CSI 370 Final Project Report

Matt C

December 9, 2025

1 Executive Summary

This project implements an automated window control system to regulate indoor temperature in a residential space lacking climate control. The system uses a SparkFun RedBoard (Arduino Uno compatible) with a DC motor to control a gear mechanism that interfaces with a window crank handle.

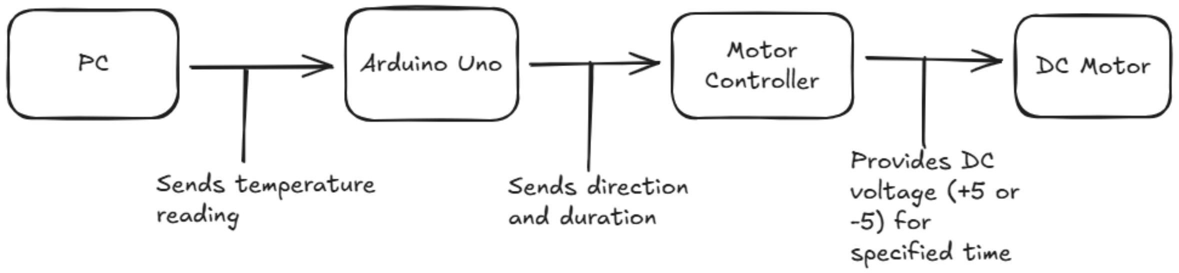
Due to time and hardware constraints, this implementation serves as a proof-of-concept demonstration of the control logic and motor interface, rather than a complete mechanical installation. The project successfully demonstrates the feedback control loop that can be replicated with appropriate window-interfacing hardware and a higher-torque motor in future implementations.

2 Project Overview

2.1 System Description

This demonstration implementation of an IOT-connected window opening device uses a 12v DC motor, controller board, and a SparkFun RedBoard with Arduino C/C++ and AVR assembly to control the motor to move forward and backward by precise amounts, to allow for repeatable movement that will allow future designs to operate the window. This project was chosen because the hardware to demonstrate the concept was already available, and this project is interesting enough to continue in the future, warranting a proof of concept.

2.2 System Architecture


PC
Arduino Uno
Motor Controller
DC Motor
Sends temperature reading
Sends direction and duration
Provides DC voltage (+5 or -5) for specified time

At the highest possible level, this project turns the temperature readings (acquired through many possible means) into +12v or -12v for a precise amount of time. The demonstration program included with the code simply asks the user for a temperature measurement and pushes it through a serial connection to the Arduino. However, this flexibility allows multiple methods to input the temperature. (Leaving final implementation flexible in the future).

3 Hardware Design

3.1 Micro-controller Selection Rationale

The chosen micro-controller was a RedBoard, instead of the initially proposed Raspberry Pi 3B. This choice was made with due consideration, choosing between the convenience of having a networked system with a whole operating system (in the case of the Pi), or a dedicated micro-controller with precise timing. In the future, the final implementation might actually involve both, which will be discussed later. The RedBoard was chosen because it implements a SparkFun branded Arduino Uno, which is the same micro-controller architecture that would likely be part of a finalized hardware design, and the RedBoard has a onboard mounted breadboard for prototyping convenience, which was a useful feature while developing this project. Although having a full OS stack would likely be useful for future expansion, the ease of using the Arduino IDE and the inline AVR assembly made it a better choice for this implementation of the project. Additionally, future expansion with a HomeAssistant-enabled ESP32 will allow for easy input of temperature data, making a full Raspberry Pi overkill for this project.

3.2 Actuator Selection Rationale

A DC motor was chosen for the convenience of being readily available and because it was a better way to simulate a future motor with sufficient torque to overcome the resistance of the window, which requires substantial torque to open or close. It was not the initial selection, but the purchased stepper motor appeared to either have substantially non-standard wiring, or there was some variety of user error. Because of the use of this 12v DC motor, the TB6612FNG motor driver was also needed to avoid damaging the RedBoard with excess voltage, and to allow for precise duty cycle control of the motor. This driver is a fairly common chipset, and was also already on-hand for executing this project.

