Smart Temperature Control System

Author: Alex Viner

Team Member: Nicholas Floyd

December 13, 2024

Embedded Systems

Professor Beichel

1. Introduction

The motivation behind making a smart temperature controller was to develop a prototype that enhanced a home's air conditioning and heating system. We wanted this device to simulate automatic temperature adjustments, as well as remote commands that could adjust the running time of the system. By using a RTC and temperature sensor, we wanted to simulate the capabilities of a smart home system with the fan acting as our heating/cooling system. We wanted to utilize the temperature sensor's data to calculate how fast to run the fan, speeding up if the true temperature was much higher than the desired. The real time clock would be utilized for users to set a time off/time on for the system. We initially planned on having an RPG to control the system's desired temperature physically. All information for the system was to be displayed on an LCD. We wanted a way for a user to enter in commands that could control the system externally, but we also wanted to include a physical pushbutton for manual on/off capabilities.

2. Implementation

For our hardware components, we opted to utilize the AHT20 temperature sensor. We chose this device for its 5V rating, along with its ability to communicate with I2C (AHT20 Datasheet), as our originally planned DHT11 temperature sensor did not have I2C support. The AHT20's SDA and SCL lines had internal pull-up resistors, so none were needed for proper I2C communication. The RTC we chose was the DS1307, which needed a 32.768 kHz crystal for a 1 Hz oscillation. This RTC also used I2C communication (DS1307 Datasheet), which required pull up resistors on the SDA and SCL lines. We also opted to switch out the RPG for a potentiometer, as most temperature dials act like a potentiometer with a lowest and highest setting. The 1602 LCD was utilized for displaying information from the system. We utilized the small CPU fan we had used in Lab 4, as we were familiar with its capabilities, along with a pushbutton for manual switching of the system. All these

components were connected to the Arduino Uno, containing the ATMega328P microcontroller.

On the software side, our C program communicated with our temperature sensor and RTC using one I2C bus. The program got the voltage from the potentiometer line and converted that number into a value between 50- and 75-degrees Fahrenheit, which was then output to the LCD display. It also took in the data from the temperature sensor, which initially came in Celsius, and converted it to Fahrenheit and then output. We utilized the Fast PWM mode of the CPU fan, and had our program run at 30 percent when the set temperature was 1 degree lower than the actual, which then increased by 5 percent for every singular increase in the gap between the two. The pushbutton triggered an ISR to manually override the fan's settings, shutting the system down if running, or turning it back on to the desired fan speed if the system was off. We also implemented ISR's for our commands, which were handled by the Arduino IDE serial monitor.

We utilized the Arduino IDE serial monitor to act as our external terminal, allowing users to enter commands to control the system's behavior. Utilizing the RTC, we implemented a setTime function to allow users to set the current time, as well as a setOn and setOff command, which took in an hour and minute value, and shut on or off the system at that desired time. We also implemented manual on and off commands in the IDE to allow immediate switching as well.

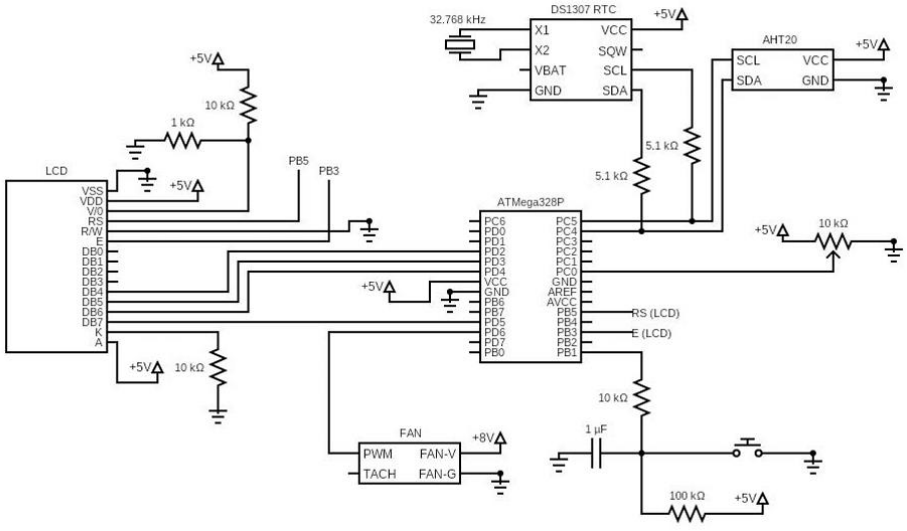

Figure 2.1: Project Schematic

Figure 1.1 shows our project schematic. We ran both the RTC and AHT20 on the same bus to the microcontroller's SDA and SCL lines. The potentiometer connected allowed us to control a set temperature dial by the user. The fan's PWM line is plugged into the board,

along with our LCD display's four data lines and the Enable and RS lines. We implemented hardware debounce for the pushbutton which was also connected to the microcontroller.

