Embedded control with movement detection: Utilizing MPU9250, MQTT, Node-RED, and InfluxDB and machine learning in esp32.

Nikolaos Mouzakitis

May 21, 2025

1 Introduction

This project integrates an ESP32 microcontroller with an MPU9250 inertial measurement unit (IMU) sensor to detect device movements and control 2 LEDs and PWM outputs accordingly. It implements a mechanism to persist and restore device state via MQTT messaging, Node-RED flow processing, and data storage in an InfluxDB time-series database.

The key functionalities are:

- Reading accelerometer and gyroscope Y-axis, and magmetometer X-axis data from the MPU9250.
- Applying threshold-based logic or machine learning to detect movement and determine LED and PWM states.
- Publishing device status data over MQTT.
- Restoring device LED and PWM states after reboot using InfluxDB queries and Node-RED.

An image of the Esp32 and the MPU sensor used in the project is presented below.

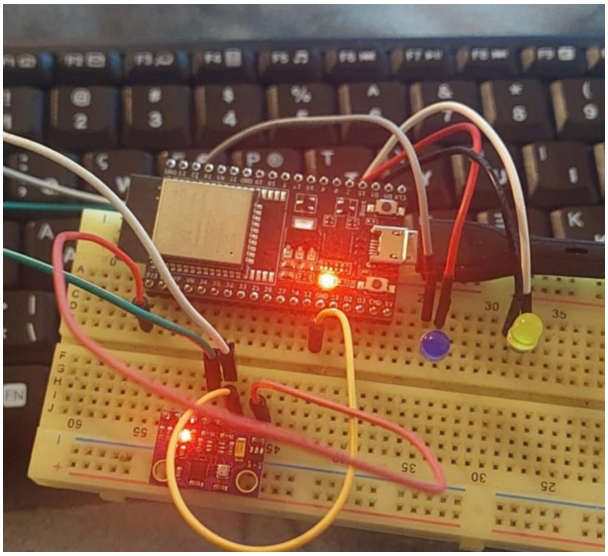

Figure 1: esp32 and mpu connected on breadboard.

2 MPU9250 Sensor Data and Movement Detection

2.1 Sensor Overview

The MPU9250 is a 9-axis IMU sensor combining a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. For this project, only the accelerometer Y-axis,gyroscope Y-axis and magnetometer X-axis values are utilized.

2.2 Data Acquisition

The ESP32 reads raw sensor values for accelerometer Y (`ay`) and gyroscope Y (`gy`) and magnetometer X (`mx`) at periodic intervals.

2.3 Threshold-Based Movement Detection

Threshold values are defined empirically to decide when the ESP32 should change the states of its LEDs and PWM output.

2.4 Controlling LEDs and PWM

Based on movement detection, the device sets the LEDs and PWM.

2.5 Status Publishing

The ESP32 publishes its current LED and PWM states on the MQTT topic `esp32/status` as a formatted string:

```
LED1:OFF LED2:ON PWM:15
```

This message serves to provide the real-time status of device outputs.

3 MQTT Communication Architecture

The ESP32 communicates with the Node-RED server via MQTT. Topics used include:

- `esp32/status` — Periodic status updates including LED and PWM states.
- `esp32/restore_request` — Sent by ESP32 at startup requesting last stored device state.
- `esp32/restore_state` — Published by Node-RED after querying InfluxDB with the restore state data.

3.1 Restore request and response flow

Upon powering on, on startup, the ESP32 immediately publishes a `get_latest` message on `esp32/restore_request`. This triggers Node-RED to query the latest device state stored in InfluxDB and respond by publishing a formatted restore state message on `esp32/restore_state`.

The ESP32 listens for this restore message and applies the LED and PWM settings accordingly before continuing normal operation. Restoration can be observed in the following Figure, where the Esp32 boots and assigns immidiately the latest saved state on the LEDs.