4 Software Design

4.1 Program Architecture

The overall flow of this program is a loop waiting for serial input from the connected PC. The program expects temperature inputs in the format: `TEMP:XX.X` (where the reading is in Fahrenheit), and uses them to determine wether to open the window, close the window, or take no action at all. The justification for this control loop is that it allows for any frequency of temperature input, without constantly moving the motor backwards and forwards nonstop, since the desired (future) hardware setup will not allow for such a precise temperature control of the inside environment, there is no reason for partial open and closed window, in a manner like airplane flaps.

4.2 Assembly Language Functions

As requested, multiple functions of the project code are implemented in AVR assembly. The functions chosen are:

1. A precise delay function, ensuring millisecond precision in motor control
2. Start motor forward
3. Start motor reverse
4. Stop motor

4.2.1 Precision Timing Function: `precise_delay_ms()`

The `precise_delay_ms()` function creates accurate millisecond delays using cycle-counted assembly loops for controlling motor runtime.

Function Signature:

```
void precise_delay_ms(uint16_t milliseconds);
```

Design Approach: The ATmega328P operates at 16 MHz, requiring 16,000 cycles per millisecond. The implementation uses nested loops: an inner loop of 250 iterations and an outer loop of 16 iterations, totaling 4,000 iterations. With approximately 4 cycles per iteration (including overhead), this yields the required 16,000 cycles per millisecond.

Assembly Implementation:

```
asm(
  ".global precise_delay_ms\n"
  "precise_delay_ms:\n"

  // Save registers
  "  push r16\n"
  "  push r17\n"
  "  push r18\n"
  "  push r19\n"

  // Outer loop: counts milliseconds
  "delay_ms_outer:\n"
  "  cp r24, r1\n"                 // Compare with 0
  "  cpc r25, r1\n"
  "  breq delay_ms_done\n"

  // Initialize nested loop counters
  "  ldi r18, 16\n"                // Outer: 16 iterations
  "  ldi r19, 250\n"               // Inner: 250 iterations

  // Inner loop: creates ~1ms delay
  "delay_ms_inner:\n"
  "  dec r19\n"
  "  brne delay_ms_inner\n"
  "  ldi r19, 250\n"
  "  dec r18\n"
  "  brne delay_ms_inner\n"

  // Decrement millisecond counter
  "  subi r24, 1\n"
  "  sbci r25, 0\n"
  "  rjmp delay_ms_outer\n"

  "delay_ms_done:\n"
  "  pop r19\n"
  "  pop r18\n"
  "  pop r17\n"
  "  pop r16\n"
  "  ret\n"
);
```

Register Usage:

- `r24:r25` – 16-bit millisecond count (input parameter)
- `r18` – Outer loop counter
- `r19` – Inner loop counter
- `r1` – Zero register (AVR convention)

Key Concepts Demonstrated:

- Stack-based register preservation (push/pop)
- 16-bit arithmetic with carry (`cpc`, `sbci`)

- Nested loop structures
- Cycle-accurate timing through instruction counting
- Proper C-to-assembly calling conventions

Accuracy: The implementation provides sufficient accuracy for motor control applications where delays are measured in seconds. Sub-millisecond timing errors are negligible compared to mechanical variations in motor response.

4.2.2 Motor Control Functions

The motor control functions use direct port manipulation to set the TB6612FNG motor driver's direction pins (AIN1 and AIN2), demonstrating low-level hardware control without Arduino abstraction layers.

AIN1 (Pin 6)	AIN2 (Pin 7)	Motor Action
HIGH	LOW	Forward rotation
LOW	HIGH	Reverse rotation
LOW	LOW	Stop (coast)

Table 1: TB6612FNG direction control logic

Port Mapping: Both control pins are on PORTD of the ATmega328P:

- Pin 6 (AIN1) → PORTD bit 6 (bitmask: `0x40`)
- Pin 7 (AIN2) → PORTD bit 7 (bitmask: `0x80`)

Function 1: `set_motor_forward()` Sets motor to rotate forward by setting AIN1 HIGH and AIN2 LOW.

```
asm(
  ".global set_motor_forward\n"
  "set_motor_forward:\n"
  "  push r16\n"                  // Save register
  "  in r16, 0x0B\n"              // Read PORTD (address 0x0B)
  "  ori r16, 0x40\n"             // Set bit 6 (AIN1) HIGH
  "  andi r16, 0x7F\n"            // Clear bit 7 (AIN2) LOW
  "  out 0x0B, r16\n"             // Write to PORTD
  "  pop r16\n"                   // Restore register
  "  ret\n"
);
```

