Universal Motor Speed Controller

Alexander Mueller and Elior Bilow Makler

ECEN 5613 FALL 2025

12/11/2025

Table of Contents:

Introduction

The intent of this project in the Embedded Systems Design curriculum is to demonstrate our ability to design an embedded system by pursuing a design of our own imagination that includes at least one novel hardware and one novel firmware component that was not explored in any of the labs. To achieve these objectives, we decided to design a motor speed controller with feedback that could control an AC-powered motor (specifically a universal motor) for use as a spindle (or the cutting tool) of a CNC router.

The novel hardware components in this design included:

- Emitter/detector circuit with a comparator to detect spindle revolutions
- Transister-output and TRIAC-output optocouplers for zero-cross detection and mains power control, respectively
- TRIAC for providing gain on the TRIAC-output optocoupler
- Zero-cross detector to remain in phase with mains power
- Rotary encoder for interacting with the user
- I2C OLED display to provide an integrated interface for the user
- On-board Serial-to-UART for ease of programming

The novel firmware components in this design included:

- Feedback controller and controller timing requirements
- AVR architecture
- Timers using input capture modes
- Interfacing with OLED displays over I2C

Hardware and firmware block diagrams can be found in figures 1 and 2.

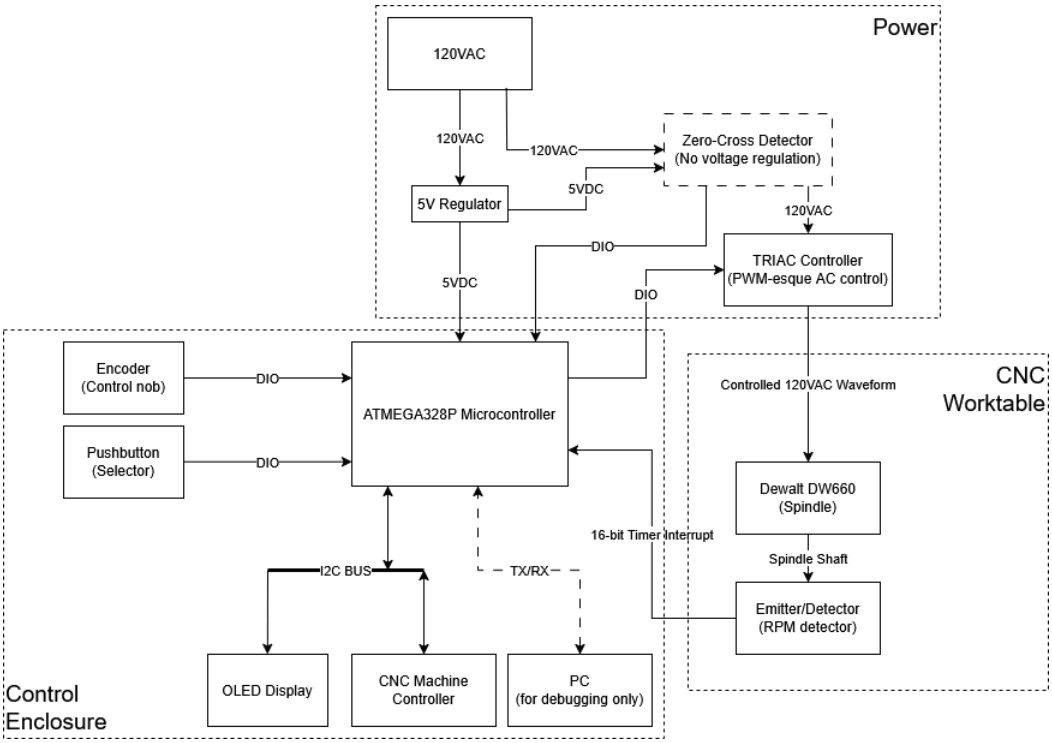

Figure 1: Hardware block diagram

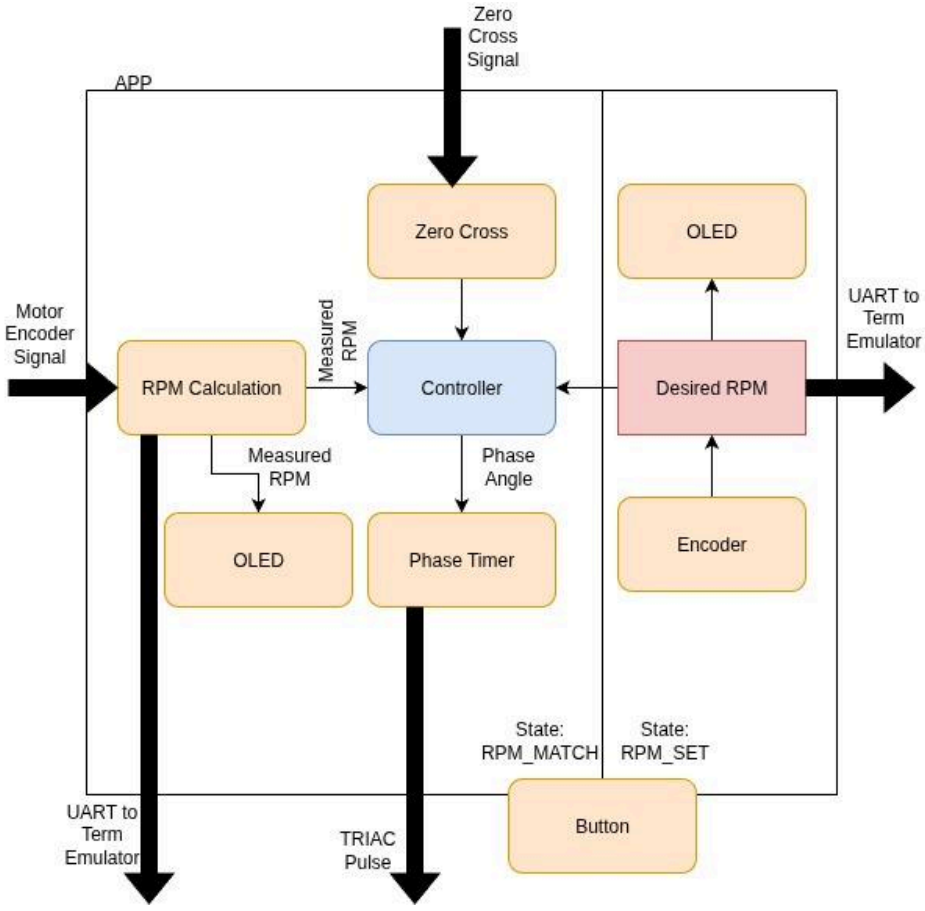

Figure 2: Firmware block diagram

Hardware Design

This section will discuss the hardware (HW) elements of this project with a focus on the challenges and learnings from the design. These elements were primarily developed by Elior Bilow. Any elements not developed by Elior Bilow will be called out as such.

Design Workflow

The progression of the hardware design was designed to be as fast as possible while reducing the risk of errors. This was due to our goal of creating a PCB for this project, which shortened our design timeline and increased our risk of catastrophic failure in hardware design.

To progress the hardware design early, the most important week for the hardware design was the first week after completing the final lab (lab 4). At this point, all initial circuit diagrams were completed and the design was ready to be tested with the hardware components that were ordered weeks before. This foresight was incredibly helpful.

