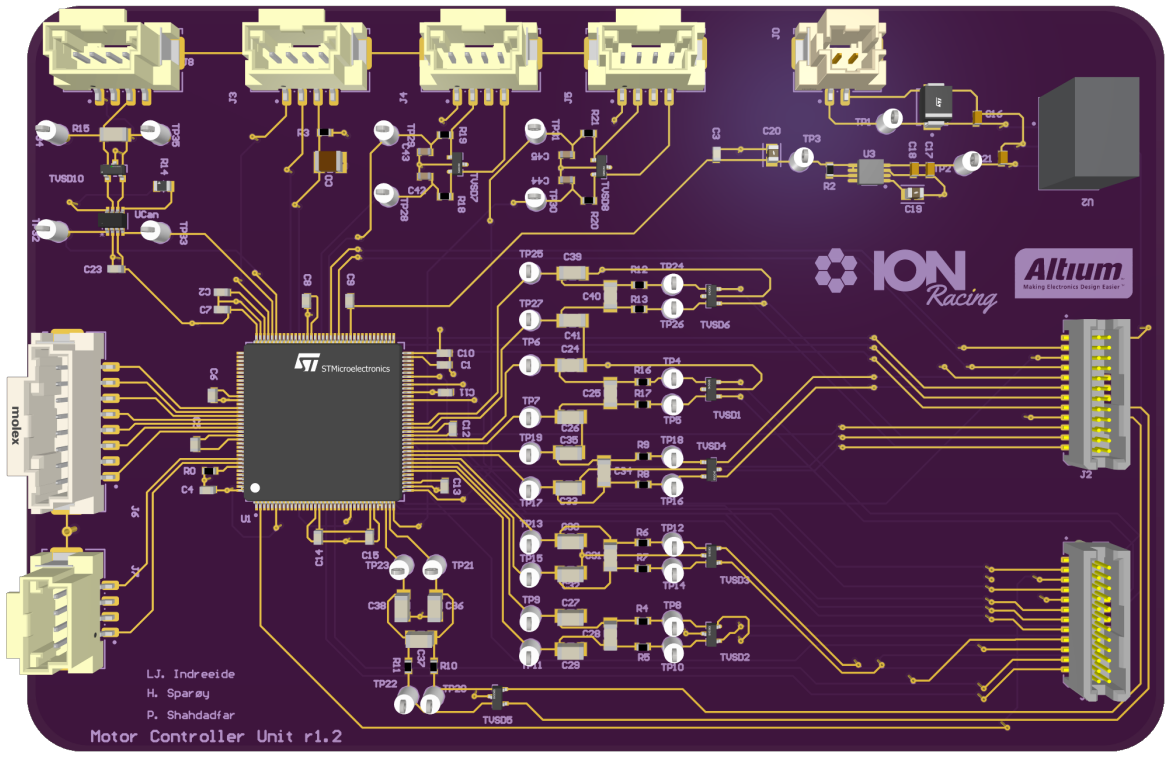
FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme / specialisation: Automation and electronics design	Spring semester, 2023 Open
Author: Lars Johann Indreeide, Herman Sparøy, Philip Shahdadfar	
Supervisor at UiS: Kristian Thorsen Supervisor(s): Kristian Thorsen	
Thesis title: Design and production of a motor controller for use in an electric race car	
Credits (ECTS): 20	
Keywords: Motor controller ION Racing Formula Student MCU	Pages: 69 + appendix: 30 Stavanger, *05.06.23*

Approved by the Dean 30 Sep 21
Faculty of Science and Technology



Design and Production of a Motor Controller for Use in an Electric Race Car

Philip Shahdadfar, Herman Sparøy, Lars Johann Indreeide

Department of Electrical Engineering and Computer Science

University of Stavanger

Stavanger, Norway, 2023

Abbreviations

ADC Analog Digital Converter

BMS Battery Management System

CAN Controller Area Network

CCR Capture Compare Register

ESL Equivalent Series Inductance

ESR Equivalent Series Resistance

FSUK Formula Student United Kingdom

FSG Formula Student Germany

FSN Formula Student Netherlands

HSE High Speed External

HV High Voltage

LED Light Emitting Diode

LL Low Level

LSE Low Speed External

LV Low Voltage

MCU Motor Control Unit

MOSFET Metal Oxide Semiconductor Field Effect Transistor

PCB Printed Circuit Board

PWM Pulse-Width Modulation

RDC Resolver To Digital Converter

SCF Switched Capacitance Filter

SCS System Critical Signal

SOC State of Charge

TS Tractive System

USART Universal Synchronous/Asynchronous Receiver/Transmitter

Abstract

This thesis is written in cooperation with the student organization ION Racing at the University of Stavanger. ION Racing is a Formula Student team that competes in the Formula Student competition each year. The choice to make a motor controller was made due to how it was the only electrical system that was not produced within ION Racing, and having self-produced systems gives a higher score at the competition. Testing and designing the motor controller has been a valuable, challenging and fun experience.

A development board and a gate driver was used to test motor control algorithms. Due to resolver availability the algorithms were only partially successful. A potential design has been drafted, which can be expanded upon.
In the end the MCU was not produced within the time of this thesis due to supply line challenges and time constraints, but will be produced and tested in the future for the 2024 car.

Acknowledgements

We want to thank:

- Our supervisor, Kristian Thorsen, for guiding and helping us with the project and thesis.
- Jon Fidjeland, for allowing the use of the laboratory and helping us produce our expansion board.
- Andreas Byskov, Technical leader for ION Racing.
- Per Gundersen Lund, for giving valuable input and assistance.
- ION Racing sponsors which has made the Formula Student project possible. Of the sponsors we'd like to give special recognition to Altium LTD and RS components for their direct contribution.
- All the members of ION Racing and its Alumni.

Contents

1 Introduction **10**
- 1.1 Formula Student . . . 11
  - 1.1.1 Point distribution in FSUK . . . 11
- 1.2 The electrical system . . . 13
- 1.3 Objective of this project . . . 14
- 1.4 Concept . . . 14
- 1.5 Tasks required for the MCU . . . 16
- 1.6 Physical placement of the card . . . 17
- 1.7 FSUK Rules . . . 19

2 Theory **22**
- 2.1 Electric motors . . . 23
  - 2.1.1 Permanent Magnet Synchronous Motors . . . 23
  - 2.1.2 EMRAX 228 . . . 25
- 2.2 Torque modulation . . . 26
- 2.3 Positional Detection of the Motor Shaft . . . 27
  - 2.3.1 Resolver Theory . . . 27
  - 2.3.2 Sensorless detection of the motor shaft position . . . 28
- 2.4 Implementation in a microcontroller . . . 30

3 Software and testing **32**
- 3.1 Testing setup . . . 33
  - 3.1.1 Micro controller . . . 33
  - 3.1.2 Gate driver . . . 34
  - 3.1.3 Expansion board . . . 34
  - 3.1.4 Software . . . 38

4 Controller Design **45**
- 4.1 Component choices and schematics . . . 46
  - 4.1.1 The controller STM32H723ZGTx . . . 46
  - 4.1.2 The TSR 1-2450 . . . 48
  - 4.1.3 The LP2989 . . . 50
  - 4.1.4 The MAX3051EKA+T . . . 51
  - 4.1.5 The MLX90380 . . . 52

4.1.6 Molex DuraClik . . . 53
4.2 Differential ADC Design . . . 54
4.3 Resolver ADC design . . . 55
4.4 Connectors . . . 56
4.5 Printed circuit board . . . 60
4.5.1 Improvements which should be considered . . . 61

5 Connectivity **62**
5.1 CAN . . . 62
5.1.1 Why use CAN? . . . 62
5.1.2 Differential signaling . . . 63
5.1.3 CAN Setup . . . 63
5.1.4 Notes regarding further design with CAN . . . 64
5.2 SPI . . . 64
5.2.1 Why use SPI . . . 64
5.2.2 How SPI works . . . 64
5.3 I2C . . . 65
5.3.1 Why use I2C . . . 65
5.3.2 How I2C works . . . 65

6 Discussion and further work **66**
6.1 The system . . . 67
6.2 Software . . . 67
6.3 Design . . . 67
6.4 Further work . . . 67
6.5 Conclusion . . . 68

A Software code **70**
A.1 main.c . . . 70
A.2 stm32h7xx_it.c . . . 87

B Altium Schematics **93**

List of Figures

1.1 ION Racing & Formula Student . . . 10
1.2 Overview of the electrical system in the car . . . 13
1.3 Placement in the car . . . 18
1.4 The incomplete motor controller box . . . 19

1.5 An illustration of the shutdown circuit 21

2.1 Permanent magnet motor with 2 pole pairs 23
2.2 Star and delta configurations 24
2.3 EMRAX 228 as shown in [[4], Figure 8, Page 8] 25
2.4 Illustration of duty cycles 26
2.5 Illustration of a Simple Unit Circle 27
2.6 Measured voltage from manually rotated motor 29
2.7 Commutation sequence from hypothetical back EMF readings 30
2.8 Output characteristics plot taken from the MLX90380 datasheet [9] 31

3.1 NUCLEO-H723ZG development board 33
3.2 EVALSTDRIVE601 demonstration board 34
3.3 Expansion board 35
3.4 Expansion board in Altium 35
3.5 Entire testing setup 37
3.6 Example of complementary channels, where yellow and green are complementary, and blue and purple are complementary 38
3.7 Phase A, B and C's high output (yellow, green and blue respectively) with a moderate duty cycle 40
3.8 Phase A, B and C's high output (yellow, green and blue respectively) with a lower duty cycle 40
3.9 Readings from 2 of the phases from top to bottom: HIN1, LIN1, HIN2 and LIN2 41
3.10 Flowchart for the main.c logic in the project (full code in appendix A.1) . . . 42
3.11 Flowchart for the interrupts in stm32h7xx_it.c (full code in appendix A.2) . . 43
3.12 Readings from HIN1 (yellow) and the three back EMF readings, A (green), B (blue) and C (purple) 44

4.1 The controller in Altium Designer 47
4.2 The power section of the controller 48
4.3 TSR in Altium Designer 49
4.4 The LP2989 in Altium Designer 51
4.5 The MAX3051 in Altium Designer 52
4.6 The MLX90380 in Altium Designer 53
4.7 Common Mode Differential Filter Design in Altium 55
4.8 ADC circuit with *Cos* and *Sin* 56
4.9 12 Volt input 57
4.10 Connector to drivers 57
4.11 Debug connector 58
4.12 Resolver Connectors 58
4.13 Connectivity: SPI, I^2C and CAN 59
4.14 Top view of controller in Altium Designer 60

4.15Placement of parts on the controller 61

List of Tables

1.1 Point Distribution 11
1.2 Arguments for and against centralized and decentralized systems 15

2.1 Mechanical & electrical EMRAX 228 data from [[3], Page 1] 25
2.2 Illustration of the sensorless commutation sequence 28

3.1 Signals between μC and gate driver 36

List of Equations

2.1 Total power supplied in a PMSM with star connection 25
2.2 Total power supplied in a PMSM with delta connection 25
2.3 Average voltage supply in PWM 26
2.4 Equation to calculate the angle. 28
4.3 Resistance in the common mode differential filter 54
4.4 Differential capacitor 54
4.5 Common mode capacitor 54

Chapter 1

Introduction

Figure 1.1: ION Racing & Formula Student

ION Racing is a formula student team and a student organisation at the Univeristy of Stavanger (UiS). Each year a race car is designed and created in order to compete in Formula Student. The organisation was founded in 2012 and mainly consists of engineering students, such as mechanical, electrical and computer science engineers. ION Racing however also consists of students from various courses including economic and marketing students.

1.1 Formula Student

Formula student is the biggest engineering competition for engineering students where students from over 650 universities compete. The competition is held in multiple countries all over the world, for example England, Netherlands and Germany. Each competition has a comprehensive list of rules and guidelines in order to compete and keep the competition safe. The rules encompass just about everything about the car from the frame of the car down to which bolts you can fasten your seat with.

