 506
LED ON (Command from Gateway)
Sensor Value: 567
```

Result:

Thus, a Smart Object API Gateway Service was successfully demonstrated and output verified successfully.

Ex. No. : 5	Working of an HTTP-to-CoAP Semantic Mapping Proxy
Date: 04-08-2025	

Aim

To simulate the working of an HTTP-to-CoAP semantic mapping proxy to demonstrate the HTTP-based requests translation into CoAP-style IoT commands.

Algorithm

Step 1. Start the Arduino serial monitor at 9600 baud rate.

Step 2. Simulate an incoming HTTP request (/temperature and /light).

Step 3. The Arduino program checks the type of HTTP request.

Step 4. It maps that HTTP request to an equivalent CoAP command (like coap://device/temp and coap://device/light).

Step 5. A response message is generated, simulating the CoAP device's reply.

Step 6. The mapped response is printed to the Serial Monitor.

Step 7. Repeat the process for different simulated requests.

Source Code

```
String httpRequest = ""; // Variable to store simulated HTTP request

void setup() {
  Serial.begin(9600);
  Serial.println("HTTP-to-CoAP Mapping Proxy Simulation");
  Serial.println("--------------------------------------");
}

void loop() {
  // Simulating different HTTP requests
  String simulatedRequests[] = {"/temperature", "/light", "/humidity"};
  for (int i = 0; i < 3; i++) {
    httpRequest = simulatedRequests[i];
    Serial.print("\nReceived HTTP Request: ");
    Serial.println(httpRequest);

    // Mapping HTTP request to CoAP equivalent
    if (httpRequest == "/temperature") {
      Serial.println("Mapped to CoAP Resource: coap://device/temp");
      Serial.println("CoAP Response: {\"temperature\": 27.5}");
```

```
    }
    else if (httpRequest == "/light") {
      Serial.println("Mapped to CoAP Resource: coap://device/light");
      Serial.println("CoAP Response: {\"light_status\": \"ON\"}");
    }
    else if (httpRequest == "/humidity") {
      Serial.println("Mapped to CoAP Resource: coap://device/humidity");
      Serial.println("CoAP Response: {\"humidity\": 45.2}");
    }
    else {
      Serial.println("Error: Unknown HTTP Request");
    }

    delay(3000); // Wait 3 seconds before next simulated request
  }

  // Stop after one full simulation cycle
  while(true);
}
```

Circuit Connection

Sensor/Actuator	Pin on Arduino Uno
DHT11/DHT22 DATA	D2
DHT11/DHT22 VCC	5V
DHT11/DHT22 GND	GND
LDR → A0 (with 10k resistor to GND)	A0
LDR other end	5V
LED (via 220Ω resistor)	D8
LED GND	GND

Output

```
HTTP-to-CoAP Mapping Proxy Simulation
-------------------------------------

Received HTTP Request: /temperature
Mapped to CoAP Resource: coap://device/temp
CoAP Response: {"temperature": 27.5}

Received HTTP Request: /light
```

Mapped to CoAP Resource: coap://device/light
CoAP Response: {"light_status": "ON"}

Received HTTP Request: /humidity
Mapped to CoAP Resource: coap://device/humidity
CoAP Response: {"humidity": 45.2}
Result:

Thus, the program has been executed and output verified successfully.

Ex. No. : 6	Gateway as a Service Deployment
Date: 04-08-2025	

Aim

To demonstrate the deployment of Gateway-as-a-Service (GaaS) for enabling seamless communication between IoT devices and cloud services.

Algorithm

Step 1. Initialize ESP32 with Wi-Fi credentials.
Step 2. Connect to Wi-Fi network.
Step 3. Initialize MQTT client and connect to broker (test.mosquitto.org).
Step 4. Read sensor values (temperature, humidity) from DHT22.
Step 5. Format data as JSON {"temperature": xx, "humidity": yy}.
Step 6. Publish sensor data to MQTT topic iot/lab/gateway.
Step 7. Repeat every 5 seconds.
Step 8. Monitor cloud via MQTT subscription tool (MQTT Explorer / mosquitto_sub).

Source Code

```
#include <WiFi.h>
#include <PubSubClient.h>
#include "DHTesp.h"

