# High Efficiency Brushed and Brushless DC Motor Controller with Simple User Interface


## Team Members

Sam D. Ellicott, CEO

Isaac J. Jones, CFO

## Team Advisor

Dr. Gerald Brown, Ph. D. EE


## Report Date

5/1/19

**Table of Contents**


Table of Contents....2

Project Definition Statement....3

Abstract....3

Background....4

Project Objectives....5

- Hard Requirements....5
- Soft Requirements....6
- Constraints....6

Engineering Design....7

- Component Selection....8
- Control and Signal Processing PCB....10
  - Inputs and Outputs....11
  - Power Supply....13
  - Subsystems....15
  - Signal Conditioning....16
  - STM32 Subsystem....18
  - TMC4671 Subsystem....21
- Power Conversion PCB....23
- Test Setup....28
  - Serial Peripheral Interface....29
  - Analog to Digital Conversion....30
  - CAN Interface....32
  - C++ Abstraction Classes....34
  - TMC4671 Interface Class....35

Settings Manager Class....36
Computer Interface Class....37
Menu System....40
PCB Temperature Under Load....43
Power Capacity....44
Snubber Performance....47
Conclusion....49
Personal Contributions....50
Sam Ellicott....50
Isaac Jones....50
Bibliography....51
Gantt Chart....53
Schematics....54
BOM....54

## Project Definition Statement

Create a reliable, efficient, flexible, and well documented motor controller for the Cedarville University Supermileage team.

## Abstract

This project's goal is to provide the Cedarville University Supermileage team with a high-efficiency electric motor controller for use in any category of vehicle. The controller should be simple enough to use that a competent freshman or sophomore engineer on the team could set it up. It shall be capable of operation without an external computer, so that it is more reliable in a competition environment. The motor controller should be able to provide a variety of control techniques including speed and torque control. The setpoints for these control techniques should be controllable by the user via

a controller area network (CAN) bus connection or an analog throttle input. Important motor controller settings may be configured in a personal computer graphical user interface (GUI). The goal of this system is to provide the Supermileage team a reliable, efficient, flexible, and easy to use system.

## Background

The Supermileage team would like to compete in the battery electric and hybrid vehicle categories at Shell Eco Marathon. This requires a custom built motor controller. Because the controller is for the Supermileage team, it should be as efficient as possible. The project also requires a simple and well documented setup process that can be used by freshman and sophomore engineering students, or the system will be unusable in future years.

The vehicles have a tablet computer to relay telemetry and vehicle information back to a server at Cedarville. This computer system performs an optimization loop based on many factors (ground speed, wind speed/direction, etc) to find the most efficient vehicle speed at each moment. However, the system is complex and is often not reliable or fully functional at the time of competition. Therefore, the motor controller must be capable of stand-alone operation without an external computer. It is a requirement for the motor controller to be able to provide most functionality with a simple analog throttle input.

One approach attempted in previous iterations of this design project was to control the duty cycle of output transistors directly with the microcontroller. Integrating the pulse width modulation (PWM) generation into the microcontroller reduces the component count of the design and increases the flexibility of the controller. However, this configuration greatly increases the complexity in the microcontroller, as the current in the motor must be monitored and corrected alongside any other functions the microcontroller performs. This has led to systems from previous teams that are either inefficient, or did not function entirely.

## Project Objectives

### Hard Requirements

The controller shall:

- have a power conversion efficiency greater than 85%
- be capable of providing over 1125 Watts (1.5HP) of output power during acceleration
- be capable of driving a brushless DC motor
- be capable of driving a brushed DC motor
- have setpoint based on a 0.0-5.0V analog input
- have setpoint based on a CAN input
- maintain a constant speed based on a setpoint (speed control loop)
- maintain a constant torque based on a setpoint (torque control loop)
- output the motor temperature over CAN
- output its transistor case temperature over CAN
- output the motor current over CAN
- output motor speed over CAN
- reduce output current in the case of an over temperature fault
- operate with an input voltage range between 24V and 55V

The GUI shall:

- run on a personal computer
- connect to the motor controller over a USB connection
- configure the control mode of the motor controller (e.g. torque or speed control loops)
- configure important motor control parameters
    - Maximum acceleration current
    - Maximum regeneration current

- Map speed values to analog input values (speed control mode)
- Map torque values to analog input values (torque control mode)
- Fault temperature thresholds
- CAN input ID’s
- CAN output ID’s

**Soft Requirements**

The controller should:

- measure battery voltage
- output battery voltage over CAN
- measure battery current
- output battery current over CAN
- have 4 digital (3.6v logic level) input pins configurable in software
- have 2 analog (0-3.6v) input pins configurable in software
- have 2 digital (3.6v logic level) output pins configurable in software
- have 2 analog (0-3.6v) output pins configurable in software
- implement a cruise control algorithm based on a pitot tube

The GUI should:

- be portable across operating systems
- configure digital outputs to map to a list of fault conditions
- configure analog outputs to map to a list of variables
- configure digital inputs as cruise control inputs

**Constraints**

- The nominal voltage of the system shall not be greater than 48V
- The voltage at any point in the system may not exceed 60V

# Engineering Design

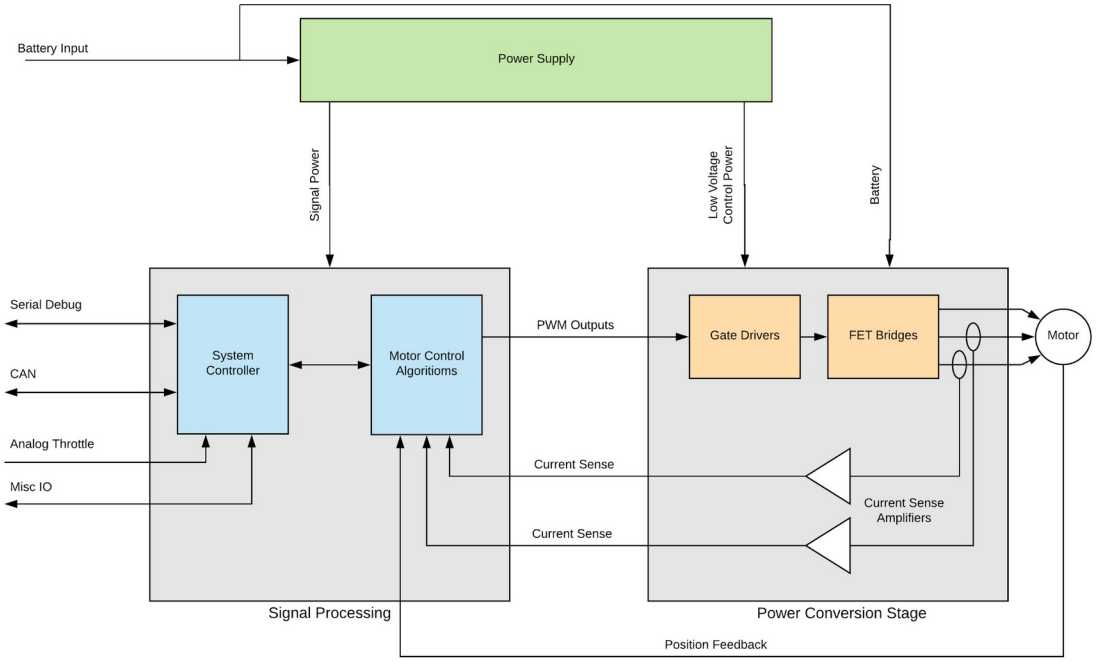



Figure 1: Motor Controller System Block Diagram

## System Overview

Our motor controller has two main sub-units, a signal processing section, and a power conversion section (See Figure 1). The Signal processing stage uses an STM32 microcontroller to convert a speed or torque set-point from the CAN bus or an analog throttle value into an SPI command for the TMC4671 PWM generation chip. The outputs from this chip are then used by the gate driver circuitry to switch the high-power MOSFETs on and off. The MOSFETs control the phase currents (or current in a DC motor) flowing through the motor. Both this current and the rotor position (when driving a BLDC motor) of the motor are fed back into the TMC4671 to form a feedback loop. This feedback allows for more efficient control of the motor. An additional power supply section is needed to convert the 48V battery voltage down to 3.3V and 5V for the digital circuitry.

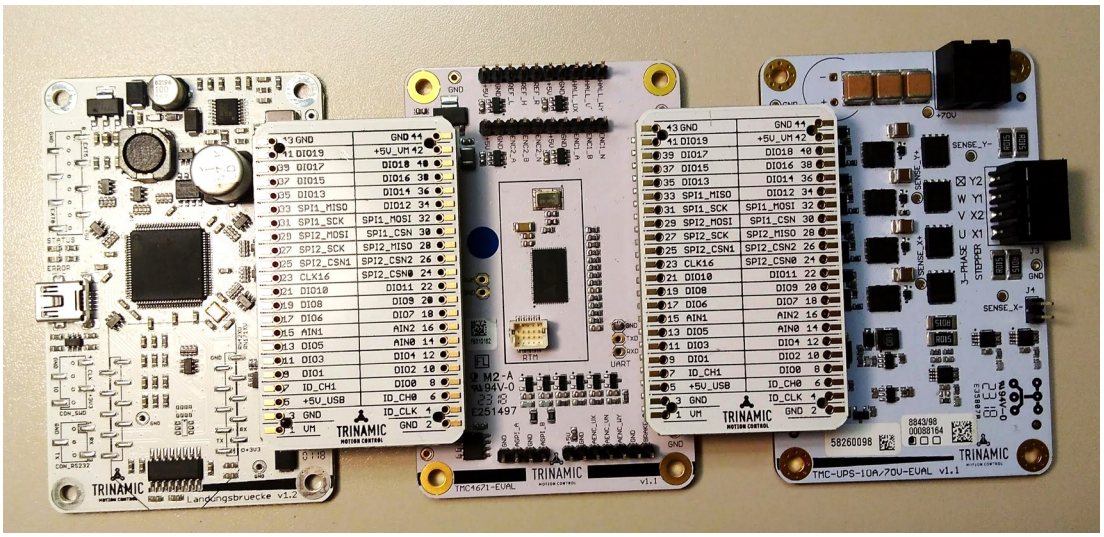



Figure 2: TMC4671 Evaluation Kit

In order to speed development we used a Trinamic TMC4671 evaluation kit (Figure 2) generously donated by Trinamic, as a reference for our design. The evaluation kit uses a modular design, split between a Microcontroller board (Landungsbrücke), TMC4671 breakout board, and a 10 amp power conversion stage. Each of these modules is connected to the next one by a PCB bridge (Eselsbrücke). We designed the two parts of our system to be modular so that we can replace the corresponding portions of the evaluation kit with our custom design. For example, the signal processing and control board from our design can replace the functionality of the Landungsbrücke and TMC4671 boards, connecting to the Trinamic power stage. Likewise, our power conversion board can replace the one from the Trinamic design. By using Trinamic's proven designs we were able to quickly test our designs against known working hardware.

**Component Selection**

Efficient motor control algorithms such as field oriented or vector control (FOC) require advanced PWM generation strategies which are beyond the scope of this project. For this reason in our design, we are using a commercial off-the-shelf microchip to implement the motor control algorithm and PWM generation for outputs to the power electronics (Motor control algorithms block in Figure 1). For our design we chose the Trinamic TMC4671 chip. This chip provides almost all of the features that we

need to implement in the controller as it supports FOC for both three-phase and brushed DC motors. It also implements velocity and current control loops in hardware. This was one of the few chips that provided all of the features that we wanted, as most of the other IC's were either designed as fan controllers, or were microcontrollers in their own right.

The microcontroller we selected (the STM32F303) was chosen because we were familiar with the STM32 line of chips, and this particular part had enough GPIO pins for our application, supported CAN, has USB programing capability, and has a hardware floating point unit. Additional reasons for the selection of the STM32 part are the fact that it has a GUI to facilitate setting up all of the peripherals in the chip (STM32CubeMX) and has much more processing power than a similarly priced 8-bit microcontroller.

For the project's power stage, we've designed a three-phase DC to AC inverter using three MOSFET half-bridges. This enables us to drive a brushless DC (BLDC) motor or a brushed DC motor, depending on the TMC4671 configuration. There is an additional half-bridge included in our design to allow the control of high-power brushed DC motor control by paralleling two half bridges for each motor terminal. The MOSFETs we've selected are the IRFP3306PbF for their high current capability (120A DC), low gate capacitance (85 nC), and low thermal resistance (0.67 °C/W). Supporting these MOSFET half-bridges are anti-parallel Schottky diodes to compensate for the body effect diode present in each MOSFET. We also included an RC snubber circuit for each half-bridge and a zener gate protection diode for each high-side MOSFET.

The high-side and low-side MOSFETs are driven by gate driver ICs, the LM5109B. Our design requires 3 of these chips to drive a BLDC motor, given that each LM5109B can drive one low-side and one high-side MOFSET. However, with the addition of the optional fourth phase, our power PCB requires four gate driver ICs. The LM5109B includes integrated capacitive charge pumps for driving the gates of the high-side MOSFETs above the positive battery voltage rail. The LM5109Bs are powered through a 12V DC/DC buck converter off the overall battery voltage.

The buck converter was designed using an LMR16020 step-down controller. We chose this IC because of its wide input voltage range (up to 60V) and 2-amp average output current. In addition, the LRM16020 has an “ENABLE” input which allows the microcontroller to disable the MOSFET power stage for power conservation or in the event of a fault. The “PGOOD” output pin adds a handy signal-level indicator of proper functionality during operation, so we utilized this pin to control a red/green LED pair on the power PCB. Our power supply design was greatly simplified due to the LMR16020 requiring few external components for DC voltage conversion – it even includes an integrated high-side MOSFET and charge pump driver for simplicity.

We have current sensors measuring motor currents in phase A and phase B, in addition to a DC-side current sensor. Our selection of current sensor is the ACS723, which is a Hall-effect current sensor with integrated amplifier to output a voltage between 0 and 5V corresponding to the current flowing through it. The specific model of ACS723 that we chose (40AB) allows for bidirectional current sensing with a maximum amplitude of 40A. The analog voltages representing currents are then fed over the Trinamic bridge (Eselsbrücke) to the TMC4671 PCB for Field-Oriented Control. We also included a test point for each current sensor in the power PCB design so that we can debug any issues with an oscilloscope probe.

**Control and Signal Processing PCB**

Detailed schematics are required in order to implement the design as a PCB. These schematics, in their current revision will be examined here. As described earlier, the motor controller is split into two PCBs, one for the microcontroller and TMC4671, and another for the power conversion circuitry. The former of these will be described first. Figure 3 shows the front of the signal processing PCB with a number of the features highlighted.

