# 一、硬件整体代码

#include <Wire.h>

#include <WiFi.h>

// #include <PubSubClient.h>

// const char \*ssid = "MiHome"; // Enter your WiFi name

// const char \*password = "c107c107";  // Enter WiFi password

// // MQTT Broker

// const char \*mqtt\_broker = "139.9.68.120";

// const char \*sugar\_topic = "send-sugar";

// const char \*mqtt\_username = "";

// const char \*mqtt\_password = "";

// const int mqtt\_port = 1883;

//   // I2C device address - 0x39

// #define \_i2cAddr (0x39)

// WiFiClient espClient;

// PubSubClient client(espClient);

// void callback(char \*topic, byte \*payload, unsigned int length)

// {

//     Serial.print("Message arrived in topic: ");

//     Serial.println(topic);

//     Serial.print("Message:");

//     for (int i = 0; i < length; i++) {

//         Serial.print((char) payload[i]);

//     }

//     Serial.println();

//     Serial.println("-----------------------");

// }

// void setup()

// {

//     // Initiate the Wire library and join the I2C bus as a master or slave

//     Wire.begin(21, 22);

//     // communication with the host computer serial monitor

//     Serial.begin(115200);

//     WiFi.begin(ssid, password);

//     while (WiFi.status() != WL\_CONNECTED)

//     {

//         delay(500);

//         Serial.println("Connecting to WiFi..");

//     }

//     client.setServer(mqtt\_broker, mqtt\_port);

//     client.setCallback(callback);

//     while (!client.connected()) {

//         String client\_id = "esp32-client-";

//         client\_id += String(WiFi.macAddress());

//         Serial.printf("The client %s connects to the public MQTT broker\n", client\_id.c\_str());

//         if (client.connect(client\_id.c\_str(), mqtt\_username, mqtt\_password)) {

//             Serial.println("Public EMQX MQTT broker connected");

//         } else {

//             Serial.print("failed with state ");

//             Serial.print(client.state());

//             delay(2000);

//         }

//     }

//     // Publish and subscribe

//     // client.publish(topic, "Hi, I'm ESP32 ^^");

//     client.subscribe(sugar\_topic);

// }

// void loop()

// {

//     measure\_once();

// }

// void measure\_once()

// {

//   //Sets the Atime for integration time from 0 to 255 in register (0x81), integration time = (ATIME + 1) \* (ASTEP + 1) \* 2.78µS

//     setATIME(byte (0x64));

//     // Sets the Astep for integration time from 0 to 65535 in register (0xCA[7:0]) and (0xCB[15:8]), integration time = (ATIME + 1) \* (ASTEP + 1) \* 2.78µS

//     setASTEP(byte (0xE7), byte (0x03));

//     // Sets the Spectral Gain in CFG1 Register (0xAA) in [4:0] bit

//     setGAIN(byte (0x09));

//     //Function defined to read out channels with SMUX configration 1- F1-F4, Clear, NIR

//     String jsonOutput = ConstructJSON();

//     Serial.println(jsonOutput);  // 打印 JSON 格式数据

//     client.publish(sugar\_topic, jsonOutput.c\_str());

//     delay(1000);

//     //Function detects Flicker for 100 and 120 Hz

//     flickerDetection();

// }

//   /\* ----- Read/Write to i2c register ----- \*/

//     // <summary>

//     //Read a single i2c register

//     // <summary>

//     // param name = "addr">Register address of the the register to be read

//     // param name = "\_i2cAddr">Device address 0x39

//     byte readRegister(byte addr)

//       {

//           Wire.beginTransmission(\_i2cAddr);

//           Wire.write(addr);

//           Wire.endTransmission();

//           Wire.requestFrom(\_i2cAddr, 1);

//           if (Wire.available())

//               {

//                 //Serial.println(Wire.read());

//                 return (Wire.read());

//               }

//           else

//               {

//                 Serial.println("I2C Error");

//                 return (0xFF); //Error

//               }

//       }

//       // <summary>

//     // Read two consecutive i2c registers

//     // <summary>

//     // param name = "addr">First register address of two consecutive registers to be read

//     // param name = "\_i2cAddr">Device address 0x39

//     uint16\_t readTwoRegister1(byte addr)

//         {

//           uint8\_t readingL; uint16\_t readingH; uint16\_t reading = 0;

//           Wire.beginTransmission(\_i2cAddr);

//           Wire.write(addr);

//           Wire.endTransmission();

//           Wire.requestFrom(\_i2cAddr, 2);

//           if (2<=Wire.available())

//             {

//             readingL = Wire.read();

//             readingH = Wire.read();

//             readingH = readingH << 8;

//             reading = (readingH | readingL);

//             return(reading);

//             }

//           else

//             {

//             Serial.println("I2C Error");

//             return (0xFFFF); //Error

//             }

//         }

//     // <summary>

//     // Write a value to a single i2c register

//     // <summary>

//     // param name = "addr">Register address of the the register to the value to be written

//     // param name = "val">The value written to the Register

//     // param name = "\_i2cAddr">Device address 0x39

//     void writeRegister(byte addr, byte val)

//       {

//         Wire.beginTransmission(\_i2cAddr);

//         Wire.write(addr);

//         Wire.write(val);

//         Wire.endTransmission();

//       }

// /\*----- Register configuration  -----\*/

//       // <summary>

//       // Setting the PON (Power on) bit on the chip (bit0 at register ENABLE 0x80)

//       // Attention: This function clears only the PON bit in ENABLE register and keeps the other bits

