School of Science and Engineering

Department of Electrical and Computer Engineering


EGC433-01: Embedded Systems

Instructor: Kerry Ford

# Final Project: HVAC System

| Name | Major |
|---|---|
| Edward Atristain | CE |
| Kevin Lopez | CE |
| Philip Hanhurst | CE |
| Brian Lopez | EE |

# Abstract

This project takes everything learned throughout the course from learning Liquid Crystal Display (LCD), Universal Synchronous Asynchronous Receiving Transmitting (USART) set up, Interrupts, Pulse width modulation (PWM), Serial Peripheral Interface (SPI), Proportional Integration Delay (PID), and Bluetooth setup. Using all these principles they will be used to create a functional HVAC system to simulate a thermostat across 3 rooms that can cool and heat dependent on the requested temperature.

# Table of Contents

# Introduction

This project entails the design and development of a multi-microcontroller HVAC (Heating, Ventilation, and Air Conditioning) control system with STM32F446RE development boards. The system has a single master controller and two slave controllers, and each of them senses and controls local temperature conditions. The devices communicate using the SPI protocol, where the master can broadcast target temperatures as well as request sensor data from each of the slaves. The master device also interfaces to a Bluetooth module and USB serial communication for debugging and external input and displays current and target temperatures on an LCD.

The main objective is to maintain a user-specified target temperature on all the controllers using feedback mechanisms based on temperature readings from analog sensors. The system utilizes PI control, a simplified form of PID (Proportional-Integral-Derivative), to adjust the PWM output used to drive a heating resistor and a cooling fan dynamically. This closed-loop mechanism allows for reactive and stable temperature control. The utilization of ADCs, PWM, timers, interrupts, and serial communication accentuates the real-time capabilities of embedded systems and their ability to operate in synchronous mode in distributed environments.

# Theory

## LCD Interface

The LCD interface uses the four-pin mode of the 1602a 16x2 character LCD. The 1602a and LCD displays like it are very common in embedded spaces and can be seen in many places in daily life, such as on vending machines or cash registers. They have numerous benefits over more complex displays. For one, they work very well with more underpowered microcontrollers, such as the ATmega chips found on Arduino devices, since they only need to be provided with per-character data, and not per-pixel data like on a more complex LCD. Though this gives them the limitation of not being able to display complex graphics, this makes them very adept for the purposes of displaying text. The 1602 also supports up to 8 programmable characters and even comes pre-programmed with a small set of non-English characters.

The 1602 has 8 data pins, 2 control pins, and an enable pin which functions as a clock. The two control pins are RS and R/W, which are Register Select and Read/Write respectively. R/W controls the direction in which data flows. When it is high, the data pins will output either the busy flag and the current address, or the contents in RAM at the current address. The RS pin controls whether an instruction is being passed, or data to be written or read from the current address.

Getting the LCD working on the F446 requires a bit of effort. The microcontroller does not have dedicated hardware for it like it does with SPI and $I^2C$. As a result, the signal for the 1602 must be “bit-banged”, which is the term given to implementing a hardware protocol in software using GPIO. The downside of this is that every time data is sent to the LCD, a blocking software delay must be used, which slows down the process immensely, and takes time away from other code that could be running. This does not end up being a significant issue in this case though, as most of the program’s time is spent idle anyway.

To initialize the display, the reset sequence must first be run. This is done by sending hex 0x3 three times with a 1 millisecond delay each time. After this, the display must be initialized with 4-bit mode by sending 0x2 over the 4 data pins with the control pin low, and then sending a function set command with the DL b Overall, the process takes about 33 milliseconds, which is hard for humans to perceive, but is an eternity compared to many of the other setup steps. Sending a command takes about a millisecond or two, which blocks all other code from running in that period of time. Writing to display is done by writing a byte to the desired address. Addresses 0x80 through 0x8F relate to the first line, and 0xC0 through 0xCF to the second line. Though the character set for the 1602 is unique, it is compatible with the ASCII English characters and can be used to display most English text.

## Timers

Timers are essential components in microcontrollers and digital systems used to measure time intervals, generate periodic events, and synchronize operations. At their core, timers are counters that increment, or decrement based on a clock source, and they can be configured to trigger events when they reach a specific value, known as the auto-reload or compare value. By using prescalers, timers can divide the system clock to suit a wide range of timing needs from microseconds to seconds allowing developers to manage precise delays, pulse generation, or event scheduling without relying on CPU-intensive software loops.

Timers are foundational in implementing critical embedded functionalities such as Pulse Width Modulation (PWM), periodic interruptions, and time-based measurements. For example, they can generate consistent clock signals to toggle outputs or trigger Analog-to-Digital Conversions (ADCs) at precise intervals. In real-time control systems, timers enable deterministic behavior by ensuring that tasks occur at exact time intervals, regardless of processor workload. Their hardware-based operation allows for accurate and low-power timekeeping, which is crucial in applications ranging from motor control and communication protocols to digital clocks and sensor sampling.

## Interrupts

In this program, several functions must run simultaneously with little or no delay and must not interfere with one another. Rather than putting all the code within the main loop, interrupts are employed to improve system efficiency and responsiveness. Interrupts enable the microcontroller to stay in low power or idle mode and respond only upon the occurrence of certain events. This method conserves power, reduces CPU usage, and gives faster response time than can be achieved by continuously polling for events in the main loop.

There are several types of interrupts available in the STM32F4 microcontroller such as external interrupts caused by GPIO pins ( a button press), timer interrupts for timely tasks such as PWM generation, and peripheral interrupts by modules such as USART, SPI, or ADC. Each interrupt has its own corresponding interrupt vector that leads the processor to the corresponding interrupt service routine (ISR) whenever there is an event.

At the heart of the interrupt system is the Nested Vectored Interrupt Controller (NVIC), which manages all interrupts in the Cortex-M4 processor. The NVIC is responsible for enabling or disabling certain interrupts, setting the priority for each interrupt, and allowing nested interrupt execution, allowing higher-priority interrupts to interrupt lower-priority ones. This priority execution ensures that important operations are carried out immediately without being delayed by less important operations.

To implement an interrupt, the process typically involves configuring the related peripheral or GPIO, enabling the interrupt within that module, activating it in the NVIC, setting its priority, and writing the corresponding ISR. The ISR should contain bare minimum code for processing the event, followed by clearing the interrupt flag to prevent repeated triggering. Interrupts in this project are utilized to handle button inputs, handle ADC conversions of temperature values, read data over SPI or USART, and modify control signals using PWM timers. Interrupts make the system more modular, efficient, and responsive to real-time without CPU overloading.

## ADC

The ADC (Analog-to-Digital Converter) is a building block in this HVAC system, which takes analog temperature readings from sensors and converts them into a digital value that the microcontroller can process. The STM32F446RE has a number of high-resolution ADCs that are capable of sampling analog signals at 12-bit resolution, the output ranges from 0 to 4095. This provides fine granularity in measuring temperature, which is necessary for precise control in a PID or PI feedback loop.

In this arrangement, ADC2 is set to read the voltage output of a temperature sensor (an LM35) connected to pin PC0. The ADC is called on at regular intervals by a timer interrupt (TIM3), ensuring that the sampling is done on a regular basis regardless of other activities being executed on the microcontroller. This regular sampling provides real-time temperature monitoring and returns feedback to the control logic in real time.

To configure the ADC, the GPIO pin where the sensor is connected is set to analog mode and the ADC peripheral is configured with appropriate settings such as sampling time, resolution, and alignment. The temperature is calculated by scaling the ADC output with respect to the reference voltage, usually 3.3V.

This computed temperature is then subtracted from the master device target temperature. This resulting error goes to the PI controller, which adjusts the PWM output according to this error so that the system is heated or cooled. This feedback process is all taken care of within the timer interrupt handler and enables fast and deterministic control without congesting the main loop.

With the delegation of temperature sampling to ADC hardware and the use of interrupts, the system implements accurate, timely, and low-overhead temperature control—a key requirement for the HVAC feature.

## PWM

Pulse Width Modulation (PWM) is a digital control technique that simulates the action of an analog signal by alternating a digital output between high and low at high rate and constant frequency. Duty cycle, being the most significant PWM parameter, is the ratio of time the signal spends high in each cycle. By varying the duty cycle, PWM effectively controls the average power delivered to a device without the inefficiencies of analog voltage regulation. It is especially useful in embedded and power electronics systems where precise control and energy efficiency are essential.

PWM is used in motor speed control, LED brightness control, audio signal generation, and power supply in switching regulators. As the signal is either one of two extremes, power dissipation within the switching devices like transistors or MOSFETs is minimized to a bare minimum, leading to improved energy efficiency. PWM is easily implemented within microcontrollers using hardware timers, and hence it is a stable and low-cost means of real-time system fine-grained control. Its ability to modulate power with low heat dissipation and high resolution has made PWM a standard practice in modern embedded and control applications.

