Electric Vehicle Research Fall 2024 Report

Dr. Lee Caraway, Derek May, Ale Loynaz Ceballos, Ankit Dhadoti, Elis Karcini, Jacob Lovisone, Jian Wang, Jimmy Sabbides, Matthew Mangsen, Nevaeh Spera, Purnjay Maruur, Shawn Steakley, Wen Wu, and Zexin Huang

Department of Electrical and Computer Engineering
Florida Institute of Technology, 150 W. University Blvd. Melbourne, FL 32901

I. INTRODUCTION

At the beginning of this project, we started essentially from rock-bottom. We were handed down bits and pieces of a plan, and a few hardware odds and ends. We started by determining a primary goal; we needed to drive 6 MOSFETs in a predetermined pattern. While this problem seems quite trivial at first glance, the high potential voltages we would need to control made this much more complex. Over the course of this semester, we have designed a robust control loop, and a nearly complete microcontroller design. We have also created various PCB designs for the gate drivers, and have arrived upon a candidate for a final design.

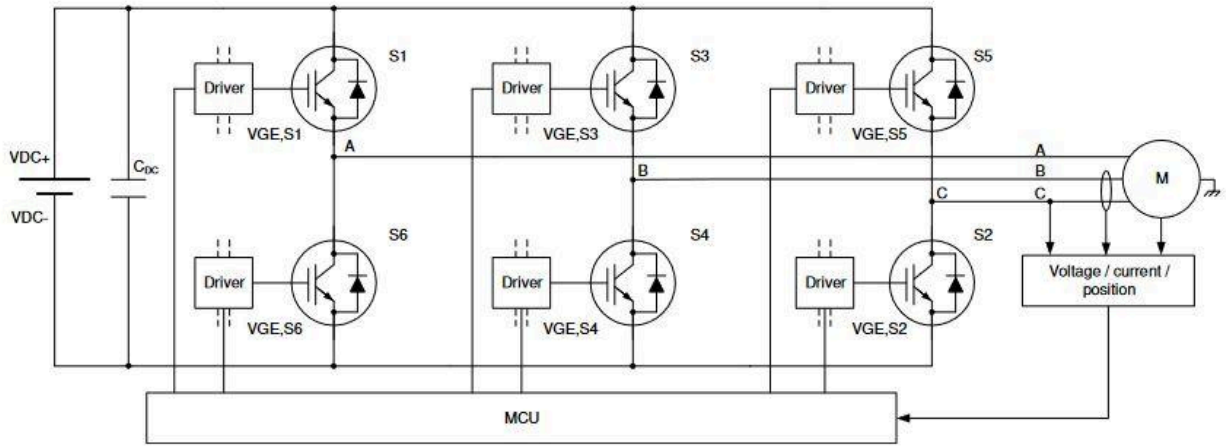


Figure 1: Control layout for a typical traction inverter [8]

II. SUMMER

Over the Summer of 2024, Elis Karcini and Shawn Steakley started evaluating the hardware left for the project from previous years to determine its viability. The focus of these evaluations was the UCC5870-Q1 Gate Driver [5] and the UCC5870QEVM-045 [7] Evaluation Module, a board with six completed gate driver circuits designed to drive a three-phase motor.

The UCC5870-Q1 is a 36-pin chip that operates based on 31 16-bit registers, and communicates using a 4-pin SPI protocol. The chip operates in four different states, RESET, CFG1, CFG2, and ACTIVE, expressed in Figure 2 below. The chip automatically exits the RESET state, moving into CFG1 after performing a series of self-tests so long as the proper power requirements are met. The CFG1 state exists solely to write a chip address to the designated chip, while CFG2 enables write access on all registers for more advanced configuration. The ACTIVE state is the only state in which the gate driver will actually perform as intended.

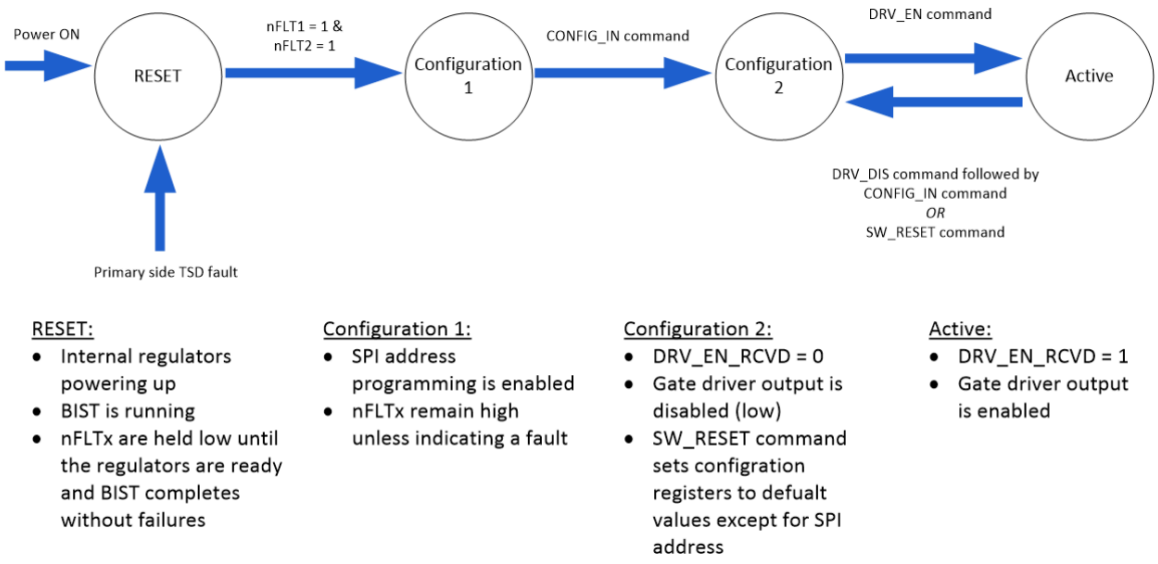


Figure 2: UCC5870 State Diagram [5]

The chip is controlled using a set of 10 commands, pictured below in Figure 3. All commands are divided broadly into two sections, a 4-bit chip address, and 12-bit command. The chip address for all commands will vary depending on the hardware configuration of the chips, with the exception of WR_CA, which will always utilize the broadcast address, as chip selection is handled by the IN+ signal instead.

Table 7-3. SPI message commands

		16-BIT DATA FRAME															
		BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
Command Name	Command Description	CHIP_ADDR				CMD + DATA											
DRV_EN	Driver output enable	CA[3]	CA[2]	CA[1]	CA[0]	0	0	0	0	0	0	0	0	1	0	0	1
DRV_DIS	Driver output disable	CA[3]	CA[2]	CA[1]	CA[0]	0	0	0	0	0	0	0	0	1	0	1	0
RD_DATA	Read data from register address RA[4:0]	CA[3]	CA[2]	CA[1]	CA[0]	0	0	0	1	0	0	0	RA[4]	RA[3]	RA[2]	RA[1]	RA[0]
CFG_IN	Enter configuration state	CA[3]	CA[2]	CA[1]	CA[0]	0	0	1	0	0	0	1	0	0	0	1	0
NOP	No operation	CA[3]	CA[2]	CA[1]	CA[0]	0	1	0	1	0	1	0	0	0	0	1	0
SW_RESET	Software RESET (Reinitialize the configurable registers)	CA[3]	CA[2]	CA[1]	CA[0]	0	1	1	1	0	0	0	0	1	0	0	0
WRH	Write D[15:8] to register RA[4:0]	CA[3]	CA[2]	CA[1]	CA[0]	1	0	1	0	D[15]	D[14]	D[13]	D[12]	D[11]	D[10]	D[9]	D[8]
WRL	Write D[7:0] to register RA[4:0]	CA[3]	CA[2]	CA[1]	CA[0]	1	0	1	1	D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]
WR_RA	Write register address RA[4:0]	CA[3]	CA[2]	CA[1]	CA[0]	1	1	0	0	0	0	0	RA[4]	RA[3]	RA[2]	RA[1]	RA[0]
WR_CA(1)	Write chip address CA[3:0]	1	1	1	1	1	1	0	1	1	0	1	0	CA[3]	CA[2]	CA[1]	CA[0]

(1) IN+ must be high to program CHIP address

Figure 3: UCC5870 SPI Message Commands [5]

Communication with a chip happens utilizing a 4-pin SPI protocol, with the chip acting as the Slave or Child. The datasheet for the UCC5870-Q1 and UCC5870QEVM-045 utilizes Master-Slave nomenclature, which, for the sake of consistency, will be used for the remainder of this report. The data frame for the transmission is below in Figure 4. The protocol operates in an active low, polarity 0, phase 1 configuration, where data is sampled on the falling edge and shifted out on the rising edge of the clock signal, and chip select (nCS) is pulled low during transmission.

