MQQT SLEEP And Awake

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include "esp_sleep.h"

// ------------------- Wi-Fi Credentials -------------------
const char* ssid = "linco";
const char* password = "12345678";

// ------------------- MQTT Broker -------------------
const char* mqtt_server = "broker.hivemq.com";
const int mqtt_port = 1883;

const char* topic_distance = "latrine_monitor/distance";
const char* topic_motion   = "latrine_monitor/motion";
const char* topic_alert    = "latrine_monitor/alert";
const char* topic_status   = "latrine_monitor/status";

// ------------------- Pins -------------------
#define PIR 2
#define ULTRASONIC_PIN 34

// ------------------- Variables -------------------
int pirState = 0;
int distance = 0;

// ------------------- MQTT Client -------------------
```

```
WiFiClient wifiClient;
PubSubClient client(wifiClient);

// ================================================
//               Wi-Fi & MQTT Functions
// ================================================
void setup_wifi() {
  Serial.print("Connecting to Wi-Fi: ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);
  int retry = 0;
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
    if (++retry > 20) {
      Serial.println("\n⚠ Wi-Fi timeout, restarting...");
      ESP.restart();
    }
  }

  Serial.println("\n✓ Wi-Fi connected!");
  Serial.print("IP: ");
  Serial.println(WiFi.localIP());
}

void reconnect_mqtt() {
  while (!client.connected()) {
    Serial.print("Connecting to MQTT...");
```

```
    if (client.connect("ESP32_Latrine_Node")) {
      Serial.println(" connected!");
    } else {
      Serial.print(" failed, rc=");
      Serial.println(client.state());
      delay(2000);
    }
  }
}

// ================================================
//                    SETUP
// ================================================
void setup() {
  Serial.begin(9600);
  delay(500);

  pinMode(PIR, INPUT);
  pinMode(ULTRASONIC_PIN, INPUT);

  Serial.println(" ESP32 Awake: Sampling sensors...");

  // --- Sensor readings ---
  int sensorValue = analogRead(ULTRASONIC_PIN);
  distance = map(sensorValue, 0, 4095, 0, 400);
  pirState = digitalRead(PIR);

  Serial.print("Fill Level: ");
  Serial.print(distance);
```

```
Serial.println(" cm");

Serial.print("Motion: ");
Serial.println(pirState == HIGH ? "Detected" : "None");

// --- Wi-Fi + MQTT transmission cycle ---
setup_wifi();
client.setServer(mqtt_server, mqtt_port);
reconnect_mqtt();

// ---------- Publish Awake Status ----------
String statusPayload = "{\"state\":\"ACTIVE\"}";
client.publish(topic_status, statusPayload.c_str());
Serial.println(" Status Published: " + statusPayload);

// ---------- Publish Distance ----------
String distancePayload = "{\"distance_cm\":" + String(distance) + "}";
client.publish(topic_distance, distancePayload.c_str());
Serial.println(“ Distance Published: " + distancePayload);

// ---------- Publish PIR ----------
String motionPayload = "{\"motion\":\"" + String(pirState == HIGH ? "Detected" : "None") + "\"}";
client.publish(topic_motion, motionPayload.c_str());
Serial.println("Motion Published: " + motionPayload);

// ---------- Alert Condition ----------
String alertPayload;
if (distance < 10) {
  alertPayload = "{\"alert\":\"Latrine Full\"}";
```

```
  } else {
    alertPayload = "{\"alert\":\"None\"}";  // Ensure alert resets
  }
  client.publish(topic_alert, alertPayload.c_str());
  Serial.println(" Alert Published: " + alertPayload);

  // ---------------- Publish Sleeping Status ----------------
  String sleepPayload = "{\"state\":\"SLEEPING\"}";
  client.publish(topic_status, sleepPayload.c_str());
  client.loop();  // ensure MQTT delivers
  delay(200);     // small delay for delivery
  Serial.println(" Status Published: " + sleepPayload);
  Serial.println(" Going to Deep Sleep for 10 seconds...");

