```
// Inclusión de bibliotecas necesarias

#include <WiFi.h>              // Biblioteca para manejar la conexión WiFi del ESP32

#include <PubSubClient.h>      // Biblioteca para comunicación MQTT

#include <Wire.h>              // Biblioteca para comunicación I2C (usada por los MPU6050)

#include <ArduinoJson.h>       // Biblioteca para manejo de datos en formato JSON

#include <math.h>              // Biblioteca para funciones matemáticas


// Credenciales de la red WiFi

const char* ssid = "emcali_609_24_Ghz";    // Nombre de la red WiFi

const char* password = "6023369133pE";      // Contraseña de la red WiFi


// Configuración del broker MQTT (servidor para comunicación de mensajes)

const char* mqtt_server = "broker.emqx.io";   // Dirección del servidor MQTT

const int mqtt_port = 1883;                   // Puerto del servidor MQTT

const char* topic_subscribe = "Unity";          // Tema al que se suscribe para recibir comandos

const char* topic_publish = "ESP32MPU";       // Tema al que publica los datos de los sensores


// Inicialización de los clientes WiFi y MQTT
```

```
WiFiClient esp32Client;                    // Cliente WiFi

PubSubClient client(esp32Client);          // Cliente MQTT que usa el cliente WiFi


// Definición de parámetros para el buffer de datos

#define NUM_DATOS 10       // Número de muestras por paquete a enviar

#define MAX_BUFFER 1400    // Límite total del buffer de almacenamiento

#define NUM_SENSORES 8     // 6 datos de MPU (3 ángulos por cada MPU), 1 de EMG y 1 de tiempo

float dataBuffer[MAX_BUFFER][NUM_SENSORES];  // Array bidimensional para almacenar datos

int bufferIndex = 0;                   // Índice actual en el buffer

bool envioIniciado = false;                // Indica si ya se ha iniciado el envío de datos


// Direcciones I2C de los sensores MPU6050

const int MPU1 = 0x68;     // Dirección del primer MPU (por defecto)

const int MPU2 = 0x69;     // Dirección del segundo MPU (alternativa)


// Constantes de conversión para los sensores MPU6050

const float A_R = 16384.0; // Factor de conversión para acelerómetro configurado a ±2g
```

```
const float G_R = 131.0;    // Factor de conversión para giroscopio configurado a ±250°/s


// Configuración para el sensor EMG

const int pinEMG = 34;     // Pin analógico donde está conectado el sensor EMG

const int samples = 20;    // Número de muestras para promediar (filtro)


// Arrays para el filtrado de EMG mediante promedio móvil

int readings[samples] = {0};    // Almacena las últimas lecturas del EMG

int currentIndex = 0;           // Índice actual en el array circular

int total = 0;                  // Suma total para calcular el promedio

int averageEMG = 0;             // Valor promedio calculado


// Arrays para filtrado de ángulos MPU1 mediante promedio móvil

float pitch1Readings[samples] = {0};  // Últimas lecturas de pitch (cabeceo) para MPU1

float roll1Readings[samples] = {0};   // Últimas lecturas de roll (balanceo) para MPU1

float yaw1Readings[samples] = {0};    // Últimas lecturas de yaw (guiñada) para MPU1

float pitch1Total = 0, roll1Total = 0, yaw1Total = 0;  // Sumas totales para promedios

float avgPitch1 = 0, avgRoll1 = 0, avgYaw1 = 0;        // Valores promediados
```

```
// Arrays para filtrado de ángulos MPU2 mediante promedio móvil
float pitch2Readings[samples] = {0};  // Últimas lecturas de pitch para MPU2
float roll2Readings[samples] = {0};   // Últimas lecturas de roll para MPU2
float yaw2Readings[samples] = {0};    // Últimas lecturas de yaw para MPU2
float pitch2Total = 0, roll2Total = 0, yaw2Total = 0;  // Sumas totales para promedios
float avgPitch2 = 0, avgRoll2 = 0, avgYaw2 = 0;        // Valores promediados

// Variables para almacenar lecturas directas de los sensores
float AcX1, AcY1, AcZ1, GyX1, GyY1, GyZ1;  // Aceleración y velocidad angular MPU1