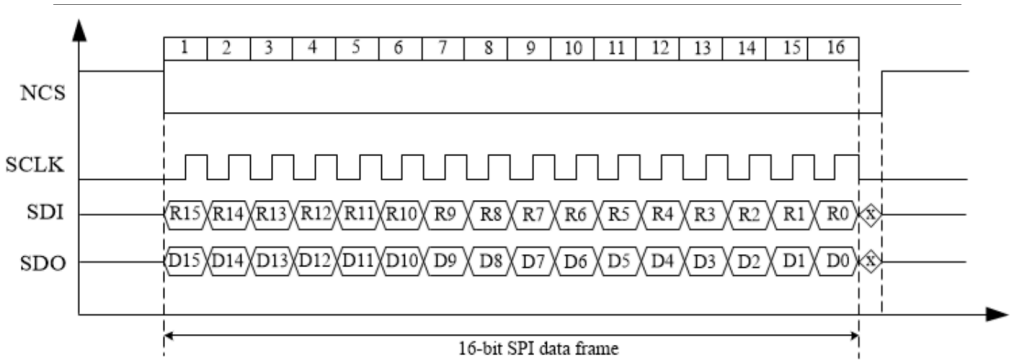
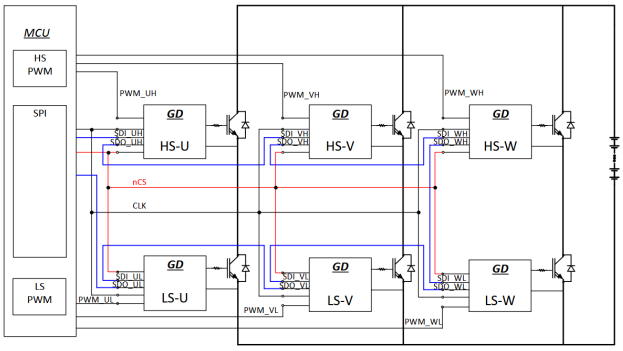


Figure 4: SPI Protocol [5]

The UCC5870QEVM-045 is assembled such that the UCC5870-Q1 chips can be communicated with using a Daisy Chain or Address based format. Address based communication was used during testing due to the relative simplicity and ease of debugging. The hardware configurations for Daisy Chain and Addressed based configuration are seen below in Figures 5 and 6, respectively.



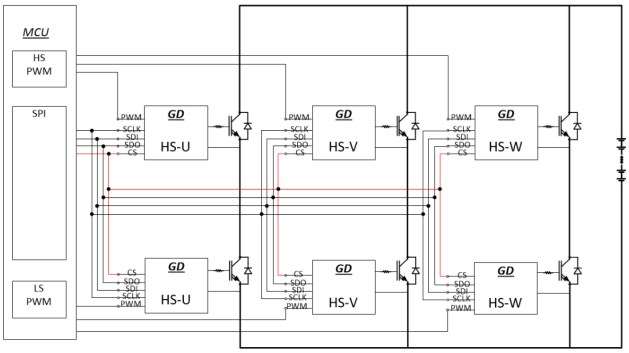


Figure 5 (left): Daisy-Chain SPI diagram for the UCC5870 in a traction inverter [5]

Figure 6 (right): Address-Based SPI diagram for the UCC5870 in a traction inverter [5]

III. BRAKE COIL PCB DESIGN

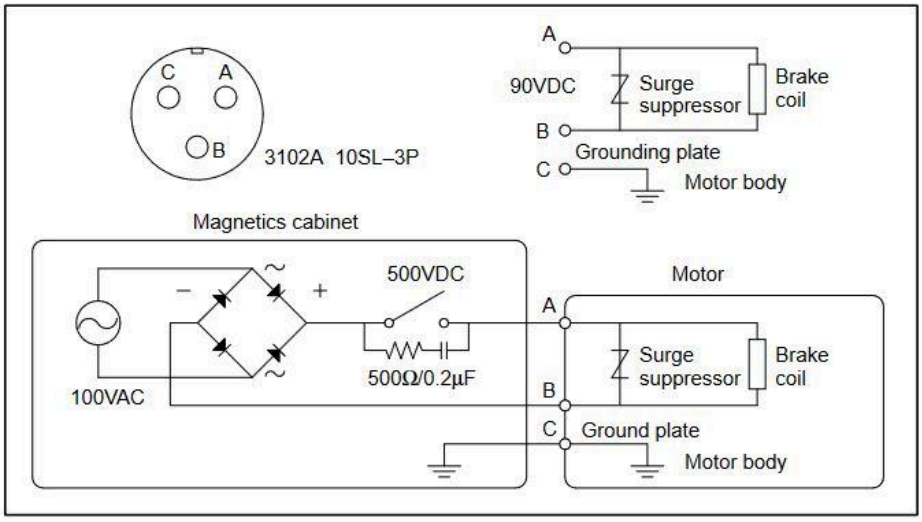


Figure 7: GE's Brake Coil Driver Reference Diagram [13]

For our preliminary testing, we are using a FANUC AC Servo Motor. This is the reference design to release the brake on this particular motor [13]. A previous group determined a set of 4 components that would work, and so we simply analyzed their prototype circuit on perf board and gave it a more polished look by ordering a dedicated PCB for it. Besides this simple redesign, we also 3D printed an enclosure for the PCB for both safety and aesthetics.

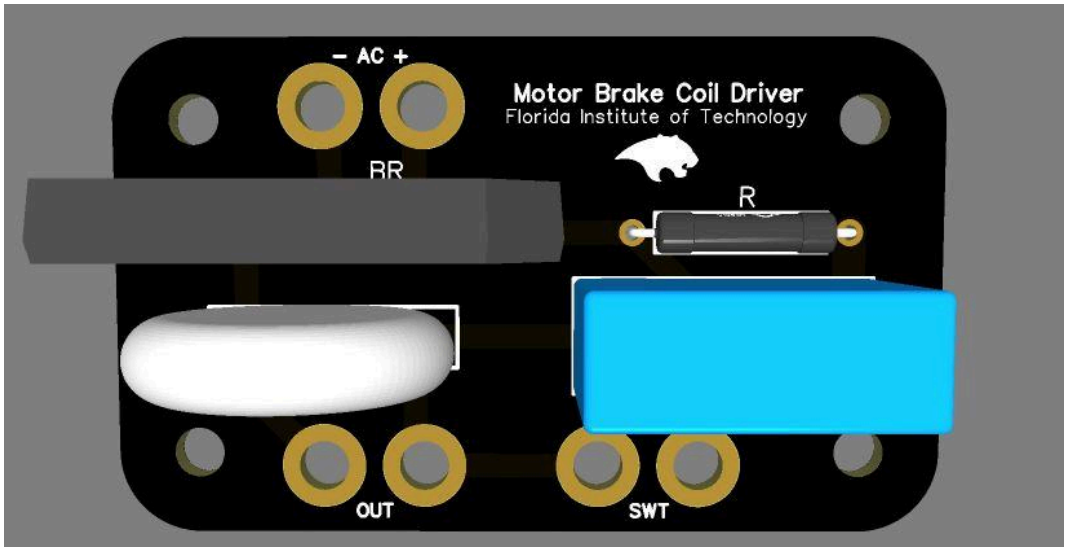
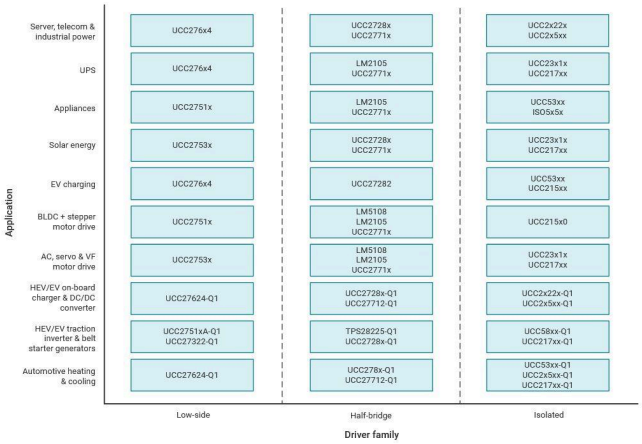


Figure 8: Ordered Brake Coil Driver PCB [5a]

IV. FINDING OUR GATE DRIVER

Our options concerning what gate drivers we could use were nearly limitless. Thus, we decided we had to place some constraints on what we would accept. Firstly, we decided that Texas Instruments would be our supplier; their superior quality and range of selection made them an easy choice. Next, we had to choose a family of gate drivers [1].