In STM32 microcontrollers, PWM is typically generated through the high control capabilities of the general-purpose or advanced timers. These timers provide users with high

accuracy in setting the PWM duty cycle and frequency through the prescaler configuration, auto-reload values, and capture/compare registers. Hardware PWM generation allows guaranteed timing on the output and reduces processor overhead so that other functions can be performed by the CPU. The PWM channels may be tied to different GPIO pins so that a number of devices or actuators may be addressed at once.

## SPI Serial Communication

In this system, there are several devices running and they need to run together. Therefore, some sort of communication must be set up between these devices. The system requires one device to have the ability to set a temperature to aim for and the others need this information as well while also sharing their current temperatures. This results in the necessity of serial communication and the logic required for the sharing of information is most popularly known as “Master/Slave” logic.

Serial Communication is a method of communication where only 1 bit of information can be sent at a time from one device to another. Serial communication is commonly used for being compact and power-effective for embedded systems utilizing it. There are several types of serial communication, however this HVAC system uses SPI which uses a clock signal, making it synchronous, and requires 2 separate connections for transmitting and receiving information, also known as Full Duplex communication. Stated prior, the system will use “Master/Slave” logic. The names “Master” and “Slave” define the roles of the host primary device and secondary devices respectively. Although the types of slave devices can vary, such as sensors or displays, all devices using SPI are microcontrollers.

For SPI to function, 4 lines must be connected between a master and a slave device. First, the full duplex communication, which is the two lines that allow the devices to transmit and receive information in both directions, one direction for each line. The SPI full duplex lines are better known as “Master Out – Slave In” (MOSI) for the transmit line of the master device and the receive line of the slave device and “Master In- Slave Out’ (MISO) for the transmit line of the slave device and the receive line of the master device. SPI is synchronous therefore all devices using SPI communication must using the same clock which is the third line’s purpose. The clock is generated from the primary device and is connected to all slave devices. Since serial communication sends a single bit at a time, the clock maintains this time and allows all processes required to transmit and receive all data correctly. Lastly, the fourth line is known as the Slave Select (SS) which enables or disables the communication between the slave and the master devices. When there are multiple slave devices in a system, the MOSI, MISO, and clock connections are connected in parallel while the slave devices select have their own separate connections with the master. This allows the master to enable and disable slave devices separately so in the case one slave device needs to be transmitted data, the other slave devices

can be disabled so data is not passed to them. It is important to know that slave devices can never voluntarily send data to the master device, the master must request information. In the most general sense, the master is in control of all communicative actions between devices.

## Control Logic

Proportional-Integral-Derivative (PID) control is a fundamental control strategy widely used in engineering and automation to maintain a desired output level in dynamic systems. The primary goal of PID control is to continuously calculate an error value as the difference between a system's desired setpoint and its measured process variable, and to apply a correction based on proportional, integral, and derivative terms. The proportional (P) component responds to the current error, producing a correction that is directly proportional to the magnitude of the deviation. The integral (I) component accounts for the accumulation of past errors over time, helping to eliminate steady-state errors that the proportional term alone cannot resolve. The derivative (D) component predicts future error based on the rate of change, adding damping to reduce overshoot and improve system stability.

Together, these three components form a feedback loop that allows PID controllers to provide accurate and responsive control in systems that are subject to disturbances, delays, or nonlinearity. PID control is versatile and can be applied across a wide range of domains, including temperature regulation, motor speed control, robotics, process control in chemical plants, and flight control systems. Its popularity stems from its relatively simple implementation and the ability to tune the three gain parameters to match specific performance requirements such as speed of response, stability, and accuracy.

For this HVAC control system, the control logic is managed by a PI controller run inside the timer interrupt routine. The controller calculates an error value as a difference between a target temperature entered by the user and the most recent sensor reading. The proportional component remedies the transient error, and the integral component accumulates past errors to cancel out long-term steady-state biases.

The final control output is used to vary the duty cycle of a Timer 2-driven PWM signal directly controlling the heating and cooling elements attached to each board. The system has precise, deterministic control over the environment by modifying the PWM signal in real time in the interrupt handler. The absence of derivative term minimizes tuning and avoids noise amplification, which is largely not required in thermal systems that vary relatively slowly. This design ensures that even with asynchronous external inputs or varying system load, the HVAC control is stable and reliable on all devices.

## UART & Bluetooth

UART (Universal Asynchronous Receiver-Transmitter) is a serial protocol allowing for transmitting data between two devices. It is primarily used for transmitting ASCII text, and for debugging embedded devices. UART, and its sibling USART, are very common in the embedded space as a method of providing a text terminal for accessing embedded devices, among many other uses.

Like SPI and I2C, UART sends data in 8-bit frames. Each frame is signified by a start bit and a stop bit. However, UART has many differences compared to SPI and I2C. For example, UART requires that both ends of the communication provide their own clocks. Usually, this clock is specified in software or is hard-wired into the device. While SPI and I2C are master-slave protocols, in UART, there is no set master or slave. Either side can initiate communication with UART, which makes it ideal for sending text data between both devices. SPI and I2C on the other hand can be multiplexed, allowing for multiple slave devices to be connected to one master device on a single data and clock bus.

Bluetooth is a protocol for transmitting data wirelessly. Though it has a wide number of use cases, it is most well-known for its use in audio transmission, such as portable speakers or headphones. However, there exist many UART Bluetooth modules for embedded devices. These modules are useful for sending and receiving text data with a device such as a smartphone or computer.

# Procedure

For the HVAC system there are differences and similarities between the pin setups and functions of the 3 microcontrollers used. However, it can be easily stated that the master microcontrollers contain many more functions that allow the temperatures to be displayed and set. The slave microcontrollers, on the other hand, will have similar functionality to the master involving the ADC reading of the temperature sensors and the methods used to heat and cool the sensor to aim for the desired temperature set by the master.

## Master: Displaying and Setting Temperature

The master device takes input over USB and Bluetooth UART devices and displays the current and desired temperatures using the LCD. The USART devices can send commands `tempXX` and `getVal` to set the desired temperature and get the current temperature respectively. Setting the temperature runs a function which updates the LCD with the most current information. This function is also run whenever the temperature sensors are read, or when the current temperature is requested from the slave devices.

For this project, the pins PC4 through PC7 were used for data transfer, and the pins PB6 and PB7 were used for RS and enable respectively. The benefits of using PC4-PC7 are that they line up very nicely with the LCD's data lines, meaning very little must be done to manipulate the data or command for transfer. Several helper functions were written to make using the LCD much easier. For example, `LCD_write_string()` takes a string to print and a line to write it to.

For the purposes of this project, the LCD display was used to display the current and desired temperatures. To format each line of the LCD, the C standard function `sprintf()` was used to write the formatted text for each line to a buffer string, which is then written to the respective line using the previously mentioned `LCD_write_string()` function.

To initialize UART on the nucleo board, the proper pins need to be set to their alternate function. For USB communication, this means initializing USART2 over the alternate functions of PA2 and PA3, where PA2 functions as the transmit pin, and PA3 functions as the receive pin. The drawback of this is that PA2 and PA3 cannot be used while USB serial is enabled. This does not prove to be an issue, however. The Baud Rate Register (BRR) of USART2 is set to hex 0683, which sets the baud rate to 9600 bits per second (16MHz / 0x0683 is approximately 9600). Initializing USART3 for Bluetooth communication is very similar, except for the use PC10 and PC11 for TX and RX respectively.

Most UART functions got abstracted away through some higher-level functions, such as `USART_write_string()`, which takes in a USART I/O pointer and a string to transmit. This made transmitting debug information over USB much easier, giving vital information on what was initialized properly, and what code could have potentially halted the program. These helper functions were placed in their own special

Though both USB and Bluetooth can be used over UART, the way text is received has some subtle differences between the two. This mostly comes down to the terminals being used. Most USB terminals transmit text to the microcontroller one character at a time, triggering the interrupt every time a character is sent. On the other hand, most Bluetooth terminals will wait until a carriage return or newline is entered before transmitting the whole block of text. This proved to be a problem when trying to implement the exact same method of input as with USB. Ultimately though, this did not affect transmitting text over Bluetooth, only receiving.

