```
/*
   ESP32 EMG publisher (筋電位データ送信プログラム)

   【主な機能】
   1. Baseline 自動取得: 起動時に「筋肉が動いていない状態」の電圧を測り、基準点にします。
   2. RAW データの送信: 生の波形データをそのまま送信します。
   3. 指標(Metrics)の計算: 平均値(avg)、積分筋電(iEMG)、実効値(RMS)を計算して送信します。
*/

#include <WiFi.h>                  // WiFi 接続用ライブラリ
#include <PubSubClient.h>      // MQTT（データ転送プロトコル）用ライブラリ
#include <ArduinoJson.h>       //  データを JSON 形式に変換するライブラリ
#include <math.h>                  //  数学計算（平方根など）用

// =====  通信設定（環境に合わせて書き換えてください） =====
const char* ssid = "xxxxxxx";                  // WiFi の SSID
const char* pass = "xxxxxxx";                  // WiFi のパスワード
const char* MQTT_HOST = "192.168.50.150"; //  データの送り先サーバの IP
const uint16_t MQTT_PORT = 1883;               // MQTT のポート番号

//  送信先（トピック）の名前
const char* TOPIC_METRICS = "emg/metrics"; //  計算された指標用
const char* TOPIC_RAW         = "emg/raw";         //  生の波形用

//  外部（PC 等）から設定を変更するための受け口（コマンド用トピック）
const char* TOPIC_CMD_WINMS         = "emg/cmd/window_ms";     //  計算区間(ms)の変更
const char* TOPIC_CMD_RAW_EN       = "emg/cmd/raw_enable";    //  生データ送信の ON/OFF
const char* TOPIC_CMD_RAW_CHUNK = "emg/cmd/raw_chunk_ms";//  生データの塊の長さ
const char* TOPIC_CMD_RAW_DECIM = "emg/cmd/raw_decim";     //  間引き率（データ量を減らす）
```

```
// ===== センサー設定 =====
static const int EMG_PIN    = 34;     // センサーをつなぐピン番号
static const int ADC_BITS = 12;     // 電圧読み取りの細かさ（12bit = 0〜4095）

// ===== サンプリング設定 =====
static const int FS = 1000;             // 1 秒間に何回測るか (1000Hz = 1ms ごと)
static const uint32_t SAMPLE_PERIOD_US = 1000000UL / FS; // 1 回あたりの待ち時間(マイクロ秒)

// 設定の限界値（メモリパンクを防ぐため）
static const int WIN_MS_MAX = 1000;
static const int N_MAX = (FS * WIN_MS_MAX) / 1000; // 最大データ保持数
static const int RAW_N_MAX = 200;

// ===== グローバル変数（プログラム全体で使う変数） =====
WiFiClient espClient;
PubSubClient mqtt(espClient);

// baseline: 筋肉が静止している時の電圧値。通常は回路で中間の 1.65V 付近になります。
volatile int adc_baseline = 2048;

// 指標計算用の変数
volatile int window_ms = 100;        // 何ミリ秒分のデータで計算するか
volatile int n_win = 100;               // 計算に使うデータ個数
int16_t buf[N_MAX];                        // データを一時保存する箱
int buf_i = 0;                               // 今、箱の何番目に書き込んでいるか

// 生データ送信用の変数
volatile bool raw_enable = true;   // 生データを送るかどうか
volatile int raw_chunk_ms = 20;     // 何ミリ秒分まとめて送るか
volatile int raw_decim = 2;             // データの「間引き」設定（2 なら 1 個飛ばしで送る）
```

```
int16_t rawBuf[RAW_N_MAX];
int raw_i = 0;
int decim_ctr = 0;

uint32_t last_sample_us = 0;          //  最後に計測した時刻（マイクロ秒）

// =====  補助関数（計算を楽にするもの）  =====
//  値を指定した範囲(lo〜hi)に収める
static int clampi(int v, int lo, int hi) {
   if (v < lo) return lo;
   if (v > hi) return hi;
   return v;
}
//  文字列が一致するかチェックする
static bool streq(const char* a, const char* b) { return strcmp(a,b)==0; }

// =====  基準値(Baseline)の自動調整  =====
//  起動後、数秒間リラックスした状態の平均電圧を「ゼロ点」として記憶します
void calibrateBaseline(uint32_t duration_ms = 3000) {
   Serial.println("[CAL]  基準値を測定中...（リラックスしてください）");
   uint64_t sum = 0;
   uint32_t cnt = 0;
   uint32_t t0 = millis();

   while (millis() - t0 < duration_ms) {
      sum += analogRead(EMG_PIN);
      cnt++;
      delay(1);
   }
   if (cnt > 0) adc_baseline = sum / cnt;
   Serial.printf("[CAL]  基準値確定: %d (約  %.3f V)¥n",
