EE 453 - Electric Drives
Brushless DC Motor Driver: a Capstone Project

Carissa Flugstad
Kyle Hess
Ryan Mills
Vladislav Sipko

March 9th 2017

Professor:
Rich Christie

Executive Summary

Our intent with this project was to design, build and test a motor driver for a three-phase brushless direct current (BLDC) motor. This driver was intended to be a standalone device with the only input being a 48V supply and the motor hall effect sensors. The driver was designed to control the motor speed, direction and braking operations with a dedicated display to indicate the real-time state of the motor. Our driver operates under four quadrant (4Q) motoring to achieve precise control, faster response time and position control of the rotor. Results from testing our motor drive have come up with an impressive 500 msec top speed to 0 rpm with no load and 500 msec start under full load to top speed. The driver also protects against short circuit and prevents user error as best as possible. The motor drive is encapsulated in our uniquely designed case to separate the user from the circuitry. This case not only protects the user from coming in close contact with high voltages, but also provides a properly labeled, easy to use interface for motor control. The overall cost of the driver is $46.73, which allows it to be competitively priced with similar products in the market today.

Carissa Flugstad
Kyle Hess
Ryan Mills
Vladislav Sipko

Table of Contents

List of Figures

List of Tables

Introduction

Our group has built a solid state motor drive for a brushless DC (BLDC) motor because they are more efficient, last longer, and are more reliable than brushed motors. We chose to implement four quadrant (4Q) motoring in our drive design to achieve precise control, faster response time, and position control of the rotor. The driver has position, speed, and direction control along with a simple user interface and an overall low cost.

Specifications

The specifications for this project which we achieved are:

1) Use the Anaheim Automation BLY 342S-48V-3200 Brushless DC (BLDC) Motor
2) Use a 60V 15A DC power supply operating at 48V
3) Phase current below 10A at all times and below motor rating during steady state
4) Start under full load to top speed in 0.5 seconds
5) Closed loop speed control over wide range of speeds
6) Position control
7) Regenerative braking from top speed to 0 with no load in 0.5 seconds
8) Bidirectional operation of all functionality and reversible while in operation
9) Control interface including:
    a) Master power switch
    b) Power on indicator light
    c) Current display
    d) Speed (rpm) display
    e) Speed control knob
    f) Brake control switch
    g) Reversing switch
    h) Position knob and enable switch
10) Safe circuit, including short circuit protection

Circuit Design

Our motor driver system consists of many sub-circuits that manage various parts of the driver. The overall driver system is shown in Figure 1 below. Although this exact design is not what was demonstrated in the lab, it has been tested and shown to work, but more time was needed to optimize the software to work with this design.

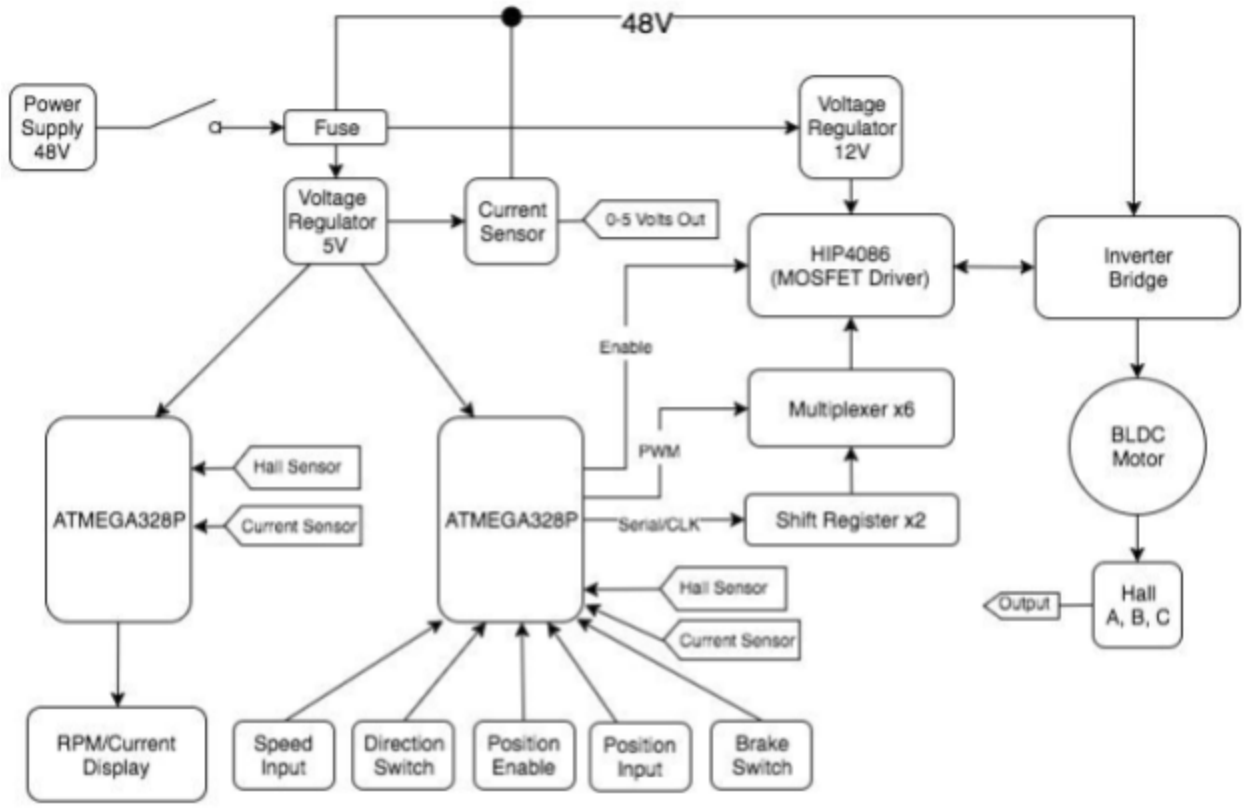

Figure 1: Motor driver system block diagram

Shown here are two individual ATMEGA328P microprocessor chips. Two microprocessors were necessary due to the added computation time needed for the RPM and current display. Thus, one of the processors is used entirely for writing to the display.