Each year to qualify for the different competitions each competition holds quizzes the teams have to take to attend. The quizzes contain questions from the rules and technical mechanical and electrical problems.

ION Racing has competed in Formula Student for several years. Last season (2022) ION Racing competed in Formula Student United Kingdom, henceforth referred to as FSUK, where the team will be competing once again this year. The competition is held at the Silverstone Circuit, Northhamptonshire England, at the end of July. This thesis will therefore be based on the rules of FSUK.

The competition is divided into two parts. A *Dynamic* event and a *Static* event. The winner of the competition is the team with most points in total from both events. There are also prizes teams can win in individual categories. Last year ION Racing won the "Efficiency" category after completing the endurance race with the most efficient electrical car.

There are three entry classes: Formula Student Class (FS Class), Formula Student - Artificial Intelligence Class (FS-AI Class) and a Concept Class. Since ION Racing has focused on electric cars the past years, the FS Class (class with a functioning car) will be considered, specifically the rules for electric vehicles (EV).

1.1.1 Point distribution in FSUK

Dynamic		Static	
Acceleration	75 pts		
		Business Plan Presentation	115 pts
Skidpad	75 pts		
		Cost and Manufacturing	115 pts
Autocross/Sprint	75 pts		
		Engineering Design Event	150 pts
Endurance	250 pts		
		Lap Time Simulation	20 pts
Efficiency	100 pts		
Total Dynamic	575 pts	Total Static	400 pts

Table 1.1: Point Distribution

The *Dynamic* events of the competition consists of: [5]

- **Acceleration:** 75 Points
  The cars acceleration is tested over 75 meters.

- **Skidpad:** 75 Points
  The car is driven in a figure 8 to test the maneuverability of the car.

- **Autocross/Sprint:** 75 Points
  The car is driven through a small track consisting of straights, constant turns, slaloms, and chicanes to test the racing capability of the car.

- **Endurance:** 250 Points
  The car is driven around a set track for a complete distance of 23km.

- **Efficiency:** 100 Points
  A measurement of how much energy the car has consumed during endurance is made, and the car is scored based on the results.

The *Static* events of the competition consists of: [5]

- **Business Plan Presentation:** 115 Points
  The team holds a presentation and the judges evaluate the team's ability to develop and deliver a comprehensive business model.

- **Cost and Manufacturing:** 115 Points
  The team is judged on their understanding of the manufacturing process and costs associated with building a car.

- **Engineering Design Event:** 150 Points
  Each team has practical design presentation of their car and answers any question given by the judges.

- **Lap Time Simulation:** 20 Points *(40 for concept class)* The team simulates the dynamic events.

1.2 The electrical system

The electrical system of the car consists of many different parts working together(see figure 1.2).

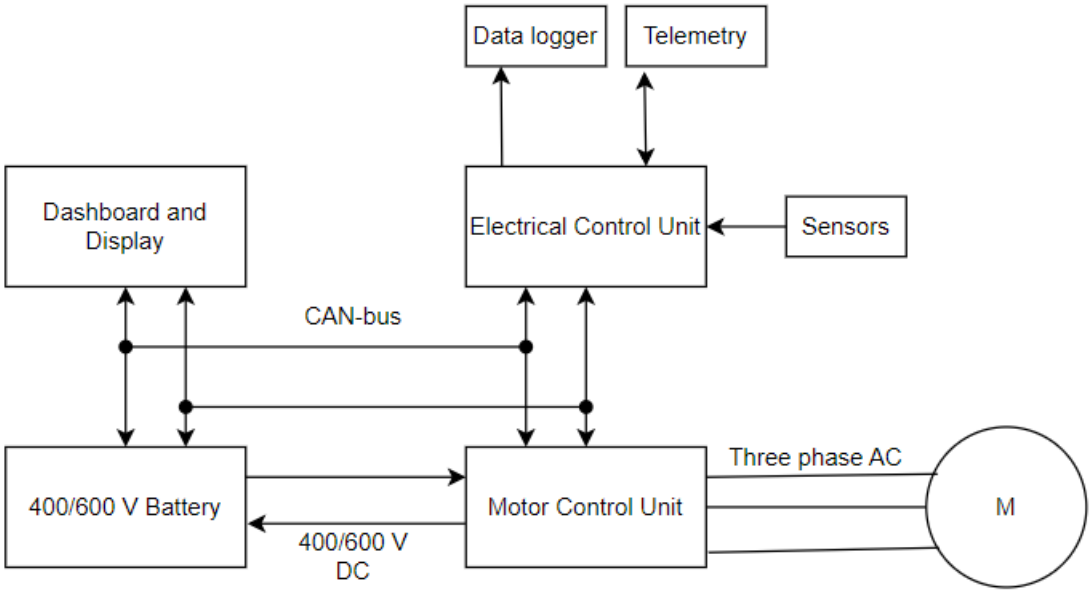


Figure 1.2: Overview of the electrical system in the car

- **Electronic Control Unit (ECU):** Viewed as the brain of the car as it is responsible for receiving and transmitting data with the rest of the electronics in the car. This includes reading data from sensors, sending torque requests to the MCU, checking for errors and determines whether the car is operating under normal conditions.

- **Motor Control Unit (MCU):** The main function of the MCU is to match the created torque in the motor to the pressure on the gas pedal. It also needs to take readings from the driver card.

- **Inverter-bridge:** The inverter-bridge is responsible for converting the DC voltage from the battery into three phase AC voltage to the motor.

- **High Voltage Battery:** Energy source of the engine of the car aswell as some other components that require a higher voltage.

- **Dashboard and Display:** The dashboard of the car includes numerous mandatory signals and switches. These are either a part of the start-up sequence of the car or warning lights informing of different states of the car. The display of the car is a

customizable display showing the driver information during driving such as speed or battery level.

- **Sensors:** There is a plethora of sensors on the car monitoring the states of different components. There are sensors monitoring the temperature of the motor, the voltage and current levels in select areas of the car, pedal positions and RPM of the wheels.

- **Data logger:** The data logger stores data of various sensors in the car so the team can review it later for optimization or trouble shooting.

- **Motor(s):** The motor is a brushless motor either placed centrally in the back of the car in the case of a single motor car, or in the case of 4 motors placed on each wheel.

1.3 Objective of this project

The increase of electric vehicles in the past 10 years has been immense and now that world is more environmentally aware, the electric type seems to be the future of vehicles. Since 2014 ION Racing's goal has been to compete in Formula Student using electric race cars. The team has over the years focused building in-house components rather than purchasing them as this results in more points in the static events of competitions.

The objective of this thesis is to design a motor controller that can control several motors simultaneously. Although the objective is to be able to control up to 4 motors, it's important to take into account the budget, difficulty and whether it is optimal to proceed to four-wheel drive over having one or two motors first. With this in mind there are a few requirements that has to be met. The controller needs to be able to communicate over CAN, output several PWM signals, take in resolver or encoder data and use ADCs to keep track of information from the driver cards.

1.4 Concept

This MCU has a centralized concept consisting of one PCB. Designing the MCU in this way has its pros and cons. Table 1.2 will show and illustrate pros and cons of the centralized concept.

Centralized	Decentralized
Occupies less space	Occupies more space
Lower cost	Higher cost
Lower complexity of design	Greater complexity of design
Higher noise generation	Lower noise generation
Difficult to expand	Easy to expand
Cannot be modified	Can be modified to an extent
Entire system must be replaced in case of damages	Subsystems can be replaced in case of damages
Many connections to one card	Few connections to each card

Table 1.2: Arguments for and against centralized and decentralized systems

Walk-through of the following arguments for and against a centralized concept.

- **Occupies less space:**
  As the rear of the car is cramped between the motor, driveshaft, battery and other mechanical components. Taking the possibility of still having one motor for the next car, and experiences with the current car. Space is essential to having a solution that is easy to work with and is less prone to accidental damage from working on the car.

- **Lower Price:**
  Fewer components and smaller area compared to decentralized will lead to a lower production price. The system will still be cheaper than a commercial solution.

- **Lower complexity of design:**
  There are fewer cards that need to be designed and there is no requirement for communication between them. This will lead to a smaller workload compared to designing and building multiple cards.

- **Increased noise:**
  In a centralized design parts that generate noise such as PWM be placed closer to other parts that are sensitive to noise. This will be countered by using more robust ADCs such as differential ADCs that have a higher tolerance for noise.

- **Expansion is somewhat harder:**
  A centralized design would require you to produce a new card every revision instead of being able to add another card.

- **Cannot easily be modified:**
  Modifying a card in a decentralized design would require you to replace the entire card compared to a decentralized card that would just require you to replace the upgraded card.

- **Replaceability:**
  If a part of the card is damaged and cannot be repaired a new card has to be produced. In a decentralized design you would have the opportunity to only have to reproduce the damaged card.

- **More cables:**
  More cables would mean more EMI, which would require sensitive signals to be shielded in certain cables. Additionally this would increase the complexity of the cable network and increase the difficulty of troubleshooting. This can be countered by practising proper cable management techniques and marking of cables.

1.5 Tasks required for the MCU

The MCU needs to do these following tasks:

- Acquire data from ECU through CAN.
- Activate the inverters.
- Output PWM to the inverters.
- Acquire signals and data from the inverters.
- Shutdown when fault is detected from the inverters.
- Log information gained from inverters.
- Communicate with inverters and ECU.
- Communicate with external PC.

The design for the inverter has yet to be finalized so the details of the signals the MCU is supposed to read and transmit is currently only conceptualized. The current list per card is as follows:

- 1x Fault
- 1-6x PWM inputs (Input capture)
- 2x Analogue current sensors (Differential if possible)
- 2x Delta sigma current signals
- 1x Digital bus (I^2C or SPI)

In order to facilitate these requirements the MCU will be designed with both I^2C and SPI, as well as using the more robust differential ADCs available on the selected µC. The MCU will also be able to connect to the CAN network in the car to transmit and log its readings. Communication with the resolvers will be through ADCs that have been filtered from high frequency noise and then further filtered and processed in software. The data from the resolver will be used to determine what PWM signals are output to the inverters and to measure the angle and speed of the motors.

1.6 Physical placement of the card

The card will be placed in the back of the car in a box containing the ECUs decentralized temperature sensor card and one to four inverters 1.3.

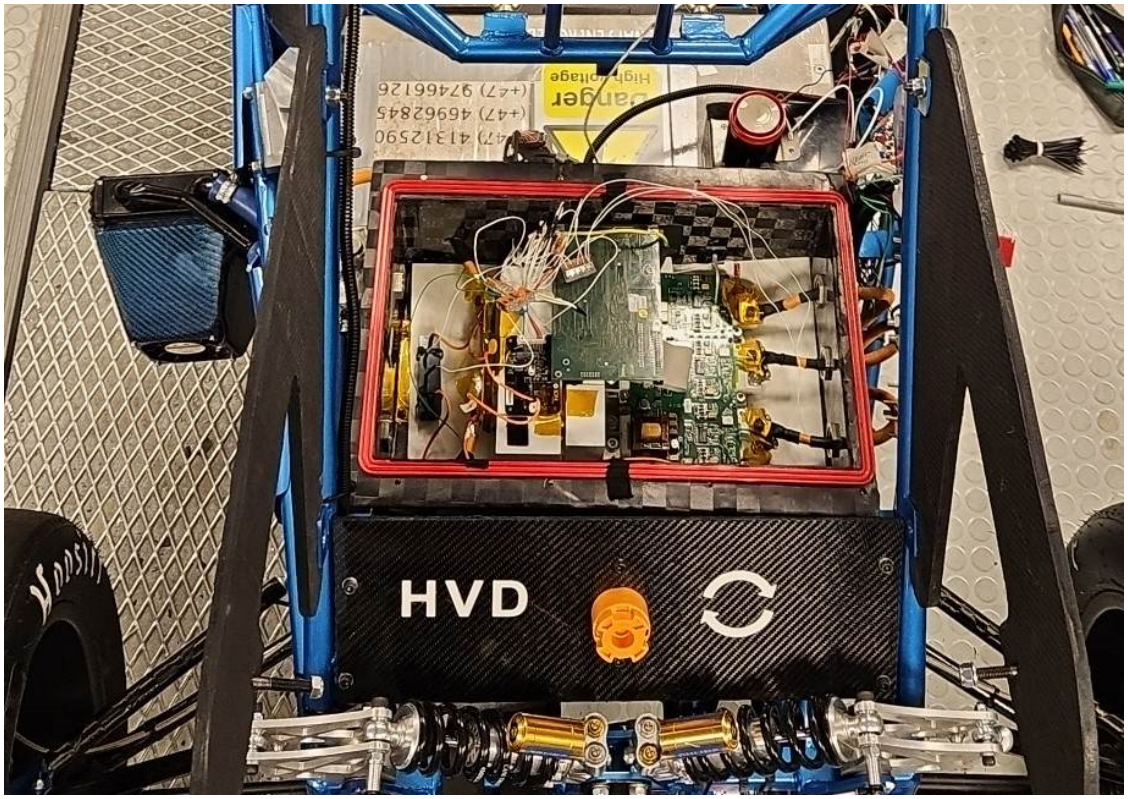


Figure 1.3: Placement in the car

The box is made by the mechanical team of ION Racing and will be made of carbon fiber. The box will have a heatsink attached to the underside as seen in figure 1.4.