#define DHT_PIN 15

// WiFi credentials
const char* ssid = "Wokwi-GUEST";  // default Wokwi WiFi
const char* password = "";

// MQTT broker
const char* mqtt_server = "test.mosquitto.org";
const int mqtt_port = 1883;

WiFiClient espClient;
PubSubClient client(espClient);
DHTesp dht;

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);
```

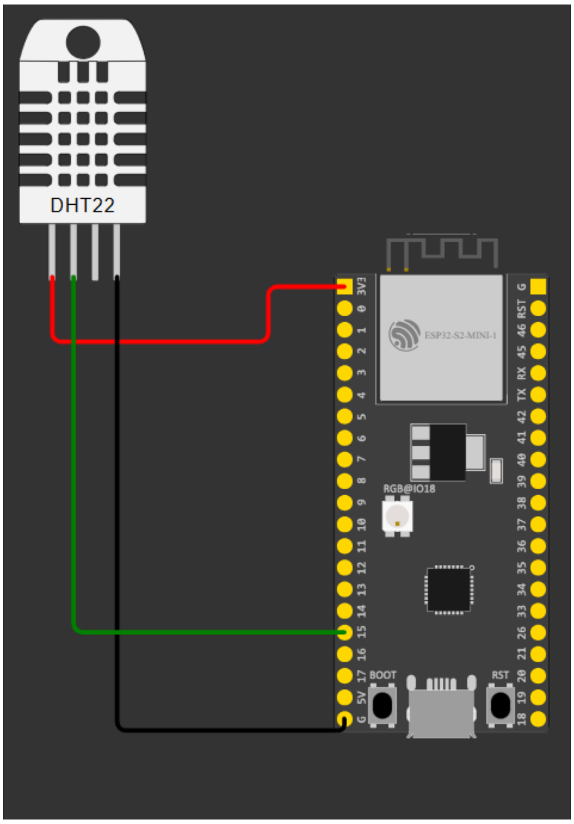
```
  WiFi.begin(ssid, password);

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

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect("ESP32Client")) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  dht.setup(DHT_PIN, DHTesp::DHT22);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  TempAndHumidity data = dht.getTempAndHumidity();

  String payload = "{ \"temperature\": " + String(data.temperature, 2) +
                   ", \"humidity\": " + String(data.humidity, 2) + " }";
```

```
  Serial.print("Publishing message: ");
  Serial.println(payload);

  client.publish("iot/lab/gateway", payload.c_str());
  delay(5000); // send every 5 seconds
}
```

Circuit Connections

DHT22 Pin	ESP32 Pin	Notes
VCC	3.3V	Power supply
GND	GND	Ground
DATA	GPIO 15	Digital data pin (DHT_PIN)

Device Setup



Output

Serial Monitor
WiFi connected
IP address: 192.168.1.20
Attempting MQTT connection... connected
Publishing message: { "temperature": 28.45, "humidity": 67.10 }
Publishing message: { "temperature": 28.49, "humidity": 66.98 }

...
MQTT Subscriber
$ mosquitto_sub -h test.mosquitto.org -t "iot/lab/gateway"
{ "temperature": 28.45, "humidity": 67.10 }
{ "temperature": 28.49, "humidity": 66.98 }
...

Result:

Thus the program has been executed and output verified successfully.

Ex. No. : 7	Application Framework and Embedded Software Agents in IoT Toolkit
Date: 15-09-2025	

Aim

To demonstrate the role of an application framework and embedded software agents in IoT.

Algorithm

Step 1: Initialize ESP32 and configure Wi-Fi connection.
Step 2: Set up MQTT client for communication with cloud broker (test.mosquitto.org).
Step 3: Initialize sensor (DHT22) to capture temperature and humidity.
Step 4: Embedded agent behavior:
  i. Read sensor values.
  ii. Apply local logic (e.g., if temperature > 30°C, mark status as "Alert").
  iii. Format data into JSON.
Step 5: Application framework behavior:
  i. Manage communication protocol (MQTT).
  ii. Publish data periodically to cloud topic (`iot/lab/agents`).
Step 6: Loop continuously for real-time operation.

Source Code

```
#include <WiFi.h>
#include <PubSubClient.h>
#include "DHTesp.h"