Inverter Bridge and MOSFET Driver

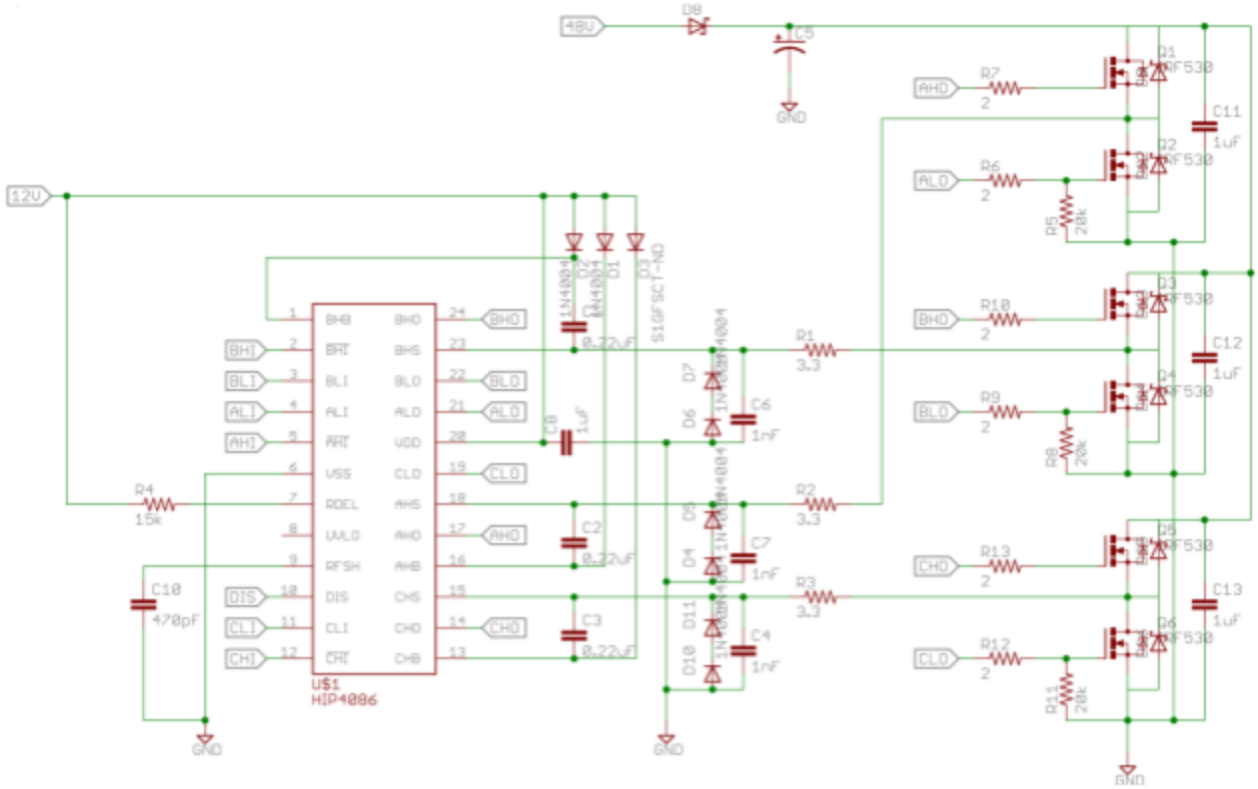

Figure 2: MOSFET driver & 3-phase inverter bridge circuit schematic

The circuit schematic, shown in Figure 2, is the heart of our motor driver. This circuit controls the average terminal voltage applied to each phase of the BLDC motor and thus the direction of current flow through the motor. Control of the current flow is achieved by switching on and off the correct set of N-channel MOSFETs depending on the current position of the rotor.

Reliable switching of both high and low-side MOSFETs is controlled by the HIP4086 three-phase driver chip. This driver uses built-in bootstrap circuitry (along with external diodes and capacitors) to boost the high side output gate voltages to ~12V higher than the respective source voltages. This ensures full turn-on of the high-side MOSFETs of each phase. Referencing the source voltage for each high-side MOSFET required great attention to the voltage spikes seen by the xHS (high-side source) input pins of the HIP driver. These pins are very limited sensitive to over-voltage so a series pair of signal diodes and a snubber capacitor were used to clamp the voltage of these nodes. A small current limiting resistor (3.3Ω) was also installed in series with this pin for further protection.

Printed Circuit Board

Using Autodesk EAGLE we created a schematic, shown in Figure 2, for the inverter bridge and MOSFET driver circuits with the appropriate component package sizes. Using this schematic we generated a compact, single-sided printed circuit board (PCB) layout with minimized trace lengths and wide traces for handling high current delivery (Figure 3). The board design was then compiled into a set of Gerber files that were then used by an LPKF ProtoMat S63 circuit board plotter to route the design into 1oz. FR4 copper board.

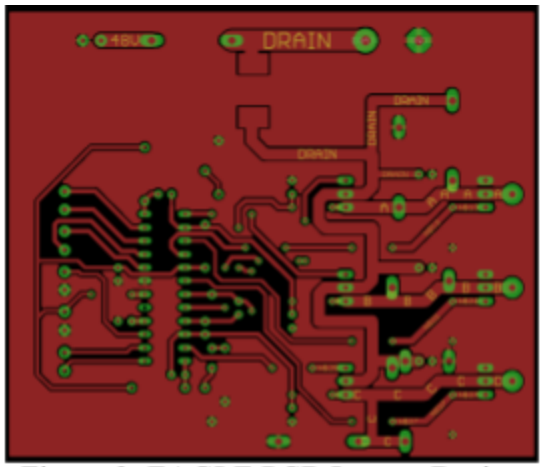

Figure 3: EAGLE PCB Layout Design

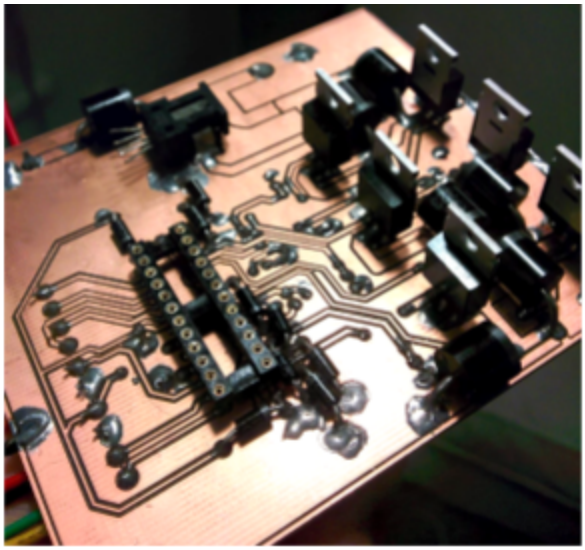

Figure 4: Completed PCB with components installed (Driver chip not inserted)

The inspiration for building a PCB came after reading the HIP4086 driver documentation. Parts of the documentation focused heavily on proper PCB layout and there was a lot of discussion about reducing stray inductance and resistance in the driver chip peripherals. These MOSFET driver chips, like many others, seem to be quite prone to breaking when operated under slightly less-than-ideal conditions. We decided that replacing the breadboard with a circuit board would greatly reduce the voltage spiking seen by the driver chip, perhaps enough to eliminate motor stutter and to stop the destruction of driver chips. After much testing the board seemed to prove its worth as our rate of component destruction plummeted. Also, when replacing components on the PCB, soldering them to the correct locations is much easier and requires very little thought. This is not the case with a protoboard.

4Q Operation PWM Generation and Control

With the desire to run the motor in 4Q operation the ability to PWM all six MOSFETs and PWM four MOSFETs during each step was necessary. Our first choice for generating and controlling the PWM signals was to use an Arduino microcontroller and later a ATMEGA328P. This decision was made due to our familiarity with Arduino and its ease of use. As we progressed with the project we discovered the limitations of the Arduino.

One of the most critical tasks needed to be performed by the microcontroller is to generate PWM signals. At each step of the Brushless DC Motor operation two phases need to be PWM'ed with inverted PWM signals and other phase needs to in the Z state. When the two phases are being PWM'ed the same phase high side and low side mosfets can never be on at the same time. On the HIP4086 driver the low side inputs are inverted inside of the driver. At first glance each phase high side and low side pin can be tied together, but upon further examination the phase in Z state needs both high side and low side MOSFETs turned off. Thus, the high side and low side pins cannot be inverse of each other. Because of this, we need to be able to individually control all six input pins on the HIP4086. In addition, the other phase being PWM'ed needs to have the opposite PWM. This is due to not wanting both high side MOSFETs on or off at the same time. If they are the motor will stop due to voltage across the motor being 0V.