float AcX2, AcY2, AcZ2, GyX2, GyY2, GyZ2;  // Aceleración y velocidad angular MPU2
float pitch1, roll1, yaw1 = 0;             // Ángulos calculados para MPU1
float pitch2, roll2, yaw2 = 0;             // Ángulos calculados para MPU2
float Gy1[3], Gy2[3];                      // Arrays para giroscopio (no utilizados)

// Offsets para calibración de giroscopios
float gyroX_offset1 = 0, gyroY_offset1 = 0, gyroZ_offset1 = 0;  // Offsets MPU1
float gyroX_offset2 = 0, gyroY_offset2 = 0, gyroZ_offset2 = 0;  // Offsets MPU2
```

```
// Variables para el cálculo del tiempo transcurrido

unsigned long tiempo_prev;     // Tiempo de la última lectura (para calcular dt)

unsigned long tiempo_inicio;   // Tiempo cuando se inició la captura de datos

float dt;                      // Delta de tiempo entre lecturas (para integración)

bool enviarDatos = false;      // Bandera para controlar si se envían datos o no



// Definición de buffer para filtrado adicional de giroscopio

#define FILTER_WINDOW 25       // Tamaño de la ventana de filtrado

// Arrays para filtrado adicional de las lecturas del giroscopio

float gyroXBuffer1[FILTER_WINDOW] = {0}, gyroYBuffer1[FILTER_WINDOW] = {0},
gyroZBuffer1[FILTER_WINDOW] = {0};

float gyroXBuffer2[FILTER_WINDOW] = {0}, gyroYBuffer2[FILTER_WINDOW] = {0},
gyroZBuffer2[FILTER_WINDOW] = {0};

int gyroIndex = 0;             // Índice para los buffers de filtrado



/**

 * Calcula la mediana de un array de valores

 * @param arr Array de valores

 * @param n Tamaño del array

 * @return Valor mediano
```

```
 */
float getMedian(float arr[], int n) {
    // Ordenar array usando bubble sort
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                float temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    // Retornar mediana (valor central del array ordenado)
    return arr[n/2];
}

/**
 * Configura la conexión WiFi
```

```
 */
void setup_wifi() {
    delay(10);
    Serial.println("Conectando a WiFi...");
    WiFi.begin(ssid, password);  // Inicia conexión con credenciales

    // Espera hasta que la conexión sea exitosa
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConectado a WiFi");
}

/**
 * Reconecta al broker MQTT si la conexión se pierde
 */
void reconnect() {
```

```
    while (!client.connected()) {
        Serial.print("Conectando al broker MQTT...");
        if (client.connect("esp32Client")) {  // Intenta conectar con ID "esp32Client"
            Serial.println("Conectado");
            client.subscribe(topic_subscribe);     // Se suscribe al tema para recibir comandos
            Serial.print("Suscrito a: ");
            Serial.println(topic_subscribe);
        } else {
            Serial.print("Fallo, rc=");
            Serial.print(client.state());  // Muestra código de error
            Serial.println(" Intentando de nuevo en 5 segundos");
            delay(5000);  // Espera 5 segundos antes de reintentar
        }
    }
}

/**
 * Función de callback que se ejecuta cuando se recibe un mensaje MQTT
```

```
 */
void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Mensaje recibido [");
    Serial.print(topic);
    Serial.print("] ");

    // Convierte el payload a String
    String message = "";
    for (unsigned int i = 0; i < length; i++) {
        message += (char)payload[i];
    }
    Serial.println(message);

    // Procesa el mensaje recibido
    if (message == "1") {  // Si se recibe "1", inicia captura
        // Calibrar ambos MPUs
        calibrateMPU(MPU1, gyroX_offset1, gyroY_offset1, gyroZ_offset1);
        calibrateMPU(MPU2, gyroX_offset2, gyroY_offset2, gyroZ_offset2);
```

```
enviarDatos = true;  // Activa la bandera para enviar datos

tiempo_inicio = millis();  // Marca el tiempo de inicio

tiempo_prev = micros();    // Inicializa tiempo previo para cálculo de dt