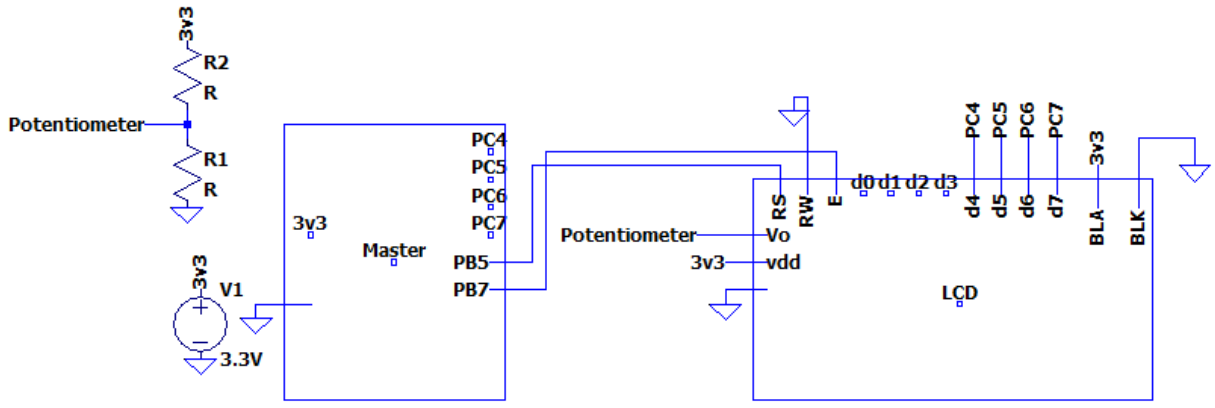


Figure 1: Master to LCD connections

## HVAC Setup

Proportional-Integral-Derivative (PID) control is a foundational technique in embedded systems used to regulate variables such as temperature, speed, or position with high accuracy. In the code above, a PI control strategy using both proportional and integral components—is employed to dynamically adjust the output of a PWM signal in response to temperature error. The proportional term (`kp = 4.0`) provides immediate correction based on the current difference between the target (`requested_temp`) and measured temperature (`current_temp`). The integral term (`ki = 0.2`) accumulates error over time, allowing the system to correct for sustained deviations and eliminate steady-state error. This combination enables smooth and adaptive control behavior as the system strives to reach and maintain the desired setpoint.

On each periodic interrupt triggered by Timer 3 (TIM3), the microcontroller samples the analog temperature input using ADC2, computes the error, updates the integral, and calculates a new control output. This output is used to modify the CCR1 register of TIM2, thereby adjusting the PWM duty cycle that controls power delivery to a heating element or motor. By tuning kp and ki, the system balances responsiveness and stability, demonstrating how PID-based control can be efficiently implemented in real-time embedded applications to maintain precise environmental conditions.

```
while (1) {
if (tim3_flag) {
  tim3_flag = 0;

  float analog = (adc_result * 3.3f) / 4095.0f;
          current_temp[0] = analog * 100 * 1.8f + 32;

          error = requested_temp - current_temp[0];
          integral += error * (PERIOD / 1000.0f);
          float output = kp * error + ki * integral;

          // Flip the sign for cooling if needed
          float pwm_val = (output < 0) ? -output : output;

          // Cap output
          if (pwm_val > 1000) pwm_val = 1000;

          if (current_temp[0] < (requested_temp - TOLERANCE)) {
                    TIM5->CCR1 = (int)(pwm_val * 100); // Heating
                    TIM2->CCR1 = 0;
          } else if (current_temp[0] > (requested_temp + TOLERANCE)) {
                    TIM2->CCR1 = (int)(pwm_val * 100); // Cooling
                    TIM5->CCR1 = 0;
          } else {
                    TIM2->CCR1 = 0;
                    TIM5->CCR1 = 0;
          }
```

Figure 2: PI-based control

In the context of the code above, PWM is implemented using Timer 2 (TIM2) on an STM32 microcontroller to regulate the power delivered to the DC motor and Timer 5 (TIM5) for the heating element based on temperature feedback. The duty cycle of the PWM signal, which is the percentage of time the signal remains high during one period, is adjusted dynamically based on a Proportional-Integral (PI) control loop. As the sensed temperature deviates from the user-defined target (`requested_temp`), the PI controller calculates an error and adjusts the output, accordingly, increasing or decreasing the duty cycle to bring the temperature closer to the desired setpoint.

```
159.    RCC->APB1ENR |= (1 << 0); // TIM2
160.    TIM2->PSC = 160 - 1;
161.    TIM2->ARR = 1000 - 1;
162.    TIM2->CCMR1 = 0x0060;
163.    TIM2->CCER = 1;
164.    TIM2->CCR1 = 1;
165.    TIM2->CR1 = 1;
```

Figure 2a: TIM2 Setup for PWM

In the code, the PWM signal is generated by configuring TIM2 in PWM mode 1, and the CCR1 register is updated every control cycle to reflect the new output level computed by the PI algorithm. This seamless integration between analog temperature sensing and digital PWM output illustrates the fundamental principle of embedded control systems—using precise timing and feedback to create responsive, adaptive behavior in real-time applications.

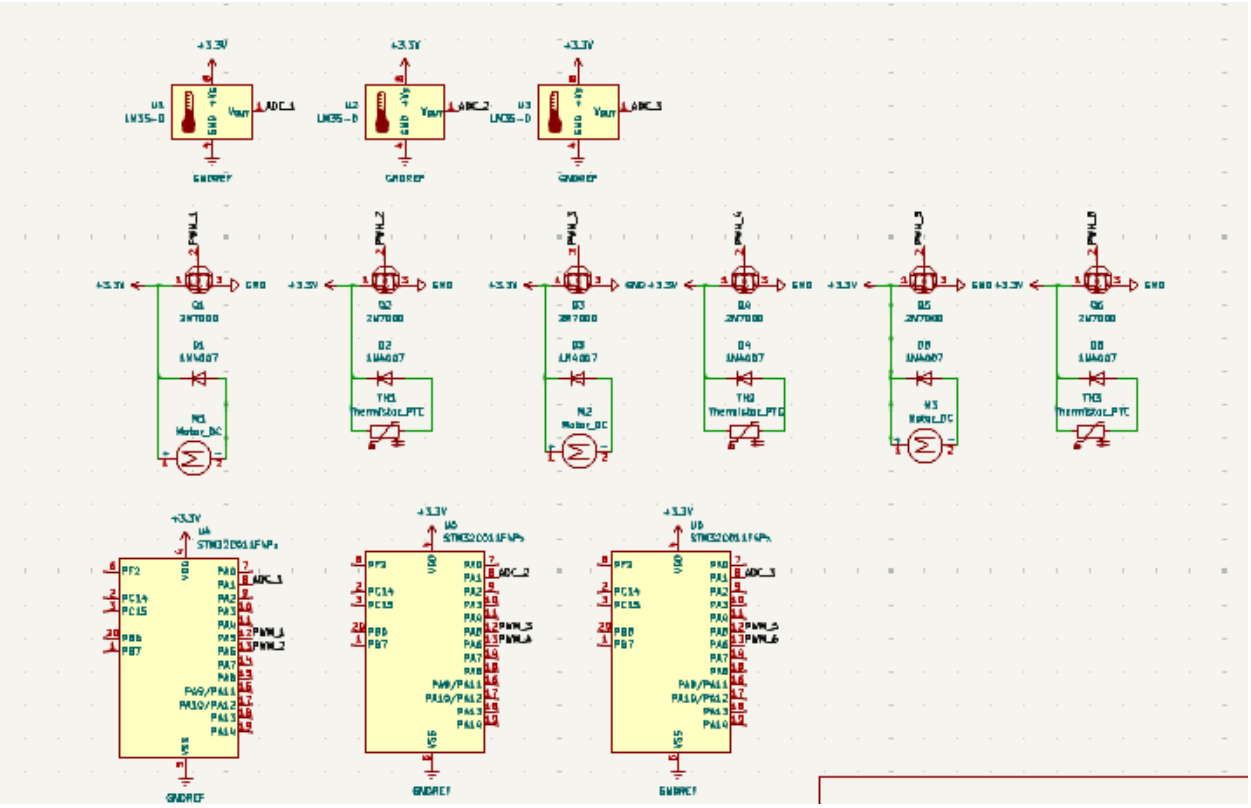

Figure 3: Temperature Sensor, Cooling fan, and Heating Element connections to the 3 Microcontrollers

## Master and Slave Communication

In the provided code, SPI communication is used to transmit data from the master microcontroller to two distinct slave devices. SPI operates as a full-duplex, synchronous serial communication protocol that involves a master device initiating communication and slaves responding. In this implementation, SPI1 is configured as the master, using standard SPI lines: SCK, MISO, and MOSI. The code utilizes manual control of the slave select lines through GPIO pins PA4 and PA1 to determine which slave is active during a data exchange. By default, all slaves are deselected by setting their respective SS lines high using GPIOA's BSRR register. When the master needs to communicate with a specific slave, it pulls the corresponding line low, effectively selecting the target device.