Function 2: `set_motor_reverse()` Sets motor to rotate in reverse by setting AIN1 LOW and AIN2 HIGH.

```
asm(
  ".global set_motor_reverse\n"
  "set_motor_reverse:\n"
  "  push r16\n"
  "  in r16, 0x0B\n"              // Read PORTD
  "  andi r16, 0xBF\n"            // Clear bit 6 (AIN1) LOW
  "  ori r16, 0x80\n"             // Set bit 7 (AIN2) HIGH
  "  out 0x0B, r16\n"             // Write to PORTD
  "  pop r16\n"
  "  ret\n"
);
```

Function 3: `set_motor_stop()` Stops the motor by setting both AIN1 and AIN2 LOW.

```
asm(
  ".global set_motor_stop\n"
  "set_motor_stop:\n"
  "  push r16\n"
  "  in r16, 0x0B\n"               // Read PORTD
  "  andi r16, 0x3F\n"             // Clear bits 6 and 7
  "  out 0x0B, r16\n"              // Write to PORTD
  "  pop r16\n"
  "  ret\n"
);
```

Key Concepts Demonstrated:

- Direct I/O port access using `in` and `out` instructions
- Bit manipulation with `ori` (OR immediate) and `andi` (AND immediate)
- Read-modify-write operations to preserve other port bits
- Register preservation following calling conventions
- Hardware abstraction bypass for performance and control

Design Rationale: These functions demonstrate assembly's advantage in hardware control: direct register manipulation without the overhead of Arduino's `digitalWrite()` function. Each function executes in only 6 clock cycles (excluding call/return overhead), compared to hundreds of cycles for the equivalent C++ Arduino functions.

4.3 C++ Implementation

The C++ portion handles high-level program flow and decision logic, delegating time-critical hardware control to the assembly functions. The `setup()` function initializes the three motor control pins (AIN1, AIN2, PWMA) as outputs and configures serial communication at 9600 baud.

4.3.1 Main Control Flow Functions

`serialEvent()` Automatically captures incoming serial data, building a complete command string until a newline character is received. This non-blocking approach allows the program to continue normal operation while waiting for input.

`parseTemperature()` Validates the input format (expecting `TEMP:XX.X`) and extracts the numeric temperature value, converting it from a string to a floating-point number for comparison operations.

`controlMotor()` Implements the decision logic based on predefined temperature thresholds (65°F and 75°F). If the temperature falls below 65°F, it calls the assembly function `set_motor_reverse()` to close the window. If above 75°F, it calls `set_motor_forward()` to open the window. In the comfortable range between thresholds, no motor action occurs. The function uses Arduino's `analogWrite()` to set PWM speed control, then delegates precise timing to the assembly `precise_delay_ms()` function, and finally stops the motor using `set_motor_stop()`.

4.3.2 Language Division Rationale

This division of labor demonstrates appropriate language selection: C++ handles string parsing, floating-point arithmetic, and conditional logic where readability matters, while assembly handles the performance-critical tasks of precise timing and direct hardware manipulation.

5 Development Process

5.1 Design Iterations

As with any project, the initial approach proposed in the research assignment did not entirely pan out to reality. We will briefly discuss the issues encountered and how they were overcome to achieve a final sample implementation of this project.

5.1.1 Stepper Motor Attempt

The initial design was to use a A9488 stepper motor controller and compatible stepper motor that was previously acquired. However, the stepper motor appeared to have non-standard coil wiring, preventing any manner of Arduino program from driving the stepper at all. Debugging was performed, ensuring that the Arduino and breadboard were providing correct voltages to the stepper controller, and that the controller was outputting correct voltage and phasing for the stepper. Additionally, multiple permutations of the phase wiring were tried, to ensure that the issue wasn't simply that the stepper had non-standard coil pairs. Because of these issues, the DC motor was added instead.