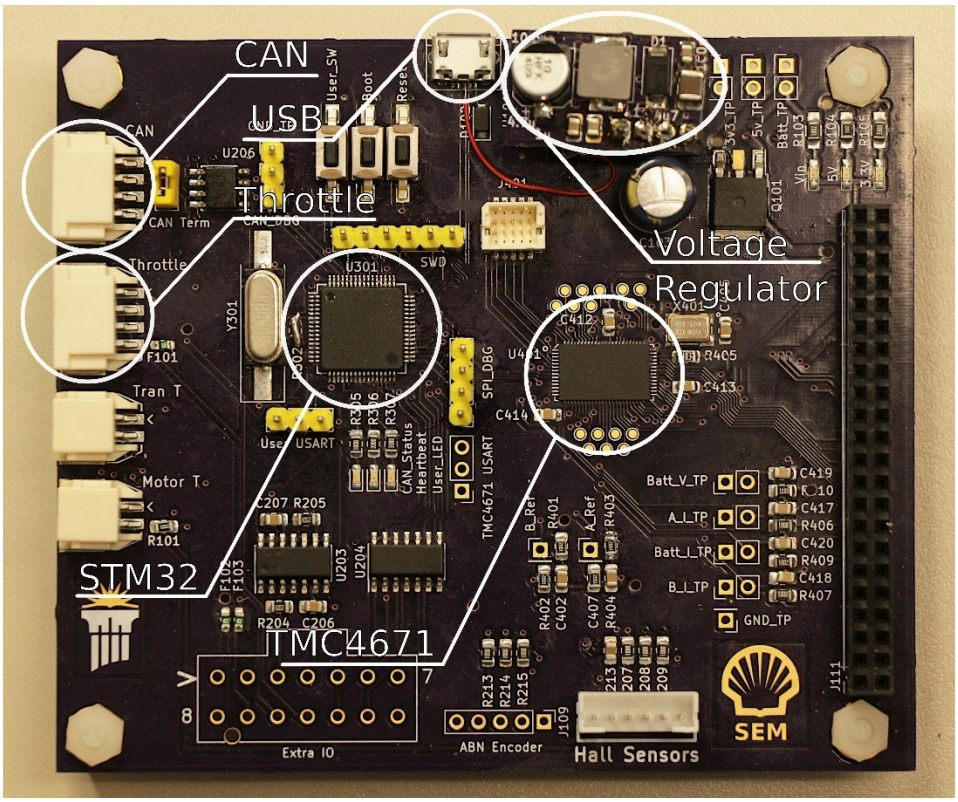



Figure 3: Signal Processing/Control PCB with key features and components highlighted

**Inputs and Outputs**

The main input and output connections for the board (IO) are shown in Figure 3 and Figure 4. The main interface to the motor controller is designed to be the CAN connection. In an attempt to keep the number of different connectors in the vehicle low, we have selected a 4-pin Molex DuraClik connector, as this is the same connector currently used in the vehicle for CAN. Additionally, the same pinout was used for these connectors as is used in the vehicle. The throttle input also uses the same 4-pin DuraClik connector. The temperature sensor inputs use 2-pin DuraClik connectors. Additional IO pins are provided on a 14-pin connector, these inputs go to the STM32 GPIO pins, after going through some signal conditioning (described later).

All of the power connections on the user facing IO ports have a small positive temperature coefficient (PTC) thermal fuse on them. This helps ensure that the board will not be damaged by an

accidental short in the vehicle (a tragically common occurrence). A thermal fuse was selected so that it would reset after the fault has been corrected.

The Trinamic bridge connector (shown in Figure 5) allows for the connection between the control board and the power conversion board. It also follows the same pinout as the TMC4671 evaluation kit, allowing for use with the Trinamic power conversion PCB.

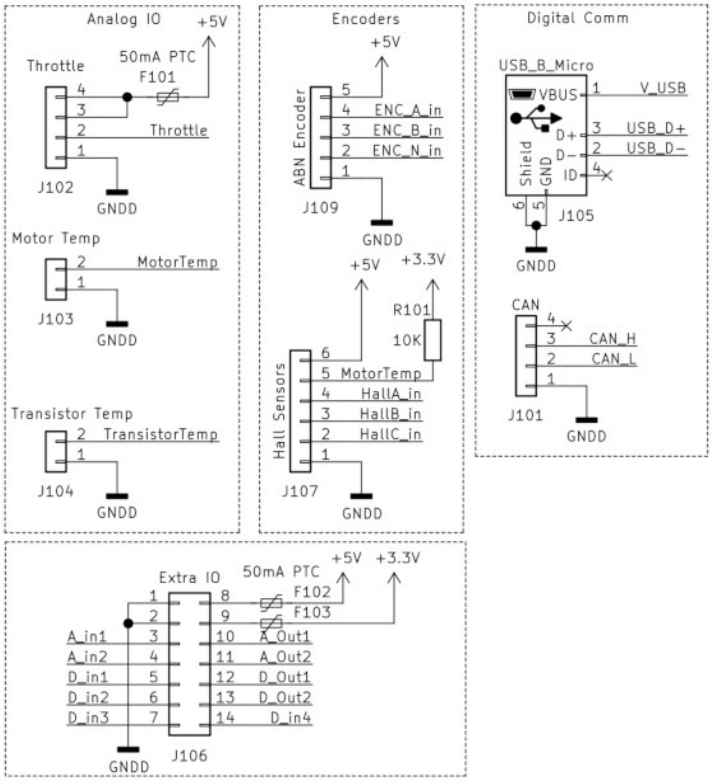



Figure 4: Input and Output Connectors

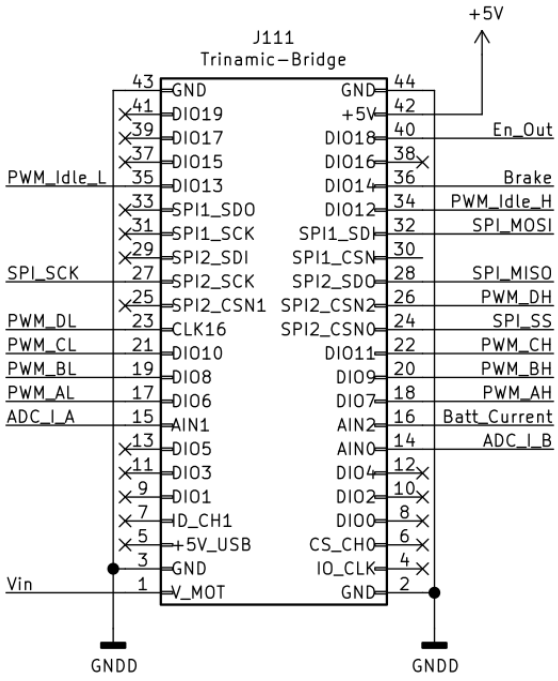



Figure 5: Trinamic compatible board connector

**Power Supply**

The system is supplied with 5V and 3.3V by a switching voltage regulator module and a linear regulator (Figure 6). The 5V regulator module is protected from reverse voltages by a P-Channel MOSFET in an ideal diode configuration. An 18V zener diode prevents the gate of the transistor from going over its maximum Vgs voltage of 20V, the 10K resistor limits the current flowing through the diode to about 3mA. The off the shelf module which we were planning on purchasing became unavailable, so our own module was developed using the LMR16010 chip. This is a smaller variant of the switching regulator used in the power section. The design was effectively copied from the design from the power section and placed on a small PCB which was made to be a drop-in replacement for the intended module. The PCB and schematic for the design is included for completeness in Figure 7 and Figure 8.

In order to power the STM32 microcontroller for programming the 5V rail is connected to the 5V from the USB port through a Schottky diode. This protects the USB port both from reverse voltages and reverse currents. A simple linear 3.3V regulator is used to convert the 5V rail down to 3.3V.

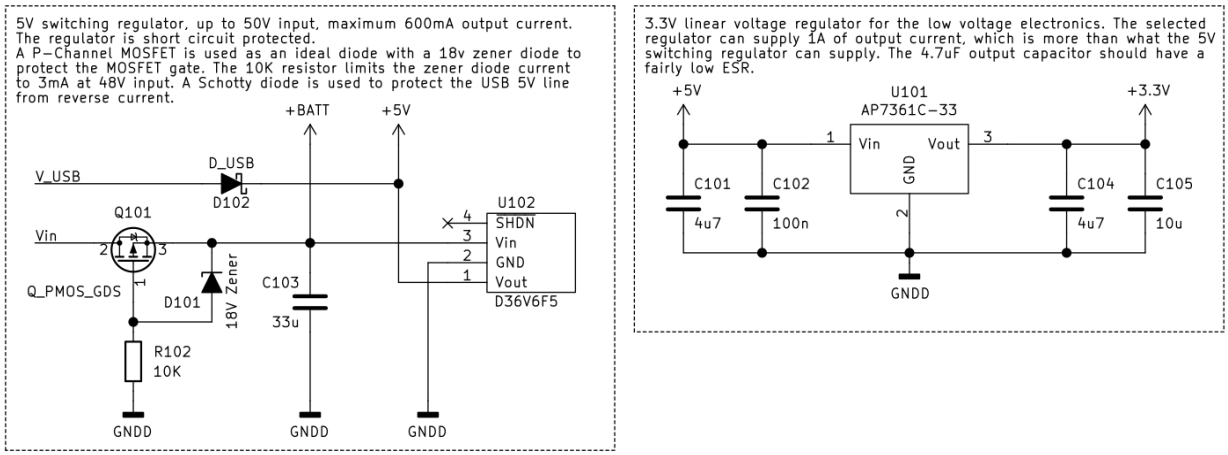



Figure 6: Low Voltage Power Supply

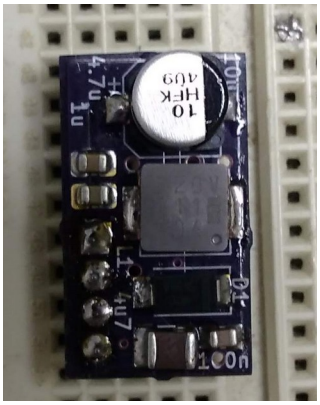


Figure 7: Voltage Regulator PCB

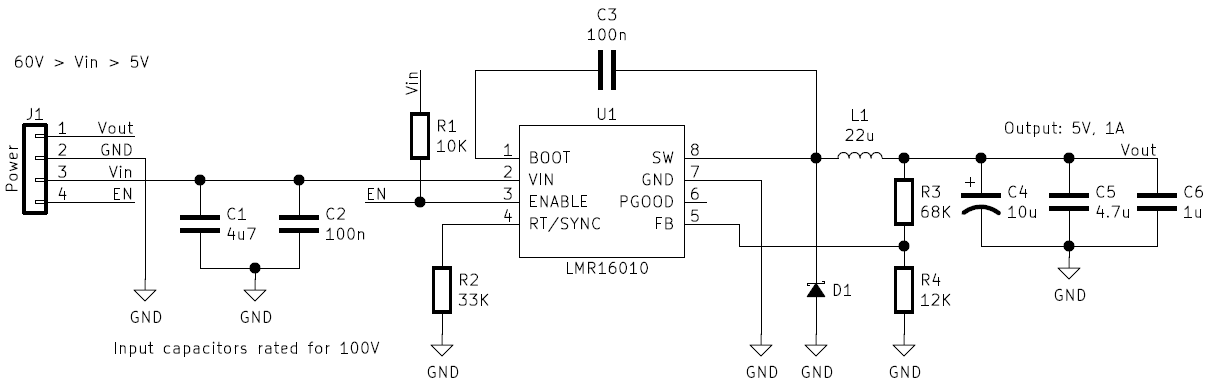



Figure 8: 5V Voltage Regulator Module

## Subsystems

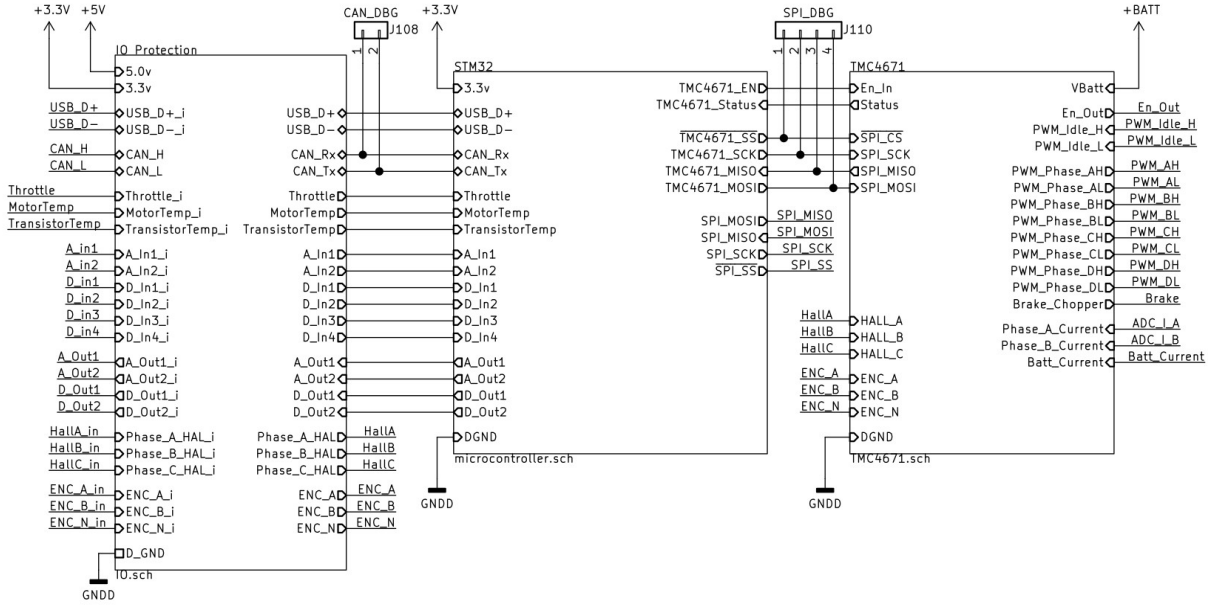



Figure 9: Detailed Signal Processing subcomponents

The other parts of the schematic are housed in three subsystems, as shown in Figure 9. These subsystems are for signal conditioning, the microcontroller, and for the TMC4671 respectively. These blocks clearly show the interconnections between the various parts of the PCB. Also note that there are two sets of connectors for attaching a logic analyzer to both the CAN and main SPI serial interfaces. Both of these will aide in debugging while doing software development.