// Reiniciar promedios y totales de los ángulos

pitch1Total = roll1Total = yaw1Total = 0;

pitch2Total = roll2Total = yaw2Total = 0;



// Limpiar arrays de lecturas

for (int i = 0; i < samples; i++) {

    pitch1Readings[i] = pitch2Readings[i] = 0;

    roll1Readings[i] = roll2Readings[i] = 0;

    yaw1Readings[i] = yaw2Readings[i] = 0;

}



// Reiniciar ángulos

pitch1 = roll1 = yaw1 = 0;

pitch2 = roll2 = yaw2 = 0;
```

```
        Serial.println("Calibración completada. Iniciando transmisión...");
        bufferIndex = 0;      // Reinicia el índice del buffer
        envioIniciado = false; // Reinicia flag de envío

        // Nota: hay código duplicado aquí en el original
    } else if (message == "0") {  // Si se recibe "0", detiene captura
        enviarDatos = false;
    }
}

/**
 * Almacena los datos en el buffer y envía cuando hay suficientes
 */
void storeData(float pitch1, float roll1, float yaw1, float pitch2, float roll2, float yaw2,
float emg, float time) {
    // Calcula tiempo transcurrido en segundos desde el inicio
    float tiempo = (millis() - tiempo_inicio) / 1000.0;
```

```
// Almacena datos si hay espacio en el buffer
if (bufferIndex < MAX_BUFFER) {
    dataBuffer[bufferIndex][0] = pitch1;
    dataBuffer[bufferIndex][1] = roll1;
    dataBuffer[bufferIndex][2] = yaw1;
    dataBuffer[bufferIndex][3] = pitch2;
    dataBuffer[bufferIndex][4] = roll2;
    dataBuffer[bufferIndex][5] = yaw2;
    dataBuffer[bufferIndex][6] = emg;
    dataBuffer[bufferIndex][7] = tiempo;
    bufferIndex++;
}

// Marca que se ha iniciado el envío cuando se llega al mínimo de datos
if (!envioIniciado && bufferIndex >= NUM_DATOS) {
    envioIniciado = true;
}

// Envía datos cuando hay suficientes y el envío está iniciado
```

```
    if (envioIniciado && bufferIndex >= NUM_DATOS) {
        sendData();
    }
}

/**
 * Envía los datos almacenados en el buffer a través de MQTT
 */
void sendData() {
    if (client.connected() && bufferIndex >= NUM_DATOS) {
        // Crea documento JSON con espacio para todos los datos
        StaticJsonDocument<4096> jsonDoc;  // Buffer grande para almacenar múltiples registros

        // Crea array anidado para los datos
        JsonArray dataArray = jsonDoc.createNestedArray("data");
        for (int i = 0; i < NUM_DATOS; i++) {
            JsonArray row = dataArray.createNestedArray();
            for (int j = 0; j < NUM_SENSORES; j++) {
```

```
        // Añade cada valor redondeado a 2 decimales
        row.add(roundf(dataBuffer[i][j] * 100) / 100);
    }
}

// Serializa el JSON a string
String jsonString;
serializeJson(jsonDoc, jsonString);

// Publica el mensaje MQTT
if (client.publish(topic_publish, jsonString.c_str())) {
    Serial.println("Datos enviados correctamente.");

    // Reorganiza el buffer moviendo los datos restantes al principio
    int remaining = bufferIndex - NUM_DATOS;
    for (int i = 0; i < remaining; i++) {
        for (int j = 0; j < NUM_SENSORES; j++) {
            dataBuffer[i][j] = dataBuffer[i + NUM_DATOS][j];
```

```
            }
        }

        bufferIndex -= NUM_DATOS; // Reduce el índice según los datos enviados
    } else {
        Serial.println("Error al enviar datos.");
    }
  }
}

/**
 * Configura un sensor MPU6050
 * @param address Dirección I2C del sensor
 */
void setupMPU(int address) {
  Wire.beginTransmission(address);
  Wire.write(0x6B);       // Registro de administración de energía
  Wire.write(0x00);       // Despertar el MPU-6050 (0 = wake up)
  Wire.endTransmission(true);
```

```
delay(100);


// Reset del dispositivo
Wire.beginTransmission(address);
Wire.write(0x6B);       // Registro de administración de energía
Wire.write(0x80);       // Bit 7 = reset
Wire.endTransmission(true);
delay(100);


// Despertar después del reset