5.1.2 Gearbox Issues

Once the basics of the final program were in place, an issue was encountered with the DC motor used to replace the stepper. The motor was from a SparkFun rover kit, which included a gearbox that likely would have been a hindrance with a final hardware design for the window interfacing gearbox. Unfortunately, the gearbox even prevented a small-scale test, since it provided too much friction to allow the motor to run at a lower duty cycle, which we believe was necessary to provide accurate control for a final hardware design. Thankfully the gearbox assembly was trivial to remove, exposing the raw motor and allowing implementation of the project to resume.

5.1.3 Pin Configuration

While developing the final code, and the ramp control program to find the minimum duty cycle of the sample DC motor, we encountered an issue where new programs were not moving the motor at all, regardless of the duty cycle chosen. Initially, we were concerned that there was some issue with the motor that would simply not allow it to run at lower duty cycles, but after a massive amount of time debugging, it was discovered that a breadboard wire had somehow hopped from pin 7 to pin 8, rendering any software changes irrelevant. Thankfully, after correcting the wiring issue, development was able to continue.

5.2 Key Takeaways

Thankfully there were alternative choices available for both the micro-controller and the actuator chosen for this project. Additionally, the SparkFun kit came with small probes that allowed the multimeter to easily be inserted into the breadboard matrix, which was invaluable in debugging the stepper motor, and later the DC motor. Despite the disappointment of not having a temperature sensor available, we believe the pivot to accepting USB measurements will allow for maximum future flexibility and still achieves the requirements of this final class project.

6 Testing and Results

6.1 Test Cases and Results

Test cases for a range of sample temperatures were tested. For brevity, only one of each "action category" is shown in the table below. This demonstrates correct control of the motor for each possible state, as well as for invalid data, demonstrating readiness for future expansion including automated temperature updates.

Test Case	Input	Expected Output	Actual Output	Result
Cold temperature	60°F	Motor reverse (close)	Motor reverse	Pass
Comfortable temperature	70°F	No action	No action	Pass
Hot temperature	80°F	Motor forward (open)	Motor forward	Pass
Invalid format	”abc”	Error message	Error message	Pass

6.2 Performance Metrics

6.2.1 Timing Accuracy (Theoretical Analysis)

The `precise_delay_ms()` function's accuracy is limited by several factors:

Theoretical Error Sources:

- **Loop overhead:** Fixed instruction cycles for loop management (branch, decrement, reload) that don't scale perfectly with target delay
- **Function call overhead:** The `call` and `ret` instructions add 4-5 cycles per invocation
- **Interrupt latency:** If interrupts are enabled (which they are for serial communication), ISR execution can introduce variable delays
- **Register save/restore overhead:** Four push/pop operations add 8 cycles total

Expected Accuracy: For a 2000ms motor runtime (typical in this application), the fixed overhead of approximately 50-100 cycles represents an error of less than 0.01%. The nested loop approximation (4000 iterations $\times$ 4 cycles $\approx$ 16000 cycles) may introduce systematic error on the order of 1-3%, but this remains well within acceptable bounds for motor control.

Motor Response Characteristics: The motor system introduces far larger timing variations than the assembly delay function:

- **Startup inertia:** Motor takes 10-50ms to reach steady-state speed after voltage is applied
- **Load variations:** Friction and mechanical load can vary motor speed by 5-10%
- **Voltage fluctuations:** Power supply variations directly affect motor speed
- **Temperature effects:** Motor resistance changes with temperature, affecting torque and speed

These mechanical factors dwarf any timing errors from the assembly implementation, making sub-millisecond precision unnecessary for this application.

6.2.2 System Response Time

The complete control loop latency consists of:

1. Serial character reception: 1-10ms depending on baud rate and message length
2. String parsing and temperature extraction: less than 1ms
3. Decision logic evaluation: less than 1ms
4. Motor activation through assembly functions: less than 1ms
5. Total response time: approximately 3-15ms from temperature command to motor activation

This response time is effectively instantaneous for the intended application, where temperature changes occur on the timescale of minutes to hours.