## Signal Conditioning

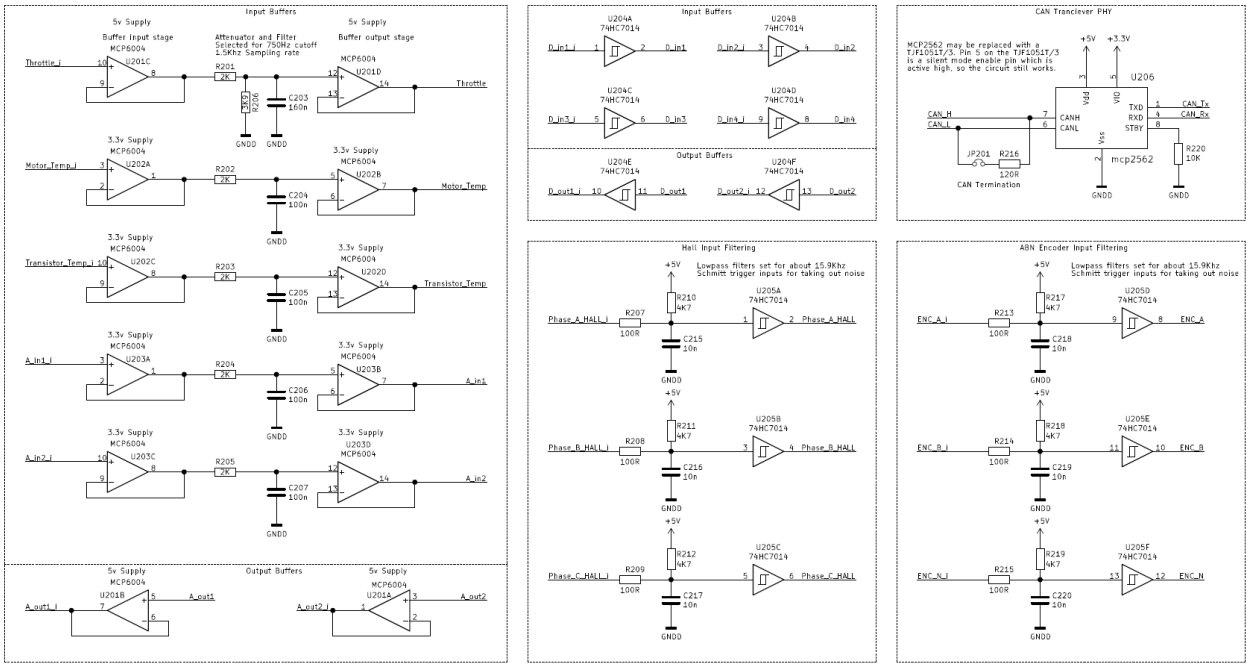



Figure 10: Input Conditioning Circuit

Figure 10 shows the signal conditioning circuitry. Three MCP6004 quad rail-to-rail op-amps are used to buffer the analog signals coming into and going out of the PCB. Incoming analog inputs are passed through a non-inverting buffer, a filter, then another buffer. The first buffer prevents high impedance signals from being distorted by the filter impedance. The filter limits the bandwidth of the incoming signal to 750Hz, making the minimum sample rate of the ADC 1.5 KHz. Because the analog input signals to the motor controller should be very low frequency, this sample rate was chosen to limit the amount of time required by the microcontroller to sample the signals. The last buffer prevents the 50KΩ input resistance of the ADC from distorting the signal. Analog outputs from the microcontroller are only passed through a non-inverting buffer.

The analog throttle input is somewhat special as it requires a 0-5V input voltage range. In order to account for this, its op-amps are supplied by the 5V rail as opposed to the 3.3V one. Then after the first buffer, before the low-pass filter, the signal is fed through an attenuator, ensuring that it remains below 3.3V.

Digital inputs are fed through 74HC7014 Schmitt trigger buffers (these were also used in the TMC4671 evaluation kit). These help keep high frequency noise from causing spurious transitions on the digital signals. Additionally, the buffers act as level shifters. The 74HC7401 has 5V tolerant inputs, but since it is running from a 3.3V supply, it outputs at 3.3V levels.

The last component in this subsection is the CAN transceiver chip (upper right corner of Figure 10). This chip converts the single ended CAN_Rx and CAN_Tx signals to a differential CAN signal. There is also a selectable jumper to enable a 120Ω termination resistor. The design calls for a MCP2562 but this was replaced with a TJF1051T/3 transceiver as they have the same pinout and are lower cost.

**STM32 Subsystem**

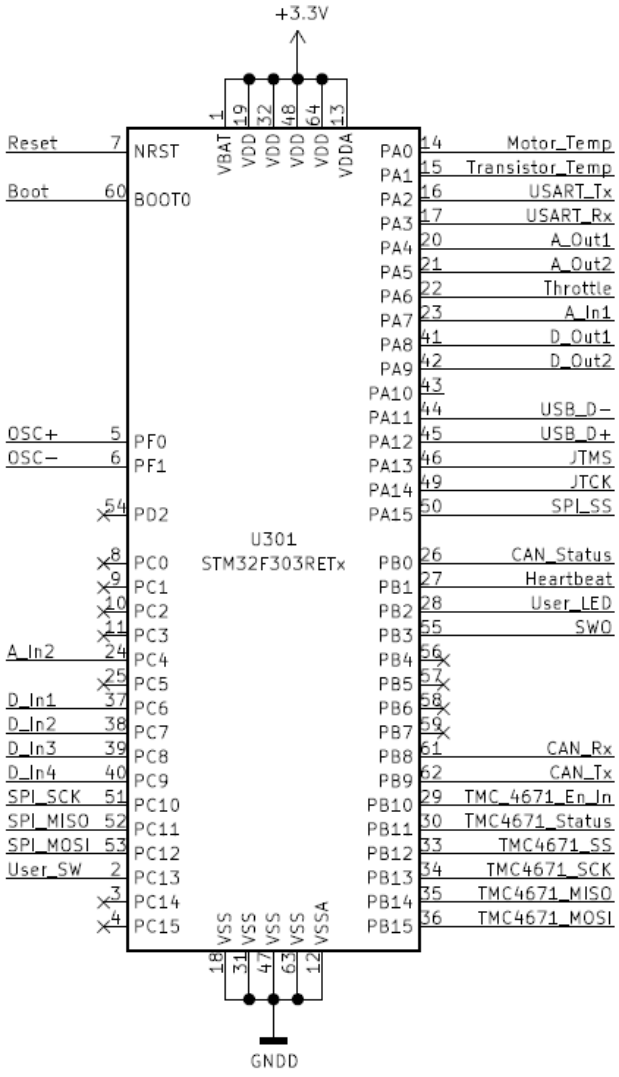



Figure 11: STM32 Microcontroller Connections

Figure 11 and Figure 12 show the pin configuration of the microcontroller and a screenshot of the STM32CubeMX software which was used to determine which peripherals mapped to which pins. This software was also used to generate base C code for the software in the project. This tool allows the user to pick which peripherals in the microcontroller they want to use, then it will configure the pins which map to those functions. This makes setting up the STM32 much simpler.

The STM32 has a USB bootloader in the chip's ROM, this is activated by pulling one of the pins (BOOT0) high after the microcontroller has been reset. The microcontroller can be made to enter this

mode by holding the boot button down while toggling the reset button (Figure 13). This mode is used to program the microcontroller. These two buttons can be seen on the signal processing PCB in Figure 3 as the middle and right buttons respectively. The STM32 also features a JTAG interface which allows both programing and debugging of the chip.

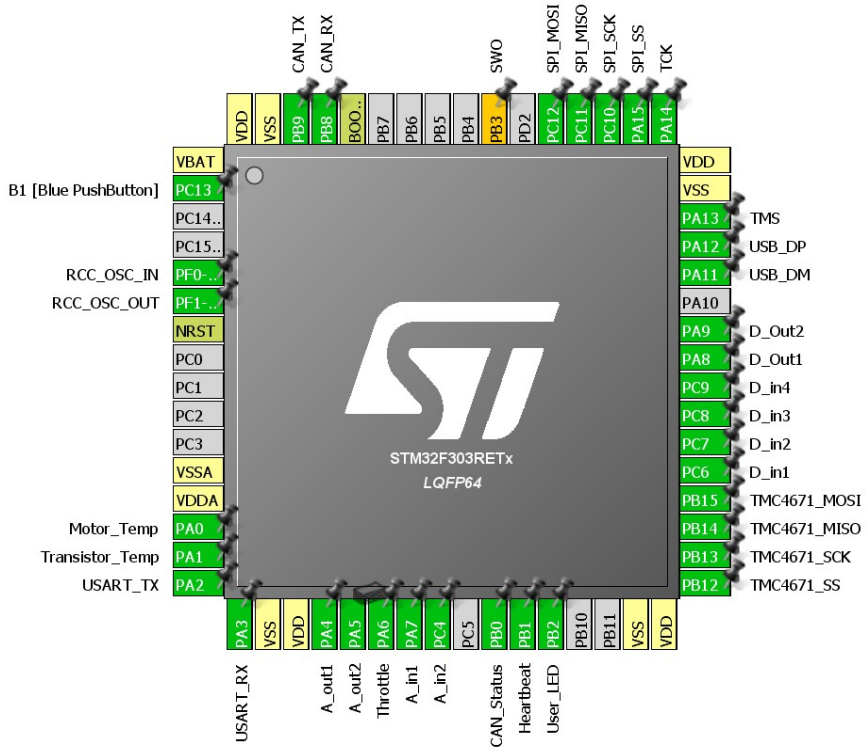



Figure 12: STM32CubeMX configuration tool screenshot

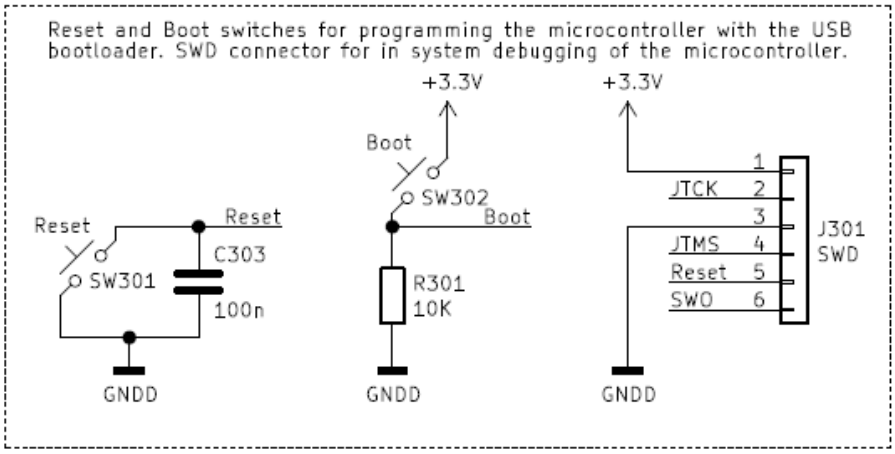



Figure 13: STM32 Programing components

In order for the USB peripheral to work, an external 8-Mhz clock crystal was needed (Figure 14). I used an article on the AllAboutCircuits website to pick my loading capacitor values. The STM32Nucleo evaluation board schematics were used to determine the current limiting resistor on the OSC- line (R302).

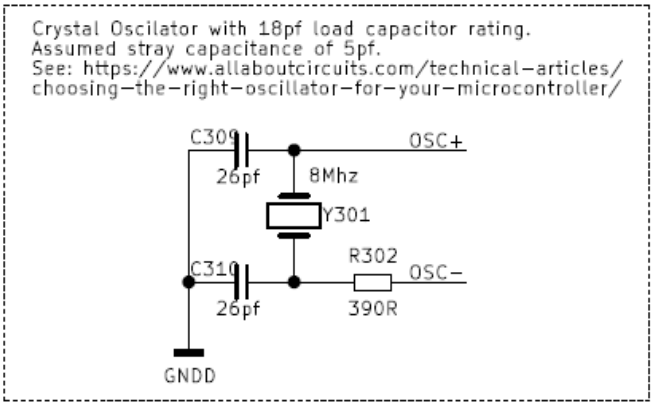



Figure 14: Crystal Oscillator

**TMC4671 Subsystem**

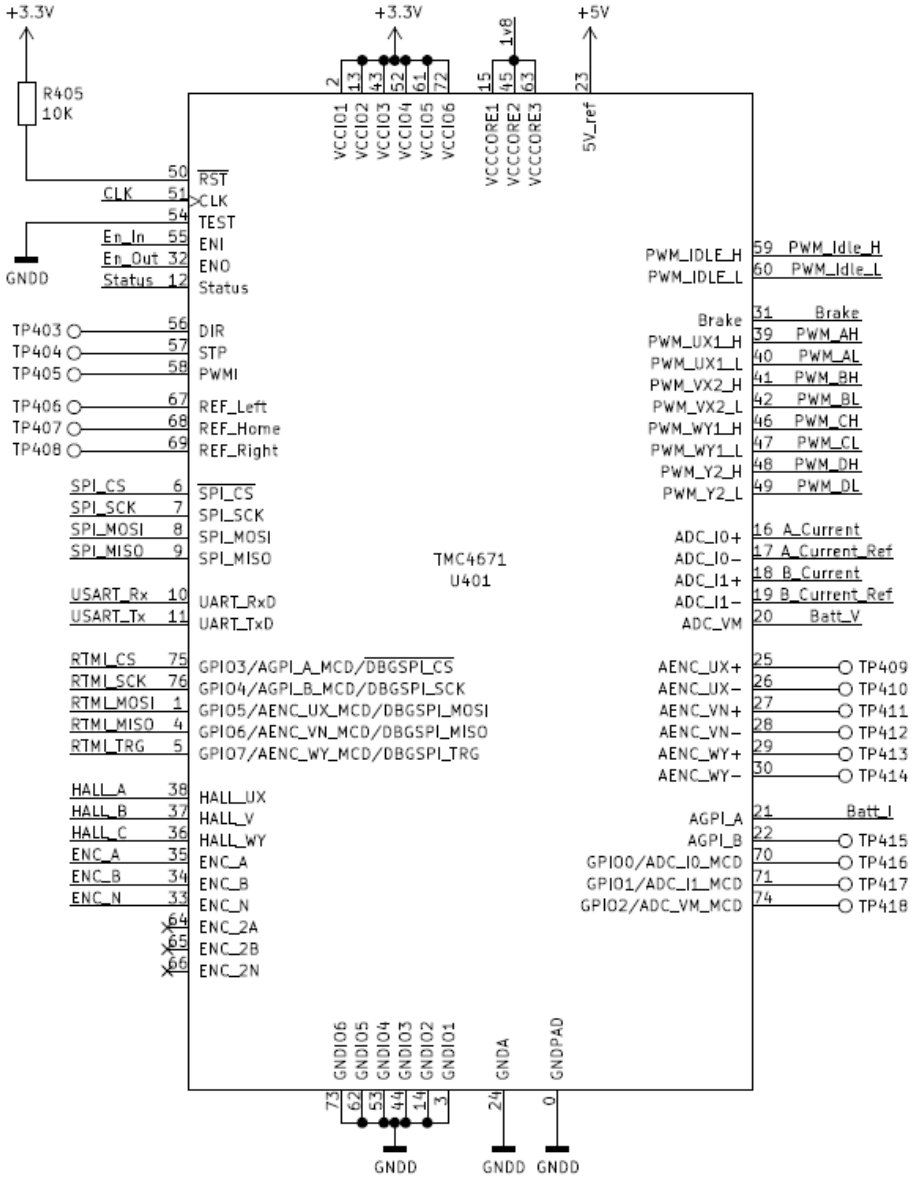



Figure 15: TMC4671 Motor Controller Chip

Figure 15 shows the TMC4671 and its pin configurations. The TMC4671 takes its primary input from the SPI bus connected to the STM32 microcontroller, however there are several other inputs which the chip needs in order to function properly. The most significant of these other inputs are the Hall-effect inputs (alternatively the ABN encoder), and the phase current inputs. The former of these signals are taken from the signal conditioning subsection, where they were cleaned by a Schmitt trigger buffer. The current signals however still need to be conditioned. The TMC4671 evaluation kit has filtering which are

supposed to produce a cutoff frequency of 194 KHz, In order to reduce the number of different values in the circuit, I chose a cutoff frequency of 159 KHz for the phase currents (Figure 16). The TMC4671 also monitors the battery voltage through a voltage divider. The values were chosen so that a 60V input would produce a 5V output to the ADC. A lower cutoff frequency was chosen for this ADC input as the battery voltage should not fluctuate as much as the current inputs.