//       // <summary>

//       void PON()

//         {

//           byte regVal = readRegister(byte(0x80));

//           byte temp = regVal;

//           regVal = regVal & 0xFE;

//           regVal = regVal | 0x01;

//           writeRegister(byte (0x80), byte (regVal));

//         }

//       // <summary>

//       // Write SMUX configration from RAM to set SMUX chain in CFG6 register 0xAF

//       // <summary>

//         void SmuxConfigRAM()

//         {

//           writeRegister(byte (0xAF), byte (0x10));

//         }

//       // <summary>

//       // Setting the SP\_EN (spectral measurement enabled) bit on the chip (bit 1 in register ENABLE)

//       // <summary>

//       // <param name="isEnable">Enabling (true) or disabling (false) the SP\_EN bit</param>

//         void SpEn(bool isEnable)

//         {

//             byte regVal = readRegister(byte(0x80));

//             byte temp = regVal;

//             regVal = regVal & 0xFD;

//             if(isEnable == true)

//               {

//               regVal= regVal | 0x02;

//               }

//             else

//               {

//               regVal = temp & 0xFD;

//               }

//             writeRegister(byte (0x80), byte (regVal));

//         }

//       // <summary>

//       // Starting the SMUX command via enabling the SMUXEN bit (bit 4) in register ENABLE 0x80

//       // The SMUXEN bit gets cleared automatically as soon as SMUX operation is finished

//       // <summary>

//         void SMUXEN()

//       {

//         byte regVal = readRegister(byte(0x80));

//         byte temp = regVal;

//         regVal = regVal & 0xEF;

//         regVal = regVal | 0x10;

//         writeRegister(byte (0x80), byte (regVal));

//       }

//       // <summary>

//       // Reading and Polling the the SMUX Enable bit in Enable Register 0x80

//       // The SMUXEN bit gets cleared automatically as soon as SMUX operation is finished

//       // <summary>

//         bool getSmuxEnabled()

//         {

//           bool isEnabled = false;

//           byte regVal = readRegister(byte(0x80));

//           if( (regVal & 0x10) == 0x10)

//             {

//               return isEnabled = true;

//             }

//           else

//             {

//               return isEnabled = false;

//             }

//           }

//       // <summary>

//       // Reading and Polling the the AVALID bit in Status 2 Register 0xA3,if the spectral measurement is ready or busy.

//       // True indicates that a cycle is completed since the last readout of the Raw Data register

//       // <summary>

//         bool getIsDataReady()

//           {

//             bool isDataReady = false;

//             byte regVal = readRegister(byte(0xA3));

//             if( (regVal & 0x40) == 0x40){

//             return isDataReady = true;

//             }

//             else

//             {

//             return isDataReady = false;

//             }

//           }

//       //<summary>

//       // Reading and polling of Flicker measurement ready bit (bit [5] on FD\_Status register

//       // True indicates that the Flicker Detection measurement was finished

//       //<summary>

//         bool getFdMeasReady()

//           {

//             bool isFdmeasReady = false;

//             byte regVal = readRegister(byte(0xDB));

//             if( (regVal & 0x20) == 0x20)

//               {

//               return isFdmeasReady = true;

//               }

//             else

//               {

//               return isFdmeasReady = false;

//               }

//           }

// /\*----- SMUX Configuration for F1,F2,F3,F4,CLEAR,NIR -----\*/

//       //<summary>

//       // Mapping the individual Photo diodes to dedicated ADCs using SMUX Configuration for F1-F4,Clear,NIR

//       //<summary>

//       void F1F4\_Clear\_NIR()

//       {

//         //SMUX Config for F1,F2,F3,F4,NIR,Clear

//         writeRegister(byte (0x00), byte (0x30)); // F3 left set to ADC2

//         writeRegister(byte (0x01), byte (0x01)); // F1 left set to ADC0

//         writeRegister(byte (0x02), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x03), byte (0x00)); // F8 left disabled

//         writeRegister(byte (0x04), byte (0x00)); // F6 left disabled

//         writeRegister(byte (0x05), byte (0x42)); // F4 left connected to ADC3/f2 left connected to ADC1

//         writeRegister(byte (0x06), byte (0x00)); // F5 left disbled

//         writeRegister(byte (0x07), byte (0x00)); // F7 left disbled

//         writeRegister(byte (0x08), byte (0x50)); // CLEAR connected to ADC4

//         writeRegister(byte (0x09), byte (0x00)); // F5 right disabled

//         writeRegister(byte (0x0A), byte (0x00)); // F7 right disabled

//         writeRegister(byte (0x0B), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x0C), byte (0x20)); // F2 right connected to ADC1

//         writeRegister(byte (0x0D), byte (0x04)); // F4 right connected to ADC3

//         writeRegister(byte (0x0E), byte (0x00)); // F6/F7 right disabled

//         writeRegister(byte (0x0F), byte (0x30)); // F3 right connected to AD2

//         writeRegister(byte (0x10), byte (0x01)); // F1 right connected to AD0

//         writeRegister(byte (0x11), byte (0x50)); // CLEAR right connected to AD4

//         writeRegister(byte (0x12), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x13), byte (0x06)); // NIR connected to ADC5

//       }

// /\*----- SMUX Configuration for F5,F6,F7,F8,CLEAR,NIR -----\*/

//       //<summary>

//       // Mapping the individual Photo diodes to dedicated ADCs using SMUX Configuration for F5-F8,Clear,NIR