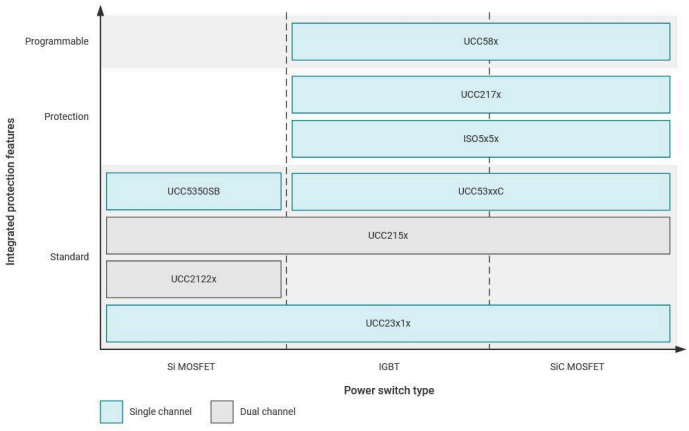


Figure 9 (left): TI Gate Driver Families [1], Figure 10 (right): TI Gate Driver Groups [2]

We could choose from either Low-side, Half-bridge, or Isolated. Since we wanted to separate our microcontroller setup from any type of high power component, we needed to make sure that we picked an isolated gate driver. Out of the isolated gate drivers, we needed to pick a specific group [2].

The two groups we tried picking from were the UCC58xx group, and the UCC215xx group [3]. The aforementioned UCC5870 belongs to the programmable group of gate drivers. The UCC5870-Q1 incorporates a series of safety features designed to comply with the safety standards of the automotive industry and to preserve the integrity of the hardware of the system, which is what initially made the driver so attractive, however testing determined that these

features were unnecessary for our own implementation, and would delay development of the total system significantly in time spent debugging.

Conveniently, a previous group had ordered a development board for the UCC21520 [6], so we went ahead and began testing with it. In the interest of developing a fully functional product, the UCC21520 was selected, as while it lacks many of the features of the UCC5870-Q1, its simplicity and reliability would accelerate the development process significantly. Additionally, the UCC5870QEVM-045 used to test the UCC5870-Q1 chips is an extremely expensive and sensitive piece of hardware, and our limited budget constraints left little room for error when using this module. The supplementary hardware for the UCC21520 proved to be significantly cheaper, allowing for further testing opportunities while limiting risk. Ultimately however, the research performed proved to be useful in optimizing the time spent once the Fall semester began, allowing the team to better focus on our final hardware solution. The improved understanding of the SPI protocol also eased hardware integration for the final system, as it is used for communication between an Arduino and Raspberry Pi.

V. DESIGNING THE PCB

The PCB was initially going to follow the design rules set by the gate driver produced by Texas Instruments, however the design was hard to work with because of the limited documentation on the schematics and the design doc. Furthermore, to deal with this and gain more insight on the design document, the gate driver manual from Texas Instruments were referenced. This resulted in a new gate driver circuit whose circuit size was reduced by 50% from the initial design promoting efficiency and optimization.

The first draft of the circuit was initially copied from the Texas Instruments Gate Driver schematic [6], however, the test points were eliminated as they were not needed and only the necessary things were kept.

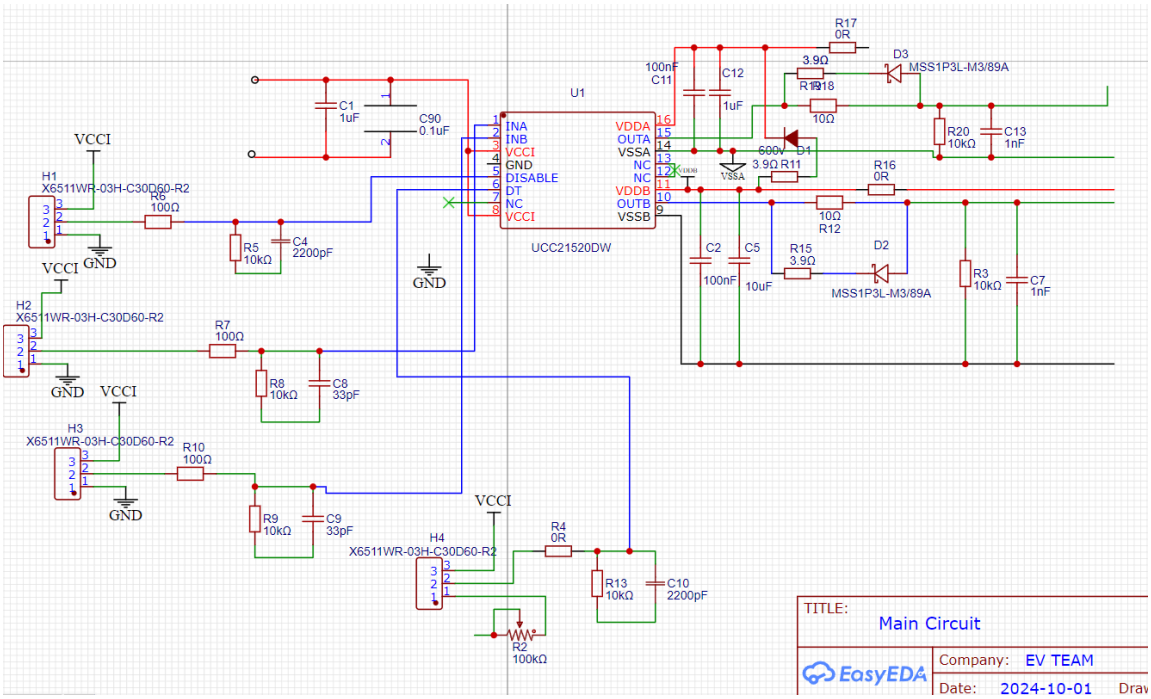


Figure 11: Initial Copy of the Design Document [6]

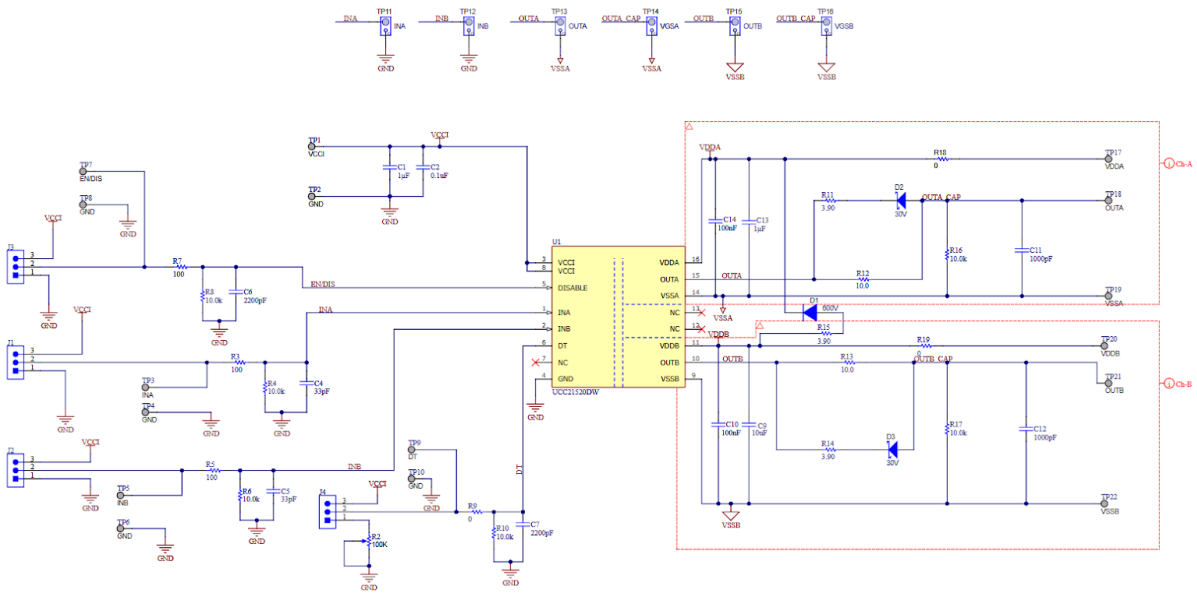


Figure 12: Initial Schematic by Texas Instruments for the Gate Driver [6]

