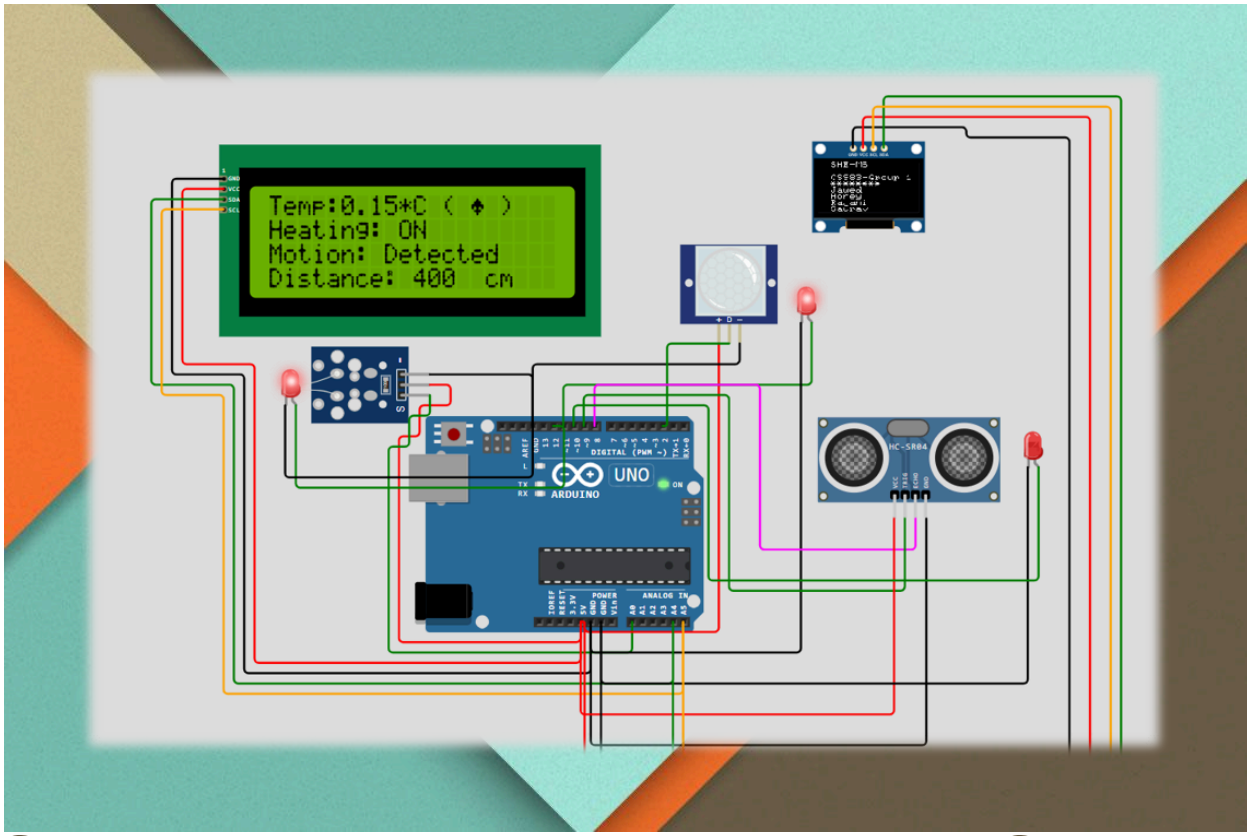


Smart Home Energy Management System

CS983: Embedded, Cyber Physical Systems and IoT Security

18.08.2024

Group 1

Member	Roll No.	Email address	Contribution
Honey Das	233560013	honeyd23@iitk.ac.in	25%
Mohammed Jawed	233560019	mjawed23@iitk.ac.in	25%
Rajani Kumari	233560044	rajanik23@iitk.ac.in	25%
Saurav Patra	23157058	sauravp23@iitk.ac.in	25%

Task-1

Task-2

Task 1

Problem Statement

Sita's modern apartment featured a Smart Home Energy Management System with temperature, PIR motion, and Ultrasonic sensors. This system intelligently adjusts heating, cooling, and lighting based on occupancy and environmental conditions, ensuring comfort year-round. It efficiently kept the apartment cool during hot summers and warm in chilly winters, all while reducing energy bills. Sita appreciated its seamless automation, making her home more comfortable and energy efficient.

Goals

- To create a circuit on a simulator with a temperature sensor, PIR motion sensor, and ultrasonic sensor so that she can simulate a realistic environment for monitoring and prediction using Arduino Uno.
- To ensure that the output readings from all sensors are visible either on the serial monitor or an LED display for monitoring and analysis.
- Ability to simulate conditions over specific days, predicting temperature changes based on sensor readings and providing a comprehensive virtual environment for testing and analysis.

Expected Functionality

1. Temperature Monitoring:

The temperature sensor has sliders to adjust the readings. By clicking on the temperature sensor, one can see the slider, whose value can be seen on the Serial Monitor or LCD Display. The LED should glow at a designated threshold that should be decided by the group.

2. Motion Detection:

On clicking on the PIR motion sensor, one will see the toggle button of Simulate motion after clicking on this; it should be visible on the Serial Monitor that the motion is detected. One has to make sure that triggering the sensor will output in 5 seconds and then go low again. The sensor will ignore any further output for the next 1.2 seconds (inhibit time), and then start sensing for motion again. An LED should glow whenever motion is detected.

3. Distance Measuring:

One has to select the sensor by clicking on it (while the simulation is running). A small popup window will open, now adjust the reading written on the "Editing Ultrasonic Distance" Sensor". By changing the readings, it must be visible on the serial monitor. (A deviation of 5-12 cm is viable). An LED should glow at a certain distance, as decided by the group.

Specifications

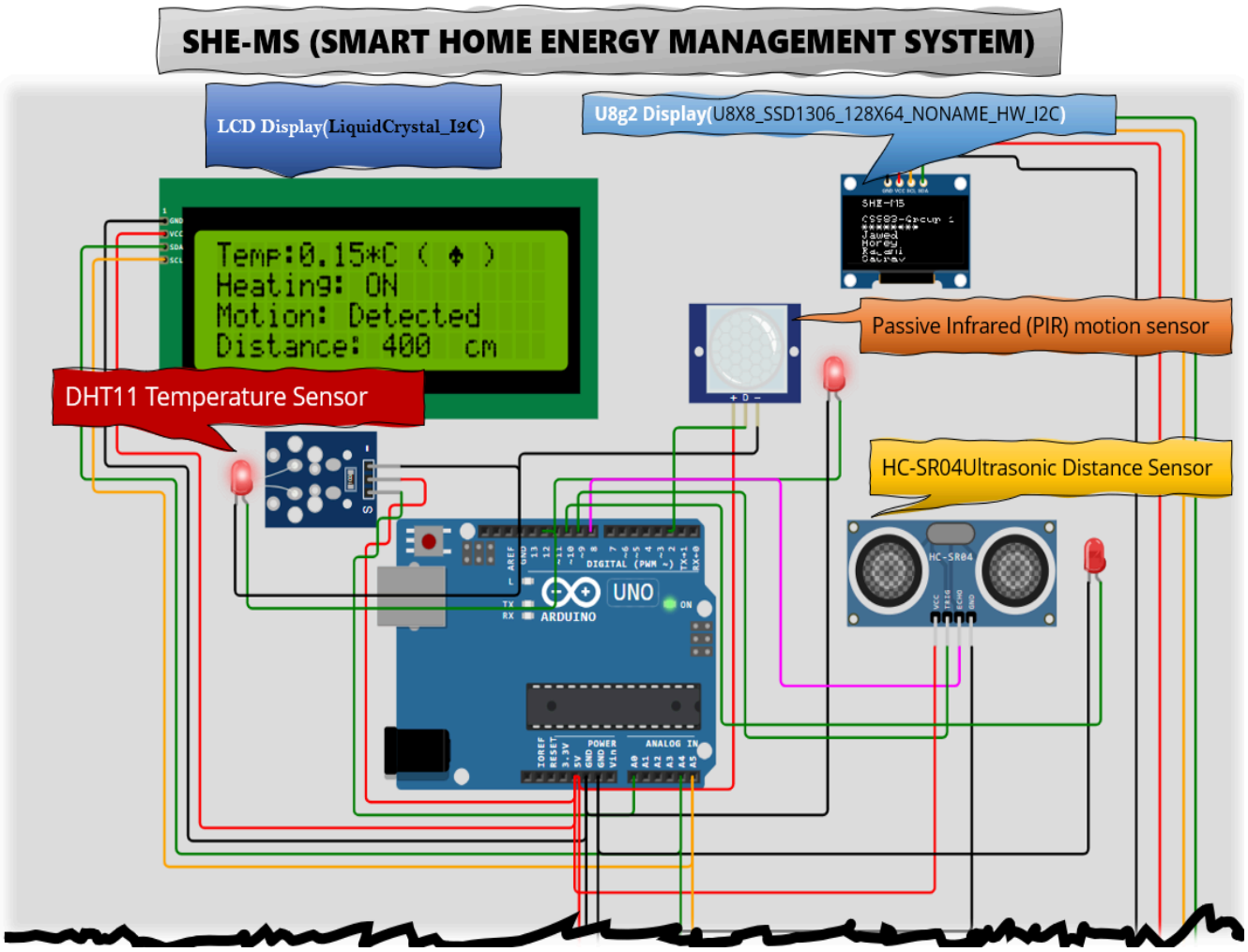
1. Arduino Uno
2. DHT11 Temperature Sensor
3. Passive Infrared (PIR) motion sensor
4. HC-SR04 Ultrasonic Distance Sensor
5. 20 X 4 LCD Display
6. 3 LEDs (1 for each sensor)
7. 128x64 OLED display

Project Link

https://wokwi.com/projects/406010597602082817

Circuit Diagram

Using the listed components in the requirement, we have setup the circuit as follows:



SHE-MS (SMART HOME ENERGY HOME MANAGEMENT SYSTEM)

Implementational logic and code flow

1. Including the libraries:

```
#include <LiquidCrystal_I2C.h>
#include <U8g2lib.h>
```

- LiquidCrystal_I2C.h : to control I2C displays with functions extremely similar to LiquidCrystal library
- U8g2lib.h : To control Monochrome LCD, OLED and eInk Library. It is a monochrome graphics library for embedded devices which we will use for controlling display for the SSD1306 module used in our project.

2. Initializing the displays:

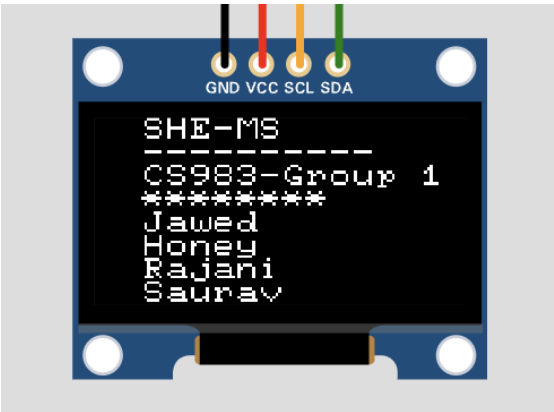
```
// Initialize the LCDs with their I2C addresses
LiquidCrystal_I2C lcd(0x27, 20, 4);
```

This line of code creates an object named `lcd` of type `LiquidCrystal_I2C` to represent a 20x4 character LCD display connected to the Arduino via I2C with the address 0x27. This object will be used to interact with the LCD display in your code, sending commands and data to control its behavior.

```
//for u8g display
U8X8_SSD1306_128X64_NONAME_HW_I2C displayWelcome(U8X8_PIN_NONE);
#define MENU_SIZE 8
char *menu[MENU_SIZE] = {"SHE-MS", "----------", "CS983-Group 1", "********", "Jawed", "Honey", "Rajani", "Saurav" };
```

These lines initialize an OLED display with the following specifications:

- **U8X8_SSD1306:** Indicates the type of OLED display (SSD1306).
- **128X64:** Specifies the display resolution (128 pixels wide, 64 pixels high).
- **NONAME_HW_I2C:** Indicates the type of interface (hardware I2C) and that no specific pin mapping is required.
- **displayWelcome:** Is the name given to the OLED display object.
- **U8X8_PIN_NONE:** Specifies that no additional pin configuration is needed for the hardware I2C interface.
- `#define MENU_SIZE 8`: Defines a constant `MENU_SIZE` with a value of 8 where each element is set to display the project name[SHE-MS: Smart Home Energy Management System], Group name[CS983-Group 1] and 4 team members[Jawed, Honey, Rajani, Saurav] respectively.