#define DHT_PIN 15

// WiFi credentials
const char* ssid = "Wokwi-GUEST";  // Default Wokwi WiFi
const char* password = "";

// MQTT broker
const char* mqtt_server = "test.mosquitto.org";
const int mqtt_port = 1883;

WiFiClient espClient;
PubSubClient client(espClient);
DHTesp dht;

void setup_wifi() {
  delay(10);
  Serial.println();
```

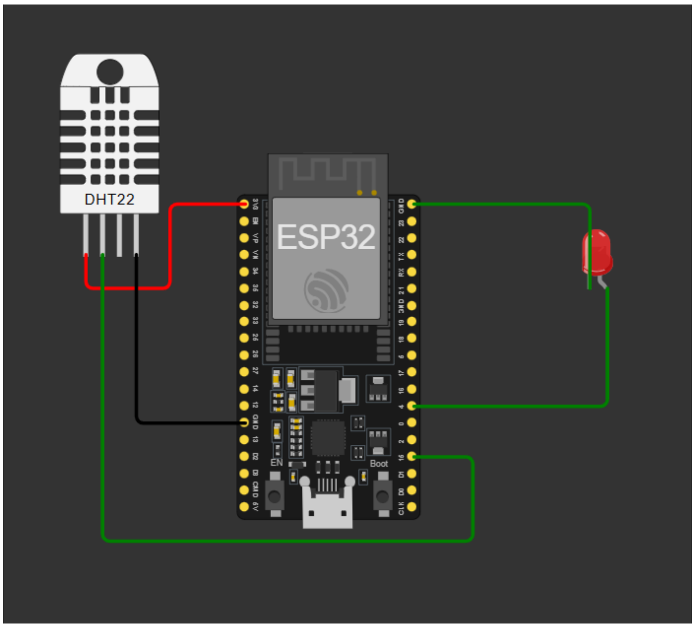
```
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("\nWiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect("ESP32Client-Agent")) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" retry in 5s");
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  dht.setup(DHT_PIN, DHTesp::DHT22);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  TempAndHumidity data = dht.getTempAndHumidity();

  // Embedded agent logic: simple threshold alert
  String status = (data.temperature > 30) ? "Alert" : "Normal";
```

```
  // Application framework: formatting into JSON
  String payload = "{ \"temperature\": " + String(data.temperature, 2) +
                   ", \"humidity\": " + String(data.humidity, 2) +
                   ", \"status\": \"" + status + "\" }";

  Serial.print("Publishing message: ");
  Serial.println(payload);

  client.publish("iot/lab/agents", payload.c_str());
  delay(5000); // send every 5 seconds
}
```

Circuit Connection

Component	ESP32 Pin	Description
DHT22 (VCC)	3.3V	Powers the sensor
DHT22 (GND)	GND	Common ground
DHT22 (DATA)	GPIO 15	Reads temperature & humidity data
WiFi (Wokwi-GUEST)	Virtual	Simulated internet connection
MQTT Broker	test.mosquitto.org	Sends JSON payloads to cloud

Device Setup



Output

Serial Monitor
WiFi connected
IP address: 192.168.1.22
Attempting MQTT connection... connected
Publishing message: { "temperature": 28.40, "humidity": 66.20, "status": "Normal" }

Publishing message: { "temperature": 31.05, "humidity": 65.90, "status": "Alert" }
...
MQTT Subscriber
$ mosquitto_sub -h test.mosquitto.org -t "iot/lab/agents"
{ "temperature": 28.40, "humidity": 66.20, "status": "Normal" }
{ "temperature": 31.05, "humidity": 65.90, "status": "Alert" }

Result

Thus the program has been executed and output verified successfully.

Ex. No. : 8	Working of Raspberry Pi
Date: 15-09-2025	

Aim

To study and understand the working of the Raspberry Pi board and its role in Internet of Things (IoT) applications.

Apparatus Requirement

- Raspberry Pi board (Model 3B/4B or equivalent)
- SD card (8GB or higher, preloaded with Raspberry Pi OS)
- Power supply (5V, 2.5A)
- HDMI cable and monitor
- USB keyboard and mouse
- Wi-Fi/Ethernet connection

Algorithm

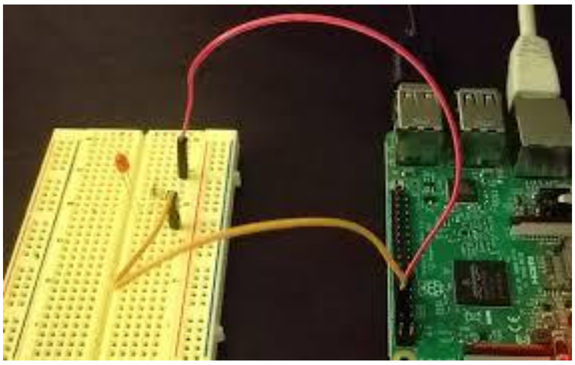
Step 1: Insert the microSD card (with Raspberry Pi OS) into the slot.
Step 2: Connect the HDMI cable to the monitor, keyboard, and mouse to the USB ports.
Step 3: Connect the power supply to boot the Pi.
Step 4: On the first boot, configure language, time zone, and Wi-Fi settings.
Step 5: Open the terminal and run the pinout command to view pin mapping
Step 6: Observe the 40-pin layout consisting of power pins, GPIO pins, and communication interfaces.
Step 7: Connect an LED to GPIO pin 17 through a 330Ω resistor.
Step 8: Open a Python editor.
Step 9: Write and execute the code given below.
Step 10: The LED blinks at 1-second intervals.
Step 11: Stop the program using Ctrl + C.

Source Code

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT)

while True:
    GPIO.output(17, GPIO.HIGH)
    print("LED ON")
    time.sleep(1)
    GPIO.output(17, GPIO.LOW)
    print("LED OFF")
```

```
    time.sleep(1)
```