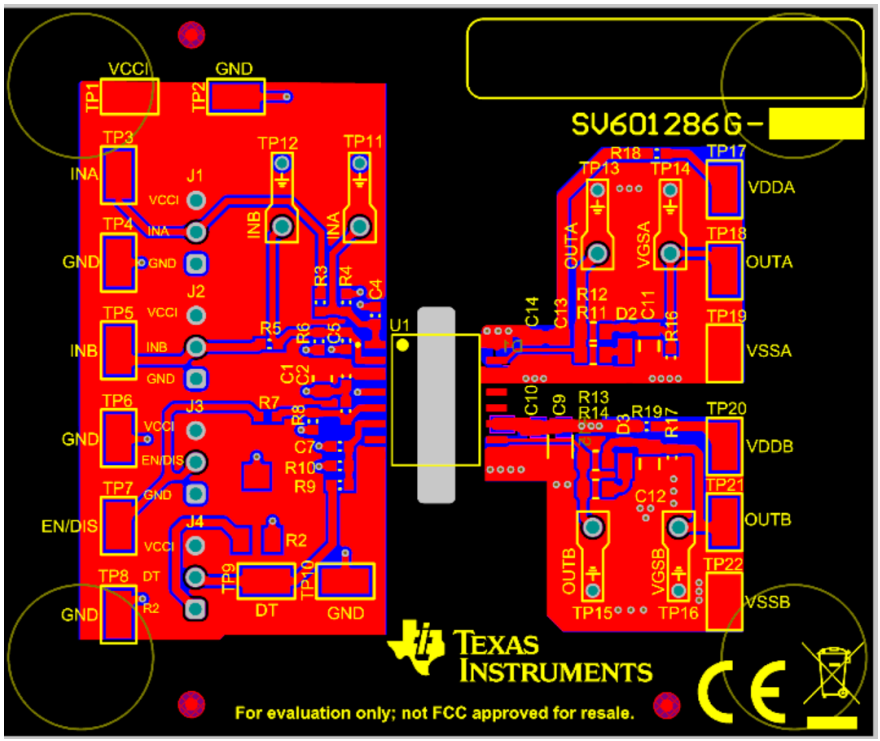


Figure 13: TI Gate Driver PCB [6]

Now, the circuit was modeled after Texas Instrument's design, however, the evaluation board did not fit all requirements, therefore the design needed to change. Some single components were also changed from their initial ones, such as the jumper resistors that were used in the design, as they were no longer needed for testing. The initial design for the circuit was limited to just the ground planes and the VCC planes, with the output from the right plane going to test holes.

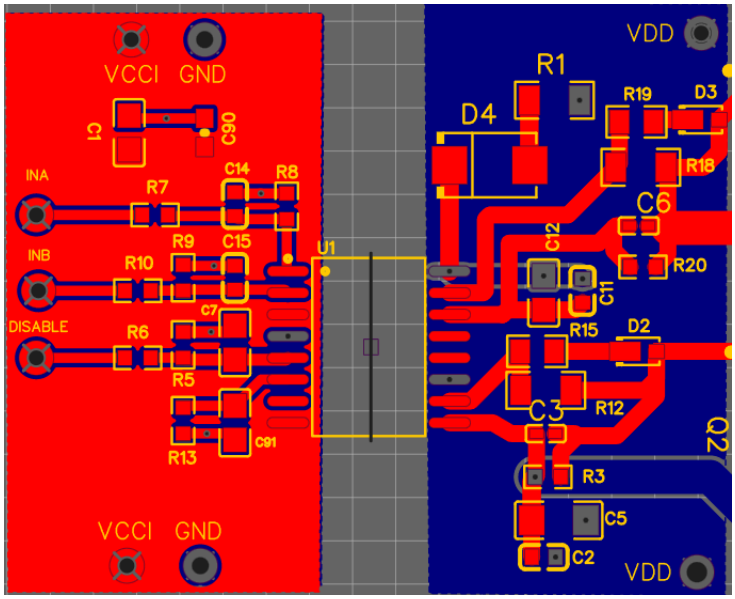


Figure 14: Initial Circuit Design

Since some components that were employed in the schematic by Texas Instruments were not available anymore, they were replaced by components of similar properties. The right side of the board was changed to include the high-power MOSFETs and points for testing them. Another diode was included with a bootstrap resistor that would limit the current supply when the MOSFETs are in HIGH side switch mode. This then allowed us to arrive at the final circuit design, which included two MOSFETs and a big diode on the top side.

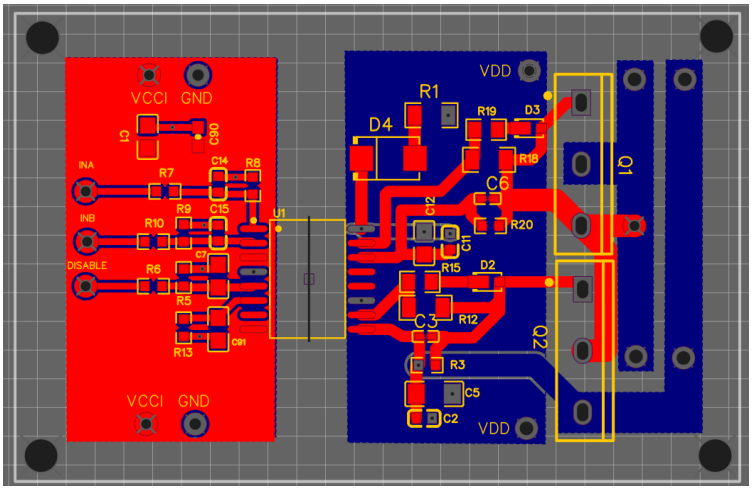


Figure 15: Proposed Final Circuit Design

This design was finalized with the red plane representing the positive part of the circuit and the blue plane representing the respective grounds for the components. The circuit components that were not available were changed to those of their counterparts with similar specifications.

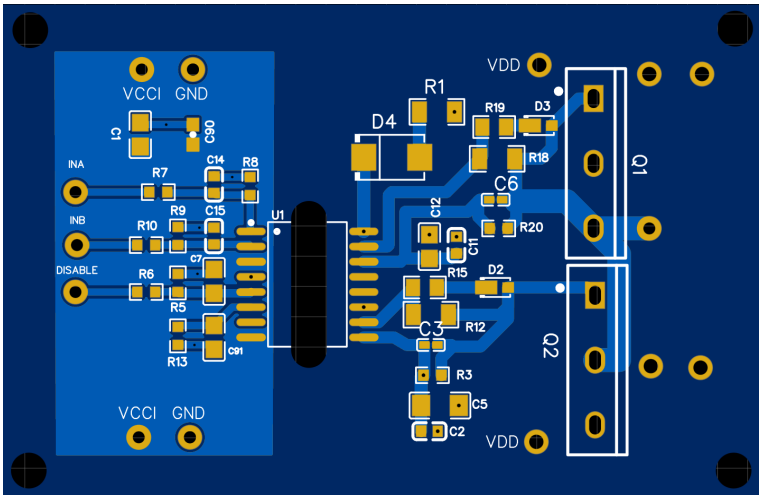


Figure 16: The 2D view of the Proposed Final PCB Design

The large slot hole that is in the middle of the gate driver chip is there for galvanic and thermal isolation. Finally, the PCB was cleaned up to look more presentable. The rendered design for the front is below in Figure 18.