To decrease the risk of a failed PCB, the design evolved in three steps. First, the design was tested on a breadboard. For elements that were not possible to test at high voltages, the circuits were still constructed but were tested under lower voltages (~20VAC as opposed to the 120VAC). Consequently, the motor and feedback controls were not testable in this stage. Furthermore, the hardware that was included in our dev board (Arduino Uno) was not tested on the breadboard or on the perfboards. This was because we had prior experience designing boards with the processor on the dev board and we were able to reuse those designs with confidence since they had already been tested and verified. These designs were augmented from examples by E. Bogatin.

After testing on breadboards, the design was transferred to perfboards. A major error was caught in this phase with the TRIAC and will be discussed later in this section. To accommodate higher voltages, pads were cut out anywhere near the high voltage lines. Also, an optocoupler was broken during a resoldering operation and we felt incredibly lucky to have had the foresight of ordering extras so that we could easily swap in the new component. The extra components also helped to confirm whether out design or a component was bad. At this stage, the firmware started to be integrated with the hardware.

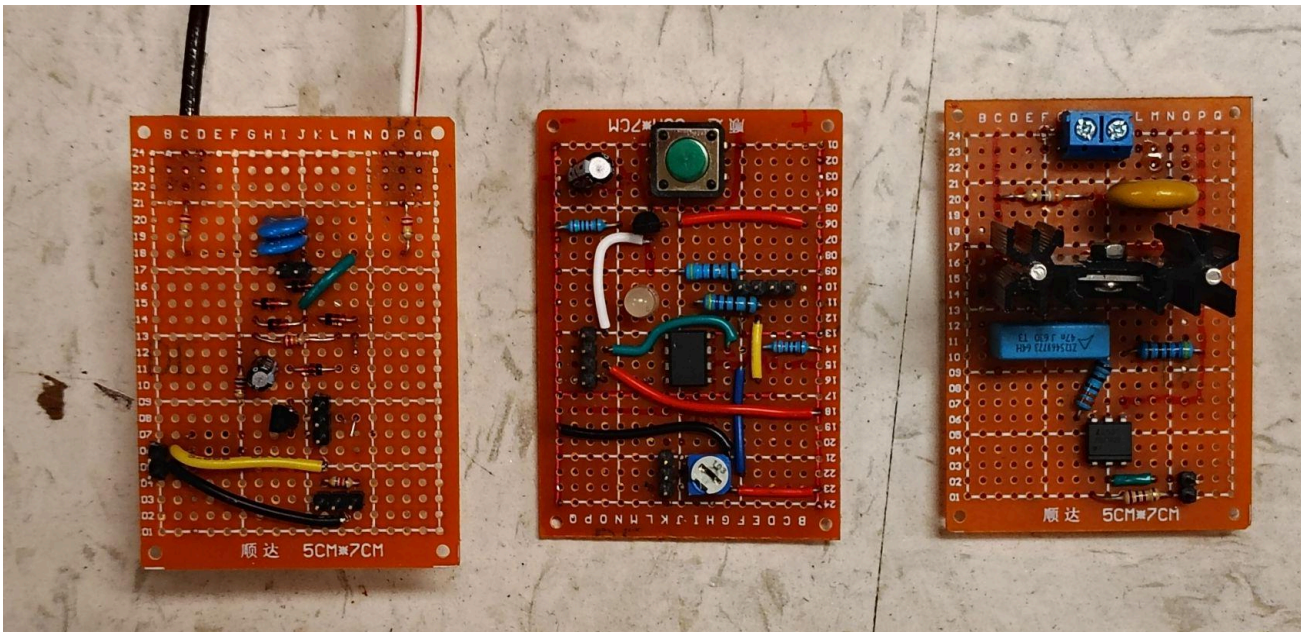

Figure 3: Assembled perfboards

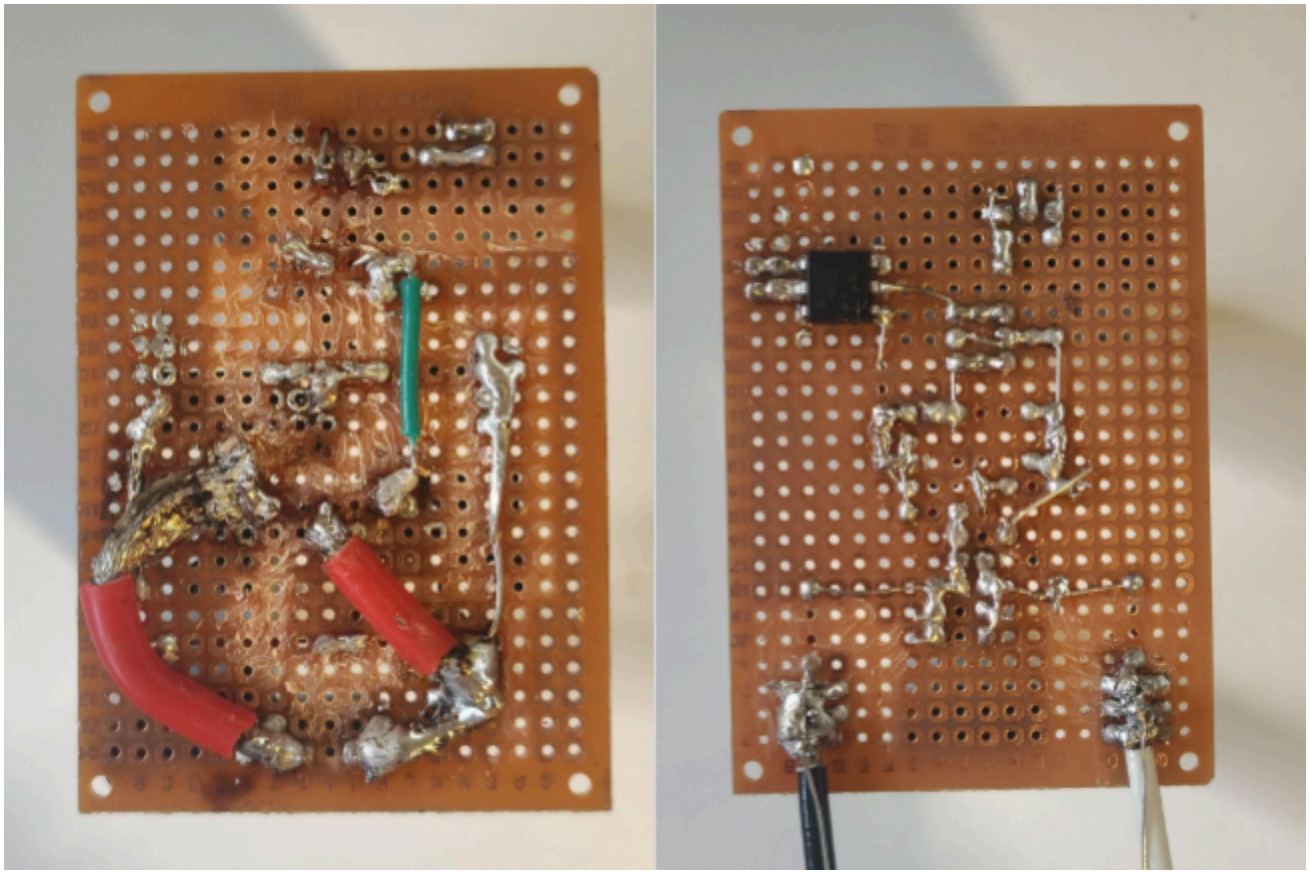

Figure 4: Back side of high-voltage perfboards. The high-voltage regions are isolated by removing pads around them.