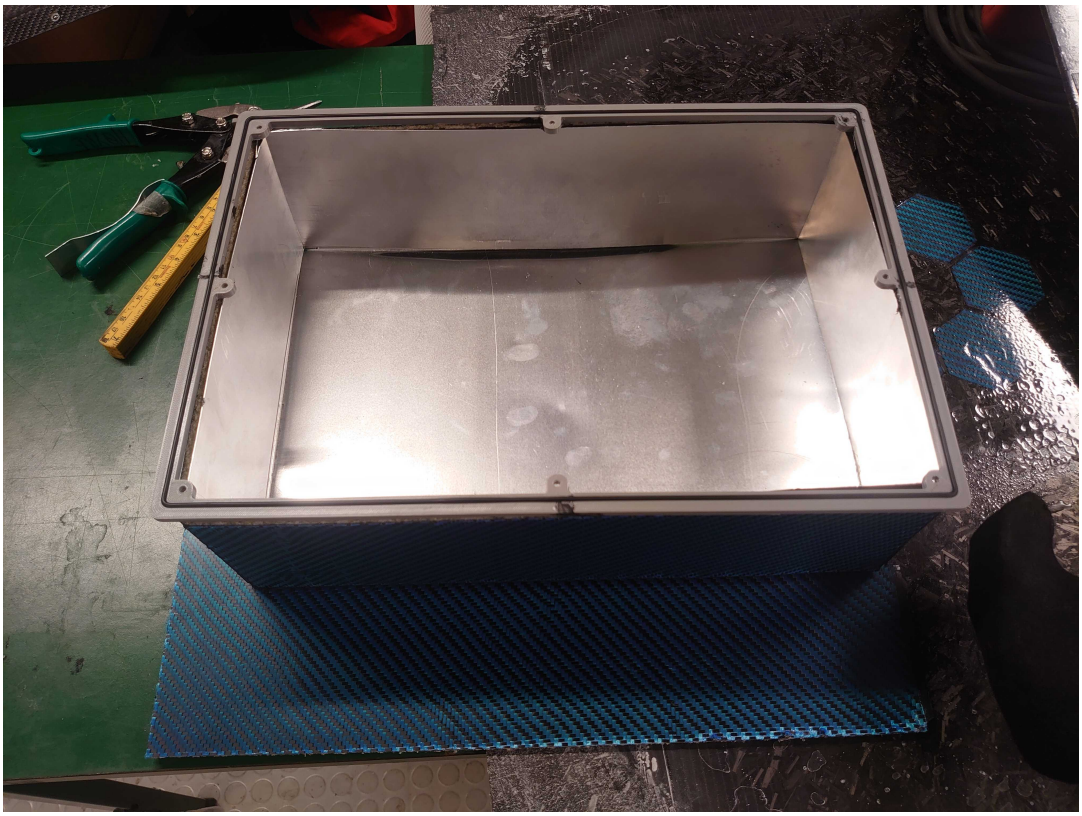


Figure 1.4: The incomplete motor controller box

1.7 FSUK Rules

Formula student UK has their own rules that are independent of the general Formula Student rules.

Here are a list of some of the relevant ones. All text in cursive has been directly copied from the FSUK rules. [5]

- **T11.9 System Critical Signal**
  System critical signals are signals that influence actions on the shutdown circuit, influence wheel torque, critical LED indicators, Tractive System Active Light, and the Isolation monitoring device.
  These are mostly handled by the ECU but will still influence the final design.

- **EV1.1 Tractive System**

  - ***EV1.1.1*** *Tractive System (TS) – every part that is electrically connected to the*

motor(s) and TS accumulators.
This rule dictates what is considered a part of the TS.

- **EV2.2 Power Limitation**

  - ***EV2.2.1*** *The TS power, measured by the Energy Meter, must not exceed +80 kW for two (2) wheel drive vehicles or +60 kW for four (4) wheel drive vehicles.*
    This rule is one that will influence how many motors the controller will control in 2024.

- **EV3.2 Overcurrent Protection**

  - ***EV3.2.1*** *All electrical systems must have appropriate overcurrent protection.*

  - ***EV3.2.4*** *All overcurrent protection devices must be rated for the highest voltage in the systems they protect. All devices used must be rated for DC.*
    These rules will be taken into consideration in the design.

  - ***EV3.2.6*** *The overcurrent protection must be designed for the expected surrounding temperature range but at least for 0°C to 85 °C.*
    Parts that are rated within these requirements will be selected.

- **EV4 Tractive System**

  - ***EV4.1.2*** *All components in the TS must be rated for the maximum TS voltage.*
    The MCU is not directly connected to the motors or the TS Accumulators and according to EV1.1 is not considered part of the TS.

  - ***EV4.1.3*** *All components must be rated for the maximum possible temperature that may occur during use.*
    Our components are chosen to be rated within required temperatures, and if the system gets too hot, it will shut down as a safety feature.

- **EV6 EV Shutdown Circuit and Systems**
  The shutdown circuit is a safety measure that has to be fulfilled and documented properly. These are handled by their respective systems. There is no direct requirement for the MCU to shutdown.

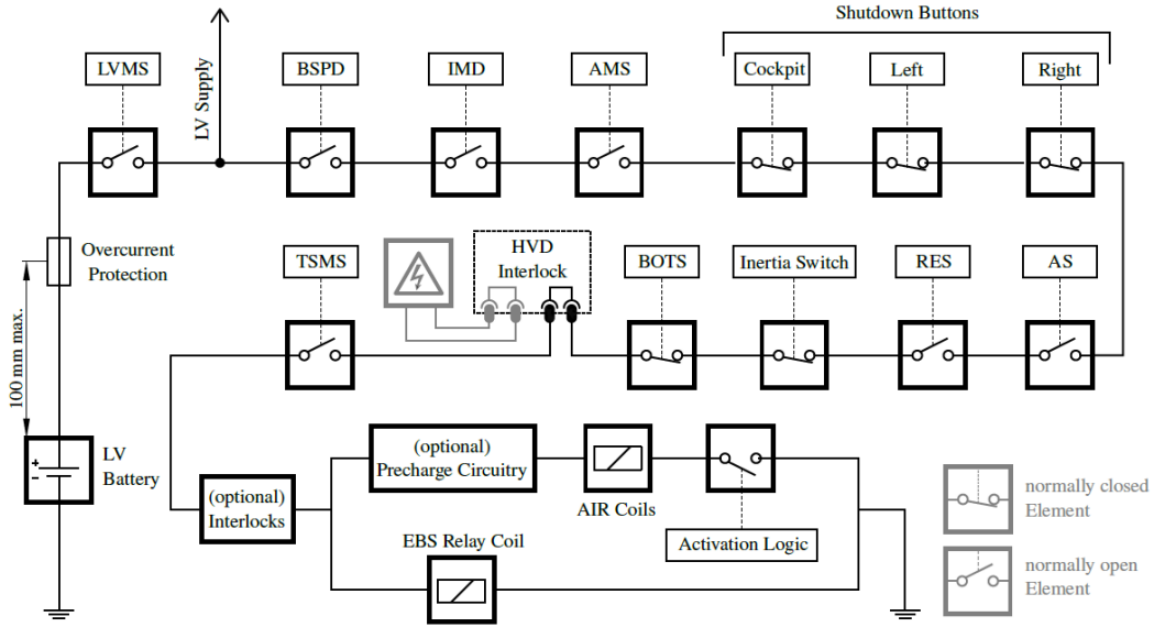


Figure 1.5: An illustration of the shutdown circuit

Chapter 2

Theory

Before going into detail about the construction of the motor controller, some theory is required. This chapter will cover useful material on electric motors, torque generation and how torque can be controlled by using various methods. It also includes the implementation of these methods in a microcontroller.

2.1 Electric motors

There are numerous types of electric motors and each type has it's strengths and weaknesses. The type to use all depends on the specific application. Electric motors used in electric vehicles are typically:

- AC induction motors
- Switched reluctance motors (SRM)
- Brushless permanent magnet synchronous motors (PMSM)
- Brushless DC motors (BLDC)

This section of the chapter will focus on brushless permanent magnet synchronous motors as ION Racing utilizes such a motor, the EMRAX 228. Information regarding the EMRAX 228 will be covered in its own section, including it's technical specifications.

2.1.1 Permanent Magnet Synchronous Motors

Permanent magnet synchronous motors (PMSM) are known for being highly efficient and providing a great amount of power for their size. Unlike motors with electromagnets, permanent magnet motors do not require external energy to be magnetized and maintain their magnetic field which makes them highly efficient. This type of motor consists of a rotor where the permanent magnets (poles) are attached and a stator with electromagnetic coils as illustrated in figure 2.1.

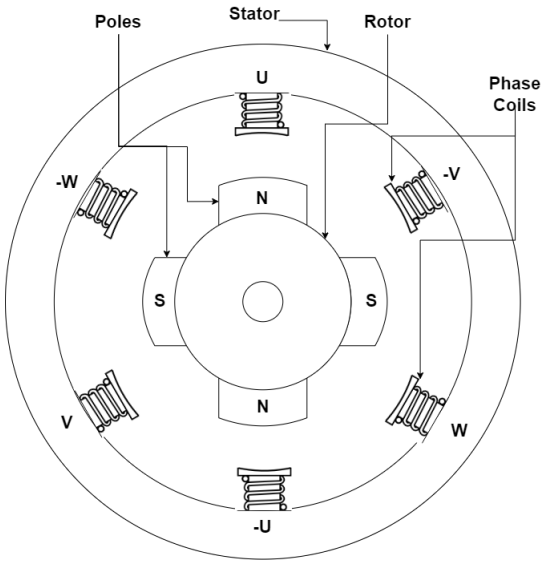


Figure 2.1: Permanent magnet motor with 2 pole pairs

When current is applied to the stator coils they create a rotating magnetic field. This force interacts with the magnetic poles of the rotor which creates torque which causes the rotor to spin. PMSM motors operate at a fixed speed synchronized with the frequency of the power supply. In other words the rotor rotates at the exact same speed as the magnetic field of the stator. This is essential for applications where precise speed control is required and is the reason this type of motor is ideal in electric vehicles.

This is a three-phase motor which means that it is supplied with alternating current with phases U, V and W. Each phase has its corresponding phase connection, respectively -U, -V and -W, which are internally connected to one another. They are labeled with a minus sign as current flows in the negative direction opposed to their equivalent connection. This means that the flow of current through U is in the opposite direction of the flow of current in -U. So when current is applied to these coils, the magnetic fields generated have opposite directions, which is necessary for the fields to combine and create the rotating magnetic field.

Phase configuration

The phase connections in motors can either be connected in a delta-configuration or a star-configuration. Figure 2.2 shows how the two different connections are made between the coils. The main difference between them is the amount of power supplied by the three phases.

