Smart Shoes Project Documentation - Meysam Jamali and Mohammad Amin Rezaei

Introduction

This project aims at fabricating a system of “smart shoes” through which several environmental and physical characteristics of the surrounding environment and the wearer can be identified including temperature, humidity, pressure, location, and speed of the smart shoes, heart rate of the wearer, gait of the wearer and the state of the battery of the smart shoes. The information which is being collected with the help of the sensors is sent to the server, namely Ubidots for detection of such states. Also, an Android application is created to depict the real-time data of Monitoring and Control.

Project Overview

Both the smart shoes project and the related research employ one or multiple sensors connected to the ESP32 microcontroller to gather data. The key components of this project include: The key components of this project include:

- **ESP32**: An embedded controller containing a wireless Internet communications module and a Bluetooth module.
- **DHT22**: An environmental sensor for temperature as well as for humidity.
- **MPU6050**: An accelerometer and gyroscopic information detecting device.
- **TinyGPS++**: A library that will support identification of an appropriate library for parsing GPS data.
- **NE06M GPS Module (Simulated)**: A replica of the GPS module was conducted due to the unavailability of the actual sensor, hence.

- **Ubidots**: A cloud based solution that in fact can serve as a data collector, analyzer and in fact a visualization tool as well in relation to web based based IoT data.
- **Android Application**: To monitor the data of the wearer of this smart shoes it is proficient to undertake in real time analyzing.

Components and Libraries

Hardware Components

- **ESP32**: The Chief processing unit.
- **DHT22**: Temperature-humidity sensor.
- **MPU6050**: Detector for movement.
- **Potentiometer**: Evaluates and imitates the pressure sensor.
- **LEDs**: Show different gait status.
- **Custom NE06M GPS Module**: Circuit simulated on Wokwi platform.

Software Libraries

- **DHT. h**: For communicating with the DHT22 sensor.
- **WiFi. h**: It will take the input from the phone and relay the signal back to the host computer for connecting to a WiFi network.
- **PubSubClient. h**: For MQTT communication.
- **TinyGPS++. h**: GPS data parser.
- **Wire. h**: For I2C communication.
- **MPU6050. h**: For handling with the MPU6050 sensor.

Code and Explanation

Library Importation

```
#include <DHT.h> // Include the DHT library for interfacing witl
#include <WiFi.h> // Include the WiFi library for connecting to
```

```
#include <PubSubClient.h> // Include the PubSubClient library fo
#include <TinyGPS++.h> // Include the TinyGPS++ library for pars
#include <Wire.h> // Include the Wire library for I2C communicat
#include <MPU6050.h> // Include the MPU6050 library for interfac
```

- **DHT.h:** For interfacing with the DHT22 temperature and humidity sensor.
- **WiFi.h:** For connecting the ESP32 to a WiFi network.
- **PubSubClient.h:** For MQTT communication with Ubidots.
- **TinyGPS++.h:** For parsing GPS data.
- **Wire.h:** For I2C communication with MPU6050.
- **MPU6050.h:** For interfacing with the MPU6050 sensor.

WiFi and MQTT Setup

```
// WiFi credentials
const char* ssid = "Wokwi-GUEST"; // The SSID (name) of the WiFi
const char* password = ""; // The password for the WiFi network
```

- Define the credentials for WiFi connection (Wokwi default SSID, and password).

```
// Ubidots credentials
const char* mqtt_server = "industrial.api.ubidots.com"; // Ubido
const char* mqtt_client_name = "ESP32Client"; // MQTT client nam
const char* token = "BBUS-2FkzRDZlFJN3RHkJ0VFzH9XqcMEkFH"; // Ub
```

- Define the credentials and API URL address for Ubidots server connection to forward sensors data.

```
WiFiClient espClient; // Create a WiFi client
PubSubClient client(espClient); // Create an MQTT client using t
```