Upon completion of the perfboards, a PCB was designed, manufactured, shipped, and assembled. One error on the board caused our ambition to create a new design element (a one-bit memory cell) that was fixed by removing a transistor and adding one jumper. Aside from this, the PCB functioned flawlessly, proving the success in this design workflow. That said, there were areas for improvement. First, some connectors didn't have labels (like the HV terminal block), and some had the connector labels obscured by the connector itself (like the LV terminal block). This made ensuring wires were plugged in properly tricky and could have been easily mitigated with a better silkscreen. Second, the ground clip was positioned very close to other headers, making it difficult to use our oscilloscope alligator clip to connect to that test point without shorting out to other pins. This could have been mitigated by moving the ground clip to a different location on the board and rotating it so that the long side of the clip was parallel with the edge of the board.

Final circuit diagrams can be found in the appendix.

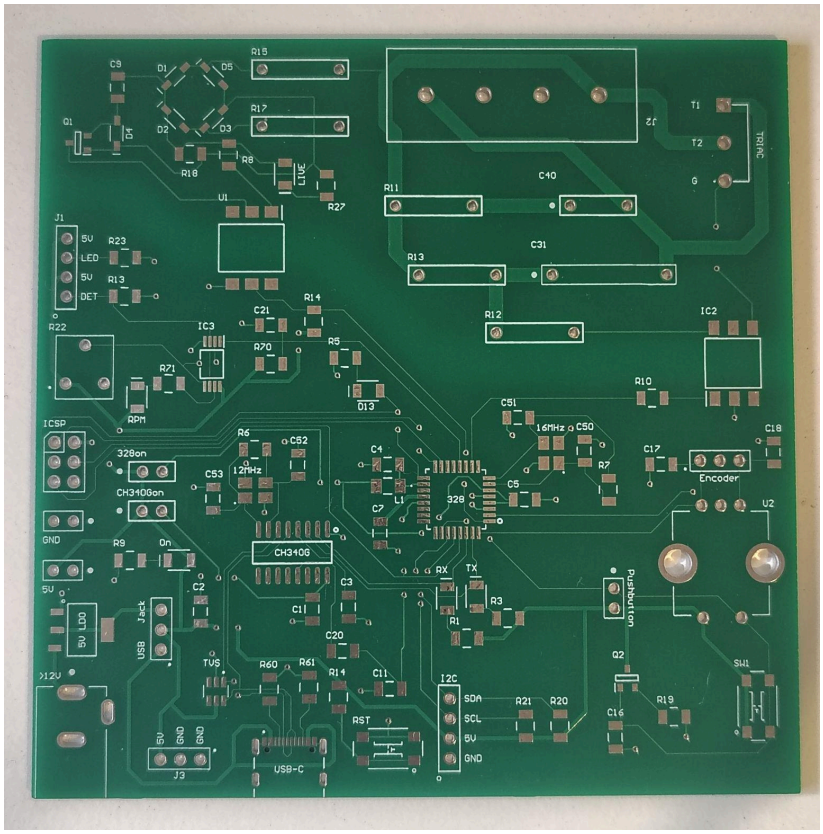

Figure 5: Unassembled PCB

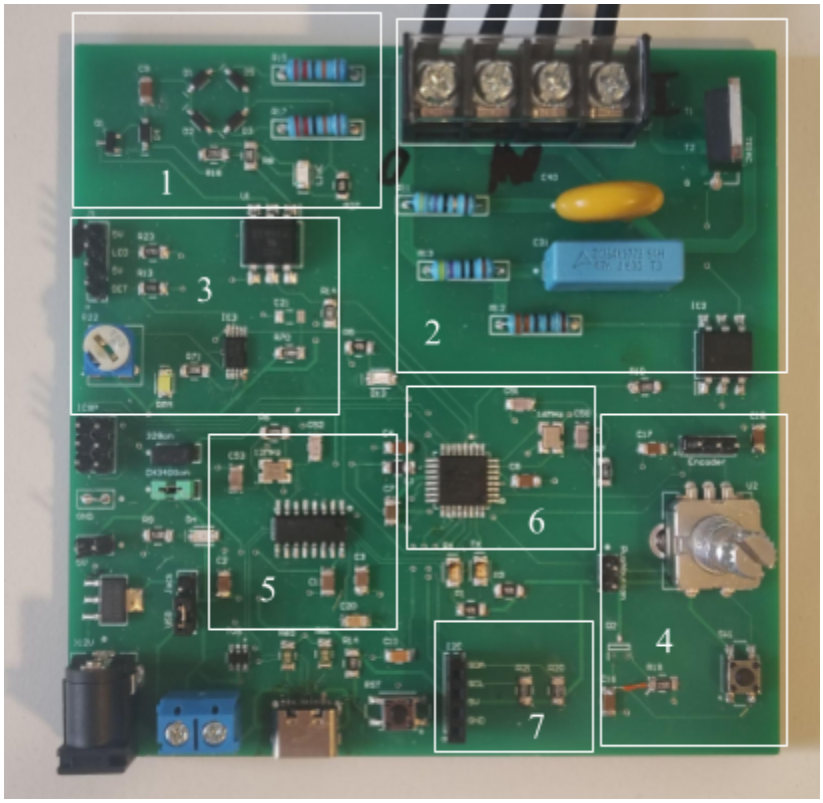

Figure 6: Assembled PCB. 1) Zero-cross detector; 2) TRIAC circuit; 3) Emitter/detector and comparator circuit; 4) Encoder and pushbutton; 5) USB-to-UART circuit; 6) AVR microprocessor; 7) I2C circuit for OLED; Unlabeled) diagnostic LEDs, test points, ICSP port, and power conditioning.

Reused Hardware Elements

There were a handful of elements in our hardware design that were already explored in the labs. These provided little learning experience and little trouble with our design. They included:

- Pushbutton
- Indicator LED
- VRM
- Oscillators

Novel Hardware Elements

The majority of the hardware elements were novel. Each one of them is covered in this section.

AVR Microprocessor

We opted to use the ATMEGA328P microprocessor on our board due to our previous experience working with this chip. While this decreased the risk in hardware, it increased the risk in software since the chip we selected only has one 16-bit timer. See the section on the firmware design of the RPM capture and Phase timer modules for more details on the impact of this decision.

USB-to-UART Converter

Rather than relying on an external USB-to-UART converter, we opted to include one on our board. We chose the CH340G since it was affordable and its implementation is well documented by E. Bogatin.

I2C OLED Display

To provide an integrated system for the user, we attempted to include an OLED that would display the current RPM when the spindle was on and could display the target RPM when the spindle was off. Unfortunately, we destroyed our OLED due to a transient (aka voltage surge) when we connected the low-voltage system to the high-voltage system. After destroying our OLED, we measured the transient and found ground noise of 800mV peak-to-peak. Our other components were able to sustain this or had sufficient decoupling capacitors to handle this transient.