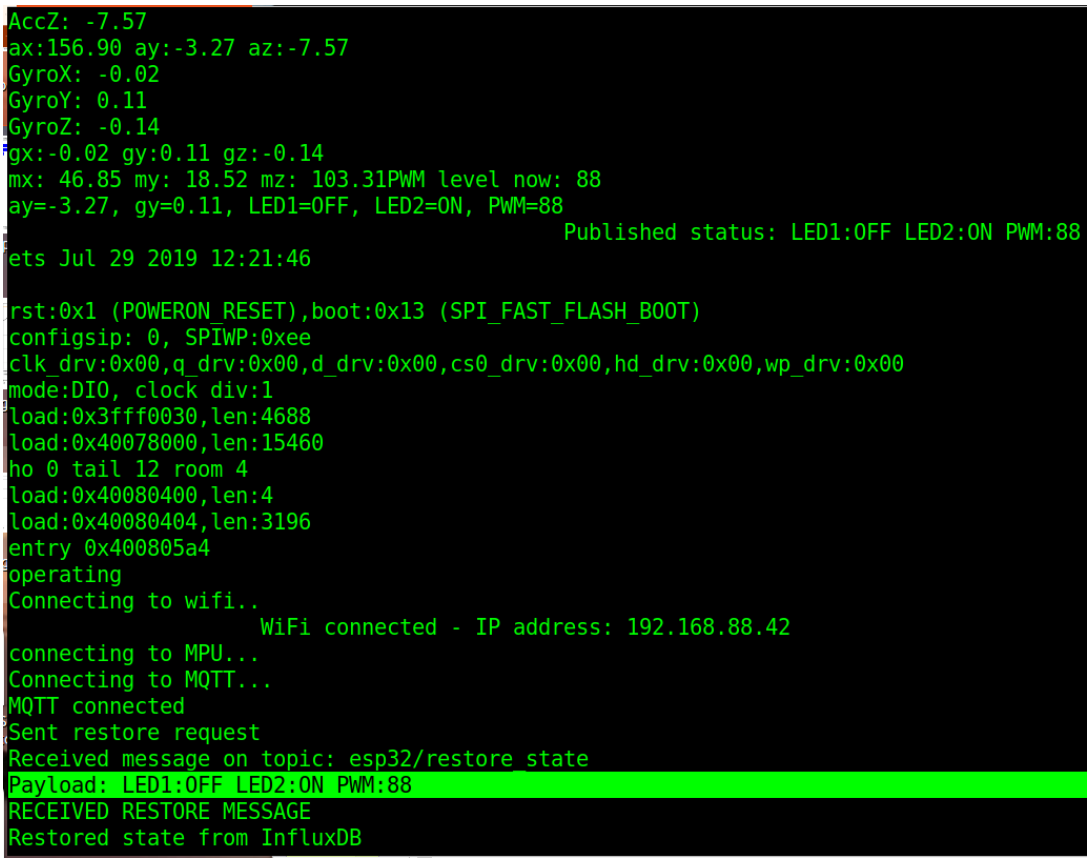

Figure 2: Latest state restoration as observed over UART

4 Node-RED and InfluxDB Integration

4.1 Node-RED Flow Design

Node-RED acts as the middleware for the bridging of MQTT and InfluxDB:

1. **MQTT In node** listens on `esp32/restore_request` and detects the `get_latest` message.
2. **InfluxDB Query node** performs a time-series query fetching the most recent LED and PWM state values.
3. **Function node** formats the InfluxDB query response into the MQTT payload string.
4. **MQTT Out node** publishes the formatted message on `esp32/restore_state`.

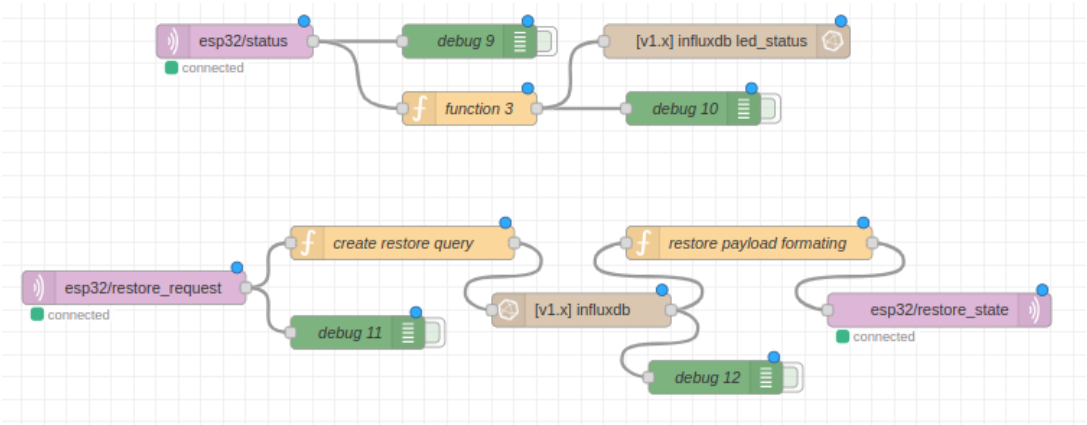

Figure 3: Node-Red data and processing flow

4.2 InfluxDB Query

As for the InfluxDB, version 1.6.4 has been utilized and installed as

```
echo "deb https://repos.influxdata.com/ubuntu bionic stable" |  sudo tee /etc/apt/sources.list.d/influxdb.list
sudo curl -sL https://repos.influxdata.com/influxdb.key | sudo apt-key add -
sudo apt get update
sudo apt-get install -y influxdb
```

After installation, a database is created to be used for this project.

```
influx
> create database esp32db
```

In `function3` of the Node-Red then, when run the following Javascript code in order to push data in the same database.

```
let parts = msg.payload.split(" ");
let led1 = parts[0].split(":")[1] === "ON" ? 1 : 0;
let led2 = parts[1].split(":")[1] === "ON" ? 1 : 0;
let pwm = parseInt(parts[2].split(":")[1]);

msg.payload = {
    LED1: led1,
    LED2: led2,
    PWM: pwm
};
msg.measurement = "led_status";
return msg;
```

On the InfluxDb out node in Node-Red is also specified and set the measurement's name as "led_status" and on the server configuration the respective host ip, port number and database name as "esp32db".