  // ---------------- Enter Deep Sleep ----------------
  esp_sleep_enable_timer_wakeup(10 * 1000000ULL);  // 10 seconds
  esp_deep_sleep_start();
}

void loop() {
  // not used – code never reaches here after deep sleep
}
```

MACHINE LEARNING SENDER

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

#define PIR 2
#define ULTRASONIC_PIN 34

BLECharacteristic *sensorCharacteristic;
int pirState = 0;
int distance = 0;

// ========================================================
// EMBEDDED DECISION TREE MODEL
// Predicts 'label' (0, 1, or 2) based on the trained model
// ========================================================
int predict_label(int distance_cm, int pir_state) {
    if (distance_cm <= 14) {
        if (distance_cm <= 10) {
            return 2;
        } else { // distance_cm > 10
            if (pir_state <= 0) {
                return 2;
            } else { // pir_state > 0
                return 1;
            }
        }
```

```
  } else { // distance_cm > 14
    if (pir_state <= 0) {
      return 0;
    } else { // pir_state > 0
      return 1;
    }
  }
}
// ========================================================

void setup() {
 Serial.begin(9600);
 pinMode(PIR, INPUT);
 pinMode(ULTRASONIC_PIN, INPUT);

 BLEDevice::init("NodeA-BLE");
 BLEServer *pServer = BLEDevice::createServer();
 BLEService *pService = pServer->createService("180C");  // Custom service UUID

 sensorCharacteristic = pService->createCharacteristic(
  "2A56",  // Custom characteristic UUID
  BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY
 );

 sensorCharacteristic->addDescriptor(new BLE2902());
 pService->start();
 BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
 pAdvertising->start();
 Serial.println("BLE Node A ready...");
```

```
}

void loop() {
  int sensorValue = analogRead(ULTRASONIC_PIN);
  distance = map(sensorValue, 0, 4095, 0, 400); // Maps analog reading to distance in cm
  pirState = digitalRead(PIR); // pirState is 0 (LOW) or 1 (HIGH)

  // Predict the label using the embedded model
  int predicted_label = predict_label(distance, pirState);

  // Construct the payload with the predicted label
  String payload = String("{\"distance_cm\":") + distance +
                   ",\"pir_state\":" + pirState +
                   ",\"label\":" + predicted_label + "}";

  sensorCharacteristic->setValue(payload.c_str());
  sensorCharacteristic->notify();

  Serial.println(" sending:");
  Serial.println(payload);
  delay(2000);
}
```

Machine learning Reciever

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>
#include <ArduinoJson.h> // Required for parsing the JSON payload

// --- BLE UUIDs and Device Configuration ---
static BLEUUID serviceUUID("180C");          // Service UUID used by the sender
static BLEUUID charUUID("2A56");             // Characteristic UUID used by the sender
static const char* deviceName = "NodeA-BLE";  // Name of the sender device

// --- Global Variables for BLE Client ---
static boolean doConnect = false;
static boolean connected = false;
static BLEAdvertisedDevice* myDevice;
static BLERemoteCharacteristic* pRemoteCharacteristic;

// --- Notification Callback Function ---
// This function is called every time the sender (NodeA-BLE) sends a notification.
void notifyCallback(BLERemoteCharacteristic* pBLERemoteCharacteristic, uint8_t* pData, size_t length, bool isNotify) {
  // Convert the received data (byte array) into a String
  std::string rxValue;
  if (pData != nullptr) {
    rxValue = (char*)pData;
  }
  String payload = String(rxValue.c_str());
```

```
Serial.print("Received Data: ");
Serial.println(payload);

// Parse the JSON payload
// Adjust the capacity based on the size of your JSON document
const size_t capacity = JSON_OBJECT_SIZE(3) + 70;
StaticJsonDocument<capacity> doc;

DeserializationError error = deserializeJson(doc, payload);

if (error) {
  Serial.print("JSON Deserialization failed: ");
  Serial.println(error.f_str());
  return;
}

// Extract values
int distance_cm = doc["distance_cm"];
int pir_state = doc["pir_state"];
int label = doc["label"];

// Print the extracted data
Serial.println("--- Parsed Values ---");
Serial.print("Distance (cm): ");
Serial.println(distance_cm);
Serial.print("PIR State (0/1): ");
Serial.println(pir_state);
Serial.print("Predicted Label: ");
```

```
    Serial.println(label);
    Serial.println("----------------------");

    // TODO: Add logic here to act on the predicted label (0, 1, or 2)
}

// --- Connection Class ---
class MyClientCallback : public BLEClientCallbacks {
    void onConnect(BLEClient* pclient) {
        connected = true;
        Serial.println("Connected to NodeA-BLE!");
    }

    void onDisconnect(BLEClient* pclient) {
        connected = false;
        Serial.println("Disconnected from NodeA-BLE. Starting scan...");
    }
};