3. Defining the constants and variables:

```
const float BETA = 3950; // Beta coefficient for the NTC thermistor

#define ntcSensesor A0 // Temperature sensor connected to analog pin A0
#define ntcLED 12  // LED for temperature indication connected to digital pin 12

#define pirSensor 2 // PIR motion sensor connected to digital pin 2
#define pirLED 11   // LED for PIR motion indication connected to digital pin 11
int pirState = LOW; // Start with no motion detected
int pirValue = 0;   // Variable for reading the PIR sensor status
unsigned long motionStartTime = 0;
//unsigned long lastCheckTime = 0;   // Timestamp of the last check for inhibition period
//For Distance
#define trigPin 9
#define echoPin 8
#define ledPin 10
```

This code snippet primarily establishes constants and variable definitions for a project involving temperature, motion, and distance sensing.

Constants:

- **`BETA`:** This constant represents the beta coefficient for an NTC thermistor, a value used in temperature calculations. Its value of 3950 is a common approximation for many NTC thermistors, but the actual value can vary depending on the specific sensor.
- **`MENU_SIZE`:** This constant defines the number of items in a menu array, which is likely used for a user interface.

Pin Definitions:

- **`ntcSensesor`:** Assigns analog pin A0 for the NTC temperature sensor.
- **`ntcLED`:** Assigns digital pin 12 for controlling an LED related to temperature.

- **`pirSensor`:** Assigns digital pin 2 for the PIR motion sensor.
- **`pirLED`:** Assigns digital pin 11 for an LED indicating motion detection.
- **`trigPin`:** Assigns digital pin 9 for the trigger pin of an ultrasonic distance sensor.
- **`echoPin`:** Assigns digital pin 8 for the echo pin of the ultrasonic distance sensor.
- **`ledPin`:** Assigns digital pin 10 for an LED related to distance measurement.

Variables:

- **`pirState`:** Stores the current state of the PIR motion sensor (LOW for no motion, HIGH for motion).
- **`pirValue`:** Temporary variable for reading the PIR sensor value.
- **`motionStartTime`:** Tracks the start time of motion detection for timing purposes.

4. Defining a custom character for colling(snowflake) and heating(flame):

```
// Define the custom character (snowflake)
byte snowflake[8] = {
  B00000,
  B00100,
  B01110,
  B00100,
  B11111,
  B00100,
  B01110,
  B00000
};
// Define the custom character (flame)
byte flame[8] = {
  B00000,
  B00100,
  B01110,
  B11111,
  B10101,
  B01110,
  B00100,
  B00000
};
```

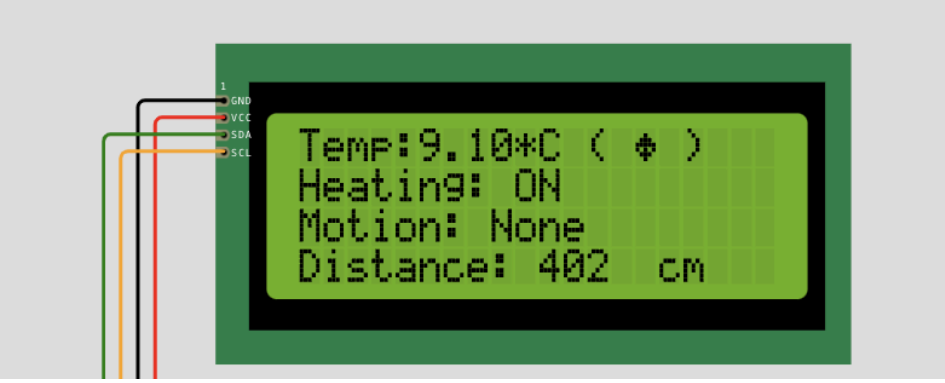
The 8-byte array represents the 8 rows of a 5x8 pixel character.

Bit Representation: Each byte contains 8 bits, where '1' represents a lit pixel and '0' represents an unlit pixel.

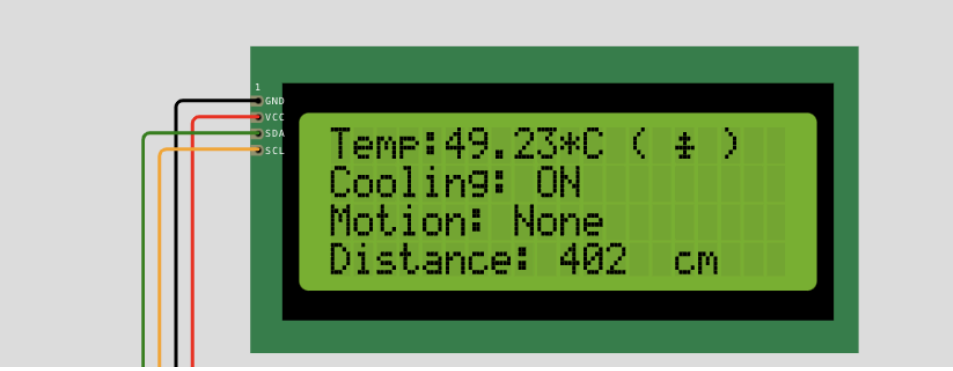
Character Creation: This defined character can be used with LCD libraries to display the snowflake symbol on the screen while cooling and flame symbol while

heating (since we would want to turn heating and cooling at the respective thresholds)

e.g. Flame while Heating:



e.g. Snowflake while Cooling:



5. Setup function to initialize Arduino board:

```
void setup() {

  displayWelcome.begin();
  displayWelcome.setPowerSave(0);
  displayWelcome.setFont(u8x8_font_pxplusibmcgathin_f);
  showWelcomeMessage();

  // put your setup code here, to run once:
  // Initialize the LCDs
  lcd.init();
  lcd.backlight();
  // Start the serial communication
  Serial.begin(9600);

  lcd.setCursor(2, 2);
  lcd.print("(CS983) Group 1");

  // Create the custom character in CGRAM
  lcd.createChar(0, snowflake);
  // Create the custom character in CGRAM
  lcd.createChar(1, flame);

  //ntc Temp Sensor
  pinMode(ntcSensesor, INPUT);
  pinMode(ntcLED, OUTPUT);

  //pir motion sensor
  pinMode(pirLED, OUTPUT); //LED as output
  pinMode(pirSensor, INPUT); //PIR Sensor as output

  // Define HC-SR04 pins
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(ledPin, OUTPUT);
  // Display the custom character
  // lcd.setCursor(0, 0); // Set cursor to the first column of the first row
  //lcd.write(byte(0)); // Display the snowflake character
  //Serial.println("Motion detected!");
}
```

The setup() function is executed once when the Arduino board starts running. It's used to initialize hardware, variables, and other resources before the main program loop begins.

- **OLED Display Initialization:**
  - `displayWelcome.begin();`: Initializes the OLED display with the specified parameters.
  - `displayWelcome.setPowerSave(0);`: Turns off power saving mode for the OLED.

    - `displayWelcome.setFont(u8x8_font_pxplusibmcgathin_f);`: Sets the font for the OLED display.
    - `showWelcomeMessage();`: Calls a function (likely defined elsewhere) to display a welcome message on the OLED.
- **LCD Initialization:**
    - `lcd.init();`: Initializes the LCD display.
    - `lcd.backlight();`: Turns on the LCD backlight.
    - `Serial.begin(9600);`: Initializes serial communication at 9600 baud rate.
- **LCD Cursor Positioning:**
    - `lcd.setCursor(2, 2);`: Sets the cursor position to the third row, second column of the LCD.
    - `lcd.print("(CS983) Group 1");`: Prints the text "(CS983) Group 1" on the LCD at the specified position.
- **Custom Character Creation:**
    - `lcd.createChar(0, snowflake);`: Creates a custom character named "snowflake" at position 0 in the LCD's character generator RAM.
    - `lcd.createChar(1, flame);`: Creates a custom character named "flame" at position 1 in the LCD's character generator RAM.
- **Sensor Pin Initialization:**
    - `pinMode(ntcSensesor, INPUT);`: Configures the NTC temperature sensor pin as an input.
    - `pinMode(ntcLED, OUTPUT);`: Configures the NTC temperature LED pin as an output.
    - `pinMode(pirLED, OUTPUT);`: Configures the PIR motion sensor LED pin as an output.
    - `pinMode(pirSensor, INPUT);`: Configures the PIR motion sensor pin as an input.
    - `pinMode(trigPin, OUTPUT);`: Configures the ultrasonic sensor trigger pin as an output.
    - `pinMode(echoPin, INPUT);`: Configures the ultrasonic sensor echo pin as an input.
    - `pinMode(ledPin, OUTPUT);`: Configures the ultrasonic sensor LED pin as an output.

6. Loop function to read and process sensor values:

```
void loop() {

  delay(2000); // Delay for 1000 milliseconds (1 second)
  clearRow(2);
  lcd.setCursor(0, 2);
  lcd.print("Motion: None    ");
  //Temperature Sensor , NTC Temperature Sensor
  ntcTemperatureSensor();
  //PIR Motion Sensor , Passive Infrared (PIR) motion sensor.
  pirMotionSensor();
  // Distance Sensor,  HC-SR04 Ultrasonic Distance Sensor.
  distanceMeasure();

}
```

The **`loop()`** function is the core of the Arduino program, executing repeatedly after the setup() function completes. In this case, it's responsible for the main logic of the smart home system, including sensor readings, data processing, and output display.

- **Delay:**
  - **`delay(2000);`** introduces a 2-second delay at the beginning of each loop iteration. This might be for timing purposes or to avoid excessive CPU usage.
- **LCD Clearing:**
  - **`clearRow(2);`**: Clears the second row of the LCD display.
  - **`lcd.setCursor(0, 2);`**: Sets the cursor to the beginning of the second row.
  - **`lcd.print("Motion: None ");`**: Displays "Motion: None" on the second row of the LCD.
- **Sensor Readings:**
  - **`ntcTemperatureSensor();`**: Calls a function to read and process temperature sensor data.
  - **`pirMotionSensor();`**: Calls a function to read and process PIR motion sensor data.
  - **`distanceMeasure();`**: Calls a function to read and process ultrasonic distance sensor data.