- Create WiFi and MQTT client (WiFi for simulating Internet connection, MQTT for Ubidots connection).

```
// Function to set up the WiFi connection
void setup_wifi() {
  delay(10); // Short delay for stability
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password); // Start the WiFi connection
  while (WiFi.status() != WL_CONNECTED) { // Wait until connecte
    delay(500);
    Serial.print(".");
  }
  Serial.println("\\nWiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP()); // Print the IP address
}
```

- This function sets up the WiFi conenction which conducted by WiFi.begin(ssid, password) code. Along this connection debug messages and delays executed as well.

```
// Function to reconnect to the MQTT server if the connection is
void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(mqtt_client_name, token, "")) { // Try to
      Serial.println("MQTT connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
```

```
    }
  }
}
```

- In order to establish and maintain the MQTT connection to the Ubidots server (due to connection corruption), the reconnect function conducts this purpose by ‘while’ loop to be ensured about the connection has been established otherwise a new one fabricated by utilizing the function ‘client.connect(mqtt_client_name, token, "")’ in ‘if’ condition. Finally, if it fails, it will retry after 5 seconds for the next attempt.

In the following explanation each sensor and functionality has been distincted even for setup and loop functions to obtain more straightforward perspective.

DHT22 Sensor (Temperature and Humidity)

```
const char* variable_label_temp = "temperature"; // Define the t
const char* variable_label_humi = "humidity"; // Define the humi
```

These labels are used to organize and send specific sensor data to the Ubidots cloud service.

- **variable_label_temp:** This label is used to store and forward the temperature data to Ubidots. It helps in tracking the temperature within the shoes.
- **variable_label_humi:** This label is used to store and forward the humidity data to Ubidots. It helps in tracking the humidity within the shoes.

```
#include <DHT.h> // Include the DHT sensor library

#define DHTPIN 15 // Pin connected to the DHT sensor
#define DHTTYPE DHT22 // Define the type of DHT sensor

DHT dht(DHTPIN, DHTTYPE); // Create a DHT sensor object
```

- **DHT22 Initialization**: In this part the DHT22 sensor is setup using the pin and type given above (DHT22), the specfied pin 15 and the type ‘DHT22’; DHT22 sensor is recognized by ‘Wokwi’.

```
void setup() {
  dht.begin(); // Initialize the DHT sensor
}
```

- The begin()` function is used to bring the sensor into the ready state for reading.

```
void loop() {
  float h = dht.readHumidity(); // Read humidity value
  h += 15; // Adjust value for more realistic simulation
  float t = dht.readTemperature(); // Read temperature value
  t += 20; // Adjust value for more realistic simulation

  // Create payloads for temperature and humidity
  char payload_temp[50];
  sprintf(payload_temp, "{\\"temperature\\": %.2f}", t);
  client.publish("/v1.6/devices/smart-shoes", payload_temp); //

  char payload_humi[50];
  sprintf(payload_humi, "{\\"humidity\\": %.2f}", h);
  client.publish("/v1.6/devices/smart-shoes", payload_humi); //

  delay(2000); // Delay for stability
}
```

- The function reads humidity and temperature data from the DHT22 sensor (dht.readHumidity() and dht.readTemperature()). To make the measurements more natural, the humidity value is increased by 10% and the temperature by 5°C due to the air conditions inside the shoes. These adjusted values are then formatted into JSON payloads for publication to Ubidots.

Pressure and Gait Simulation

```
const char* variable_label_pressure = "pressure"; // Define the
const char* variable_label_gait = "status"; // Define the gait v
```

These labels are utilized to organize and send specific sensor data to the Ubidots cloud service:

- **variable_label_pressure:** This label is used to store and forward the pressure data to Ubidots, aiding in monitoring the pressure at the bottom of the shoes.
- **variable_label_gait:** This label is used to store and forward the gait data (wearer's status) to Ubidots, aiding in tracking the gait.

```
int potPin = 34; // Pin for potentiometer (simulated)
int pressureValue; // Variable to store pressure value
int potValue; // Variable to store potentiometer value
```