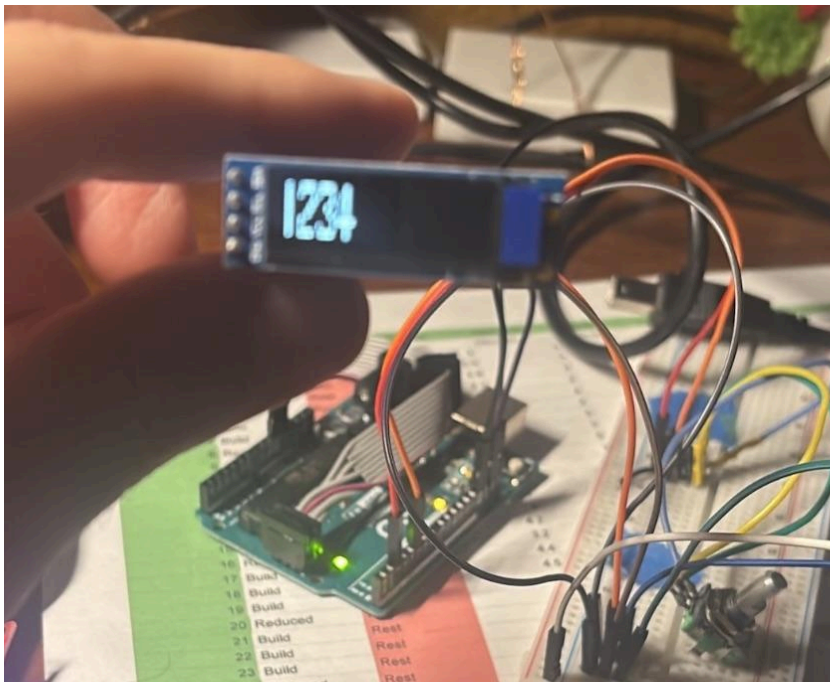

Figure 7: OLED working

Emitter/Detector Circuit with Comparitor

This circuit consisted of an IR emitter and detector with the output of the detector feeding into a comparator. The purpose of this circuit was to detect a revolution of the spindle (which has a max speed of 30,000 RPM, or a revolution frequency of 500Hz, or a revolution period of 2μs). This circuit encountered many hiccups in the breadboarding phase due to the sensitivity of these components and made it apparent to us early on that we should add a comparator.

The emitter (which was a special LED) was outfitted with a resistor that would drive it as bright as possible. The detector was originally assumed to be a photodiode that could provide a voltage bias on its own, but upon further inspection, it was a phototransistor. This phototransistor was then paired with a pull-down resistor on the emitter and the resistor value was tuned such that the voltage delta of the transistor emitter when viewing the shiny part of the spindle collet and the black part of the spindle collet was maximized. This delta was approximately 1.2V, which was significant, but not enough to trigger between the logic levels of the microprocessor, so a comparator with a set trigger voltage (set with a potentiometer) was put on the emitter of the photodiode to make the circuit have a clear high and low voltage (when viewing the shiny and black faces, respectively) that the microprocessor could sense with a GPIO pin.

Figure 8: Emitter/detector facing dark part of collet.

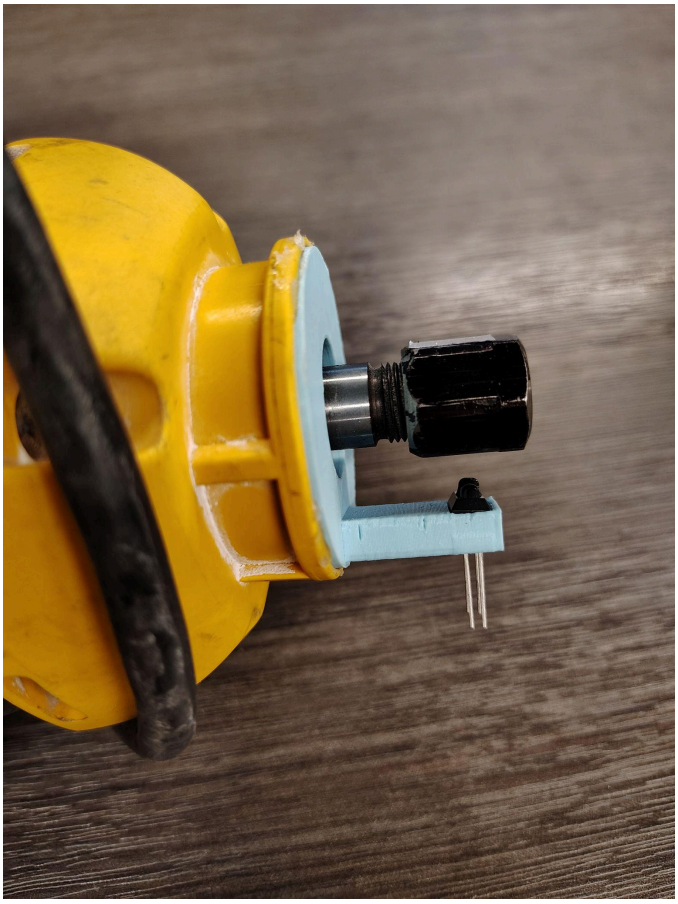

Figure 9: Emitter/detector facing shiny part of collet

Rotary Encoder

A rotary encoder was added to act as a dial that the user could interact with. This circuit was designed by Alex Mueller. We opted to use an encoder to give us flexibility with the user interface (UI). The encoder had debounce problems, though these were mitigated with some capacitors during unit testing of the firmware modules. However in the integration testing we found that the capacitors were not sufficient. The encoder (whose model was changed between the perfboard and the PCB) characteristics had changed enough that the HW debounce was no longer enough to prevent double counting. See the firmware section for detail on the software debounce.

TRIAC

The purpose of this circuit was to control the power going to the spindle. It consisted of a power TRIAC, a TRIAC-output optocoupler, and a few noise filtering RC components. The circuit for this was modified from the datasheet snippet on K. De Craemer's blog.