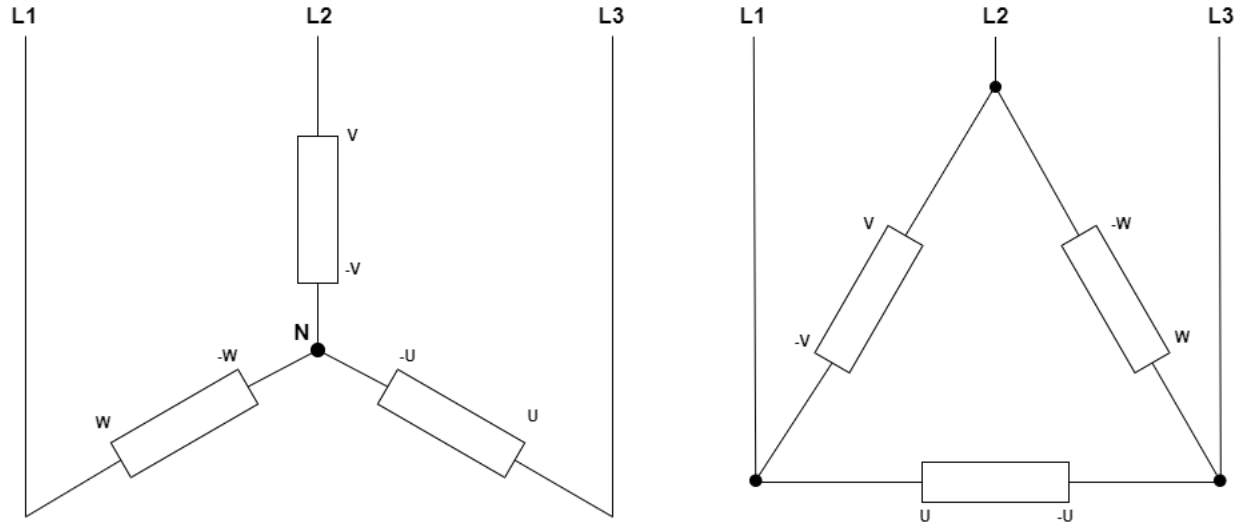


Figure 2.2: Star and delta configurations

The total power supplied in a star connection is calculated by:

$$P = 3 \cdot V_{PH} \cdot I_{PH} \cdot cos\phi = \sqrt{3} \cdot V \cdot I \tag{2.1}$$

and the total power supplied in a delta connection:

$$P = 3 \cdot V_{PH} \cdot I_{PH} \cdot cos\phi = 3 \cdot V \cdot I \tag{2.2}$$

In delta connected motors the power is higher due to each phase voltage being equal the total line voltage. They provide a higher torque, but also require higher current which can cause instability during startup. The voltage of each phase in star connected motors however is $\frac{1}{\sqrt{3}}$ of the total line voltage. Star connected motors have more balanced currents and can operate under normal conditions without overheating. Compared to motors with a delta-configuration, the star-configuration is preferred for applications such as driving a car over longer distances. This is one of the reasons ION Racing uses the EMRAX 228, a star-configurated motor.

2.1.2 EMRAX 228

The motor that will be utilized and modulated in this thesis is the EMRAX 228 by EMRAX, a Slovenian company who manufactures and develops electric motors. The 228 is well suited for automotive and airplane applications due to it's powerful 124 kW peak output and it being compact and lightweight at only 12,9 kg.

Figure 2.3: EMRAX 228 as shown in [[4], Figure 8, Page 8]

Motor	Axial flux permanent magnet synchronous electric motor	**Operating Voltage**	50 - 710 V
Weight	12,9 - 13,5 kg	**Peak / Continous Power**	124 kW / 75 kW*
Cooling	Air / Water / Combined	**Peak / Continous Torque**	230 Nm / 130 Nm*
Diameter / Length	228 mm / 86 mm	**Maximum Speed**	6500 RPM

Table 2.1: Mechanical & electrical EMRAX 228 data from [[3], Page 1]

2.2 Torque modulation

To modulate the torque of the EMRAX 228 a technique called pulse width modulation (PWM) will be utilized. With PWM the power supply to the motor will be switched on and off in pulses. With a voltage supply of 400 V any voltage output between 0 V and 400 V can be achieved by varying the width of these pulses.

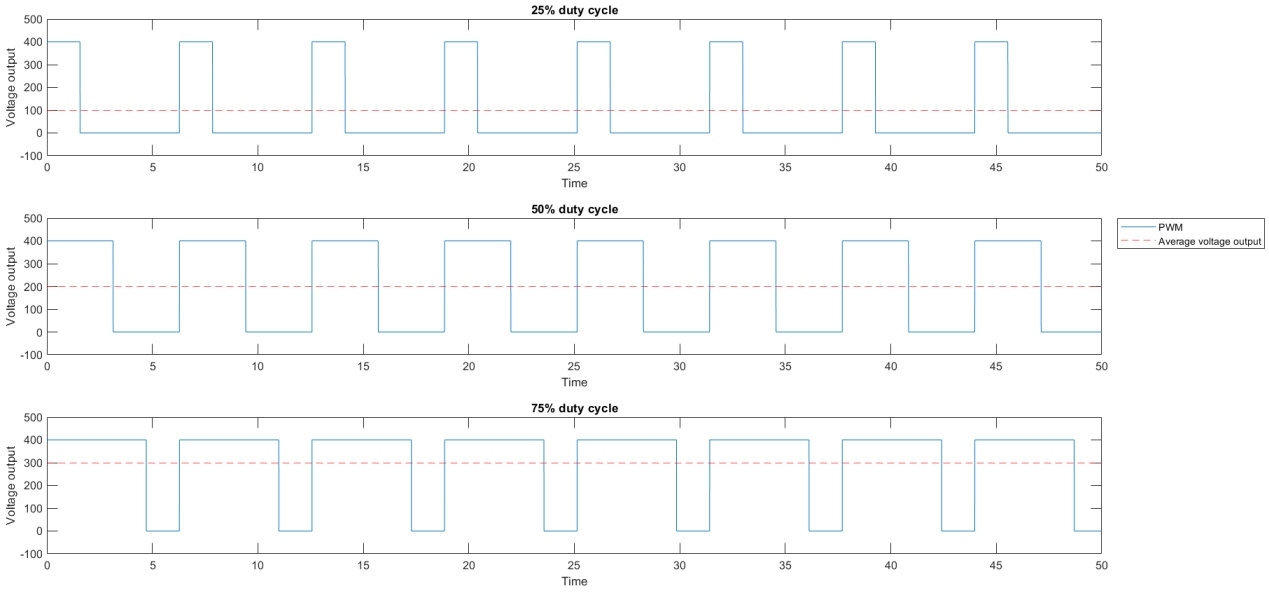


Figure 2.4: Illustration of duty cycles

The figure above (2.4) shows some fundamental duty cycles. To calculate the average voltage output of different duty cycles equation 2.3 is used.

$$V_{rms} = V_{amplitude} \cdot dutycycle \tag{2.3}$$

With a 400 V source the average output for these fundamental duty cycles are as follows: A 25% duty cycle leads to a 100 V average output, a 50% duty cycle leads to a 200 V average output, while a 75% duty cycle leads to a 300 V average output. Solving equation 2.3 for the duty cycle, the duty cycle for a desired average voltage output can be found.

2.3 Positional Detection of the Motor Shaft

Knowing the rotation of the motor shaft is one of the most useful pieces of information to know. Knowing the position allows the MCU to precisely control the motor. Precise control of the motor allows for much greater control of motor speed and torque, which is critical for efficient operation.

There are two main ways of detecting the position of the motor shaft, using an Encoder or a Resolver. These components both have their strengths and weaknesses. In the case of a motorized vehicle going with a resolver is the better choice, as encoders usually are a lot more sensitive to shocks and vibrations which the car would generate a lot of.

2.3.1 Resolver Theory

A resolver is an electrical device that is used to measure the angular position and velocity of a rotating motor shaft. It operates based on electromagnetic induction, where a coil is used to excite a secondary coil that is fixed to the motor shaft. This allows for the resolver to detect angular position and output them as two signals, one *sin* and one *cos*.

With the sine and cosine signals the MCU can use the unit circle 2.5 to determine exactly where in a rotation the motor shaft currently is.

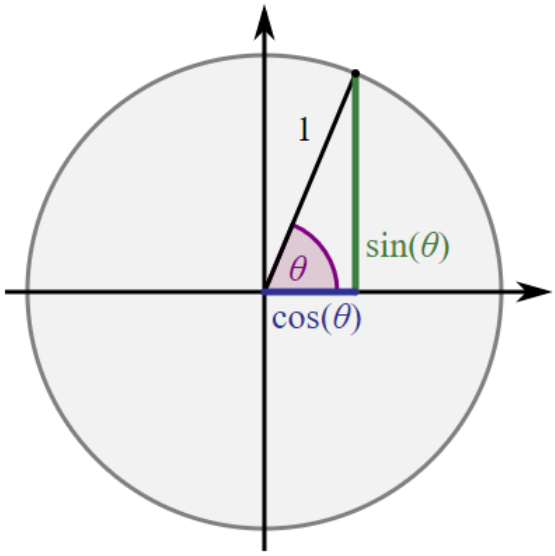


Figure 2.5: Illustration of a Simple Unit Circle

From this information the MCU can apply a inverse trigonometric function for example the following equation(2.4) to calculate the angle.

$$\alpha = atan(\frac{cos}{sin}) \tag{2.4}$$

2.3.2 Sensorless detection of the motor shaft position

Due to availability issues during testing, a sensorless detection method had to be implemented temporarily. A common way to achieve this is reading the back EMF (BEMF) from the motor phases. These levels will then be used in conjunction with a 6 step commutation algorithm. The back EMF readings will inform the algorithm when to move on to the next step in the 6 step sequence. The 6 step sequence is displayed in table 2.2.

Commutation step	Phase A	Phase B	Phase C
1	High	Low	-
2	High	-	Low
3	-	High	Low
4	Low	High	-
5	Low	-	High
6	-	Low	High

Table 2.2: Illustration of the sensorless commutation sequence

When the rotor in the motor is induced by the electrical fields created by the motor windings, a magnetic field resisting the induced change is created. This magnetic field induces a current back through the windings. This current is referred to as the back EMF, counter EMF, or BEMF, and the magnitude of it will vary for each winding based on rotor position.[11]

Figure 2.6 shows the output from the motor while it is not being supplied and manually rotated. This signal is technically not back EMF as the rotor is not being excited by an electrical field, but the relevant phase should look similar. As seen in table 2.2, at each step in the commutation sequence there is one phase that is switched off, this phase is referred to as the "floating phase". The algorithm will look for zero-crossings on the floating phase as the signal to move to the next step in the commutation sequence.