- Pin 34 is assigned to the potentiometer for pressure simulation (since Wokwi lacks a 'Pressure' sensor). Other variables are defined according to the comments.

```
// LED pins for indicating status
const int walkingLED = 2; // LED for walking status
const int runningLED = 4; // LED for running status
const int stoppedLED = 5; // LED for stopped status
```

- Define LED pins for various statuses: Pins 2, 4, and 5 are assigned to walkingLED, runningLED, and stoppedLED, respectively.

```
// Enumeration for gait types
enum GaitType { STOPPED, WALKING, RUNNING };
GaitType currentGait = STOPPED; // Initialize gait to STOPPED
unsigned long gaitStartTime; // Store the start time of the gait
unsigned long currentTime; // Store the current time
```

```
unsigned long walkDuration = 900; // Duration for walking - Seco
unsigned long runDuration = 300; // Duration for running - Secor
```

- Gait refers to the various statuses of the wearer, including 'STOPPED,' 'WALKING,' and 'RUNNING,' which are defined in the 'GaitType' enum.
- Initially, the wearer's status is set to 'STOPPED' (before walking or running).
- The variable gaitStartTime stores the start time of the gait in the setup function which executed once, while the currentTime variable stores the current time for each iteration in the loop function.

```
void updateGaitStatus() {
  if (currentGait == WALKING) {
    gaitString = "Walking"; // Set gait status to walking
    gaitNumber = 1; // Set gait number to 1 for walking
    digitalWrite(walkingLED, LOW); // Turn on the walking LED
    digitalWrite(runningLED, HIGH); // Turn off the running LED
    digitalWrite(stoppedLED, HIGH); // Turn off the stopped LED
  } else if (currentGait == RUNNING) {
    gaitString = "Running"; // Set gait status to running
    gaitNumber = 2; // Set gait number to 2 for running
    digitalWrite(runningLED, LOW); // Turn on the running LED
    digitalWrite(walkingLED, HIGH); // Turn off the walking LED
    digitalWrite(stoppedLED, HIGH); // Turn off the stopped LED
  } else if (currentGait == STOPPED) {
    gaitString = "Stopped"; // Set gait status to stopped
    gaitNumber = 0; // Set gait number to 0 for stopped
    digitalWrite(stoppedLED, LOW); // Turn on the stopped LED
    digitalWrite(walkingLED, HIGH); // Turn off the walking LED
    digitalWrite(runningLED, HIGH); // Turn off the running LED
  } else {
    gaitString = "Unknown"; // Set gait status to unknown
    gaitNumber = -1; // Set gait number to -1 for unknown status
    digitalWrite(stoppedLED, HIGH); // Turn off all LEDs
    digitalWrite(walkingLED, HIGH);
    digitalWrite(runningLED, HIGH);
```

```
  }
  Serial.println("\nCurrent Gait: " + gaitString); // Print the
}
```

- Depending on the ‘currentGait’ status, the corresponding LED on the breadboard will be turned ‘ON’.
- Gait numbers are as follows:
  - 1 = Walking
  - 2 = Running
  - 0 = Stopped
  - 1 = Unknown
- For each status (gait), the corresponding LED will be ‘ON’ and the others ‘OFF,’ except for the Unknown status, where all LEDs will be ‘ON’.

```
void loop() {
  potValue = analogRead(potPin); // Read potentiometer value
  pressureValue = map(potValue, 0, 4095, 950, 1050); // Map pote

  currentTime = millis(); // Get the current time

  // Update gait status
  if (currentGait == WALKING) {
    if (currentTime - gaitStartTime >= walkDuration) {
      currentGait = RUNNING; // Switch to running gait
      gaitStartTime = currentTime; // Reset gait start time
    } else {
      pressureValue += 8; // Increase pressure slightly when wal
    }
  } else if (currentGait == RUNNING) {
    if (currentTime - gaitStartTime >= runDuration) {
      currentGait = STOPPED; // Switch to stopped gait
      gaitStartTime = currentTime; // Reset gait start time
```

```
    } else {
      pressureValue += 30; // Increase pressure more when runni
    }
  } else if (currentGait == STOPPED) {
    if (currentTime - gaitStartTime >= walkDuration) {
      currentGait = WALKING; // Switch to walking gait
      gaitStartTime = currentTime; // Reset gait start time
    } else {
      pressureValue -= 4; // Decrease pressure when stopped
    }
  }

  // Publish pressure data to Ubidots
  char payload_pressure[100];
  sprintf(payload_pressure, "{\"%s\": %d}", variable_label_press
  client.publish("/v1.6/devices/smart-shoes", payload_pressure)

  updateGaitStatus(); // Update LED status based on gait

  // Publish gait data to Ubidots
  char payload_gait[200];
  sprintf(payload_gait, "{\"%s\": {\"value\": %d, \"context\": 
  client.publish("/v1.6/devices/smart-shoes", payload_gait); //
}
```