The InfluxDB query used in `create_restore_query` node is:

```
msg.query = 'SELECT LAST("LED1"), LAST("LED2"), LAST("PWM") FROM "led_status"';
return msg;
```

This query retrieves the most recent LED1, LED2, and PWM values stored.

In the `restore payload formating` node of Node-Red response if formated to be given as an input on the `restore_state` topic as:

```
let row = msg.payload[0]; // first row of result

let LED1 = row.last === 1 ? "ON" : "OFF";
let LED2 = row.last_1 === 1 ? "ON" : "OFF";
let PWM = row.last_2;

msg.payload = `LED1:${LED1} LED2:${LED2} PWM:${PWM}`;
msg.topic = "esp32/restore_state";
return msg;
```

4.3 State Restoration

Upon receiving the restore payload, the ESP32 parses the string and applies the saved LED and PWM states, ensuring the desired recovery.

4.4 UI visualization

For the UI visualization 2 switches reflecting the current state of LED have been employed and a slider for representing the common PWM applied to them.

A new function named `function4` have been used to do the same operation as `function3` with the difference of returning3 outputs, one by one to be passed on the switches and the slider.

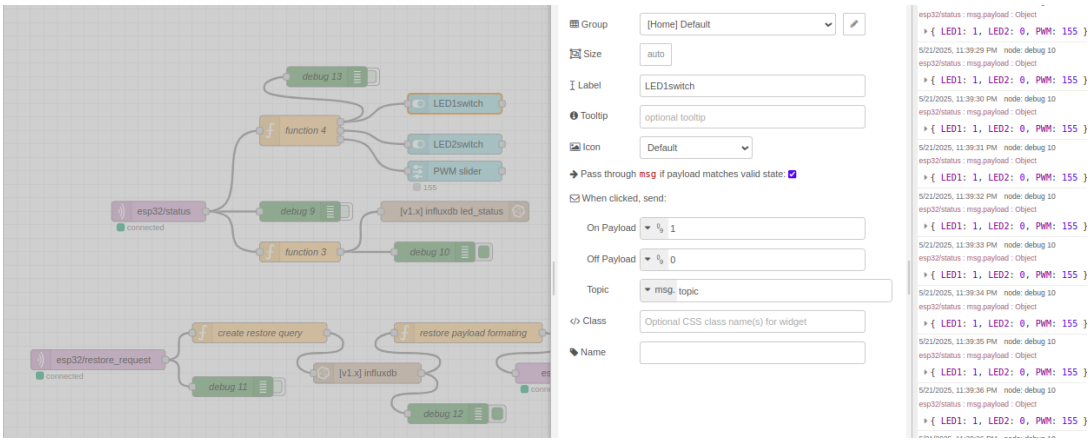

Figure 4: configuration show for slider of led1.

A screenshot of the visualization is presented below, having the 2 switches and the slider, and for debug purposes the plots of the incoming data from accelerometer and gyrometer.