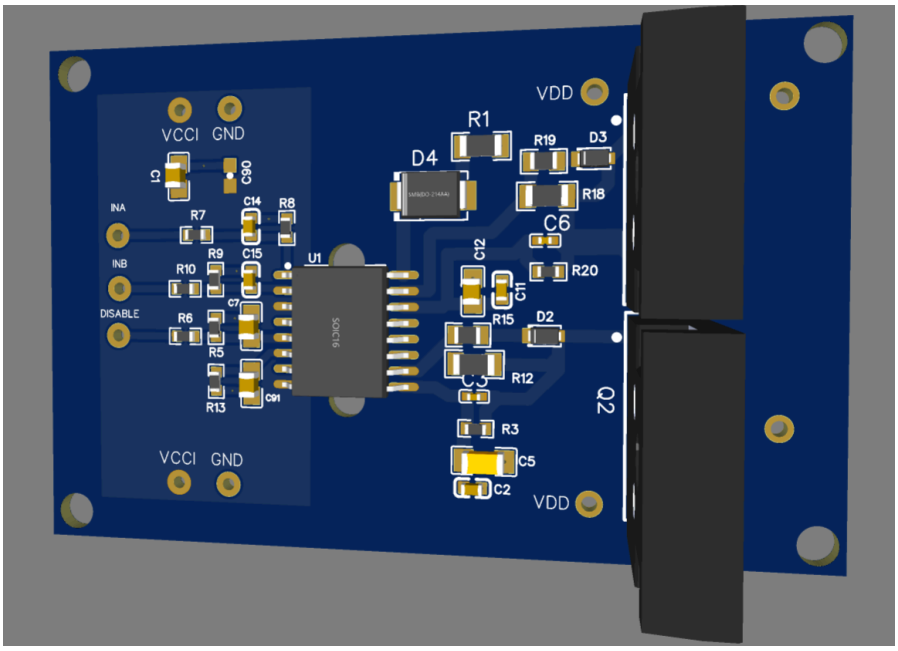


Figure 17: Front of the Proposed Final Circuit

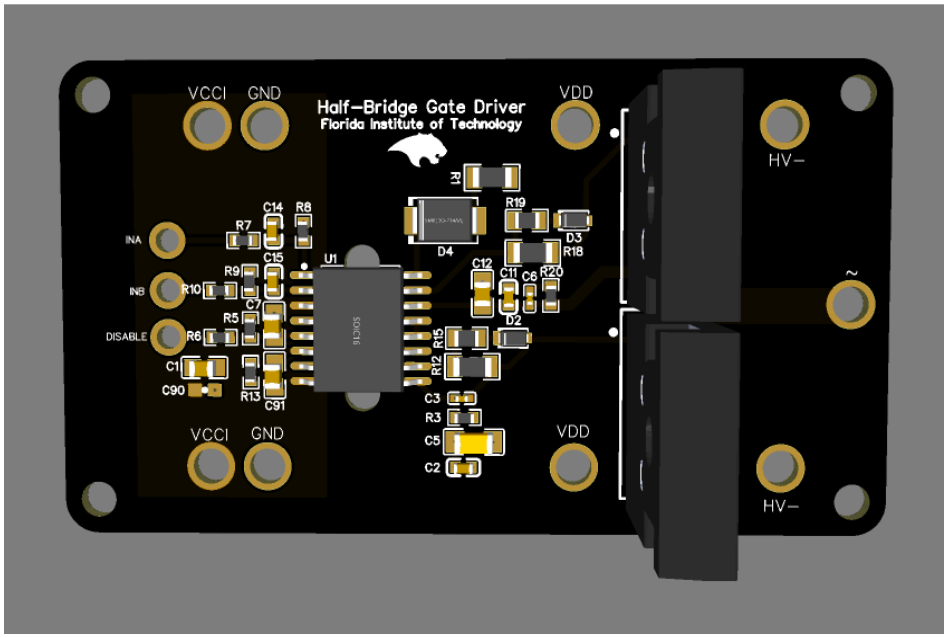


Figure 18: Front of the Final Ordered PCB [6a]

VI. THERMAL CONSIDERATIONS

One of the important aspects for the success of the traction inverter is keeping the MOSFETs within operable temperature range. If this is not done they can go into thermal runaway, which is a loop where high temperatures cause energy release that further increases temperature. This process then results in the complete breakdown of the MOSFET rendering it unusable. This can be prevented in a handful of ways, such as air cooling and water cooling.

Air cooling is the simpler of the two methods as it requires just heatsinks and airflow, typically provided by fans. While this is easy it is not as effective at removing the large amounts of heat that the MOSFETs will likely be producing. For water cooling, it is a bit more complicated as there is a water pump, reservoir, and radiator involved, as well as the fact that there would be flowing water near electric components. Despite the complexities, the benefits from water's heat capacity and conductivity will greatly outweigh them.

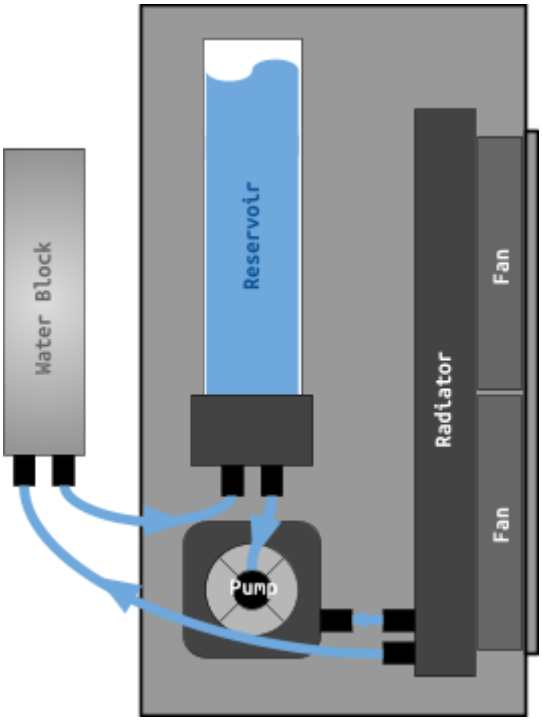


Figure 19: Proposed water-cooling loop

VII. CONTROL LOOP

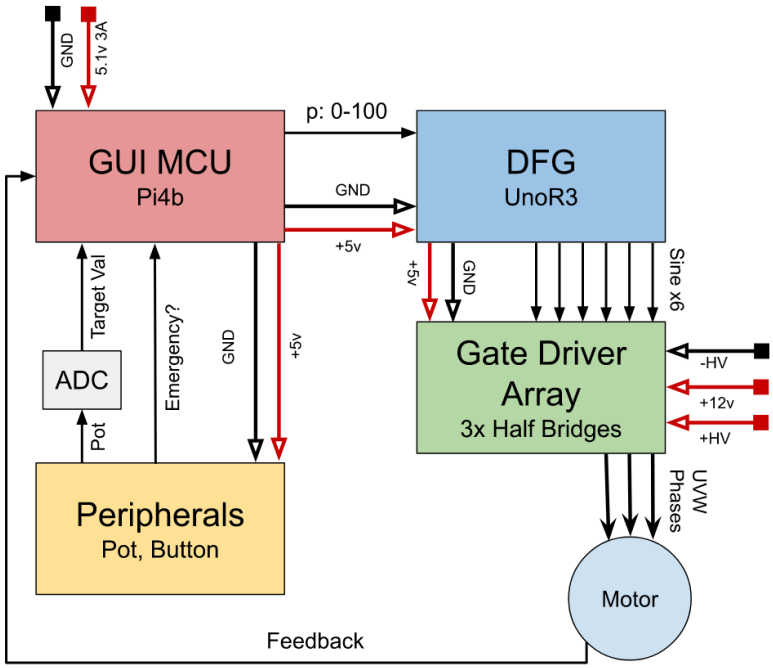


Figure 20: Proposed Main Control Loop

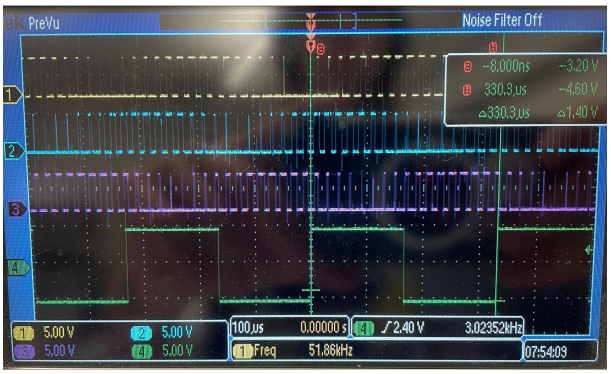
The control loop shown in figure 20 shows all necessary components to "convert" some high DC voltage to usable, variable, electrical power. In other words, this is a diagram for a Variable Frequency Drive (VFD). The red component is the GUI MCU, which will be a Raspberry Pi 4b. This component handles communication with peripherals, and feedback from the motor. This component will combine the peripheral data and the feedback from the motor to output a single desired frequency. Since this component is a Raspberry Pi, we can implement a GUI to accompany the computational purposes.

The next component, the blue component, is the Deterministic Function Generator. This term, which we have created as an alternative to the term "Arbitrary Function Generator", describes a function generator which outputs a scaled transformation of some function based on a single value. Since this definition is quite abstract, we will go into more detail in the next section.

The green component is the final component which we needed to design. This is the gate driver array. This is essentially 3 of the final PCBs described in section 5. These 3 PCBs will have their VCCI, GND, VDD, HV+, and HV- ports connected to each other. This will allow us to have one VCCI and VDD power supply to serve the gate driver array.

To summarize, as the motor moves, it produces data which the GUI MCU takes in. The GUI MCU also reads its peripherals. This data is then combined, and boiled down into a single number 0 to 100. This number is fed to the DFG over an SPI connection, which interprets this value as a percentage of the maximum allowed frequency and amplitude. The DFG then outputs 3 separate signals, doubled to 6 with physical inverters, to the gate driver array's 6 inputs. With the gate drivers hooked up in the correct configuration, the inverter will output the correct on-off sequence for the MOSFETs. As power runs through the MOSFETs and the inductors in the motor, the motor will continue to spin and send data back to the GUI MCU. This is the complete control loop for our traction inverter.

VIII. DETERMINISTIC FUNCTION GENERATOR (DFG)



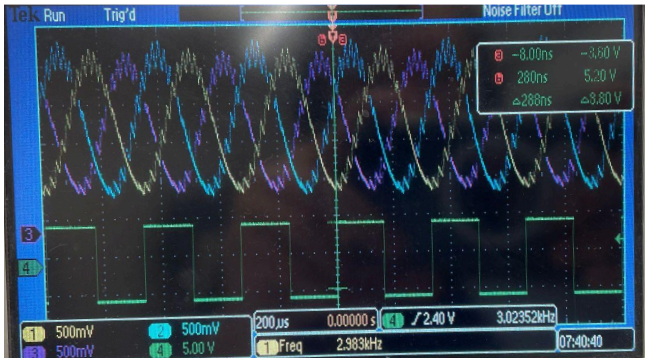


Figure 21 (left): Un-smoothed 3-Phase PWM Output [3a]
Figure 22 (right): Smoothed 3-Phase Output Visualization [3a]
(Note: Figure 22 is choppy; this is just an approximation with a mild low-pass filter)

The DFG, as described previously, generates a pre-defined control sequence for the MOSFETs. Since this is a relatively simple task, an Arduino, or more accurately an ATMEGA329P, suffices. The duty cycle is modulated from a near-zero percent value, to some maximum value no greater than 99% at some speed proportional to the value the DFG receives from the GUI MCU. The speed in which the duty cycle modulates represents the frequency. This modulation follows a basic sine function. The maximum duty cycle is proportional to the value the DFG receives from the GUI MCU and correlates to the amplitude of the wave.