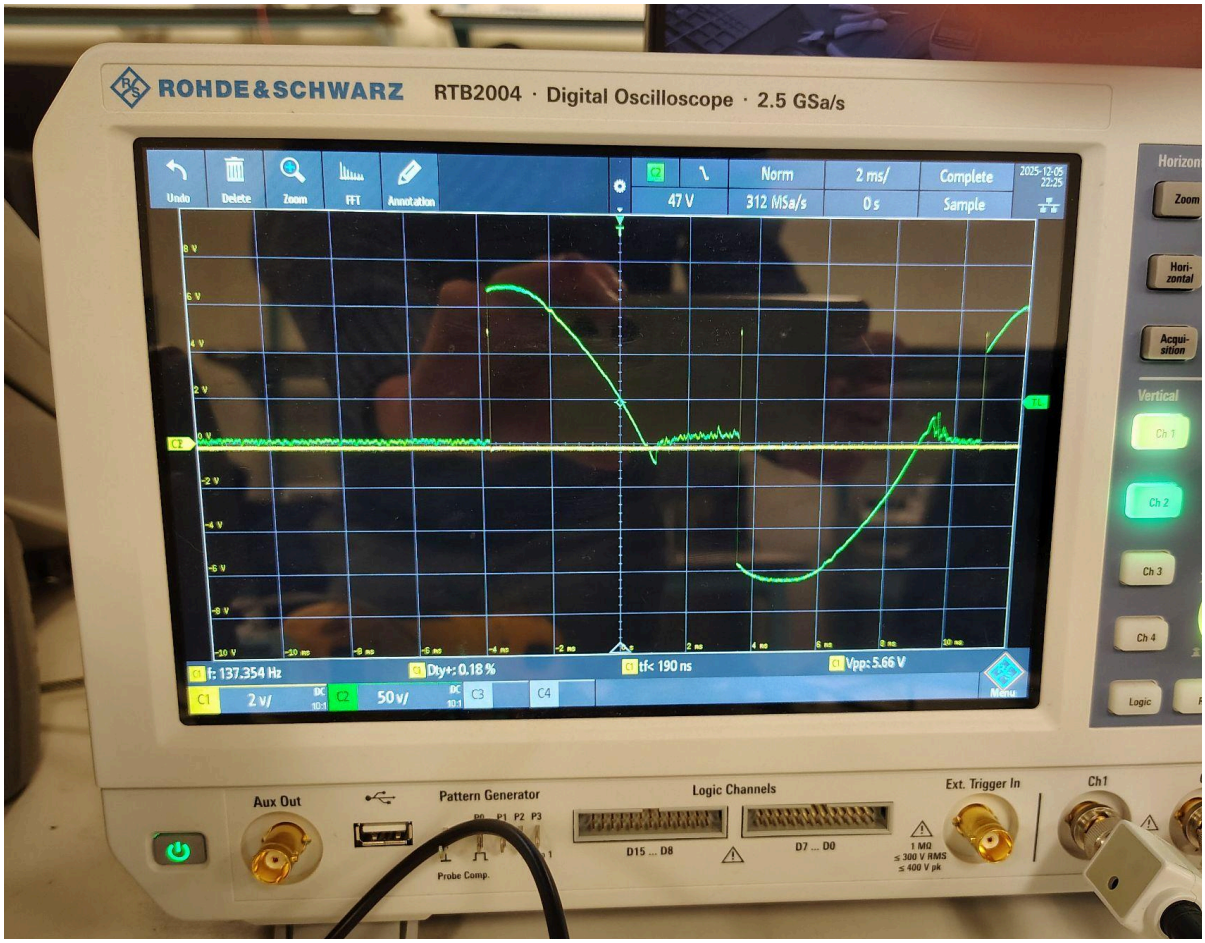

Figure 10: TRIAC output. The output (green) is in phase with the input from mains and the output turns on after a pulse (yellow) is sent to the gate of the TRIAC.

One major error that we encountered was when we switched from a breadboard to a perfboard. In this move, we swapped the terminal leads that were used at optocoupler. Namely, the lead that was connected to optocoupler input was flipped from HOT to NEUTRAL and visa versa. In this condition, the TRIAC would never turn on since the optocoupler would never receive power. Furthermore, the terminal leads on the TRIAC were flipped, causing it to never turn on.

This troubleshooting taught us two important lessons. First, it pays to have extra components to confirm whether the design or the component is broken, as I used our extras in this stage to narrow down what went wrong. Second, it is important to understand a component's capabilities and limitations before designing a circuit that utilizes it. In the future, I will continue to order spares and I will at least read a Wikipedia article and create a test circuit to understand a new component before adding it to a larger design.

Zero-cross detector

This element allowed us to keep our PWM-esque output to the motor in phase with mains. This is important for two reasons. First, the power coming out of a mains circuit is variable since the voltage is not constant. Second, and more importantly, the TRIACs can be turned on, but they cannot be turned off. Instead, they automatically turn off at a zero-cross. This means that a zero-cross signal was used to start a timer that determined when to send a pulse to turn on the TRIAC (see the firmware section for more details).

This circuit consisted of a voltage divider, a full-bridge rectifier, and a voltage-pump-like circuit that would create a pulse on the transistor-output optocoupler whenever the voltage was near zero. The circuit was modified from K. De Craemer's blog.

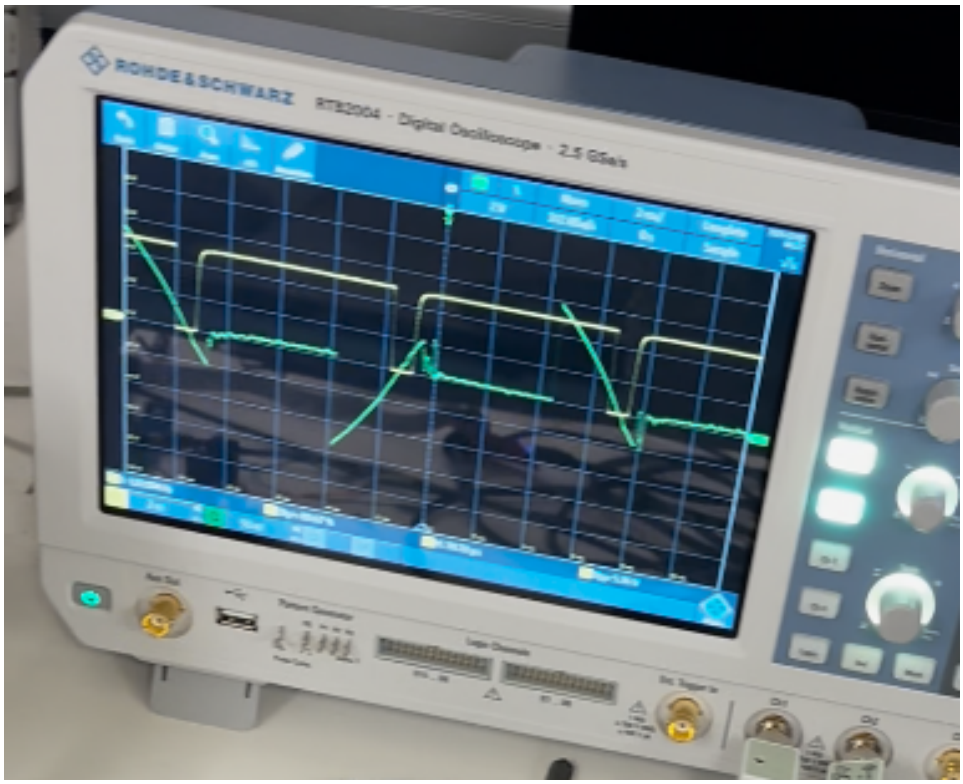

Figure 11: Zero-cross detector pulsing in-phase with mains (yellow). Output is in-phase with mains (green).

Firmware Design

This section will discuss the firmware (FW) elements of this project with a focus on the challenges and learnings from the design. These elements were primarily developed by Alex Mueller.

Design Workflow

Firmware design was completed in its initial stages before even opening a text editor to write code. A general diagram for software flow was made and then iterated upon. There were some changes made from the original design.