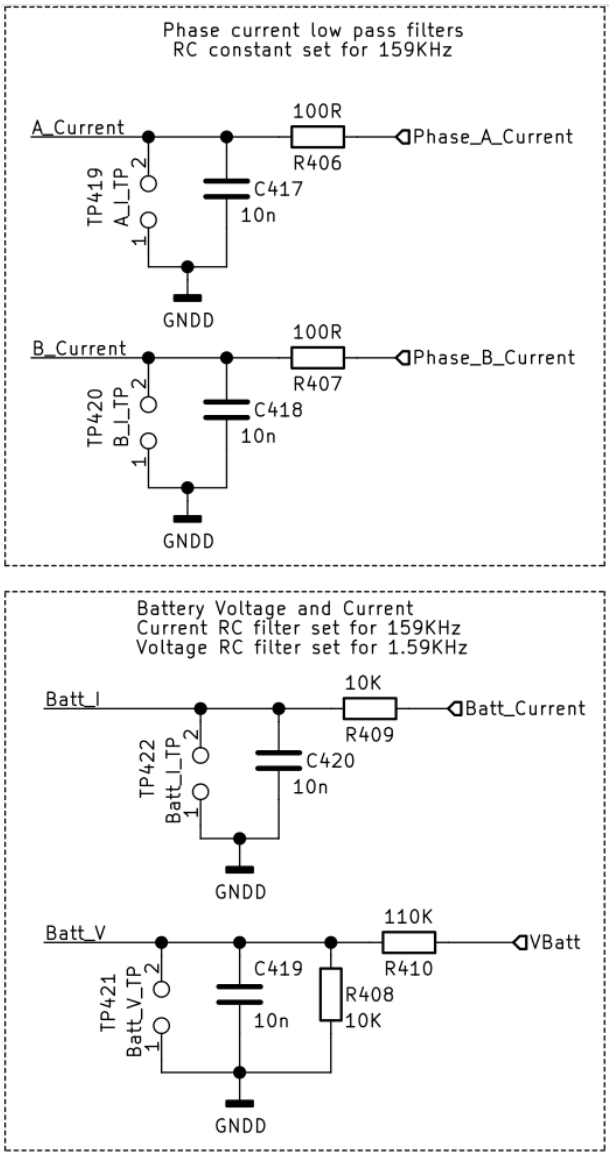



Figure 16: TMC4671 ADC filtering

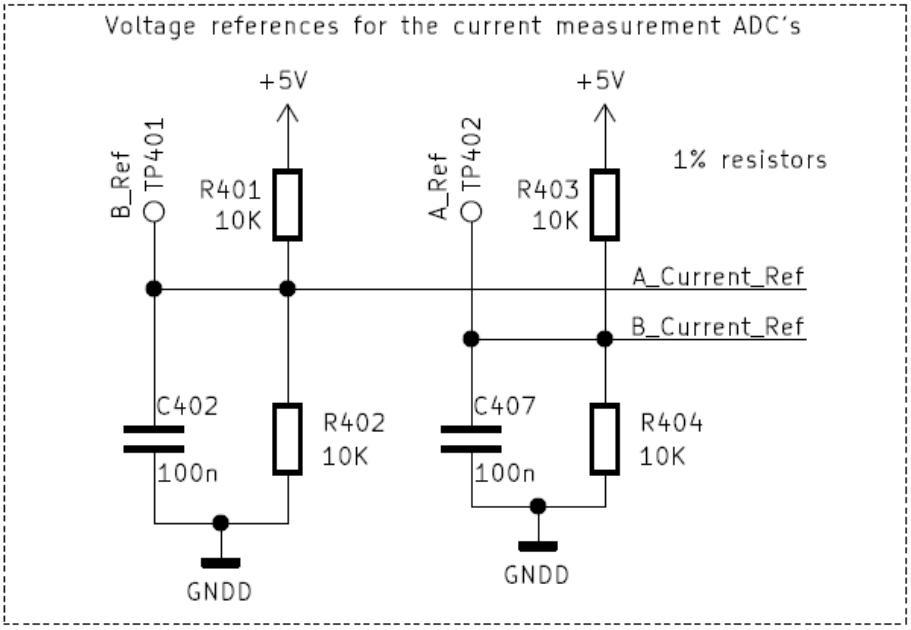



Figure 17: ADC negative reference voltage

The ADC inputs for the phase currents on the TMC4671 chip are differential, and for the most accurate measurements the inverting input should be set to be whatever the reference is for the current sensors. For our phase current measurement devices, no current flow corresponds to a 2.5V output. Therefore, two 10K resistors are placed in a divider from the 5V rail to produce a 2.5V reference (Figure 17). Any small difference between the current amplifier and the reference can be calibrated out in software.

**Power Conversion PCB**

For this portion of the motor controller, we have 4 MOSFET half-bridges to be able to drive a BLDC motor with an optional braking resistor bank. The 4 phases allow us to drive a DC motor with more current, given that we can parallel 2 phases for each side of the DC motor. Of course, a final user can avoid the paralleling and simply use phases 1 and 2 for brushed DC motor control if desired.

The detail of a half-bridge MOSFET configuration can be seen in Figure 18 below. Each MOSFET has an anti-parallel Schottky diode connected to its drain and source to compensate for the MOSFET's lossy and slow body diode. The Schottky is oriented in the same direction as the MOSFET

body diode, and provides a voltage drop of 0.3V instead of the MOSFET diode's 1.3V drop. This allows for a more efficient PWM operation, because less energy is burned up as heat – this separate diode also removes some heating from the MOSFETs to let them run cooler.

In order to reduce voltage ringing during switching and further protect the MOSFETs, we included a simple R-C snubber across the lower switch (PH4 to MOS_GND in Figure 18). Through empirical data collection and experimentation with the actual PCB, we determined that R = 10Ω and C = 10nF would yield the best balance between ringing-suppression and snubber heat loss using parts on-hand. The testing of these circuits is described later.

Another protection element we included was a 15V zener diode to ensure that the high-side transistor gate-to-source voltage does not exceed its breakdown voltage of 20V. With this voltage rating, our gate drive ICs should never cause an overvoltage issue (each LM5109 drives 12V gate-to-source to an ON gate), and our MOSFET gates will be protected from transient voltage surges. The 10Ω resistors on each MOSFET gate limit the in-rush current peak into each gate and dampen potential oscillations between the gate capacitance and stray PCB inductance.

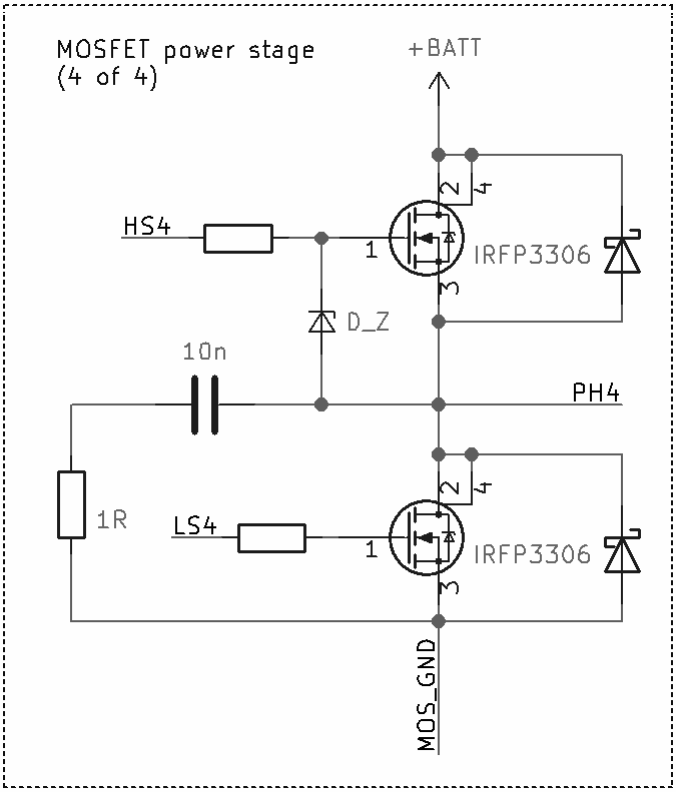



Figure 18: MOSFET half-bridge and peripheral components

In order to drive each MOSFET with its respective signal from the TMC4671 board, we're using 4 LM5109B half-bridge gate driver ICs. Figure 19 details the circuit of one example LM5109B in our circuit. The gate driver IC is powered by a 12V buck DC/DC supply (discussed below) to drive a low-side MOSFET gate safely below the MOSFET gate breakdown voltage (+/-20V). In addition, each gate driver includes a capacitor charge pump to drive the high-side MOSFET gate with a voltage greater than the positive power bus. Two digital signals (0-3.3V) from the TMC4671 enter each gate driver to control the low-side and high-side driver components of the LM5109. Using the recommended datasheet configuration, we selected a value of 0.1uF for the bootstrap capacitor.

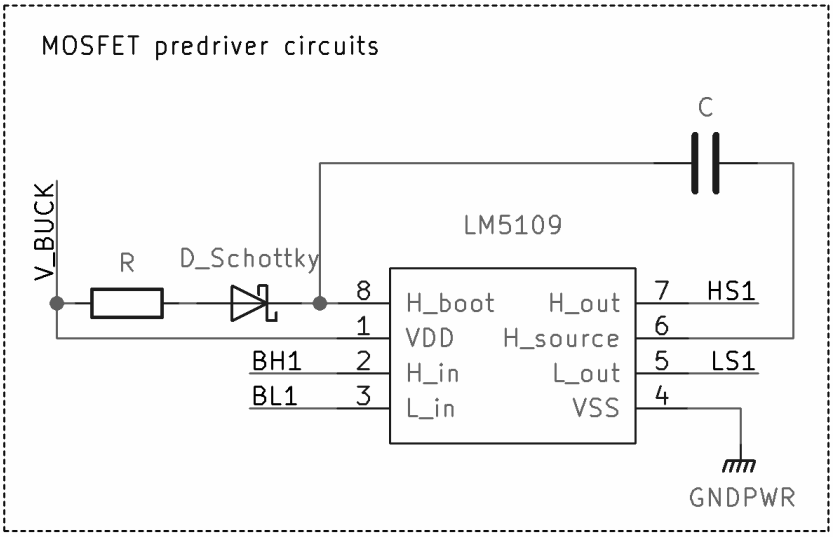



Figure 19: Phase A MOSFET pre-driver circuit

In order to enable current monitoring and FOC, we are using three hall-effect current sensor ICs. We decided upon the ACS723, given its surface-mount package, current-measurement range, and ease of design. Our 3 current sensors measure Phase A, Phase B, and DC currents individually; these analog signals are then routed back to the TMC4671 analog inputs. Each ACS723 has a bandwidth-selectable value of 20kHz or 80kHz, determined by the digital value on pin 6 (“BW_SEL”) we use a 3-pin header jumper on the PCB to allow simple bandwidth changes after PCB manufacture. We are using the full 80KHz bandwidth as the signal was clean without the additional 20KHz filter. Figure 20 shows the schematic of the phase A current sensor.

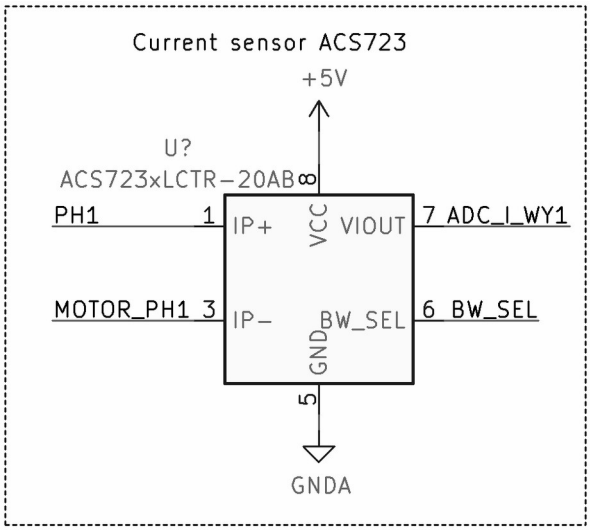



Figure 20: This current sensor measures phase A current

The buck DC/DC converter on the power PCB receives polarity protected battery voltage (up to 48V nominal) and supplies 12V to the gate drive ICs. We selected the LMR16020, which offers an integrated MOSFET and 2A of current throughput (Figure 21). Note that the signal processing PCB and the power PCB have separate power supplies, due to different supply voltage requirements. Required supporting circuitry includes an L-C filter loop with diode to smooth the output voltage, a resistor voltage divider feedback, and a bootstrap capacitor for driving the integrated P-channel MOSFET. The LMR16020 also includes an "Enable" pin (connected to the TMC4671 board in our design) and a "Power Good" open-drain output pin (controlling an LED on the board). This buck controller IC is versatile, even allowing the designer to set switching frequency through a pull-down resistor or a digital clock signal. For a buck voltage of 12V, we selected a feedback resistor divider of 150kΩ and 10kΩ resistors to regulate the output voltage to 12 volts, using a design equation in the LMR16020 datasheet.

In order to verify that our design was correct, we performed experiments on the 12V buck power supply. Data that we recorded indicates that we have an average output of 12.1 volts, which is tolerably close to the designed 12V.