Device Setup



Result

Thus the program has been executed and output verified successfully.

Ex. No. : 9	Connecting Raspberry Pi with System Components
Date: 22-09-2025	

Aim

To interface the Raspberry Pi with sensors, actuators, and peripheral devices.

Apparatus Requirement

- Raspberry Pi board (Model 3B/4B or equivalent)
- SD card with Raspberry Pi OS
- Power adapter (5V, 2.5A)
- USB keyboard and mouse
- HDMI monitor
- Breadboard and jumper wires
- LED and 330Ω resistor
- DHT11 (Temperature and Humidity) sensor
- Internet/Wi-Fi connection

Algorithm

Step 1: Power up the Raspberry Pi with Raspberry Pi OS.
Step 2: Open the terminal and update packages "sudo apt update" and "sudo apt upgrade"
Step 3: Enable GPIO interface by running "sudo rapsi-config" then select Interface Options → GPIO → Enable.
Step 4: Connect LED anode to GPIO17 through a 330Ω resistor.
Step 5: Connect cathode to GND.
Step 6: From DHT11 Connect VCC to 3.3V
Step 7: From DHT11 DATA to GPIO4
Step 8: From DHT11 GND to GND
Step 9: Install the libraries "sudo apt install python3-pip" and "pip3 install RPi.GPIO Adafruit_DHT"

Source Code

```
import RPi.GPIO as GPIO
import Adafruit_DHT
import time

sensor = Adafruit_DHT.DHT11
DHT_PIN = 4
LED_PIN = 17

GPIO.setmode(GPIO.BCM)
GPIO.setup(LED_PIN, GPIO.OUT)
```

```
while True:
    humidity, temperature = Adafruit_DHT.read_retry(sensor, DHT_PIN)
    if humidity is not None and temperature is not None:
        print(f"Temp={temperature:.1f}°C  Humidity={humidity:.1f}%")
        if temperature > 30:
            GPIO.output(LED_PIN, GPIO.HIGH)
            print("High temperature! LED ON")
        else:
            GPIO.output(LED_PIN, GPIO.LOW)
            print("Normal temperature. LED OFF")
    else:
        print("Sensor failure. Check connection.")
    time.sleep(2)
```

Result

Thus the program has been executed and output verified successfully.

Ex. No. : 10	Overview of Zetta
Date: 22-09-2025	

Introduction

Zetta is an open-source IoT platform built on Node.js for creating server-based Internet of Things (IoT) applications. It is designed to connect devices, sensors, and cloud services seamlessly through APIs and WebSockets. Zetta enables developers to build real-time, reactive systems that can monitor and control connected devices efficiently.

