California Polytechnic State University Pomona

Department of Electrical and Computer Engineering

Intro to Microcontrollers

ECE 3301L

Final Lab Report

Prepared by:

Daniel Romero

&

Isaias Rafael-Sepulveda

Presented to:

Sasoun Torousian

December 8, 2024

Table of Contents

Page

Wet Bulb Globe Temperature

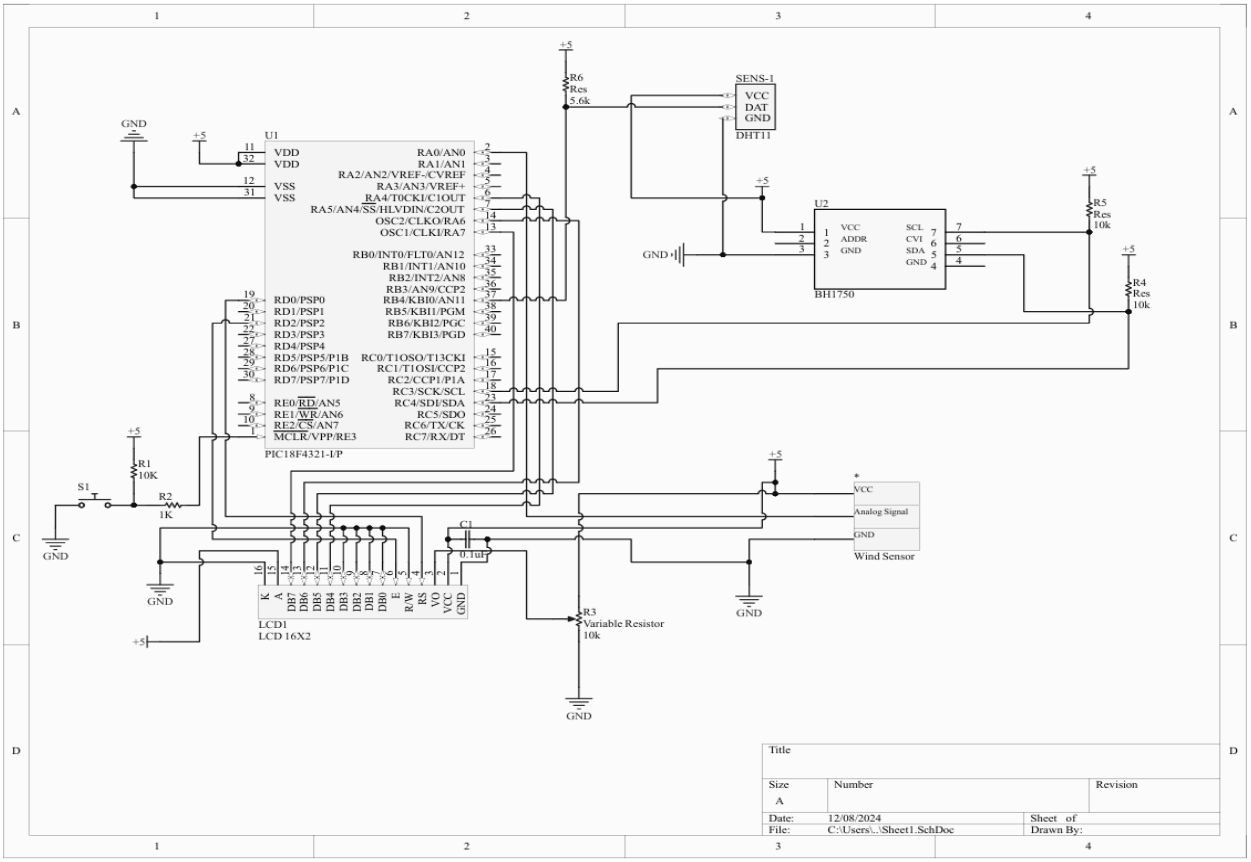
Our project focuses on measuring heat stress for outdoors activities that take into account temperature, humidity, wind speed, and solar radiation. These measurements help schools and other outdoor recreational organizations determine how outdoor conditions will affect their athletes' bodies. There is a chart associated with these measurements that gives more detailed information on what actions to take based on the measurement results.