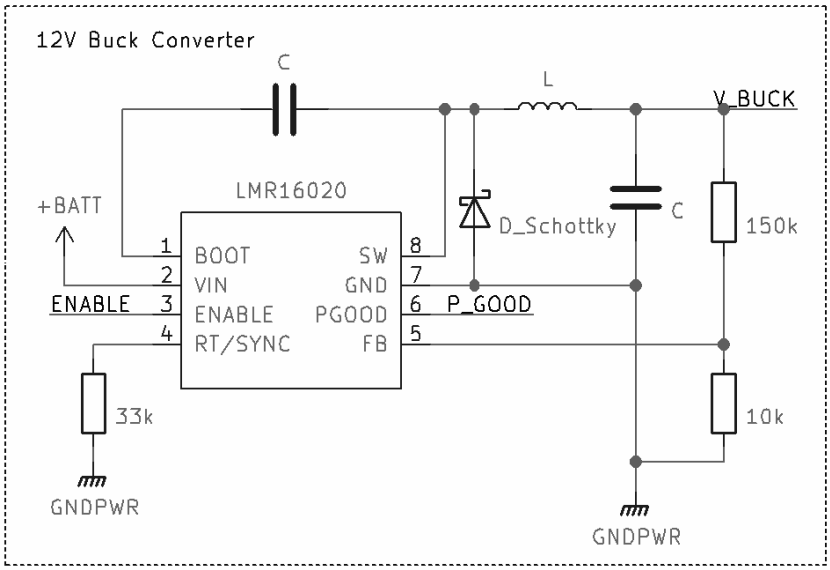



Figure 21: The buck converter supplies +12V to the gate driver Ics

**Test Setup**

In order to load the motor controller with an attached motor, we repurposed an existing motor test stand from a previous year's senior design team. Figure 22 shows our modified version of the test stand. Previously, the test stand had mounting brackets for two Maxon 200W brushed DC motors, so we designed and fabricated our own aluminum bracket for the brushless DC motors we were using. The test stand allows two motors to be mechanically coupled: one motor to operate as an electrically-driven motor and the other to operate as a generator-dynamometer load.

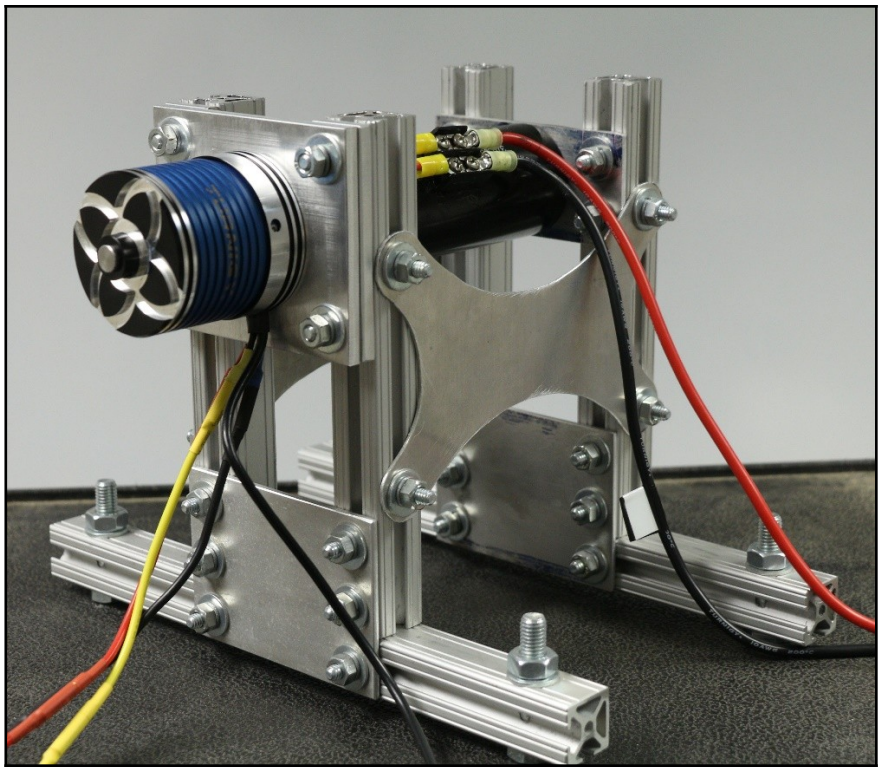


Figure 22: Brushless and brushed DC motor mounted securely on our test stand.

**Software**

The software portion of this project can be divided into three aspects. First, initialization of peripherals on the STM32 microcontroller. These include the analog to digital converter CAN, and SPI subsystems. Second, an abstraction layer for setting up and controlling the TMC4671. Finally, the menu system for displaying and interpreting commands from the serial terminal.

**Serial Peripheral Interface**

As was mentioned earlier the STM32 and TMC4671 communicate over an SPI bus. The ability for these two devices to communicate was first tested with a TMC4671 breakout board and an STM32 Nucleo evaluation kit (Figure 23). Then, after the PCB was designed and constructed testing moved to the completed PCB.

Before the STM32 and TMC4671 could communicate the proper SPI mode had to be correctly setup. The basic SPI settings were initialized with the STM32CubeMX tool (Figure 12); however several incorrect assumptions were made that prevented the communication from working initially. The first of

these issues arose from an invalid assumption that chip select (CS) pin could be tied low and the SPI bus would still function; however, this was not the case. After the CS pin was connected to the STM32 and toggled at the appropriate time, the TMC4671 started responding to SPI commands. Next the SPI mode had to be found. While the chip was providing signals back to the STM32, they were not the expected result from the command being sent. After some debugging, it was found that the SPI mode needed to be changed to have SPI clock idle high, and read the SPI data on the second clock edge. Once these changes were made to the code, all communication worked properly even on the final PCB.

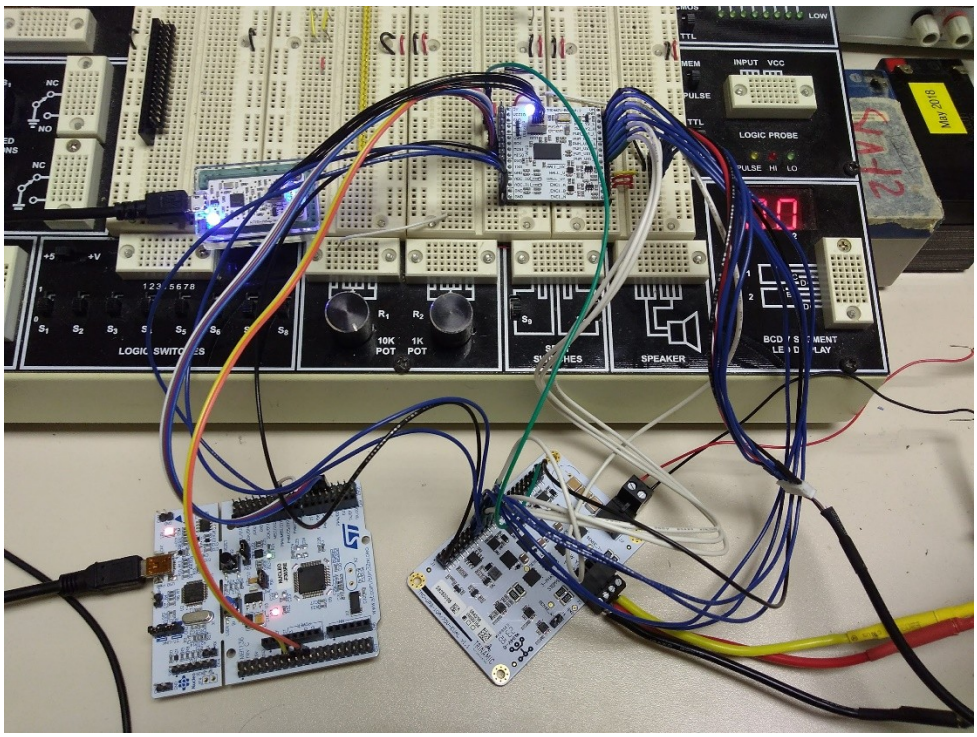



Figure 23: Initial TMC4671 Testing setup

**Analog to Digital Conversion**

Figure 24 shows a flow diagram for the ADC code. The software uses the ADC's sequence conversion feature and its timer trigger feature for a consistent sample period and minimal processor overhead. The main process initializes the microcontroller including the ADC and the timer (Timer 6). The timer is setup to be refreshed approximately every 667 microseconds for a sample rate of 1.5 KHz (which matches the filters described above), the refresh triggers an ADC conversion sequence. Each conversion sequence digitizes a set of the used ADC channels; in this case, ADC1 converts the two

temperature sensors (motor temperature and transistor temperature), and ADC2 converts the throttle and two user analog inputs. After each conversion occurs, an interrupt is fired which stores the converted value into a global variable which can be accessed by the main loop.

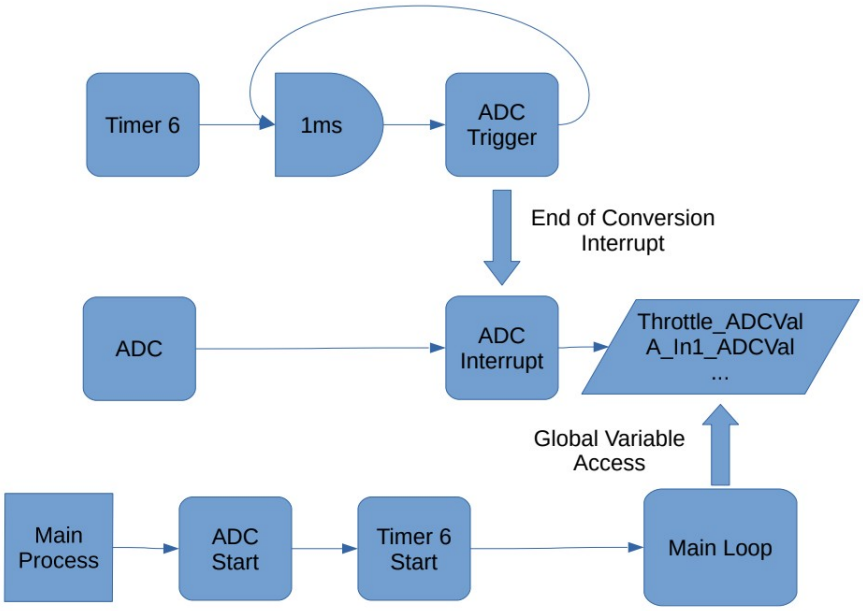



Figure 24: ADC Conversion Flow Chart

Every time an ADC conversion is complete the function `HAL_ADC_ConvCpltCallback()` is called. The following code snippet shows the code for this callback. The code begins by defining some static variables and initializing them to zero. Static variables are scope-limited global variables; they maintain their value across function calls, but cannot be accessed outside of their function. These variables track which ADC channel was converted. The initialization at the beginning of the function only occurs on startup, not every time the function is called. Next, a bit flag is checked to see if it was the end of a conversion sequence, and the ADC value is read from the hardware. The flag is read before the value from the ADC is read because the accessing of the conversion register clears the status flag.

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
  static int8_t adc1_conv_count = 0;
  static int8_t adc2_conv_count = 0;

  // check if the sequence is over with
  bool end_of_sequence = __HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOS);

  // read the ADC value (this will clear the end of sequence flag)
  auto ADCValue = static_cast<uint16_t>(HAL_ADC_GetValue(hadc));

  if (hadc->Instance == ADC1)
  {
    // which channel to put the ADC value in
    switch(adc1_conv_count)
    {
      // Motor temperature
      default:
      case 0: MotorTemp_ADCVal = ADCValue; break;

      // Transistor temperature
      case 1: TransistorTemp_ADCVal = ADCValue; break;
    }
    adc1_conv_count++;

    // go back to the beginning of the sequence
    if (end_of_sequence) adc1_conv_count = 0;
  }
  else if (hadc->Instance == ADC2)
  {
    // which channel to put the ADC value in
    switch(adc2_conv_count)
    {
      //throtle reading
      default:
      case 0: Throttle_ADCVal = ADCValue; break;

      // A_In1 reading
      case 1: A_In1_ADCVal = ADCValue; break;

      // A_In2 reading
      case 2: A_In2_ADCVal = ADCValue; break;
    }
    adc2_conv_count++;

    // go back to the beginning of the sequence
    if (end_of_sequence) adc2_conv_count = 0;
  }
}
```

Next the ADC value is put into the appropriate global variable by checking which ADC called the function and the conversion count variable. The values can now be used elsewhere in the code to see the current value of each of the analog pins.

**CAN Interface**

It is outside the scope of this design report to describe the operation of CAN at a basic level, so only the important features will be discussed. Once again, I used the stm32CubeMX software to initialize the pins and base CAN hardware. The prescalar and timing values were selected for 500 Kbs operation.

Instead of directly using the STM32 provided functions, I am using a C++ wrapper library for CAN which was written for the Supermileage team a few years ago. This class simplifies the transmission and reception of messages as it provides functions for packing and unpacking commonly used datatypes (uint8, uint16, etc). It also calls user-defined callback functions on the reception of messages.

For the motor controller to meet the design requirements there were a number of different messages that needed to be sent out over CAN. Table 2 lists the messages and IDs used by the motor controller. Note that the base ID and the throttle ID can be set by the user from the motor controller menu.

A note about the formatting of the CAN messages: the first byte of each CAN message contains formatting information (from the C++ CAN library). In most cases this can be safely ignored and the data extracted from the remaining bytes of the message. Table 1 shows the byte-level arrangements of the message types used by the motor controller. The Int16 Arr data format can send between 1 and 3 integers, if the data to be sent is a 2 integer array, then the 3rd integer will not be sent in the message (replace with XX).

Table 1: CAN Message Formats (XX is an unused byte and will not be sent)

| Message Format | CAN Data Bytes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Uint8 | 0x00 | data(7:0) | XX | XX | XX | XX | XX | XX |
| Int32 | 0xA0 | data(7:0) | data(15:8) | data(23:16) | data(31:24) | XX | XX | XX |
| Uint32 | 0x80 | data(7:0) | data(15:8) | data(23:16) | data(31:24) | XX | XX | XX |
| Float | 0xC0 | data(7:0) | data(15:8) | data(23:16) | data(31:24) | XX | XX | XX |
| Int16 Arr | 0x60 | data0(7:0) | data0(15:8) | data1(7:0) | data1(15:8) | data2(7:0) | data2(15:8) | XX |

Table 2: Motor Controller CAN IDs (Transmit is from the motor controller)

| Name | ID | Format | Direction | Description |
|---|---|---|---|---|
| Base ID | Base ID | RTR | Receive | Send out current motor controller data on RTR request. |
| Temperature | Base ID + 1 | Int16 array | Transmit | Sends the motor and transistor temperatures in a single CAN message (in that order) |
| Motor RPM | Base ID + 2 | Int32 | Transmit | Sends the motor RPM |
| Motor Current | Base ID + 3 | Float | Transmit | Sends the motor current (in amps) |
| Battery Voltage | Base ID + 4 | Float | Transmit | Sends the battery voltage (in volts) |
| Battery Current | Base ID + 5 | Float | Transmit | Sends the battery current (in amps) |

| | | | | |
|---|---|---|---|---|
| Motor Direction | Base ID + 6 | Uint8 | Receive | Listens for the motor direction.<br>0: Forward<br>1: Reverse |
| Control Mode | Base ID + 7 | Uint8 | Receive | Listens for the control mode.<br>0: Torque (current)<br>1: Velocity (RPM) |
| Max Current | Base ID + 8 | Uint32 | Receive | Listens for the maximum current (in mA) |
| Max RPM | Base ID + 9 | Uint32 | Receive | Listens for the maximum RPM<br>(only effective in velocity mode) |
| Max Acceleration | Base ID + 10 | Uint32 | Receive | Listens for the maximum acceleration<br>(Not implemented, use the maximum current to limit acceleration) |
| Enable Pulse | Base ID + 11 | Any Data | Receive | Listens for any valid data. A signal must be received every 500ms to keep the motor controller active in CAN mode. |
| Setpoint | Throttle ID | Uint32 | Receive | Listen for a setpoint.<br>Torque mode: Motor current in mA<br>Velocity mode: Motor RPM |

The “Enable Pulse” CAN ID is a safety feature we implemented for the CAN mode of operation. It is used to disable the motor controller, if for some reason during a race the CAN-based throttle was disconnected from the motor controller or the CAN bus is compromised. Because the motor controller will maintain a setpoint until it receives another command, there needed to be a method to turn off motor power if the CAN bus goes into an error state. We thought that the best method to determine if the CAN bus is still functioning properly would be to use an ID that sends out a pulse (some piece of data) at least once every 500ms. If the motor controller can still “hear” this ID, it means that the bus is still working properly, and the setpoint is still valid. If for some reason the motor controller cannot hear this ID then power to the motor is shut off. Another option considered would be to keep the motor running as long as any CAN message was received once every 500ms. We decided not to use this option as we thought using a dedicated ID would be easier to debug if problems occurred, and gives the team more flexibility to disable the motor controller by stopping the pulse transmission.

**C++ Abstraction Classes**

After initializing all of the peripherals to operate the motor controller, the actual work of communicating with the TMC4671 must be accomplished, along with managing all of the user

configurable settings which are able to be set through the menu system. These tasks are accomplished through a series of C++ classes which abstract some of the more complex functions into more manageable pieces. The main classes which do this are the `TMC4671Interface` class, the `SettingsManager` class, the `ComputerInterface,` and the `ComputerMenu` class. The menu class is used by the computer interface class to generate a menu, but is not seen by the main application.

**TMC4671 Interface Class**

The TMC4671 interface class takes a C application programing interface (API) provided by Trinamic and converts it into a C++ style class structure and limits some of the options to make it more user friendly. It also uses the same settings structure as the `SettingsManager` class so communication between the various parts of the code is easier. The following code snippet provides the functions defined by the class for external use. Do not be alarmed by the lack of comments in the code, they were removed for the sake of brevity, but remain in the actual code.