- To simulate the pressure of the shoes, a potentiometer has been utilized. The sensor values (0-4095) are mapped to (950-1050) to obtain more realistic readings. Additionally, depending on the wearer's status (gait), the pressureValue is adjusted: +8 for walking, +30 for running, and -4 for stopped.
- The calculation ‘currentTime - gaitStartTime’ is crucial for determining the status (gait) and performing other computations. The variable ‘currentTime’ is updated in each iteration within the loop() function, while the variable ‘gaitStartTime’ is set once in the setup() function.
- The pressure values can be manually adjusted by turning the potentiometer on the breadboard.

- The function updateGaitStatus() updates the LED pins based on the wearer's status (gait).
- Finally, the adjusted pressure values are converted into JSON messages and sent to Ubidots for the specified variable name using the client.publish function.

Heart Rate Simulation

```
const char* variable_label_heart_rate = "heart_rate"; // Defi
ne the heart rate variable label
```

This label is used to organize and send specific sensor data to the Ubidots cloud service:

- **variable_label_heart_rate:** This label is used to store and forward the heart rate data to Ubidots, aiding in monitoring the wearer's health status.

```
void loop() {
  int heartRate; // Variable

 for heart rate
  if (currentGait == WALKING) {
    heartRate = random(80, 100); // Random heart rate for wal
king
  } else if (currentGait == RUNNING) {
    heartRate = random(100, 120); // Random heart rate for ru
nning
  } else {
    heartRate = random(60, 80); // Random heart rate for stop
ped
  }

  // Create payload for heart rate
  char payload_heart_rate[50];
  sprintf(payload_heart_rate, "{\\"heart_rate\\": %d}", heart
Rate);
```

```
  client.publish("/v1.6/devices/smart-shoes", payload_heart_r
ate); // Publish heart rate
}
```

- The variable heartRate stores the simulated heartbeat data. The heart rate values are generated according to the current gait status, producing realistic data within specific ranges for each gait type. For walking, the heart rate ranges from 80 to 100 bpm, for running, it ranges from 100 to 120 bpm, and for stopping, it ranges from 60 to 80 bpm. These values are then organized into a JSON message and sent to Ubidots using the client.publish() function.

Battery Simulation

```
const char* variable_label_battery = "battery"; // Define the
battery variable label
```

This label is used to organize and send specific sensor data to the Ubidots cloud service:

- **variable_label_battery:** This label is used to store and forward the battery level data to Ubidots, aiding in monitoring the battery level of the device.

```
static int batteryLevel = 100; // Initialize battery level
  batteryLevel -= 0.5; // Decrease battery level
  if (batteryLevel <= 0) {
    batteryLevel = 100; // Reset battery level
  }

  // Create payload for battery
  char payload_battery[50];
  sprintf(payload_battery, "{\\"battery\\": %d}", batteryLevel);
  client.publish("/v1.6/devices/smart-shoes", payload_battery);

  delay(2000); // Delay for stability
```

- The battery level decreases by 0.5 percent in each iteration until it reaches 0 percent, at which point it is recharged back to 100 percent, mimicking real-life usage. This battery level is then converted into a JSON payload and sent to Ubidots using the client.publish() function.

