# Dog Activity Tracker - Solution

## 1. Introduction

The goal of this project is to develop a Dog Activity Tracker using the ESP32-S3 microcontroller. This tracker uses an accelerometer to monitor a dog's activity, such as walking, running, and resting, and stores the data in LittleFS for efficient management. The device is designed to work autonomously, utilizing deep sleep for power savings. Additionally, the tracker syncs activity data with a mobile app over BLE. We are using a simulated accelerometer for testing, which will later be replaced with the real sensor.

## 2. Simulated Accelerometer (SimulatedQMI8658 Class)

To simulate the behavior of the QMI8658 accelerometer, I created a class that mimics the accelerometer's data output. The class generates random accelerometer and gyroscope values to simulate real-world sensor readings.

**Code:**

```
// Simulated Accelerometer and Gyroscope Data Class

struct IMUdata {
    float x, y, z;
};

class SimulatedQMI8658 {
private:
    bool motionDetected;
    void (*wakeupCallback)();

public:
    SimulatedQMI8658() : motionDetected(false), wakeupCallback(nullptr) {}

    bool begin() {
        // Simulate sensor initialization
        Serial.println("Simulated Sensor Initialized.");
        return true;
    }

    bool readFromFifo(IMUdata* acc, int accCount, IMUdata* gyr, int gyrCount) {
        // Simulate accelerometer and gyroscope data
        for (int i = 0; i < accCount; i++) {
            acc[i].x = random(-32768, 32767) / 1000.0;
            acc[i].y = random(-32768, 32767) / 1000.0;
            acc[i].z = random(-32768, 32767) / 1000.0;
        }
        for (int i = 0; i < gyrCount; i++) {
```

```
      gyr[i].x = random(-32768, 32767) / 100.0;
      gyr[i].y = random(-32768, 32767) / 100.0;
      gyr[i].z = random(-32768, 32767) / 100.0;
    }
    return true;
  }

  void configWakeOnMotion() {
    // Simulate wake-on-motion configuration
  }

  void setWakeupMotionEventCallBack(void (*callback)()) {
    wakeupCallback = callback;
  }

  void simulateMotion() {
    if (random(100) < 5 && !motionDetected) { // 5% chance of motion detection
      motionDetected = true;
      if (wakeupCallback) {
        wakeupCallback(); // Trigger the callback when motion is detected
      }
    } else {
      motionDetected = false;
    }
  }
};
```

### 3. Time Management (TimeManager Class)

To manage time, I used an NTP client to synchronize the device's time with an NTP server when Wi-Fi is available. The TimeManager class updates the time and allows retrieval of the current time, which is used for timestamping activity data.

Code:

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

// Time management class
class TimeManager {
private:
  WiFiUDP ntpUDP;
  NTPClient timeClient;

public:
  TimeManager() : timeClient(ntpUDP, "pool.ntp.org") {}

  void begin() {
    timeClient.begin();
    timeClient.setTimeOffset(0);  // UTC time, adjust if needed
  }

  void update() {
    timeClient.update();
  }

  String getCurrentTime() {
    return timeClient.getFormattedTime(); // Returns time as HH:MM:SS
  }
};
```

**4. Wake-up Mechanism Using Deep Sleep**

The ESP32 enters deep sleep to conserve power when idle. It wakes up either after a set time (every 2 minutes) or when motion is detected by the simulated accelerometer. For this, I used the `esp_sleep` API to manage deep sleep behavior.

```
#include <esp_sleep.h>

void goToSleep() {
  // Put the device into deep sleep mode
  esp_sleep_enable_timer_wakeup(120000000);  // Sleep for 2 minutes
  Serial.println("Going to sleep now...");
  esp_deep_sleep_start();
}

void setup() {
  Serial.begin(115200);
  delay(1000);  // Ensure Serial is ready

  // Start WiFi and NTP Time Sync
  WiFi.begin("your-SSID", "your-PASSWORD");  // Update with your Wi-Fi credentials
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("Connected to WiFi");

  TimeManager timeManager;
  timeManager.begin();
  timeManager.update();
  Serial.println("Current time: " + timeManager.getCurrentTime());

  goToSleep();  // Enter deep sleep
}
```

```
void loop() {
   // This will never run as the ESP32 will be in deep sleep
}
```

**5. Activity Classification**

The ActivityClassifier class classifies activities based on accelerometer data. Using simple thresholds, it determines if the dog is resting, walking, running, or playing based on the z-axis value of the accelerometer.

Code:

```
class ActivityClassifier {
public:
   enum Activity {
      RESTING,
      WALKING,
      RUNNING,
      PLAYING
   };

   Activity classifyActivity(IMUdata* acc) {
      // Simple classification based on the accelerometer's z-axis value
      float zValue = acc[0].z; // Consider only one sample for simplicity

      if (zValue < 0.1) {
         return RESTING;
      } else if (zValue < 0.5) {
         return WALKING;
      } else if (zValue < 1.0) {
         return RUNNING;
      } else {
         return PLAYING;
      }
   }
};
```

**6. Data Storage Using LittleFS**

To store the activity data, I used LittleFS to save data in text files. Each file is named with the current date (`YYYYMMDD.txt`) and contains the activity type and timestamp.

Code:

```
#include <LittleFS.h>

// Function to store activity data to a file
void storeActivityData(String activity, String time) {
    if (!LittleFS.begin()) {
        Serial.println("LittleFS Mount Failed");
        return;
    }

    String filePath = "/data/" + time + ".txt";
    File dataFile = LittleFS.open(filePath, FILE_WRITE);
    if (dataFile) {
        dataFile.println("Activity: " + activity);
        dataFile.close();
        Serial.println("Data written to: " + filePath);
    } else {
        Serial.println("Failed to open file for writing");
    }
}
```

## 7. BLE Communication for Syncing Data

The BLE communication is set up to sync data with a mobile app. The ESP32 advertises itself as a BLE server, allowing a mobile app to connect and retrieve the activity data. BLE characteristics are defined for this purpose.

Code:

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

```
#include <BLEServer.h>

BLECharacteristic *pCharacteristic;

class MyServerCallbacks: public BLEServerCallbacks {
  void onConnect(BLEServer* pServer) {
    Serial.println("Device connected");
  }

  void onDisconnect(BLEServer* pServer) {
    Serial.println("Device disconnected");
  }
};

void setupBLE() {
  BLEDevice::init("Dog Activity Tracker");
  BLEServer *pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());

  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE
  );

  pService->start();
  BLEAdvertising *pAdvertising = pServer->getAdvertising();
  pAdvertising->start();
  Serial.println("Waiting for a client to connect...");
}
```

```
void loop() {
    // BLE communication logic for sending data
}
```

## 8. Main Program for Testing All Features

This is the main program where all features come together. The system simulates motion detection, classifies activity, and stores the data. It also syncs time with NTP, stores data to LittleFS, and communicates with a mobile app over BLE.

Code:

```
SimulatedQMI8658 sensor;
TimeManager timeManager;
ActivityClassifier activityClassifier;

void motionDetectedCallback() {
    Serial.println("Motion Detected! Triggering Callback...");

    // Simulate accelerometer reading
    IMUdata acc[1];
    sensor.readFromFifo(acc, 1, nullptr, 0);

    // Classify the activity
    ActivityClassifier::Activity activity = activityClassifier.classifyActivity(acc);
    String activityStr;
    switch (activity) {
        case ActivityClassifier::RESTING:
            activityStr = "Resting";
            break;
        case ActivityClassifier::WALKING:
            activityStr = "Walking";
            break;
        case ActivityClassifier::RUNNING:
            activityStr = "Running";
            break;
        case ActivityClassifier::PLAYING:
            activityStr = "Playing";
            break;
    }

    // Get current time
    timeManager.update();
    String currentTime = timeManager.getCurrentTime();

    // Store the data
    storeActivityData(activityStr, currentTime);
}

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

    // Initialize components
    sensor.begin();
    sensor.setWakeupMotionEventCallBack(motionDetectedCallback);

    // Initialize TimeManager and sync time
    timeManager.begin();
    timeManager.update();
    Serial.println("Current Time: " + timeManager.getCurrentTime());

    // Setup BLE
    setupBLE();

    // Enter deep sleep to save power
    goToSleep();
```

```
}

void loop() {
   // Simulate motion detection every second
   sensor.simulateMotion();
   delay(1000);  // 1 second delay for the motion simulation loop
}
```