To communicate with Slave A (connected to PA4), the `select_slave_A4()` is called, which first deselects all slaves and then sets PA4 low. Data is then sent over SPI1 using `SPI1_write_byte ()` or `SPI1_xfer ()` to issue commands or transmit temperature setpoints. Once the transfer is complete, PA4 is set high again to release the slave. Similarly, Slave B is selected using `select_slave_A1()` which pulls PA1 low. This method of toggling slave select

lines manually allows the master to control multiple slaves over a single SPI bus. The master sends a command byte followed by a data byte to each slave, enabling tasks such as synchronizing requested temperatures or reading back current values.

```
82. static uint8_t SPI1_xfer(uint8_t out) {
83.   while (!(SPI1->SR & SPI_SR_TXE));
84.   *(volatile uint8_t *)&SPI1->DR = out;
85.   while (!(SPI1->SR & SPI_SR_RXNE));
86.   return *(volatile uint8_t *)&SPI1->DR;
87. }
```

Figure 4: Master's SPI1_xfer Function that sends the Slave Devices Commands

```
 89. void write_temps(void) {
 90.   select_slave_A4();
 91.       delayMs(1);
 92.   SPI1_xfer(0x10);
 93.   SPI1_xfer(requested_temp);
 94.   delayMs(1);
 95.   GPIOA->BSRR = (1 << 4);
 96.
 97.   select_slave_A1();
 98.       delayMs(1);
 99.   SPI1_xfer(0x10);
100.   SPI1_xfer(requested_temp);
101.   delayMs(1);
102.   GPIOA->BSRR = ( 1 << 1);
103.       update_display();
104. }
105. void read_temps(void){
106.                                     select_slave_A4();
107.                                     delayMs(1);
108.             SPI1_xfer(0x1F);
109.                                     delayMs(1);
110.             uint8_t val = SPI1_xfer(0x00);
111.
112.             delayMs(1);
113.             deselect_all_slaves();
114.             current_temp[1]=val;
115.             val=0;
116.
117.                                     select_slave_A1();
118.                                           delayMs(1);
119.             SPI1_xfer(0x1F);          // Send command 0x10 to Slave
120.                                     delayMs(1);
121.             uint8_t val2 = SPI1_xfer(0x00);
122.             delayMs(1);
123.             // Optional: send or receive more bytes here...
124.             GPIOA->BSRR = (1U << 1);   // Set PA1 HIGH (set bit 1)
125.             current_temp[2]=val2;
126.             val2=0;
127. }
```

Figure 4a: read and write temps methods

As the master microcontroller writes commands and requests to the slave, the slave takes in these specific commands and acts upon them. In the code, there are only two commands that will initiate any actions within the slave. When the command TEMP, defined by 0x10 (Fig.4b), is sent to the slave, the temperature set by the user is updated in `requested_temp` so the slave can use this value for its PID and HVAC system. The GET command, defined by 0x1F, occurs when the master needs to receive the updated temperature value from the slave's temperature sensor to be display on the LCD. To send the current temperature, the slave waits for the transmitter to be empty and will then implement the `current_temp` into the data register coded in *Figure 4c*.

```
40. typedef enum Command {
41.         TEMP = 0x10,
42.         GET  = 0x1F
43. } Command;
```

Figure 4b: SPI Commands defined in Slave

```
141.        if (SPI_has_incoming(SPI1)) {
142.                // Get the command
143.                Command command = SPI1->DR;
144.                char str[64] = {0};
145.                char a[30];
146.                sprintf(str, "Got command: %2X\n", command);
147.                switch (command) {
148.                        case TEMP: {
149.                                SPI_await_incoming(SPI1);
150.                                requested_temp = SPI1->DR;
151.                                TIM2->PSC = ((100 - requested_temp) * 1600) - 1;
152.                                sprintf(a,"%d degrees!\n", requested_temp);
153.                                USART_write_string(USART2, a);
154.                        } break;
155.
156.                        case GET: {
157.                                char c[64];
158.                                int adc = (ADC_read(ADC2) * 3.3) / 4096;
159.                                while(!(SPI1->SR & SPI_SR_TXE));
160.                                SPI1->DR = (uint8_t) current_temp;
161.                                sprintf(c, "POGGERS: %d sent\n", adc);
162.                                USART_write_string(USART2, c);
163.
164.                        } break;
165.                        default: {
166.                                char s[64] = {0};
167.                                sprintf(s, "Bad command: 0x%02X\n", command);
168.                                USART_write_string(USART2, s);
169.                        } break;
```

Figure 4c: Slave Main SPI Communication;

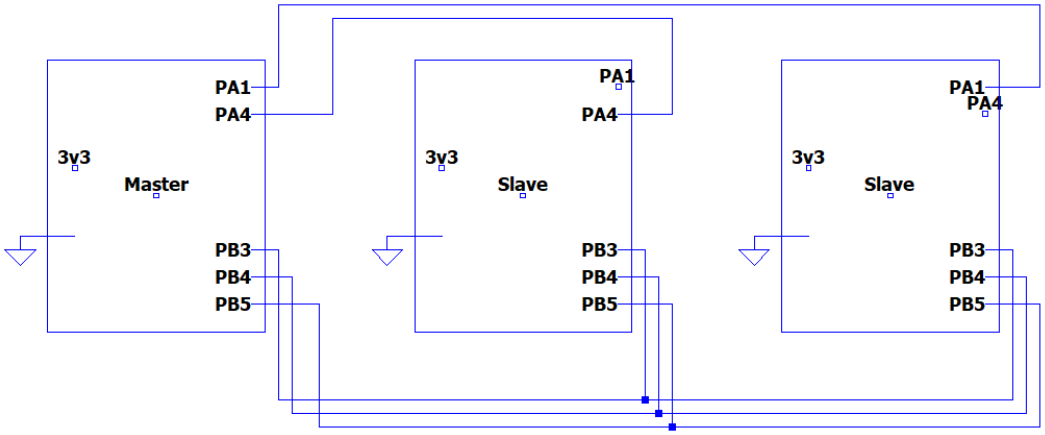


Figure 5: SPI Connection: SysClk (PB3), MISO (PB4), MOSI (PB5), Slave Selects (PA4 & PA1)

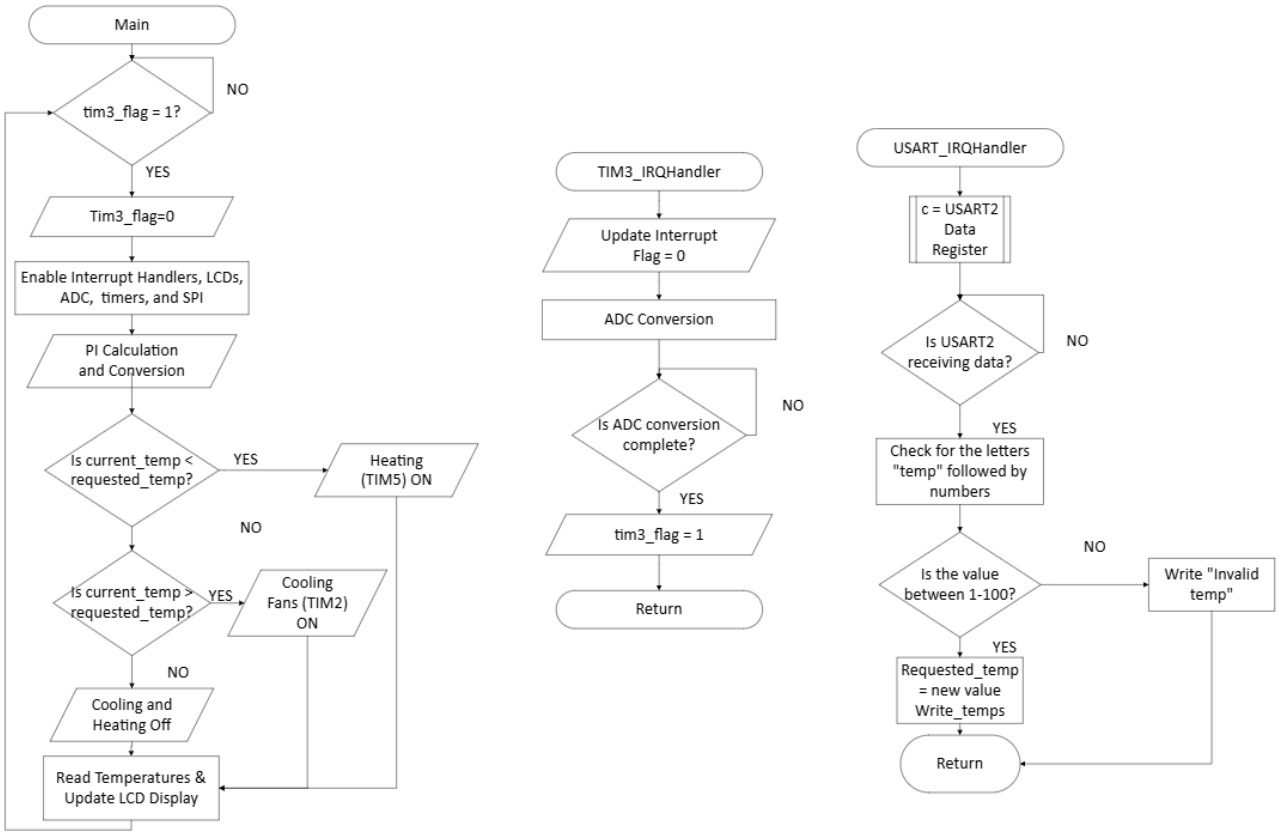


Figure 5a: Flowchart of Master Program: Main and the Interrupts

# Results

The project was a success overall, resulting in a functional HVAC system capable of adjusting temperatures across three separate rooms. Temperature control was achieved using a 3V DC motor for cooling and a PTC heating element for heating (Fig. 6). One of the main challenges encountered during development was enabling both the heating and cooling systems simultaneously, as the components required different current levels. This led to only the master motor functioning reliably, since it operated at a relatively low current draw of 110 mA (Fig. 9).