Wire.beginTransmission(address);
Wire.write(0x6B);
Wire.write(0x00);       // 0 = wake up
Wire.endTransmission(true);
delay(100);


// Configurar filtro paso bajo digital (DLPF)
Wire.beginTransmission(address);
```

```
  Wire.write(0x1A);       // Registro CONFIG
  Wire.write(0x03);       // Configurar DLPF a 44Hz (reduce ruido)
  Wire.endTransmission(true);

  // Configurar rango del giroscopio (±250°/s)
  Wire.beginTransmission(address);
  Wire.write(0x1B);       // Registro de configuración del giroscopio
  Wire.write(0x00);       // 0 = ±250°/s (mayor sensibilidad)
  Wire.endTransmission(true);

  // Configurar rango del acelerómetro (±2g)
  Wire.beginTransmission(address);
  Wire.write(0x1C);       // Registro de configuración del acelerómetro
  Wire.write(0x00);       // 0 = ±2g (mayor sensibilidad)
  Wire.endTransmission(true);

  delay(100);  // Espera para estabilización
}
```

```
/**
 * Calibra un sensor MPU6050 calculando los offsets del giroscopio
 * @param address Dirección I2C del sensor
 * @param offsetX Referencia para almacenar offset X
 * @param offsetY Referencia para almacenar offset Y
 * @param offsetZ Referencia para almacenar offset Z
 */
void calibrateMPU(int address, float &offsetX, float &offsetY, float &offsetZ) {
    const int samples = 3000;  // Número de muestras para calibración
    float tempReadingsX[100], tempReadingsY[100], tempReadingsZ[100];  // Guarda últimas 100 lecturas
    int tempIndex = 0;

    Serial.print("Calibrando MPU en dirección 0x");
    Serial.println(address, HEX);

    // Primera pasada para recolectar datos
    for (int i = 0; i < samples; i++) {
```

```
Wire.beginTransmission(address);
Wire.write(0x43);  // Registro donde comienzan los datos del giroscopio
Wire.endTransmission(false);
Wire.requestFrom(address, 6, true);  // Solicita 6 bytes (3 ejes x 2 bytes)


// Lee los 3 ejes del giroscopio (cada uno son 2 bytes)
int16_t gx = Wire.read() << 8 | Wire.read();
int16_t gy = Wire.read() << 8 | Wire.read();
int16_t gz = Wire.read() << 8 | Wire.read();


// Almacena solo las últimas 100 lecturas para el cálculo de la mediana
if (i >= samples - 100) {
    tempReadingsX[tempIndex] = gx;
    tempReadingsY[tempIndex] = gy;
    tempReadingsZ[tempIndex] = gz;
    tempIndex++;
}
delay(2);  // Pequeña pausa entre lecturas
```

```
    }


    // Calcula las medianas que serán los offsets
    offsetX = getMedian(tempReadingsX, 100);
    offsetY = getMedian(tempReadingsY, 100);
    offsetZ = getMedian(tempReadingsZ, 100);


    Serial.println("Calibración completada");
    Serial.print("Offsets -> X: ");
    Serial.print(offsetX);
    Serial.print(" Y: ");
    Serial.print(offsetY);
    Serial.print(" Z: ");
    Serial.println(offsetZ);
}

/**
 * Lee los datos del sensor MPU6050
```

```
 * @param address Dirección I2C del sensor

 * @param AcX, AcY, AcZ Referencias para almacenar valores del acelerómetro

 * @param GyX, GyY, GyZ Referencias para almacenar valores del giroscopio

 * @param gyroX_offset, gyroY_offset, gyroZ_offset Offsets del giroscopio

 */

void readMPU(int address, float &AcX, float &AcY, float &AcZ, float &GyX, float &GyY,
float &GyZ,

        float &gyroX_offset, float &gyroY_offset, float &gyroZ_offset) {

   Wire.beginTransmission(address);

   Wire.write(0x3B);  // Registro donde comienzan los datos del acelerómetro

   Wire.endTransmission(false);

   Wire.requestFrom(address, 14, true);    // Solicita 14 bytes (acelerómetro +
temperatura + giroscopio)



   // Lee acelerómetro (3 ejes x 2 bytes)

   int16_t rawAcX = Wire.read() << 8 | Wire.read();  // Combina byte alto y bajo

   int16_t rawAcY = Wire.read() << 8 | Wire.read();

   int16_t rawAcZ = Wire.read() << 8 | Wire.read();
```

```
int16_t temp = Wire.read() << 8 | Wire.read();  // Lee temperatura (no usada)



// Lee giroscopio (3 ejes x 2 bytes)