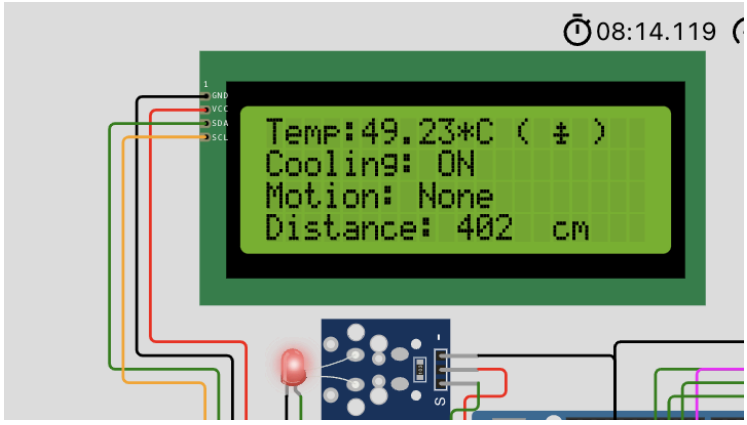
7. Defining the functioning of NTC Temperature sensor:

```
/**
  NTC Temperature Sensor
  • Cooling: start cooling if the temperature goes above 76°F (24°C) and aim to cool down to 73°F (
  • Heating: start heating if the temperature drops below 70°F (21°C) and aim to warm up to 71°F (2
  LED will glow when either Cooling or Heating is ON
  Everthing will be off when Temp is between 21-24 (Threshold)
**/
//Temperature Sensor
void ntcTemperatureSensor()
{
  // Temperature Sensor Code
  int val = analogRead(ntcSensesor);
  float temp = 1 / (log(1 / (1023.0 / val - 1)) / BETA + 1.0 / 298.15) - 273.15;
  lcd.setCursor(0, 0);
  lcd.print("Temp:");
  lcd.print(temp);
  lcd.print("*C");

  if (temp > 24) {
    digitalWrite(ntcLED, HIGH);

    // Clear the 2nd  row
    //clearRow(2);
    // Display the snowflake character
    lcd.print(" ( ");
    lcd.write(byte(0));
    lcd.print(" )");
    lcd.setCursor(0, 1);
    //lcd.print("(cool)");
    lcd.print("Cooling: ON ");
    //lcd.setCursor(0, 3);
    //lcd.print("Heating: OFF");

  }
  else if (temp < 21) {
    digitalWrite(ntcLED, HIGH);
    //lcd.setCursor(0, 2);
    //lcd.print("Cooling: OFF");
    //lcd.setCursor(0, 3);
    // Display the flame character
    lcd.print(" ( ");
    lcd.write(byte(1));
    lcd.print(" )");
    lcd.setCursor(0, 1);
    //lcd.print("(heat)");
    lcd.print("Heating: ON ");

  } else {
    digitalWrite(ntcLED, LOW);
    lcd.print("       ");
    //clearRow(1);
    lcd.setCursor(0, 1);
    lcd.print("            ");
    //lcd.setCursor(0, 2);
    //lcd.print("Cooling: OFF");
    //lcd.setCursor(0, 3);
    //lcd.print("Heating: OFF");
  }

  //delay(2000);
}
```

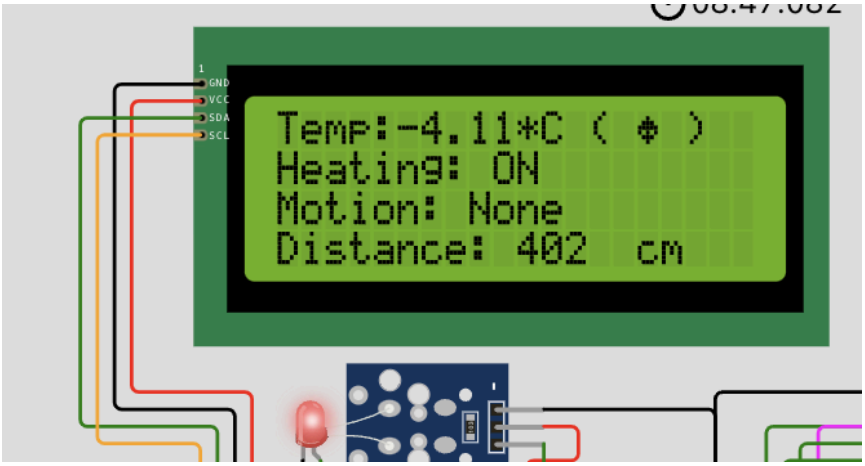
The `ntcTemperatureSensor()` function is responsible for reading the temperature from an NTC thermistor, converting the analog reading to temperature, displaying the temperature on an LCD, and controlling an LED based on temperature thresholds. The code defines a basic temperature control logic based on fixed thresholds as decided by the group. The LCD display is used to provide visual feedback about the temperature and heating/cooling status. The function assumes that the necessary libraries for LCD control and temperature calculations are included.

- **Reading Sensor Value:**
  - `int val = analogRead(ntcSensesor);`: Reads the analog value from the NTC thermistor connected to pin `ntcSensesor`.
- **Temperature Calculation:**
  - `float temp = 1 / (log(1 / (1023.0 / val - 1)) / BETA + 1.0 / 298.15) - 273.15;`: Converts the analog value to temperature in Celsius using the Steinhart-Hart equation and the provided BETA coefficient.
- **Display Temperature:**
  - `lcd.setCursor(0, 0);`: Sets the cursor to the first row, first column of the LCD.
  - `lcd.print("Temp:");`: Prints the label "Temp:" on the LCD.
  - `lcd.print(temp);`: Prints the calculated temperature value on the LCD.
  - `lcd.print("*C");`: Prints the degree Celsius symbol.
- **Temperature Control:**
  - **Cooling:** If the temperature is above 24°C (76°F), the `ntcLED` is **turned on** to indicate cooling, and the LCD displays a snowflake symbol.
  - **Heating:** If the temperature is below 21°C (70°F), the `ntcLED` is **turned on** to indicate heating, and the LCD displays a flame symbol.
  - **Normal:** If the temperature is between 21°C and 24°C, the `ntcLED` is **turned off**, and the LCD displays spaces to clear previous messages.

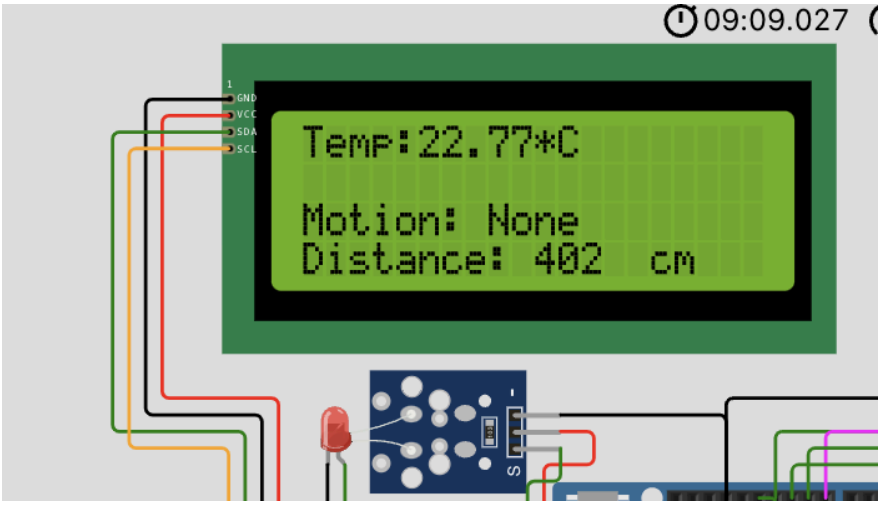
e.g. LED ON and LCD indicating Cooling ON at temp > 24°C



e.g. LED ON and LCD indicating Heating ON at temp < 21°C



e.g. LED off and no Heating/Cooling at temp between 21°C and 24°C



8. Defining the functioning of Passive Infrared(PIR) sensor:

```
/**
  Passive Infrared (PIR) motion sensor.
  the sensor will output in 5 seconds and then go low again. The
  sensor will ignore any further output for the next 1.2 seconds (inhibit time), and then
  start sensing for motion again. An LED should glow whenever motion is detected.
  LED will display for MOtion detected.
**/
// PIR Motion Sensor
void pirMotionSensor()
{
  // PIR Motion Detection Code
  pirValue = digitalRead(pirSensor); // Read the PIR sensor status

  if (pirValue == HIGH && pirState == LOW) {
    // Motion detected

    digitalWrite(pirLED, HIGH);
    pirState = HIGH;
    motionStartTime = millis();
    lcd.setCursor(0, 2);
    // clearRow(2);
    lcd.println("Motion: Detected");
    Serial.println("Motion detected!");
  }


  if (pirState == HIGH) {
    if (millis() - motionStartTime > 5000) {
      // 5 seconds have passed since motion was detected

      pirState = LOW;
      digitalWrite(pirLED, LOW);
      lcd.setCursor(0, 2);
      // clearRow(2);
      lcd.print("Motion: None    ");
      Serial.println("Motion ended!");
    }
  }
```

```
  // Ignore further motion for 1.2 seconds after motion ends
  static unsigned long lastCheckTime = 0;
  if (millis() - lastCheckTime >= 100) {
    if (pirState == LOW && millis() - motionStartTime < 6200) {
      // Do nothing (inhibit time)
    } else {
      lastCheckTime = millis();
    }
  }


}
```

The `pirMotionSensor()` function implements a basic PIR motion detection system.

The code effectively monitors the PIR sensor for changes in state, indicating the presence or absence of motion. Upon detecting motion, the function activates an LED and displays a

corresponding message on the LCD. To prevent false triggers, the code incorporates a 5-second timeout after which the motion is considered ended. Additionally, an inhibit period of 1.2 seconds is implemented to ignore subsequent motion detections immediately after a motion event.

- **Reading Sensor Value:**

`pirValue = digitalRead(pirSensor);` reads the digital value from the PIR sensor pin.

- **Motion Detection:**

`if (pirValue == HIGH && pirState == LOW):` Checks if motion is detected (sensor output is high and previous state was low).

Turns the LED on (`digitalWrite(pirLED, HIGH)`).

- Sets the `pirState` to HIGH to indicate motion detected.
- Records the start time of motion detection (`motionStartTime = millis()`).
- Updates the LCD display to indicate motion detected.

- **Motion Timeout:**

`if (pirState == HIGH):` Checks if motion is currently detected.

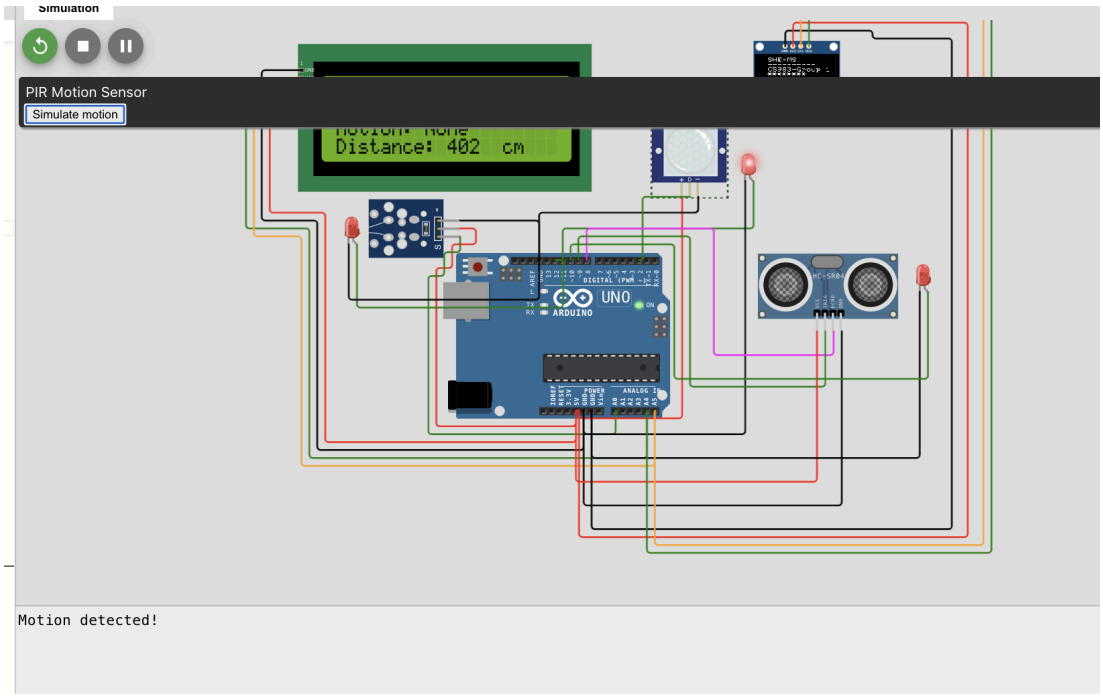
  - If 5 seconds have passed since motion started (`millis() - motionStartTime > 5000`), the motion is considered ended.
  - Turns the LED off (`digitalWrite(pirLED, LOW)`).
  - Updates the LCD display to indicate no motion.
  - Resets the `pirState` to LOW.