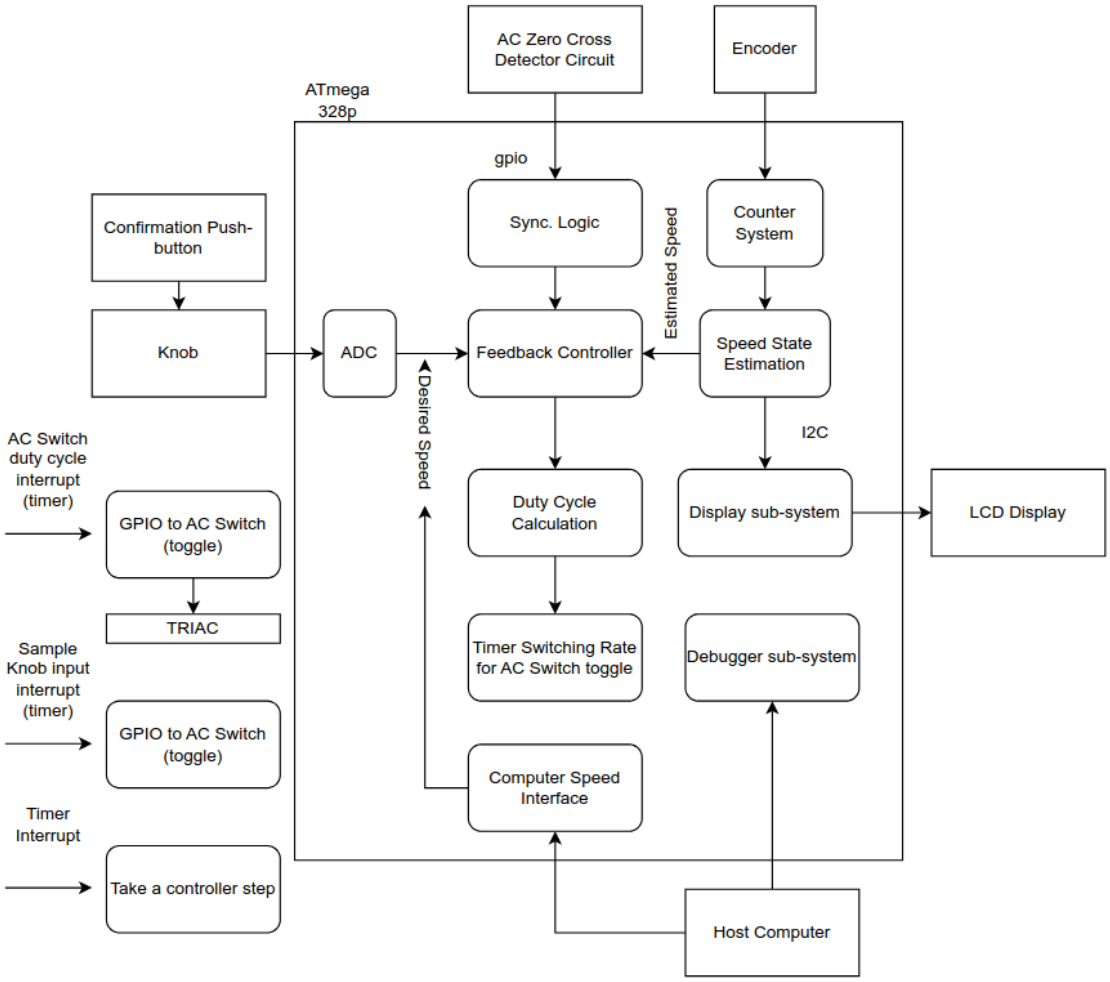

Figure 12: Initial Firmware Design

After designing the general modules with black boxes for more complex logic. A general file structure, build system, and sanity check code was made. After the toolchain was figured out, the sanity-check implementation (blink code) was flashed to the dev board. This helped us to get to our baseline very quickly and start developing firmware modules.

Once it was known that the board could be programmed, the code was iterated upon adding feature by feature and conducting unit testing along the way. This is where prototype boards and small experimental HW setups were useful. A simple breadboard circuit was made for the encoder and OLED. It was fairly easy to unit test the small elements, but the larger features like the RPM measurement module were more involved to test. A signal generator was used to unit test this module by acting as a signal from our spindle encoder.

The firmware was designed in a way to make it portable and easily maintained. It was designed in a modular fashion that helps accomplish this. Pre-processor conditionals are included throughout the code to help change images between debugging builds and production builds. This is fairly limited in scope at this point, but can easily be extended to add more debugging features. It is also set up to switch back and forth between the OLED display of RPM and a terminal emulator view over UART.

Novel FW Elements

New lines of code are around 600. As we know the number of lines of C code do not provide a useful metric in almost anything (quality, complexity, functionality). The majority of it was written by Alex Mueller and it is noted when it was not.

New elements for FW include the timing modes used for external event interval timing, interacting with OLED displays over I2C, the AVR architecture, proportional control on resource constrained microcontrollers, and variable-frequency PWM that was in-phase with an external source.

Modules

App

This module contains the main code for tying the whole application together. It ensures the synchronous and deterministic operation of our hardware. The application module contains the system level initialization code and the super loop that runs the system operation. This module contains the logic to perform the operations necessary depending on the current state of the finite state machine.

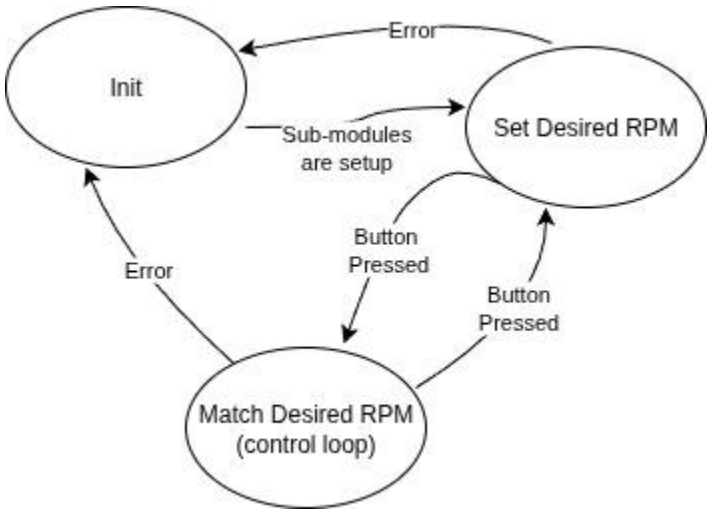

Figure 13: State-Transition Diagram

State Machine

The state machine module contains getters and setters for the private state variable. This helps the design as it grows in complexity to maintain a known state at all points by providing protections for the current state. The logic in this module is not very complex at the current point, but is designed with expansion in mind as the complexity of the application grows. Currently, the state machine is being used to switch between a “spindle on” state where the spindle matches the set RPM, and a “spindle off” state where the user can set a target RPM.

Button

The button module is designed with simplicity in mind. It contains code to help reset the latching circuit on the board through the GPIO pins even though this functionality is not demonstrated in the final HW/FW design. Button state change is simply triggered on a pin change interrupt. This allows us to trigger a main application state change on button release and aids in making this part of the design very expedient.

Encoder

The encoder module initializes one pin to be used as an external interrupt so the encoder can be operated asynchronously. It uses another GPIO pin to determine the direction of the encoder movement depending on which pin is brought low first. This is sampled in the falling edge interrupt of the external interrupt pin. The final design needed a little extra debounce added in software in addition to hardware debounce that was added in the circuit. This was discovered in the integration phase as the same filtering circuit (capacitance wise) was used in unit testing. Simply adding non operation commands helped aid in reducing double counting of incrementation and decrementation.

I2C

This module leverages a two wire interface library that was found on github for the AVR MCUs within this family. It required few modifications and the code gives attributions to the proper author. I2C is used for initialization and displaying digits on the OLED display. The final version of the project does not use this method to display the RPM, but all the code is present and tested.