//       //<summary>

//       void F5F8\_Clear\_NIR()

//       {

//         //SMUX Config for F5,F6,F7,F8,NIR,Clear

//         writeRegister(byte (0x00), byte (0x00)); // F3 left disable

//         writeRegister(byte (0x01), byte (0x00)); // F1 left disable

//         writeRegister(byte (0x02), byte (0x00)); // reserved/disable

//         writeRegister(byte (0x03), byte (0x40)); // F8 left connected to ADC3

//         writeRegister(byte (0x04), byte (0x02)); // F6 left connected to ADC1

//         writeRegister(byte (0x05), byte (0x00)); // F4/ F2 disabled

//         writeRegister(byte (0x06), byte (0x10)); // F5 left connected to ADC0

//         writeRegister(byte (0x07), byte (0x03)); // F7 left connected to ADC2

//         writeRegister(byte (0x08), byte (0x50)); // CLEAR Connected to ADC4

//         writeRegister(byte (0x09), byte (0x10)); // F5 right connected to ADC0

//         writeRegister(byte (0x0A), byte (0x03)); // F7 right connected to ADC2

//         writeRegister(byte (0x0B), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x0C), byte (0x00)); // F2 right disabled

//         writeRegister(byte (0x0D), byte (0x00)); // F4 right disabled

//         writeRegister(byte (0x0E), byte (0x24)); // F7 connected to ADC2/ F6 connected to ADC1

//         writeRegister(byte (0x0F), byte (0x00)); // F3 right disabled

//         writeRegister(byte (0x10), byte (0x00)); // F1 right disabled

//         writeRegister(byte (0x11), byte (0x50)); // CLEAR right connected to AD4

//         writeRegister(byte (0x12), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x13), byte (0x06)); // NIR connected to ADC5

//       }

// /\*----- //SMUX Configuration for Flicker detection - register (0x13)left set to ADC6 for flicker detection-----\*/

//       //<summary>

//       // Mapping the individual Photo diodes to dedicated ADCs using SMUX Configuration for Flicker detection

//       //<summary>

//       void FDConfig()

//       {

//         //SMUX Config for Flicker- register (0x13)left set to ADC6 for flicker detection

//         writeRegister(byte (0x00), byte (0x00)); // disabled

//         writeRegister(byte (0x01), byte (0x00)); // disabled

//         writeRegister(byte (0x02), byte (0x00)); // reserved/disabled

//         writeRegister(byte (0x03), byte (0x00)); // disabled

//         writeRegister(byte (0x04), byte (0x00)); // disabled

//         writeRegister(byte (0x05), byte (0x00)); // disabled

//         writeRegister(byte (0x06), byte (0x00)); // disabled

//         writeRegister(byte (0x07), byte (0x00)); // disabled

//         writeRegister(byte (0x08), byte (0x00)); // disabled

//         writeRegister(byte (0x09), byte (0x00)); // disabled

//         writeRegister(byte (0x0A), byte (0x00)); // disabled

//         writeRegister(byte (0x0B), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x0C), byte (0x00)); // disabled

//         writeRegister(byte (0x0D), byte (0x00)); // disabled

//         writeRegister(byte (0x0E), byte (0x00)); // disabled

//         writeRegister(byte (0x0F), byte (0x00)); // disabled

//         writeRegister(byte (0x10), byte (0x00)); // disabled

//         writeRegister(byte (0x11), byte (0x00)); // disabled

//         writeRegister(byte (0x12), byte (0x00)); // Reserved or disabled

//         writeRegister(byte (0x13), byte (0x60)); // Flicker connected to ADC5 to left of 0x13

//       }

// /\*----- Set integration time = (ATIME + 1) \* (ASTEP + 1) \* 2.78µS -----\*/

//       //<summary>

//       // Sets the ATIME for integration time from 0 to 255, integration time = (ATIME + 1) \* (ASTEP + 1) \* 2.78µS

//       //<summary>

//       // param name = "value"> integer value from 0 to 255 written to ATIME register 0x81

//       void setATIME(byte value)

//         {

//         writeRegister(byte (0x81), value);

//         }

//       //<summary>

//       // Sets the ASTEP for integration time from 0 to 65535, integration time = (ATIME + 1) \* (ASTEP + 1) \* 2.78µS

//       //<summary>

//       // param name = "value1,"> Defines the lower byte[7:0] of the base step time written to ASTEP register 0xCA

//       // param name = "value2,"> Defines the higher byte[15:8] of the base step time written to ASTEP register 0xCB

//       void setASTEP(byte value1, byte value2)

//         {

//           // astep[7:0]

//           writeRegister(byte (0xCA), value1);

//           // astep[15:8]

//           writeRegister(byte (0xCB), value2);

//         }

//       //<summary>

//       // Sets the Spectral Gain in CFG1 Register (0xAA) in [4:0] bit

//       //<summary>

//       // param name = "value"> integer value from 0 to 10 written to AGAIN register 0xAA

//           void setGAIN(byte value)

//         {

//           writeRegister(byte (0xAA), value);

//         }

// /\*----- Device ID, revision ID and auxiliary ID are read(These function are not implemented in main code. This is to give just an idea regarding these register) -----\*/

//       //<summary>

//       // Reads Identification number register in 0x92

//       //<summary>

//       void readID()

//         {

//           readRegister(byte(0x92));

//         }

//       //<summary>

//       // Reads Revision identification number in 0x91

//       //<summary>

//       void readREVID()

//         {

//           readRegister(byte(0x91));