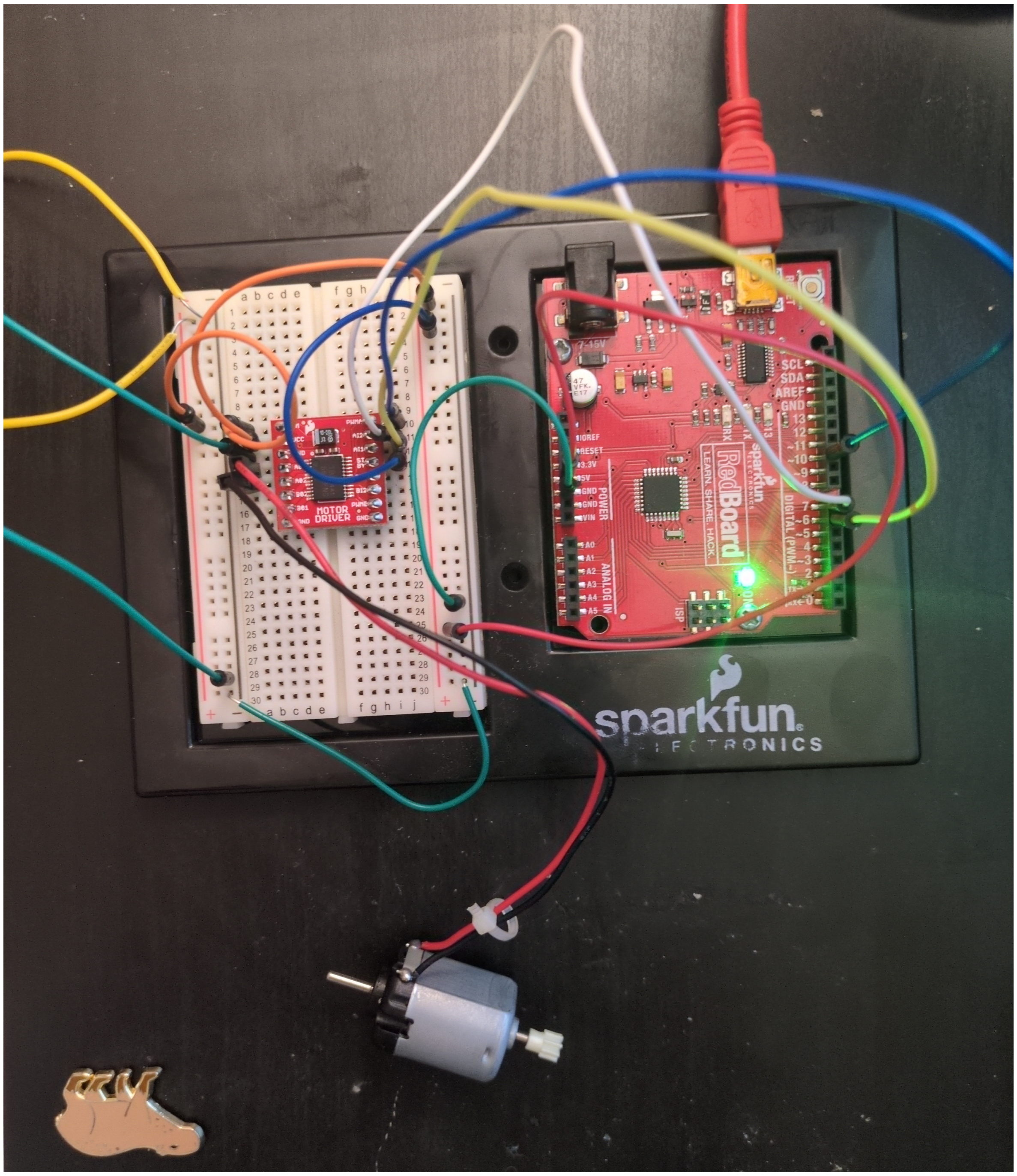
6.2.3 Reliability Considerations

Open-Loop Control Limitations: The system operates without position feedback, relying on timed motor operation to achieve consistent positioning. This approach has inherent limitations:

- Position drift accumulates over multiple operations
- No compensation for varying mechanical loads
- No detection of motor stall or obstruction