Since we are using a relatively weak chip to do this, we need to make use of interrupts to generate the PWM sequence fast enough [10]. The "carrier frequency" of 50kHz is out of the functional capability of Arduinio's "void loop()" function. This is why we previously stated saying we used an ATMEGA329P [11] is more accurate. We directly manipulate registers, and write to program memory to ensure we can generate the PWM sequence fast enough. The exact code is far too complex to completely delve into, but it has been attached in the appendices. Instead, we will take a brief overview of the main functions used. First is the "void Setup_timers(void)" function which is run only once and configures the timers to overflow at the

correct time. Next is the “ISR(TIMER2_OVF_vect)” function which updates the duty cycle every 1/50,000th of a second. Finally, the “ISR(SPI_STC_vect)” receives the SPI value from the GUI MCU, and updates the frequency accordingly. Finally, not a function, is our sine look-up table defined by “const uint8_t sine256[100][256] PROGMEM,” which is a 100 by 256 array of 8-bit numbers. Each of the 100 arrays represents a sine function scaled to some percentage. For example, the 14th array is a copy of the 100th array but scaled to 14%.

The reason we did not use a single 256-length array is because that would imply that we need to scale each value in that array. This was actually our first of two failed approaches to the issue. The first, and most obvious is to scale the value as it is accessed. However, scaling a value by a non-integer amount, such as 0.50,. requires floating point arithmetic which is exceptionally slow. In fact, too slow to meet our speed requirements.

We also tried updating the entire array during downtime outside of the interrupt service routines. However, this will not work as you cannot modify program memory in runtime. Moving the array outside of program memory and allowing it to be modified, nullified the method we used to quickly find positions in memory. This method “pgm_read_byte_near()” also accounts for rollover. Even if we could search memory fast enough without this function, we would need to implement rollover logic, which would add further overhead. There likely is a more elegant solution than the one we found, but we determined that this is fully functional. If our requirements change in the future, we will switch to an STM32.

IX. GRAPHICAL USER INTERFACE

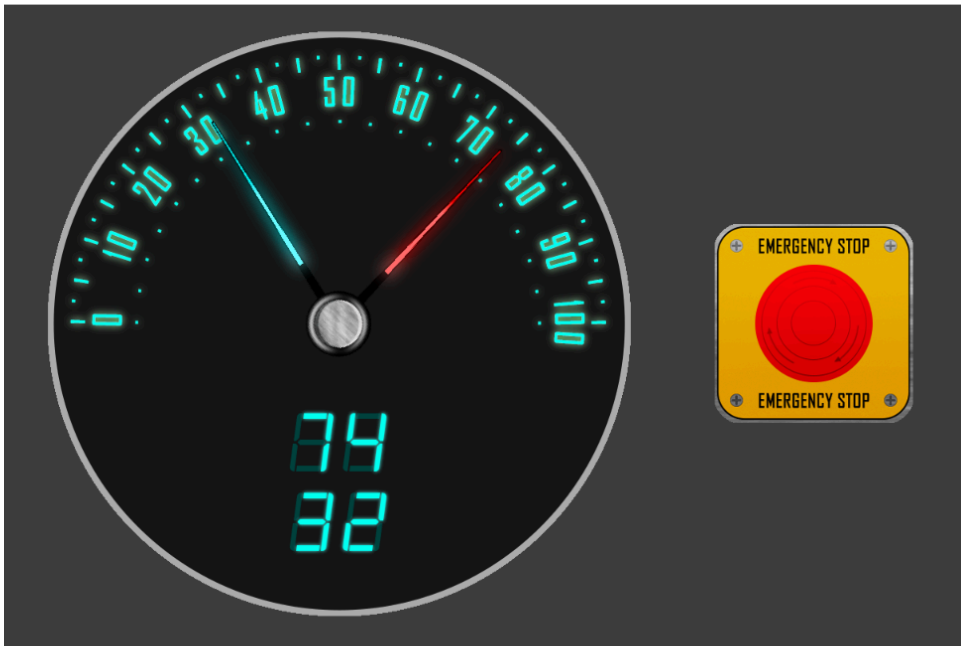


Figure 23: Graphical User Interface visible on the Raspberry Pi [1a]

The Graphical User Interface uses the Pygame Python library to display the custom-made graphics we designed for this project. This allows us to change the layout and look of the heads-up display on the fly. At this time, the Pygame GUI displays the target frequency, or position of the potentiometer, and the actual frequency being generated by the DFG [12]. This number, 0 to 100, is the percentage of the maximum frequency. The Emergency Stop button moves the target frequency to 0 and the actual outputted frequency to 0 instantly. This is one way of immediately stopping the motor from producing torque. We will also implement many other physical safety features as development progresses.

For IO, one special consideration we had to make for the Pi was the lack of an on-board ADC. This meant that we would need to resolve the position of the potentiometer. To achieve this, we had to use an Analog-to-Digital Converter (ADC) to be able to read the potentiometer position through the Pi. The code developed for this takes the reading of the potentiometer that is

from 0-1023, converts that into a percentage representing the target speed for the motor. To make the program run smoother we implemented a function that would store an array of potentiometer readings and output the average value, the size of the array was kept constant and the reading values would get updated according to FIFO principle (First In First Out).

Aside from the GUI and IO itself, the Pi also handles some computation as well. Based upon the target frequency, the current frequency being fed to the DFG, and the position and speed of the motor at a given time. The Pi will use a PID controller to determine the best next frequency to feed to the DFG. Since we do not yet have a motor moving, or a way of collecting motor data, we have not yet implemented the feedback portions. Instead we have implemented a smooth transition between frequencies with an ease-in type function. Once feedback from the motor can be resolved, we can implement a more complex PID controller to contribute to the control loop.

X. CONCLUSION

This semester, we've made significant strides in our project. We've successfully designed a robust control system and printed numerous PCBs. The thermal system has also been designed, and is ready for assembly. Additionally, the programming required for the GUI MCU, and DFG has been completed.

Our next steps include testing the PCBs and assembling the thermal system according to the design specifications. We will also test the FANUC motor using our newly developed PCBs, GUI MCU, and DFG. By recording and analyzing data from the motor, we will identify and resolve any issues. Once all components within our control loop are functioning as intended and the performance from the FANUC motor aligns with our simulation results, we will proceed to the final step. This will involve developing more robust circuit designs, and additional testing to determine the requirements of the ELUMINATOR M9000 MACH-E electric motor.

We’ve made significant progress, especially considering the lack of prior development seen from previous groups. Next semester, we expect to reach our goal of powering, and precisely controlling the FANUC AC Servo Motor. If time permits, and resources are available, we will also make as much progress as possible with the ELUMINATOR M9000 MACH-E. With a large fraction of our budget left, we are excited to be more experimental next semester, and gain additional useful insight into traction inversion technology.

XI. REFERENCES

[1] “Gate drivers | TI.com,” *ti.com*. https://www.ti.com/power-management/gate-drivers/overview.html

[2] “Isolated gate drivers | TI.com,” *ti.com*. https://www.ti.com/power-management/gate-drivers/isolated-gate-drivers/overview.html

[3] “Isolated gate drivers product selection | TI.com,” *ti.com*. https://www.ti.com/power-management/gate-drivers/isolated-gate-drivers/products.html#-1=UCC215%3Bfalse&

[4] “UCC21520-Q1,” *ti.com*, Jun. 17, 2024. https://www.ti.com/product/UCC21520-Q1

[5] “UCC5870-Q1,” *ti.com*, Sep. 01, 2021. https://www.ti.com/product/UCC5870-Q1

[6] “UCC21520EVM-286,” *ti.com*, Nov. 08, 2024. https://www.ti.com/tool/UCC21520EVM-286

[7] “UCC5870QEVM-045,” *ti.com*, Sep. 01, 2021. https://www.ti.com/tool/UCC5870QEVM-045

[8] A. Dearien, “HEV/EV Traction Inverter Design Guide Using Isolated IGBT and SiC Gate Drivers,” Texas Instruments Incorporated, Oct. 2022. https://www.ti.com/lit/an/slua963b/slua963b.pdf?ts=1733864944175&ref_url=https%253A%252F%252Fwww.google.com%252F

[9] hare_rushi, “3 Phase SPWM(sinusoidal PWM waveforms using the Arduino) to fed 3 Phase Inverter,” *Arduino Forum*, Mar. 23, 2024. https://forum.arduino.cc/t/3-phase-spwm-sinusoidal-pwm-waveforms-using-the-arduino-to-fed-3-phase-inverter/1238938