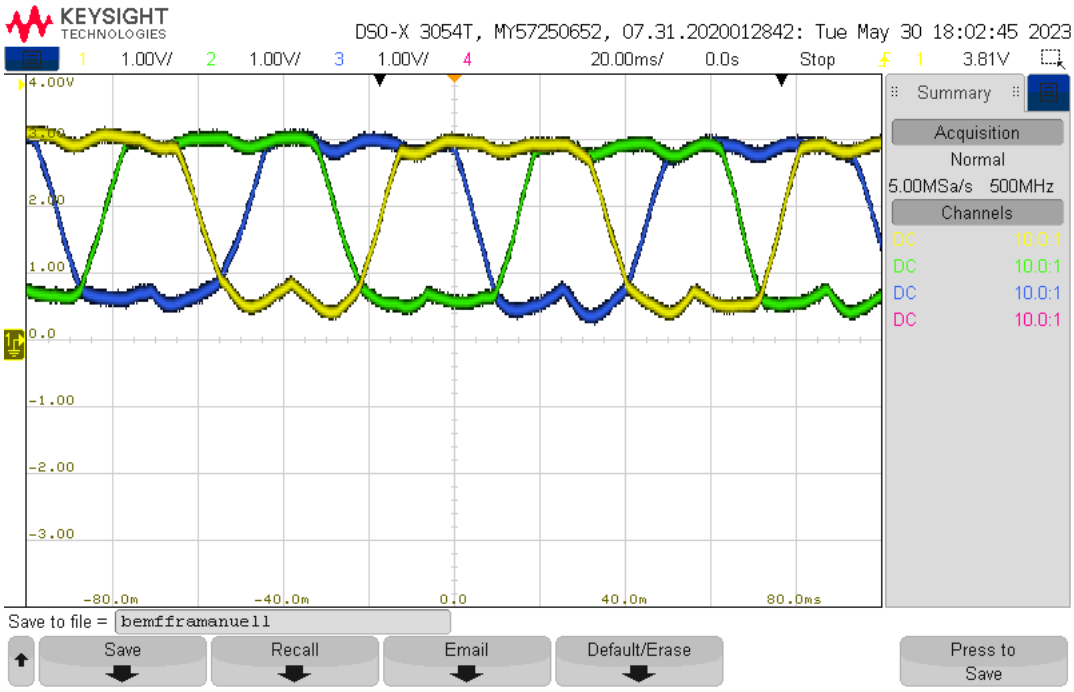


Figure 2.6: Measured voltage from manually rotated motor

An illustration of how the commutation sequence would be commuted from the back EMF readings is shown in figure 2.7. Again these are not actual back EMF signals as the motor was rotated manually. If these were actual back EMF signals from a running motor the floating phase would have more noise, while the two active phases would look completely different. If table 2.2 and figure 2.7 is compared the algorithm can be seen. From the table during step 3 the floating phase is phase A, and from the figure it can be seen that it is when phase A crosses the zero line that the sequence commutates to step 4. In step 4 the floating phase is phase C, so when phase C crosses zero the sequence commutates to step 5. These steps will then be cycled through as long as needed.

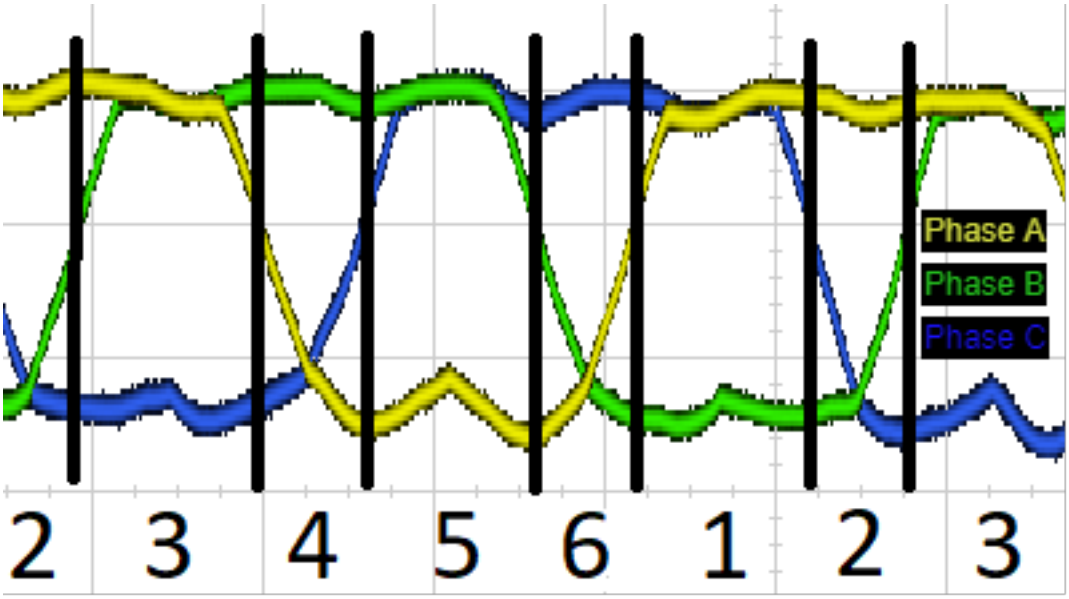


Figure 2.7: Commutation sequence from hypothetical back EMF readings

2.4 Implementation in a microcontroller

The resolver that is planned to be used is the MLX90380. The MLX90380 is a monolithic contactless sensor IC sensitive to the flux density applied orthogonally and parallel to the IC surface. [9]

The MLX90380 outputs its signals as a percentage of V_{dd} where for an example $sin \approx 1$ is reached at 90% of V_{dd} and $sin \approx -1$ is reached at 10% of V_{dd} as shown in figure 2.8.

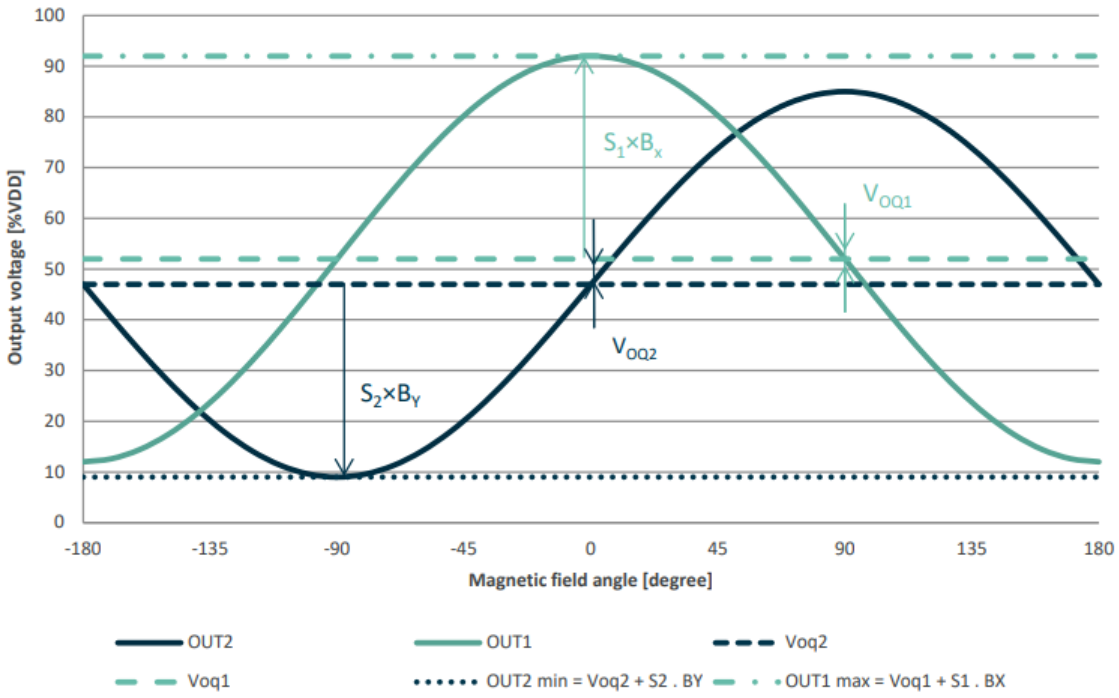


Figure 2.8: Output characteristics plot taken from the MLX90380 datasheet [9]

Using ADCs to measure these voltages allows the MCU to calculate the position and velocity of the motor.

Chapter 3

Software and testing

In order to design and eventually produce a well functioning motor controller, it is essential to test the system with development boards. This chapter covers the setup of the early motor controller system and the software used to operate the motor.

3.1 Testing setup

The setup of the early system consists of the following boards:

- **Micro controller development board**
- **Gate driver demonstration board**
- **Expansion board**

The micro controller and gate driver were purchased at the beginning of this thesis. After some evaluation an in-house expansion board was designed and produced at the university.

3.1.1 Micro controller

The micro controller used for testing is the *NUCLEO-H723ZG* micro controller (μC) development board from STMicroelectronics as depicted in figure 3.1.

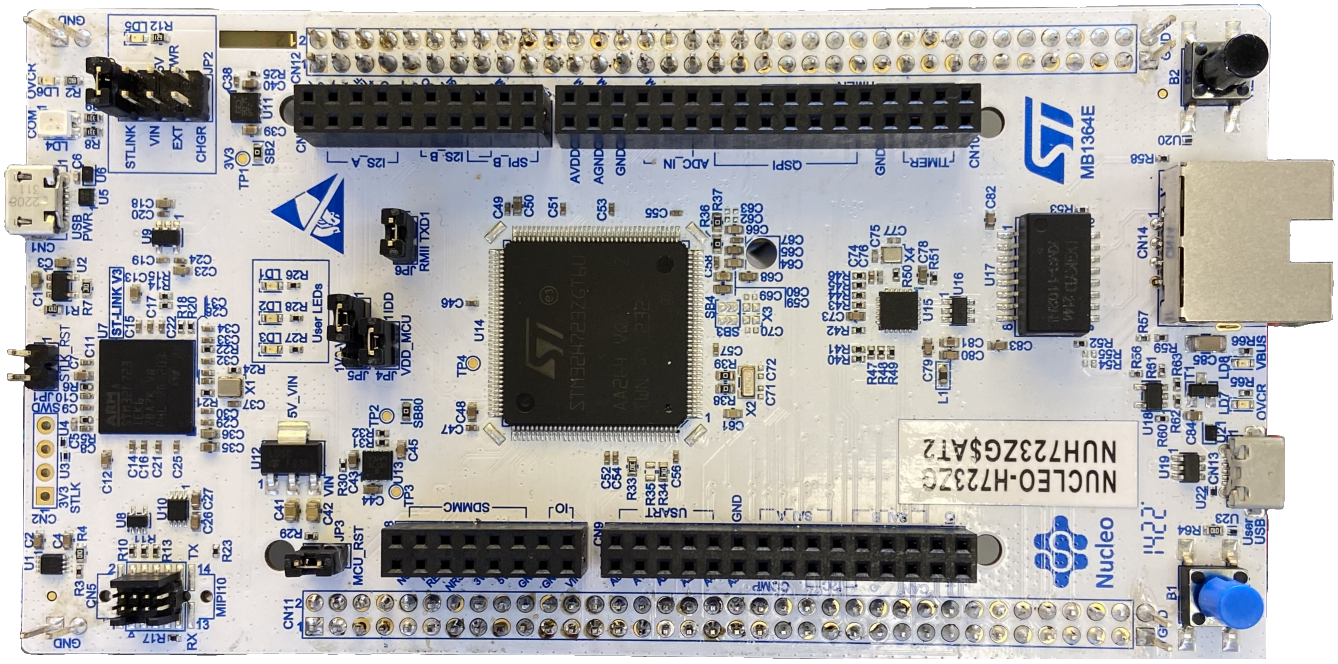


Figure 3.1: NUCLEO-H723ZG development board

The *NUCLEO-H723ZG* unit's main goal is to control the EMRAX228 by sending pulse width modulation signals to the gate driver board.

3.1.2 Gate driver

The pulse width modulation signals from the μC are received by the *EVALSTDRIVE601*, a three phase power board from STMicroelectronics. This board has been chosen due to compatibility and ease of connection by using a 34-pin ribbon cable on the J4 connector (marked in red) as shown in figure 3.2.