To solve this, we tried several different methods. First was to imitate a PWM signal by changing the value of the digital general purpose input output (GPIO) pins at the desired frequency and at the same time. This method is called bit banging and allowed the flexibility to control which pins were high and low and at what time. This method requires precise timing. Unfortunately, as the program ran and the processor started to execute multiple tasks at once. Causing timing to be delayed and not achieving the desired results.

Next, was to manipulate the PWM generated by the arduino timers directly. One way was to have one PWM signal be the inverse the other by having the duty cycle of the inverted PWM be 1-dt. While the PWM duty ratio was correct, the signals were offset from each other. Our use required that when one PWM has a rising edge at the exact same time the inverted PWM must have a falling edge. With the previous method, this was not the case. To accomplish this, we had to dig into data sheet for the microcontroller of the Arduino. The solution was to adjust the register values that correspond to two pins with the same timer of the Arduino that we were using to PWM.

The solution to the inverted PWM problem uncovered additional problems. On the Arduino Uno and MEGA 2560 there are three timers and six timers respectively. Each timer controls the PWM of the pins it is connected to. Sadly both Arduino types did not have a timer that was connected to 6 pins. When trying to use multiple timers the PWM was offset, as the rising edges were not aligned.

Finally, our end solution was to use multiplexers and inverted PWM from changing the register. Using multiplexers allowed us to select one signal from any of the four being input into the multiplexer. With a multiplexer on each input of the HIP4086 we could send each multiplexer the same inverted and non inverted PWM signal. Along with a 5V high and 0V low for when the phase is in a Z state. This solved the timing issues by only needing to generate two PWM signals

on the same timer chip. The downside to this method that each multiplexer requires two selector pins. Therefore, for the whole system twelve GPIO pins were needed to control the selector pins and two pins needed for the PWM signals.

In order to reduce costs, we decided to use an ATMEGA328P. This is the same microcontroller used in an Arduino Uno, but without all the unnecessary peripherals that the Uno has. Consequently, the ATMEGA328P microcontroller only has 13 digital GPIO pins and as discussed before, the project needs fourteen just to generate the PWM and control the multiplexers. As a result, we ended up using shift registers to take a serial input from the ATMEGA328P and output 8-bits in parallel. This reduced the number of GPIO pins used from 14 down to 6.

The circuit for controlling the inputs into the HIP4086 MOSFET gates can be seen on Figure 5 and Figure 6. As shown the output PWM, inverted PWM, 0V and 5V are sent into the multiplexers. Then the shift registers are used to control the selector pins of the multiplexers. The shift registers are sent a serial input that when the SRCLK has a rising edge the value on the input is sent into a D flip-flop. The old values stored on the flip-flop are shifted into the neighboring flip-flop. The register can store up to eight bits of data. When the desired bits have been input into the register then a RCLK is sent a rising edge, resulting in the values stored in the flip-flops to be moved onto the output pins. Therefore, we can control which signal gets set into the HIP4086 and only use six pin on the ATMEGA328.

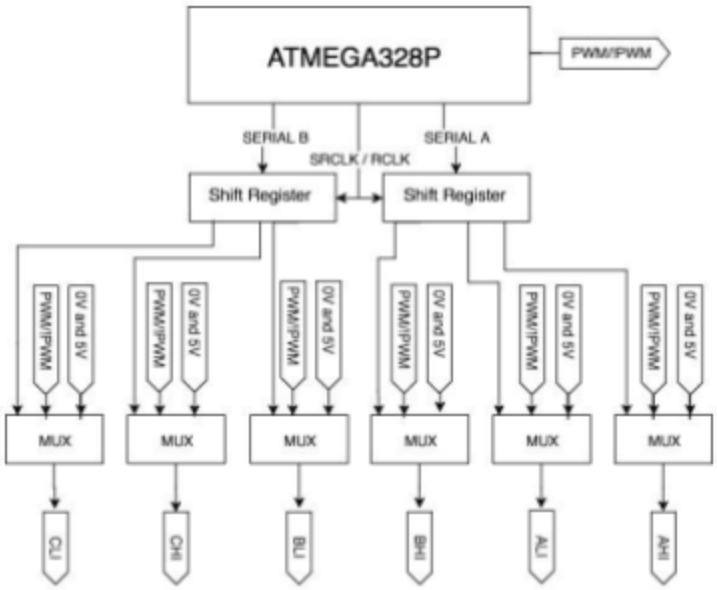

Figure 5: Microcontroller and supporting hardware to control 4Q motor operation.

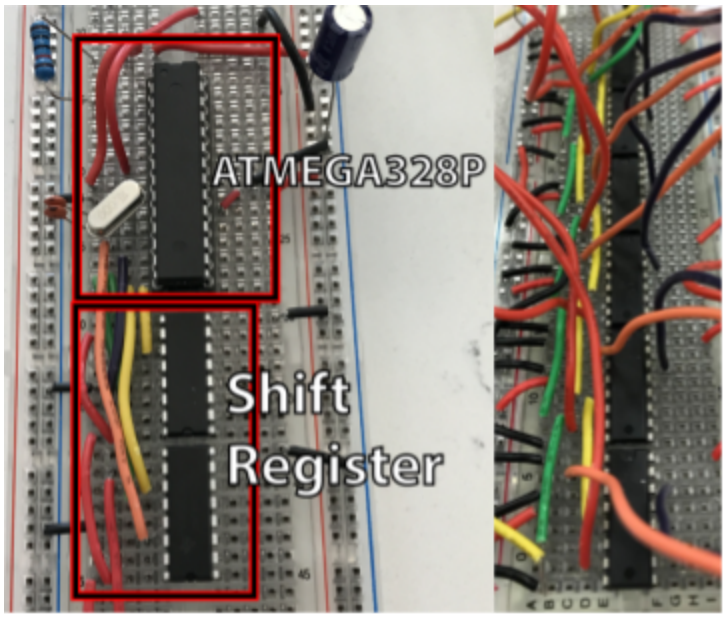

Figure 6: ATMEGA328P micro controller and SN74HC595N Shift Register on left side. 74LS153 multiplexer on right side.

Circuit Safety and Power Distribution

Power from the power supply enters the motor driver system in the module shown in Figure 7 below. The power is first sent through a 12A fuse. This is in place to protect the power supply from short circuits that occur in our system. The fuse is not expected to protected any

components because time for the fuse to blow is slow and the individual component current ratings can be lower than our max 10A being used by the motor windings. Short circuit cases can occur when the MOSFETs in the inverter bridge break and allow current to flow freely.