Significant time was dedicated to designing a circuit capable of delivering sufficient current to the remaining heating elements and motors. An IRFZ44N MOSFET was considered for switching, but its high gate threshold voltage (4–10V) was incompatible with our system's logic-level control. As a result, the final design, shown in *Fig. 3*, was implemented in the completed system (Fig. 8), allowing stable operation within the constraints of our power delivery and control hardware.

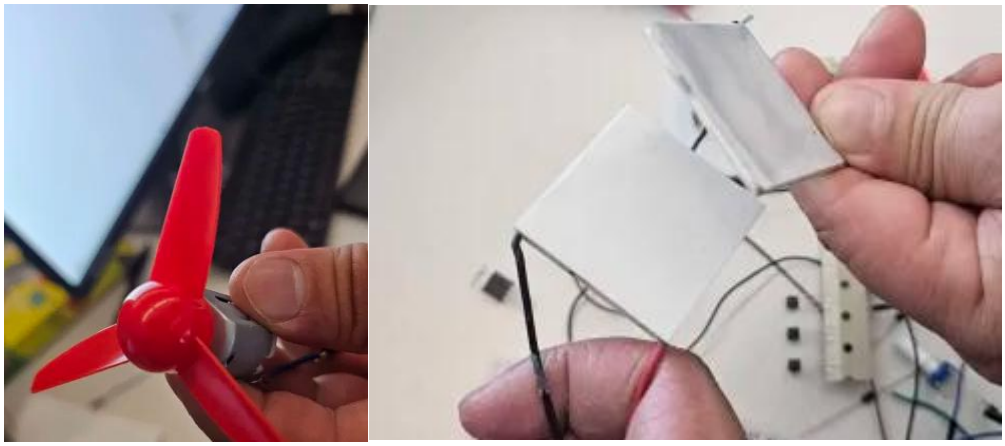

Figure 6: Cooling and Heating Elements for HVAC system

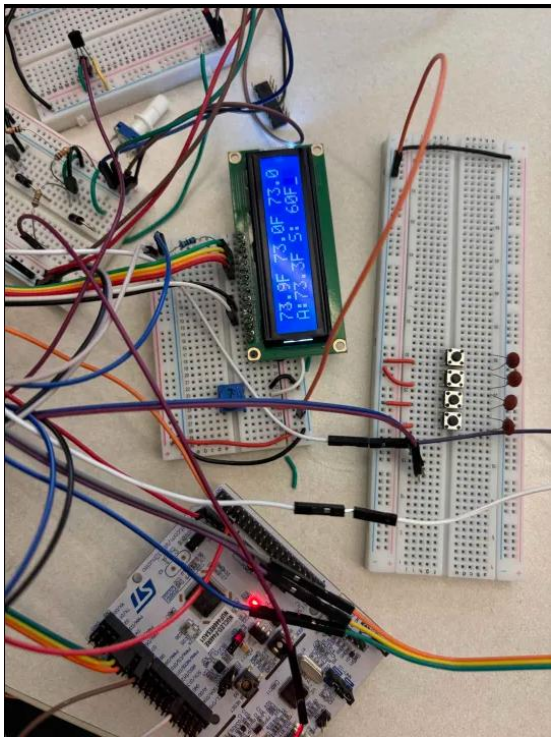


Figure 7: LCD showcasing temperatures in different rooms

In this system, all communication was routed through the master device, which successfully retrieved data from the two slave units and displayed the values on the LCD. Additionally, the master could issue commands to adjust the temperature in each room (Fig. 7). By default, the system maintained a target temperature of 60°F, keeping the motor active to provide continuous cooling unless a new setpoint was received.

For example, when the target temperature was changed to 70°F, the fans gradually slowed as the room temperature approached the setpoint and stopped completely upon reaching it. If the temperature began to rise above 70°F, the system would reactivate the fan to maintain the desired value. When a higher temperature, such as 80°F, was requested, the system recognized the need to switch from cooling to heating, turning off the motor and activating the PTC heating elements to warm the room accordingly.

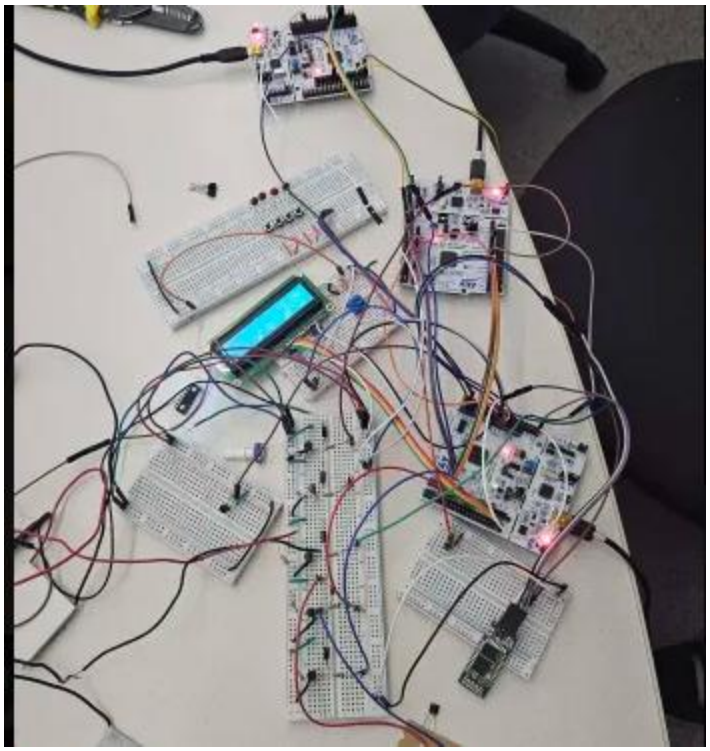

Figure 8: A full photo of our circuit


Figure 9: Current draw showcasing cooling was enabled

Future improvements to this project would include incorporating logic-level MOSFETs in place of the analog MOSFETs initially used. This change would allow the heating elements to be fully driven, enabling more reliable and efficient temperature regulation. Additional refinements would involve addressing electrical clearance and prioritizing interruptions appropriately. One of the major issues encountered was related to push-button interruptions, which would continuously trigger and cause the setpoint value to fluctuate. This behavior disrupted the PID control system, as it could not determine which element heating or cooling should be activated. The issue was mitigated by moving the button-handling logic to a separate code file, isolating it from the primary master control loop.

Another area of focus for future development is the debugging and optimization of the slave devices. While the slave code was largely derived from the master and could acquire and transmit data correctly, it struggled to regulate its PWM output when the master was already drawing current. This resulted in the slave being unable to drive its connected load effectively. Although this may not have been purely a software issue, it points to a potential hardware limitation that warrants further investigation. Addressing this could lead to a more balanced and scalable HVAC control system capable of consistent operation across all zones.

# Conclusion

This project successfully demonstrated the implementation of a fully functional HVAC system using embedded system principles. Through the integration of ADC, PWM, timers, SPI communication, and PI control logic, we were able to manage temperature regulation across three independent zones. The master microcontroller effectively coordinated the system by receiving temperature feedback from each slave, processing control logic, and displaying real-time data to the user via an LCD interface. Moreover, the use of USART and Bluetooth added flexible methods for adjusting setpoints and monitoring system behavior remotely.

While the project met its core objectives, it also revealed opportunities for future enhancement. Challenges related to current handling, MOSFET switching thresholds, and interrupt reliability pointed toward hardware limitations and design trade-offs. Moving forward, replacing analog MOSFETs with logic-level alternatives and optimizing interrupt handling would improve system robustness. Additionally, further debugging of the slave units' PWM control and load management could lead to a more scalable and efficient multi zone HVAC solution. Overall, the project provided valuable hands-on experience with real-time embedded systems and control strategies in a distributed environment.