- **Inhibition period:**

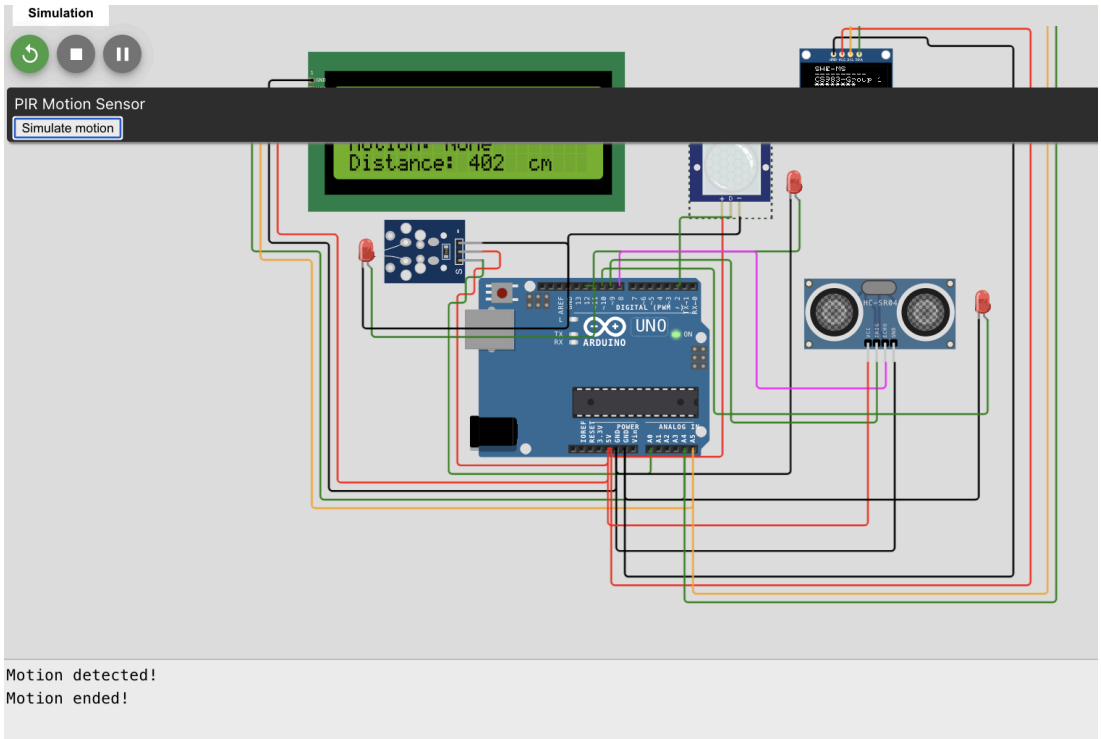
```
if (millis() - lastCheckTime >= 100) {
   if (pirState == LOW && millis() - motionStartTime < 6200)
```

Adds an inhibit period of 1.2 seconds to ignore subsequent motion detections immediately after a motion event.

e.g. LED on after 5s of motion detection



e.g. LED off after motion ends



9. Defining the functioning of HC-SR04 Ultrasonic Distance Sensor:

```
/**
  HC-SR04 Ultrasonic Distance Sensor.
  To change the distance while the simulation is running, click on the HC-SR04 drawing in the diagram and use the slider to set the distance value.
  You can choose any value between 2cm and 400cm.
  To LED to glow – threshold distance 5-12 CM
  Distance in cm will be visible on LED screen also
**/
//Code for Distance Sensor
void distanceMeasure()
{
  // Measure Distance
  long duration, distance;
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Measure the pulse width
  duration = pulseIn(echoPin, HIGH);

  // Convert duration to distance in centimeters
  float distanceCm = (duration / 2.0) * 0.0344;
  // Convert distance to meters
  //distance = distanceCm / 100.0;
  distance = distanceCm;


  // Display distance on Serial Monitor
  lcd.setCursor(0, 3);
  lcd.print("Distance: ");
  //Serial.print("Distance: ");
  //Serial.print(distance);
  //Serial.println(" cm");
  lcd.println(distance);
  lcd.println("cm");

  // Glow LED if distance is within a certain range
  if (distance >= 5 && distance <= 12) {
    digitalWrite(ledPin, HIGH);
  } else {
    digitalWrite(ledPin, LOW);
  }
  //Code end for Distance
  // delay(200);
}
```

The `distanceMeasure()` function is responsible for measuring the distance to an object using an HC-SR04 ultrasonic sensor, displaying the measured distance on an LCD, and controlling an LED based on the distance value.

Triggering the Sensor:

- A short pulse is sent to the trigger pin of the HC-SR04 sensor to initiate a measurement.

Measuring Pulse Width:

- The function measures the duration of the echo pulse returned by the sensor.
- The pulse width is converted into distance using the known speed of sound.

Distance Calculation:

Ultrasonic Pulse Travel Time: An ultrasonic sensor sends out a pulse and measures the time it takes for the echo to return. This time is stored in the duration variable.

Round Trip: The pulse travels to the object and back, so the measured time corresponds to the round trip distance. To get the distance to the object, you need to halve this duration.

Speed of Sound: The speed of sound in air is approximately 0.0344 cm/μs (or 344 meters/second). This means that sound travels 0.0344 centimeters per microsecond.

Formula for Distance: To convert the duration of the pulse to distance in centimeters - Divide Duration by 2: Since the duration includes the travel to and from the object, dividing by 2 gives the one-way distance.

Multiply by Speed of Sound: The speed of sound in centimeters per microsecond is 0.0344. Multiplying by this factor converts the time (in microseconds) into distance (in centimeters).

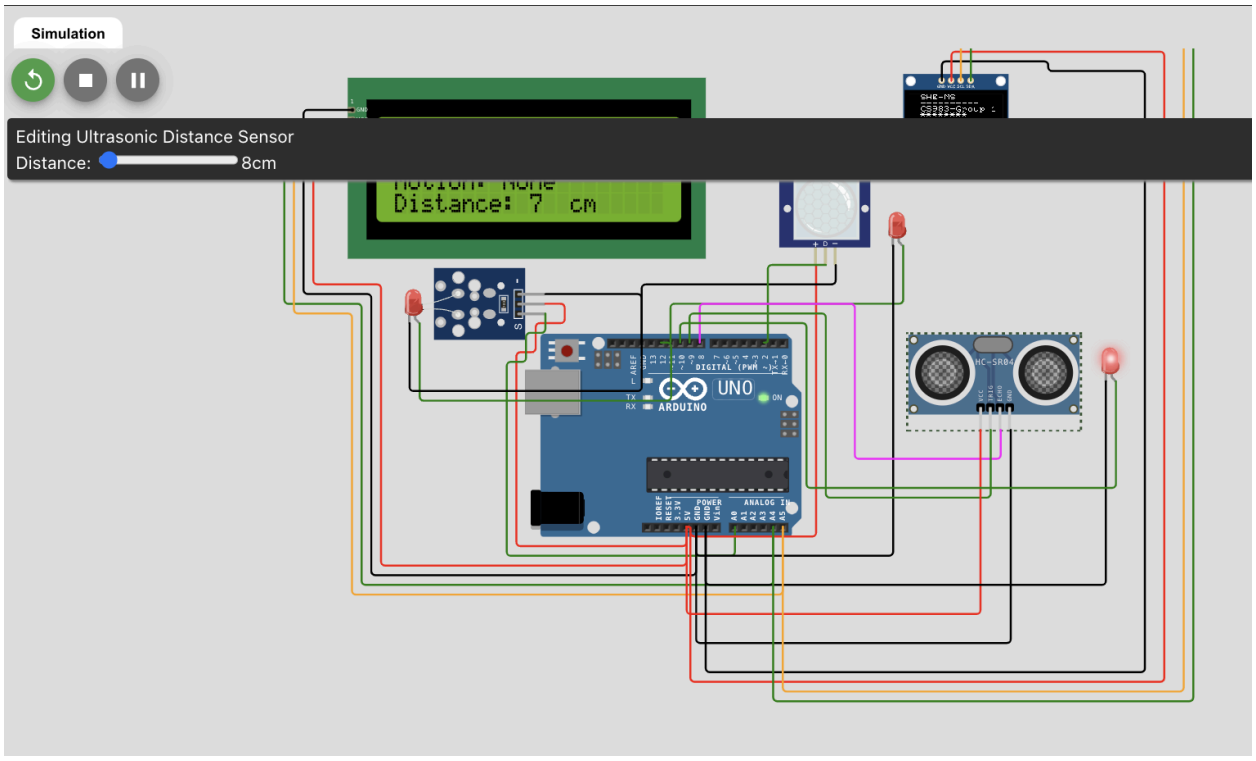
Displaying Distance:

- The measured distance is displayed on the LCD.
- The function also includes commented-out code for potentially displaying the distance on the serial monitor.

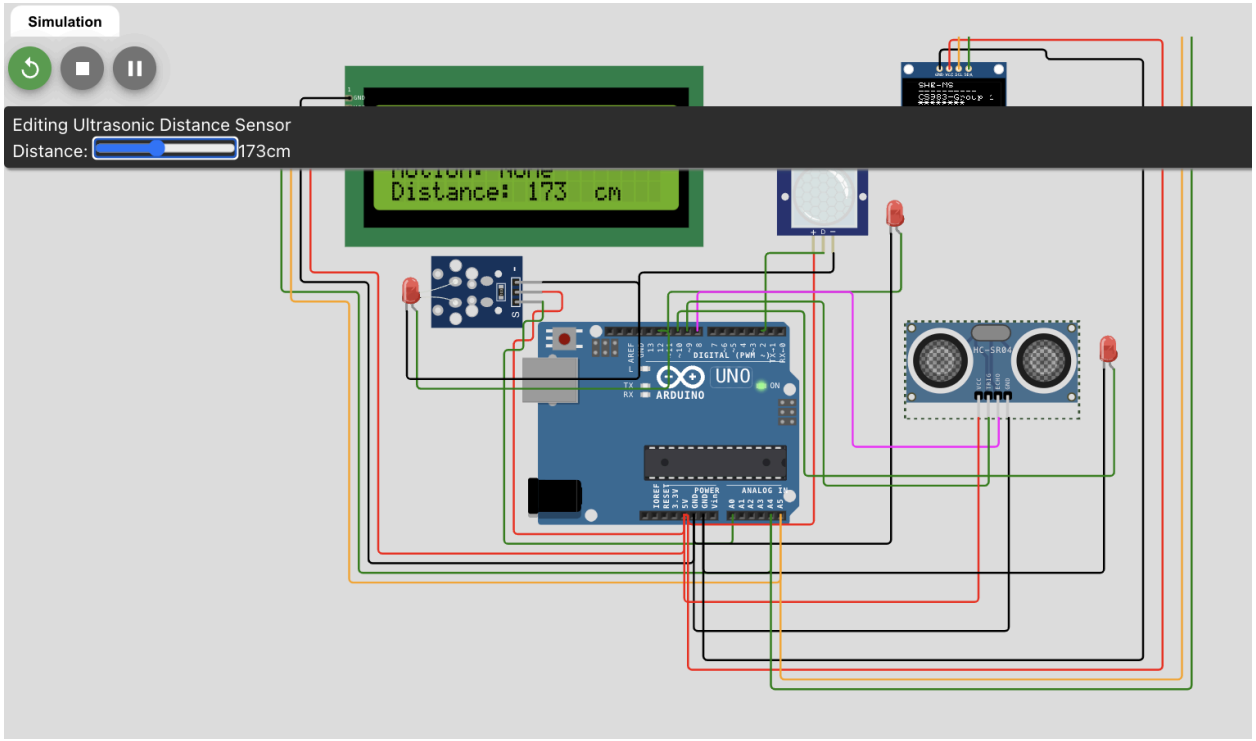
LED Control:

- If the measured distance is within a specified range (5-12 cm), the LED is turned on; otherwise, it's turned off.

e.g. LED on when distance is between 5-12 cm



e.g. LED off when distance is greater is outside 5-12 cm range



10. Clearing off LCD:

```
/**
  Method to Clear any specified Row of LCD screen
**/

void clearRow(int row) {
  // Set cursor to the start of the specified row
  lcd.setCursor(0, row);

  // Write spaces to clear the row
  for (int i = 0; i < 20; i++) { // Assuming a 20x4 display
    lcd.write(' ');
  }
}
```

The `clearRow` function is designed to clear a specific row on an LCD display.