The main components we decided to use for our project are as follows:

Parts:

- Pic18F4321
- Breadboard
- LCD Display
- DHT11
- BH1750
- Elecfreaks Octopus Wind Speed Sensor
- And several miscellaneous components including resistors, wires, capacitors, etc...
-

Wiring Diagram:



Challenges to Consider

1. **Calibration:**
    - Calibrate all sensors to improve measurement accuracy.

Solution: Data sheet for the BHT1750 recommended we use a factor of 1.2 to calibrate the sensor when operating in high-resolution mode.

```
while (1) {
    // Read light intensity from BH1750
    unsigned int rawLight = BH1750_Read_Light();
    char debugBuffer[16];
    sprintf(debugBuffer, "Raw Lux: %u", rawLight);
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string(debugBuffer);
    __delay_ms(1000);
    lux = (float)rawLight / 1.2f;
```

2. **Power Requirements:**
    - Ensure the PIC18F4321 and all sensors are properly powered. You may need a regulated 5V or 3.3V power source depending on the sensor requirements.

    Solution: All of our components worked off of 5V except the BHT1750 which required 3.3V but we purchased the sensor with a breakout board that automatically stepped this voltage down.

3. **Data Interpretation and Conversion:**
    - For solar radiation, determine a realistic lux-to-radiation conversion factor based on empirical data or research.

Solution: We found a conversion factor that helped us interpret radiation from lux. (See DHT11 Radiation section for details).

    - For wind speed, ensure the output (pulses or voltage) is correctly translated into a speed value.

Solution: We multiplied the wind speed sensor's value by the 5V Vref+ that we selected in the ADCON register and then divided by 1.23 (max output voltage) to scale our result of wind speed accurately. (See Wind Speed Sensor Section for details)

4. **Programming:**
    - Implement communication protocols for each sensor (I2C for BH1750, single wire for DHT11, ADC or timer for the wind sensor).

Solution: Communication protocol for the BHT1750 was in I^2C but everything else was in SPI so to make the sensor work in SPI we had to simulate I^2C through bit bashing in our programming. (See the BHT1750 section for details)

- Develop the WBGT calculation algorithm and test thoroughly to ensure the outputs match expectations.

5. Memory Allocation:

   - Once our project code was close to finishing, we ran out of space on the microcontroller's program memory.

Solution: We converted most of our variable to integers from doubles and floats to save space, we eliminated several unused functions in the LCD.c and LCD.h files, and we set stricter optimization settings on the code compiler.

LAB

WBGT Calculation = (0.1 * Dry bulb temp) + (0.7 * Natural Wet Bulb Temp) + (0.2 * Globe Temp)

Where,

Dry Bulb Temp = air temperature (DHT11)

Natural Wet Bulb Temp = Humidity + Wind + Radiation (DHT11 + Wind Speed Sensor)

Globe Temperature = Radiation + Wind (BH1750 + Wind Speed Sensor)

Radiation:

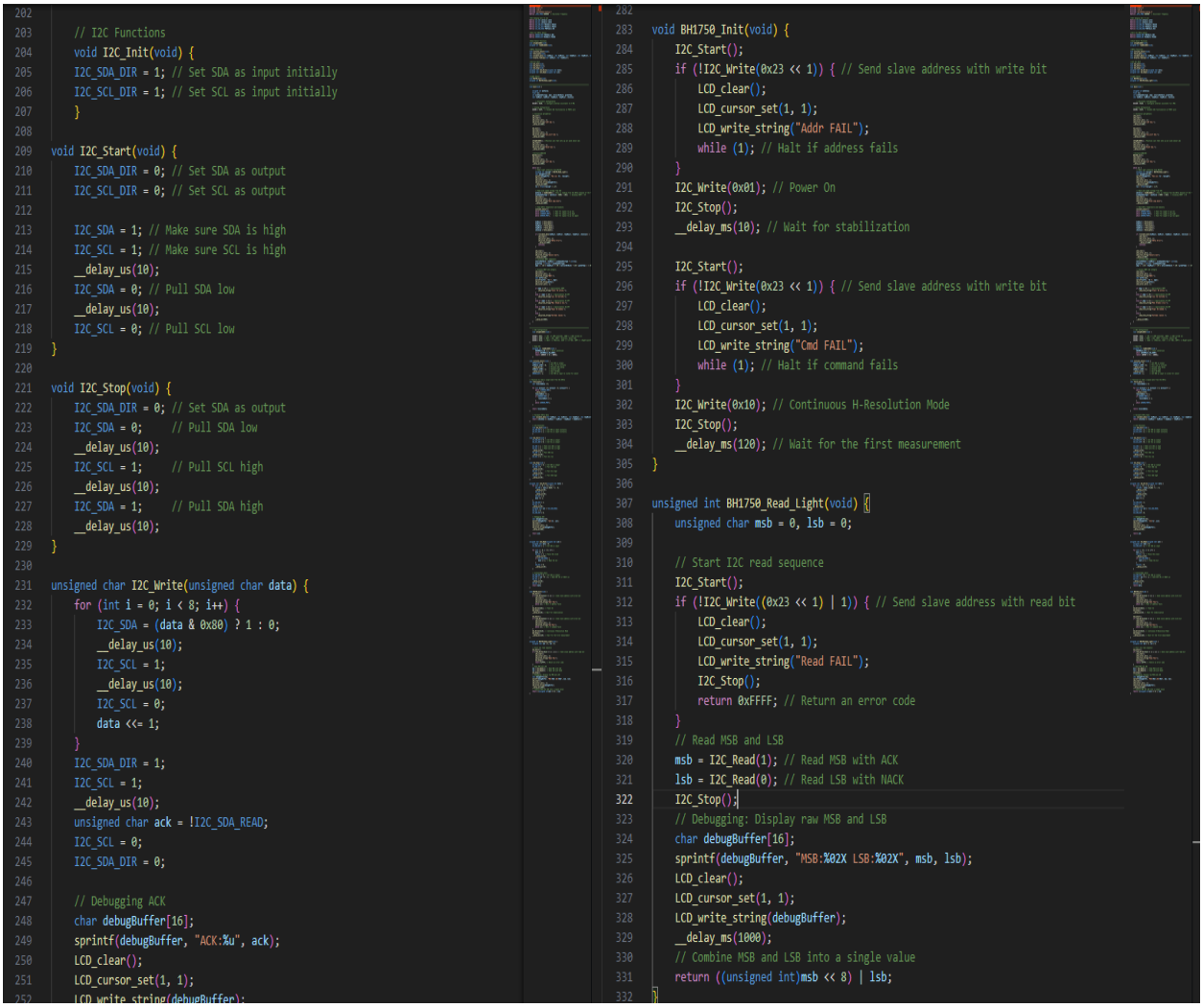
Solar Irradiance is typically measured as $\frac{W}{m^2}$, where the luminous efficacy of daylight is averaged at about 126.7 lumens per watt (energy across all wavelengths). Our light sensor (BH1750) measures light in lux which is light as perceived by the human eye. Since what we could afford for our project was a light sensor and not a radiation sensor, we will use the conversion factor of:

$$1\ lux\ =\ \frac{1}{126.7\left(\frac{W}{m^2}\right)} \approx 0.0079\left(\frac{W}{m^2}\right)$$

This is how we will derive a value for radiation to calculate into our equations.

The BHT1750 sensor outputs a 16-bit raw value that is proportional to the light intensity. That value will then be calibrated with the recommended 1.2 factor per the data sheet when operating in high-resolution mode.