GPS Data Simulation (Custom NE06M GPS Module)

gps-neo6m.chip.c

```
#include "wokwi-api.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // For string operations

#define LEN(arr) ((int)(sizeof(arr) / sizeof(arr)[0])) // Mac
ro to get array length
#define SECOND 1000000 // Define one second in microseconds

// Simulated NMEA sentences for GPS data
const char gps_tx_data[][80] = {
  "$GPGGA,172914.049,2327.985,S,05150.410,W,1,12,1.0,0.0,M,0.
0,M,,*60\\r\\n",
  "$GPGSA,A,3,01,02,03,04,05,06,07,08,09,10,11,12,1.0,1.0,1.0
*30\\r\\n",
  "$GPRMC,172914.049,A,2327.985,S,05150.410,W,009.7,025.9,060
622,000.0,W*74\\r\\n",
  // More NMEA sentences...
};

typedef struct {
  uart_dev_t uart0;
  uint32_t gps_tx_index;
} chip_state_t;

static void chip_timer_event(void *user_data);
```

```
void chip_init(void) {
  setvbuf(stdout, NULL, _IOLBF, 1024); // Disable buffering f
or debug logs
  chip_state_t *chip = malloc(sizeof(chip_state_t));

  const uart_config_t uart_config = {
    .tx = pin_init("TX", INPUT_PULLUP),
    .rx = pin_init("RX", INPUT),
    .baud_rate = 9600,
    .user_data = chip,
  };

  chip->uart0 = uart_init(&uart_config);
  chip->gps_tx_index = 0;

  const timer_config_t timer_config = {
    .callback = chip_timer_event,
    .user_data = chip,
  };

  timer_t timer = timer_init(&timer_config);

  timer_start(timer, SECOND, true); // Start the timer

  printf("GPS Chip initialized!\\n");
}

void chip_timer_event(void *user_data) {
  chip_state_t *chip = (chip_state_t*) user_data;

  printf("chip_timer_event\\n");

  const char *message = gps_tx_data[chip->gps_tx_index++]; //
Get current NMEA sentence

  uart_write(chip->uart0, (uint8_t *)message, strlen(messag
```

```
e)); // Write to UART

  if (chip->gps_tx_index >= LEN(gps_tx_data)) {
    chip->gps_tx_index = 0; // Reset index if at the end
  }
}
```

- **NMEA Sentences:** This part describes the simulated GPS data (already generated and stored in the array) in terms of the NMEA sentences format, which is a conventional used in GPS data. These sentences are store in an array.
- **Chip Initialization:** The second function, the chip_init, initializes the UART pins for serial communication and then sets a timer to send out the GPS data.
- **Timer Event Handler:** Finally the chip_timer_event function prints the current NMEA sentence to the UART and the index is set to reflect the next sentence. This produces a cycle of consistent sending of GPS data.

gps-neo6m.chip.json

```
{
  "name": "gps-neo6m",
  "author": "Jamali-Rezaei Sepehr",
  "pins": [
    "VCC",
    "GND",
    "RX",
    "TX"
  ],
  "controls": []
}
```

- **Name and Author**: Gives the name of the custom chip and author.
- **Pins**: Describes the pins used in the design of the developed GPS chip as:

    - VCC: Power pin for the GPS chip:
    - GND: For the connection of GPS chip one ground pin is there for the connection.
    - RX: Get receive pin for serial communication.
    - TX: Transmit pin for serial communication by deselecting it .
- **Controls**: An empty array signifying that there are no other controls for this chip.