Key Features

1. Each device connected to Zetta is exposed as a RESTful API, making it easy to interact with devices over the web.
2. Uses the asynchronous and event-driven capabilities of Node.js for fast, scalable IoT applications.
3. Allows devices to publish data streams that can be processed or forwarded to cloud platforms like Azure, AWS, or IBM Bluemix.
4. Runs on small computers like Raspberry Pi, as well as on large cloud servers.
5. Supports real-time communication between devices and applications.
6. Provides easy-to-use tools for developers to simulate, monitor, and manage IoT devices via the Zetta Browser interface.

Architecture

Zetta architecture consists of three main layers:

1. Device Layer:
   Physical devices like sensors, actuators, or embedded controllers connected to the system.
2. Zetta Server Layer:
   The Zetta server runs on Node.js and acts as a bridge between devices and the Internet.
   Each device is represented by a “driver” in Zetta, which defines how the device behaves.
3. Cloud and Application Layer:
   Data from devices can be streamed to cloud services or web dashboards for visualization and analytics.

Working Principle

1. Devices are registered in Zetta using drivers.
2. Each driver defines the state and transitions (actions) of the device.
3. The Zetta server exposes devices as RESTful APIs and WebSocket endpoints.
4. Clients (like browsers or mobile apps) communicate with the devices via these APIs.
5. Zetta can connect multiple servers to form a distributed IoT network.

Applications

- Smart home automation
- Environmental monitoring
- Industrial IoT (IIoT) systems
- Connected vehicles
- Remote device management

Conclusion

Zetta provides a powerful yet flexible platform for building IoT systems that connect physical devices to digital applications.

Ex. No. : 11	Home Automation (Level 0)
Date: 29-09-2025	

Aim

To implement a basic Home Automation system (Level 0) using ESP32 to control a home appliance (simulated by an LED).

Algorithm

Step 1: Initialize ESP32 pins:
   i. Configure button pin as input.
   ii. Configure LED pin as output.

Step 2: Read input from the switch.

Step 3: If switch is pressed, turn LED ON (appliance ON).

Step 4: If switch is released, turn LED OFF (appliance OFF).

Step 5: Repeat continuously.

Source Code

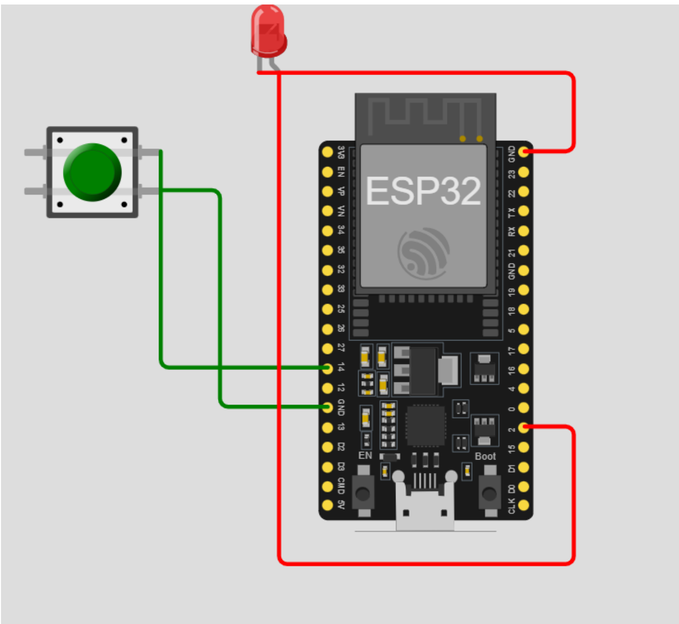
```
#define LED_PIN 2     // Built-in LED on ESP32 (GPIO2)
#define SWITCH_PIN 4  // Button input pin

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(SWITCH_PIN, INPUT_PULLUP); // Button with internal pull-up
  Serial.begin(115200);
}

void loop() {
  int switchState = digitalRead(SWITCH_PIN);

  if (switchState == LOW) {  // Button pressed
    digitalWrite(LED_PIN, HIGH);  // Turn ON LED
    Serial.println("Appliance Status: ON");
  } else {
    digitalWrite(LED_PIN, LOW);   // Turn OFF LED
    Serial.println("Appliance Status: OFF");
  }
  delay(200); // Small delay to debounce
}
```

Circuit Connections:

Component	**Pin/Connection**
ESP32 GPIO2	Connected to built-in LED

Component	Pin/Connection
Pushbutton one terminal	Connected to ESP32 GPIO4
Pushbutton other terminal	Connected to GND
ESP32 GPIO4	Input pin, configured with internal pull-up resistor (no external pull-up resistor required)

Device Setup



Output

Appliance Status: OFF

Appliance Status: ON

Appliance Status: OFF

...