# Appendix

Appendix A Master Code

```
1. ************************************************
2. * Final Integrated Project / Master.c
3. *
4. * Master SPI controllers with LCD, PID control, USART input, and ADC temperature sensing
5. *
6. * Author: Kevin Lopez, Philip Hanhurst, Brian Lopez, Edward Atristain
7. * Date Last Modified: 5/9/2025
8. *
9. ************************************************/
10.
11. #include “stm32f4xx.h”
12. #include <stdint.h>
13. #include <string.h>
14. #include <stdlib.h>
15. #include <stdio.h>
16.
17. #include “common.h”
18. #include “lcd.h”
19. #include “usart.h”
20. #include “spi.h”
21.
22. #define TOLERANCE 2
23. #define PERIOD 5000
24.
25. volatile int adc_result = 0;
26. volatile int tim3_flag = 0;
27.
28. int requested_temp = 60;
29. float current_temp[3] = {72.0f, 72.0f, 72.0f};
30.
31. const uint8_t TEMP_CHAR[] = {
32.   0b00100,
33.   0b01010,
34.   0b01110,
35.   0b01010,
36.   0b01010,
37.   0b10101,
38.   0b10101,
39.   0b01110
40. };
41.
42. double kp = 4.0, ki = 0.2;
43. //double kp= 4.59, ki=0.54;  // These are for the two slaves PID
44. // double kp2=8.3, ki=0.72;
45. // double kp= 5.55, ki = 0.64;
46. // double kp2= 8.77, ki= 0.8;
47. double kp2= 8.0, ki2 =0.6;
48. double error = 0, prev_error = 0, integral = 0;
49.
50. void deselect_all_slaves(void) {
51.   GPIOA->BSRR = 1 << 4;
52.   GPIOA->BSRR |= 1 << 1;
53. }
54.
55. void select_slave_A4(void) {
56.   deselect_all_slaves();
57.   GPIOA->BSRR = 1 << 20;
```

```
}

void select_slave_A1(void) {
  deselect_all_slaves();
  GPIOA->BSRR |= 1 << 17;
}

void update_display(void) {
  LCD_clear();
  char line0[16] = {0};
  char line1[16] = {0};
  float avg = (current_temp[0] + current_temp[1] + current_temp[2]) / 3.0f;

  24print(line0, "%2.1fF %2.1fF %2.1fF", current_temp[0], current_temp[1], current_temp[2]);
  24print(line1, "A:%2.1fF S:%3dF", avg, requested_temp);

  LCD_write_string(line0, Line0);
  LCD_write_string(line1, Line1);

  char msg[128];
  24print(msg, "Avg: %.1fF | Setpoint: %d\n", avg, requested_temp);
  USART_write_string(USART2, msg);
}

static uint8_t SPI1_xfer(uint8_t out) {
  while (!(SPI1->SR & SPI_SR_TXE));
  *(volatile uint8_t *)&SPI1->DR = out;
  while (!(SPI1->SR & SPI_SR_RXNE));
  return *(volatile uint8_t *)&SPI1->DR;
}

void write_temps(void) {
  select_slave_A4();
      delayMs(1);
  SPI1_xfer(0x10);
  SPI1_xfer(requested_temp);
  delayMs(1);
  GPIOA->BSRR = (1 << 4);

  select_slave_A1();
      delayMs(1);
  SPI1_xfer(0x10);
  SPI1_xfer(requested_temp);
  delayMs(1);
  GPIOA->BSRR = ( 1 << 1);
      update_display();
}
void read_temps(void){
                                    select_slave_A4();
                                    delayMs(1);
            SPI1_xfer(0x1F);
                                    delayMs(1);
            uint8_t val = SPI1_xfer(0x00);

            delayMs(1);
            deselect_all_slaves();
            current_temp[1]=val;
            val=0;

                                    select_slave_A1();
                                          delayMs(1);
            SPI1_xfer(0x1F);          // Send command 0x10 to Slave
                                    delayMs(1);
            uint8_t val2 = SPI1_xfer(0x00);
            delayMs(1);
```

```
123.             // Optional: send or receive more bytes here...
124.             GPIOA->BSRR = (1U << 1);   // Set PA1 HIGH (set bit 1)
125.             current_temp[2]=val2;
126.             val2=0;
127. }
128.
129. void LCD_set_custom_char(int idx, uint8_t* c) {
130.   LCD_command(0x40 | (idx << 3));
131.   for (int I = 0; I < 8; i++) {
132.     LCD_data(c[i]);
133.   }
134. }
135.
136. void init_adc(void) {
137.   RCC->AHB1ENR |=  4;              /* enable GPIOC clock */
138.   GPIOC->MODER |=  3;             /* PC0 analog */
139.
140.       RCC->APB2ENR |= 0x00000200;     /* enable ADC2 clock */
141.
142.             ADC2->CR2=0;
143.             ADC2->SQR3=10;                                                  /* conversion
sequence starts at ch 10 */
144.             ADC2->SQR1=0;                                                       /*
conversion sequence starts at ch 10 */
145.             ADC2->CR2|=1;
146.             ADC2->SMPR1=3;
147.
148.
149.
150.   RCC->APB2ENR |= (1 << 8); // Enable ADC2 clock
151.   ADC2->CR2 = 0;
152.   ADC2->SQR3 = 10; // Channel 10 (PC0)
153.   ADC2->SQR1 = 0;
154.   ADC2->SMPR1 = 3;
155.   ADC2->CR2 |= 1; // Enable ADC2
156. }
157.
158. void init_timer(void) {
159.   RCC->APB1ENR |= (1 << 0); // TIM2
160.   TIM2->PSC = 160 - 1;
161.   TIM2->ARR = 1000 - 1;
162.   TIM2->CCMR1 = 0x0060;
163.   TIM2->CCER = 1;
164.   TIM2->CCR1 = 1;
165.   TIM2->CR1 = 1;
166.
167.   RCC->APB1ENR |= (1 << 1); // TIM3
168.   TIM3->PSC = 16000 - 1;
169.   TIM3->ARR = PERIOD - 1;
170.   TIM3->DIER |= 1;
171.   TIM3->CR1 = 1;
172.
173.       RCC-> APB1ENR |=8;      //Timer 5
174.                   TIM5->PSC = 160-1;            /*divided by 1600*/
175.                   TIM5->ARR=1000-1;       /*1000 ticks a second*/
176.                   TIM5->CNT=0;
177.                   TIM5->CCMR1 = 0x0060;   /* PWM mode 1*/
178.                   TIM5->CCER =1;                              /* PWM channel 1*/
179.                   TIM5->CCR1=1-1; /*set 0% duty cycle*/
180.                   TIM5-> CR1= 1 ;   /* enable timer */
181. }
182.
183. void TIM3_IRQHandler(void) {
184.   TIM3->SR &= ~TIM_SR_UIF;
185.   ADC2->CR2 |= 0x40000000;
```

```
186.   while (!(ADC2->SR & 2));
187.   adc_result = ADC2->DR;
188.   tim3_flag = 1;
189. }
190. /* USART2 Interrupt Handler */
191. void USART2_IRQHandler(void) {
192.   static char input_buffer[32];
193.   static int buffer_index = 0;
194.
195.   if (USART2->SR & 0x0020) {
196.     char c = USART2->DR;
197.
198.     if (c == '\n' || c == '\r') {
199.       input_buffer[buffer_index] = '\0';
200.       if (strncmp(input_buffer, "temp", 4) == 0) {
201.         int val = atoi(&input_buffer[4]);
202.         if (val >= 1 && val <= 100) {
203.           requested_temp = val;
204.           write_temps();
205.         } else {
206.           USART_write_string(USART2, "Invalid temp range (1-100)\n");
207.         }
208.       }
209.       buffer_index = 0;
210.     } else if (buffer_index < sizeof(input_buffer) - 1) {
211.       input_buffer[buffer_index++] = c;
212.     }
213.   }
214. }
215. void SPI1_init(void) {
216.     RCC->AHB1ENR |= 3;              /* enable GPIOB & GPIOA clock */
217.     RCC->APB2ENR |= 0x1000;         /* enable SPI1 clock */
218.
219.     // Configure PB3, PB4, PB5 for SPI1
220.     GPIOB->MODER &= ~0x00000FC0;
221.     GPIOB->MODER |=  0x00000A80;
222.     GPIOB->AFR[0] &= ~0x00FFF000;
223.     GPIOB->AFR[0] |=  0x00555000;
224.
225.     // Configure PA1 and PA4 as outputs for SS
226.     //GPIOB->MODER &= ~0x000C0000;
227.    // GPIOB->MODER |=  0x00040000;
228.     GPIOA->MODER &= ~0x0000030C;
229.     GPIOA->MODER |=  0x00000104;
230.
231.     deselect_all_slaves();
232.
233.     SPI1->CR1 = 0x31C;  // Baud rate, master, CPOL/CPHA = 0, 8-bit data
234.     SPI1->CR2 = 0;
235.     SPI1->CR1 |= 0x40;  // Enable SPI1
236. }
237. int main(void) {
238.             __disable_irq();               /* global disable IRQs */
239.             USART2_init();
240.             RCC->AHB1ENR |=  1;                /* enable GPIOA clock */
241.             GPIOA->AFR[0] |= 0x00100000;       /*PA5 pin for tim2 */
242.     GPIOA->MODER &= ~0x00000C00;    /* clear pin mode */
243.     GPIOA->MODER |=  0x00000800;    /* set pin to output mode */
244.             // Set PA6 to Alternate Function mode
245. GPIOA->MODER &= ~(3U << (6 * 2));    // Clear mode bits for PA6
246. GPIOA->MODER |=  (2U << (6 * 2));    // Set to Alternate Function
247.
248. // Set AF2 (TIM5_CH1) on PA6
249. GPIOA->AFR[0] &= ~(0xF << (6 * 4));  // Clear AF bits for PA6
250. GPIOA->AFR[0] |=  (0x2 << (6 * 4));  // Set AF2 (TIM5)
```

```
251.             LCD_init();
252.             LCD_set_custom_char(0, (uint8_t*)TEMP_CHAR);
253.             init_adc();
254.             init_timer();
255.
256.    SPI1_init();
257.    deselect_all_slaves();
258.    update_display();
259.
260.    NVIC_EnableIRQ(TIM3_IRQn);
261.    NVIC_EnableIRQ(USART2_IRQn);
262.
263.        __enable_irq();                /* global enable IRQs */
264.
265.    while (1) {
266.    if (tim3_flag) {
267.      tim3_flag = 0;
268.
269.      float analog = (adc_result * 3.3f) / 4095.0f;
270.             current_temp[0] = analog * 100 * 1.8f + 32;
271.
272.             error = requested_temp - current_temp[0];
273.             integral += error * (PERIOD / 1000.0f);
274.             float output = kp * error + ki * integral;
275.
276.             // Flip the sign for cooling if needed
277.             float pwm_val = (output < 0) ? -output : output;
278.
279.             // Cap output
280.             if (pwm_val > 1000) pwm_val = 1000;
281.
282.             if (current_temp[0] < (requested_temp - TOLERANCE)) {
283.                          TIM5->CCR1 = (int)(pwm_val * 100); // Heating
284.                          TIM2->CCR1 = 0;
285.             } else if (current_temp[0] > (requested_temp + TOLERANCE)) {
286.                          TIM2->CCR1 = (int)(pwm_val * 100); // Cooling
287.                          TIM5->CCR1 = 0;
288.             } else {
289.                          TIM2->CCR1 = 0;
290.                          TIM5->CCR1 = 0;
291.             }
292.
293.                          // Optional: Debug print
294.                          char c[64];
295.                          sprintf(c, “Temp: %.2f, Output: %.1f, H:%d, C:%d\r\n”, current_temp[0],
output, TIM5->CCR1, TIM2->CCR1);
296.                          USART_write_string(USART2, c);
297.
298.                          read_temps();
299.                          update_display();
300.                    }
301.             }
302.                    }
303.
```