Sketch.ino

```
// GPS variables
TinyGPSPlus gps; // Create an instance of the TinyGPS++ library
HardwareSerial serialGPS(1); // Create a hardware serial instanc
double latitude = 0.0; // Declare a variable to store latitude
double longitude = 0.0; // Declare a variable to store longitude
double speed = 0.0; // Declare a variable to store speed
double prev_latitude = 0.0; // For distance calculation
double prev_longitude = 0.0; // For distance calculation
```

- TinyGPSPlus gps creates an instance of the TinyGPS++ library, which is used to parse the GPS data.
- HardwareSerial serialGPS(1) sets up a serial communication channel for the GPS module on hardware serial port 1.
- Variables latitude, longitude, and speed are used to store the current GPS coordinates and speed. prev_latitude and prev_longitude are used to store the previous coordinates for distance calculation.

```
const char* variable_label_date = "date"; // Define the date var
const char* variable_label_time = "time"; // Define the time var
const char* variable_label_accel_x = "acceleration_x"; // Define
const char* variable_label_accel_y = "acceleration_y"; // Define
const char* variable_label_accel_z = "acceleration_z"; // Define
```

```
const char* variable_label_gyro_x = "gyroscope_x"; // Define the
const char* variable_label_gyro_y = "gyroscope_y"; // Define the
const char* variable_label_gyro_z = "gyroscope_z"; // Define the
const char* variable_label_altitude = "altitude"; // Define the
```

These labels are used to organize and send specific sensor data to the Ubidots cloud service.

- **variable_label_date:** This label is used to store and forward the date data to Ubidots. It helps in tracking the exact date when the data was collected.
- **variable_label_time:** This label is used to store and forward the time data to Ubidots. It helps in tracking the exact time when the data was collected.
- **variable_label_accel_x:** This label is for the accelerometer data along the X-axis. It helps in understanding the motion and orientation of the smart shoes in the X direction.
- **variable_label_accel_y:** This label is for the accelerometer data along the Y-axis. It helps in understanding the motion and orientation of the smart shoes in the Y direction.
- **variable_label_accel_z:** This label is for the accelerometer data along the Z-axis. It helps in understanding the motion and orientation of the smart shoes in the Z direction.
- **variable_label_gyro_x:** This label is for the gyroscope data along the X-axis. It helps in understanding the rotational movement of the smart shoes in the X direction.
- **variable_label_gyro_y:** This label is for the gyroscope data along the Y-axis. It helps in understanding the rotational movement of the smart shoes in the Y direction.
- **variable_label_gyro_z:** This label is for the gyroscope data along the Z-axis. It helps in understanding the rotational movement of the smart shoes in the Z direction.
- **variable_label_altitude:** This label is used to store and forward the altitude data to Ubidots. It helps in tracking the elevation level of the smart shoes, which can be useful for activities like hiking or climbing.

```
void displayGPSData() {
  if (gps.location.isValid()) {
    Serial.print(F("Latitude: ")); // Print the latitude label
    Serial.println(gps.location.lat(), 6); // Print the latitude
    Serial.print(F("Longitude: ")); // Print the longitude label
    Serial.println(gps.location.lng(), 6); // Print the longitud

    // Create a JSON payload with the location data
    char payload_location[100];
    sprintf(payload_location, "{\"%s\": {\"value\": 1, \"context
            variable_label_location, gps.location.lat(), gps.loc
    client.publish("/v1.6/devices/smart-shoes", payload_location

  } else {
    Serial.println(F("Location: INVALID")); // Print an error me
  }

  if (gps.speed.isValid()) {
    Serial.print(F("Speed: ")); // Print the speed label
    Serial.println(gps.speed.kmph()); // Print the speed value i
    // Create a JSON payload with the speed data
    char payload_speed[50];
    sprintf(payload_speed, "{\"%s\": %.2f}", variable_label_spee
    client.publish("/v1.6/devices/smart-shoes", payload_speed);
  } else {
    Serial.println(F("Speed: INVALID")); // Print an error messa
  }

  if (gps.date.isValid()) {
    Serial.print(F("Date: ")); // Print the date label
    Serial.print(gps.date.month()); // Print the month
    Serial.print(F("/"));
    Serial.print(gps.date.day()); // Print the day
    Serial.print(F("/"));
    Serial.println(gps.date.year()); // Print the year
```

```
    char payload_date[50];
    sprintf(payload_date, "{\"%s\": \"%02d/%02d/%04d\"}", variab
    client.publish("/v1.6/devices/smart-shoes", payload_date); /
  } else {
    Serial.println(F("Date: INVALID")); // Print an error messag
  }

  if (gps.time.isValid()) {
    Serial.print(F("Time: ")); // Print the time label
    Serial.print(gps.time.hour()); // Print the hour
    Serial.print(F(":"));
    Serial.print(gps.time.minute()); // Print the minute
    Serial.print(F(":"));
    Serial.println(gps.time.second()); // Print the second

    char payload_time[50];
    sprintf(payload_time, "{\"%s\": \"%02d:%02d:%02d\"}", variab
    client.publish("/v1.6/devices/smart-shoes", payload_time); /
  } else {
    Serial.println(F("Time: INVALID")); // Print an error messag
  }
("Time: INVALID")); // Print an error message if the time data i
  }

  // Update distance calculation
  double current_latitude = gps.location.lat();
  double current_longitude = gps.location.lng();
  if (prev_latitude != 0.0 && prev_longitude != 0.0) {
    double distance = haversine(prev_latitude, prev_longitude, c
    total_distance += distance; // Add the calculated distance t
  }
  prev_latitude = current_latitude; // Update previous latitude
  prev_longitude = current_longitude; // Update previous longitu
}
```