- **Function Definition:**
  - `void clearRow(int row)`: Defines a function named `clearRow` that takes an integer `row` as input. The function returns no value (void).
- **Setting Cursor Position:**
  - `lcd.setCursor(0, row);`: Sets the cursor to the beginning of the specified row. The first parameter (0) indicates the starting column, and the second parameter (`row`) specifies the row number.
- **Clearing the Row:**
  - `for (int i = 0; i < 20; i++) { lcd.write(' '); }`: A loop iterates 20 times, writing a space character to the current cursor position in each iteration. This effectively overwrites the existing characters on the row with spaces, clearing the row.

11. Clearing off OLED and preparing for new content:

```
/**
  Display Welcome message
**/
/**
   Clear display and show the menu.
*/
void showWelcomeMessage() {

  displayWelcome.clearDisplay();
  // show menu items:
  for (int i = 0; i < MENU_SIZE; i++) {
    displayWelcome.drawString(2, i, menu[i]);
  }
  // display.setCursor(0,0);
  //display.print('>');
}
```

The `showWelcomeMessage()` function is responsible for clearing the display and displaying a menu on the OLED display. The `displayWelcome` object is an instance of an OLED display library, likely `U8G2`

- **Clear Display:**
  - `displayWelcome.clearDisplay();`: Clears the entire OLED display, preparing it for new content.
- **Display Menu Items:**
  - `for (int i = 0; i < MENU_SIZE; i++) { displayWelcome.drawString(2, i, menu[i]); }`: Iterates through the `menu` array, displaying each menu item on a new line of the OLED display.
    - `displayWelcome.drawString(2, i, menu[i])`: Draws the `i`-th menu item at the specified x and y coordinates (2, i) on the OLED display.

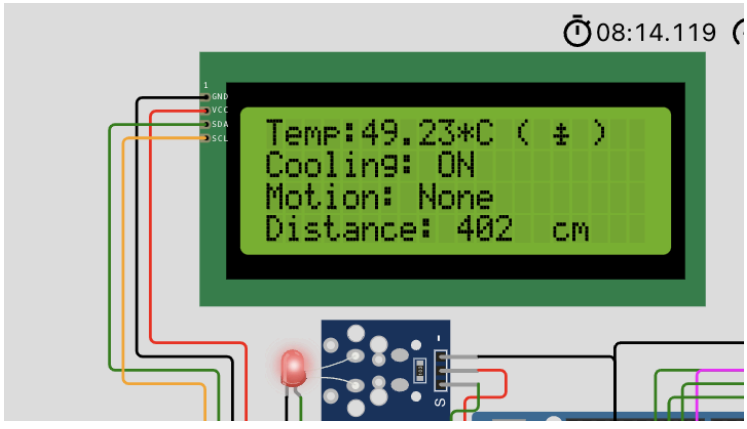
Expected Functionality Results

I. Use case 1:

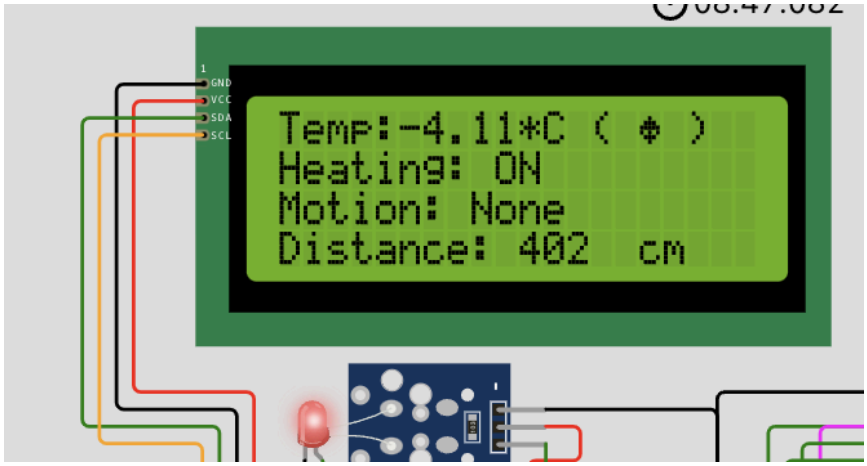
Monitoring room temperature and turning LED on/off based on thresholds. Thereby turning heating/cooling on.

Simulation:

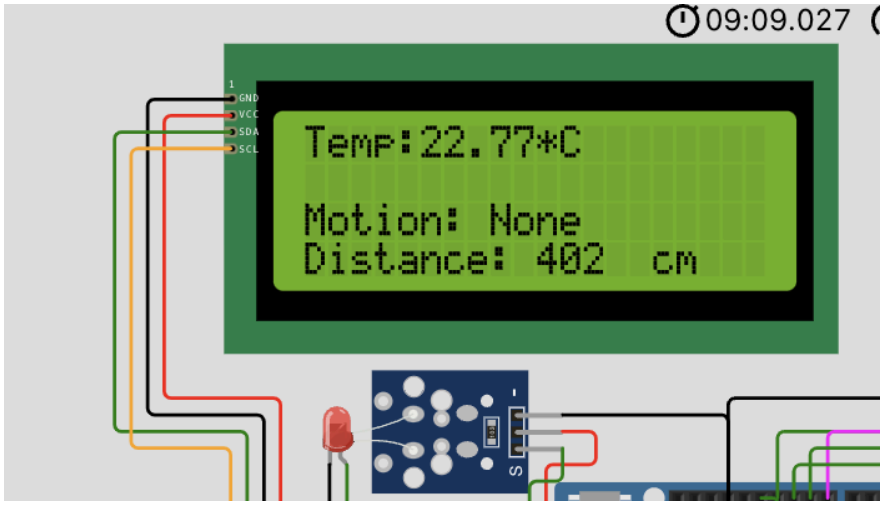
1. LED ON and LCD indicating Cooling ON at temp > 24°C



2. LED ON and LCD indicating Heating ON at temp < 21°C



3. LED off and no Heating/Cooling at temp between 21°C and 24°C

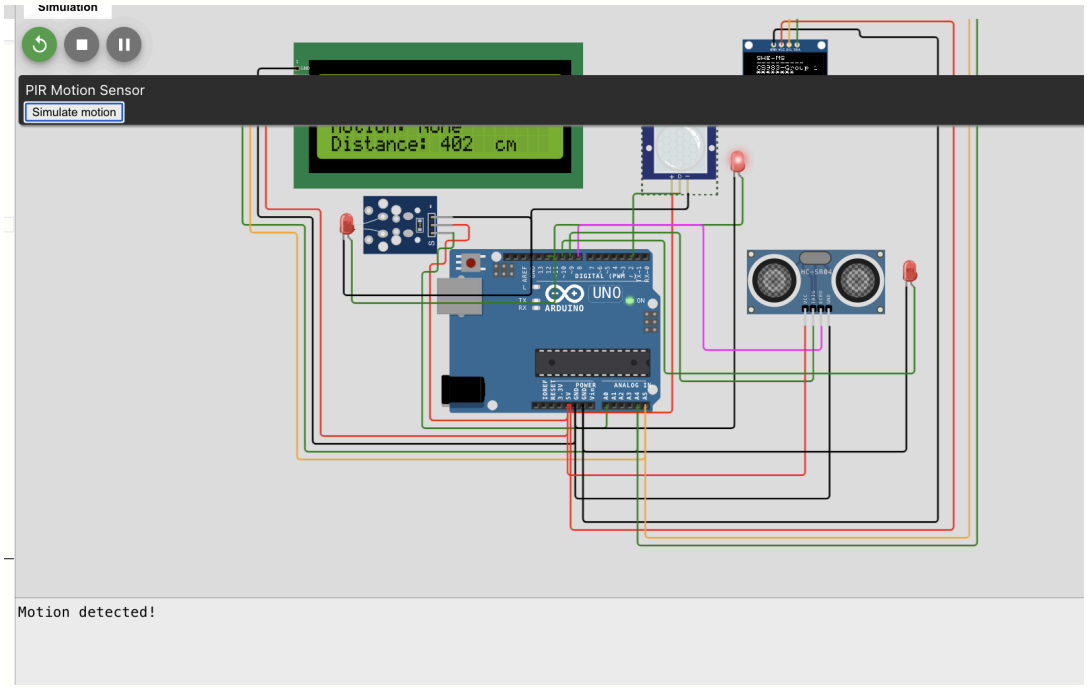


II. Use case 2:

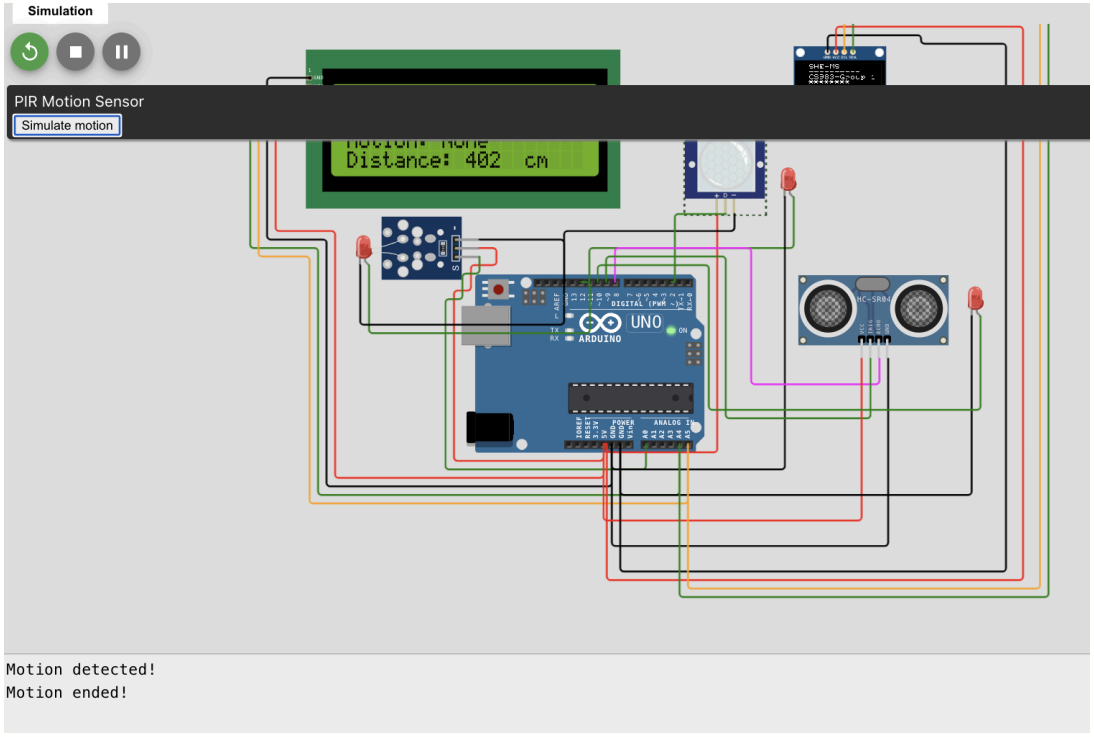
Monitoring motion and turning LED on after 5s.