int16_t rawGyX = Wire.read() << 8 | Wire.read();

int16_t rawGyY = Wire.read() << 8 | Wire.read();

int16_t rawGyZ = Wire.read() << 8 | Wire.read();



// Convierte a unidades físicas

AcX = rawAcX / A_R;  // Convierte a g (aceleración de la gravedad)

AcY = rawAcY / A_R;

AcZ = rawAcZ / A_R;



// Aplica offsets y convierte a grados por segundo

GyX = (rawGyX - gyroX_offset) / G_R;

GyY = (rawGyY - gyroY_offset) / G_R;

GyZ = (rawGyZ - gyroZ_offset) / G_R;



// Aplica umbral para eliminar ruido del giroscopio
```

```
    const float threshold = 0.05;  // Umbral en grados/segundo
    if (abs(GyX) < threshold) GyX = 0;
    if (abs(GyY) < threshold) GyY = 0;
    if (abs(GyZ) < threshold) GyZ = 0;
}

/**
 * Actualiza los promedios de los ángulos utilizando un filtro de promedio móvil
 */
void updateAngleAverages(float newPitch, float newRoll, float newYaw,
                         float *pitchReadings, float *rollReadings, float *yawReadings,
                         float &pitchTotal, float &rollTotal, float &yawTotal,
                         float &avgPitch, float &avgRoll, float &avgYaw) {
    // Resta los valores antiguos del total
    pitchTotal -= pitchReadings[currentIndex];
    rollTotal -= rollReadings[currentIndex];
    yawTotal -= yawReadings[currentIndex];
```

```
    // Agrega los nuevos valores al array
    pitchReadings[currentIndex] = newPitch;
    rollReadings[currentIndex] = newRoll;
    yawReadings[currentIndex] = newYaw;

    // Actualiza totales con los nuevos valores
    pitchTotal += newPitch;
    rollTotal += newRoll;
    yawTotal += newYaw;

    // Calcula promedios dividiendo por el número de muestras
    avgPitch = pitchTotal / samples;
    avgRoll = rollTotal / samples;
    avgYaw = yawTotal / samples;
}

/**
 * Calcula los ángulos de orientación a partir de los datos del acelerómetro y giroscopio
```

```
 */

void computeAngles(float AcX, float AcY, float AcZ, float GyZ, float &pitch, float &roll,
float &yaw) {

    // Calcula pitch y roll usando acelerómetro (método arcotangente)

    // Pitch = ángulo entre el eje Y y el plano horizontal

    pitch = atan2(AcY, sqrt(AcX * AcX + AcZ * AcZ)) * 180.0 / M_PI;

    // Roll = ángulo entre el eje X y el plano horizontal

    roll = atan2(-AcX, AcZ) * 180.0 / M_PI;



    // Actualiza yaw solo si hay movimiento significativo en el giroscopio

    const float yawThreshold = 0.1;

    if (abs(GyZ) > yawThreshold) {

        // Integra la velocidad angular para obtener la posición angular

        yaw += GyZ * dt;

        // Mantiene yaw entre -180 y 180 grados

        if (yaw > 180) yaw -= 360;

        else if (yaw < -180) yaw += 360;

    }
```

```
}

/**
 * Lee el sensor EMG y aplica filtro de promedio móvil
 * @return Valor promediado del EMG
 */
int readEMG() {
    // Resta la lectura más antigua del total
    total -= readings[currentIndex];
    // Lee el valor actual del pin analógico
    readings[currentIndex] = analogRead(pinEMG);
    // Suma la nueva lectura al total
    total += readings[currentIndex];
    // Calcula el promedio
    averageEMG = total / samples;
    // Actualiza el índice circularmente
    currentIndex = (currentIndex + 1) % samples;
    return averageEMG;
```

```
}

/**
 * Función setup() - se ejecuta una vez al iniciar
 */
void setup() {
    Wire.begin();             // Inicia la comunicación I2C
    Serial.begin(115200);     // Inicia la comunicación serial a 115200 baudios
    Serial.println("\nIniciando configuración de MPUs...");

    // Configura los dos sensores MPU
    setupMPU(MPU1);
    setupMPU(MPU2);

    Serial.println("Esperando estabilización de sensores...");
    delay(2000);  // Espera 2 segundos para estabilización

    Serial.println("Iniciando calibración...");
```

```
    Serial.println("Mantenga los sensores inmóviles...");