//         }

//       //<summary>

//       // Reads Auxiliary identification number in 0x93

//       //<summary>

//       void readAUXID()

//         {

//           readRegister(byte(0x93));

//         }

// /\*----- Function to detect Flickering at 100 and 120 Hz(default detection in XWing Sensor) -----\*/

//       //<summary>

//       // Executing a flicker measurement cycle, displaying the status from FD\_Status register

//       //<summary>

//       void flickerDetection()

//         {

//             bool isEnabled = true;

//             bool isFdmeasReady = false;

//             writeRegister(byte(0x80), byte(0x00));

//             // Setting the PON bit in Enable register 0x80

//             PON();

//             // Write SMUX configuration from RAM to set SMUX chain registers (Write 0x10 to CFG6)

//             SmuxConfigRAM();

//             // Write new configuration to all the 20 registers for detecting Flicker

//             FDConfig();

//             // Start SMUX command: Enable the SMUXEN bit (bit 4) in register ENABLE

//             SMUXEN();

//             // Checking on the enabled SMUXEN bit whether back to zero- Poll the SMUXEN bit -> if it is 0 SMUX command is started

//             while(isEnabled)

//               {

//                 isEnabled = getSmuxEnabled();

//               }

//             // Enable SP\_EN bit

//             SpEn(true);

//             /\*----- Functions for setting Flicker Sample, Flicker time, Flicker Gain (not implemented for default flicker detection)------\*/

// //            writeRegister(byte(0xD7), byte(0x21)); //33 default value, function for setting for Fd\_sample and Fd\_compare\_value

// //

// //            writeRegister(byte(0xD8), byte(0x68)); //104 default value, function for setting for Fd\_time lower bit(7:0)

// //

// //            writeRegister(byte(0xDA), byte(0x49)); //73 default value, function for setting for fd\_gain and fd\_time higher bit(10:8)

//             // Function to set the Flicker detection via enabling the fden bit in 0x80 register

//             writeRegister(byte (0x80), byte(0x41));

//             delay(500);

//             // reading the flicker status in FD\_STATUS register 0xDB

//             int flicker\_value = readRegister(byte(0xDB));

//             Serial.print("Flicker value-");

//             Serial.println(flicker\_value);

//             if(flicker\_value == 44)

//             {

//               Serial.println("Unknown frequency");

//             }

//             else if(flicker\_value == 45)

//             {

//               Serial.println("100 Hz detected");

//             }

//             else if(flicker\_value == 46)

//             {

//               Serial.println("120 Hz detected");

//             }

//             else

//             {

//               Serial.println("Error in reading");

//             }

//             // Setting back the PON bit in the ENABLE Register

//             writeRegister(byte (0x80), byte (0x01));

//             Serial.println("");

//         }

// /\*----- Function defined to read out channels with SMUX configration 1 -----\*/

//       //<summary>

//       // Executing raw data measurement cycle for 6 channels F1,F2,F3,F4,NIR,Clear

//       //<summary>

//     uint16\_t\* ReadRawValuesMode1() {

//     static uint16\_t values[6];  // 使用静态数组存储结果，避免返回局部变量指针

//     bool isEnabled = true;

//     bool isDataReady = false;

//     // Setting the PON bit in Enable register 0x80

//     PON();

//     // Write SMUX configuration from RAM to set SMUX chain registers (Write 0x10 to CFG6)

//     SmuxConfigRAM();

//     // Write new configuration to all the 20 registers

//     F1F4\_Clear\_NIR();

//     // Start SMUX command: Enable the SMUXEN bit (bit 4) in register ENABLE

//     SMUXEN();

//     // Checking on the enabled SMUXEN bit whether back to zero - Poll the SMUXEN bit -> if it is 0 SMUX command is started

//     while (isEnabled) {

//         isEnabled = getSmuxEnabled();

//     }

//     // Enable SP\_EN bit

//     SpEn(true);

//     // Reading and Polling the AVALID bit in Status 2 Register 0xA3

//     while (!isDataReady) {

//         isDataReady = getIsDataReady();

//     }

//     // 读取 6 个通道数据

//     values[0] = readTwoRegister1(0x95);  // ADC0/F1

//     values[1] = readTwoRegister1(0x97);  // ADC1/F2

//     values[2] = readTwoRegister1(0x99);  // ADC2/F3

//     values[3] = readTwoRegister1(0x9B);  // ADC3/F4

//     values[4] = readTwoRegister1(0x9D);  // ADC4/Clear

//     values[5] = readTwoRegister1(0x9F);  // ADC5/NIR

//     return values;  // 返回数组指针

// }

// /\*----- Function defined to read out channels with SMUX configration 2 -----\*/

//       //<summary>

//       // Executing raw data measurement cycle for 6 channels F1,F2,F3,F4,NIR,Clear

//       //<summary>

//     uint16\_t\* ReadRawValuesMode2()

//     {

//     static uint16\_t values[6];

//     bool isEnabled = true;

//     bool isDataReady = false;

//     PON();

//     SpEn(false);

//     SmuxConfigRAM();

//     F5F8\_Clear\_NIR();

//     SMUXEN();

//     while (isEnabled) {

//         isEnabled = getSmuxEnabled();

//     }

//     SpEn(true);

//     while (!isDataReady) {

//         isDataReady = getIsDataReady();