BHT1750 (Light Sensor) : This sensor communicates through I^2C protocol using the SDA and SCL pins that have built-in pull-up resistors (10k) implemented into the break-out board. Since our microcontroller will be running in as *SPI* protocol with all the other components, we decided to simulate the I^2C protocol using GPIO pins (RC0 and RC1) and software-controlled timing. This method was further made available to us because the BH1750 sensor does not require high-speed communication or complex timing.



```
    // I2C Functions
    void I2C_Init(void) {
    I2C_SDA_DIR = 1; // Set SDA as input initially
    I2C_SCL_DIR = 1; // Set SCL as input initially
    }

void I2C_Start(void) {
    I2C_SDA_DIR = 0; // Set SDA as output
    I2C_SCL_DIR = 0; // Set SCL as output

    I2C_SDA = 1; // Make sure SDA is high
    I2C_SCL = 1; // Make sure SCL is high
    __delay_us(10);
    I2C_SDA = 0; // Pull SDA low
    __delay_us(10);
    I2C_SCL = 0; // Pull SCL low
}

void I2C_Stop(void) {
    I2C_SDA_DIR = 0; // Set SDA as output
    I2C_SDA = 0;     // Pull SDA low
    __delay_us(10);
    I2C_SCL = 1;     // Pull SCL high
    __delay_us(10);
    I2C_SDA = 1;     // Pull SDA high
    __delay_us(10);
}

unsigned char I2C_Write(unsigned char data) {
    for (int i = 0; i < 8; i++) {
        I2C_SDA = (data & 0x80) ? 1 : 0;
        __delay_us(10);
        I2C_SCL = 1;
        __delay_us(10);
        I2C_SCL = 0;
        data <<= 1;
    }
    I2C_SDA_DIR = 1;
    I2C_SCL = 1;
    __delay_us(10);
    unsigned char ack = !I2C_SDA_READ;
    I2C_SCL = 0;
    I2C_SDA_DIR = 0;

    // Debugging ACK
    char debugBuffer[16];
    sprintf(debugBuffer, "ACK:%u", ack);
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string(debugBuffer);
```

```
void BH1750_Init(void) {
    I2C_Start();
    if (!I2C_Write(0x23 << 1)) { // Send slave address with write bit
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Addr FAIL");
        while (1); // Halt if address fails
    }
    I2C_Write(0x01); // Power On
    I2C_Stop();
    __delay_ms(10); // Wait for stabilization

    I2C_Start();
    if (!I2C_Write(0x23 << 1)) { // Send slave address with write bit
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Cmd FAIL");
        while (1); // Halt if command fails
    }
    I2C_Write(0x10); // Continuous H-Resolution Mode
    I2C_Stop();
    __delay_ms(120); // Wait for the first measurement
}

unsigned int BH1750_Read_Light(void) {
    unsigned char msb = 0, lsb = 0;

    // Start I2C read sequence
    I2C_Start();
    if (!I2C_Write((0x23 << 1) | 1)) { // Send slave address with read bit
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Read FAIL");
        I2C_Stop();
        return 0xFFFF; // Return an error code
    }
    // Read MSB and LSB
    msb = I2C_Read(1); // Read MSB with ACK
    lsb = I2C_Read(0); // Read LSB with NACK
    I2C_Stop();
    // Debugging: Display raw MSB and LSB
    char debugBuffer[16];
    sprintf(debugBuffer, "MSB:%02X LSB:%02X", msb, lsb);
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string(debugBuffer);
    __delay_ms(1000);
    // Combine MSB and LSB into a single value
    return ((unsigned int)msb << 8) | lsb;
}
```

DHT11

The DHT11 is a digital sensor used to calculate temperature and relative humidity. It operates using a resistive humidity sensor and a thermistor for temperature detection. The sensor communicates with the microcontroller using a single-wire protocol, transmitting data through a single data line. To initiate communication, the microcontroller sends a start signal by pulling the data line low for at least 18 milliseconds. The DHT11 then responds with a 40-bit data packet comprising 16 bits for humidity (8 bits integer, 8 bits decimal), 16 bits for temperature (8 bits integer, 8 bits decimal), and an 8-bit checksum for error detection. Data is transmitted as a series of high and low pulses, with the duration of the high pulses indicating whether the bit is a 1 or 0. A pull-up resistor (typically 4.7kΩ to 10kΩ) is required on the data line to ensure the line remains at a stable logic high when not actively driven by the sensor or microcontroller

```
1
2  void initiate_sensor(void) {
3      SENSOR_DIR = 0;       // Set RB4 as output
4      LATBbits.LATB4 = 0;   // Pull low to initiate
5      __delay_ms(18);       // Minimum 18ms delay
6      LATBbits.LATB4 = 1;   // Release high
7      __delay_us(35);       // Wait 20-40us
8      SENSOR_DIR = 1;       // Set RB4 as input to listen for sensor
9  }
10
11 char fetch_byte() {
12     char receivedData = 0;
13
14     for (int bitCount = 0; bitCount < 8; bitCount++) {
15         while (!SENSOR_PORT);  // Wait for signal to go high
16         __delay_us(35);        // Wait for the high pulse duration
17         receivedData <<= 1;    // Shift to make space for the next bit
18         if (SENSOR_PORT) {     // Check if the bit is 1
19             receivedData |= 1;
20         }
21         while (SENSOR_PORT);   // Wait for signal to go low
22     }
23     return receivedData;
24 }
25
26 char validate_data(char humMajor, char humMinor, char tempMajor, char tempMinor,
27 char checkSum) {
28     return (checkSum == (humMajor + humMinor + tempMajor + tempMinor));
29 }
30
```