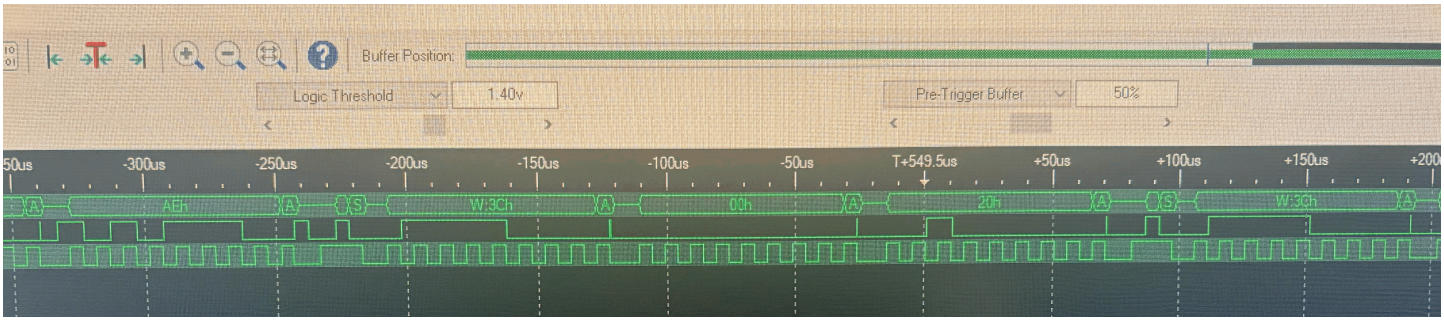

Figure 14: Initialization Sequence for OLED

OLED

The oled module configures the display by using a sequence described in the OLED module's data sheet. It also contains functionality to send individual commands over I2C to the display. The module contains a private byte-map that aids in displaying digits on the OLED with little computation on the processor's side. Unfortunately the OLED module broke too early into integration testing so we could not determine if it met timing requirements. Unit testing on the code with the module was successful and allowed for dynamic modification of displayed content.

UART

The UART module contains very barebones functionality for receiving RPM information on a host PC. This module simply contains code to transmit the RPM digits at 9600 baud. A helpful function was created to transmit the individual digits. This code was developed quickly as a fallback when the OLED display stopped working. Many features to make the interface more user friendly could easily be added with more time.

Zero Cross

The zero cross module provides a way to interact with the signal the zero cross circuit generates whenever the mains sinusoidal wave crosses the zero voltage point. This allows us to synchronize our controller and its timing mechanisms with the wave we are attempting to modulate. We use the microcontroller's edge detection circuitry on the interrupt pins to trigger the timer/counter circuit that counts until we reach the correct phase angle.

One issue that we encountered with the zero-cross detector was that it would go low right before the zero-cross, then go high right after. Originally, our firmware was set up to capture the falling edge, though this could cause the pulse on the TRIAC to happen right after the falling edge was detected, so the TRIAC output would remain off for a whole cycle. This was addressed by sacrificing some motor power and triggering the zero-cross interrupt on the rising edge.

RPM Capture

This module uses the onboard 16 bit timer in input capture mode to capture the interval between falling edges on the input capture pin. This is fed by the motor encoder which creates a pulse once per rotation. Each time the capture occurs, the time since the last interrupt can be saved and used in a simple division operation to obtain the current RPM. This value is used as an input to our feedback controller to calculate the difference between the RPM we wish to achieve and our current RPM.

The RPM capture module requires a 16-bit timer because of the range of RPMs we are trying to capture. For feedback purposes we would like to capture RPMs of less than 6,000 to greater than 30,000 with enough resolution to see slight changes in RPM. This would require the timer to overflow after more than 1ms (to be able to capture 6,000 RPM), and maintain enough resolution of no more than 0.04ms (to differentiate between 30,000 and 30,500 RPM). If we were to use the 8 bit timer, we would have to pre-scale it down extremely far and would then only have 256 values of resolution between 6,000 and 30,000 RPM. This would be more impactful than limiting the phase timer to 8-bits and would make the design in-operable.

Phase Timer

The phaser timer module provides a way to count up to a point and then trigger the TRIAC to switch on the power at any particular point in the AC wave's phase. It uses a timer interrupt that was initially tuned at roughly 8KHz that ticks up a secondary counter. The timer itself was slowed down slightly through testing to make it so the max value for the counter met at a little bit less than 240 Hz. This allows us to modulate the signal on a half phase basis (for the AC mains signal).

Unfortunately this microcontroller only has one 16 bit counter which is necessary in use of the RPM measurement. This limits our phase timer to use of the 8 bit timer. This necessitates us to use a secondary counter to achieve an "overflow" frequency of ~240 Hz. In order to save on cycle time with the timer interrupt overhead, we limit it to 8KHz. This all comes together to allow us to control the power of the signal with 64 discrete steps (possible counter compare values). Unfortunately this is rather limiting when it comes to developing a well-behaving feedback controller. The requirements for the micro-controller were not flushed out enough before picking a controller, which led to this mistake.

Controller

This module contains the logic for the feedback controller. The feedback controller modifies the phase angle at which the AC signal is turned on. This code currently just uses a proportional term, but is provisioned for an integral term once the proportional controller is performing as expected. Its function is to calculate a difference between what RPM we would like our head to be spinning at and the measured RPM. It then attempts to drive this to zero by modifying the amount of power delivered by the motor.

Integration

This section covers the procedure we followed while integrating and the issues that came up during that process. This process occurred predominately during the final week of the project. Integration was done by both Alex Mueller and Elior Bilow collaboratively on-campus.

The general idea with the integration process was to test one part of the system at a time, then to integrate the next part of the system. This was first done by testing each module or subsystem individually by using the perfboards for custom hardware and the dev board for the microcontroller. Based on these results, we could confirm that our interrupts fired when we expected. During this integration testing, we broke the OLED. We did not test integration with two modules: the pushbutton and the RPM detector. For the button, we were confident in our design, and for the RPM detector, the mechanical design of mounting the emitter/detector on the spindle took until the PCB was assembled so it was never tested independently on the perfboards.

When the PCB was assembled and the microprocessor bootloaded, we first tested that the zero-cross detector worked by toggling an on-board LED wherever a cross was detected. Then, we tested the emitter/detector by plugging in the spindle and verifying over UART that we were reading the expected speed. Next, we tested the TRIAC to ensure that we could turn it on through the microcontroller.

With the major subsystems verified on the PCB, we began integration starting with the zero-cross detector and the TRIAC. We hard-coded a 50% duty cycle and verified on an oscilloscope that our output had a 50% duty cycle that was in-phase with mains. Then, we tried integrating feedback through the RPM reader. Originally, our controller had trouble maintaining a constant speed and would instead jump around the hard-coded target RPM, so we added a P-controller. This had some problems with overshooting, so we turned down the values until it didn't overshoot its target.