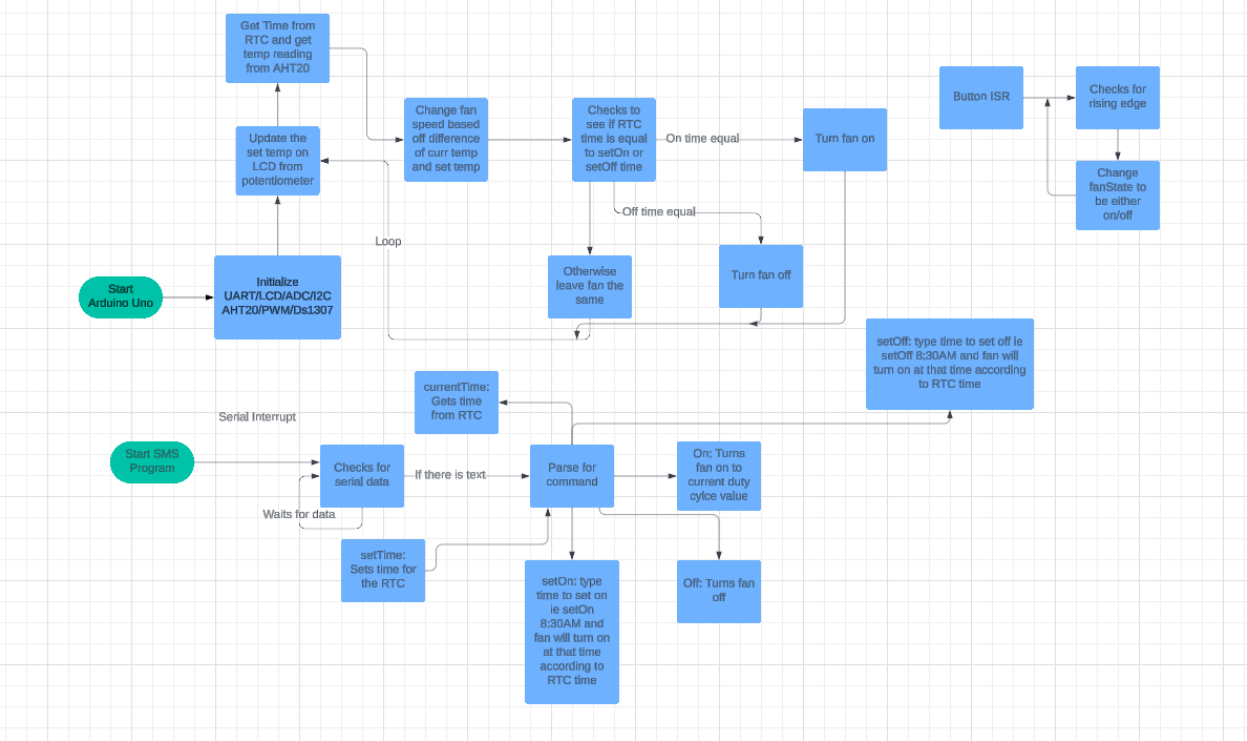

Figure 2.2: Project Flowchart

Figure 2.2 is a flowchart demonstrating the flow of the project code. With the main loop code, it reads RTC and temperature values and changes fan speed and LCD values off that. Shown are also the two ISR routines that are used for serial communication and button override.

3. Experimental Methods

To test our project, we initially ensured functionality of each of our new components, the AHT20 and the DS1307 RTC. We tested the AHT20, comparing using a digital thermometer as well as the thermostat in the Embedded Lab at varying temperatures. We tested the RTC at 5-, 20-, and 60-minute intervals and compared it to the world clock. We ensured these two were functional and then went about implementation of our project design. After this, we initiated tests on our system-wide functions, like the pushbutton manual on and off, as well as tests for all our commands on the serial monitor input. We tested the serial monitor commands that toggled the system by timing its reaction to the specified input, such as ensuring a setOff function for 6:30PM shut the system down at exactly 6:30 PM.

4. **Results**

Test	Result	Data (if applicable)
AHT20 Responsiveness Test	Pass	Time to display: <1s
AHT20 Temperature Accuracy Tests	Pass	Within 2 degrees F
Pushbutton Manual On Test	Pass	Time: <1 s
Pushbutton Manual Off Test	Pass	Time: 1-2 s
Command On Test	Pass	Time: <1 s
Command Off Test	Pass	Time: 1-2 s
Command SetTime	Pass	
RTC Accuracy Test (After 5, 20, 60 mins)	Pass	Comparison to real time: Within 1 s)
Command CurrentTime Test	Pass	Time: <1 s
Command SetOn Test	Pass	Time: <1 s
Command SetOff Test	Pass	Time: 1-2s