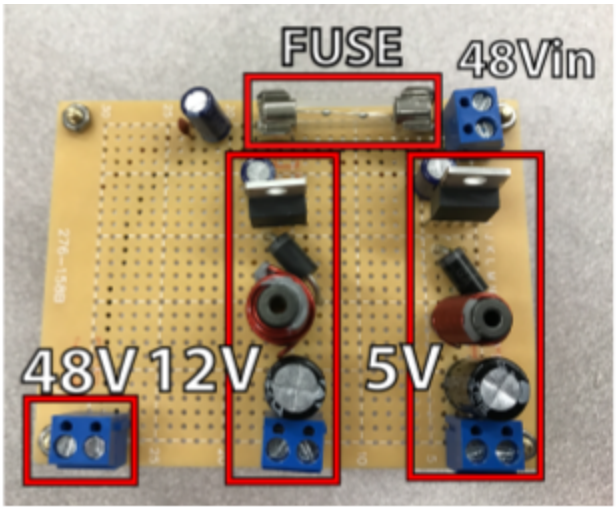

Figure 7: Power distribution circuit with fuse and voltage outputs: 48V, 12V, 5V (From left to right)

Several of components in our system do not work at the input voltage of 48V. This has led to the need for converting the 48V input into several different voltages. As seen in the Figure 7, in addition to 48V there is also 12V and 5V supplied. To accomplish the conversion two buck converters, LM2576HVT-12 and LM2576HVT-5.0, were used. For the converters to work additional capacitors, inductor, and catch diode were needed. The parts selected for the buck converter circuit followed the recommendations of the LM2576HVT datasheet.

Hardware Selection

HIP4086 Three-Phase MOSFET Driver

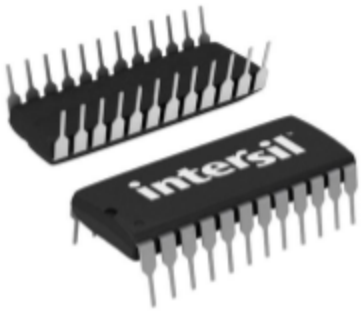

Figure 8: HIP4086APZ MOSFET Driver
Image Credit: www.digikey.com

The HIP4086 driver chip was chosen because of its level of available documentation online, relatively simple I/O structure and its easily adjustable "dead time" delay between switching high and low side MOSFETs of the same phase. The programmable dead time delay was a significant deciding factor when choosing this chip, as it removes some of the possibility of current shoot-through when switching between opposite pairs of MOSFETs. The HIP4086 also operates well within our specified parameters with a supply voltage of 12V, a maximum bootstrap supply of 95V and rise and fall times of 60ns and 40ns, respectively. These times correspond to gate capacitances of 1000pF, thus being acceptable for our MOSFETs given our switching frequency of 31kHz.

LM2576HVT-12 and LM2576HVT-5 Buck Regulator

Figure 9: LM2576HVT Voltage Regulator
Image Credit: www.digikey.com

In order to provide power to the ATMEGA328P, hall effect sensor, current sensor, LCD and HIP4086 driver, we used the LM2576HVT 5V and 12V buck regulators. These regulators were chosen due to their high input voltage of up to 65V and maximum supply current of 5.8A. These regulators also demonstrated very small ripple voltage given the load they are supplying, as well as being very efficient when compared to linear type regulators for the same application. These regulators did require some passive peripheral components in order to operate which increased the total cost, but was much more simple to implement than building them ourselves.

15SQ100 Freewheeling Diode

Figure 10: 15SQ100 Schottky Diode
Image Credit: www.digikey.com

To manage the large back-EMF generated by the motor when switching between phases we used fast Schottky

diodes as freewheeling diodes across each MOSFET. These diodes in particular were rated appropriately for our driver with a peak reverse voltage of 100V and a continuous forward current rating of 15A. Most importantly these diodes have a maximum recovery time of 500ns which allows very little time for negative current spikes to persist.

Figure 11: IRF530N N-channel MOSFET Image Credit: www.digikey.com

IRF530N N-Channel MOSFET

Considering the large load that our motor drive supplies, we were very particular on our choice of switches. That is why we went with the IRF530N. These FETs have a voltage rating of 100V, a continuous current rating of 17A, a relatively low gate capacitance of 920pF (for faster switching times) and perhaps most importantly, an on resistance of 90mΩ. Given these specifications, they were not excessively expensive and were actually cheaper than many FETs with similar ratings. The low on resistance was a very desirable trait of this device because it meant very low losses even at higher switching speeds (such as 31kHz). The low gate capacitance was also appealing to us because it allowed the HIP4086 driver chip to perform to the best of its abilities.

ATMEGA328P

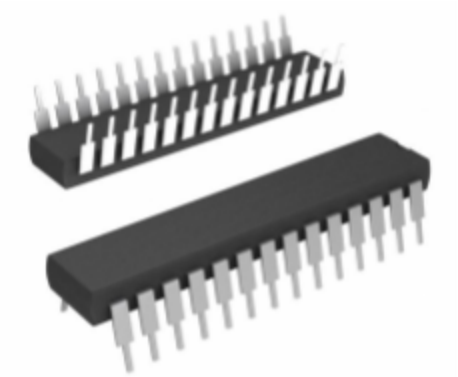

Figure 12: ATMEGA328P microcontroller Image Credit: www.digikey.com

For the motor to run and be controlled, multiple tasks need to be performed. These include generating PWM signals, adjusting duty ratio, switching multiplexer selector inputs, measuring current, writing to a LCD, and switching to position control. In order to accomplish these tasks we used an ATMEGA328P 8-bit microcontroller. The ATMEGA328P was chosen due to its lower cost than using an Arduino Uno or MEGA 2560. An Arduino MEGA 2560 costs $45.95 because of all the added hardware and money going to support the Arduino platform. These costs were not necessary for mass production because there is no need for the controller to be reprogram and there already is a 5V voltage regulator in our design. By using an ATMEGA328P we reduced the cost down to $1.15. Also, the ATMEGA328P was easier to integrate into our circuit because it can be placed directly onto a breadboard or PCB. Additionally, we could use the same code and IDE that an Arduino uses.

74LS153 Dual 4:1 Multiplexer

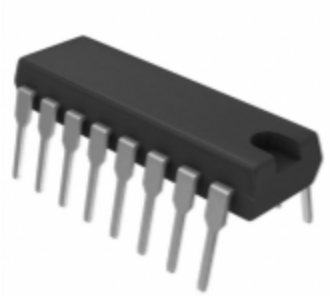

Figure 13: 74LS153 Multiplexer
Image Credit: www.digikey.com

In order to send the proper set of six simultaneous signals to the HIP4086 driver chip we needed a means of controlling which inputs received what. Although the Arduino was capable of sending six individual PWM signals at the same frequency, not all of these signals were referencing the same timer signal. This resulted in the PWM signals being out-of-phase from each other and ultimately unusable. To solve this problem we used six 4-to-1 multiplexers each with the same inputs of PWM, $\overline{PWM}$, 5V, and 0V. Each of these multiplexers also received two binary inputs from the Arduino to select which input signal the respective multiplexer would output. This allowed for much more control over which signals the driver chip received and ensured that the signals were consistent.

SN74HC595N 8 Bit Shift Register

Figure 14: SN74HC595N
Shift Register
Image Credit: www.digikey.com

A consequence of using six 74LS153 multiplexers is that each multiplexer uses two selector pins. Thus, twelve GPIO pins on our microcontroller are needed to control the multiplexers. Unfortunately, the ATMEGA328P microcontroller only has 13 GPIO pins and other parts of the project also needed GPIO pins. To reduce the number of pins needed to control the multiplexers two SN74HC595N shift registers were used. A shift register takes in a serial input on every rising edge of the SRCLK from the ATMEGA328P and outputs all eight bits in parallel on the rising edge of RCLK. The parallel output is used to control the multiplexers. This reduced the total amount of GPIO pins down from twelve to four.