Result

Thus the program has been executed and output verified successfully

Ex. No. : 12	Home Automation (Level 4)
Date: 29-09-2025	

Aim

To implement a Home Automation system (Level 4) using ESP32 controlled remotely via MQTT communication with a cloud broker.

Algorithm

Step 1: Initialize ESP32 with Wi-Fi credentials.
Step 2: Connect to public MQTT broker (test.mosquitto.org).
Step 3: Subscribe to MQTT topic: home/automation/device1.
Step 4: Wait for messages from broker:
  i. If message = "ON", turn LED ON (appliance ON).
  ii. If message = "OFF", turn LED OFF (appliance OFF).
Step 5: Publish status back to topic home/automation/status.
Step 6: Repeat continuously to allow remote control.

Source Code

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

#define LED_PIN 2   // Built-in LED on ESP32

// WiFi credentials
const char* ssid = "Wokwi-GUEST";  // Default WiFi in Wokwi
const char* password = "";

// MQTT broker
const char* mqtt_server = "test.mosquitto.org";
const int mqtt_port = 1883;

WiFiClient espClient;
PubSubClient client(espClient);

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to WiFi...");
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
```

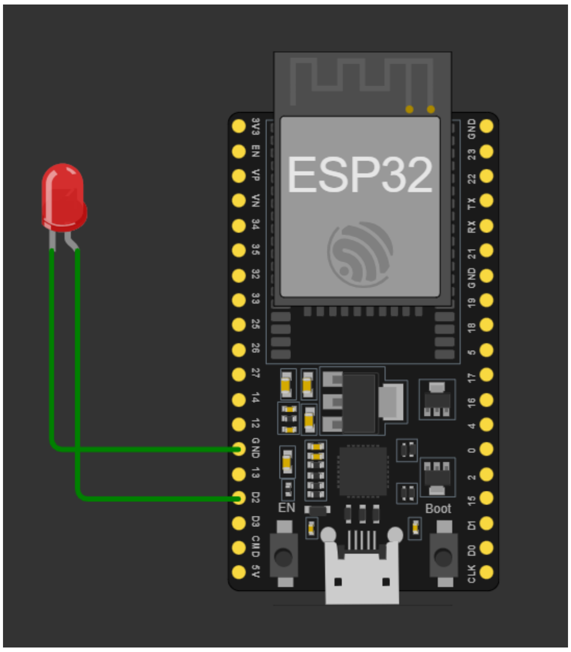
```
    Serial.print(".");
  }
  Serial.println("WiFi connected");
}

void callback(char* topic, byte* message, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  String msg;
  for (int i = 0; i < length; i++) {
    msg += (char)message[i];
  }
  Serial.println(msg);

  if (msg == "ON") {
    digitalWrite(LED_PIN, HIGH);
    Serial.println("Appliance Status: ON");
    client.publish("home/automation/status", "Device is ON");
  } else if (msg == "OFF") {
    digitalWrite(LED_PIN, LOW);
    Serial.println("Appliance Status: OFF");
    client.publish("home/automation/status", "Device is OFF");
  }
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect("ESP32Client-Level4")) {
      Serial.println("connected");
      client.subscribe("home/automation/device1");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
  client.setCallback(callback);
}

void loop() {
```

```
  if (!client.connected()) {
    reconnect();
  }gate
  client.loop();
}
```

Circuit Connections

Component	ESP32 Pin	Description
Built-in LED	GPIO 2	Used to indicate ON/OFF state
WiFi (Wokwi virtual)	—	Connected using “Wokwi-GUEST” network
MQTT Broker	Internet (test.mosquitto.org)	Receives and sends MQTT messages

Device Setup



Output

Serial Monitor
Connecting to WiFi...
WiFi connected
Attempting MQTT connection... connected
Message arrived [home/automation/device1] ON