Figure 4.1: Test Results

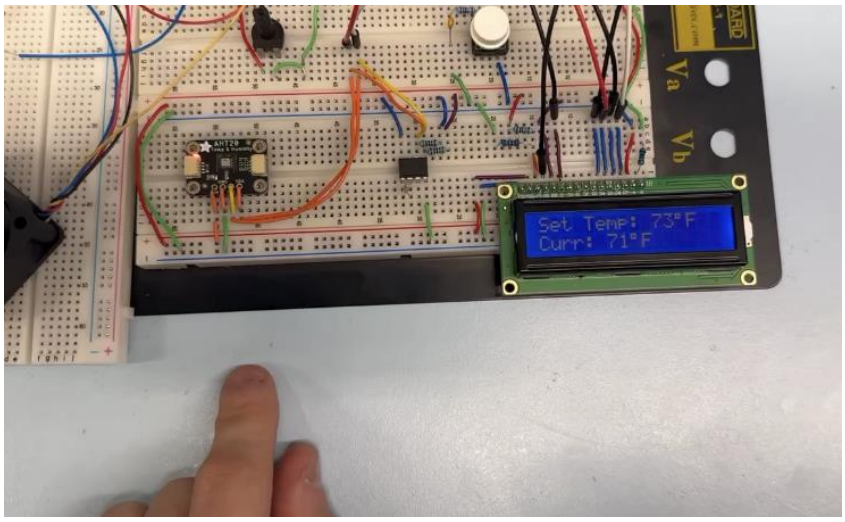

Figure 4.2a: AHT20 Responsiveness Test

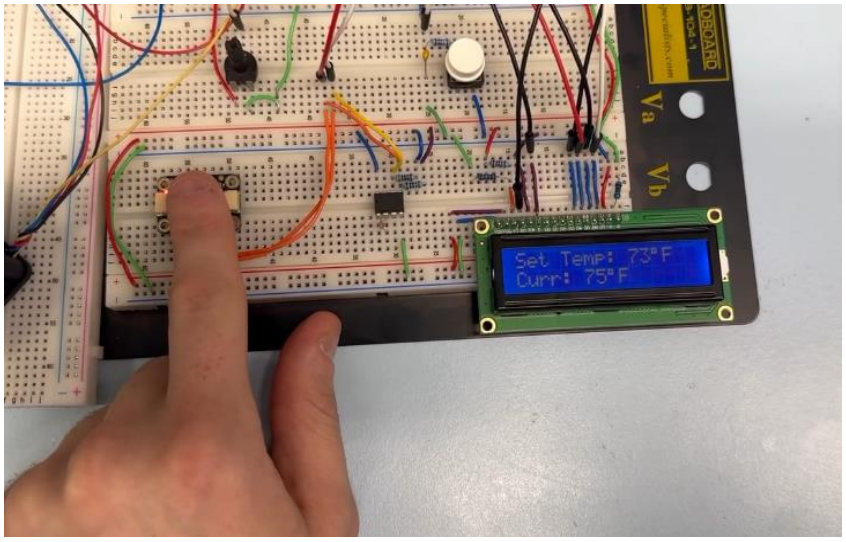

Figure 4.2b: AHT20 Responsiveness Test

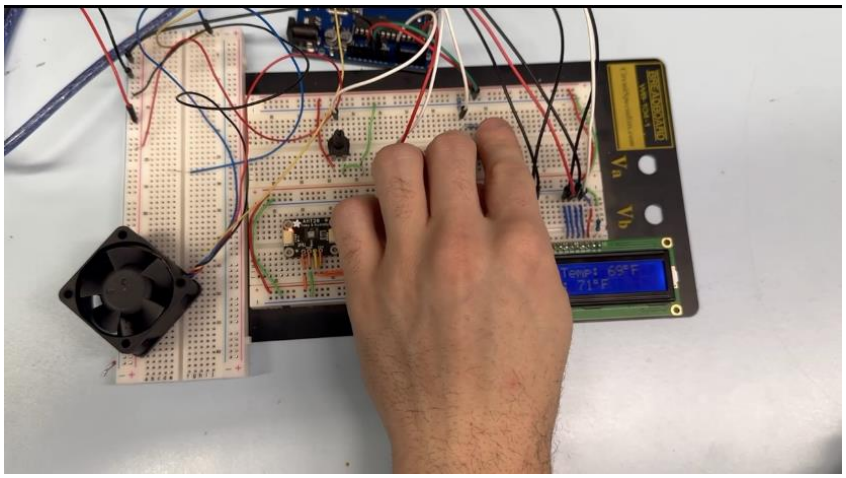

Figure 4.3a Pushbutton Manual On Test

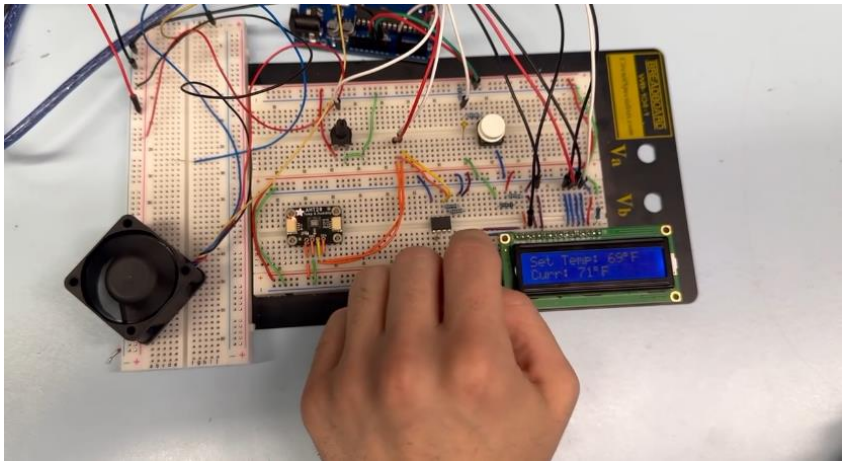

Figure 4.3b Pushbutton Manual On Test

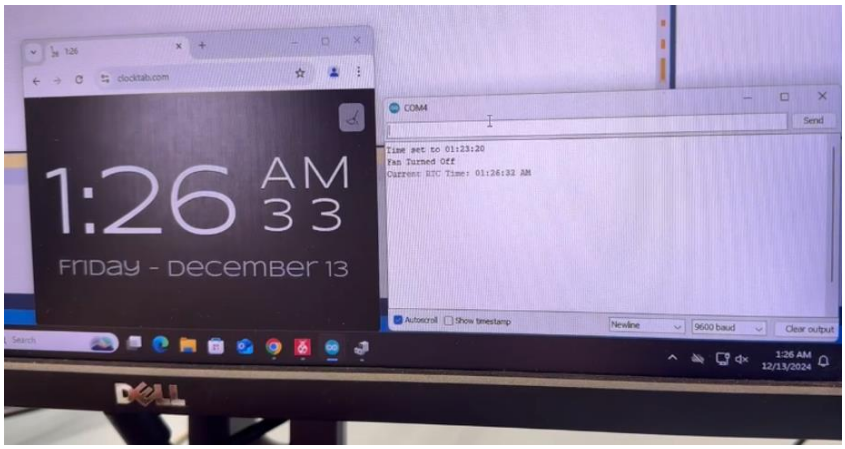

Figure 4.4: CurrentTime Command Test

5. Discussion of Results

With our final prototype, our device worked almost exactly as we intended in the beginning, with a few minor quirks and tweaks as we developed the project. All the tests we conducted passed our intended set points, with each command performing its desired action within a reasonable time (<2 seconds). Along with those, we tested the system's handling of values that commands should not react to, such as a set time for 13:00 PM, which just gave an error in the terminal for out of bounds values. We also ensured that a setOn and setOff command set for the same time only reacted to the most recent command in the terminal. One small issue we ran into was the voltage regulator on our Arduino Uno not functioning properly, which caused us to have to use a barrel jack connector, instead of being able to use our VIN pin for the wall-adapter power jack. Another small issue with our Arduino UNO was the SDA and SCL pins not correctly reading on the first Arduino we used. After switching Arduino's, this pin then functioned as intended for the remainder of the project. As hardware and physical components go, those were the two main issues we ran into for the project.