Simulation:

1. LED on after 5s of motion detection



2. LED off after motion ends

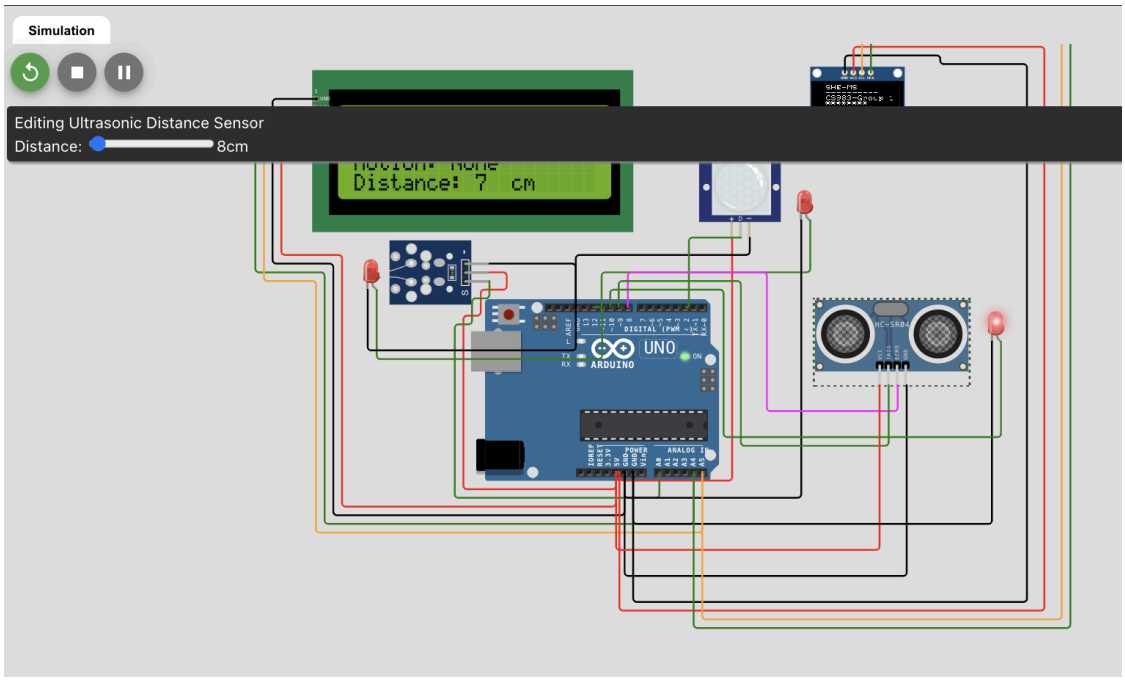


III. Use case 3:

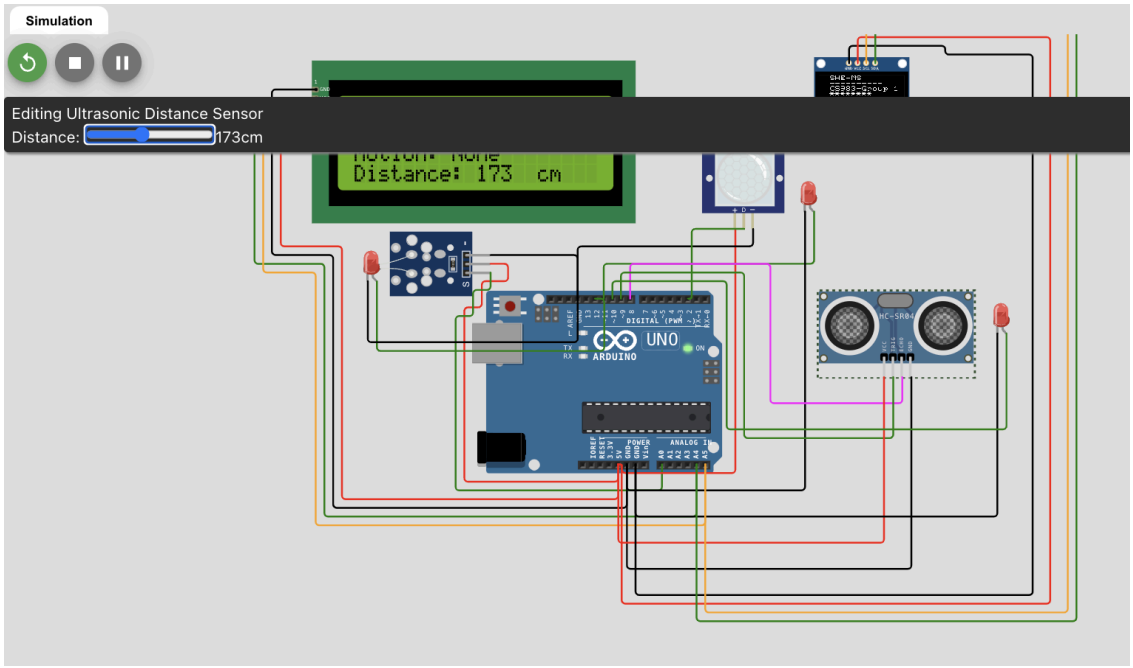
Monitoring distance and turning LED on between a certain distance range.

Simulation:

1. LED on when distance is between 5-12 cm



2. LED off when distance is greater is outside 5-12 cm range



Component Specifications

1. **Arduino Uno** - Powered by the ATmega328P microcontroller, it offers a balance of processing power and low cost. 14 digital input/output pins and 6 analog inputs provide ample flexibility for connecting sensors, actuators, and other components.Can be powered by USB or an external power supply, making it convenient for various projects. Uses the Arduino IDE, a user-friendly environment for writing and uploading code.

   More details at: https://docs.wokwi.com/parts/wokwi-arduino-uno

2. **DHT11 Temperature Sensor** - The DHT11 is a basic, low-cost digital sensor that measures both temperature and humidity. Outputs data in a digital format, simplifying interfacing with microcontrollers. Economical option for basic temperature and humidity monitoring. Requires only a single data pin for communication.

   More details at: https://docs.wokwi.com/parts/wokwi-ntc-temperature-sensor

3. **Passive Infrared (PIR) motion sensor** - A Passive Infrared (PIR) motion sensor detects changes in infrared radiation emitted by objects within its field of view. The core component is a pyroelectric sensor, which generates an electrical charge when exposed to changing infrared radiation. The sensor typically has multiple sensing elements arranged in pairs. When an object with a different temperature (like a human) enters the sensor's field of view, it emits different amounts of infrared radiation to each sensor, creating a voltage difference. The generated electrical signal is amplified and processed to determine if it represents actual motion or just background noise.

   More details at: https://docs.wokwi.com/parts/wokwi-pir-motion-sensor

4. **HC-SR04 Ultrasonic Distance Senso**r - The HC-SR04 is a popular ultrasonic distance sensor capable of measuring distances between 2cm and 400cm with an accuracy of about 3mm. It's widely used in robotics, automation, and other applications requiring non-contact distance measurement. A short pulse (typically 10 microseconds) is sent to the TRIG pin of the sensor. The sensor emits eight 40kHz ultrasonic pulses.

   More details at: https://docs.wokwi.com/parts/wokwi-hc-sr04

5. **20 X 4 LCD Display** - A 20x4 LCD display is a character-based display that can show 4 lines of text, with each line capable of displaying 20 characters. It's a common component in embedded systems and electronics projects due to its readability and simplicity.

   More details at: https://docs.wokwi.com/parts/wokwi-lcd2004

6. **LED** - An LED (Light-Emitting Diode) is a semiconductor device that converts electrical energy into light energy. When a current flows through the LED, electrons recombine with holes, releasing energy in the form of photons, which is emitted as light.

   More details at: https://docs.wokwi.com/parts/wokwi-led

7. **128x64 OLED display** - A 128x64 OLED display is a compact, energy-efficient display capable of showing high-quality images and text. It's widely used in various electronic devices due to its superior contrast, wide viewing angles, and fast response times.

   More details at: https://docs.wokwi.com/parts/board-ssd1306

Task 2

Problem Statement

Given the short duration of the project, we have collected a dataset that consists of humidity and temperature values along with the time stamps. You have access to a dataset containing temperature and humidity values, each associated with timestamps. Your objective is to categorize each day based on predefined criteria:

- Hot (temperature > 30°C and humidity ≤ 70%)
- Hot and Humid (temperature > 30°C and humidity > 70%)
- Cold (temperature ≤ 30°C and humidity > 30%)
- Cold and dry (temperature ≤ 30°C and humidity ≤ 30%).

Goals

Your task involves accurately classifying a given date according to these predefined categories and their respective thresholds. You also have to examine the dataset using graphs or plots to analyze the changes in weather patterns over the years.

Dataset

Dataset: https://drive.google.com/file/d/1wtGzFLtIW80mAPi-Pij8MQJN3Me6823v/view

Deliverable

Please see the attachment IOT-Group-1-Sol.ipynb

Implementational logic and code flow

1. Import the modules and load the dataset:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display
from matplotlib.colors import ListedColormap, BoundaryNorm
import calplot
import warnings
import numpy as np
import calendar
# Load the dataset
file_path = './Temp_and_humidity_dataset.csv'
data = pd.read_csv(file_path)
# Display the first few rows of the dataset to understand its structure
data.head()
```

The code aims to import several Python libraries and load the CSV file containing temperature and humidity data into a pandas DataFrame for further analysis and processing.

- **Import libraries:**
  - **pandas:** For data manipulation and analysis.
  - **matplotlib.pyplot:** For creating static, animated, and interactive visualizations.
  - **seaborn:** For creating informative and attractive statistical graphics.
  - **IPython.display:** For displaying objects within the Jupyter Notebook environment.
  - **matplotlib.colors:** For color mapping and normalization.
  - **calplot:** For visualizing data over a calendar.
  - **warnings:** For handling warnings.
  - **numpy:** For numerical operations.

- **Specify file path:** `file_path = './Temp_and_humidity_dataset.csv'` sets the file path to the CSV file containing the temperature and humidity data.

- **Load data into DataFrame:** `data = pd.read_csv(file_path)` reads the CSV file into a pandas DataFrame named `data`. This DataFrame is a two-dimensional tabular data structure with labeled axes (rows and columns).
- **Display initial data:** `data.head()` prints the first few rows of the DataFrame to provide a quick overview of the data structure and content.

This is how the loaded data will look like:

	DATETIME	TEMPERATURE	HUMIDITY
0	2015-01-01 00:00:00	19.0	75.0
1	2015-01-01 01:00:00	19.0	77.0
2	2015-01-01 02:00:00	19.0	78.0
3	2015-01-01 03:00:00	19.0	80.0
4	2015-01-01 04:00:00	19.0	81.0

2. Validate whether DATETIME column is in the expected datetime format

```
# Check if DATETIME column is in the expected datetime format
def check_datetime_format(df):
    try:
        df['DATETIME'] = pd.to_datetime(df['DATETIME'], format='%Y-%m-%d %H:%M:%S')
        print("DATETIME column is in the correct format.")
    except Exception as e:
        print(f"Error with DATETIME column format: {e}")
```

```
# Run the checks
check_datetime_format(data)
```

This code snippet defines a function `check_datetime_format` to verify if the 'DATETIME' column in a given DataFrame is in the expected format ('%Y-%m-%d %H:%M:%S').

- **Function Definition:**
  - `def check_datetime_format(df):` defines a function named `check_datetime_format` that takes a DataFrame `df` as input.
- **Error Handling:**
  - `try-except` block is used to handle potential exceptions during the datetime conversion.
- **Datetime Conversion:**
  - `df['DATETIME'] = pd.to_datetime(df['DATETIME'], format='%Y-%m-%d %H:%M:%S')`: Attempts to convert the 'DATETIME' column to datetime objects using the specified format.
  - If successful, prints a message indicating the correct format.

- **Error Handling:**
    - If an exception occurs during conversion, it prints an error message indicating the issue.

3. Convert 'DATETIME' to DateTime Object and Extract Date:

```
# Convert 'DATETIME' to datetime object and extract date
data['DATETIME'] = pd.to_datetime(data['DATETIME'])
data['DATE'] = data['DATETIME'].dt.date
```

This code snippet manipulates a pandas DataFrame named `data` to extract the date information from an existing column named "DATETIME".

- **Convert 'DATETIME' to datetime object:**

  `data['DATETIME'] = pd.to_datetime(data['DATETIME'])`: This line converts the values in the "DATETIME" column of the DataFrame to pandas datetime objects. Datetime objects represent dates and times with additional functionalities for manipulation.

- **Extract Date:**

  `data['DATE'] = data['DATETIME'].dt.date`: This line extracts the date information (excluding time) from the datetime objects in the "DATETIME" column. The result is stored in a new column named "DATE".

4. Validate whether TEMPERATURE and HUMIDITY are in the expected format

```
# Check TEMPERATURE and HUMIDITY for numeric values (float)
def check_numeric_values(df, columns):
    for column in columns:
        if not pd.api.types.is_numeric_dtype(df[column]):
            print(f"Warning: {column} column does not contain numeric values. Checking for any non-numeric values...")
            non_numeric_values = df[~df[column].apply(pd.to_numeric, errors='coerce').notnull()]
            if not non_numeric_values.empty:
                print(f"Non-numeric values found in {column}:")
                print(non_numeric_values[column].unique())
        else:
            print(f"{column} column contains numeric values.")
```