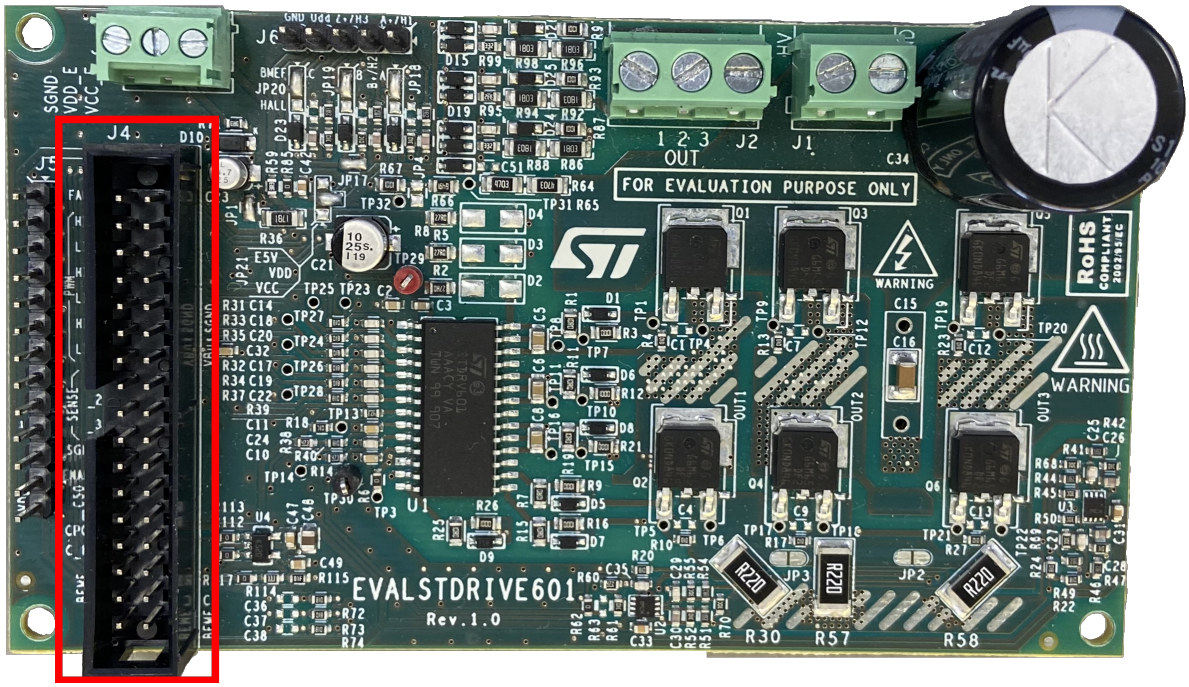


Figure 3.2: EVALSTDRIVE601 demonstration board

The board is suitable for a six step algorithm or field oriented control and allows driving PMSM motors, such as the EMRAX228. Additional functions of the *EVALSTDRIVE601* are sensing the phase currents, back EMF sensing and over-current protection.

3.1.3 Expansion board

In order to establish the required connections between the μC and the gate driver, a expansion board has been constructed (see figure 3.3). The expansion board has been designed in Altium Designer[1] and has been milled out and soldered in the laboratory at UiS.

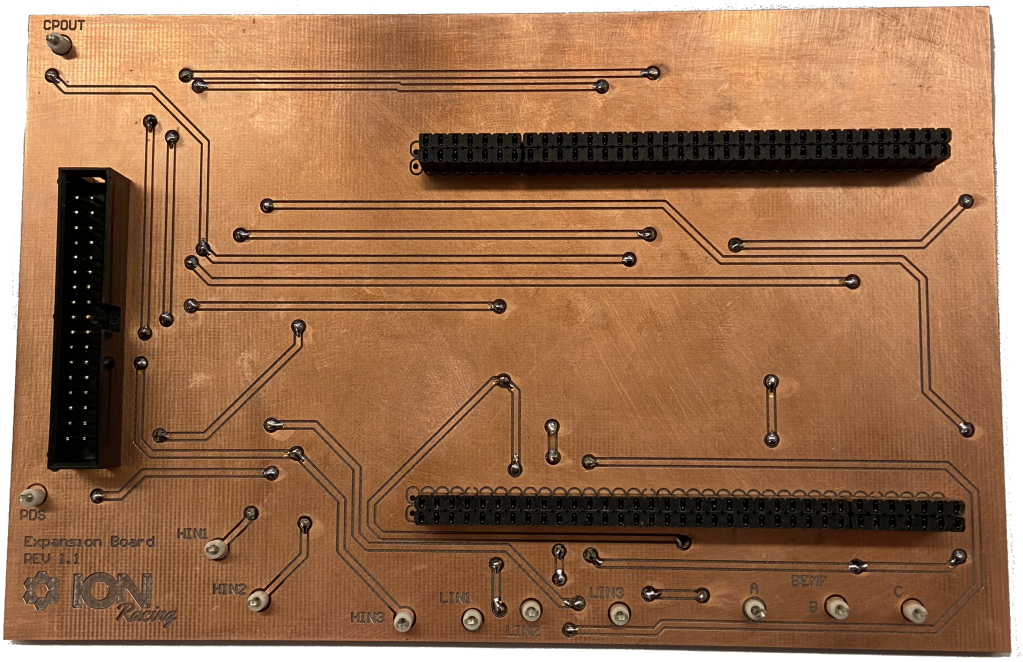


Figure 3.3: Expansion board

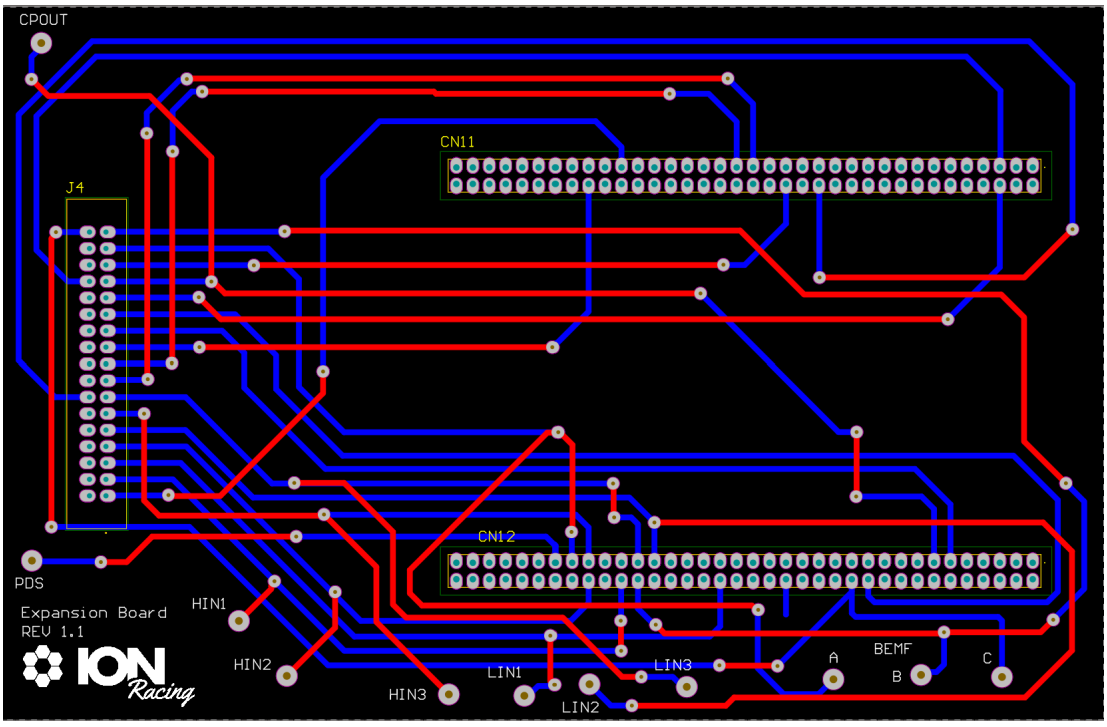


Figure 3.4: Expansion board in Altium

A overview of all signals from the micro controller to the gate driver via the expansion board are shown in table 3.1.

EVALSTDRIVE601			NUCLEO-H723ZG
Pin J4	Name	Description	Pin uC
1	FAULT	Driver FAULT signal (high when signal is active)	PF1
2	GND	Ground	GND
3	HIN1	High input transistor U	PE9
4	GND	Ground	GND
5	LIN1	Low input transistor U	PE8
6	GND	Ground	GND
7	HIN2	High input transistor V phase	PE11
8	GND	Ground	GND
9	LIN2	Low input transistor V phase	PE10
10	GND	Ground	GND
11	HIN3	High input transistor W phase	PE13
12	GND	Ground	GND
13	LIN3	Low input transistor W phase	PE12
14	VBUS	Bus Voltage	PA0
15	SENSE_1	Current sensing in phase U	PC2_C
16	GND	Ground	GND
17	SENSE_2	Current sensing in phase V	PC3_C
18	GND	Ground	GND
19	SENSE_3	Current sensing in phase W	PF9
20	GND	Ground	GND
21	GPIO_BEMF	Divider enable for BEMF sensing	PA5
22	GND	Ground	GND
23	ENABLE	Driver enable signal from uC	PB2
24	GND	Ground	GND
25	E5V	External 5V for Hall power supply	5V_EXT
26	NC	Not connected	-
27	CPOUT	Current comparator output signal to uC	PA6
28	VDD	Supply voltage	3V3_VDD
29	C_REF	Current reference signal from uC	PA4
30	GND	Ground	GND
31	BEMF A	BEMF output A signal to uC	PF13
32	GND	Ground	GND
33	BEMF B	BEMF output B signal to uC	PF14
34	BEMF C	BEMF output C signal to uC	PB1

Table 3.1: Signals between μC and gate driver

The entire testing setup with all three mentioned boards and the rest of equipment used can be seen in the figure below(3.5). The EMRAX 228 and the displacement sensor were acquired from ION Racing. The remaining equipment used for testing were available for use in the laboratory at the University of Stavanger.

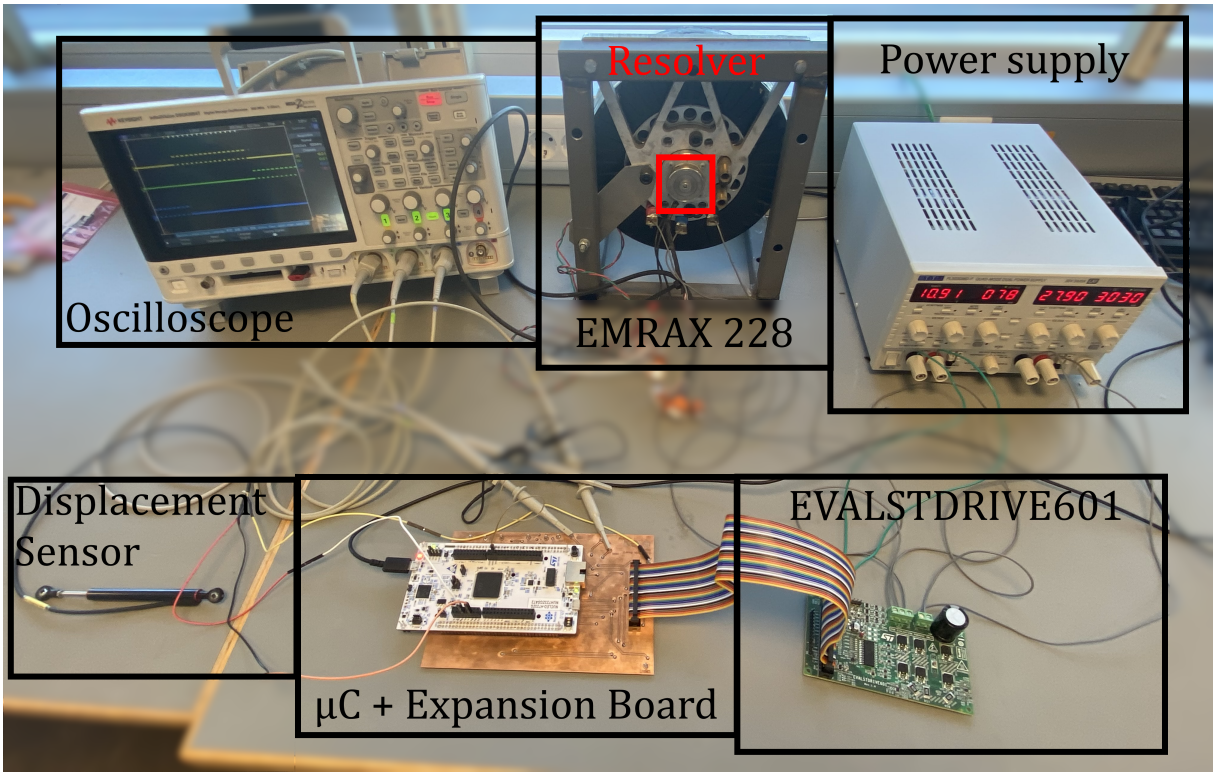


Figure 3.5: Entire testing setup

The EMRAX 228 used during the testing period was equipped with a resolver. The resolver is an outdated model and requires a specific RDC. The resolver could unfortunately not be used as the manufacturer of the resolver no longer produces the compatible model of the RDC.