Appendix B Slave code

```
/************************************************
* HVAC System / Slave #1
*
* Program for Slave Microcontroller with NSS of PA4
*
*
* Edward A, Philip H, Kevin L, Brian L
*
************************************************/

#include "adc.h"
#include "spi.h"
#include "stm32f446xx.h"
#include "stm32f4xx.h"
#include "pwm.h"
#include "usart.h"
#include <stdint.h>
#include <stdio.h>

#define TOLERANCE 2
#define USART2_write_string(STR) USART_write_string(USART2, STR)

void ADCx_init();
void TIMx_init(void);
void delayMs(int n);

int requested_temp;
double current_temp;

double error=0, prev_error=0, integral=0;

double kp= 4.0, ki=0.54;
double kp2=8.3, ki2=0.72;
//double kp= 5.55, ki = 0.64;
//double kp2= 8.77, ki2= 0.8;

volatile int adc_result = 0;
volatile int tim3_flag = 0;

typedef enum Command {
    TEMP = 0x10,
    GET  = 0x1F
} Command;

void SPI1_Slave_Init(void) {
    RCC->APB2ENR |= 0x1000; // 0x1000

    // SPI pins: PA4 = NSS, PB3 = SCK, PB4 = MISO, PB5 = MOSI
    GPIOA->MODER &= ~0x00300;    /* clear pin mode */
    GPIOA->MODER |=  0x00200;// PORT A AF for NSS


    GPIOB->MODER &= ~0x00C0;
    GPIOB->MODER |=  0x0080; // AF for SCK
    GPIOB->MODER &= ~0x0300;
    GPIOB->MODER |=  0x0200; // AF for MISO
    GPIOB->MODER &= ~0x0C00;
    GPIOB->MODER |=  0x0800; // AF for MOSI

    //GPIOB->AFR[0] |= 0x00050000; // AF5  for PB9
    GPIOA->AFR[0] |= 0x00050000; // AF5 for PA4
```

```
62.      GPIOB->AFR[0] |= 0x00005000; // AF5 for PB3
63.      GPIOB->AFR[0] |= 0x00050000; // AF5 for PB4
64.      GPIOB->AFR[0] |= 0x00500000; // AF5 for PB5
65.
66.      SPI1->CR1 = 0;        // Clear settings
67.      SPI1->CR1 &= ~0x04;   // Slave mode
68.      SPI1->CR1 &= ~0x02;   // CPOL = 0
69.      SPI1->CR1 &= ~0x01;   // CPHA = 0
70.      SPI1->CR1 &= ~0x0200; // Use hardware NSS
71.      SPI1->CR2 = 0;
72.
73.      SPI1->CR1 |= SPI_CR1_SPE; // (0x40) Enable SPI
74. }
75.
76. void init() {
77.      RCC->AHB1ENR |= 3; // Enable GPIOA and GPIOB clock
78.      USART2_init();
79.      SPI1_Slave_Init();
80. }
81.
82. int main(void) {
83.       /* set up pin PA5 for LED */
84.               __disable_irq();               /* global disable IRQs */
85.     RCC->AHB1ENR |=  1;             /* enable GPIOA clock */
86.
87.               GPIOA->AFR[0] |= 0x00100000;           /*PA5 pin for tim2 */
88.     GPIOA->MODER &= ~0x0000C00;    /* clear pin mode */
89.     GPIOA->MODER |=  0x0000800;   /*set pin to alternate */
90.
91.     GPIOA->AFR[1] |= 0x0000002;             /*PA8 pin for tim5 */
92.     GPIOA->MODER &= ~0x00030000;    /* clear pin mode */
93.     GPIOA->MODER |=  0x00020000;    /* set pin to alternate */
94.
95.
96.               ADCx_init();
97.               TIMx_init();
98.               init();
99.               USART_write_string(USART2, "Initialized!\n");               /* enable UIE */
100.     NVIC_EnableIRQ(TIM3_IRQn);      /* enable interrupt in NVIC */
101.              __enable_irq();                /* global enable IRQs */
102.
103.
104.
105.   while(1)
106.      {
107.              if(tim3_flag)
108.              {
109.                       tim3_flag =0;
110.                       //PI System
111.                       float adc_read = ((adc_result * 3.3) / 4096);
112.                       current_temp=(adc_read*100)*1.8+32;
113.
114.
115.                       error = requested_temp-current_temp; //P equation
116.                       integral+= error * TIM3->ARR/1000; //I equation
117.
118.                       float output = kp*error +ki*integral;
119.
120.                       TIM5->CCR1 = (int)(output * 100);
121.
122.                       char mag[100];
123.                       sprintf(mag, "%.1f F\n\r", current_temp);
124.                       USART2_write_string(mag);
125.              }
126.
```

```
127.                if(TIM5->CCR1 >1000) TIM5->CCR1=1000;
128.
129.     if (current_temp < (requested_temp - TOLERANCE)) {
130.                                     TIM5->CCR1 = (int)(TIM5->CCR1 * 100); // Heating
131.                                     TIM2->CCR1 = 0;
132.      } else if (current_temp > (requested_temp + TOLERANCE)) {
133.                                               TIM2->CCR1 = (int)(TIM5->CCR1 * 100); // Cooling
134.                                               TIM5->CCR1 = 0;
135.       } else { //Turned off when current temp is requested temp
136.                                                         TIM2->CCR1 = 0;
137.                                                         TIM5->CCR1 = 0;
138.                           }
139.
140.
141.                if (SPI_has_incoming(SPI1)) {
142.                                     // Get the command
143.                                     Command command = SPI1->DR;
144.                                     char str[64] = {0};
145.                                     char a[30];
146.                                     sprintf(str, "Got command: %2X\n", command);
147.                                     switch (command) {
148.                                               case TEMP: {
149.                                                         SPI_await_incoming(SPI1);
150.                                                         requested_temp = SPI1->DR;
151.                                                         TIM2->PSC = ((100 - requested_temp) * 1600) - 1;
152.                                                         sprintf(a,"%d degrees!\n", requested_temp);
153.                                                         USART_write_string(USART2, a);
154.                                               } break;
155.
156.                                               case GET: {
157.                                                         char c[64];
158.                                                         int adc = (ADC_read(ADC2) * 3.3) / 4096;
159.                                                         while(!(SPI1->SR & SPI_SR_TXE));
160.                                                         SPI1->DR = (uint8_t) current_temp;
161.                                                         sprintf(c, "POGGERS: %d sent\n", adc);
162.                                                         USART_write_string(USART2, c);
163.
164.                                               } break;
165.                                               default: {
166.                                                         char s[64] = {0};
167.                                                         sprintf(s, "Bad command: 0x%02X\n", command);
168.                                                         USART_write_string(USART2, s);
169.                                               } break;
170.                                     }
171.                }
172.      }
173. }
174. void TIMx_init(void)
175.      {
176.                           RCC-> APB1ENR |= 1; //Timer 2
177.                           TIM2->PSC=160-1;                        /*divided by 1600*/
178.                           TIM2->ARR=1000-1;                                 /*1000 ticks a
second*/
179.                           TIM2->CNT=0;
180.                           TIM2->CCMR1 = 0x0060;   /* PWM mode 1*/
181.                           TIM2->CCER =1;                                                  /* PWM
channel 1*/
182.                           TIM2->CCR1=1-1; /*set 0% duty cycle*/
183.                           TIM2-> CR1= 1 ;   /* enable timer */
184.
185.
186.                           RCC-> APB1ENR |=2;      /*enable TIM3*/
187.                           TIM3->PSC=16000-1;                      /*divided by 1600*/
188.                           TIM3->ARR=10000 - 1;                                  /*resets
every 100ms*/
```

```
189.                        TIM3->DIER |= 1;
190.                        TIM3->CR1=1;
191.
192.                        RCC-> APB1ENR |=8;  //Timer 5
193.                        TIM5->PSC = 160-1;           /*divided by 1600*/
194.                        TIM5->ARR=1000-1;            /*1000 ticks a second*/
195.                        TIM5->CNT=0;
196.                        TIM5->CCMR1 = 0x0060;   /* PWM mode 1*/
197.                        TIM5->CCER =1;                                          /* PWM
channel 1*/
198.                        TIM5->CCR1=1-1; /*set 0% duty cycle*/
199.                        TIM5-> CR1= 1 ;   /* enable timer */
200.      }
201. void ADCx_init()
202. {
203.      /* set up pin PA1 for analog input */
204.     GPIOA->MODER |=  0xC;           /* PA1 analog */
205.
206.               RCC->AHB1ENR |=  4;             /* enable GPIOC clock */
207.     GPIOC->MODER |=  3;             /* PC0 analog */
208.
209.     /* setup ADC2 */
210.
211.               RCC->APB2ENR |= 0x00000200;     /* enable ADC2 clock */
212.
213.               ADC2->CR2=0;
214.               ADC2->SQR3=10;
                  /* conversion sequence starts at ch 10 */
215.               ADC2->SQR1=0;
                           /* conversion sequence starts at ch 10 */
216.               ADC2->CR2|=1;
217.               ADC2->SMPR1=3;
218. }
219. void TIM3_IRQHandler(void) {
220.
221.                                    char a[100]; // holds the %
222.                                    TIM3->SR &= ~TIM_SR_UIF;                   /* clear UIF */
223.
224.                                    ADC2->CR2 |= 0x40000000;        /* start a conversion */
225.                                    while(!(ADC2->SR & 2)) {}       /* wait for conv complete */
226.                                    adc_result = ADC2->DR;
227.                                    tim3_flag = 1;              /* read conversion result */
228.
229.
230. }
231.
232. void delayMs(int n) {
233.      int i;
234.
235.      // Configure SysTick
236.      SysTick->LOAD = 16000; // reload with number of clocks per millisecond
237.      SysTick->VAL = 0;      // clear current value register
238.      SysTick->CTRL = 0x5;   // Enable the timer
239.
240.      for (i = 0; i < n; i++) {
241.               while ((SysTick->CTRL & 0x10000) == 0)
242.                        ; // wait until the COUNTFLAG is set
243.      }
244.      SysTick->CTRL = 0; // Stop the timer (Enable = 0)
245. }
246.
```