```
# Run the checks
check_numeric_values(data, ['TEMPERATURE', 'HUMIDITY'])
```

This code snippet verifies if the 'TEMPERATURE' and 'HUMIDITY' columns in a DataFrame contain numeric values. It identifies and prints any non-numeric values found in these columns.

- **Function Definition:**
  - `def check_numeric_values(df, columns):` defines a function to check for numeric values in specified columns of a DataFrame.
- **Iterate Through Columns:**
  - `for column in columns:` iterates over each column in the provided list of columns.
- **Check Numeric Data Type:**
  - `if not pd.api.types.is_numeric_dtype(df[column]):` checks if the column's data type is not numeric.
- **Identify Non-Numeric Values:**
  - `non_numeric_values = df[~df[column].apply(pd.to_numeric, errors='coerce').notnull()]` creates a DataFrame containing rows with non-numeric values in the current column.
- **Print Results:**
  - If non-numeric values are found, prints a warning and the unique non-numeric values.
  - If all values are numeric, prints a message indicating the column contains numeric values.

5. Group by Date and Calculate Average Temperature and Humidity

```
# Group by date and calculate average temperature and humidity
daily_data = data.groupby('DATE').agg({'TEMPERATURE': 'mean', 'HUMIDITY': 'mean'}).reset_index()
```

The primary goal of this code is to summarize the temperature and humidity data on a daily basis. By grouping the data by date and calculating the mean values, it provides a condensed view of the overall trends. The provided code performs the following steps:

- **Group by date:** It groups the original DataFrame `data` by the 'DATE' column. This creates groups for each unique date in the dataset.
- **Calculate averages:** For each group (date), it calculates the mean of the 'TEMPERATURE' and 'HUMIDITY' columns using the `agg` function.
- **Create new DataFrame:** The results of the aggregation are stored in a new DataFrame named `daily_data`.
- **Reset index:** The `reset_index()` method is applied to convert the group names (dates) from the index to a regular column.

6. Define a Function to Classify Each Day

```python
# Define a function to classify each day based on the criteria
def classify_day(row):
    if row['TEMPERATURE'] > 30 and row['HUMIDITY'] <= 70:
        return 'Hot'
    elif row['TEMPERATURE'] > 30 and row['HUMIDITY'] > 70:
        return 'Hot and Humid'
    elif row['TEMPERATURE'] <= 30 and row['HUMIDITY'] > 30:
        return 'Cold'
    else:
        return 'Cold and Dry'# Define a function to classify each day based on the criteria

def classify_day_heatmap(row):
    if row['TEMPERATURE'] > 30 and row['HUMIDITY'] <= 70:
        return 3  # Hot
    elif row['TEMPERATURE'] > 30 and row['HUMIDITY'] > 70:
        return 4  # Hot and Humid
    elif row['TEMPERATURE'] <= 30 and row['HUMIDITY'] > 30:
        return 2  # Cold
    else:
        return 1  # Cold and Dry
```

`classify_day` function: This is designed to categorize a given day's weather conditions based on its temperature and humidity values. It takes a row of data as input and employs a series of `if-elif-else` conditions to classify the day based on temperature and humidity thresholds:

- If the temperature is above 30 and humidity is less than or equal to 70, the day is classified as '**Hot**'.
- If both temperature and humidity are above 30, the day is classified as '**Hot and Humid**'.
- If the temperature is less than or equal to 30 and humidity is greater than 30, the day is classified as '**Cold**'.
- Otherwise, the day is classified as '**Cold and Dry**'.

`classify_day_heatmap` function: Similar to `classify_day` but returns numerical values instead of strings. The numerical values are intended for use in a heatmap visualization.

7. Apply the classification function to the grouped data.

```
# Group by date and calculate average temperature and humidity
#daily_data = data.groupby('DATE').agg({'TEMPERATURE': 'mean', 'HUMIDITY': 'mean'}).reset_index()
# Apply the classification function to each row
daily_data['CLASSIFICATION'] = daily_data.apply(classify_day, axis=1)

# Save the DataFrame to a CSV file
csv_filename = 'daily_data_with_classification.csv'
daily_data.to_csv(csv_filename, index=False)

# Set display option to show all rows
#pd.set_option('display.max_rows', None)

# Set display option to show 15 rows
pd.set_option('display.max_rows', 15)

# Display the DataFrame
display(daily_data)
```

This code snippet provides a workflow for data processing, analysis, and output. It effectively transforms raw data into a more informative format with added insights. The code effectively classifies each day based on temperature and humidity using the previously defined `classify_day` function. It saves the classified data to a CSV file for later use or analysis. The code controls the output format to display a specific number of rows.

- **Data Classification:**

  `daily_data['CLASSIFICATION'] = daily_data.apply(classify_day, axis=1)` applies the `classify_day` function to each row of the `daily_data` DataFrame, creating a new column named 'CLASSIFICATION' to store the classification results.

- **Data Saving:**

  `csv_filename = 'daily_data_with_classification.csv'` defines the name of the CSV file to save the data.

  `daily_data.to_csv(csv_filename, index=False)` saves the DataFrame `daily_data` to a CSV file without including the index.

- **Display Configuration:**

  `pd.set_option('display.max_rows', 15)` sets the maximum number of rows to display in the DataFrame output to 15. This is used to control the output format.

- **Data Display:**

`display(daily_data)` displays the first 15 rows of the classified DataFrame.

e.g. Sample DataFrame

	DATE	TEMPERATURE	HUMIDITY	CLASSIFICATION
0	2015-01-01	20.333333	83.208333	Cold
1	2015-01-02	23.041667	76.250000	Cold
2	2015-01-03	24.375000	76.750000	Cold
3	2015-01-04	23.916667	68.250000	Cold
4	2015-01-05	21.250000	51.375000	Cold
...	...	...	...	...
2187	2020-12-27	23.083333	28.583333	Cold and Dry
2188	2020-12-28	23.958333	30.250000	Cold
2189	2020-12-29	24.750000	32.458333	Cold
2190	2020-12-30	23.875000	34.125000	Cold
2191	2020-12-31	23.541667	37.375000	Cold

2192 rows × 4 columns

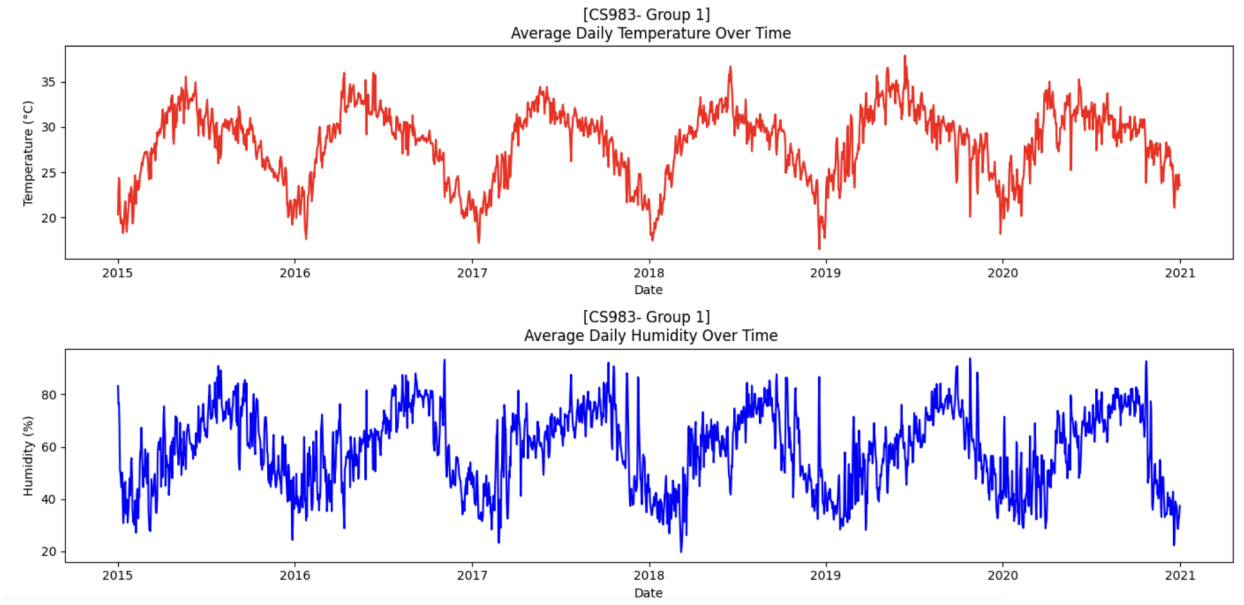
8. Plot the average daily temperature and humidity over time

```
# Plot average daily temperature and humidity over time
plt.figure(figsize=(14, 7))

# Temperature plot
plt.subplot(2, 1, 1)
plt.plot(daily_data['DATE'], daily_data['TEMPERATURE'], color='red')
plt.title('[CS983- Group 1] \n Average Daily Temperature Over Time')
plt.xlabel('Date')
plt.ylabel('Temperature (°C)')

# Humidity plot
plt.subplot(2, 1, 2)
plt.plot(daily_data['DATE'], daily_data['HUMIDITY'], color='blue')
plt.title('[CS983- Group 1] \n Average Daily Humidity Over Time')
plt.xlabel('Date')
plt.ylabel('Humidity (%)')

plt.tight_layout()
plt.show()
```

This code snippet utilizes the `matplotlib.pyplot` library (`plt`) to create informative visualizations based on the previously analyzed data. The line plots depict the trends in average daily temperature and humidity over time.

- **Figure creation:** `plt.figure(figsize=(14, 7))` creates a figure with a specific size for the plots.
- **Subplots:** `plt.subplot(2, 1, 1)` creates a subplot grid with 2 rows and 1 column. The first plot occupies the top half.
- **Temperature plot:**
    - `plt.plot(daily_data['DATE'], daily_data['TEMPERATURE'], color='red')` plots the average daily temperature (y-axis) against the date (x-axis). The line color is set to red.
    - Titles, labels, and formatting are added for clarity.
- **Humidity plot:** Similar approach as the temperature plot, but using the 'HUMIDITY' column and blue color.
- **Layout and display:** `plt.tight_layout()` adjusts spacing, and `plt.show()` displays the figure containing both subplots.

Plot : Temperature and Humidity Plots



9. Preparing data for creating a calendar heatmap

```
data_heatmap = pd.read_csv(file_path)
# Display the first few rows of the dataset to understand its structure
#data_heatmap.head()
# Convert 'DATETIME' to datetime object and extract date
data_heatmap['DATETIME'] = pd.to_datetime(data_heatmap['DATETIME'])
data_heatmap['DATE'] = data_heatmap['DATETIME'].dt.date

# Group by date and calculate average temperature and humidity
daily_data_heatmap = data_heatmap.groupby('DATE').agg({'TEMPERATURE': 'mean', 'HUMIDITY': 'mean'}).reset_index()

# Apply the classification function to each row
daily_data_heatmap['CLASSIFICATION'] = daily_data_heatmap.apply(classify_day_heatmap, axis=1)
daily_data_heatmap.head()
```

This code snippet prepares the data for creating a heatmap based on weather classifications.

- **Data Loading:** (Presumably already done) Loads the CSV data into a DataFrame named `data_heatmap` using `pd.read_csv`.
- **Date Extraction:**
  - `data_heatmap['DATETIME'] = pd.to_datetime(data_heatmap['DATETIME'])` converts the 'DATETIME' column to datetime objects for easier date manipulation.
  - `data_heatmap['DATE'] = data_heatmap['DATETIME'].dt.date` extracts the date part from the datetime objects and creates a new 'DATE' column.
- **Data Aggregation:**
  - `daily_data_heatmap = data_heatmap.groupby('DATE').agg({'TEMPERATURE': 'mean', 'HUMIDITY': 'mean'}).reset_index()`: Similar to previous analysis, this groups the data by 'DATE', calculates the average temperature and humidity for each day, and stores the result in a new DataFrame named `daily_data_heatmap`.
- **Data Classification:**
  - `daily_data_heatmap['CLASSIFICATION'] = daily_data_heatmap.apply(classify_day_heatmap, axis=1)` applies the `classify_day_heatmap` function (presumably defined earlier) to each row of the `daily_data_heatmap` DataFrame. This likely assigns numerical codes (1-4) based on temperature and humidity to a new 'CLASSIFICATION' column.
- **Displaying Results:**
  - `.head()` displays the first few rows of the `daily_data_heatmap` DataFrame to show the newly created 'DATE' and 'CLASSIFICATION' columns.

10. Visualizing data through a calendar heatmap