The device performs very well, is very responsive with its commands and the fan adjusts to changes in the difference between the real temperature and the set temperature. All commands work as intended and the system reacts quickly to them. While the fan could not do much to alter the temperature in the room, it worked as a simulator to what a real implementation of a smart control system would look like full-scale. All design goals we initially had in mind were met, and the device functioned as we intended. Current implementation alternatives could involve the substitute of a larger LCD for displaying more data (Humidity data, time, etc.). With only two lines on the 1602 LCD, it put a throttle on the amount of data we could output to the user on the physical interface. The DS1307 real time clock we used could also be connected to a battery, which would allow a user to

only input the current time once, and the clock would keep running even if power to the system was lost. The next steps for improvement of this project would involve an application that could control the system remotely from a smartphone. This could involve a UI that displays temperature readings and allows users to perform the commands we have implemented. For hardware, the next step would be implementing this into a full HVAC system, toggling air conditioning systems and heating remotely, as opposed to our small CPU fan that only worked for simulation purposes.

6. Conclusion

Overall, our project successfully accomplished what we had envisioned. We successfully utilized two new devices, the AHT20 and the DS1307 (as well as its 32.768 kHz crystal), using their I2C capabilities to properly communicate with our program and simulate a smart temperature control system. We also utilized serial monitoring on the Arduino IDE and the 1602 LCD to properly display information and allow the user to implement commands to control the project. We gained experience with multiple devices using I2C communication on the same bus. We also learned more about reading datasheets to correctly determine the device best fit for our project, as well as using RTC devices to implement clock-dependent functions. We dived deeper into ISR's in C, which we used for all commands that a user could input to the system. For practical implications of this project, the next steps would be figuring out interfacing with current HVAC systems, as well as an application to deploy this project remotely.

7. Acknowledgements & References

- AHT20 Datasheet: https://www.digikey.com/en/htmldatasheets/production/5750909/0/0/1/4566
- DS1307 Datasheet: https://www.alldatasheet.com/datasheet-pdf/pdf/58481/dallas/ds1307.html

Software Libaries Used:

- I2CMaster Library by Peter Fleury:

  https://github.com/damadmai/pfleury/blob/master/i2cmaster.h

8. Source Code

```
//Final Project RTC operated Cooling System //Nicholas Floyd and Alex Viner

#define F_CPU 16000000UL // 16 MHz for delay and timer calculations #include <avr/io.h>

#include <util/delay.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include "i2cmaster.h"

#include <avr/interrupt.h> #include <stdbool.h>

#define LCD_RS PB5 // Register Select pin for LCD #define LCD_EN PB3 // Enable pin for LCD #define LCD_D4 PD2 #define LCD_D5 PD3 #define LCD_D6 PD4 #define LCD_D7 PD5

// Button Pin #define BUTTON PB0

// I2C Addresses for connected devices #define AHT20_ADDRESS 0x38 // I2C address for AHT20 temperature sensor #define DS1307_ADDRESS 0x68 // I2C address for DS1307 RTC module

static uint8_t fanOnHour = 9; // Fan ON time: hour static uint8_t fanOnMinute = 0; // Fan ON time: minute static uint8_t fanOffHour = 10; // Fan OFF time: hour static uint8_t fanOffMinute = 0; // Fan OFF time: minute static uint8_t fanTargetTemp = 70; // Target temperature for fan control (°F)

// PWM control for the fan #define FAN_PIN PD6 // Fan connected to PD6

// UART communication settings #define BAUD 9600 // UART baud rate #define MYUBRR 103 // Value for UBRR register for 9600 baud at 16 MHz

// Command buffer for UART-based user input #define CMD_BUFFER_SIZE 64 // Maximum size for a command string volatile char commandBuffer[CMD_BUFFER_SIZE]; // Buffer to store commands volatile uint8_t commandIndex = 0; // Index for storing command characters volatile uint8_t commandReady = 0; // Flag to indicate a command is ready

bool fanState = false; // Current fan state (ON/OFF) uint8_t dutyCyclePercentage = 0; // Current PWM duty cycle for the fan #define ADC_SAMPLES 10

void LCD_Command(unsigned char cmd); // Send command to LCD void LCD_Char(unsigned char data); // Display a single character on LCD void LCD_Init(void); // Initialize the LCD void LCD_String(const char *str); // Display a string on LCD void LCD_Clear(void); // Clear the LCD screen void LCD_SetCursor(unsigned char row, unsigned char col); // Set cursor position on LCD

void DS1307_EnableOscillator(void); // Enable oscillator for DS1307 RTC

void DS1307_SetTime(uint8_t hour, uint8_t min, uint8_t sec); // Set time on DS1307 void DS1307_GetTime(uint8_t *hour, uint8_t *min, uint8_t *sec); // Get current time from DS1307

void AHT20_Init(void); // Initialize AHT20 sensor void AHT20_TriggerMeasurement(void); // Trigger measurement on AHT20 void AHT20_ReadData(float *temperatureC); // Read temperature data from AHT20

void ADC_Init(void); // Initialize ADC uint16_t ADC_Read(uint8_t channel); // Read value from ADC channel uint16_t ADC_GetSmoothedReading(uint8_t channel); // Get smoothed ADC reading (average of multiple samples)

void PWM_Init(void); // Initialize PWM for fan control void Set_Fan_Speed(uint8_t duty_cycle); // Set fan speed based on duty cycle

void USART_init(unsigned int ubrr); // Initialize UART communication void USART_transmit(char data); // Transmit a character via UART void USART_sendString(char *str); // Transmit a string via UART void USART_getString(char *str, int maxLength); // Receive a string via UART

void Update_Display_Set_Temp(void); // Update LCD with target temperature void ParseCommand(char *cmd); // Parse user command from UART input

uint8_t DecToBcd(uint8_t val); // Function prototype
```

```
// Main program int main(void) { USART_init(MYUBRR); // Initializations LCD_Init();

 ADC_Init();

 i2c_init();

 AHT20_Init();

 PWM_Init();

 DS1307_EnableOscillator();

float current_tempC;   // Current temperature in Celsius
int current_tempF;     // Current temperature in Fahrenheit
uint8_t current_hour, current_min, current_sec; // Current time from DS1307


DS1307_SetTime(12, 0, 0); // Set DS1307 time to 12:00:00 by default


while (1) {
    Update_Display_Set_Temp(); // Update LCD with potentiometer temp


    // Check if a UART command is ready to process otherwise skip
    if (commandReady) {
        ParseCommand((char *)commandBuffer); // Parse and execute the command
        commandReady = 0; // Clear the command ready flag
    }


    // Get the current time from the DS1307
    DS1307_GetTime(&current_hour, &current_min, &current_sec);


    // Get the current temperature from AHT20
    AHT20_TriggerMeasurement();       // Trigger measurement
    AHT20_ReadData(&current_tempC);   // Read data
    current_tempF = (int)((current_tempC * 9.0 / 5.0) + 32.0 + 0.5) - 4; // Convert to Fahrenheit


    // Display the current temperature on the LCD
    LCD_SetCursor(1, 0); // Move to the second line
    LCD_String("Curr: ");
    char buffer[16];
    snprintf(buffer, sizeof(buffer), "%d", current_tempF);
    LCD_String(buffer); // Print temperature value
    LCD_Char(223);      // Print degree symbol
    LCD_Char('F');      // Print "F" for Fahrenheit


    // Fan control logic based on temperature and time
    int currentTimeInMin = current_hour * 60 + current_min; // Current time in minutes
    int fanOnTimeInMin = fanOnHour * 60 + fanOnMinute;      // Fan ON time in minutes
    int fanOffTimeInMin = fanOffHour * 60 + fanOffMinute;   // Fan OFF time in minutes
    int difference = current_tempF - fanTargetTemp;         // Temperature difference from target


    // Calculate PWM duty cycle for fan
    if (difference <= 0) {
        dutyCyclePercentage = 0; //0 if no difference or target temp is hotter
        } else {
        dutyCyclePercentage = 30 + (difference - 1) * 5; //default 30% + 5% for each additional degree
        if (dutyCyclePercentage > 100) dutyCyclePercentage = 100; // Cap at 100%
    }


    // Turn fan ON/OFF based on schedule
    if ((currentTimeInMin == fanOnTimeInMin) && (current_sec <= 2)) {
        fanState = true; // Turn fan ON
        dutyCyclePercentage = 30 + (difference - 1) * 5; //default 30% + 5% for each additional degree
        if (dutyCyclePercentage > 100) dutyCyclePercentage = 100; // Cap at 100%
        uint8_t dutyValue = (uint8_t)((dutyCyclePercentage * 255UL) / 100UL);
        Set_Fan_Speed(dutyValue);
        } else if ((currentTimeInMin == fanOffTimeInMin) && (current_sec <= 2)) {
```

```
        Set_Fan_Speed(0); // Turn fan OFF
        fanState = false;
    }
}



}
```