3.1.4 Software

When choosing a micro controller the original plan was to find one compatible with the software "Motor Control Workbench" by STMicroelectronics. Due to availability issues at the time, the NUCLEO-H723ZG was chosen as it was the only one available. Unfortunately the NUCLEO-H723ZG is not supported by motor control workbench, meaning the software had to be written from manually.

This proved to be challenging for a number of reasons. First was the phase-shifting of the three phases. The plan was to set the timer in slave mode and its channels in output compare mode. Each channel would also have a complementary output which always outputs the opposite signal of the channel (see figure 3.6) .

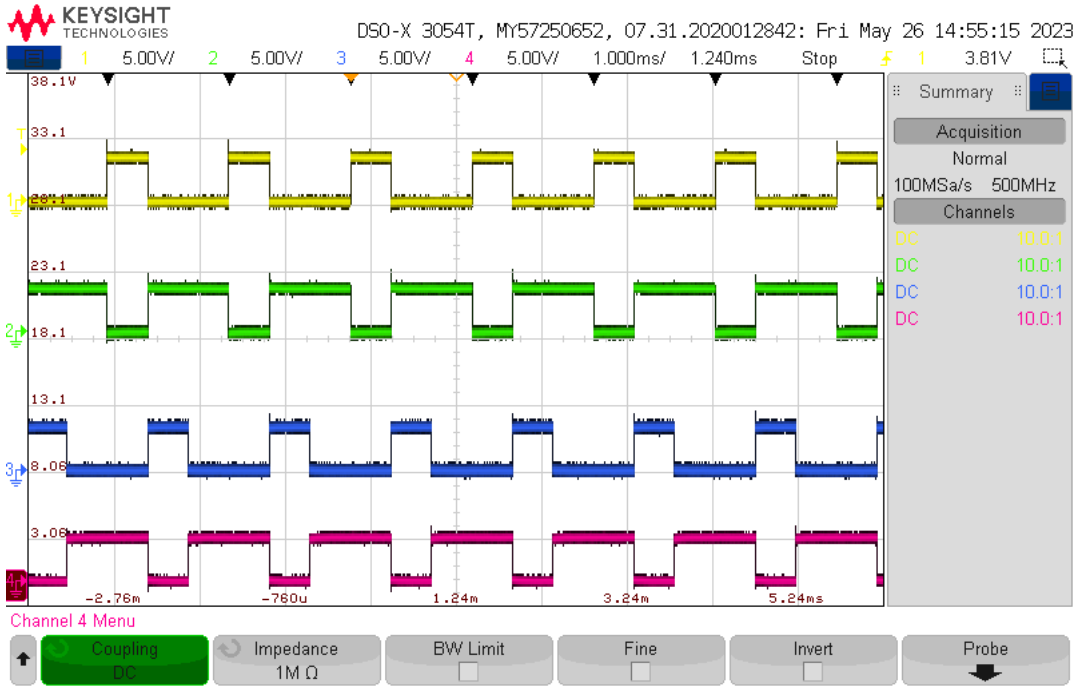


Figure 3.6: Example of complementary channels, where yellow and green are complementary, and blue and purple are complementary

With this setup the phase shift and duty cycle could easily be set by altering the Capture Compare Registers (CCR's) of the channels. The value in a channels CCR is what the slave timer compares to the master timer. When the master timer has counted up to the CCR value the channels output toggles between low and high. On the NUCLEO-H745 each channel has two CCR's, one for CHx and one for the complementary output CHxN. The value in CHx's CCR would instruct when to output a high signal on CHx, also forcing the output on CHxN to a low signal. While the value in CHxN' s CCR would instruct when to output a high signal on CHxN, also forcing the output on CHx to a low signal.

The phase shift would then be set by spacing out the values of the three CHx CCR's, letting the different channels be turned on at different times. The duty cycle would be set by changing the three CHxN CCR's, forcing the CHx signals to a low signal when desired.

Unfortunately the NUCLEO-H723ZG only has one CCR for each channel, meaning the complementary signal on the NUCLEO-H723ZG is a passive signal that only follows the channels ouput and reverses it. This means that on the NUCLEO-H723ZG this method can still achieve a phase shift, but would be locked to a 50% duty cycle. The output with this setup would be toggled between high and low every time the master counter reaches the CCR value of the channel. Duty cycle control could still be achieved by changing the CCR value back and forth for every pulse, but this method would be inefficient.

Instead an alternate method was implemented. By putting the channels in regular PWM mode the context for the CCR value changes. The channels CCR value is compared to the clock counter to check if its higher or lower. With polarity mode set to low, if the CCR value is higher than the count the channel outputs a low signal, and if it is lower than the count the channel outputs a high signal. To set a duty cycle the CCR value would now just be set as the desired percentage of the max counter value. All phases still run at the same time however. To achieve the phase shift a separate timer was set up with a global interrupt that changes the CCR values of the channels to turn them on or off. With this method both phase shift and duty cycle control is demonstrated in figure 3.7 and 3.8. The duty cycle here is managed by the displacement sensor, by reading the ADC and changing the CCR values.

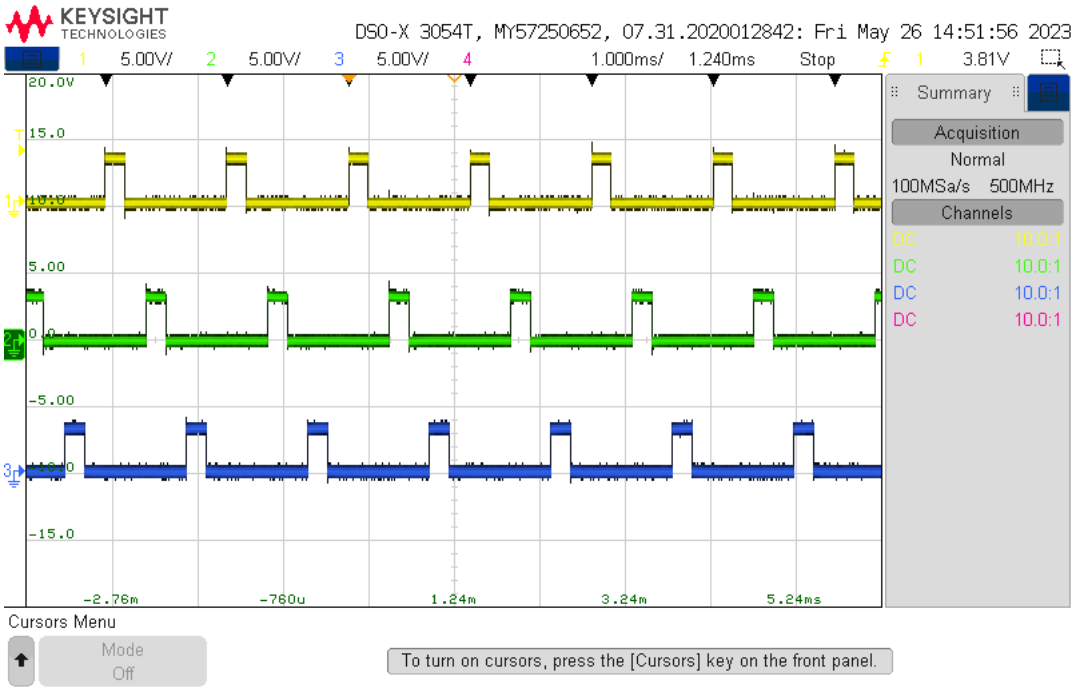


Figure 3.7: Phase A, B and C's high output (yellow, green and blue respectively) with a moderate duty cycle

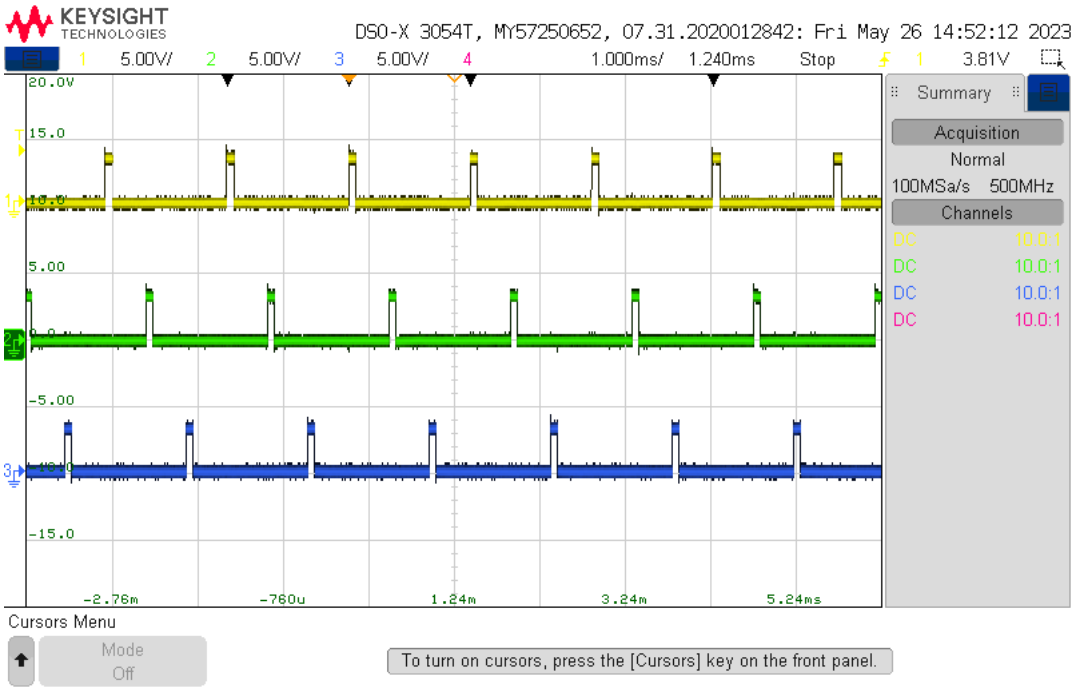


Figure 3.8: Phase A, B and C's high output (yellow, green and blue respectively) with a lower duty cycle

Due to the passive nature of the complementary channel on the NUCLEO-H723ZG the sensorless commutation sequence (see table 2.2) could not be done with a single timer. At least not without interference from other phases. To make the commutation sequence work with the NUCLEO-H723ZG during testing a second timer was also used for the PWM signals. With this configuration timer 1 will have three channels representing HIN1, HIN2 and HIN3. While timer 8 have three channels representing LIN1, LIN2 and LIN3. Figure 3.9 shows HIN1 (yellow), LIN1 (green), HIN2 (blue) and LIN2 (purple) with this configuration. The logic of this configuration is also illustrated as flowcharts in figure 3.10 and 3.11. The full code for these flowcharts is shown in appendix A.1 and A.2 respectively.