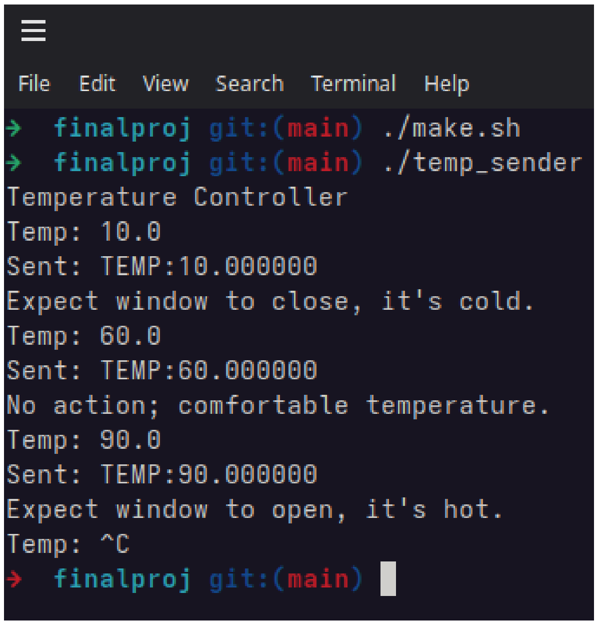
Future Improvements: Closed-loop control with position encoding (limit switches, rotary encoder, or potentiometer feedback) would eliminate timing-dependent positioning and provide true repeatability. However, for a proof-of-concept demonstration, the open-loop timing approach adequately demonstrates the control logic and assembly integration.

6.3 Demonstration



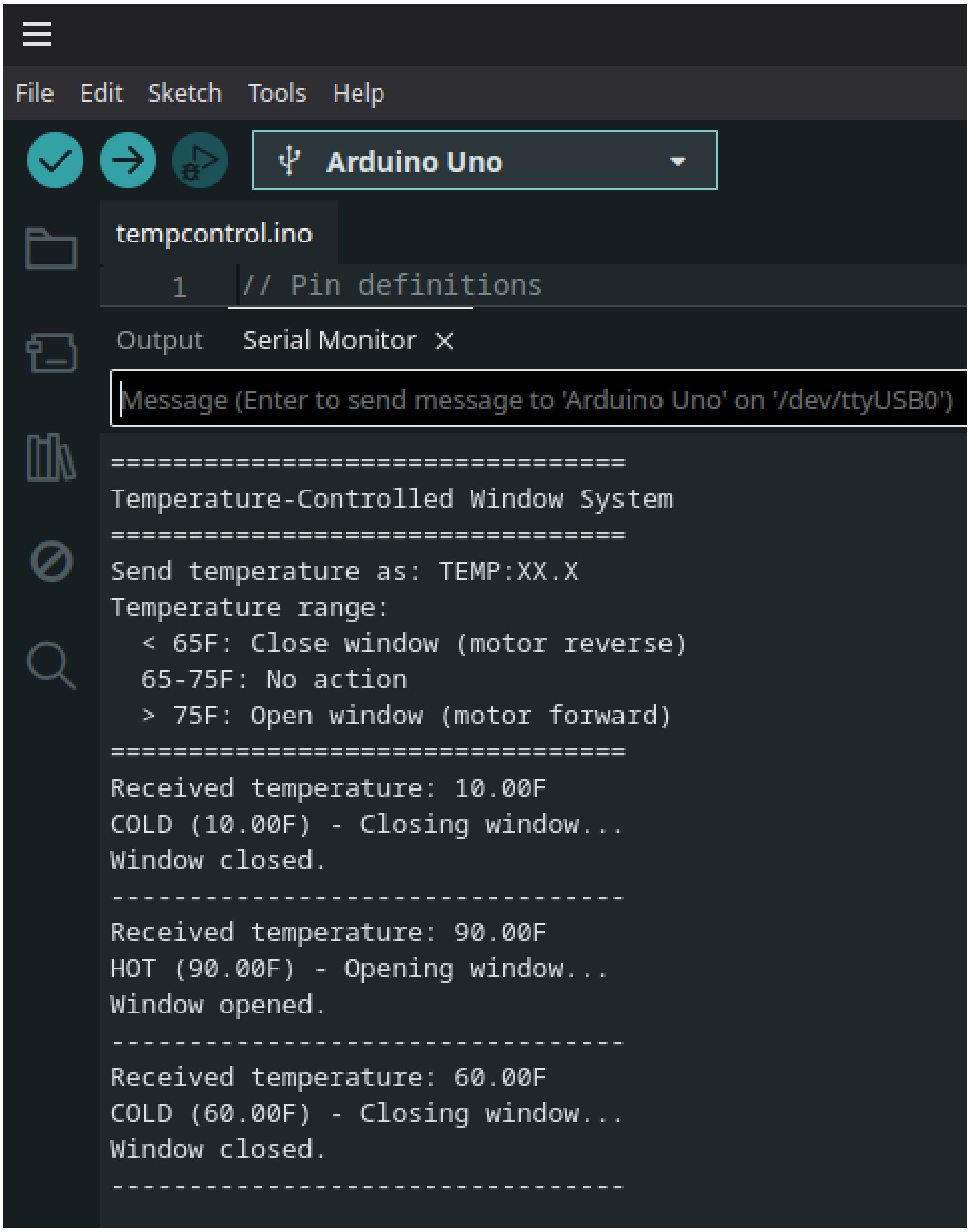
The breadboard implementation can be seen in the image above, as well as the tiny sample 12v motor.