void DS1307_SetTime(uint8_t hour, uint8_t min, uint8_t sec) { i2c_start((DS1307_ADDRESS << 1) | I2C_WRITE); i2c_write(0x00); // Point to the seconds register i2c_write(DecToBcd(sec) & 0x7F); // Clear CH bit (bit 7) to start oscillator i2c_write(DecToBcd(min)); // Write minutes i2c_write(DecToBcd(hour)); // Write hours i2c_stop(); }

void USART_init(unsigned int ubrr) { UBRR0H = (unsigned char)(ubrr >> 8); // Set baud rate high byte UBRR0L = (unsigned char)ubrr; // Set baud rate low byte UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0); // Enable RX, TX, and RX interrupt UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // 8 data bits, 1 stop bit

```
DDRB &= ~(1 << BUTTON); // Set PB5 as input
PORTB |= (1 << BUTTON); // Enable pull-up resistor on PB5


PCICR |= (1 << PCIE0);  // Enable Pin Change Interrupt for PCINT0-PCINT7
PCMSK0 |= (1 << PCINT0); // Enable interrupt on PB0


sei(); // Enable global interrupts



}
```

void USART_transmit(char data) { // Wait for the transmit buffer to be ready (UDRE0 bit set) while (!(UCSR0A & (1 << UDRE0))); // Load the data into the UART Data Register (UDR0) to send it UDR0 = data; }

void USART_sendString(char *str) { // Loop through each character in the string while (*str) { // Wait for the transmit buffer to be ready and send the character while (!(UCSR0A & (1 << UDRE0))); UDR0 = *str++; } }

void USART_getString(char *str, int maxLength) { int i; char c; for (i = 0; i < maxLength - 1; i++) { // Wait for a character to be received (RXC0 bit set) while (!(UCSR0A & (1 << RXC0))); c = UDR0; // Read the received character str[i] = c; // Store the character in the string buffer USART_transmit(c); if (c == '\n' || c == '\r') { // Stop if newline or carriage return is received break; } } str[i] = '\0'; // Null-terminate the string }

void LCD_Command(unsigned char cmd) { // Send a command to the LCD PORTB &= ~(1 << LCD_RS); // RS=0 for command mode PORTD = (PORTD & 0x03) | ((cmd & 0xF0) >> 2); // Send the higher nibble PORTB |= (1 << LCD_EN); // Enable the LCD _delay_us(1); PORTB &= ~(1 << LCD_EN); // Disable the LCD PORTD = (PORTD & 0x03) | ((cmd & 0x0F) << 2); // Send the lower nibble PORTB |= (1 << LCD_EN); // Enable the LCD _delay_us(1); PORTB &= ~(1 << LCD_EN); // Disable the LCD _delay_ms(2); }

void LCD_Char(unsigned char data) { // Send a character to the LCD PORTB |= (1 << LCD_RS); // RS=1 for data mode PORTD = (PORTD & 0x03) | ((data & 0xF0) >> 2); // Send the higher nibble PORTB |= (1 << LCD_EN); // Enable the LCD _delay_us(1); PORTB &= ~(1 << LCD_EN); // Disable the LCD PORTD = (PORTD & 0x03) | ((data & 0x0F) << 2); // Send the lower nibble PORTB |= (1 << LCD_EN); // Enable the LCD _delay_us(1); PORTB &= ~(1 << LCD_EN); // Disable the LCD _delay_ms(2); }