// --- Scanner Class ---
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
        // Check if the advertised device name matches our target
        if (advertisedDevice.haveName() && advertisedDevice.getName() == deviceName) {
            Serial.print("Found target device: ");
            Serial.println(deviceName);

            // Stop scanning and save the device reference
            BLEDevice::getScan()->stop();
```

```
        myDevice = new BLEAdvertisedDevice(advertisedDevice);
        doConnect = true; // Flag to initiate connection in loop()
      }
    }
};

// ======================================================
// --- CORRECTED Connection Logic ---
// Fixes the 'no member named subscribe' error.
// ======================================================
bool connectToServer() {
    Serial.print("Attempting to connect to ");
    Serial.println(myDevice->getAddress().toString().c_str());

    BLEClient* pClient = BLEDevice::createClient();
    pClient->setClientCallbacks(new MyClientCallback());

    // Connect to the remote BLE Server (Peripheral)
    if (!pClient->connect(myDevice)) {
        Serial.println("Failed to connect.");
        return false;
    }

    // Get the service and characteristic
    BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
    if (pRemoteService == nullptr) {
        Serial.print("Failed to find service UUID: ");
        Serial.println(serviceUUID.toString().c_str());
        pClient->disconnect();
```

```
    return false;
  }

  pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
  if (pRemoteCharacteristic == nullptr) {
    Serial.print("Failed to find characteristic UUID: ");
    Serial.println(charUUID.toString().c_str());
    pClient->disconnect();
    return false;
  }

  // Check if the characteristic supports notifications
  if (pRemoteCharacteristic->canNotify()) {

    // 1. Set the callback function to handle incoming data
    pRemoteCharacteristic->registerForNotify(notifyCallback);

    // 2. Write 0x01 to the Client Characteristic Configuration Descriptor (CCCD 0x2902)
    // This explicitly enables notifications.
    const uint8_t notifyOn[] = {0x1, 0x0};
    pRemoteCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notifyOn, 2, true);

    Serial.println("Successfully subscribed to notifications.");
  } else {
    Serial.println("Characteristic does not support notifications or is read-only.");
    pClient->disconnect();
    return false;
  }
```

```
    return true;
}
// =====================================================

// --- Setup Function ---
void setup() {
    Serial.begin(115200);
    Serial.println("Starting BLE Client...");

    BLEDevice::init(""); // Initialize the BLE stack

    // Start scanning for the peripheral
    BLEScan* pScan = BLEDevice::getScan();
    pScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pScan->setActiveScan(true); // Active scan uses more power, but gets results faster
    pScan->setInterval(100);
    pScan->setWindow(99);  // Time to scan (in milliseconds)
    pScan->start(5, false); // Start scan for 5 seconds
}

// --- Loop Function ---
void loop() {
    // If we found the device but haven't connected, attempt connection
    if (doConnect) {
        if (connectToServer()) {
            Serial.println("Connection successful.");
            doConnect = false;
        } else {
```

```
        Serial.println("Connection failed, retrying in 5 seconds...");
        delay(5000); // Wait before attempting reconnection
        doConnect = true; // Try again
      }
    } else if (!connected) {
      // Not connected and haven't found a device, restart scan
      Serial.println("Not connected, scanning for devices...");
      BLEDevice::getScan()->start(5, false);
    }