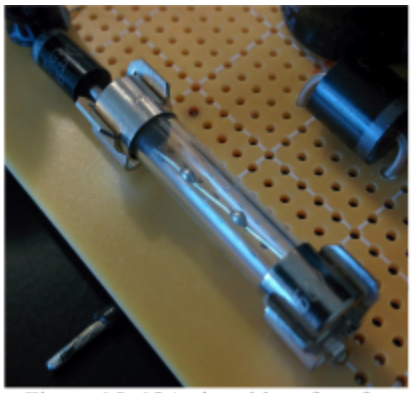

Figure 15: 12A slow-blow fuse for short-circuit protection

Short Circuit Protection Fuse

In the case of a short circuit, a 12A slow blow fuse followed the 48V power supply. This prevents damage to the power supply if power and ground accidentally get connected. The 12A rating on the fuse will work because there will be a very large amount of current running through it in the case of a short circuit. The fuse, however, will not protect the circuitry in case of currents over 10A, since even a fast acting fuse would not blow quick enough to protect from transients.

Circuit Containment Vessel

We produced a box to help create a safer, easier to use product. We laser-cut the box out of ⅛" birch wood. The front of the box has all the controls and displays that the user of this product will need. This makes our driver easy to figure out as well as looking professional and clean. The sides and back of the box have large ventilation slots cut out to allow for ample airflow and cooling of the MOSFETs.

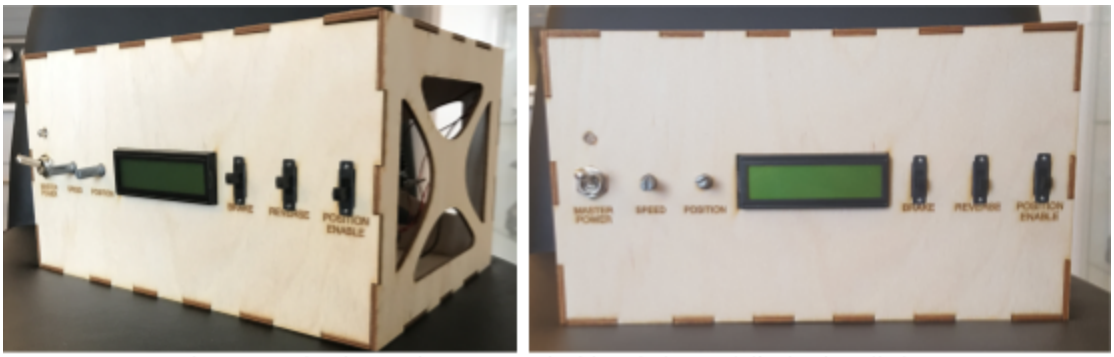

Figure 16: Circuit containment vessel with switches and display installed

Figure 17: Three-phase common mode filter winding diagram
Image Credit: www.bytemark.com

Common Mode Filter

In series with each phase output going into the motor is a winding around a common inductor core. We used a powder inductor core wound with three sets of 20 turns of 16 gauge magnet wire. This provides each phase with an inductance of about 77µH (From Professor Christie). The purpose of the common mode choke is to reduce the amount of noise seen by the output of each driver phase, generated by the motor while imposing very little impedance on the normal current flow.

LCM-S01602DSR/A LCD Display

For our speed and current display, we used the LCM-S01602DSR/A LCD display. This display was chosen because of the level of available documentation online, as well as its ability to easily work with one of the built-in Arduino libraries (LiquidCrystal.h). This display also has a bright backlight and adjustable contrast for enhanced readability under various lighting conditions.

Figure 18: LCM-S01602DSR LCD Display
Image Credit: www.digikey.com

CQ-209A Current Sensor

We placed a CQ-209A current sensor so that it samples the current going directly into the H-bridge from the 48V rail. This location was chosen so we would ensure that the current reading displays the current flowing into the phases, and not the current used for the voltage regulators, microcontrollers, and the rest of the circuit. The current sensor samples the current flowing through the pins using a Hall sensor, and outputs a voltage which is proportional to current value. Figure 19 shows the relationship.

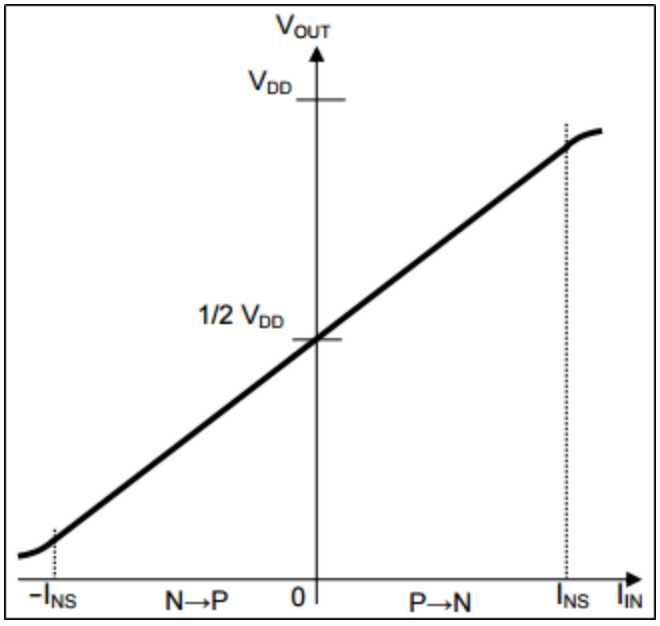

Figure 19: Output voltage vs current input (from CQ209A datasheet)

The sensitivity of the sensor, 66mV/A, is found in the datasheet and gives the slope of the above graph. Equation 1 shows the input current with respect to output voltage:

$$I_{in} = 15.15 * (V_{out} - 2.5) \quad (1)$$

The processor reads a voltage from 0V to 5V as an integer value from 0 to 1023, so we converted the analog input to the voltage measurement using equation 2:

$$V_{out} = V_{analog} * (\frac{5}{1023}) \quad (2)$$

Combining equations 1 and 2, we got the equation to calculate and display the current going into the phases:

$$I_{in} = 15.15 * \left((V_{analog} * (\frac{5}{1023})) - 2.5\right) \quad (3)$$

However, when this was implemented with the circuit, the current readings on the display were lower than the ammeter readings at higher values. To find the relationship between the actual current and what our equation was producing, we plotted several points of each. The closest relationship we found was:

$$Current = 26.06 - (10.64 * V_{analog}) \quad (4)$$

This is the final equation we used to output the correct current onto the display based on a best fit line from measured data points over a range of current levels.

4Q Operation

Commutation

We used the following commutation table to program which MOSFETS need to receive the PWM signal, which MOSFETS need to receive the inverted PWM signal, and which MOSFETS need to be off in each of the six steps.

Table 1: Commutation Table

	Step					
	1	2	3	4	5	6
Phase A	+	Z	-	-	Z	+
Phase B	Z	+	+	Z	-	-
Phase C	-	-	Z	+	+	Z
Hall A	1	1	0	0	0	1
Hall B	0	1	1	1	0	0
Hall C	0	0	0	1	1	1
MOSFET PWM (dT)	A1	B1	A2	A2	B2	A1
MOSFET PWM (dT)	C2	C2	B1	C1	C1	B2
MOSFET PWM ((1-d)T)	A2	B2	A1	A1	B1	A2
MOSFET PWM ((1-d)T)	C1	C1	B2	C2	C2	B1

Speed Control Knob

The speed control knob lets the user adjust the speed of the rotor from 0 rpm to 3200 rpm. This is achieved by mapping the output of a potentiometer from 0-5V to a speed range of 0-3200rpm. The analog to digital converter (ADC) of the Arduino has a resolution of 10bits, so our speed setting has a theoretical resolution of ~3rpm. This is good enough for our purposes.