    Serial.println("Sistema listo!");


    // Configura WiFi y MQTT si no está conectado
    if (WiFi.status() != WL_CONNECTED) {
        setup_wifi();
        client.setServer(mqtt_server, mqtt_port);  // Configura servidor MQTT
        client.setCallback(callback);              // Establece función callback
        client.setBufferSize(4096);                // Establece tamaño del buffer MQTT
        Serial.println("Esperando envío");
    }


    // Muestra calidad de la señal WiFi
    Serial.print("Señal WiFi (RSSI): ");
    Serial.println(WiFi.RSSI());  // RSSI = indicador de fuerza de señal
}

/**
```

```
 * Función loop() - se ejecuta repetidamente

 */

void loop() {

   // Verifica conexión MQTT y reconecta si es necesario

   if (!client.connected()) {

     reconnect();

   }

   client.loop();  // Mantiene la conexión MQTT activa



   // Solo procesa datos si la bandera está activa

   if (enviarDatos) {

      // Calcula el tiempo transcurrido desde la última lectura

      unsigned long tiempo_actual = micros();

      dt = (tiempo_actual - tiempo_prev) / 1.0e6;  // Convierte a segundos

      tiempo_prev = tiempo_actual;



      // Lee datos de ambos sensores MPU

      readMPU(MPU1, AcX1, AcY1, AcZ1, GyX1, GyY1, GyZ1, gyroX_offset1,
gyroY_offset1, gyroZ_offset1);
```

```
readMPU(MPU2, AcX2, AcY2, AcZ2, GyX2, GyY2, GyZ2, gyroX_offset2, gyroY_offset2, gyroZ_offset2);


// Calcula ángulos instantáneos para ambos MPUs

computeAngles(AcX1, AcY1, AcZ1, GyZ1, pitch1, roll1, yaw1);

computeAngles(AcX2, AcY2, AcZ2, GyZ2, pitch2, roll2, yaw2);


// Actualiza promedios de ángulos para MPU1

updateAngleAverages(pitch1, roll1, yaw1,

                    pitch1Readings, roll1Readings, yaw1Readings,

                    pitch1Total, roll1Total, yaw1Total,

                    avgPitch1, avgRoll1, avgYaw1);


// Actualiza promedios de ángulos para MPU2

updateAngleAverages(pitch2, roll2, yaw2,

                    pitch2Readings, roll2Readings, yaw2Readings,

                    pitch2Total, roll2Total, yaw2Total,

                    avgPitch2, avgRoll2, avgYaw2);


// Lee y filtra el valor del sensor EMG
```

```
int emgValue = readEMG();


// Calcula el tiempo transcurrido desde el inicio de la captura

unsigned long tiempo_transcurrido = millis() - tiempo_inicio;


// Crea un documento JSON con los datos actuales para mostrar en Serial

StaticJsonDocument<350> jsonDoc;

jsonDoc["tiempo_ms"] = tiempo_transcurrido;  // Tiempo en milisegundos

jsonDoc["tiempo_s"] = round(tiempo_transcurrido / 1000.0 * 100) / 100.0;   // Tiempo en segundos

jsonDoc["pitch1"] = round(avgPitch1 * 100) / 100.0;  // Redondea a 2 decimales

jsonDoc["roll1"] = round(avgRoll1 * 100) / 100.0;

jsonDoc["yaw1"] = round(avgYaw1 * 100) / 100.0;

jsonDoc["pitch2"] = round(avgPitch2 * 100) / 100.0;

jsonDoc["roll2"] = round(avgRoll2 * 100) / 100.0;

jsonDoc["yaw2"] = round(avgYaw2 * 100) / 100.0;

jsonDoc["EMG"] = emgValue;


// Serializa y muestra el JSON en el monitor serial

String jsonString;
```

```
        serializeJson(jsonDoc, jsonString);

        Serial.println(jsonString);


        // Prepara los datos para enviar al buffer

        float pitchmpu1 = avgPitch1; // Antebrazo

        float rollmpu1 = avgRoll1;   // Antebrazo

        float yawmpu1 = avgYaw1;     // Antebrazo

        float pitchmpu2 = avgPitch2; // Escápula

        float rollmpu2 = avgRoll2;   // Escápula

        float yawmpu2 = avgYaw2;     // Escápula

        float emg = emgValue;              // señal EMG

        float tiempo = round(tiempo_transcurrido / 1000.0 * 100) / 100.0; //Tiempo


        storeData(pitchmpu1, rollmpu1, yawmpu1, pitchmpu2, rollmpu2, yawmpu2, emg,
tiempo); //Vector con 8 datos

        }

        delay(600); // Tiempo de espera para volver a mandar otro vector

    }
```