```
class TMC4671Interface {

public:
    TMC4671Interface(TMC4671Settings_t *settings);
    void change_settings(const TMC4671Settings_t *settings);
    void set_control_mode(ControlMode_t mode);
    void set_direction(MotorDirection_t dir);
    void set_setpoint(std::uint32_t set_point);
    void enable();
    void disable();
    float get_motor_current();
    float get_battery_current();
    float get_battery_voltage();
    std::int32_t get_motor_RPM();
   //...  private constants
};
```

This class takes in a TMC4671Settings_t structure, which contains all of the pertinent settings needed to initialize the motor controller. At any time, the user can call the `change_settings()` function to change one of the settings not exposed by its own function (such as setpoint and direction). This function is called by the computerInterface class when a user prompts a change in one of these settings. The following code provides a list of the settings that can be changed through this function.

```
struct TMC4671Settings_t {
    MotorDirection_t    MotorDir;
    ControlMode_t       ControlMode;
    std::int32_t        Setpoint;

    std::uint32_t       CurrentLimit;
    std::uint32_t       VelocityLimit;
    std::uint32_t       AccelerationLimit;

    MotorType_t         MotorType;
    std::uint8_t        PolePairs_KV;
    struct {
        std::uint8_t    HallPolarity    : 1;
        std::uint8_t    HallInterpolate : 1;
        std::uint8_t    HallDirection   : 1;
    } HallMode;
    std::int16_t        HallMechOffset;
    std::int16_t        HallElecOffset;

    // PID values
    std::uint16_t       FluxP;
    std::uint16_t       FluxI;
    std::uint16_t       TorqueP;
    std::uint16_t       TorqueI;
    std::uint16_t       VelocityP;
    std::uint16_t       VelocityI;

    std::uint16_t       OpenAccel;
    std::uint16_t       OpenVel;
    std::uint32_t       OpenMaxI;
    std::uint16_t       OpenMaxV;
};
```

**Settings Manager Class**

There are a few settings for the motor controller that are not part of the TMC4671 chip. Also, all of the motor controller settings need to be saved into a non-volatile location so they can be used to initialize the controller during startup. The extra settings, along with the settings for the TMC4671 are placed in a larger structure. The code for these structures are listed below.

```
struct GeneralSettings_t {
    std::uint16_t ControllerCanId;
    std::uint16_t ThrottleCanId;
    range_t ThrottleRange;
```

```
    struct {
        std::uint8_t useAnalog     : 1;
        std::uint8_t enableOutputs : 1;
    } bool_settings;
};

struct MotorControllerValues_t {
    GeneralSettings_t General;
    TMC4671Settings_t tmc4671;
};
```

As can be seen from the code, the additional settings control the base CAN id (ControllerCanId variable), the Throttle CAN id, whether the controller should use the analog throttle input or the CAN setpoint, and whether the outputs should be enabled.

The settings manager class takes in the overall settings structure (MotorControllerValues_t). On startup the class checks whether the settings stored in flash are invalid (all 0xFF, the value of erased flash memory) then the manager saves values of the settings structure stored in RAM to the flash copy. This allows the values of the settings to be put into a known-good after the first programming of the microcontroller. If there are valid settings in the flash structure, they are loaded into the RAM structure and used to setup the TMC4671. This class also implements functions that set the various motor controller parameters by passing in a value and an enumeration type that identifies the structure member, and a function that returns the value of the parameter as a string. These functions are used by the computer interface class to handle menu requests.

**Computer Interface Class**

The computer interface class provides an abstraction for communicating with the computer. It manages the menu system (using the ComputerMenu class) and transmits and receives data using the USB to serial code provided by ST Microelectronics. Last, the class manages calls from the menu system to read and write settings.

In order for settings to be changed, user input is required. This is also accomplished over the serial terminal. The USB to serial conversion code provided uses a callback function to transfer data from the USB buffer to the user. This data buffer must be parsed by the microcontroller in order to validate and

use the information. However, because the callback function is written in pure C code, the computer interface class implements a C wrapper function, which actually calls the class buffer-update code. This wrapper function is the one actually called in the callback.

To handle user input well, the user interface needs to be able to handle backspaces gracefully (along with rejecting invalid characters). This task is accomplished using a state machine that runs over the command buffer whenever a new buffer is loaded. Figure 25 shows the state machine used for updating the buffer. As can be seen, when a backspace character is encountered the command buffer pointer is decremented so that any further characters will overwrite the “bad” character. The escape key or ‘u’ are used to go up in the menu system so they are copied into the command buffer. Newline or carriage return characters are used to indicate the end of a command for processing by the parser. The only other characters allowed into the buffer are numbers 0-9, or the - character, but only if it is the first character in the command buffer. Allowing the negative sign allows for negative numbers, but forcing it to be the first character prevents malformed input. Other edge cases such as backspacing out of the buffer are dealt with as well, but not shown for clarity. The other edge case worth noting is when the command buffer is filled. The solution used in this project is that when the buffer is full the first new character overwrites the last position in the command buffer. This ensures that the menu is always responsive to user input.

Now that the buffer contains a useable string, it needs to be parsed into an integer, or as command to the menu system. This is accomplished by another state machine (shown in Figure 26). This function goes through every character in the buffer; if it hits a newline character it triggers conversion of the buffer to an integer. If the Escape key or ‘u’ key are found it triggers the output of the UP_LEVEL command to go up one level in the menu system. If none of these conditions are met, then the function outputs the KEEP_MENU command which tells the menu system to do nothing. This parser function is run every time the main function asks to redraw the menu, which happens about every 0.2 seconds.

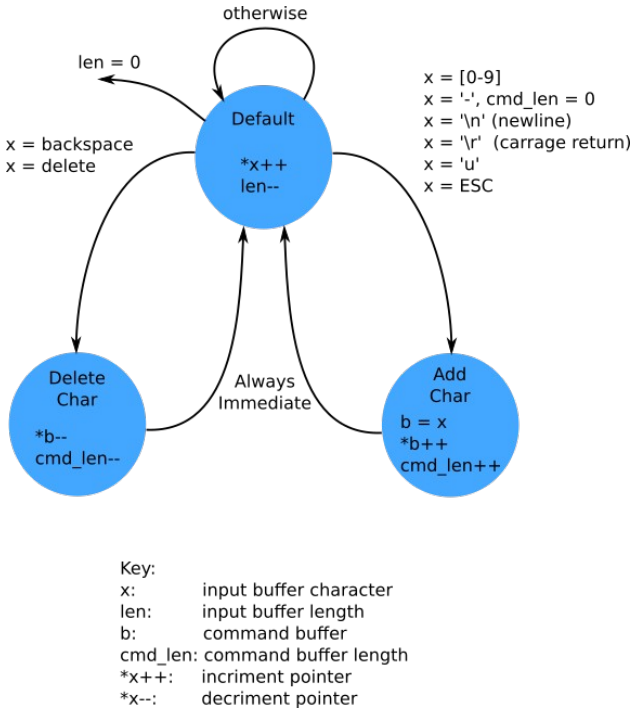



Figure 25: Buffer Update State Machine Graph

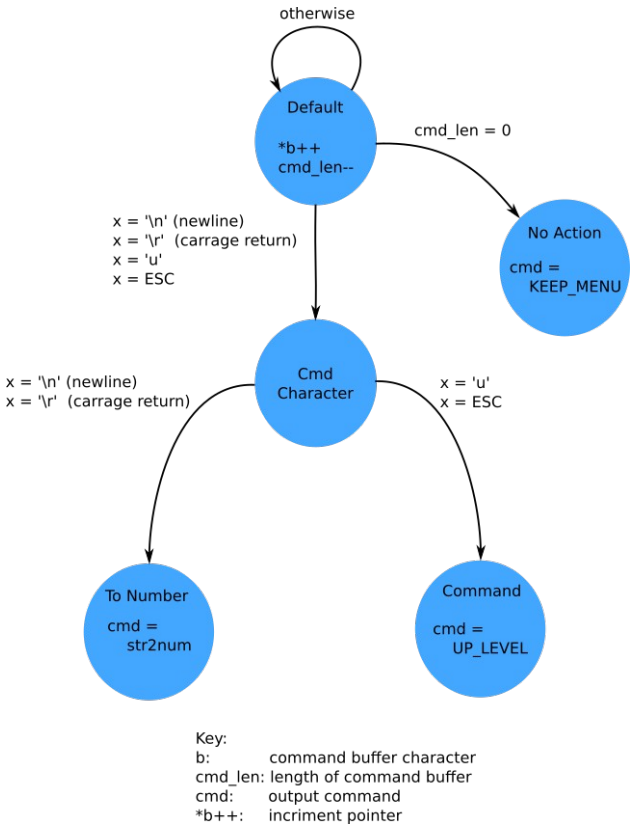



Figure 26: Command Parser State Machine Graph

The computer interface class also allows for the menu to modify and read the motor controller settings through a function, access_setting_value(). This function allows the menu to either write the parameter specified with a int32 variable, or read the parameter out as a string.

**Menu System**

It was determined this semester that producing a working GUI application would be unfeasible due to the time remaining in the project; therefore an alternate solution was pursued. We decided to implement a simple menu system on the microcontroller itself which communicates to a host PC via a USB to serial connection. For all of my testing I have used the open-source, cross-platform, terminal program PuTTY (Figure 27). This interface attempts to meet all of the design goals originally outlined for the GUI program.

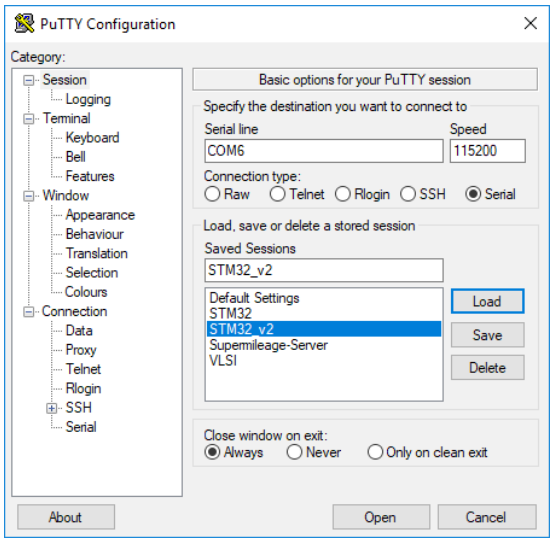



Figure 27: PuTTY Serial Terminal Interface

Switching to this design offered a few key advantages over the original GUI design. The first advantage is development time. Producing a well-constructed GUI program requires a large amount of time and a complex development environment that would need to be setup. This GUI would also become outdated quickly if future teams decided to improve our design as there would be two separate applications to maintain (the microcontroller code and the GUI code) whereas the serial terminal program is embedded into the microcontroller code. Last, having the interface built into the microcontroller makes the design more robust, as the GUI program might become incompatible with the PC as operating systems

and graphics libraries are updated. In contrast, we are reasonably confident that all operating systems will continue to support some kind of serial interface program. Now that the reason for the design switch has been made, we will examine the design and implementation of it.

The overall communication to the computer is handled by the ComputerInterface class. The menu is created as a doubly-linked tree structure which is completely constructed at compile time. Each menu item is constructed from a structure which contains pointers to an array of sub nodes, and a pointer to its parent node. Code for the MenuItem structure is listed below.