Elecfreaks Octopus Wind Speed Sensor

Our wind speed sensor was a simple device that would spin under the effect of wind power which then forced a DC motor to generate a DC voltage that could generate a maximum output voltage of 1.2V. We used AN0 on our Pic18f4321 to read this analog input and used the appropriate ADCON registers to properly handle the data according to our internal oscillator frequency.

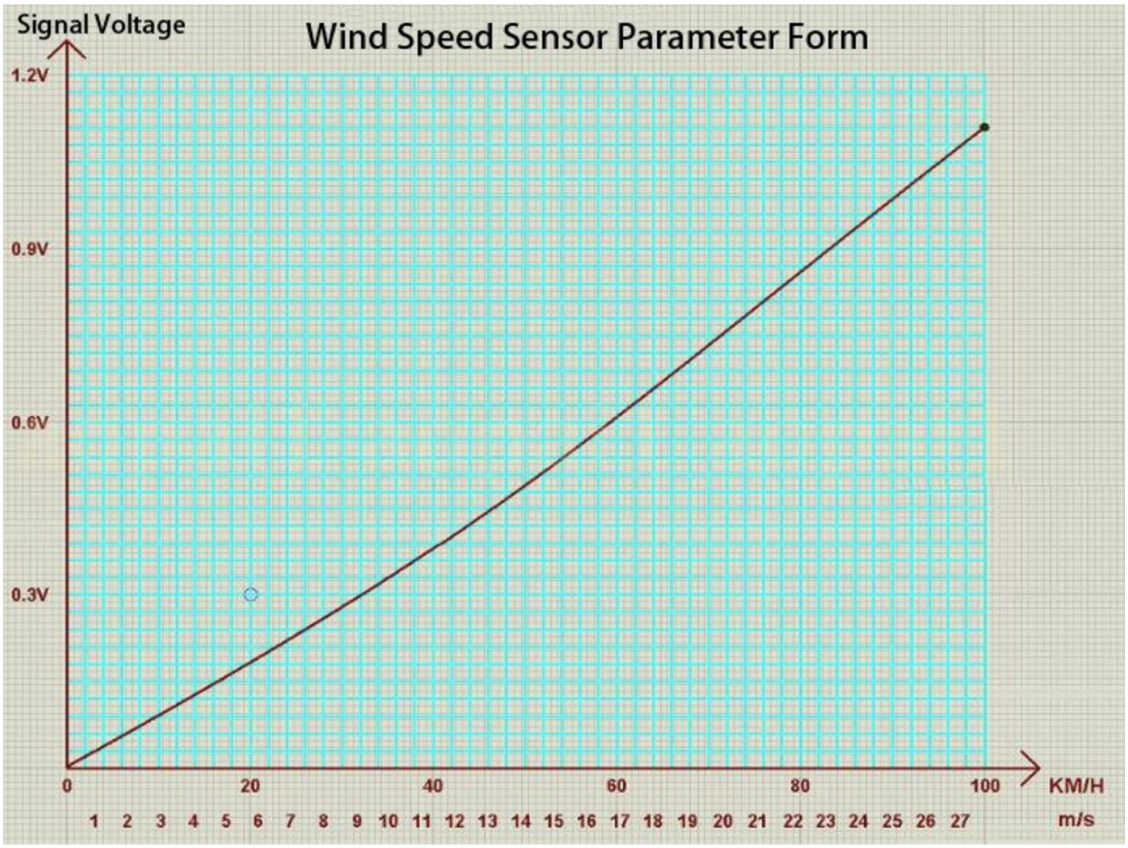
```
// ADC Configuration
void setupWindADC(void) {

    ADCON0 = 0x01;  // CHS = 0 (AN0 selected), ADON = 1 (ADC turned on)
    ADCON1 = 0x0E; // VREF- = VSS, VREF+ = External, Select AN1
    ADCON2 = 0x2A;  // ADCS = 2 (FOSC/32), ACQT = 2 (4 TAD), ADFM = 1 (Right-justified)
}

// Read ADC
unsigned int readWindADC(void) {
    ADCON0bits.GO = 1; // Start conversion
    while (ADCON0bits.GO_nDONE);
    return (ADRESH << 8) | ADRESL;
}
```