void LCD_Init(void) { // Initialize the LCD in 4-bit mode DDRB |= (1 << LCD_RS) | (1 << LCD_EN); // Set RS and EN as output DDRD |= (1 << LCD_D4) | (1 << LCD_D5) | (1 << LCD_D6) | (1 << LCD_D7); // Set data lines as output _delay_ms(20); // Wait for LCD to power up LCD_Command(0x02); // Initialize in 4-bit mode LCD_Command(0x28); LCD_Command(0x0C); // Display ON, Cursor OFF LCD_Command(0x06); LCD_Command(0x01); // Clear display _delay_ms(2); }

void LCD_String(const char *str) { // Display a string on the LCD while (*str) { LCD_Char(*str++); // Send each character } }

void LCD_Clear(void) { // Clear the LCD display LCD_Command(0x01); // Clear display command _delay_ms(2); // Wait for the command to execute }

void LCD_SetCursor(unsigned char row, unsigned char col) { // Set the cursor position on the LCD unsigned char address = (row == 0) ? (0x80 + col) : (0xC0 + col); // Calculate DDRAM address LCD_Command(address); // Send the address to the LCD }

void ADC_Init(void) { // Initialize the ADC ADMUX = (1 << REFS0); // Set reference voltage to AVcc ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // Enable ADC and set prescaler to 128 }

uint16_t ADC_Read(uint8_t channel) { // Read an analog value from the specified ADC channel channel &= 0x07; // Limit the channel to valid range (0-7) ADMUX = (ADMUX & 0xF8) | channel; // Select the ADC channel ADCSRA |= (1 << ADSC); // Start the conversion while (ADCSRA & (1 << ADSC)); // Wait for the conversion to complete return ADC; // Return the ADC value }

uint16_t ADC_GetSmoothedReading(uint8_t channel) { // Get an averaged ADC reading over multiple samples uint32_t sum = 0; for (uint8_t i = 0; i < ADC_SAMPLES; i++) { sum += ADC_Read(channel); // Accumulate multiple readings _delay_ms(10); // Small delay between samples } return (uint16_t)(sum / ADC_SAMPLES); // Return the average }

void PWM_Init(void) { // Initialize PWM on Timer0 for fan control TCCR0A |= (1 << WGM01) | (1 << WGM00); // Set Timer0 to Fast PWM mode TCCR0A |= (1 << COM0A1); // Enable non-inverted PWM on OC0A (PD6) TCCR0B |= (1 << CS01); // Set prescaler to 8 DDRD |= (1 << FAN_PIN); // Set PD6 as output OCR0A = 0; // Initialize duty cycle to 0 (fan OFF) }

void Set_Fan_Speed(uint8_t duty_cycle) { // Set the fan speed using PWM duty cycle if (fanState == true) { OCR0A = duty_cycle ; // Set the duty cycle if fan is ON } else { OCR0A = 0; // Set duty cycle to 0 (fan OFF) } }

//converts dec num like 45 to 0x45 uint8_t DecToBcd(uint8_t val) { return ((val / 10) * 16) + (val % 10); }

//converts bcd like 0x45 to 45 uint8_t BcdToDec(uint8_t val) { return ((val / 16) * 10) + (val % 16); }

void DS1307_GetTime(uint8_t *hour, uint8_t *min, uint8_t *sec) { // Read the current time from the DS1307 RTC i2c_start((DS1307_ADDRESS << 1) | I2C_WRITE); // Send start condition and select DS1307 in write mode i2c_write(0x00); // Point to the seconds register i2c_stop(); // Stop the I2C communication

```
i2c_start((DS1307_ADDRESS << 1) | I2C_READ); // Send start condition and select DS1307 in read mode
uint8_t s = i2c_readAck(); // Read seconds and acknowledge
uint8_t m = i2c_readAck(); // Read minutes and acknowledge
uint8_t h = i2c_readNak(); // Read hours and do not acknowledge
i2c_stop(); // Stop the I2C communication


// Convert BCD values to decimal and store in the provided pointers
*sec = BcdToDec(s & 0x7F); // Mask the CH bit in seconds
*min = BcdToDec(m);
*hour = BcdToDec(h & 0x3F); // Mask the 24-hour format bit in hours



}
```

void AHT20_Init(void) { i2c_start((AHT20_ADDRESS << 1) | I2C_WRITE); // Start I2C communication with AHT20 i2c_write(0xBA); // Soft reset command i2c_stop(); // Stop I2C communication _delay_ms(20); // Wait for the sensor to reset

```
// Send initialization command
i2c_start((AHT20_ADDRESS << 1) | I2C_WRITE); // Start I2C communication
i2c_write(0xBE); // Command to initialize the sensor
i2c_write(0x08); // Enable data mode
i2c_write(0x00); // Reserved byte
i2c_stop(); // Stop I2C communication
_delay_ms(10); // Wait for the sensor to initialize



}
```

void AHT20_TriggerMeasurement(void) { i2c_start((AHT20_ADDRESS << 1) | I2C_WRITE); // Start I2C communication i2c_write(0xAC); // Measurement trigger command i2c_write(0x33); // Enable temperature measurement i2c_write(0x00); // Reserved byte i2c_stop(); // Stop I2C communication _delay_ms(80); // Wait for the measurement to complete }

void AHT20_ReadData(float *temperatureC) { // Read temperature data from the AHT20 sensor uint8_t rawData[6]; // Array to store raw data i2c_start((AHT20_ADDRESS << 1) | I2C_READ); // Start I2C communication in read mode // Read 6 bytes of data from the sensor

```
for (uint8_t i = 0; i < 5; i++) {
    rawData[i] = i2c_readAck(); // Acknowledge after each byte
}
rawData[5] = i2c_readNak(); // Do not acknowledge the last byte
i2c_stop(); // Stop I2C communication


// Extract the temperature data from the raw bytes
uint32_t tempData = ((uint32_t)(rawData[3] & 0x0F) << 16) | // Combine bytes into a 20-bit temperature value
((uint32_t)rawData[4] << 8) |
rawData[5];


// Convert the raw temperature data to degrees Celsius
*temperatureC = ((float)tempData / 1048576.0) * 200.0 - 50.0;



}
```