Closed Loop Speed Control

To eliminate the possibility of incorrectly coding a control algorithm, we opted for downloading a PID controller library built for Arduino. This robust library was developed by Brett Beauregard. In practice the reference rpm from the speed control knob is compared to the actual rpm calculated from the Hall effect sensors. This error is computed by the PI controller and the algorithm automatically adjusts the duty ratio accordingly (if the actual rpm is lower than the reference, the duty ratio is set higher and vice versa). The Kp and Ki constants were adjusted to achieve low rise time and overshoot and fast settling time.

Regenerative Braking

To achieve regenerative braking, a diode is placed in series with the power supply input to prevent current from flowing into the non-regenerative power supply. A large parallel capacitor is used to store the energy released from braking. A switch is used to change the duty ratio of the MOSFETs from the running duty ratio to 50% over a short period of time. This allows for smooth braking without causing too much voltage rise on the regenerative capacitance.

Bidirectional Operation

At 50% duty ratio, the average terminal voltage is 0 volts and the rotor is at a standstill. At a duty ratio higher than 50% the average terminal voltage is positive and so the rotor rotates in the positive direction. At a duty ratio lower than 50% the average terminal voltage is negative and so the rotor rotates in the negative direction. The reversing switch changes the direction of rotor rotation by sampling the current duty ratio and mirroring it across the 50% point. (If the current duty ratio is 100%, the reversing switch makes it 0%, if the current duty ratio is 25%, the reversing switch makes it 75%, and so on). The change in duty ratio is applied over a period of time to allow for a more smooth transition from one direction to the other without a massive spike in current.

Speed Display

The speed is calculated by counting the number of step changes the rotor makes in a half second, which is sensed by the Hall Effect Sensor. Each revolution has 24 steps, so to convert the speed to RPM, we take the number of step changes per half second, divide by 24 to get revolutions per second, and multiply by 120 to get revolutions per minute.

$$RPM = (StepCount / 24) * 120 \quad (5)$$

Position Control

For added safety, we added a switch to enable and disable position control. This allows the user to choose between operating the motor under high speed with speed control, or under zero speed with position control. When the position control switch is enabled, a position knob references to a specific location on the rotor. One revolution on the rotor has 24 steps that can be sensed by the Hall effect sensors, so the position knob can reference between 24 locations on the rotor, from 0 to 345 degrees, in 15 degree increments.

Standards

IEC 60287) - Calculation of permissible current in cables at steady state rating

This standard describes the maximum current that a conductor of given dimensions can safely carry. We paid attention to this standard when designing the high current carrying traces of the inverter bridge PCB, as well as when we chose the magnet wire gauge for the common mode filter.

1-1986) - IEEE Standard General Principles for Temperature Limits in the Rating of Electric Equipment and for the Evaluation of Electrical Insulation

Our design followed these standards by ensuring that the parts of our circuit with the highest power dissipation received ample airflow, and the conductors used for high current applications were sized appropriately to minimize resistive losses and used insulation with a temperature rating higher than what is expected under normal operation.

315-1975) - IEEE Standard for Graphic Symbols for Electrical and Electronics Diagrams

This IEEE standard relates to how circuit diagrams and schematics are drawn. It gives guidance on the standards to follow when creating a diagram such that another party may be able to easily interpret the circuit diagram themselves.

UL-1741) - Standard for Inverters, Converters and Controllers for Use In Independent Power Systems

This standard covers information about constructing inverters and converters that are not connected to any grid. Therefore, our circuit applies to this standard. The standard covers how to properly choose your components based on the worst case current and voltage that would be applied to it. We followed this while choosing out MOSFETs and diodes [1].

IEEE Policies Section 7.8) - IEEE Code of Ethics

Throughout the project, we have followed the ethics policies set by IEEE. First, we have been forthright with our design and its capabilities. Next, our members have improved their understanding of the concepts used in the project, its uses, and its possible consequences. Also, members have sought out help when it came to our design, took into consideration the criticism other have given, and corrected mistakes. Likewise, we have helped and supported our fellow colleagues in their work. Lastly, all persons that helped or interacted with our group were never discriminated against.

Bill of Materials

Below is the bill of materials for our driver circuit. We based the unit prices on 1000 or more units being produced. This is more realistic for actual manufacturing costs rather than the more expensive prototypes. Our motor driver circuits will costs approximately $46.73 for each unit.

Table 2: Bill of Materials (prices from www.digikey.com)

Item Description	Item Number	Quantity Needed	1000+ units price	Total Cost
FET Driver	HIP4086ABZT	1	3.304	3.304
MOSFET N-Channel	IRF530NSTRLPBF	6	0.6705	4.023
Signal Diode	1N4007	9	0.03645	0.32805
Freewheeling Diode	15SQ100	6	0.16576	0.99456
Voltage Converter 48-12VDC	LM2576HVSX-12/NOPB	1	3.22	3.22
Voltage Converter 48-5VDC	LM2576HVSX-5.0/NOPB	1	3.22	3.22
General Purpose Diode	MUR420	2	0.17651	0.35302
Inductor	PCV-0-104-03L	2	0.61	1.22
Controller Chip	ATMEGA328PB-AU	2	1.1536	2.3072
16MHz Crystal	ABLS-16.000MHZ-D-4-T	2	0.1036	0.2072
Shift Register	SN74HC595N	2	0.2046	0.4092
Multiplexer	SN74LS153NSR	6	0.3696	2.2176
Current Sensor	CQ209A	1	3.19	3.19
Common mode inductor choke	TF5228TU-202Y10R0-01	1	7.875	7.875
Resistor	Multiple	16	0.01292	0.20672
Ceramic Capacitor	Multiple	12	0.0017	0.0204
Electrolytic Capacitor	Multiple	7	0.0524	0.3668
Display	LCM-S01602DSR/A	1	6.3024	6.3024
LED	LTST-C191KFKT	1	0.0376	0.0376
Master power switch	GRS-4011-1600	1	0.464	0.464
Reverse/braking switch	JS202011SCQN	2	0.2338	0.4676
Potentiometer/Knob	PKES60B1/4	2	1.393	2.786
PCB		1	1.47	1.47
PCB Assembly		1	1.74	1.74
			Total:	46.73035

Business Plan

Our company, Loomis Drives, has designed a motor drive that operates under four quadrant (4Q) motoring to achieve precise control, fast response time, and position control of the rotor, proving to be more capable than a motor drive with two quadrant (2Q) motoring.

Our motor drive is intended for use with a three-phase brushless direct current (BLDC) motor. It has been built as a standalone device with the inputs consisting of a 48V supply and the motor hall effect sensors and is rated for 10 amps. Our motor drive is capable of speed control, direction control, position control, and also has regenerative braking, all while showing the real-time state of the motor on a dedicated display.

The motor drive is encapsulated in our uniquely designed case to separate the user from the circuitry. The case not only protects the user from coming in close contact with high voltages,

but it also protects the drive from being damaged due to user negligence. The case is designed with large holes on the sides to allow for proper ventilation and all the user input knobs and switches are properly labeled to make for an easy to use interface.