Appendix C LCD setup

```c
#include "lcd.h"
#include "common.h"
#include <stdint.h>

// TODO: Move button pin initialization to its own file or function.

/*  LCD Initialization
 * Currently uses PB6 and PB7 for R/S and EN respectively.
 *
 * Uses PC4-PC7 for LCD D4-D7, and PC0-PC3 for the buttons.
 */
void LCD_ports_init(void) {
    RCC->AHB1ENR |=  0x06;          /* enable GPIOB/C clock */

    /* PORTB 6 for LCD R/S */
    /* PORTB 7 for LCD EN */
    GPIOB->MODER &= ~0x0000F000;    /* clear pin mode */
    GPIOB->MODER |=  0x00005000;    /* set pin output mode */
    GPIOB->BSRR = EN << 16;       /* turn off EN */

    // PC4-PC7 for LCD D4-D7, respectively.
    GPIOC->MODER &= ~0x0000FF00;    /* clear pin mode */
    GPIOC->MODER |=  0x00005500;    /* set pin output mode */
}

void LCD_nibble_write(char data, bool control) {
    /* populate data bits */
    GPIOC->BSRR = 0x00F00000;       /* clear data bits */
    GPIOC->BSRR = data & 0xF0;      /* set data bits */

    /* set R/S bit */
    if (control)
        GPIOB->BSRR = RS;
    else
        GPIOB->BSRR = RS << 16;

    /* pulse E */
    GPIOB->BSRR = EN;
    delayMs(1);
    GPIOB->BSRR = EN << 16;
}

void LCD_command(unsigned char command) {
    LCD_nibble_write(command & 0xF0, 0);    /* upper nibble first */
    LCD_nibble_write(command << 4, 0);      /* then lower nibble */

    if (command < 4)
            delayMs(2);         /* command 1 and 2 needs up to 1.64ms */
    else
            delayMs(1);         /* all others 40 us */
}

void LCD_data(char data) {
    LCD_nibble_write(data & 0xF0, true);      /* upper nibble first */
    LCD_nibble_write(data << 4, true);        /* then lower nibble */

    delayMs(1);
}
```

```
 59.
 60. void LCD_set_custom_char(int idx, uint8_t* c) {
 61.     // Set initial CGRAM address
 62.     LCD_command(ADDR(0b01000000 | (idx << 3)));
 63.     for (int i = 0; i < 8; i++) {
 64.         LCD_data(c[i]);
 65.     }
 66. }
 67.
 68. void LCD_init(void) {
 69.     LCD_ports_init();
 70.
 71.     delayMs(20);                /* LCD controller reset sequence */
 72.     LCD_nibble_write(0x30, false);
 73.     delayMs(5);
 74.     LCD_nibble_write(0x30, false);
 75.     delayMs(1);
 76.     LCD_nibble_write(0x30, false);
 77.     delayMs(1);
 78.
 79.     LCD_nibble_write(0x20, false);  /* use 4-bit data mode */
 80.     delayMs(1);
 81.     LCD_command(0x28);          /* set 4-bit data, 2-line, 5x7 font */
 82.     LCD_command(0x06);          /* move cursor right */
 83.     LCD_command(0x01);          /* clear screen, move cursor to home */
 84.     LCD_command(0x0F);          /* turn on display, cursor blinking */
 85. }
 86.
 87. void LCD_clear() {
 88.     LCD_command(CLR);
 89. }
 90.
 91. inline uint8_t lcd_index_to_ptr(uint8_t idx) {
 92.     // Convert memory address pointer to an index into `LCDdata`
 93.     return (idx & 0x0F) | ((idx > 0x0F) ? 0xC0 : 0x80);
 94. }
 95.
 96. void LCD_write_string(char* str, LCDLine line) {
 97.     switch (line) {
 98.         case Line0: {
 99.             LCD_command(ADDR(0x80));
100.         } break;
101.         case Line1: {
102.             LCD_command(ADDR(0xC0));
103.         } break;
104.     }
105.     for (int i = 0; i < 16 && str[i]; i++) {
106.         LCD_data(str[i]);
107.     }
108. }
109.
```