//     }

//     values[0] = readTwoRegister1(0x95);  // F5

//     values[1] = readTwoRegister1(0x97);  // F6

//     values[2] = readTwoRegister1(0x99);  // F7

//     values[3] = readTwoRegister1(0x9B);  // F8

//     values[4] = readTwoRegister1(0x9D);  // Clear

//     values[5] = readTwoRegister1(0x9F);  // NIR

//     return values;

// }

// // 构建 JSON 字符串

// String ConstructJSON() {

//     uint16\_t\* values1 = ReadRawValuesMode1();

//     uint16\_t\* values2 = ReadRawValuesMode2();

//     String json = "{";

//     json += "\"F1\": " + String(values1[0]) + ", ";

//     json += "\"F2\": " + String(values1[1]) + ", ";

//     json += "\"F3\": " + String(values1[2]) + ", ";

//     json += "\"F4\": " + String(values1[3]) + ", ";

//     json += "\"F5\": " + String(values2[0]) + ", ";

//     json += "\"F6\": " + String(values2[1]) + ", ";

//     json += "\"F7\": " + String(values2[2]) + ", ";

//     json += "\"F8\": " + String(values2[3]) + ", ";

//     json += "\"Clear1\": " + String(values1[4]) + ", ";

//     json += "\"NIR1\": " + String(values1[5]) + ", ";

//     json += "\"Clear2\": " + String(values2[4]) + ", ";

//     json += "\"NIR2\": " + String(values2[5]);

//     json += "}";

//     return json;

// }

#include <Wire.h>

#include <WiFi.h>

#include <PubSubClient.h>

#include <Ticker.h>

// Motor pin definitions

const int pulsePin = 18;  // Connect to TB6600 PULSE+

const int dirPin = 19;    // Connect to TB6600 DIR+

const int enaPin = 25;    // Connect to TB6600 ENA+

const int gpio15 = 15;    // GPIO15 for control

const int spectrum\_light = 14 ;

const int stripPin=17;//灯带

// Motor parameters

const int stepsPerRevolution = 64000;  // Steps per revolution

float rotationTime = 8.0;              // Time for one revolution (seconds)

// WiFi and MQTT configuration

const char \*ssid = "MiHome";              // WiFi SSID

const char \*password = "c107c107";     // WiFi password

const char \*mqtt\_broker = "139.9.68.120";  // MQTT broker address

const char \*sugar\_topic = "send-sugar";      // Topic to send spectrum data

const char \*measure\_topic = "measure\_once";  // Topic to trigger measurement

const char \*mqtt\_username = "";              // MQTT username (empty)

const char \*mqtt\_password = "";              // MQTT password (empty)

const int mqtt\_port = 1883;

                // MQTT port

// I2C device address

#define \_i2cAddr (0x39)

// Global objects

WiFiClient espClient;

PubSubClient client(espClient);

Ticker measurementTicker;  // Timer for spectrum measurement

// Measurement data

String spectrumData[4];    // Array to store 4 spectrum measurements

int measurementCount = 0;  // Counter for measurements

// Motor state

bool motorRunning = false; // Flag to indicate motor status

int motorStep = 0;         // Current step count

unsigned long lastPulseTime = 0;  // Time of last motor pulse

bool isMeasure = false;

// Function prototypes

void startRotate();

void startMeasurement();

void measureSpectrum();

void sendSpectrumData();

void runMotorPulse();

String ConstructJSON();

// Setup function

void setup() {

  // Initialize serial communication

  Serial.begin(115200);

  // Initialize I2C

  Wire.begin(21, 22);  // SDA = 21, SCL SCL= 22

  // Initialize motor pins

  pinMode(pulsePin, OUTPUT);

  pinMode(dirPin, OUTPUT);

  pinMode(enaPin, OUTPUT);

  pinMode(spectrum\_light, OUTPUT);

  pinMode(stripPin, OUTPUT);

  digitalWrite(enaPin, LOW);   // Enable motor (active low)

  digitalWrite(dirPin, HIGH);  // Set direction (clockwise)

  digitalWrite(stripPin, LOW);

  digitalWrite(spectrum\_light, HIGH);

//

  // Initialize GPIO15

  pinMode(gpio15, OUTPUT);

  digitalWrite(gpio15, LOW);   // Default low

  // Connect to WiFi

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL\_CONNECTED) {

    delay(500);

    Serial.println("Connecting to WiFi...");

  }

  Serial.println("Connected to WiFi");

  // Connect to MQTT broker

  client.setServer(mqtt\_broker, mqtt\_port);

  client.setCallback(callback);

  while (!client.connected()) {

    String client\_id = "esp32-client-" + String(WiFi.macAddress());

    if (client.connect(client\_id.c\_str(), mqtt\_username, mqtt\_password)) {

      Serial.println("Connected to MQTT broker");

    } else {

      Serial.print("MQTT connection failed, state: ");

      Serial.println(client.state());

      delay(2000);

    }

  }

  client.subscribe(measure\_topic);  // Subscribe to measure\_once topic

}

// Loop function

void loop() {

  client.loop();  // Handle MQTT messages

  // Non-blocking motor control

  if (motorRunning) {

    runMotorPulse();

  }

  else if (isMeasure){

    startMeasurement();

    isMeasure = false;

  }

}

// MQTT callback function

void callback(char \*topic, byte \*payload, unsigned int length) {

  if (strcmp(topic, measure\_topic) == 0) {

    Serial.println("Received measure\_once command");

    startRotate();