A video demonstration was considered, but given the small form factor of the motor, it was discovered that proving the motor moved in a particular direction was not possible in a manner discernible to video.



```
→  finalproj git:(main) ./make.sh
→  finalproj git:(main) ./temp_sender
Temperature Controller
Temp: 10.0
Sent: TEMP:10.000000
Expect window to close, it's cold.
Temp: 60.0
Sent: TEMP:60.000000
No action; comfortable temperature.
Temp: 90.0
Sent: TEMP:90.000000
Expect window to open, it's hot.
Temp: ^C
→  finalproj git:(main)
```

Seen above is a sample session using `temp_sender.c` to send manual temperature readings over `/dev/ttyUSB0` from the workstation. This program's only advantage over the IDE's serial monitor is user convenience in that they do not need to install the IDE, nor know how to talk directory to a USB device from their terminal. However, it does work as a sample for how one could design a program to extend the functionality of the existing Arduino sketch on the micro-controller.



```
// Pin definitions
```

```
==================================
Temperature-Controlled Window System
==================================
Send temperature as: TEMP:XX.X
Temperature range:
  < 65F: Close window (motor reverse)
  65-75F: No action
  > 75F: Open window (motor forward)
==================================
Received temperature: 10.00F
COLD (10.00F) - Closing window...
Window closed.
----------------------------------
Received temperature: 90.00F
HOT (90.00F) - Opening window...
Window opened.
----------------------------------
Received temperature: 60.00F
COLD (60.00F) - Closing window...
Window closed.
----------------------------------
```

Seen above is a demonstration of using the Arduino IDE to communicate directly with the serial TUI exposed via the board's serial interface, showing the text prompts included directly in the sketch, which strike a balance between usability for programmers to extend, or for a layperson to directly interact with.

7 Conclusion

7.1 Meeting Requirements

Despite the hardware limitations in availability and functionality changing what was possible to implement within the time constraints of finals week, we believe that the project implementation still achieves the objective of demonstrating the feasibility of room temperature control by manipulating a window opened and closed. The project also fulfills the requirement to implement Assembly code that improves the speed and accuracy of the desired portions of the code.

7.2 Personal Takeaways

This project was an excellent demonstration to myself of just how alien hardware work feels to me, as someone who works with high-powered enterprise hardware and software solutions on a daily basis and part of my classwork. While I thankfully had lots of related equipment and spare parts (breadboard wires, soldering equipment, multiple micro-controllers), it was still a stark reminder to myself of just how little time I have actually spent appreciating the low power, reliable computers that make so many other systems around the world function, and just how complex it can be to troubleshoot and develop those systems.

8 References

1. Atmel Corporation. *ATmega328P Datasheet: 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash.* Atmel, 2015.

2. Toshiba Corporation. *TB6612FNG Datasheet: Driver IC for Dual DC Motor.* Toshiba Semiconductor, 2007.

3. SparkFun Electronics. *SparkFun RedBoard Hookup Guide.*
   https://learn.sparkfun.com/tutorials/redboard-hookup-guide

8.1 AVR Assembly and Microcontroller Programming

1. Atmel Corporation. *AVR Instruction Set Manual.* Atmel, 2016.
   http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

2. Microchip Technology. *AVR Assembler User Guide.*
   https://ww1.microchip.com/downloads/en/DeviceDoc/40001917A.pdf

3. Arduino Documentation. *Inline Assembly in Arduino.*
   https://www.arduino.cc/reference/en/language/structure/further-syntax/asm/