Code for how the data was then handle

```
1  adcValue = readWindADC(); // Read the ADC value
2  windSpeedVoltage = (adcValue * 5000) / 1023;  // Convert to voltage
3
```



Conclusion:

In conclusion, we successfully designed and implemented a functional Wet Bulb Globe Temperature (WBGT) measurement system using the PIC18F4321 microcontroller. By properly calibrating and positioning the DHT11, BH1750, and Elecfreaks Octopus Wind Speed Sensor, we achieved accurate measurements of temperature, humidity, light intensity, and wind speed, which were used to compute the WBGT index. The project included real-time data display on an LCD and addressed challenges such as implementing software-based I2C communication for the BH1750, optimizing ADC readings for the wind sensor, and ensuring reliable sensor performance. This project provided valuable experience in environmental monitoring and sensor integration, showcasing a practical approach to combining multiple sensors into a cohesive system.

Project Code:

```
#include <xc.h>
#include <stdio.h>
#include "LCD.h"
#include "pic18f4321-Config.h"
#define _XTAL_FREQ 4000000  // Oscillator frequency

//Bit banging bits to use
#define I2C_SDA LATCbits.LATC4
#define I2C_SCL LATCbits.LATC3
#define I2C_SDA_DIR TRISCbits.TRISC4
#define I2C_SCL_DIR TRISCbits.TRISC3
#define I2C_SDA_READ PORTCbits.RC4

//Port for DHT11 Sensor
#define SENSOR_PORT PORTBbits.RB4
```

```c
#define SENSOR_DIR TRISBbits.TRISB4

//Wind Sensor Functions
void setupWindADC(void);
unsigned int readWindADC(void);

//DHT11 Functions
void initiate_sensor(void);
char fetch_byte(void);
char validate_data(char humMajor, char humMinor, char tempMajor, char tempMinor,
char checkSum);
void display_readings(char humMajor, char tempMajor);

//I2C Bit Bash
void I2C_Init(void);
void I2C_Start(void);
void I2C_Stop(void);
unsigned char I2C_Write(unsigned char data);
unsigned char I2C_Read(unsigned char ack);

//Light Sensor
void BH1750_Init(void);
unsigned int BH1750_Read_Light(void);

////////////////////////////////////////////////////////////////////////////////
void main(void) {

    unsigned int adcValue;
    float lux;
    int windSpeedVoltage, wbgt, naturalWetBulb, globeTemp;
    char humMajor, humMinor, tempMajor, tempMinor, checksum;

        // Oscillator Configuration
    OSCCON = 0x60;  // Configure internal oscillator to 4 MHz

    // Analog Configuration
    ADCON1 = 0x0F;  // Disable ADC functionality on PORTC pins

    // Initialize peripherals
    LCD_init();
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string("LCD? Dub.");
```

```
    __delay_ms(1000);

    I2C_Init();
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string("I2C_init? Dub.");
    __delay_ms(1000);

    setupWindADC(); //Function call that sets up our wind sensor adc
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string("Wind? Dub.");
    __delay_ms(1000);

    //InitializeBH1750
    BH1750_Init();
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string("Lux? Dub.");
    __delay_ms(1000);

    while (1) {
        // Read light intensity from BH1750
        unsigned int rawLight = BH1750_Read_Light();
        char debugBuffer[16];
        sprintf(debugBuffer, "Raw Lux: %u", rawLight);
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string(debugBuffer);
        __delay_ms(1000);
        lux = (float)rawLight / 1.2f;

        // Read wind speed voltage from ADC