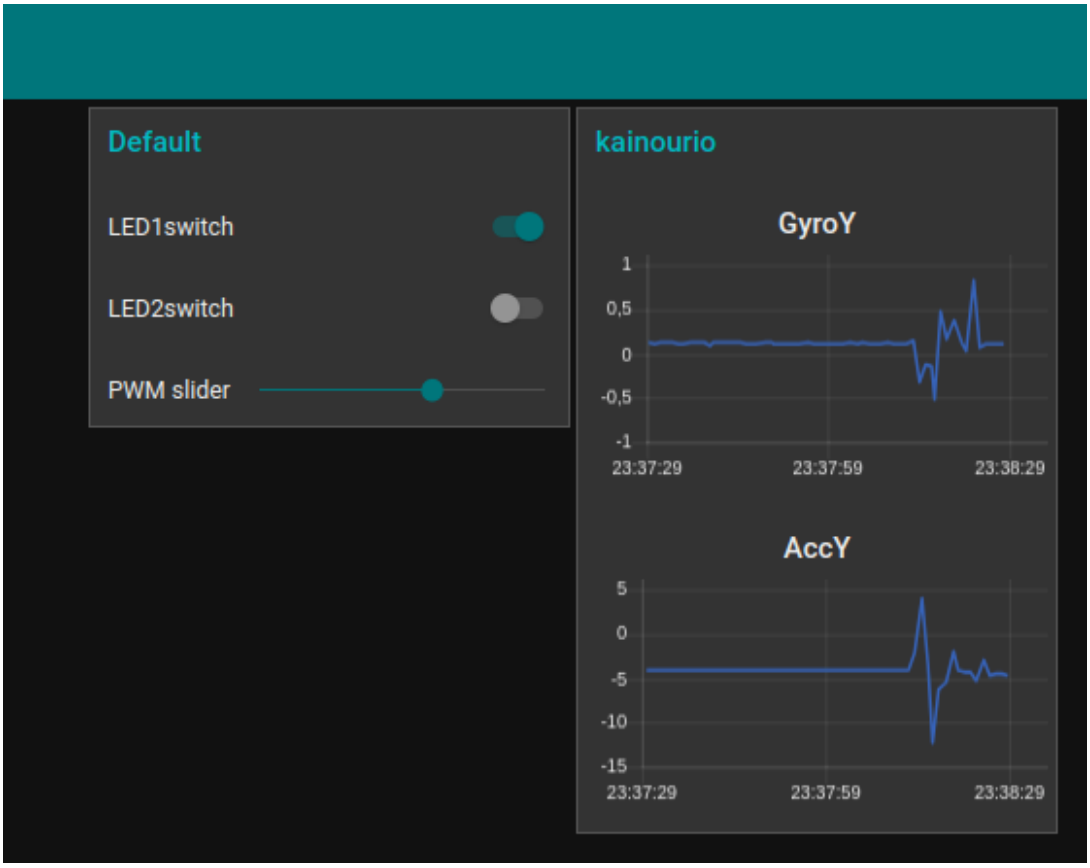

Figure 5: Node-Red UI.

5 Summary of Data Flow

Component	Topic / Action	Description
ESP32	Publish `esp32/restore_request`	Sends `get_latest` on startup
Node-RED	MQTT In on `esp32/restore_request`	Detects restore request
Node-RED	InfluxDB query	Fetch last saved LED/PWM states
Node-RED	Function node	Format restore state payload
Node-RED	MQTT Out on `esp32/restore_state`	Publishes restore state
ESP32	MQTT Subscribe `esp32/restore_state`	Receives and applies restore state
ESP32	Publish `esp32/status`	Periodic status updates

6 Motion detection via machine learning

In the second part, the goal is to move from the threshold supported movement detection and achieve the operational functionalities by employing machine learning. Communication with Node-Red remains the same, as the machine learning classifier is executing only in ESP32 having as a result the usage of the same topics and the same integration across ESP32-NodeRed.

6.1 Data acquisition

For data acquisition for each gesture, `mpu_record.py` have been utilized in order to send over UART a message "record" and then create samples for each of the two classifiers respectively. After manually we create the appropriate files to pass in the Python scripts to generate the C functions to be used in ESP32 and use the trained classifiers.

6.2 LEDs activation

The system in this configuration uses the readings from the MPU9250 IMU and machine learning to control the LEDs. For the data collection and feature processing the IMU continuously reads acceleration (ay) and gyroscope (gy) data along the Y-axis. These values are stored in an implemented circular buffer (featureBuffer)(CircularBuffer.h) that maintains the 20 most recent samples(10 measurements of each Y axis of accelerometer and gyroscope)

When the buffer is full (20 samples), the features are processed for classification. The raw features are first scaled using the StandardScaler parameters which were generated on the `micro2.py` Python training script (scaler_mean and scaler_scale). The system then calculates Euclidean distances between the current scaled features and three reference scaled patterns (LED1, LED2, and Neutral) as a similarity measure. If the distance to the Neutral pattern is below 2.0, our implemented system considers it a neutral position and takes no action

Otherwise if any of the euclidean distances between the current stored sample and the three classes(LED1 TOGGLE,LED2 TOGGLE and NEUTRAL), the features are passed to the SVM classifier (trained in `micro2.py` which predicts whether the motion corresponds to LED1 toggle, LED2 toggle, or Neutral. When a "LED1_TOGGLE" prediction is made, and at least 900ms have passed since the last toggle (debouncing method). The led1State boolean is flipped (true to false or false to true) The PWM duty cycle is set to pwmLevel variable if ON, or 0 if OFF. Same logic as LED1, is followed for controlling "LED2_TOGGLE" prediction.

If confidence is below 75%, the system falls back into using just the distance metrics for classification. As an addition the 900ms delay between toggles is used in order to prevent rapid accidental toggling that can occur inside the same running sliding window as traversing the circular buffer of features. Also the confidence threshold helps for filtering out uncertain predictions.

6.3 Control of PWM

For controlling the common PWM applied in each LED in case it is "ON", we create a second classifier which works with data acquired from magnetometer X axis readings.