                        adc_baseline, adc_baseline * 3.3 / 4095.0);
```

```
}

// ===== MQTT 受信処理 =====
// 外部（PC など）から「設定を変えて」という命令が届いたときに動きます
void onMqttMessage(char* topic, byte* payload, unsigned int length) {
  char tmp[32];
  unsigned int L = (length < sizeof(tmp)-1) ? length : sizeof(tmp)-1;
  memcpy(tmp, payload, L);
  tmp[L] = '¥0';
  int cmd = atoi(tmp); // 届いた文字を数字に変換

  if (streq(topic, TOPIC_CMD_WINMS)) {
    window_ms = clampi(cmd, 10, WIN_MS_MAX);
    n_win = (FS * window_ms) / 1000;
    buf_i = 0; // 計算設定が変わったら一度リセット
  }
  if (streq(topic, TOPIC_CMD_RAW_EN)) {
    raw_enable = (cmd != 0);
  }
  // ※他の設定(CHUNK, DECIM)も同様に更新
}

// ===== ネットワーク接続処理 =====
void connectWiFi() {
  if (WiFi.status() == WL_CONNECTED) return;
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(300);
  }
}
void connectMQTT() {
```

```
  while (!mqtt.connected()) {
    String cid = "esp32-emg-" + String((uint32_t)ESP.getEfuseMac(), HEX);
    if (mqtt.connect(cid.c_str())) {
      // 接続できたら、コマンドを受け取れるように登録(subscribe)する
      mqtt.subscribe("emg/cmd/#");
    } else delay(1000);
  }
}

// ===== 指標(Metrics)の送信 =====
// 溜まったデータから、筋肉の活動量を計算して送信します
void publishMetrics() {
  double sum_abs = 0.0;
  double sum_sq  = 0.0;

  for (int i = 0; i < n_win; i++) {
    int x = buf[i];
    sum_abs += x;                          // 足し算（積分用）
    sum_sq  += (double)x * (double)x; // 2 乗（RMS 用）
  }

  StaticJsonDocument<256> doc;
  doc["avg"] = sum_abs / n_win;        // 平均
  doc["iemg"] = sum_abs;                  // 積分（合計）
  doc["rms"] = sqrt(sum_sq / n_win);// RMS（実効値：パワーのようなもの）

  char out[256];
  serializeJson(doc, out, sizeof(out));
  mqtt.publish(TOPIC_METRICS, out);
}

// ===== 初期設定 =====
```

```
void setup() {
  Serial.begin(115200);
  analogReadResolution(ADC_BITS); //  読み取り解像度を設定

  // 1. 基準値を取得
  calibrateBaseline(3000);

  // 2. ネットワーク設定
  connectWiFi();
  mqtt.setServer(MQTT_HOST, MQTT_PORT);
  mqtt.setCallback(onMqttMessage);
  mqtt.setBufferSize(8192); //  通信バッファを大きめに確保
  connectMQTT();

  last_sample_us = micros(); //  開始時刻を記録
}

// =====  メインループ（繰り返し実行） =====
void loop() {
  connectWiFi();
  if (!mqtt.connected()) connectMQTT();
  mqtt.loop(); // MQTT の命令チェック

  uint32_t now_us = micros();
  //  指定した周期(1ms ごと)が来たかチェック
  if ((uint32_t)(now_us - last_sample_us) >= SAMPLE_PERIOD_US) {
    last_sample_us += SAMPLE_PERIOD_US;

    // --- 1. センサーから値を読み取る  ---
    int raw_adc = analogRead(EMG_PIN);
```

```
// --- 2. 基準値(中心)を引く ---
// 基準より上ならプラス、下ならマイナスの値になる
int raw_signed = raw_adc - adc_baseline;

// --- 3. 指標計算用のデータ加工 ---
// マイナスの振れも「力が入っている」ことに変わりないので、
// ここでは単純化のため「基準以下の値は 0」として計算用バッファに保存
int x = raw_signed;
if (x < 0) x = 0;
buf[buf_i++] = (int16_t)x;

// 規定の個数が溜まったら計算して送信
if (buf_i >= n_win) {
  buf_i = 0;
  publishMetrics();
}

// --- 4. 生データの送信（オプション） ---
if (raw_enable) {
  decim_ctr++;
  if (decim_ctr >= raw_decim) { // 指定された間隔（間引き）ごとに保存
    decim_ctr = 0;
    rawBuf[raw_i++] = (int16_t)raw_signed;

    // 塊(Chunk)が溜まったら一気に送信
    if (raw_i >= (raw_chunk_ms / raw_decim)) {
      // ※ここで publishRawChunk(raw_i) を呼び出して送信
      raw_i = 0;
    }
  }
}
```

```
    }
}
```