```
# Define the color map and boundaries
cmap = ListedColormap(['#008000', '#0000FF', '#FF6347', '#FFA500'])  # Green, Blue, Red, Orange
bounds = [0.5, 1.5, 2.5, 3.5, 4.5]  # Boundaries for classification
norm = BoundaryNorm(bounds, cmap.N)

# Ensure the DataFrame has a DatetimeIndex
daily_data_heatmap['DATE'] = pd.to_datetime(daily_data_heatmap['DATE'])
daily_data_heatmap.set_index('DATE', inplace=True)

# Get unique years from the data
years = daily_data_heatmap.index.year.unique()
print(years)
# Plotting for each year
for year in years:
    yearly_data = daily_data_heatmap[daily_data_heatmap.index.year == year]

    fig, ax = plt.subplots(figsize=(12, 8))

    # Plot the calendar heatmap for the current year
    calplot.yearplot(yearly_data['CLASSIFICATION'], cmap=cmap, ax=ax, fillcolor='white')

    # Manually create a colorbar
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = fig.colorbar(sm, ax=ax, orientation='horizontal', pad=0.05)
    cbar.set_ticks([1.5, 2.5, 3.5, 4.5])
    cbar.set_ticklabels(['Cold and Dry', 'Cold', 'Hot and Humid', 'Hot'])

    plt.title(f'Calendar Heatmap for {year}')
    plt.show()
```

This code snippet focuses on creating and visualizing yearly calendar heat maps based on the preprocessed weather classification data. This code iterates through each year in the data, creating a separate calendar heatmap for each year. The heatmap visually represents the daily weather classifications using color coding. The custom colorbar provides a legend for interpreting the heatmap.

- **Colormap and Normalization:**

  `cmap = ListedColormap(['#008000', '#0000FF', '#FF6347', '#FFA500'])`: Defines a colormap with specific colors for each classification category (Green, Blue, Red, Orange).

  `bounds = [0.5, 1.5, 2.5, 3.5, 4.5]`: Sets the boundaries for each classification category to be used with the colormap.

`norm = BoundaryNorm(bounds, cmap.N)`: Creates a normalization object that maps the classification codes (likely 1-4) to the appropriate colors in the colormap based on the defined boundaries.

- **Data Formatting:**

  `daily_data_heatmap['DATE'] = pd.to_datetime(daily_data_heatmap['DATE'])`: Ensures the 'DATE' column is in datetime format.

  `daily_data_heatmap.set_index('DATE', inplace=True)` Sets the 'DATE' column as the index for easier date-based manipulation.

- **Iterating Through Years:**

  `years = daily_data_heatmap.index.year.unique()`: Extracts unique years from the 'DATE' index.

  A `for` loop iterates through each year.

- **Yearly Data Selection:**

  `yearly_data = daily_data_heatmap[daily_data_heatmap.index.year == year]`: Selects data for the current year from the main DataFrame.

- **Heatmap Creation:**

  `fig, ax = plt.subplots(figsize=(12, 8))`: Creates a figure and an axis for plotting.

  `calplot.yearplot(yearly_data['CLASSIFICATION'], cmap=cmap, ax=ax, fillcolor='white')`: Uses the `calplot` library to create a calendar heatmap for the year's classification data. The colormap and axis are specified, and the background is set to white.

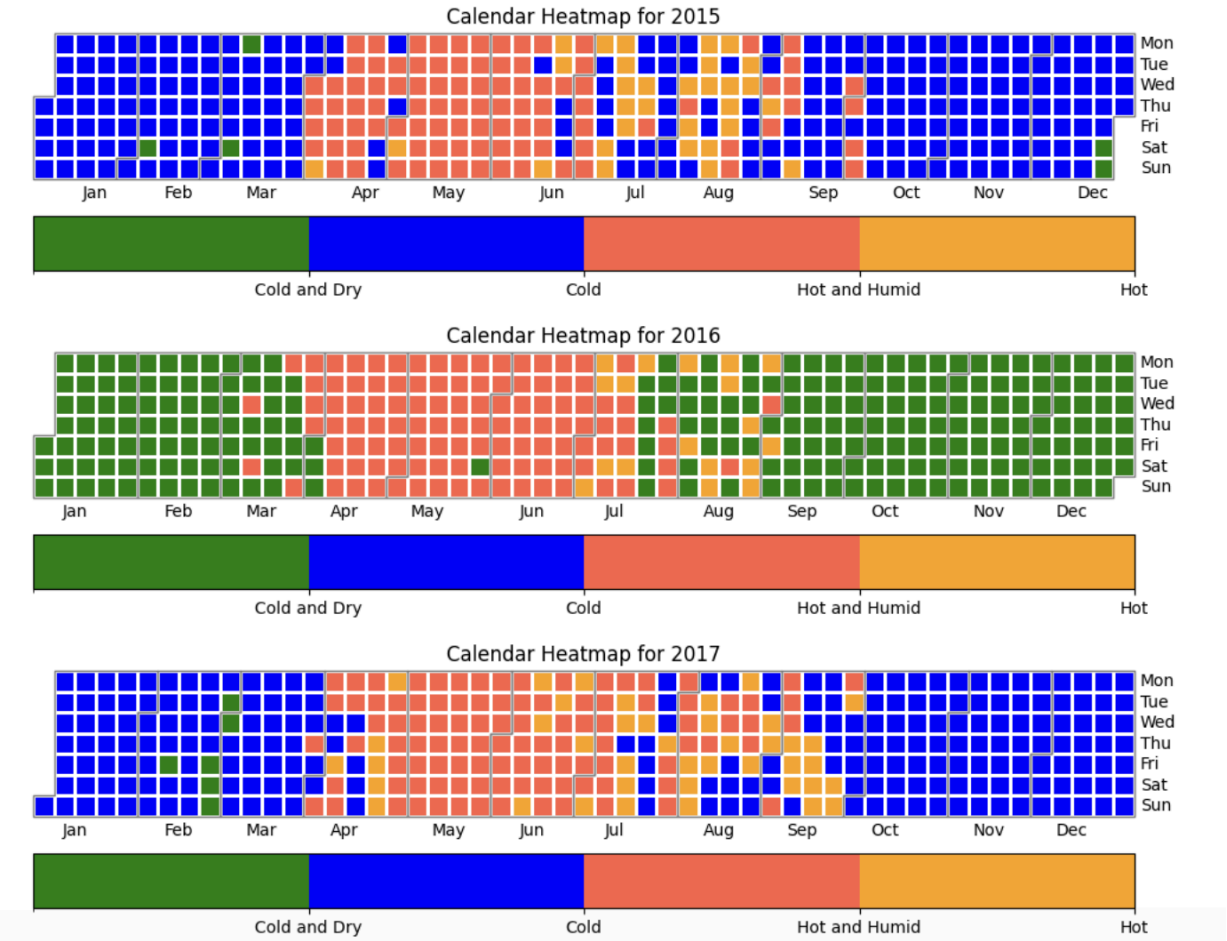
- **Custom Colorbar:**

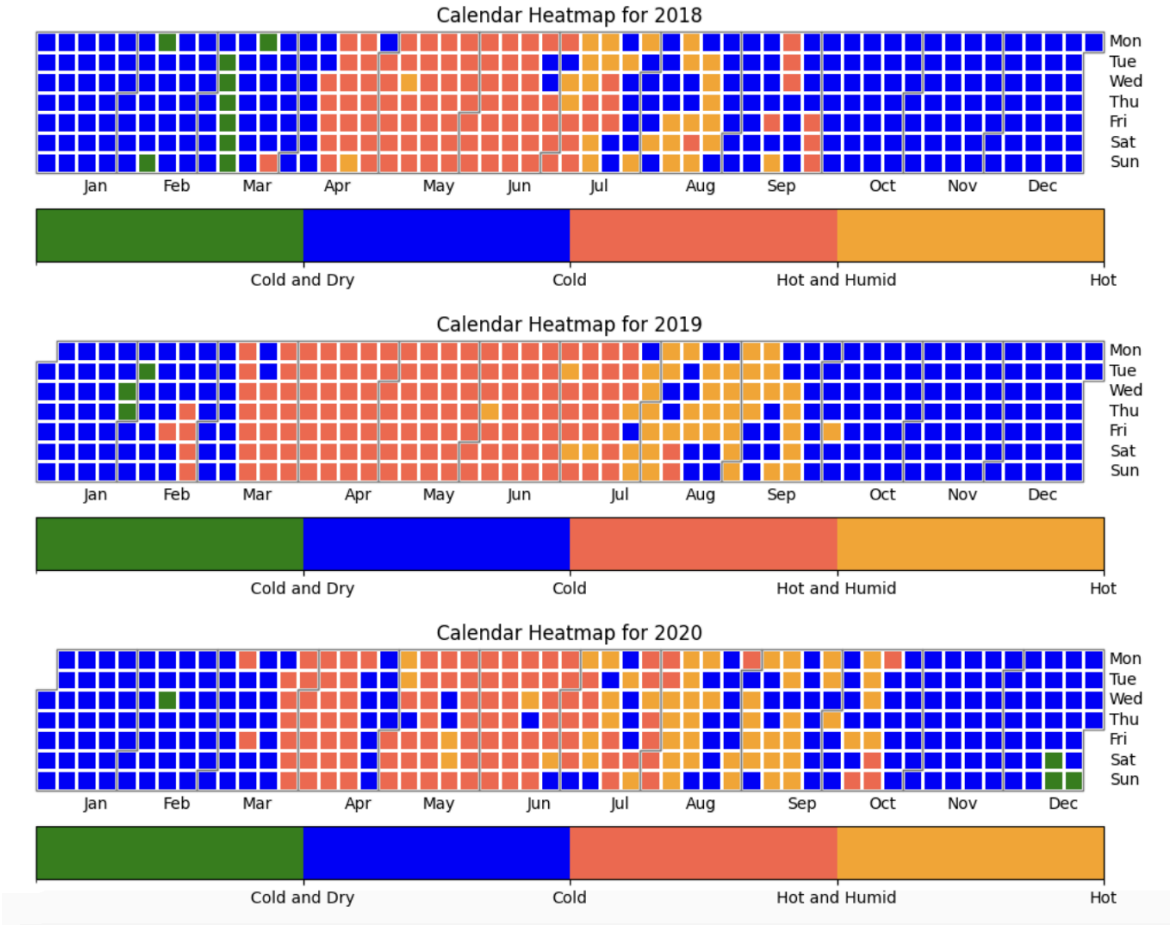
  A custom colorbar is created using `plt.cm.ScalarMappable` and `fig.colorbar`. Tick labels are manually set based on the classification categories and corresponding colors.

- **Figure Title and Display:**

The figure title is set with the current year. `plt.show()` displays the calendar heatmap for the year.

e.g. Sample Heat maps for respective years




Calendar Heatmap for 2018
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Cold and Dry
Cold
Hot and Humid
Hot
Calendar Heatmap for 2019
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Cold and Dry
Cold
Hot and Humid
Hot
Calendar Heatmap for 2020
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Cold and Dry
Cold
Hot and Humid
Hot

References & Credits

1. We would like to express our sincere gratitude to **Prof. Priyanka Bagade** for her invaluable insights shared through her video lectures and live interactive sessions. Her clear explanations and ability to break down complex concepts like IoT, sensors, and actuators were instrumental in deepening our understanding and shaping our approach to the subject.
2. We would like to extend our thanks to the Wokwi platform for providing a wealth of valuable resources in their documentation section (https://docs.wokwi.com). These resources greatly facilitated our research and experimentation with various IoT concepts.
3. Last but not least, we would like to extend our thanks to our TA, Nitish Kumar, for his timely and prompt responses to our queries related to the IoT assignment.