  }

}

void startRotate(){

  digitalWrite(gpio15, HIGH);

  delay(10);  // Hold high for 100ms (blocking here is acceptable as it's short)

  digitalWrite(gpio15, LOW);

  motorRunning = true;

  motorStep = 0;

  lastPulseTime = micros();

  isMeasure = true;

}

// Start measurement and motor rotation

void startMeasurement() {

  // Reset measurement state

  measurementCount = 0;

  digitalWrite(spectrum\_light, LOW);

  digitalWrite(stripPin, HIGH);

  Serial.println("Sent spectrum data: ");

  for (int i = 0; i < 4; i++) {

    spectrumData[i] = "";  // Clear previous data

  }

  delay(500);

  // Set GPIO15 high

  // digitalWrite(gpio15, HIGH);

  // delay(10);  // Hold high for 100ms (blocking here is acceptable as it's short)

  // digitalWrite(gpio15, LOW);

  // Start motor rotation

  motorRunning = true;

  motorStep = 0;

  lastPulseTime = micros();

  // Schedule spectrum measurements every 2 seconds

  measurementTicker.attach(2.0, measureSpectrum);

  // digitalWrite(spectrum\_light, HIGH);

}

// Measure spectrum once

void measureSpectrum() {

  if (measurementCount < 4) {

    // Perform spectrum measurement

    setATIME(0x64);         // Set integration time

    setASTEP(0xE7, 0x03);   // Set step time

    setGAIN(0x09);          // Set gain

    String jsonOutput = ConstructJSON();

    spectrumData[measurementCount] = jsonOutput;

    Serial.print("Measurement ");

    Serial.print(measurementCount + 1);

    Serial.println(": " + jsonOutput);

    measurementCount++;

  } else {

    // All measurements complete

    measurementTicker.detach();  // Stop the timer

    sendSpectrumData();          // Send the collected data

  }

}

// Send spectrum data as JSON array

void sendSpectrumData() {

  for (int i = 0; i < 4; i++) {

    String json = "{ ";

    json += "\"" + String(i) + "\"" + ":" ;

    json += spectrumData[i];

    json += "}";

    client.publish(sugar\_topic, json.c\_str());

  }

  // client.publish(sugar\_topic, json.c\_str());

  // Serial.println("Sent spectrum data: " + json);

  digitalWrite(spectrum\_light, HIGH);

  digitalWrite(stripPin, LOW);

}

// Non-blocking motor pulse control

void runMotorPulse() {

  float T = (rotationTime / stepsPerRevolution) \* 1000000.0;  // Pulse period in microseconds

  int delayTime = T / 2.0;                                    // Half period (high/low)

  unsigned long currentTime = micros();

  if (currentTime - lastPulseTime >= delayTime) {

    if (motorStep < stepsPerRevolution) {

      digitalWrite(pulsePin, !digitalRead(pulsePin));  // Toggle pulse pin

      lastPulseTime = currentTime;

      if (digitalRead(pulsePin) == LOW) {

        motorStep++;  // Increment step on falling edge

      }

    } else {

      motorRunning = false;  // Stop motor after one revolution

      digitalWrite(pulsePin, LOW);

    }

  }

}

// Spectrum measurement functions (unchanged from original code)

void setATIME(byte value) {

  writeRegister(0x81, value);

}

void setASTEP(byte value1, byte value2) {

  writeRegister(0xCA, value1);

  writeRegister(0xCB, value2);

}

void setGAIN(byte value) {

  writeRegister(0xAA, value);

}

byte readRegister(byte addr) {

  Wire.beginTransmission(\_i2cAddr);

  Wire.write(addr);

  Wire.endTransmission();

  Wire.requestFrom(\_i2cAddr, 1);

  if (Wire.available()) {

    return Wire.read();

  } else {

    Serial.println("I2C Error");

    return 0xFF;

  }

}

uint16\_t readTwoRegister1(byte addr) {

  uint8\_t readingL;

  uint16\_t readingH;

  uint16\_t reading = 0;

  Wire.beginTransmission(\_i2cAddr);

  Wire.write(addr);

  Wire.endTransmission();

  Wire.requestFrom(\_i2cAddr, 2);

  if (Wire.available() >= 2) {

    readingL = Wire.read();

    readingH = Wire.read();

    readingH = readingH << 8;

    reading = (readingH | readingL);

    return reading;

  } else {

    Serial.println("I2C Error");

    return 0xFFFF;

  }

}

void writeRegister(byte addr, byte val) {

  Wire.beginTransmission(\_i2cAddr);

  Wire.write(addr);

  Wire.write(val);

  Wire.endTransmission();

}

void PON() {

  byte regVal = readRegister(0x80);

  regVal = (regVal & 0xFE) | 0x01;

  writeRegister(0x80, regVal);

}

void SmuxConfigRAM() {

  writeRegister(0xAF, 0x10);

}

void SpEn(bool isEnable) {

  byte regVal = readRegister(0x80);

  regVal = (regVal & 0xFD) | (isEnable ? 0x02 : 0x00);

  writeRegister(0x80, regVal);

}

void SMUXEN() {

  byte regVal = readRegister(0x80);

  regVal = (regVal & 0xEF) | 0x10;

  writeRegister(0x80, regVal);

}

bool getSmuxEnabled() {

  return (readRegister(0x80) & 0x10) == 0x10;

}

bool getIsDataReady() {

  return (readRegister(0xA3) & 0x40) == 0x40;