        adcValue = readWindADC(); //This is reading from the DHT11 because of the
function call
        windSpeedVoltage = (adcValue * 5000) / 1023;  // Assuming VREF+ = 5V
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Wind comp done");
        __delay_ms(1000);

        // Read DHT11 temperature and humidity
        initiate_sensor();
```

```
        while (SENSOR_PORT);    // Wait for signal to go low
        while (!SENSOR_PORT);   // Wait for signal to go high
        while (SENSOR_PORT);    // Wait for signal to go low again


        humMajor = fetch_byte();
        humMinor = fetch_byte();
        tempMajor = fetch_byte();
        tempMinor = fetch_byte();
        checksum = fetch_byte();

        if (!validate_data(humMajor, humMinor, tempMajor, tempMinor, checksum)) {
            LCD_clear();
            LCD_cursor_set(1, 1);
            LCD_write_string("DHT11 Error");
            __delay_ms(1000);
            continue;
        }

        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Sensors dun");
        __delay_ms(500);
        // Calculate WBGT components
        naturalWetBulb = humMajor + windSpeedVoltage + (int)lux;
        globeTemp = (int)lux + windSpeedVoltage;
        wbgt =((10 * tempMajor)+(70 * naturalWetBulb)+(20 * globeTemp))/ 100;

        // Display WBGT and category
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("WBGT:");
        char buffer[16];
        sprintf(buffer, "%d C", wbgt);
        LCD_write_string(buffer);
        LCD_cursor_set(2, 1);

        if (wbgt <= 26) { // Approximately 80°F
            LCD_write_string("Clear: No action.");
        }
        else if (wbgt <= 29) { // Approximately 84.9°F
            LCD_write_string("Low: 5m rest/25m.");
        }
```

```
        else if (wbgt <= 31) { // Approximately 87.9°F
            LCD_write_string("Med: Breaks & Ice.");
        }
        else if (wbgt <= 32) { // Approximately 88.9°F
            LCD_write_string("High: Ob needed.");
        }
        else {
            LCD_write_string("Extreme: Cancel.");
        }
        __delay_ms(4000);
    }
}

////////////////////////////////////////////////////////////////////////////////
    // ADC Configuration
    void setupWindADC(void) {

    ADCON0 = 0x01;  // CHS = 0 (AN0 selected), ADON = 1 (ADC turned on)
    ADCON1 = 0x0E; // VREF- = VSS, VREF+ = External, Select AN1
    ADCON2 = 0x2A;  // ADCS = 2 (FOSC/32), ACQT = 2 (4 TAD), ADFM = 1 (Right-
justified)
}

    // Read ADC
    unsigned int readWindADC(void) {
        ADCON0bits.GO = 1; // Start conversion
        while (ADCON0bits.GO_nDONE);
        return (ADRESH << 8) | ADRESL;
    }

void initiate_sensor(void) {
    SENSOR_DIR = 0;       // Set RB5 as output
    LATBbits.LATB4 = 0;   // Pull low to initiate
    __delay_ms(18);       // Minimum 18ms delay
    LATBbits.LATB4 = 1;   // Release high
    __delay_us(35);       // Wait 20-40us
    SENSOR_DIR = 1;       // Set RB5 as input to listen for sensor
}

// Function to read a single byte from the DHT11
char fetch_byte() {
    char receivedData = 0;
```

```
    for (int bitCount = 0; bitCount < 8; bitCount++) {
        while (!SENSOR_PORT);
        __delay_us(35);
        receivedData <<= 1;
        if (SENSOR_PORT) {
            receivedData |= 1;
        }
        while (SENSOR_PORT);
    }

    return receivedData;
}

    // Validate DHT11 Data
    char validate_data(char humMajor, char humMinor, char tempMajor, char