Various information extracted from the GPS as follows:

- **Latitude and Longitude**: Checks if the GPS location is valid. If valid, it prints the latitude and longitude to the serial monitor and formats these values into a JSON payload to publish to Ubidots (client.publish).
- **Speed**: Checks if the GPS speed data is valid. If valid, it prints the speed to the serial monitor and formats this value into a JSON payload to publish to Ubidots (client.publish).
- **Date**: Checks if the GPS date is valid. If valid, it prints the date to the serial monitor and formats this value into a JSON payload to publish to Ubidots (client.publish).
- **Time**: Checks if the GPS time is valid. If valid, it prints the time to the serial monitor and formats this value into a JSON payload to publish to Ubidots (client.publish).
- **Distance Calculation**: Uses the Haversine formula to calculate the distance traveled between the current and previous GPS coordinates. If previous coordinates are available, it calculates the distance and adds it to total_distance. The previous coordinates are then updated with the current coordinates.

```
// Read gyroscope and accelerometer data
 int16_t accelX, accelY, accelZ;
 mpu.getAcceleration(&accelX, &accelY, &accelZ);

 int16_t gyroX, gyroY, gyroZ;
 mpu.getRotation(&gyroX, &gyroY, &gyroZ);

 // Format and publish gyroscope data
 char payload_gyro[100];
 sprintf(payload_gyro, "{\"%s\": %d, \"%s\": %d, \"%s\": %d}",
   variable_label_accel_x, accelX,
   variable_label_accel_y, accelY,
   variable_label_accel_z, accelZ);
 client.publish("/v1.6/devices/smart-shoes", payload_gyro);
```

```
  // Format and publish accelerometer data
  char payload_accel[100];
  sprintf(payload_accel, "{\"%s\": %d, \"%s\": %d, \"%s\": %d}",
    variable_label_gyro_x, gyroX,
    variable_label_gyro_y, gyroY,
    variable_label_gyro_z, gyroZ);
  client.publish("/v1.6/devices/smart-shoes", payload_accel);
```