}

void F1F4\_Clear\_NIR() {

  writeRegister(0x00, 0x30);

  writeRegister(0x01, 0x01);

  writeRegister(0x02, 0x00);

  writeRegister(0x03, 0x00);

  writeRegister(0x04, 0x00);

  writeRegister(0x05, 0x42);

  writeRegister(0x06, 0x00);

  writeRegister(0x07, 0x00);

  writeRegister(0x08, 0x50);

  writeRegister(0x09, 0x00);

  writeRegister(0x0A, 0x00);

  writeRegister(0x0B, 0x00);

  writeRegister(0x0C, 0x20);

  writeRegister(0x0D, 0x04);

  writeRegister(0x0E, 0x00);

  writeRegister(0x0F, 0x30);

  writeRegister(0x10, 0x01);

  writeRegister(0x11, 0x50);

  writeRegister(0x12, 0x00);

  writeRegister(0x13, 0x06);

}

void F5F8\_Clear\_NIR() {

  writeRegister(0x00, 0x00);

  writeRegister(0x01, 0x00);

  writeRegister(0x02, 0x00);

  writeRegister(0x03, 0x40);

  writeRegister(0x04, 0x02);

  writeRegister(0x05, 0x00);

  writeRegister(0x06, 0x10);

  writeRegister(0x07, 0x03);

  writeRegister(0x08, 0x50);

  writeRegister(0x09, 0x10);

  writeRegister(0x0A, 0x03);

  writeRegister(0x0B, 0x00);

  writeRegister(0x0C, 0x00);

  writeRegister(0x0D, 0x00);

  writeRegister(0x0E, 0x24);

  writeRegister(0x0F, 0x00);

  writeRegister(0x10, 0x00);

  writeRegister(0x11, 0x50);

  writeRegister(0x12, 0x00);

  writeRegister(0x13, 0x06);

}

uint16\_t\* ReadRawValuesMode1() {

  static uint16\_t values[6];

  bool isEnabled = true;

  bool isDataReady = false;

  PON();

  SmuxConfigRAM();

  F1F4\_Clear\_NIR();

  SMUXEN();

  while (isEnabled) {

    isEnabled = getSmuxEnabled();

  }

  SpEn(true);

  while (!isDataReady) {

    isDataReady = getIsDataReady();

  }

  values[0] = readTwoRegister1(0x95);  // F1

  values[1] = readTwoRegister1(0x97);  // F2

  values[2] = readTwoRegister1(0x99);  // F3

  values[3] = readTwoRegister1(0x9B);  // F4

  values[4] = readTwoRegister1(0x9D);  // Clear

  values[5] = readTwoRegister1(0x9F);  // NIR

  return values;

}

uint16\_t\* ReadRawValuesMode2() {

  static uint16\_t values[6];

  bool isEnabled = true;

  bool isDataReady = false;

  PON();

  SpEn(false);

  SmuxConfigRAM();

  F5F8\_Clear\_NIR();

  SMUXEN();

  while (isEnabled) {

    isEnabled = getSmuxEnabled();

  }

  SpEn(true);

  while (!isDataReady) {

    isDataReady = getIsDataReady();

  }

  values[0] = readTwoRegister1(0x95);  // F5

  values[1] = readTwoRegister1(0x97);  // F6

  values[2] = readTwoRegister1(0x99);  // F7

  values[3] = readTwoRegister1(0x9B);  // F8

  values[4] = readTwoRegister1(0x9D);  // Clear

  values[5] = readTwoRegister1(0x9F);  // NIR

  return values;

}

String ConstructJSON() {

  uint16\_t\* values1 = ReadRawValuesMode1();

  uint16\_t\* values2 = ReadRawValuesMode2();

  String json = "{";

  json += "\"F1\": " + String(values1[0]) + ",";

  json += "\"F2\": " + String(values1[1]) + ",";

  json += "\"F3\": " + String(values1[2]) + ",";

  json += "\"F4\": " + String(values1[3]) + ",";

  json += "\"F5\": " + String(values2[0]) + ",";

  json += "\"F6\": " + String(values2[1]) + ",";

  json += "\"F7\": " + String(values2[2]) + ",";

  json += "\"F8\": " + String(values2[3]) + ",";

  json += "\"Clear1\": " + String(values1[4]) + ",";

  json += "\"NIR1\": " + String(values1[5]) + ",";

  json += "\"Clear2\": " + String(values2[4]) + ",";

  json += "\"NIR2\": " + String(values2[5]);

  json += "}";

  return json;

}

# 二、新增硬件（屏幕与语音）代码

# This file is executed on every boot (including wake-boot from deepsleep)

#import esp

#esp.osdebug(None)

#import webrepl

#webrepl.start()

import machine

import time

from umqttsimple import MQTTClient

import network

from machine import Timer, UART, Pin, PWM

import ujson

MAX\_DATA\_LENGTH = 8 # 或根据需要设置最大数据长度

TIMEOUT = 0.1 # 超时设置，单位：秒

# 初始化串口

uart\_screen = UART(2, 115200)

uart\_voice = UART(1, 9600, rx=Pin(2), tx=Pin(4))