[10] M. Nawrath, “Arduino DDS Sinewave Generator,” *Laboratory for Experimental Computer Science at the Academy of Media Arts Cologne* . https://interface.khm.de/index.php/lab/interfaces-advanced/arduino-dds-sinewave-generator/index.html

[11] “ATmega328P 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash,” Atmel, Jan. 2015. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[12] Rabbid76, “How do I rotate an image around its center using Pygame?,” *stackoverflow.com*, Feb. 15, 2019. https://stackoverflow.com/questions/4183208/how-do-i-rotate-an-image-around-its-center-using-pygame/54714144#54714144

[13] “GE Fanuc Automation: α Series AC Servo Motor Descriptions Manual,” GE Computer Numerical Control Products, Mar. 1995. https://www.magnaproducts.com/wp-content/uploads/2019/09/a-Series-AC-Servo-Motor-Descriptions-Manual_GFZ-65142E02_compressed.pdf

XII. APPENDICES

[1a] "GUI_MCU_Software_v1_EVFIT2425.py"

D. May, S. Steakley, E. Karcini, A. L. Ceballos [12]

https://github.com/dmay2020/Senior_Design_Electric_Vehicle_24-25_Resources/blob/main/GUI_MCU_Software_v1_EVFIT2425.py

```
"""
GUI MCU Software

Electric Vehicle Capstone Research Team
Florida Institute of Technology Department of Electrical and Computer Engineering

Contributors : Derek May, Shawn Steakley, Elis Karcini, Alejandro Loynaz Ceballos

Last Updated : Fall 2024

MIT License

Copyright (c) [2024] [Derek May]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
"""

# Import(s)
import pygame, time, sys, random, math, smbus, statistics
import RPi.GPIO as GPIO
from pygame.locals import *
pygame.init()

# GFX Window Setup
BACKGROUND = (64, 64, 64)
WINDOW_WIDTH = 1920
WINDOW_HEIGHT = 1080
WINDOW = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption('Motor Control Panel')
FPS = 120
fpsClock = pygame.time.Clock()
```

```
# SPI Setup
CLK = 11
MISO = 21
MOSI = 13
SS = 15
GPIO.setmode(GPIO.BOARD)
GPIO.setup(CLK, GPIO.OUT)
GPIO.setup(MISO, GPIO.IN)
GPIO.setup(MOSI, GPIO.OUT)
GPIO.setup(SS, GPIO.OUT)

# POT Setup
ADS7830_ADDRESS = 0x4b
ADC_CHANNEL_0 = 0x00
bus = smbus.SMBus(1)

# SPI Send Byte Function
# Send 1 byte using the SPI protocol
def spiSendByte(data):
    GPIO.output(SS, GPIO.LOW)
    for bit in range(8):
        if data & 0x80:
            GPIO.output(MOSI, GPIO.HIGH)
        else:
            GPIO.output(MOSI, GPIO.LOW)
        data <<= 1

        GPIO.output(CLK, GPIO.HIGH)
        time.sleep(0.001)
        GPIO.output(CLK, GPIO.LOW)
        time.sleep(0.001)
    GPIO.output(SS, GPIO.HIGH)
    GPIO.output(MOSI, GPIO.LOW)

# Self-Contained Image Object
# Can be easily defined, updated and drawn to the screen multiple times.
class blitObject:
    def __init__(self, window, image_name, pos_x, pos_y, angle):
        self.window = window
        self.image = pygame.image.load(image_name)
        self.image.convert_alpha()
        self.width, self.height = self.image.get_size()
        self.center = (self.width/2, self.height/2)
        self.position = (pos_x, pos_y)
        self.angle = angle
        self.button_radius = min(self.width, self.height)/2

    def update(self):
        image_rect = self.image.get_rect(topleft = (self.position[0] - self.center[0], self.position[1] -
self.center[1]))
        offset_center_to_pivot = pygame.math.Vector2(self.position) - image_rect.center
        rotated_offset = offset_center_to_pivot.rotate(-self.angle)
        rotated_image_center = (self.position[0] - rotated_offset.x, self.position[1] - rotated_offset.y)
        rotated_image = pygame.transform.rotate(self.image, self.angle)
        rotated_image_rect = rotated_image.get_rect(center = rotated_image_center)
        self.window.blit(rotated_image, rotated_image_rect)

    def clicked(self, click_position):
        x1 = self.position[0]
```

```
        y1 = self.position[1]
        x2 = click_position[0]
        y2 = click_position[1]
        dx = abs(x2 - x1)
        dy = abs(y2 - y1)
        d = math.sqrt((dx * dx) + (dy * dy))
        if (self.button_radius >= d): return True
        else: return False

# Self-Contained 7-Segemnt Display Image Object
# Can be easily defined, updated and drawn to the screen multiple times.
class blit7seg:
    def __init__(self, window, pos_x, pos_y, size):
        if (size == "50" or size == "75" or size == "100"): self.size = size
        else: self.size = "100"
        file_ext = "__7seg_"+self.size+".png"
        self.d0 = pygame.image.load("0"+file_ext)
        self.d1 = pygame.image.load("1"+file_ext)
        self.d2 = pygame.image.load("2"+file_ext)
        self.d3 = pygame.image.load("3"+file_ext)
        self.d4 = pygame.image.load("4"+file_ext)
        self.d5 = pygame.image.load("5"+file_ext)
        self.d6 = pygame.image.load("6"+file_ext)
        self.d7 = pygame.image.load("7"+file_ext)
        self.d8 = pygame.image.load("8"+file_ext)
        self.d9 = pygame.image.load("9"+file_ext)
        self.d0.convert_alpha(window)
        self.d1.convert_alpha(window)
        self.d2.convert_alpha(window)
        self.d3.convert_alpha(window)
        self.d4.convert_alpha(window)
        self.d5.convert_alpha(window)
        self.d6.convert_alpha(window)
        self.d7.convert_alpha(window)
        self.d8.convert_alpha(window)
        self.d9.convert_alpha(window)
        self.window = window
        self.position = (pos_x, pos_y)
        self.width, self.height = self.d0.get_size()
        self.center = (self.width/2, self.height/2)

    def show(self, digit):
        image_rect = self.d0.get_rect(topleft=self.position)
        if (digit == 0): self.window.blit(self.d0, image_rect)
        elif (digit == 1): self.window.blit(self.d1, image_rect)
        elif (digit == 2): self.window.blit(self.d2, image_rect)
        elif (digit == 3): self.window.blit(self.d3, image_rect)
        elif (digit == 4): self.window.blit(self.d4, image_rect)
        elif (digit == 5): self.window.blit(self.d5, image_rect)
        elif (digit == 6): self.window.blit(self.d6, image_rect)
        elif (digit == 7): self.window.blit(self.d7, image_rect)
        elif (digit == 8): self.window.blit(self.d8, image_rect)
        elif (digit == 9): self.window.blit(self.d9, image_rect)
        elif (digit == 10):
            self.window.blit(self.d1, image_rect)
            self.window.blit(self.d0, image_rect)

# Read Potentiometer
# Reads the output of the ADC and averages it over 50 samples.
def read_pot(channel):
```

```
    inputs = [0] * 50
    bus.write_byte(ADS7830_ADDRESS, channel)
    for i in range(50):
        data = bus.read_i2c_block_data(ADS7830_ADDRESS, 0, 2)
        adc_value = (data[0] & 0xFF)
        inputs.append(adc_value)
        inputs.pop(0)
    avg_adc = statistics.mean(inputs)
    return math.floor((avg_adc / 255.0) * 100)

def main () :
    # Initialize Window Objects
    WINDOW.fill(BACKGROUND)
    gauge = blitObject(WINDOW, '0to100gauge100.png', WINDOW_WIDTH/2, WINDOW_HEIGHT/2, 0)
    red_needle = blitObject(WINDOW, 'redneedle100.png', WINDOW_WIDTH/2, WINDOW_HEIGHT/2, 180)
    blue_needle = blitObject(WINDOW, 'blueneedle100.png', WINDOW_WIDTH/2, WINDOW_HEIGHT/2, 180)
    estop = blitObject(WINDOW, 'estop_100.png', 600 + (WINDOW_WIDTH/2), WINDOW_HEIGHT/2, 0)
    estop.button_radius = 74
    tf_tens_7seg = blit7seg(WINDOW, (WINDOW_WIDTH/2)-67, 100+(WINDOW_HEIGHT/2), "100")
    tf_ones_7seg = blit7seg(WINDOW, WINDOW_WIDTH/2, 100+(WINDOW_HEIGHT/2), "100")
    f_tens_7seg = blit7seg(WINDOW, (WINDOW_WIDTH/2)-67, 200+(WINDOW_HEIGHT/2), "100")
    f_ones_7seg = blit7seg(WINDOW, WINDOW_WIDTH/2, 200+(WINDOW_HEIGHT/2), "100")

    # PID Dependencies
    max_rate_of_change = 4
    current_rate_of_change = 0.0
    emergency = False
    target_frequency = 0
    frequency = 0

    # Update Blit Objects
    estop.update()
    gauge.update()
    blue_needle.update()
    red_needle.update()
    tf_tens_7seg.show(0)
    tf_ones_7seg.show(0)
    f_tens_7seg.show(0)
    f_ones_7seg.show(0)

    # Main Loop
    # Main loop where variables and graphics are updated, where PID is implemented, and where data is sent.
    looping = True
    while looping :
        # Event Handling
        ev = pygame.event.get()
        for event in ev :
            if event.type == QUIT :
                spiSendByte(0)
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                if (estop.clicked(pos)):
                    emergency = not emergency

        # Data Input
        target_frequency = read_pot(ADC_CHANNEL_0)

        # PID Controller
```

```
        if not emergency:
            if target_frequency > 100: target_frequency = 100
            elif target_frequency < 0: target_frequency = 0
            current_rate_of_change = max_rate_of_change * (target_frequency - frequency) / 100.0
            frequency = frequency + current_rate_of_change
            if frequency > 100:
                frequency = 100
            elif frequency < 0:
                frequency = 0
        else:
            target_frequency = 0
            frequency = 0

        # Data Output
        spiSendByte(int(math.floor(frequency)))

        # Update Blit Objects
        estop.update()
        gauge.update()
        blue_needle.angle = 180.0 - (frequency / 100.0 * 180.0)
        red_needle.angle = 180.0 - (target_frequency / 100.0 * 180.0)
        blue_needle.update()
        red_needle.update()
        tf_tens_7seg.show(math.floor(target_frequency/10))
        tf_ones_7seg.show(target_frequency - 10*(math.floor(target_frequency/10)))
        f_tens_7seg.show(math.floor(frequency/10))
        f_ones_7seg.show(math.floor(frequency - 10*(math.floor(frequency/10))))

        # Render Frame
        pygame.display.flip()
        dt = fpsClock.tick(FPS)/1000
        pygame.display.set_caption(f"MOTOR GUI | {math.floor(fpsClock.get_fps())}")

    GPIO.cleanup()

main()
```