- This section reads data (mpu.getAcceleration(&accelX, &accelY, &accelZ) and mpu.getRotation(&gyroX, &gyroY, &gyroZ)) from the MPU6050 sensor. The accelerometer data is stored in accelX, accelY, and accelZ, while the gyroscope data is stored in gyroX, gyroY, and gyroZ.
- The accelerometer data is formatted into a JSON payload using sprintf and then published to Ubidots using client.publish.
- The gyroscope data is formatted into a JSON payload using sprintf and then published to Ubidots using client.publish.
- These readings can be manually adjusted on the MPU6050 sensor.

```
// Haversine formula to calculate distance
double haversine(double lat1, double lon1, double lat2, doubl
e lon2) {
    // Earth radius in kilometers
    const double R = 6371.0;

    // Convert degrees to radians
    lat1 = radians(lat1); // Convert latitude 1 from degrees
to radians
    lon1 = radians(lon1); // Convert longitude 1 from degrees
to radians
    lat2 = radians(lat2); // Convert latitude 2 from degrees
to radians
    lon2 = radians(lon2); // Convert longitude 2 from degrees
to radians
```

```
    // Differences
    double dlat = lat2 - lat1; // Difference in latitude
    double dlon = lon2 - lon1; // Difference in longitude

    // Haversine formula
    double a = pow(sin(dlat / 2), 2) + cos(lat1) * cos(lat2)
* pow(sin(dlon / 2), 2); // Compute the Haversine formula
    double c = 2 * atan2(sqrt(a), sqrt(1 - a)); // Compute th
e central angle

    // Distance in kilometers
    double distance = R * c; // Calculate the distance

    return distance; // Return the calculated distance
}
```

The Haversine formula is used to calculate the distance between two points on the surface of a sphere given their latitude and longitude. The formula is:

- **Earth Radius**: The Earth's radius is defined as 6371.0 kilometers.
- **Degree to Radian Conversion**: Latitude and longitude values are converted from degrees to radians, which is necessary for the Haversine formula to work correctly.
- **Latitude and Longitude Differences**: The differences between the two latitude and longitude values are calculated (point 1 and point 2).
- **Haversine Formula**: The formula calculates the central angle between two points on a sphere using their latitude and longitude.
  - `a` is the square of half the chord length between the points.
  - `c` is the angular distance in radians.
- **Distance Calculation**: The distance is then calculated by multiplying the Earth's radius by the central angle `c`.
- **Return Value**: The function returns the calculated distance in kilometers.

- **Haversine Formula:**

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)$$

$$c = 2 \cdot \operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

$$d = R \cdot c$$

  - Delta phi = phi_2 - phi_1 (difference in latitude)
  - Delta lambda = lambda_2 - lambda_1 (difference in longitude)
  - R is the Earth's radius (mean radius = 6,371 km)
  - d is the distance between the two points

```
// Read and display GPS data
 while (serialGPS.available() > 0) {
   gps.encode(serialGPS.read()); // Encode GPS data
 }
 displayGPSData(); // Display GPS data
```

- Continuously (while) checks if data is available on the GPS serial port (while condition serialGPS.available() > 0). If data is available, it reads and encodes the GPS data using the gps.encode function from the TinyGPS++ library.
- Calls the displayGPSData function to process and display the GPS data on the serial monitor and publish it to Ubidots (this function already discussed).

Summary

This project integrates the various sensors and modules such that they make up a smart shoe system. Each of them is instrumental in Acquiring data, Emulating realistic measurements and Transmitting this data for analysis and monitoring. About the project, it is essential to integrate it with Ubidots, and the non-standard GPS simulation using Wokwi significantly expands the project’s capabilities.

Ubidots Dashboard and Android Application

Ubidots Dashboard: The actions that Ubidots performs is to make a dashboard to display the data gathered from the smart shoes. The possible widgets for the dashboard are temperature, humidity, pressure, location, speed, heart rate, gait status, and battery level.

Android Application: The Android application designed in this paper is aimed at monitoring the data received from smart shoes in real time. The application's interface and, therefore, how the data is presented to the user are described by screenshots of the application.

Conclusion

The smart shoes project is able to incorporate different sensors and add on modules to measure and transmit vital environmental and physical information. Real data and comfortable custom simulations accompanied by the wide use of data visualization tools make this project a worthy source for getting practical knowledge in the real-data analyzing.