tempMinor, char checkSum) {
    return (checkSum == (humMajor + humMinor + tempMajor + tempMinor));
}

    // I2C Functions
    void I2C_Init(void) {
    I2C_SDA_DIR = 1; // Set SDA as input initially
    I2C_SCL_DIR = 1; // Set SCL as input initially
    }

void I2C_Start(void) {
    I2C_SDA_DIR = 0; // Set SDA as output
    I2C_SCL_DIR = 0; // Set SCL as output

    I2C_SDA = 1; // Make sure SDA is high
    I2C_SCL = 1; // Make sure SCL is high
    __delay_us(10);
    I2C_SDA = 0; // Pull SDA low
    __delay_us(10);
    I2C_SCL = 0; // Pull SCL low
}

void I2C_Stop(void) {
    I2C_SDA_DIR = 0; // Set SDA as output
    I2C_SDA = 0;     // Pull SDA low
    __delay_us(10);
    I2C_SCL = 1;     // Pull SCL high
    __delay_us(10);
```

```
    I2C_SDA = 1;     // Pull SDA high
    __delay_us(10);
}

unsigned char I2C_Write(unsigned char data) {
    for (int i = 0; i < 8; i++) {
        I2C_SDA = (data & 0x80) ? 1 : 0;
        __delay_us(10);
        I2C_SCL = 1;
        __delay_us(10);
        I2C_SCL = 0;
        data <<= 1;
    }
    I2C_SDA_DIR = 1;
    I2C_SCL = 1;
    __delay_us(10);
    unsigned char ack = !I2C_SDA_READ;
    I2C_SCL = 0;
    I2C_SDA_DIR = 0;

    // Debugging ACK
    char debugBuffer[16];
    sprintf(debugBuffer, "ACK:%u", ack);
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string(debugBuffer);
    __delay_ms(500);

    return ack;
}


unsigned char I2C_Read(unsigned char ack) {
    unsigned char data = 0;
    I2C_SDA_DIR = 1; // Set SDA as input

    for (int i = 0; i < 8; i++) {
        data <<= 1;
        I2C_SCL = 1; // Pulse the clock
        __delay_us(10);
        if (I2C_SDA_READ) {
            data |= 1; // Read the bit
        }
```

```
        I2C_SCL = 0;
        __delay_us(10);
    }

    // Acknowledge phase
    I2C_SDA_DIR = 0; // Set SDA as output
    I2C_SDA = ack ? 0 : 1; // Send ACK (0) or NACK (1)
    I2C_SCL = 1;
    __delay_us(10);
    I2C_SCL = 0;
    return data;
}

void BH1750_Init(void) {
    I2C_Start();
    if (!I2C_Write(0x23 << 1)) { // Send slave address with write bit
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Addr FAIL");
        while (1); // Halt if address fails
    }
    I2C_Write(0x01); // Power On
    I2C_Stop();
    __delay_ms(10); // Wait for stabilization

    I2C_Start();
    if (!I2C_Write(0x23 << 1)) { // Send slave address with write bit
        LCD_clear();
        LCD_cursor_set(1, 1);
        LCD_write_string("Cmd FAIL");
        while (1); // Halt if command fails
    }
    I2C_Write(0x10); // Continuous H-Resolution Mode
    I2C_Stop();
    __delay_ms(120); // Wait for the first measurement
}

unsigned int BH1750_Read_Light(void) {
    unsigned char msb = 0, lsb = 0;

    // Start I2C read sequence
    I2C_Start();
    if (!I2C_Write((0x23 << 1) | 1)) { // Send slave address with read bit
        LCD_clear();
```

```
        LCD_cursor_set(1, 1);
        LCD_write_string("Read FAIL");
        I2C_Stop();
        return 0xFFFF; // Return an error code
    }
    // Read MSB and LSB
    msb = I2C_Read(1); // Read MSB with ACK
    lsb = I2C_Read(0); // Read LSB with NACK
    I2C_Stop();
    // Debugging: Display raw MSB and LSB
    char debugBuffer[16];
    sprintf(debugBuffer, "MSB:%02X LSB:%02X", msb, lsb);
    LCD_clear();
    LCD_cursor_set(1, 1);
    LCD_write_string(debugBuffer);
    __delay_ms(1000);
    // Combine MSB and LSB into a single value
    return ((unsigned int)msb << 8) | lsb;
}
```