```
class MenuItem {
public:
  const char* name_str;
  const char* menu_description;
  MotorControllerParameter_t param;

  const MenuItem *sub_menu;
  const MenuItem *parent_menu;
  int sub_menu_items;
};
```

As you can see, in addition to the pointers, each menu item contains a pointer to its name string, a description string, and an enumeration parameter that defines the command that the menu item controls. The following code snippet shows how the menu is initialized. The `main_menu_items` variable is a statically-allocated array of MenuItems which is populated using an initializer list. The following snippet only initializes the main menu; the other sub-menus are constructed in a similar fashion. The displayed menu is shown in Figure 28.

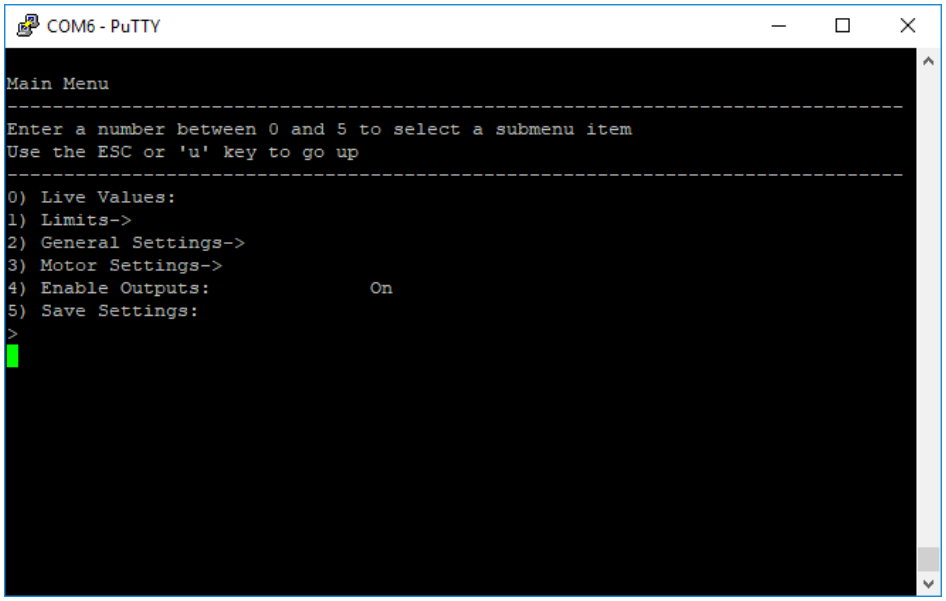



Figure 28: Motor Controller Main Menu

```
using mc_param = MotorControllerParameter_t;
auto default_param = mc_param::NO_ACTION;

main_menu = MenuItem{"Main Menu", "",
                     default_param,
                     main_menu_items.data(), nullptr,
                     main_menu_items.size()};

main_menu_items[0] = MenuItem{"Live Values", "Shows realtime values",
                        mc_param::LIVE_VALUES,
                        nullptr, &main_menu,
                        0};

main_menu_items[1] = MenuItem{"Limits", "",
                              default_param,
                              limits_menu_items.data(), &main_menu,
                              limits_menu_items.size()};

main_menu_items[2] = MenuItem{"General Settings", "",
                              default_param,
                              general_setting_items.data(), &main_menu,
                              general_setting_items.size()};

main_menu_items[3] = MenuItem{"Motor Settings", "",
                              default_param,
                              motor_setting_items.data(), &main_menu,
                              motor_setting_items.size()};

main_menu_items[4] = MenuItem{"Save Settings",
      "Save settings to flash\n\r 0 to exit, 1 to save current
settings",
      mc_param::SAVE_SETTINGS,
```