void DS1307_EnableOscillator(void) { i2c_start((DS1307_ADDRESS << 1) | I2C_WRITE); // Start I2C communication in write mode i2c_write(0x00); // Point to the seconds register i2c_start((DS1307_ADDRESS << 1) | I2C_READ); // Restart I2C communication in read mode uint8_t seconds = i2c_readNak(); // Read the seconds register i2c_stop(); // Stop I2C communication

```
if (seconds & 0x80) { // If the oscillator is halted
    seconds &= ~0x80; // Clear the CH bit to enable the oscillator
    i2c_start((DS1307_ADDRESS << 1) | I2C_WRITE); // Start I2C communication in write mode
    i2c_write(0x00); // Point to the seconds register
    i2c_write(seconds); // Write the updated seconds value
    i2c_stop(); // Stop I2C communication
}
```

}

void Update_Display_Set_Temp(void) { static uint8_t previous_temp = 0; // Store the previous temperature to avoid unnecessary updates

```
// Read the potentiometer value from ADC
uint16_t adcValue = ADC_GetSmoothedReading(0);
adcValue = 1023 - adcValue; // Reverse the ADC value to get desired mapping
float voltage = (adcValue * 5.0) / 1024.0; // Convert ADC value to voltage
uint8_t temperature = 50 + (uint8_t)(voltage * 5); // Map voltage to temperature range (50–75°F)


// Update the global target temperature
fanTargetTemp = temperature;


// Update the LCD only if the temperature has changed
if (temperature != previous_temp) {
    char buffer[16]; // Buffer to store the formatted temperature string
    LCD_SetCursor(0, 0); // Set LCD cursor to the first line
    LCD_String("Set Temp: "); // Display "Set Temp:"
    snprintf(buffer, sizeof(buffer), "%d", temperature); // Format the temperature as a string
    LCD_String(buffer); // Display the temperature
    LCD_Char(223); // Display the degree symbol
    LCD_Char('F'); // Display "F" for Fahrenheit
    previous_temp = temperature; // Update the previous temperature
}
```