    delay(200); // Standard loop delay
}
```

Machine learning with MQQT

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>
#include <ArduinoJson.h> // Required for parsing the JSON payload

// --- BLE UUIDs and Device Configuration ---
static BLEUUID serviceUUID("180C");          // Service UUID used by the sender
static BLEUUID charUUID("2A56");             // Characteristic UUID used by the sender
static const char* deviceName = "NodeA-BLE";  // Name of the sender device

// --- Global Variables for BLE Client ---
static boolean doConnect = false;
static boolean connected = false;
static BLEAdvertisedDevice* myDevice;
```

```
static BLERemoteCharacteristic* pRemoteCharacteristic;


// --- Notification Callback Function ---
// This function is called every time the sender (NodeA-BLE) sends a notification.
void notifyCallback(BLERemoteCharacteristic* pBLERemoteCharacteristic, uint8_t* pData, size_t length, bool isNotify) {
  // Convert the received data (byte array) into a String
  std::string rxValue;
  if (pData != nullptr) {
    rxValue = (char*)pData;
  }
  String payload = String(rxValue.c_str());

  Serial.print("Received Data: ");
  Serial.println(payload);

  // Parse the JSON payload
  // Adjust the capacity based on the size of your JSON document
  const size_t capacity = JSON_OBJECT_SIZE(3) + 70;
  StaticJsonDocument<capacity> doc;

  DeserializationError error = deserializeJson(doc, payload);

  if (error) {
    Serial.print("JSON Deserialization failed: ");
    Serial.println(error.f_str());
    return;
  }
```

```
    // Extract values
    int distance_cm = doc["distance_cm"];
    int pir_state = doc["pir_state"];
    int label = doc["label"];

    // Print the extracted data
    Serial.println("--- Parsed Values ---");
    Serial.print("Distance (cm): ");
    Serial.println(distance_cm);
    Serial.print("PIR State (0/1): ");
    Serial.println(pir_state);
    Serial.print("Predicted Label: ");
    Serial.println(label);
    Serial.println("----------------------");

    // TODO: Add logic here to act on the predicted label (0, 1, or 2)
}

// --- Connection Class ---
class MyClientCallback : public BLEClientCallbacks {
    void onConnect(BLEClient* pclient) {
        connected = true;
        Serial.println("Connected to NodeA-BLE!");
    }

    void onDisconnect(BLEClient* pclient) {
        connected = false;
        Serial.println("Disconnected from NodeA-BLE. Starting scan...");
    }
```

```
};

// --- Scanner Class ---
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    // Check if the advertised device name matches our target
    if (advertisedDevice.haveName() && advertisedDevice.getName() == deviceName) {
      Serial.print("Found target device: ");
      Serial.println(deviceName);

      // Stop scanning and save the device reference
      BLEDevice::getScan()->stop();
      myDevice = new BLEAdvertisedDevice(advertisedDevice);
      doConnect = true; // Flag to initiate connection in loop()
    }
  }
};

// ======================================================
// --- CORRECTED Connection Logic ---
// Fixes the 'no member named subscribe' error.
// ======================================================
bool connectToServer() {
  Serial.print("Attempting to connect to ");
  Serial.println(myDevice->getAddress().toString().c_str());

  BLEClient* pClient = BLEDevice::createClient();
  pClient->setClientCallbacks(new MyClientCallback());
```

```
// Connect to the remote BLE Server (Peripheral)
if (!pClient->connect(myDevice)) {
   Serial.println("Failed to connect.");
   return false;
}

// Get the service and characteristic
BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
if (pRemoteService == nullptr) {
   Serial.print("Failed to find service UUID: ");
   Serial.println(serviceUUID.toString().c_str());
   pClient->disconnect();
   return false;
}

pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
if (pRemoteCharacteristic == nullptr) {
   Serial.print("Failed to find characteristic UUID: ");
   Serial.println(charUUID.toString().c_str());
   pClient->disconnect();
   return false;
}

// Check if the characteristic supports notifications
if (pRemoteCharacteristic->canNotify()) {

   // 1. Set the callback function to handle incoming data
   pRemoteCharacteristic->registerForNotify(notifyCallback);
```

```
      // 2. Write 0x01 to the Client Characteristic Configuration Descriptor (CCCD 0x2902)
      // This explicitly enables notifications.
      const uint8_t notifyOn[] = {0x1, 0x0};
      pRemoteCharacteristic->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notifyOn, 2, true);

      Serial.println("Successfully subscribed to notifications.");
    } else {
      Serial.println("Characteristic does not support notifications or is read-only.");
      pClient->disconnect();
      return false;
    }

    return true;
}
// ====================================================

// --- Setup Function ---
void setup() {
    Serial.begin(115200);
    Serial.println("Starting BLE Client...");

    BLEDevice::init(""); // Initialize the BLE stack

    // Start scanning for the peripheral
    BLEScan* pScan = BLEDevice::getScan();
    pScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pScan->setActiveScan(true); // Active scan uses more power, but gets results faster
    pScan->setInterval(100);
```

```
    pScan->setWindow(99);  // Time to scan (in milliseconds)
    pScan->start(5, false); // Start scan for 5 seconds
}

// --- Loop Function ---
void loop() {
    // If we found the device but haven't connected, attempt connection
    if (doConnect) {
        if (connectToServer()) {
            Serial.println("Connection successful.");
            doConnect = false;
        } else {
            Serial.println("Connection failed, retrying in 5 seconds...");
            delay(5000); // Wait before attempting reconnection
            doConnect = true; // Try again
        }
    } else if (!connected) {
        // Not connected and haven't found a device, restart scan
        Serial.println("Not connected, scanning for devices...");
        BLEDevice::getScan()->start(5, false);
    }

    delay(200); // Standard loop delay
}
```