In order to minimize our risk factors and maximize our profits, our company is implementing several precautionary measures. First of all, we plan on obtaining sufficient capital before starting production (at least one million dollars to cover first quarter costs). We will also research the customer base and their needs and then produce different drive variations to best suit customer needs. Finally, by buying and assembling our motor drives in China, we will be able to undercut competitor prices.

Our main market is the industrial robots industry. As seen in Figure 20, 2017 is expected to see a sale of approximately 330,000 industrial robots with an annual 15% increase. A typical robotic arm requires about five drives, which provides a potential market for 1.65 million drives. Capturing just 5% of this market space will provide us with a demand of 82,500 drives in our first year. Our drives cost $46.73 (cost break down in Table 2) to produce, so at a sale cost of $200 per drive, we expect annual profits of 12.6 million dollars with a minimum 15% annual growth. In concurrent years, we plan to add to our customer base by pursuing the 3D printing market, as well as other robotic markets.

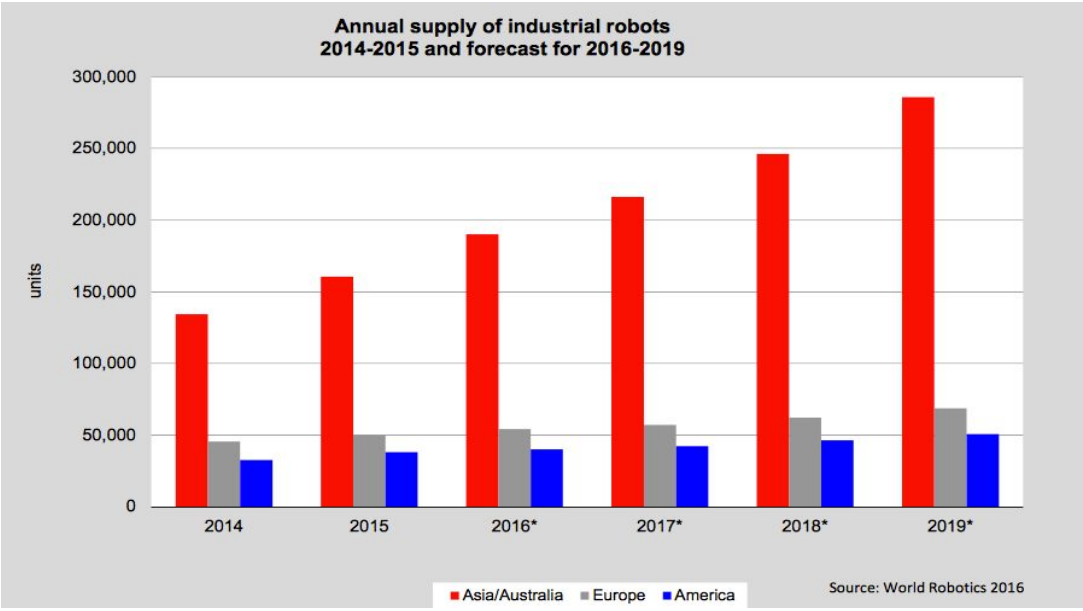

Figure 20: Industrial robot sales from 2014 to 2015 and forecasted 2016 to 2017 sales. The year to year trend suggests a 15% annual increase [2].

Advice

Some advice for future students building a drive for a BLDC motor:

- The common mode filter is useful for making the motor run more smoothly
- Adding snubbers helps to eliminate a lot of the voltage spikes that tend to harm chips
- Using shorter wires/circuit boards cuts down on stray inductance
- Order more parts than you think you might need, and order them well ahead of time
- Try to organize purchasing common parts with other groups to get bulk discounts and spread the cost of shipping between groups.
- Making a PCB will make your circuit more robust/likely to work every time you test it
- Familiarize students with the makerspace and other resources available on campus.
- Test all previous functionality of the driver after every software update

References

[1] Zgonena, Tim. "UL1741 The Standard For Inverters, Converters And Controllers For Use In Independent Power Systems". N.p., 2017. Web. 9 Mar. 2017.

[2] "Statistics - IFR International Federation Of Robotics". *Ifr.org*. N.p., 2017. Web. 5 Mar. 2017.