```
Appliance Status: ON
Message arrived [home/automation/device1] OFF
Appliance Status: OFF

# Turn device ON
mosquitto_pub -h test.mosquitto.org -t "home/automation/device1" -m "ON"

# Turn device OFF
mosquitto_pub -h test.mosquitto.org -t "home/automation/device1" -m "OFF"

# Listen to device status
mosquitto_sub -h test.mosquitto.org -t "home/automation/status"
Device is ON
Device is OFF
```

Result

Thus the program has been executed and output verified successfully

Ex. No. : 13	Smart Irrigation System
Date: 06-10-2025	

Aim

To implement a Smart Irrigation System using ESP32

Algorithm

Step 1: Initialize ESP32 GPIO pins:
   i. Soil moisture sensor → Analog input (A0).
   ii. Pump (LED/relay) → Digital output.

Step 2: Read soil moisture value from sensor.

Step 3: Compare with threshold**:**
   i. If soil is dry (value below threshold) → Turn ON pump.
   ii. If soil is wet (value above threshold) → Turn OFF pump.

Step 4: Display status on Serial Monitor.

Step 5: Repeat continuously.

Source Code

```
#define SOIL_PIN 34   // Soil moisture sensor connected to analog pin
#define PUMP_PIN 2    // LED/relay for pump simulation

int threshold = 2000; // Adjust threshold based on dry/wet values

void setup() {
  Serial.begin(115200);
  pinMode(PUMP_PIN, OUTPUT);
}

void loop() {
  int soilValue = analogRead(SOIL_PIN); // Read soil moisture value
  Serial.print("Soil Moisture Value: ");
  Serial.println(soilValue);

  if (soilValue < threshold) {
    digitalWrite(PUMP_PIN, HIGH);  // Pump ON
    Serial.println("Soil is Dry → Pump ON");
  } else {
    digitalWrite(PUMP_PIN, LOW);   // Pump OFF
    Serial.println("Soil is Wet → Pump OFF");
```

```
  }

  delay(2000); // Check every 2 seconds
}
```

Circuit Connection

Component	ESP32 Pin	Description
Soil Moisture Sensor (A0)	GPIO 34	Reads analog soil moisture value
Soil Moisture Sensor (VCC)	3.3V	Power supply for sensor
Soil Moisture Sensor (GND)	GND	Common ground
LED (Anode +)	GPIO 2	Simulates water pump ON/OFF
LED (Cathode -)	GND (via 220Ω resistor)	Current limiting path

Output

Soil Moisture Value: 1800

Soil is Dry → Pump ON

Soil Moisture Value: 3000

Soil is Wet → Pump OFF

Result:

Thus the program has been executed and output verified successfully.

Ex. No. : 14	Weather Reporting System
Date: 06-10-2025	

Aim

To implement a Weather Reporting System using ESP32 using DHT22 sensor

Algorithm

Step 1: Initialize ESP32 and configure DHT22 sensor.
Step 2: Read temperature and humidity values from the sensor.
Step 3: Check for validity of sensor data.
Step 4: Display readings on Serial Monitor.
Step 5: Repeat periodically (every few seconds).

Source Code

Include Library: DHT Sensor Library for ESPX

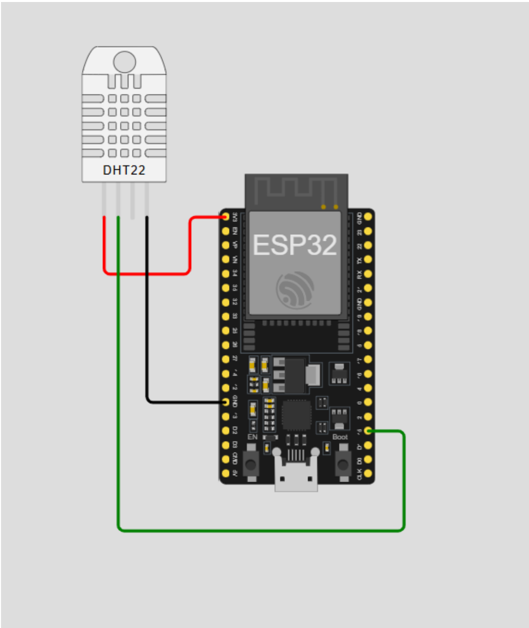
```
#include "DHTesp.h"