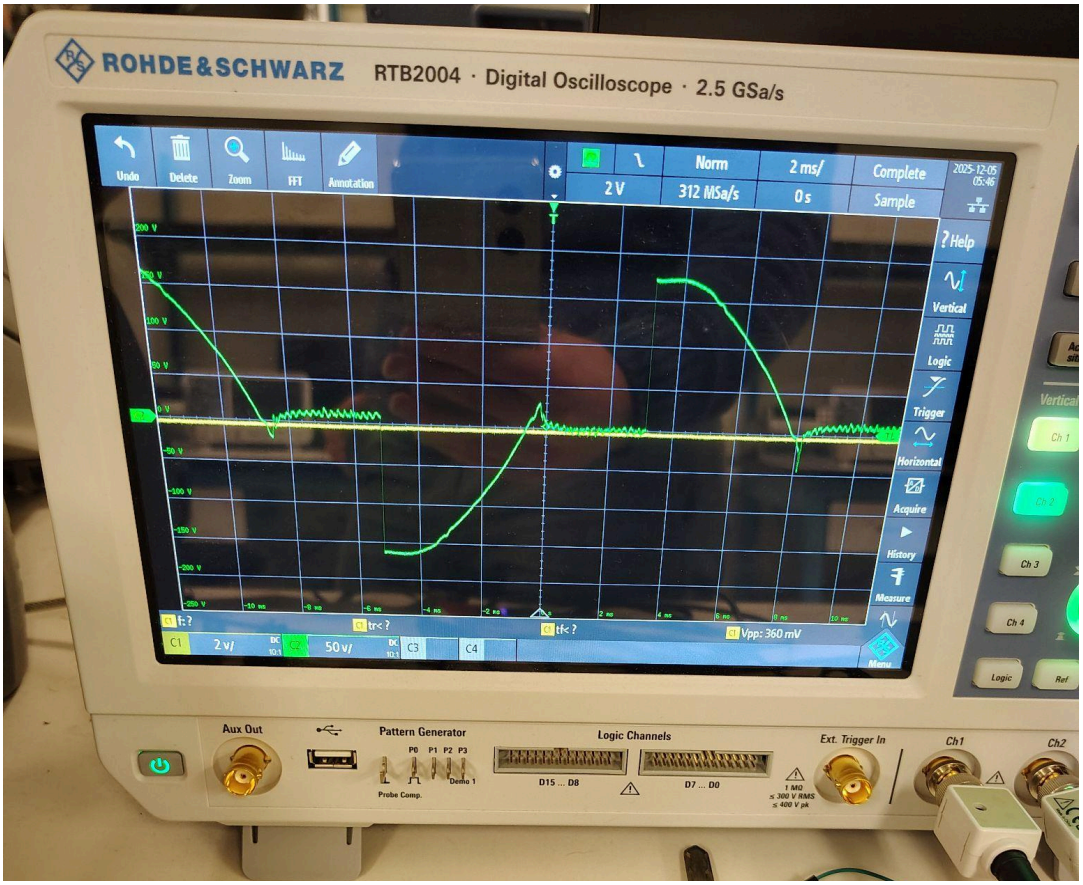

Figure 15: Controlled 50% duty cycle.

Next, we tried to integrate the UI. The encoder was verified independently and confirmed working, though there were some skipping issues that were never resolved. We assume that this could be fixed by tuning the debounce capacitors on the encoder. Notably, a different model of encoder was used on the PCB as on the perfboards. This taught us the value of using the same model of components between different stages of testing.

Surprisingly, the button didn't work as expected. In our design, we tried making a 1-bit memory cell with a capacitor where the pushbutton would pull the cell high, then the microprocessor could read that GPIO port and perform a function if it was high, then reset that pin back to low. However, we were not able to get this functioning so instead, the capacitor was hooked up to a pull-down resistor, turning the capacitor into a debounce capacitor and the GPIO pin into a read-only pin. Then we added some code that would recognise a button press only after seeing both a rising and falling edge. This fixed our button functionality, but it required some edits to the PCB that could have been caught beforehand, demonstrating the value of testing even the simplest circuits on a breadboard or perfboard before ordering a PCB.

After fixing the encoder and pushbutton independently, we tried integrating them into the feedback controller to be able to change the target RPM on the fly. This led to a whole host of issues. First, we had trouble getting the button to switch modes as expected, though we were able to solve this by debugging our firmware logic with the help of the onboard diagnostic LED. Then, we found that including the encoder broke our feedback mechanism. We still aren't certain why this happens, though we performed many tests to attempt to come to a conclusion. We found that integrating a device with a lot of power and momentum brings about an interesting control problem where the motor would be given 100% power, overshoot the target, then would be given 0% power. The motor speed would decrease until it was under the target, get given a large surge of power, greatly overshoot the target, then repeat. This problem is exacerbated by the fact that the spindle slows down much slower than its speeds up. A simple plug-and-play controller was not possible. Many tests were performed to validate the method of controlling the timing and whether this could have been affected by our encoder/button. We tested different gain values for the controller. We have many speculations at this time as to why this happened, but we aren't confident enough in a particular theory to put it forward in this report. More time would yield better answers. We don't believe this specific issue could have been adequately tested pre-integration, demonstrating the value of planning for delays to come up during the integration process.

To cool down our heads while we were frustrated with the integration of the encoder, we added UART functionality to our Makefile so that we could program and have a terminal interface all over a single USB-C cable. The main error we ran into during this integration step was forcing the processor to sync over UART. Our success rate was increased by leaving the processor unplugged for a minute whenever we were unable to sync, and it was also improved by creating a Python script that would send an extra reset signal to the microprocessor right before uploading our code.

Frequently during integration, we found ourselves probing points on the PCB where there were no test points. This made probing more difficult and could have been easily accommodated by adding through-holex to place a jumper connectors at all important points on the board. The tricky part is figuring out which points should be tested, but as a general rule, it would be good to include a test point on every GPIO pin that is being used and any analog inputs. Another thing that would have been nice to have during integration would have been an additional diagnostic LED so that we could have two indicators instead of one.

Conclusion

Lessons Learned

From an academic perspective, this project was a success. While building our microcontroller, we absorbed many lessons summarized in the list below.

- Don't assume anything will work, even simple designs like a pushbutton. Testing all circuits may cost some time on some subsystems, but it will save an order of magnitude more on the system that fails.
- Don't assume that the processor will be "fast enough". Use best practices like keeping ISRs short and ensure they only fire when absolutely necessary.
- Verify microprocessor capabilities against a flushed out requirements list from the firmware perspective before selecting one. For instance, verify the number of 16-bit timers that are required.
- Always label pins and connectors on PCB silkscreens and ensure those labels are not obscured by the connector.
- Learn about new components with test circuits before integrating them in a more complex design.
- Order at least two extra of each component to enable testing whether the circuit you designed or the component is broken, and to have spares on-hand in case one breaks.
- Add test points to every GPIO pin and every analog input.
- Place the ground clip in its own spot far away from anything else on the board and oriented so that the long side is parallel to the edge of the board.
- Maintain consistency between components for each phase of development. Don't assume that a component is a drop-in replacement.
- Include decoupling capacitors for all active components, like OLEDs, microprocessors, and I2C components.
- Find some way to simulate input when you have none like using a waveform generator to test RPM or simulated motor characteristics to test feedback controls.

We also achieved the learning goal of exploring new hardware and firmware elements, as explored in previous sections of this report.

Future Work

From a product development perspective, the project leaves more to be desired. The most significant gap is integrating the encoder to have better on-the-fly RPM control and adding an OLED to allow the device to operate independently. Other smaller elements could be tuned to make the user experience better, such as including mounting holes and a better silkscreen on the PCB.

More work can be done to improve the feedback controller's characteristics and fix issues with its timing. It would be worthwhile to investigate other methods of timing the pulse to turn on the TRIAC because of how limiting the 8 bit timer was. Integration of the integral term would be useful in our application when it finally comes to attaching a cutting head to the motor.

Acknowledgements

[1] E. Bogatin and T. Swettlen, *Workbook for Practical PCB Design and Manufacture*, 6th ed. Addie Rose Press, 2025.

[2] K. De Craemer, “Spindle controller,” KlassDC. [Online]. Available: https://sites.google.com/site/klaasdc/cnc-router/spindle-controller?authuser=0

[3] M. Venn, “Arduino AC motor PID.” [Online]. Available: https://hackaday.io/project/2229/logs

Division of Labor

Unless called out specifically above, Elior Bilow was responsible for designing all hardware elements and Alex Mueller was responsible for designing all hardware elements.

Both Elior Bilow and Alex Mueller contributed equally in the integration and debugging steps.

Both Elior Bilow and Alex Mueller contributed equally to the development of deliverables, though their contributions were focused on the elements that they developed. Alex Mueller was responsible for creating the outline for the report and Elior Bilow was responsible for its formatting. Elior Bilow was responsible for editing the demo video and for recording voiceovers.