```
nullptr, &main_menu,
0};
```

**PCB Temperature Under Load**

The power PCB is intended to carry a power of 1125W for a limited time of vehicle acceleration – on the DC side, this corresponds to an average current of 24.5A for a voltage of 46V. Due to the thin PCB traces, we designed our 4-layer power PCB to utilize at least two layers for high-current traces where possible. However, we still had concerns about whether the power PCB could handle the required current. Thus, one test we performed was to connect a lab DC power supply to supply a current over time and measure temperature of the PCB traces using an IR no-contact thermometer. This process does not yield as precise results as a thermal imaging camera would, but we were able to move the temperature sensor along each trace to obtain an average temperature reading. As seen in Figure 29, a maximum temperature of 77.5 °F is achieved for the Phase 2 trace with 6A of current. This current is definitely not the maximum current that the PCB will see, but lower temperatures in this test indicate a good trend for higher powers flowing through our PCB. Later qualitative observations of PCB temperature while in operation under load indicate that we won't have an issue with PCB overheating.

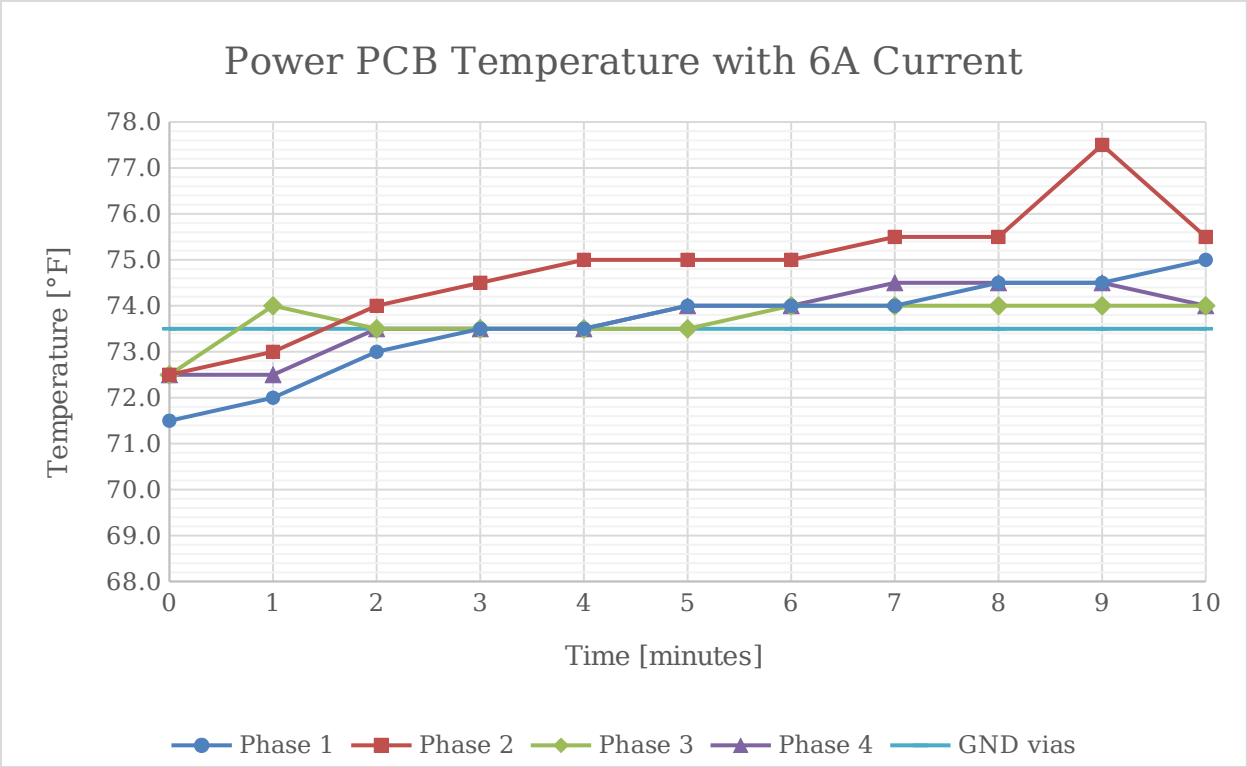



Figure 29: PCB temperature doesn't increase significantly with 6A of current

**Power Capacity**

Using the motor test stand described in the Engineering Design section above, we have been able to collect data on our controller in operation under load. Because the desired application of our controller-motor combination is to propel a Supermileage vehicle, the ideal loading would be high torque at low rotor speeds (for accelerating the vehicle from rest) and lower torque at high rotor speeds (for vehicle cruising). Due to limited resources however, we used a constant-resistance load on the brushed DC motor. This does not allow either constant-torque or constant-power loading for a range of motor speeds; load current (torque) is directly proportional to rotor speed because the brushed DC motor generates a voltage corresponding to speed.

Recording DC battery voltage and current in addition to three-phase voltage and current (using the two-wattmeter method), we are able to monitor power through our controller and calculate efficiency. In order to measure the seven quantities required, we synchronized two 4-channnel lab oscilloscopes with a trigger line and recorded battery voltage and current, three line-to-ground PWM motor voltages, and two motor AC currents. Using MATLAB for data processing, we can calculate average power and efficiency.

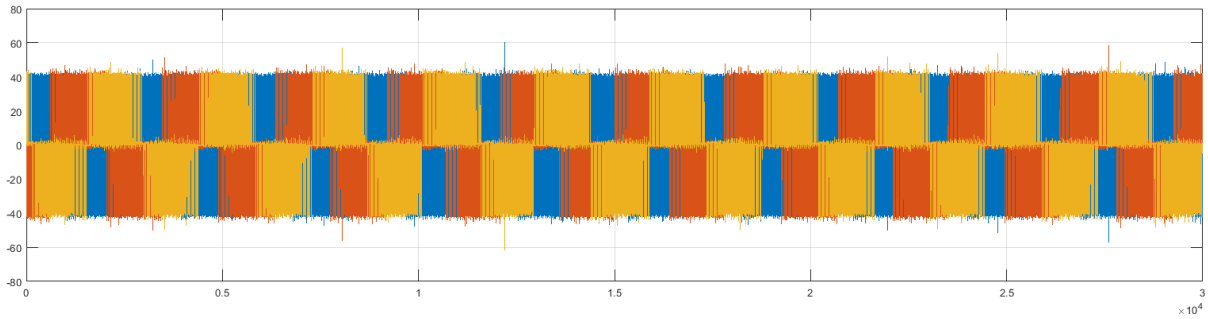



Figure 30: Three-phase line-to-line PWM motor voltages are offset 120°.

Using the ACS723 current sense test points on our power PCB, we can measure phase currents for both phase A and phase B. Again with MATLAB, we calculate the third phase current knowing that the sum of all three phase currents must equal 0. Figure 31 shows the plot of these three currents for one motor operating point; the currents have a magnitude of 40A and approximate the shape of sinusoids.

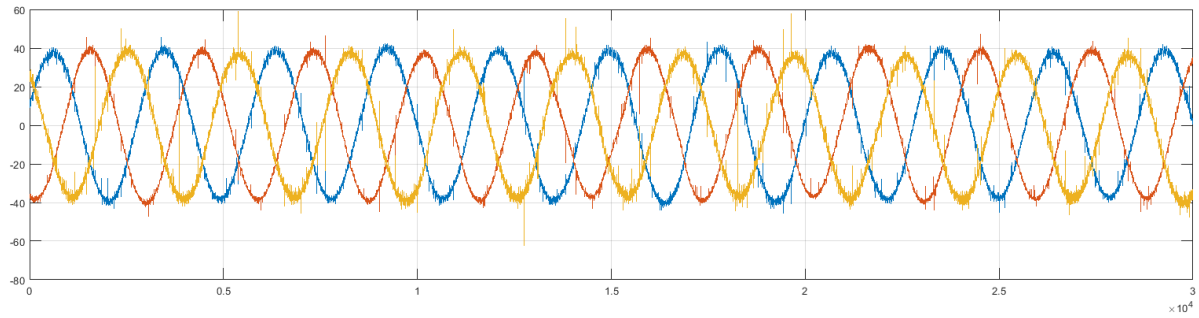



Figure 31: Phase currents add to zero

The two-wattmeter method specifies that three-phase real power consumption is the sum of the time-average power from two wattmeters (ref. Bibliography). Multiplying the instantaneous line-to-line voltages by their corresponding currents yields the instantaneous power, illustrated in Figure 32 below.

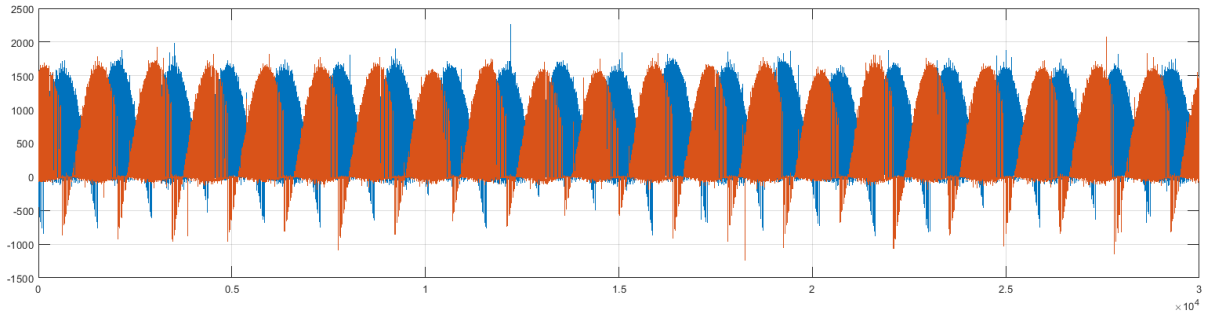



Figure 32: Reactive power of motor causes small negative instantaneous powers.

Calculating the time average of each wattmeter, we are able to sum them up to a total real power of

$$P = W_1 + W_2 = 437.1\,W + 389.2\,W = 826.3\,W$$

Using the equation for reactive power from the two-wattmeter method, we calculate a reactive power of

$$Q = \sqrt{3}\left(W_1 - W_2\right) = \sqrt{3}\left(437.1 - 389.2\right) = 83.0\,VAR$$

These powers correspond to a total complex power magnitude of $830.4\,VA$, using trigonometry on the power triangle. Also, this motor operating point yields a power factor $PF = 0.995$. It should be noted that this operating point is not the maximum power that we can pass through our controller; in the midst of testing we ran into some 12V buck power supply issues that prevented us from higher-power testing.

**Snubber Performance**

In the snubber design process, we captured data for the ringing effect present prior to including snubbers. Figure 33 shows the Phase A no-snubber voltages measured at the positive DC rail, high-side MOSFET gate, bridge middle output to motor, and low-side gate voltage, respectively. Channel 3 is especially of interest, showing the motor output voltage with ringing effects. With a source voltage of 16V average, the voltage peak on the bridge-middle is about 24V – this spike could be fatal for our controller at full DC supply voltage. The purpose of the snubber is to dampen this spiking/ringing and reduce switching stress on the MOSFETs and associated voltage-sensitive components.

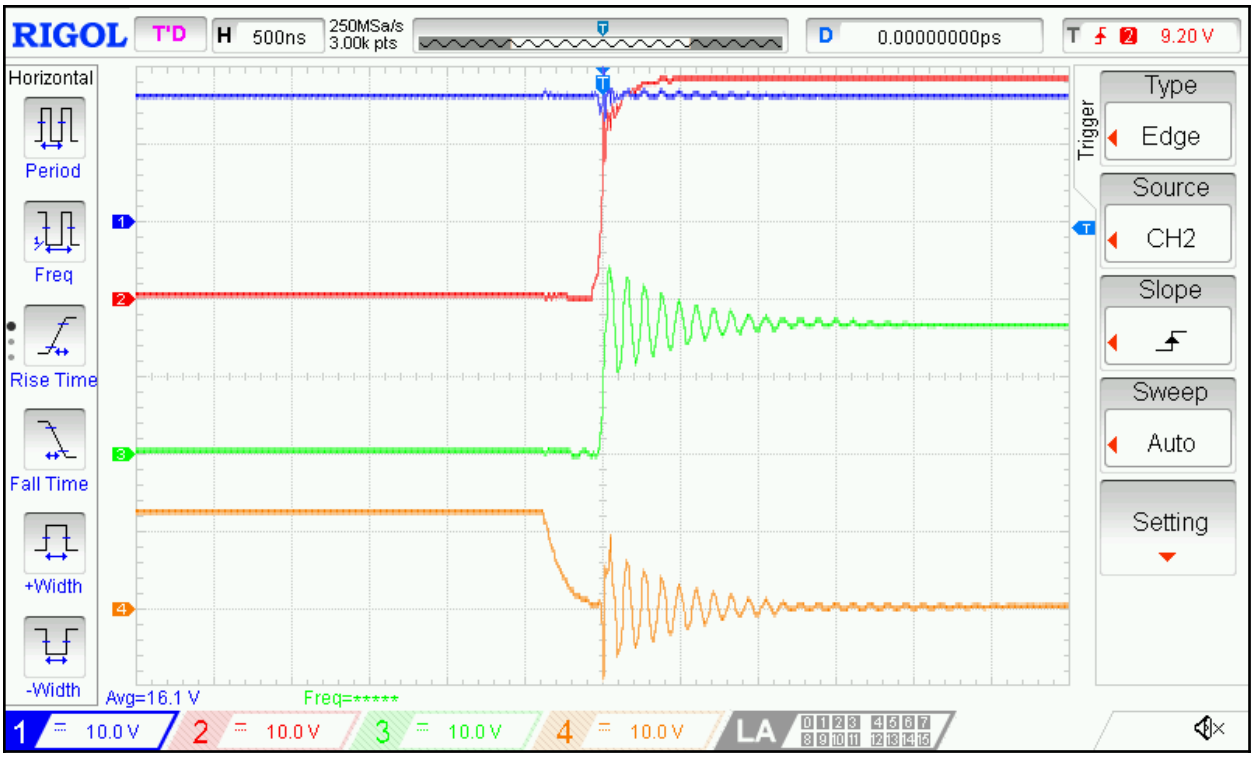



Figure 33: Channel 3 (green) shows H-bridge output ringing as low-side MOSFET turns OFF.

Using resistors and capacitors available in the lab, we empirically selected a combination that yielded the results shown below in Figure 34. With the same DC source voltage of 16V, the bridge-middle voltage still overshoots to a peak of about 23V, which is slightly less than without the snubber present.

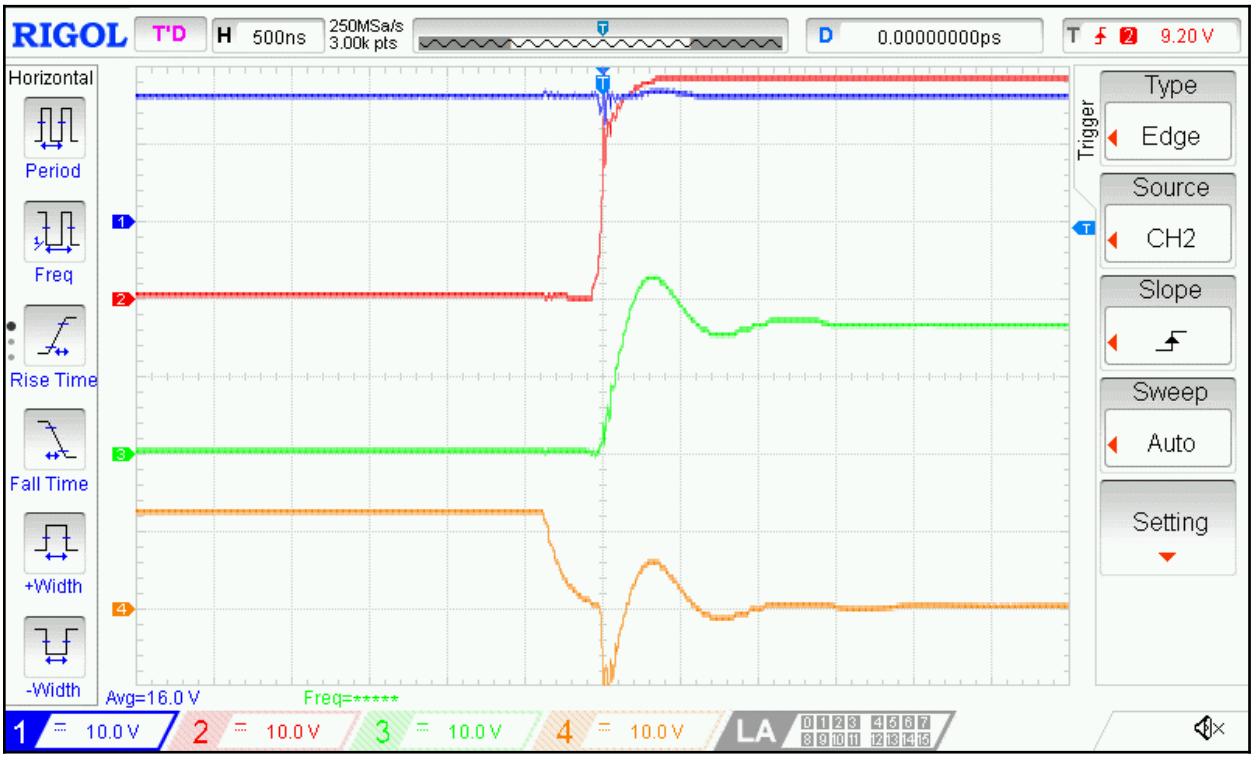



Figure 34: Addition of the R-C snubber decreases ringing.

## Conclusion

This project's objectives address needs of the Cedarville Supermileage team for an electric motor controller. We divided the project into two main subsections (the signal processing and the power conversion), and thus were able to divide up the work between the two team members working on the project. We believe we have adequately reached our project goals for this project by producing a motor controller that can be configured by using a menu interface and that is able to control both brushed and brushless motors efficiently.

## Personal Contributions

**Sam Ellicott**

My contributions to the project so far were to produce the schematics for the signal processing and control PCB, along with the code for the microcontroller. As CEO one of my responsibilities was to write and compile weekly reports for our project advisor (Dr. Brown). These were compiled on Monday morning and submitted by Monday evening. For the first part of the fall semester, most of my time was spent finalizing the project requirements. At the beginning of the semester I had a good general idea on the scope of the project, but the specifics were somewhat less clear. After the project was fully scoped I focused my attention on producing schematics and layout for the control PCB. This has taken most of the rest of this semester.

During the spring semester I assembled and tested the control PCB, then designed and tested the software for the controller. This involved writing libraries to communicate with the TMC4671 and to display the menu interface to the user.

**Isaac Jones**

As I've been tasked with the power circuit board, my work has focused on the MOSFETs, gate drivers, and power board PCB design. I selected a MOSFET that can be used for the power half-bridges and gate driver ICs for our circuit. Because Sam and I are using some lesser-known parts in our KiCad schematic design (Trinamic's Esellbruecke header, the LM5109, etc.), I've assisted with creating custom parts in the schematic editor. I've worked with Dr. Brown, my previous experience in power electronics, and the Trinamic evaluation kit to develop our design for the overall power circuitry.

My role as Chief Financial Officer has included keeping track of the budget and purchases; both Sam and I have been keeping track of our engineering hours cost in a central document.

## Bibliography


All About Circuits: Guide to selecting crystal oscillators and loading capacitors

https://www.allaboutcircuits.com/technical-articles/choosing-the-right-oscillator-for-your-microcontroller/

Trinamic DC motor control application note

https://www.trinamic.com/fileadmin/assets/Support/Appnotes/AN025-DC_Motor_Control_with_TMC4671.pdf

Trinamic TMC4671 Evaluation kit (page contains full schematics)

https://www.trinamic.com/support/eval-kits/details/tmc4671-eval/

TMC4671 Datasheet

https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC4671_datasheet_v1.04.pdf

STM32F303RE Datasheet

https://www.st.com/resource/en/datasheet/stm32f303re.pdf

STM32F3 Reference manual

https://www.st.com/resource/en/reference_manual/dm00043574.pdf

IRFB3306 Datasheet

https://www.infineon.com/dgdl/irfp3306pbf.pdf?fileId=5546d462533600a401535628df311ff2

LM5109B Datasheet

http://www.ti.com/lit/ds/symlink/lm5109b.pdf

ACS723 Datasheet

https://www.allegromicro.com/~/media/Files/Datasheets/ACS723-Datasheet.ashx

LMR16020 Datasheet

http://www.ti.com/lit/ds/symlink/lmr16020.pdf

VESC Motor controller

http://vedder.se/2015/01/vesc-open-source-esc/

Shane Coulton Motor 3-phase motor controller documentation

https://scolton-www.s3.amazonaws.com/motordrive/3phduo.pdf

Basic Two-Wattmeter calculation

https://meettechniek.info/measuring/three-phase-power.html

Two-Wattmeter reactive power calculation

https://circuitglobe.com/two-wattmeter-method-balanced-load-condition.html

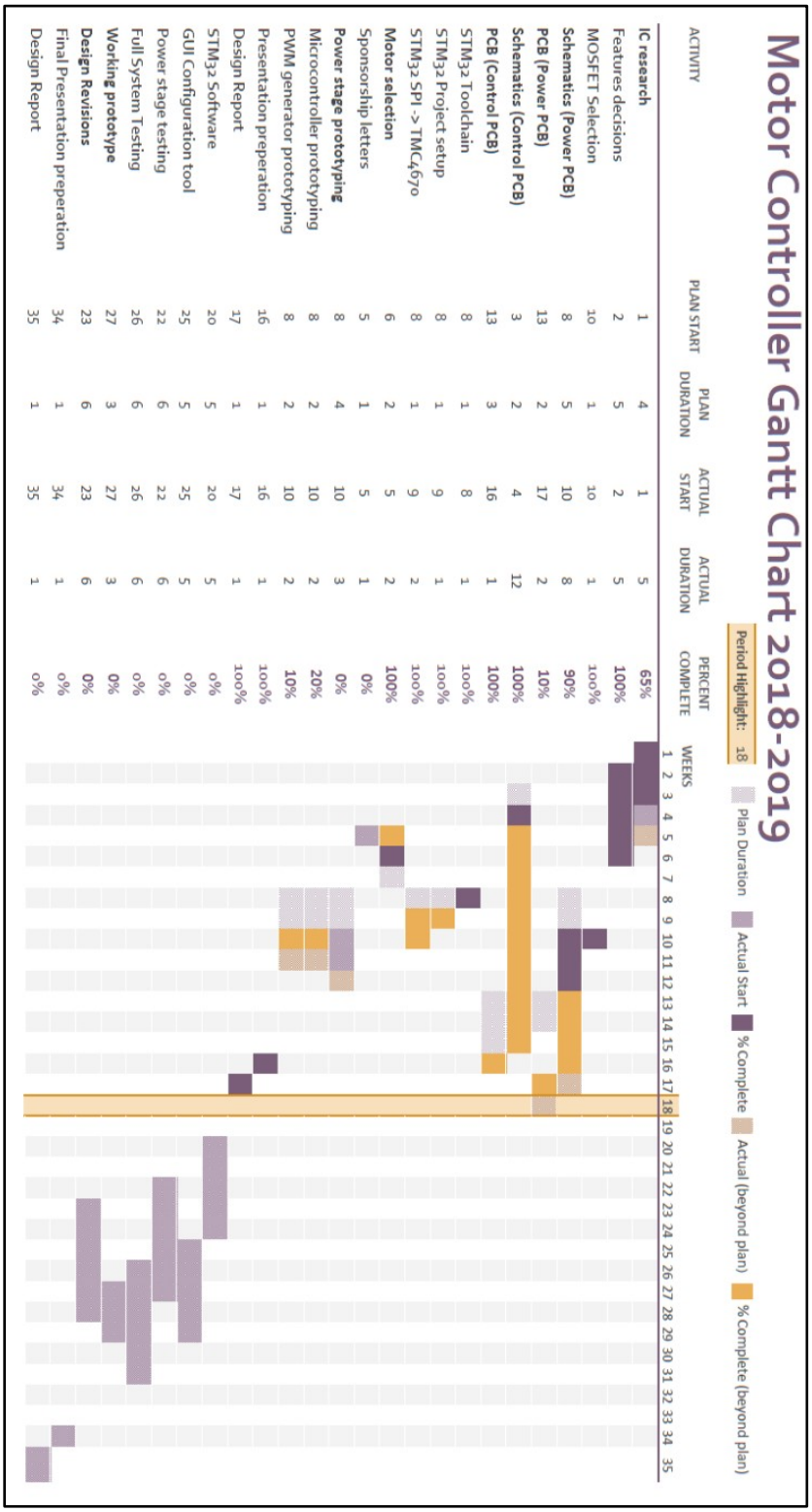
Schematic and PCB source files

https://github.com/HEEV/motor-controller-2018

Signal Processing PCB software

https://github.com/HEEV/motor-controller-2018-sw

## Gantt Chart


Motor Controller Gantt Chart 2018-2019
Period Highlight: 18
Plan Duration
Actual Start
% Complete
Actual (beyond plan)
% Complete (beyond plan)
ACTIVITY
PLAN START
PLAN DURATION
ACTUAL START
ACTUAL DURATION
PERCENT COMPLETE
WEEKS
IC research 1 4 1 5 65%
Features decisions 2 5 2 5 100%
MOSFET Selection 10 1 10 1 100%
Schematics (Power PCB) 8 5 10 8 90%
PCB (Power PCB) 13 2 17 2 10%
Schematics (Control PCB) 3 2 4 12 100%
PCB (Control PCB) 13 3 16 1 100%
STM32 Toolchain 8 1 8 1 100%
STM32 Project setup 8 1 9 1 100%
STM32 SPI -> TMC4670 8 1 9 2 100%
Motor selection 6 2 5 2 100%
Sponsorship letters 5 1 5 1 0%
Power stage prototyping 8 4 10 3 0%
Microcontroller prototyping 8 2 10 2 20%
PWM generator prototyping 8 2 10 2 10%
Presentation preperation 16 1 16 1 100%
Design Report 17 1 17 1 100%
STM32 Software 20 5 20 5 0%
GUI Configuration tool 25 5 25 5 0%
Power stage testing 22 6 22 6 0%
Full System Testing 26 6 26 6 0%
Working prototype 27 3 27 3 0%
Design Revisions 23 6 23 6 0%
Final Presentation preperation 34 1 34 1 0%
Design Report 35 1 35 1 0%

## Schematics

Due to the size and formatting of the schematics, they are attached to the very end of this document, not in this section. If they were attached here, the text would probably be unreadable, and the overall quality would be diminished.

## BOM

Table 3: Project Cost

| Item | Total Qty | Unit Cost (1Ku) | Total Cost | Supplier |
|---|---|---|---|---|
| Rev 1 Signal PCB | 3 | $43.33 | $130.00 | Osh Park |
| Rev 1 Power PCB | 1 | $91.96 | $91.96 | Adv Circuits |
| Rev 1 Parts Order | 1 | $312.02 | $312.02 | Digikey |
| Rev 2 Signal PCB | 10 | $3.10 | $31.00 | JLCPCB |
| Rev 2 Power PCB | 3 | $51.87 | $155.60 | Osh Park |
| Rev 2 Parts Order | 1 | $541.57 | $541.57 | Digikey/Mouser |
| TMC4671 EVAL Kit | 1 | $521.50 | donated | Trinamic |
| RTMI-USB Adapter | 1 | $97.20 | donated | Trinamic |
| Engineering Labor | 589.5 | $50.00 | $29,475.00 | Sam and Isaac |
| Total | | | $30,737.15 | |