Appendix

Arduino Motoring Code:

```
#include <PID_v1.h>

//Setup PID
double Setpoint, Input, Output;
double Kp = 0.02;
double Ki = 0.05;
double Kd = 0.0;
PID motorPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
double pidDuty = 0;

//Misc vars
long count = 0;
long prevCount = 0;
long pos = 0;
long prevPos = 0;
int dStep = 0;
long dPos = 0;
float RPM = 0;
long prevTime = 0;
int prevRev = 0;
long stepCount = 0;
int counter = 0;
int loops = 250;

//Reverse Var
int revPin = 7;
boolean reverse = false;
boolean prevReverse = false;
int rDist = 0;

//BRAKING
int brakePin = 8;
boolean braking = false;
boolean prevBraking = false;
int releaseBrakes = 0;

//Position Control vars
int currentPos = 0;
int oldPos = 0;
int posSwitch = 6;
int posPin = A2;
boolean posControlEnabled = false;
long deltaX = 0;
long count4 = 0;

//Duty ratio var
int duty = 0;

// PREP FOR FAST ANALOGREAD
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif
```

```
// MUX BIT OUTPUTS
int mux1A = 49; //PL0
int mux1B = 48; //PL1
int mux2A = 47; //PL2
int mux2B = 46; //PL3
int mux3A = 45; //PL4
int mux3B = 44; //PL5
int mux4A = 43; //PL6
int mux4B = 42; //PL7
int mux5A = 22; //PA0
int mux5B = 23; //PA1
int mux6A = 24; //PA2
int mux6B = 25; //PA3

// HALL SENSOR INPUTS
int HallA = 50;
int HallB = 51;
int HallC = 52;

// HALL EFFECT READING VARS
char my_varA = 0;
char my_varB = 0;
char my_varC = 0;
int A;
int B;
int C;

// MOTOR STEP LOCATION
long curStep = 0;
long prevStep = 0;
long prevStep2 = 0;


void setup() {
  //Initialize Hall effect input pins
  pinMode(HallA, INPUT);
  pinMode(HallB, INPUT);
  pinMode(HallC, INPUT);

  //SETUP HALL INTERRUPTS
  attachInterrupt(digitalPinToInterrupt(HallA), runMotor, CHANGE);
  attachInterrupt(digitalPinToInterrupt(HallB), runMotor, CHANGE);
  attachInterrupt(digitalPinToInterrupt(HallC), runMotor, CHANGE);

  //Initialize mux outputs
  DDRL = 0b11111111; //outputs
  DDRA = 0b11111111; //outputs

  // SET PWM FREQ (31kHz)
  TCCR2B = TCCR2B & 0b11111000 | 0x01;

  // SET INVERSE PWM
  TCCR2A = TCCR2A | 0x30;

  // INCREASE analogRead() SPEED
  sbi(ADCSRA, ADPS2) ;
  cbi(ADCSRA, ADPS1) ;
  cbi(ADCSRA, ADPS0) ;
  //Initialize pot input
  pinMode(A1, INPUT);
```

```
  //SET INITIAL PWM DUTY RATIO
  duty = 128; //50%
  analogWrite(10, duty);
  analogWrite(9, duty);

  //Detect the initial rotor step location
  A = digitalRead(HallA);
  B = digitalRead(HallB);
  C = digitalRead(HallC);
  if (A == 1 && B == 0 && C == 0) {
    curStep = 1;
  } else if (A == 1 && B == 1 && C == 0) {
    curStep = 2;
  } else if (A == 0 && B == 1 && C == 0) {
    curStep = 3;
  } else if (A == 0 && B == 1 && C == 1) {
    curStep = 4;
  } else if (A == 0 && B == 0 && C == 1) {
    curStep = 5;
  } else {
    curStep = 6;
  }

  pinMode(11, OUTPUT);
  pinMode(posSwitch, INPUT);
  pinMode(posPin, INPUT);
  prevStep = curStep;
  prevStep2 = curStep;

  //Initialize all MOSFETs LOW!
  turnOff();

  //Initialize PID function
  Input = RPM;
  Setpoint = (double) map(analogRead(A1), 0, 1024, 0, 3000);
  motorPID.SetMode(AUTOMATIC);
  motorPID.SetOutputLimits(128, 250);
  motorPID.SetSampleTime(50);
}

void loop() {
  //Output to MUX's
  if (!posControlEnabled) {
    runMotor();
    getRpm();
    Input = RPM;
    Setpoint = (double) map(analogRead(A1), 0, 1024, 0, 3000);
    motorPID.Compute();
    managePWM();

    // Run once per ### loops
    if (count - prevCount >= loops) {
      //Check direction/braking and set the PWM duty ratio accordingly
      prevCount = count;
    }
    count++;
  }

  updateRotorPosition();
  //Position control
```

```
  int enab = (int)(PINH & (1 << PH3));
  if (enab != 0) {
    enab = 1;
  }
  if (RPM == 0 && Setpoint == 0 && enab == 1) {
    posControlEnabled = true;
    currentPos = map(analogRead(posPin), 0, 1024, 0, 23);

    if (count4 >= 1000) {
      if (currentPos != oldPos) {
        deltaX = currentPos - oldPos;
        oldPos = currentPos;
      }
      count4 = 0;
    }
    count4++;

    if (pos < 0) {
      pos = pos + 24;
    }

    while (deltaX > 0) {
      duty = 132;
      updateRotorPosition();
      if (curStep != prevStep) {
        deltaX--;
      }
      //UPDATE PWM DUTY RATIO
      analogWrite(10, duty);
      analogWrite(9, duty);
      runMotor();
    }
    duty = 128;
    while (deltaX < 0) {
      duty = 123;
      updateRotorPosition();
      if (curStep != prevStep) {
        deltaX++;
      }
      //UPDATE PWM DUTY RATIO
      analogWrite(10, duty);
      analogWrite(9, duty);
      runMotor();
    }
    duty = 128;

    //UPDATE PWM DUTY RATIO
    analogWrite(10, duty);
    analogWrite(9, duty);
    if (digitalRead(posSwitch) == LOW) {
      posControlEnabled = false;
    }
  }

}


void updateRotorPosition() {
  //GET THE ROTOR POSITION
  if (pos == 0 && prevPos != 0) {
    dPos++;
```

```
  }
  prevPos = pos;
  if (prevStep != curStep) {
    dStep = curStep - prevStep;
    if (dStep == 1 || dStep == -5) {
      //FWD
      pos++;
    } else {
      //BWD
      pos--;
    }
    if (pos == 24 || pos == -24) {
      pos = 0;
    }
    prevStep = curStep;
  }
}

void runMotor() {
  // READ & GET THE HALL SENSOR STATES
  my_varA = (PINB & (1 << PB3));
  my_varB = (PINB & (1 << PB2));
  my_varC = (PINB & (1 << PB1));
  A = (int)my_varA;
  B = (int)my_varB;
  C = (int)my_varC;
  if (A != 0) {
    A = 1;
  }
  if (B != 0) {
    B = 1;
  }
  if (C != 0) {
    C = 1;
  }

  // SET MUX STATES BASED ON STEP LOCATION
  // AND MAKE THE MOTOR TURN!
  if (A == 1 && B == 0 && C == 0) {
    PORTL = 0b11100101;
    PORTA = 0b00000000;
    curStep = 1;
  } else if (A == 1 && B == 1 && C == 0) {
    PORTL = 0b01011110;
    PORTA = 0b00000000;
    curStep = 2;
  } else if (A == 0 && B == 1 && C == 0) {
    PORTL = 0b01010000;
    PORTA = 0b00001110;
    curStep = 3;
  } else if (A == 0 && B == 1 && C == 1) {
    PORTL = 0b11100000;
    PORTA = 0b00000101;
    curStep = 4;
  } else if (A == 0 && B == 0 && C == 1) {
    PORTL = 0b00001110;
    PORTA = 0b00000101;
    curStep = 5;
  } else {
    PORTL = 0b00000101;
    PORTA = 0b00001110;
```

```
    curStep = 6;
  }
}

void turnOff() {
  PORTL = 0b11101110;
  PORTA = 0b00001110;
}

void managePWM() {
  // Adjust duty normally if not changing direction/braking
  if (rDist == 0 && !braking && releaseBrakes == 0) {
    if (reverse) {
      duty = 250 - (int)Output;
      //duty = map(analogRead(A1), 0, 1024, 128, 10);
    } else {
      duty = (int)Output;
      //duty = map(analogRead(A1), 0, 1024, 128, 215);
    }
  }

  //READ REVERSE PIN
  reverse = digitalRead(revPin);
  if (reverse != prevReverse) {
    reverseMotor();
    prevReverse = reverse;
  }

  //READ BRAKING PIN
  braking = digitalRead(brakePin);
  if (braking != prevBraking) {
    brakeMotor();
    prevBraking = braking;
  }

  //If braking switch is turned off go back to
  //normal duty over a period of time
  if (releaseBrakes > 0 && !braking) {
    if (reverse) {
      duty--;
      releaseBrakes--;
    } else {
      duty++;
      releaseBrakes--;
    }
  }

  //Adjust duty ratio to new value over a period of time
  if (rDist > 0) {
    if (reverse) {
      if (braking) {
        duty++;
        rDist--;
      } else {
        duty--;
        rDist--;
      }
    } else {
      if (braking) {
        duty--;
        rDist--;
```

```
      } else {
        duty++;
        rDist--;
      }
    }
  }
  //UPDATE PWM DUTY RATIO
  analogWrite(10, duty);
  analogWrite(9, duty);
}

void reverseMotor() {
  //Reverse switch has been flipped
  if (reverse) {
    //prev state was forward
    rDist = (2 * (duty - 128));
  } else {
    //prev state was reverse
    rDist = (2 * (128 - duty));
  }
}

void brakeMotor() {
  //Brake switch has been flipped
  if (braking) {
    if (!reverse) {
      rDist = duty - 128;
    } else {
      rDist = 128 - duty;
    }
    releaseBrakes = rDist;
  }
}

void getRpm() {
  // CALCULATE RPM
  if (curStep != prevStep) {
    stepCount++;
  }
  prevStep = curStep;
  if (millis() - prevTime >= 250) {
    RPM = (stepCount / 24.0) * 240.0;
    stepCount = 0.0;
    prevTime = millis();
  }
}
```