}

void ParseCommand(char *cmd) { char buffer[64]; // Buffer to store response messages

```
// Check if the command starts with "setOn"
if (strncmp(cmd, "setOn", 5) == 0) {
    // Parse the "setOn" command to set the fan ON time
    char *token = strtok(cmd, " "); // Extract "setOn"
    token = strtok(NULL, " ");     // Extract time argument (e.g., "8:30AM")
    uint8_t hour = 0, minute = 0;  // Variables to store parsed hour and minute


    if (token) {
        char timeStr[8]; // Buffer to store the time string
        strncpy(timeStr, token, 7); // Copy the time string
        timeStr[7] = '\0'; // Null-terminate the string


        int pm = 0; // Flag to indicate PM
        char *ampm = strstr(timeStr, "AM"); // Check for "AM" (strstr used to find first position of second argument in string)
        if (!ampm) {
            ampm = strstr(timeStr, "PM"); // Check for "PM"
            pm = 1; // Set PM flag
        }


        if (ampm) *ampm = '\0'; // Remove AM/PM from the time string


        char *colon = strchr(timeStr, ':'); // Find the colon separating hour and minute
        if (colon) {
            *colon = '\0'; // Split the string at the colon
```

```
        hour = (uint8_t)atoi(timeStr); // Convert hour string to integer
        minute = (uint8_t)atoi(colon + 1); // Convert minute string to integer
    }


    // Convert 12-hour format to 24-hour format
    if (pm && hour < 12) hour += 12; // Convert PM to 24-hour
    if (!pm && hour == 12) hour = 0; // Handle midnight case


    // Update global fan ON time variables
    fanOnHour = hour;
    fanOnMinute = minute;


    // Respond with confirmation message
    snprintf(buffer, sizeof(buffer), "Set On Time: %02d:%02d\n", fanOnHour, fanOnMinute);
    USART_sendString(buffer);
    } else {
    // Respond with an error message if the format is invalid
    USART_sendString("Invalid setOn command format. Use: setOn HH:MMAM/PM\n");
}


} else if (strncmp(cmd, "setOff", 6) == 0) {
// Parse the "setOff" command to set the fan OFF time
char *token = strtok(cmd, " "); // Extract "setOff"
token = strtok(NULL, " ");     // Extract time argument (e.g., "8:30AM")
uint8_t hour = 0, minute = 0;  // Variables to store parsed hour and minute


if (token) {
    char timeStr[8]; // Buffer to store the time string
    strncpy(timeStr, token, 7); // Copy the time string
    timeStr[7] = '\0'; // Null-terminate the string


    int pm = 0; // Flag to indicate PM
    char *ampm = strstr(timeStr, "AM"); // Check for "AM"
    if (!ampm) {
        ampm = strstr(timeStr, "PM"); // Check for "PM"
        pm = 1; // Set PM flag
    }


    if (ampm) *ampm = '\0'; // Remove AM/PM from the time string


    char *colon = strchr(timeStr, ':'); // Find the colon separating hour and minute
    if (colon) {
        *colon = '\0'; // Split the string at the colon
        hour = (uint8_t)atoi(timeStr); // Convert hour string to integer
        minute = (uint8_t)atoi(colon + 1); // Convert minute string to integer
    }


    // Convert 12-hour format to 24-hour format
    if (pm && hour < 12) hour += 12; // Convert PM to 24-hour
    if (!pm && hour == 12) hour = 0; // Handle midnight case


    // Update global fan OFF time variables
    fanOffHour = hour;
    fanOffMinute = minute;


    // Respond with confirmation message
    snprintf(buffer, sizeof(buffer), "Set Off Time: %02d:%02d\n", fanOffHour, fanOffMinute);
    USART_sendString(buffer);
    } else {
    // Respond with an error message if the format is invalid
    USART_sendString("Invalid setOff command format. Use: setOff HH:MMAM/PM\n");
}


} else if (strncmp(cmd, "setTime", 7) == 0) {
// Parse the "setTime" command to set the current RTC time
char *token = strtok(cmd, " "); // Extract "setTime"
```

```
token = strtok(NULL, " ");     // Extract time argument (e.g., "HH:MM:SSAM/PM")


if (token) {
    uint8_t hour = 0, min = 0, sec = 0; // Variables to store parsed time
    char *colon1 = strchr(token, ':');  // Find the first colon
    char *colon2 = (colon1 ? strchr(colon1 + 1, ':') : NULL); // Find the second colon


    if (colon1 && colon2) {
        *colon1 = '\0';
        *colon2 = '\0';
        hour = (uint8_t)atoi(token);         // Extract hour
        min = (uint8_t)atoi(colon1 + 1);     // Extract minute
        sec = (uint8_t)atoi(colon2 + 1);     // Extract second


        // Check for AM/PM format and adjust time if necessary
        char *ampm = colon2 + strlen(colon2 + 1);
        if (ampm) {
            if (strstr(ampm, "PM")) {
                if (hour < 12) hour += 12;   // Convert PM to 24-hour
                } else if (strstr(ampm, "AM")) {
                if (hour == 12) hour = 0;    // Handle midnight case
            }
        }


        // Validate the time components
        if (hour < 24 && min < 60 && sec < 60) {
            DS1307_SetTime(hour, min, sec); // Set the time on the DS1307 RTC
            snprintf(buffer, sizeof(buffer), "Time set to %02d:%02d:%02d\n", hour, min, sec);
            USART_sendString(buffer);
            } else {
            USART_sendString("Invalid time. Ensure HH:MM:SS is valid.\n");
        }
        } else {
        // Respond with an error if the format is invalid
        USART_sendString("Invalid time format. Use HH:MM:SS[AM/PM] or HH:MM:SS.\n");
    }
    } else {
    // Respond if no time argument is provided
    USART_sendString("Please provide time in HH:MM:SS[AM/PM] or HH:MM:SS format.\n");
}


} else if (strncmp(cmd, "currentTime", 11) == 0) {
// Display the current time from the DS1307 RTC
uint8_t h, m, s; // Variables to store the current time
DS1307_GetTime(&h, &m, &s); // Retrieve the current time


// Convert to 12-hour format with AM/PM
char ampm[3];
uint8_t hour12 = h;
if (hour12 == 0) {
    hour12 = 12;          // Midnight (00:xx) -> 12:xx AM
    strcpy(ampm, "AM");
    } else if (hour12 == 12) {
    strcpy(ampm, "PM");    // Noon (12:xx) -> 12:xx PM
    } else if (hour12 > 12) {
    hour12 -= 12;          // Convert PM hours
    strcpy(ampm, "PM");
    } else {
    strcpy(ampm, "AM");    // Morning hours remain as AM
}


// Respond with the formatted time
snprintf(buffer, sizeof(buffer), "Current RTC Time: %02d:%02d:%02d %s\n", hour12, m, s, ampm);
USART_sendString(buffer);


} else if (strcmp(cmd, "Off") == 0) {
// Turn off the fan
Set_Fan_Speed(0); // Set PWM duty cycle to 0
fanState = false; // Update fan state
USART_sendString("Fan Turned Off\n");
```

```
    } else if (strcmp(cmd, "On") == 0) {
    // Turn on the fan
    fanState = true; // Update fan state
    uint8_t dutyValue = (uint8_t)((dutyCyclePercentage * 255UL) / 100UL); // Calculate PWM duty cycle
    Set_Fan_Speed(dutyValue);
    snprintf(buffer, sizeof(buffer), "Fan Turned On Duty Value: %d%%\n", dutyCyclePercentage);
    USART_sendString(buffer);


    } else {
    // Handle unknown commands
    USART_sendString("Unknown command\n");
    snprintf(buffer, sizeof(buffer), "Received command: '%s'\n", cmd);
    USART_sendString(buffer);
}



}
```

ISR(USART_RX_vect) { char c = UDR0; // Read the received character from the USART data register

```
// Check if a previous command has been fully processed
if (commandReady == 0) { // Only accept input if the previous command is not being processed
    if (c == '\n' || c == '\r') { // Check if the received character is a newline or carriage return
        // End of the command input
        commandBuffer[commandIndex] = '\0';
        commandReady = 1; // Set the flag to indicate that a complete command is ready
        commandIndex = 0; // Reset the index for the next command
        } else if (commandIndex < CMD_BUFFER_SIZE - 1) {
        // Add the received character to the command buffer if there is space
        commandBuffer[commandIndex++] = c;
        } else {
        // If the buffer overflows, reset the index
        commandIndex = 0;
    }
}



}
```

ISR(PCINT0_vect) { static uint8_t lastButtonState = 0; // Store the last known state of the button (assume pull-up, active HIGH)

```
// Read the current state of the button (active LOW logic: LOW when pressed, HIGH when released)
uint8_t currentButtonState = (PINB & (1 << BUTTON)); // Check the state of the button pin


// Check for a rising edge: last state was LOW (pressed), current state is HIGH (released)
if (currentButtonState && !lastButtonState) {
    // Button was released (transition from LOW to HIGH)


    // Toggle the fan state
    if (!fanState) {
        // If the fan is currently off, turn it on
        uint8_t dutyValue = (uint8_t)((dutyCyclePercentage * 255UL) / 100UL); // Calculate PWM duty cycle
        fanState = true; // Update the fan state to ON
        Set_Fan_Speed(dutyValue); // Set the fan speed
        USART_sendString("Fan Turned On\n"); // Send a status message over USART
        } else {
        // If the fan is currently on, turn it off
        Set_Fan_Speed(0); // Set PWM duty cycle to 0 to stop the fan
        fanState = false; // Update the fan state to OFF
        USART_sendString("Fan Turned Off\n"); // Send a status message over USART
    }
}


// Update the last known button state to the current state
lastButtonState = currentButtonState;


}
```