[2a] "TRIMMED_DFG_Firmware_v1.0_EVFIT2425.ino"

D. May, S. Steakley, E. Karcini, A. L. Ceballos [9,10]

https://github.com/dmay2020/Senior_Design_Electric_Vehicle_24-25_Resources/blob/main/TRIMMED_DFG_Firmware_v1.0_EVFIT2425.ino

```
/*
Deterministic Function Generator Firmware

Electric Vehicle Capstone Research Team
Florida Institute of Technology Department of Electrical and Computer Engineering

Contributors : Derek May, Shawn Steakley, Elis Karcini, Alejandro Loynaz Ceballos

Last Updated : Fall 2024

MIT License
Copyright (c) [2024] [Derek May]
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
*/
// Include(s)
#include <SPI.h>

// 0%-100% Scaled Sine LUT with precision 256
const uint8_t sine256[100][256] PROGMEM =
{
// ...
// 100 LINES OF RAW DATA OMITTED FOR BREVITY AND READABILITY. FULL CODE WILL BE ATTACHED ELSEWHERE.
// ...
};
uint8_t sine256tmp[256] = {0};

// Output Pins
#define PWM1 3        // Phase 1 = OC2B
#define PWM2 9        // Phase 2 = OC1A
#define PWM3 0        // Phase 3 = OC1B
#define SCK_PIN   13  // D13 = pin19 = PortB.5
#define MISO_PIN  12  // D12 = pin18 = PortB.4
#define MOSI_PIN  11  // D11 = pin17 = PortB.3
#define SS_PIN    10  // D10 = pin16 = PortB.2

// Phase Shifts
const int Phase2Offset = 256 / 3;       // 120° phase shift
const int Phase3Offset = (256 * 2) / 3; // 240° phase shift

int freqPercent               = 0;        // Default percentage of the maximum frequency
double OutputFrequency        = 0.0;      // Default frequency in Hz
double MaximumFrequency       = 3000.0    // Maximum frequency in Hz
const double CarrierFrequency = 31376.6;  // Carrier frequency in Hz

// Information about how and why the tuning value is calculated like this is from NHM 2009 / Martin Nawrath
// https://interface.khm.de/index.php/lab/interfaces-advanced/arduino-dds-sinewave-generator/index.html
volatile unsigned long TuningValue;
volatile unsigned long TuningValueMax = pow(2, 32) * MaximumFrequency / CarrierFrequency;

// Timer Setup Procedure
// To get MaximumFrequency we need TOP = (16 MHz / MaximumFrequency / prescale / 2) - 1 = 159
void Setup_timers(void)
{
  // Timer 1 setup
  // WGM=8 Phase/Frequency Correct PWM with TOP in ICR1
  TCCR1A = 0;                           // Clear Timer/Counter Control Register 1A
```

```
  TCCR1B = 0;                           // Clear Timer/Counter Control Register 1B
  TIMSK1 = 0;                           // Disable all Timer1 interrupts
  ICR1 = 159;                           // TOP value = 159
  TCCR1A |= _BV(COM1A1) | _BV(COM1B1);  // Enable PWM on A and B
  TCCR1B |= _BV(WGM13) | _BV(CS10);     // WGM = 8, Prescale = 1

  // Timer 2 setup
  // WGM=5 Phase Correct PWM with TOP in OCR2A
  TCCR2A = 0;                           // Clear Timer/Counter Control Register 2A
  TCCR2B = 0;                           // Clear Timer/Counter Control Register 2B
  TIMSK2 = 0;                           // Disable all Timer2 interrupts
  OCR2A = 159;                          // TOP value = 159
  TCCR2A |= _BV(COM1B1);                // Enable PWM on B only. (OCR2A holds TOP)
  TCCR2A |= _BV(WGM20);                 // WGM = 5
  TCCR2B |= _BV(WGM22) | _BV(CS20);     // WGM = 5, Prescale = NONE
  TIMSK2 |= _BV(TOIE2);                 // Enable Timer2 Overflow Interrupt
}
void setup()
{
  // Define Outputs
  pinMode(PWM1, OUTPUT);
  pinMode(PWM2, OUTPUT);
  pinMode(PWM3, OUTPUT);
  pinMode(MISO_PIN, OUTPUT);  // Set MISO as OUTPUT (required for SPI slave)
  pinMode(MOSI_PIN, INPUT);   // MOSI as INPUT (master out, slave in)
  pinMode(SCK_PIN, INPUT);    // SCK as INPUT (clock from master)
  pinMode(SS_PIN, INPUT);     // SS as INPUT (to know when selected as slave)

  // Configure SPI
  SPCR |= _BV(SPE);           // Enable SPI in slave mode
  SPI.attachInterrupt();      // Enable SPI interrupt

  // Setup Timers
  TuningValue = pow(2, 32) * OutputFrequency / CarrierFrequency;
  Setup_timers();
}

// Timer 2 Interrupt
// Runs every 1/CarrierFrequency seconds
ISR(TIMER2_OVF_vect)
{
  static uint32_t phase_accumulator = 0;
  phase_accumulator += TuningValue;
  uint8_t current_count = phase_accumulator >> 24;
  OCR2B = pgm_read_byte_near(sine256[freqPercent] + current_count);
  OCR1A = pgm_read_byte_near(sine256[freqPercent] + (uint8_t)(current_count + Phase2Offset));
  OCR1B = pgm_read_byte_near(sine256[freqPercent] + (uint8_t)(current_count + Phase3Offset));
}

// SPI Interrupt
// Runs whenever data has been received over SPI
ISR(SPI_STC_vect) { freqPercent = SPDR; }

void loop() {
  OutputFrequency = (int)(((freqPercent * MaximumFrequency) / 100.0));
  TuningValue = pow(2, 32) * OutputFrequency / CarrierFrequency;
  delay(5);
}
```

[3a] “DFG_Firmware_v1.0_EVFIT2425.ino”

D. May, S. Steakley, E. Karcini, A. L. Ceballos [9,10]

https://github.com/dmay2020/Senior_Design_Electric_Vehicle_24-25_Resources/blob/main/DFG_Firmware_v1.0_EVFIT2425.ino

[4a] “dds_calc.xls”

Martin Nawrath [10]

https://interface.khm.de/wp-content/uploads/2009/12/dds_calc.xls

[5a] “Motor Brake Coil Driver PCB Specification”

D.May, P. Maruur [13]

https://github.com/dmay2020/Senior_Design_Electric_Vehicle_24-25_Resources/blob/main/PCB-Specification_HalfBridgeGateDriverF24.zip

[6a] “Half Bridge Gate Driver PCB Specification”

D.May, P. Maruur [4,6]

https://github.com/dmay2020/Senior_Design_Electric_Vehicle_24-25_Resources/blob/main/PCB-Specification_MotorBrakeCoilDriverF24.zip