song\_name\_dict = {

"0": [0x01, 0.5],

"1": [0x02, 1.0],

"十": [0x03, 0.5],

"2": [0x04, 1.5],

"3": [0x05, 1.0],

"4": [0x06, 0.5],

"5": [0x07, 1.0],

"6": [0x08, 1.0],

"7": [0x09, 0.5],

"8": [0x0A, 0.5],

"9": [0x0B, 0.5],

"开始检测": [0x0C, 0.5],

"糖度为": [0x0D, 0.5],

"PH值": [0x0E, 0.5],

"水分": [0x0F, 0.5],

"黑点数": [0x10, 0.5],

"摄像头": [0x11, 0.5],

"厘米": [0x12, 0.5],

"百分之": [0x13, 0.5],

"擦伤面积": [0x14, 0.5],

".": [0x15, 0.5],

"个": [0x16, 0.5],

"枇杷": [0x17, 0.5],

"平方厘米": [0x18, 0.5],

"等待中": [0x19, 0.5],

"检测结束": [0x1A, 0.5],

"初等糖度": [0x1B, 0.5],

"中等糖度": [0x1C, 0.5],

"高等糖度": [0x1D, 0.5],

"低甜度": [0x1E, 1.0],

"中等甜度": [0x1F, 0.5],

"高甜度": [0x20, 0.5],

"超高甜度": [0x21, 0.5],

"果径": [0x22, 0.5],

"小果": [0x23, 0.5],

"中果": [0x24, 0.5],

"大果": [0x25, 0.5],

"特大果": [0x26, 0.5],

"十": [0x27, 0.5],

"百": [0x28, 1.5],

}

def receive\_data():

received\_data = b'' # 初始化空字节串

start\_time = time.time() # 记录开始时间

while True:

if uart\_screen.any(): # 检查是否有数据接收

chunk = uart\_screen.read() # 读取当前数据

received\_data += chunk # 累加数据

# 检查接收数据是否已满或超时

if len(received\_data) >= MAX\_DATA\_LENGTH or (time.time() - start\_time) > TIMEOUT:

break # 数据接收完成或超时，退出循环

# 打印接收到的数据长度和内容

if len(received\_data) != 8:

return

int\_val = int.from\_bytes(received\_data[7:], "big")

print(int\_val)

if int\_val < 32 or int\_val > 33:

return

if int\_val == 32:

client.publish(b'measure\_once\_result', f"")

elif int\_val == 33:

client.publish(b'measure\_once', f"")

prefix = bytes.fromhex("7EFF06030000")

voice\_info\_list = song\_name\_dict.get("开始检测")

index = voice\_info\_list[0].to\_bytes(1, 'big')

prefix += index + bytes.fromhex("EF")

uart\_voice.write(prefix)

def parser\_spectrum\_to\_hex(spectrum):

order\_list = ["F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "Clear1", "NIR1", "Clear2", "NIR2"]

spectrum\_list = [spectrum[item] for item in order\_list]

data\_bytes = b''.join([num.to\_bytes(2, 'big') for num in spectrum\_list])

write\_cmd = b'\x10'

id\_address = b'\xC0\x01'

prefix = b'\x5A\xA5\x1B'

instruction = prefix + write\_cmd + id\_address + data\_bytes

return instruction

# MQTT相关函数保持不变

def receive\_gb2312\_to\_uart(str\_encode, addr):

prefix = bytes.fromhex("5AA5")

str\_encode\_len = len(str\_encode)

frame\_len = str\_encode\_len + 5

prefix += frame\_len.to\_bytes(1, 'big') + bytes.fromhex("10") + addr + str\_encode + bytes.fromhex("0000")

return prefix

def subscribe\_list():

client.subscribe('send-sugar')

client.subscribe('sugar\_server/show\_text') # 订阅温度信息

client.subscribe('sugar\_server/play\_voice') # 订阅温度信息

def do\_connect(ssid, password):

wlan = network.WLAN(network.STA\_IF)

wlan.active(True)

if not wlan.isconnected():

wlan.connect(ssid, password)

i = 1

while not wlan.isconnected():

print(f"正在连接网络：{ssid}...{i}s")

time.sleep(1)

i += 1

print('网络已连接，IP:', wlan.ifconfig()[0])

def timer\_task(self):

subscribe\_list()

def sub\_cb(topic, msg):

print(topic, msg)

msg = msg.decode('utf-8')

if topic == b'send-sugar':

try:

spectrum = ujson.loads(msg).get("3")

print(spectrum)

if spectrum is None:

return

uart\_screen.write(bytes.fromhex("5AA50310E001"))

spectrum\_instruct = parser\_spectrum\_to\_hex(spectrum)

uart\_screen.write(spectrum\_instruct)

except:

pass

elif topic == b'sugar\_server/show\_text':

try:

str\_encode = bytes.fromhex(msg)

prefix = receive\_gb2312\_to\_uart(str\_encode, bytes.fromhex("0026"))

uart\_screen.write(prefix)

except:

pass

elif topic == b'sugar\_server/play\_voice':

try:

tokenized\_list = ujson.loads(msg)["voice"]

for voice\_token in tokenized\_list:

prefix = bytes.fromhex("7EFF06030000")

voice\_info\_list = song\_name\_dict.get(voice\_token)

if voice\_info\_list:

index = voice\_info\_list[0].to\_bytes(1, 'big')

prefix += index + bytes.fromhex("EF")

uart\_voice.write(prefix)

print(prefix)

time.sleep(voice\_info\_list[1])

else:

print(f"Text '{voice\_token}' not found in song\_name\_dict")

except:

pass

# 连接网络

do\_connect('MiHome', 'c107c107')

# 初始化MQTT客户端

client = MQTTClient("sugar\_server/screen", "139.9.68.120", 1883)

client.connect()

client.set\_callback(sub\_cb)

timer = Timer(0)

timer.init(period=5000, mode=Timer.PERIODIC, callback=timer\_task)

subscribe\_list()

while True:

receive\_data()