#define DHT_PIN 15   // DHT22 data pin

DHTesp dht;

void setup() {
  Serial.begin(115200);
  dht.setup(DHT_PIN, DHTesp::DHT22);
}

void loop() {
  TempAndHumidity data = dht.getTempAndHumidity();

  Serial.print("Temperature: ");
  Serial.print(data.temperature);
  Serial.println(" °C");

  Serial.print("Humidity: ");
  Serial.print(data.humidity);
  Serial.println(" %");

  if (isnan(data.temperature) || isnan(data.humidity)) {
    Serial.println("Failed to read from DHT sensor!");
  }
  Serial.println("-------------------------");
  delay(3000); // update every 3 seconds
```

}

Circuit Connection

DHT22 Pin	Connection
VCC	3.3V
GND	GND
DATA	GPIO 15 (DHT_PIN)
Pull-up resistor	10kΩ between DATA and 3.3V

Device Setup



Output

Temperature: 28.5 °C

Humidity: 66.3 %

Temperature: 29.1 °C

Humidity: 65.8 %

Result:

Thus the program has been executed and output verified successfully.

Ex. No. : 15	Air Pollution Monitoring System
Date: 13-10-2025	

Aim

To implement an Air Pollution Monitoring System using ESP32

Algorithm

Step 1: Initialize ESP32 pins:
   i. Potentiometer (simulated gas sensor) → Analog input (A0).
   ii. LED (optional) → Output to indicate pollution alert.

Step 2: Read analog value from the sensor.

Step 3: Compare with threshold:
   i. If air quality value is high → mark status “Polluted”.
   ii. Else → mark status “Good”.

Step 4: Display readings on Serial Monitor.

Step 5: Repeat continuously.

Source Code

Include Library: MQ2_LPG

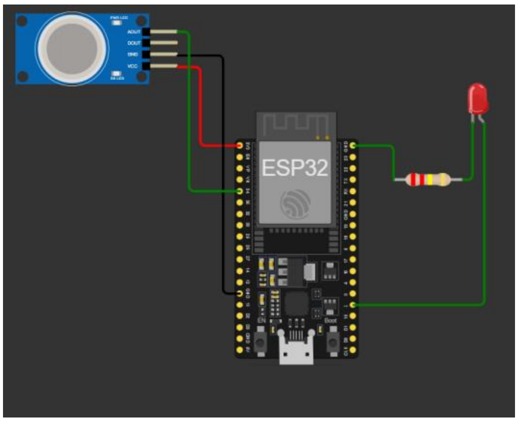
```
#define GAS_SENSOR_PIN 34   // Analog input pin for MQ135 simulation

#define ALERT_LED 2         // Built-in LED for pollution alert


int threshold = 2000; // Adjust threshold (0–4095 on ESP32 ADC)


void setup() {

  Serial.begin(115200);

  pinMode(ALERT_LED, OUTPUT);

}


void loop() {

  int airValue = analogRead(GAS_SENSOR_PIN); // Read air quality value

  Serial.print("Air Quality Value: ");

  Serial.println(airValue);
```

```
  if (airValue > threshold) {
    digitalWrite(ALERT_LED, HIGH);
    Serial.println("Status: Polluted Air");
  } else {
    digitalWrite(ALERT_LED, LOW);
    Serial.println("Status: Good Air");
  }

  Serial.println("--------------------------");
  delay(2000); // Check every 2 seconds
}
```

Circuit Connection

Component	ESP32 Pin	Description
MQ135 VCC	3.3V	Power supply for gas sensor
MQ135 GND	GND	Common ground
MQ135 AOUT (Analog Out)	GPIO 34	Reads analog gas level
LED (Anode +)	GPIO 2	Indicates pollution alert
LED (Cathode -)	GND (through 220Ω resistor)	Current limiting path

Device Setup



Output

Air Quality Value: 1500

Status: Good Air

Air Quality Value: 3200

Status: Polluted Air

Result:

Thus the program has been executed and output verified successfully.