Figure 3.9: Readings from 2 of the phases from top to bottom: HIN1, LIN1, HIN2 and LIN2

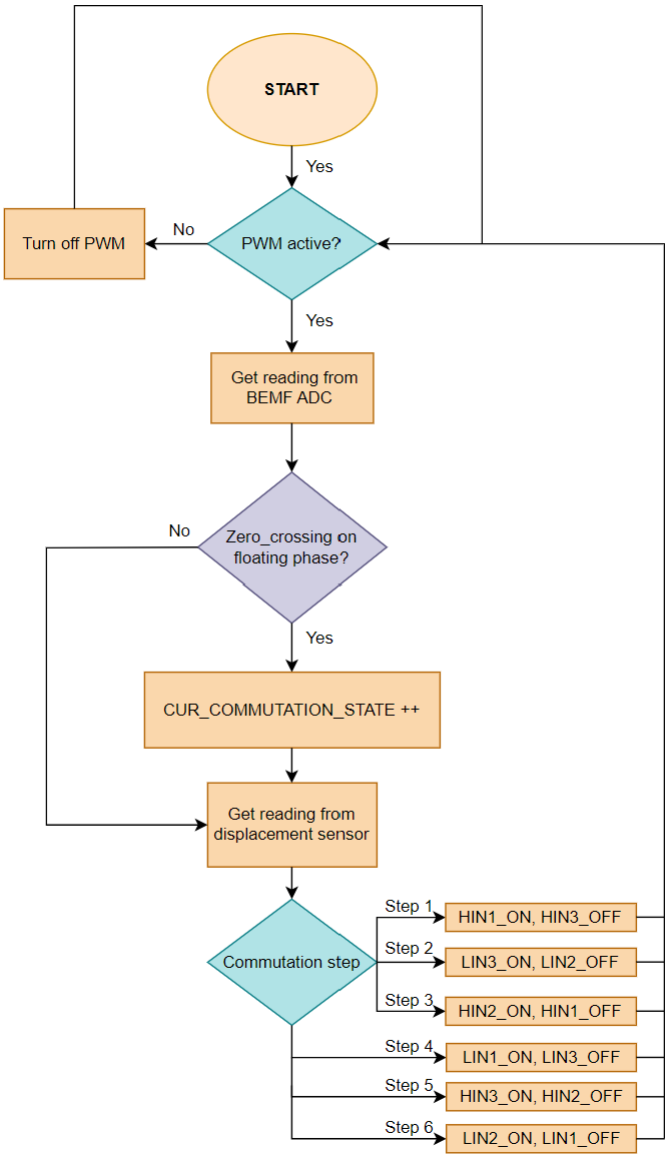


Figure 3.10: Flowchart for the main.c logic in the project (full code in appendix A.1)

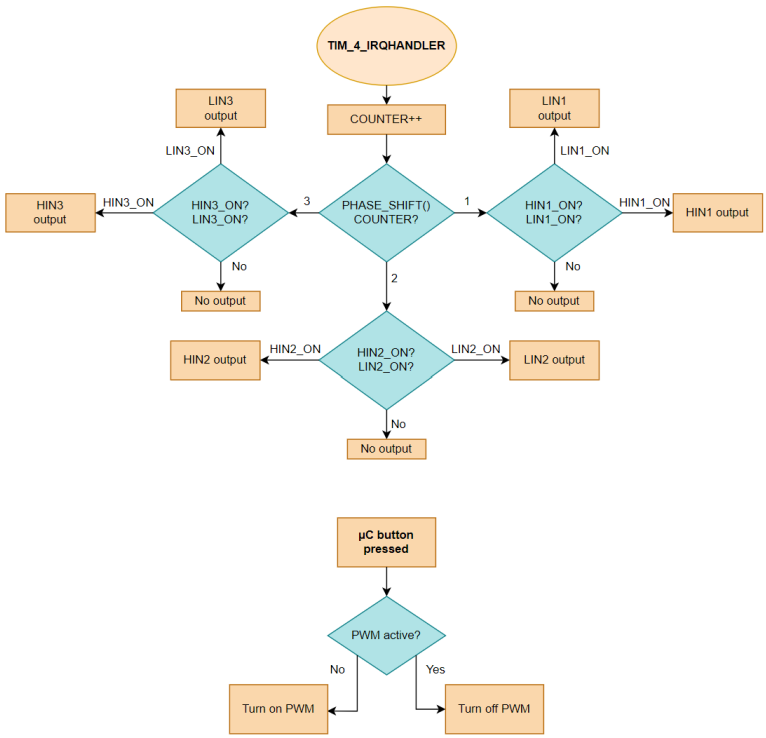


Figure 3.11: Flowchart for the interrupts in stm32h7xx_it.c (full code in appendix A.2)

For the back EMF values to be usable it requires the motor to already be rotating at a certain speed. This would be done during a startup sequence. This is another aspect that was planned for the "Motor Control Workbench" software to handle. Unfortunately the startup algorithm is typically proprietary knowledge among MCU manufacturers. During this startup phase of the program the back EMF values and other motor parameters used is unreliable even with simple motors with only 3 windings. The Emrax-228's 10 pole pairs make them even less readable. Figure 3.12 shows HIN1 (yellow) aswell as the three back EMF readings, A (green), B (blue) and C (purple) when the power supply is switched on.

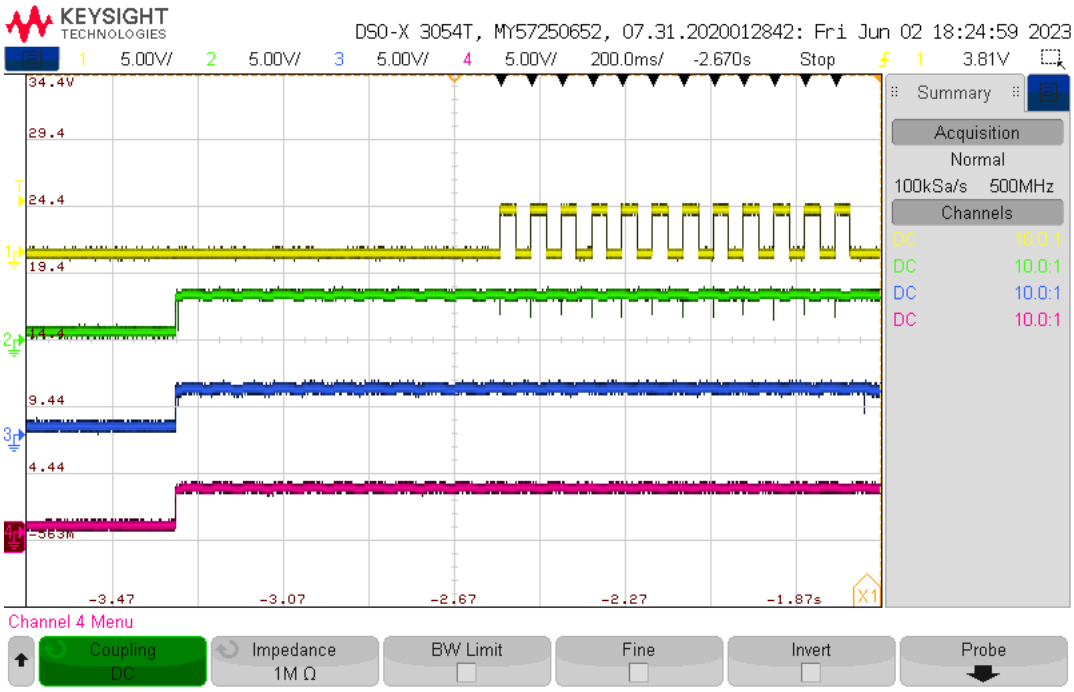


Figure 3.12: Readings from HIN1 (yellow) and the three back EMF readings, A (green), B (blue) and C (purple)

Figure 3.12 above shows that all the back EMF readings jumps to and stay at or close to their maximum level when the power supply is switched on. This nullifies the algorithm and the commutation sequence would be stuck at a certain step.

Chapter 4

Controller Design

This chapter focuses on the design of the motor controller, providing a detailed analysis of the chosen components and examination of the schematics. Additionally, the controller's PCB design will also be presented, including the design decisions made.

4.1 Component choices and schematics

To minimize the manual workload, the PCBs and most surface mounted components will be produced and mounted by the PCB-Producer JLC PCB. Therefore whether the components are in stock heavily weighs on the components selected. Some components are chosen because ION Racing has them in stock.

4.1.1 The controller STM32H723ZGTx

The controller used is the *STM32H723ZGTx*. The controller was selected based on the following requirements:

- **Sufficient amount of timers**
  In order to control a motor you need 1-2 timers depending on your setup. The controller has two advanced-control timers, twelve general-purpose timers, two basic timers, five low-power timers, two watchdogs and a SysTick timer [12]. An Advanced-control timer can usually control 1 motor by itself using 6 channels to generate the needed PWM signals.

- **Sufficient amount of ADC channels**
  There are a lot of different sensor readings that could be beneficial to measure and process in order to control a motor more efficiently. The controller has two 16-bit ADCs with up to 18 channels, and one 12-bit ADC with up to 12 channels. Several of these channels can be set to differential mode.

- **A high clock speed.**
  Having a high clock speed is beneficial when controlling multiple motors because it allows the microcontroller to perform more calculations and execute more instructions in a shorter amount of time. A higher clock speed can improve the precision of motor control signals by allowing the microcontroller to generate more accurate PWM signals.

- **In Stock at JLC PCB**
  Being in stock is a massive bonus as this cuts down the amount of soldering work and logistical issues stemming from having to source the part separately.

These points make the STM32H723ZGTx a solid choice. The microcontroller has a 144 pin package which is what is used in this thesis. According to the datasheet, the controller boots from flash if BOOT0 is connected to ground. Connecting a crystal oscillator to its respective pins (PC14 and PC15) was considered, but the internal oscillators were determined to be robust enough for the needed use-cases. Several pins have been made ready to control LEDs but later decisions caused these plans to be scrapped. This was due to the inaccessible location the card would be placed in would not allow for the LEDs

to be seen.

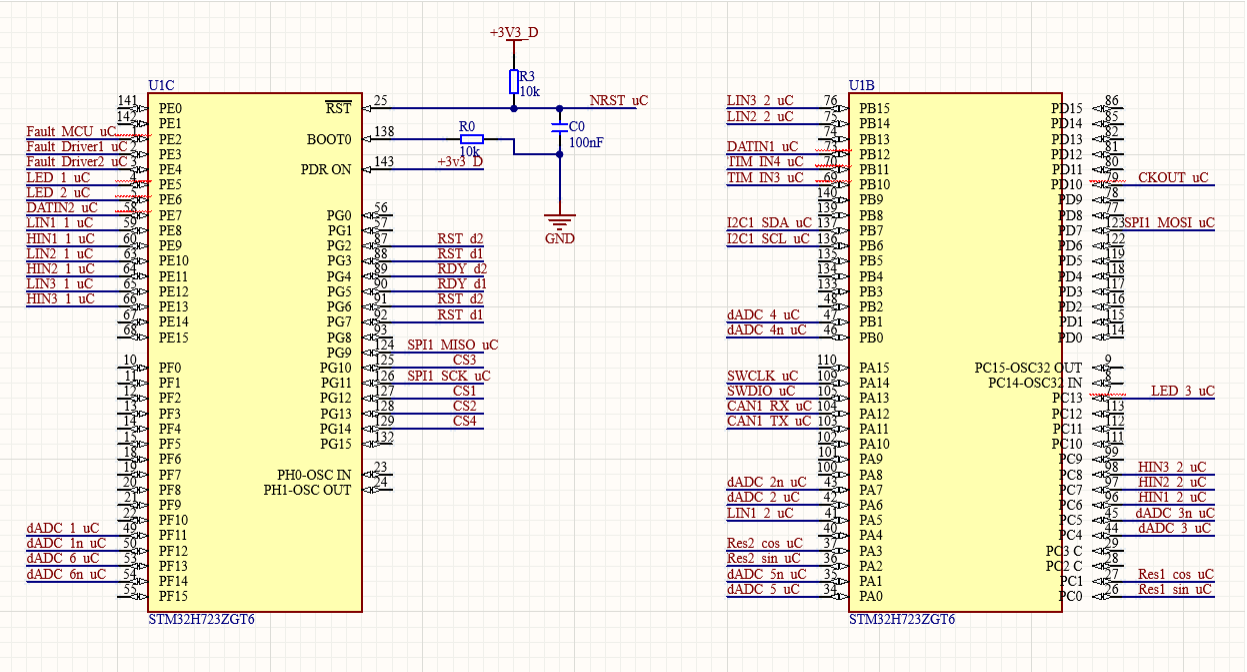


Figure 4.1: The controller in Altium Designer

On the power side of the uC, 3.3 V has been connected to all the necessary pins. The decoupling capacitors are put as its own line below due to readability but will be connected to their respective pins on the layout part of the design. Decoupling capacitors are used to filter out voltage spikes and pass through only the DC component of the signal. For this one capacitor of value 100nF is used per input.

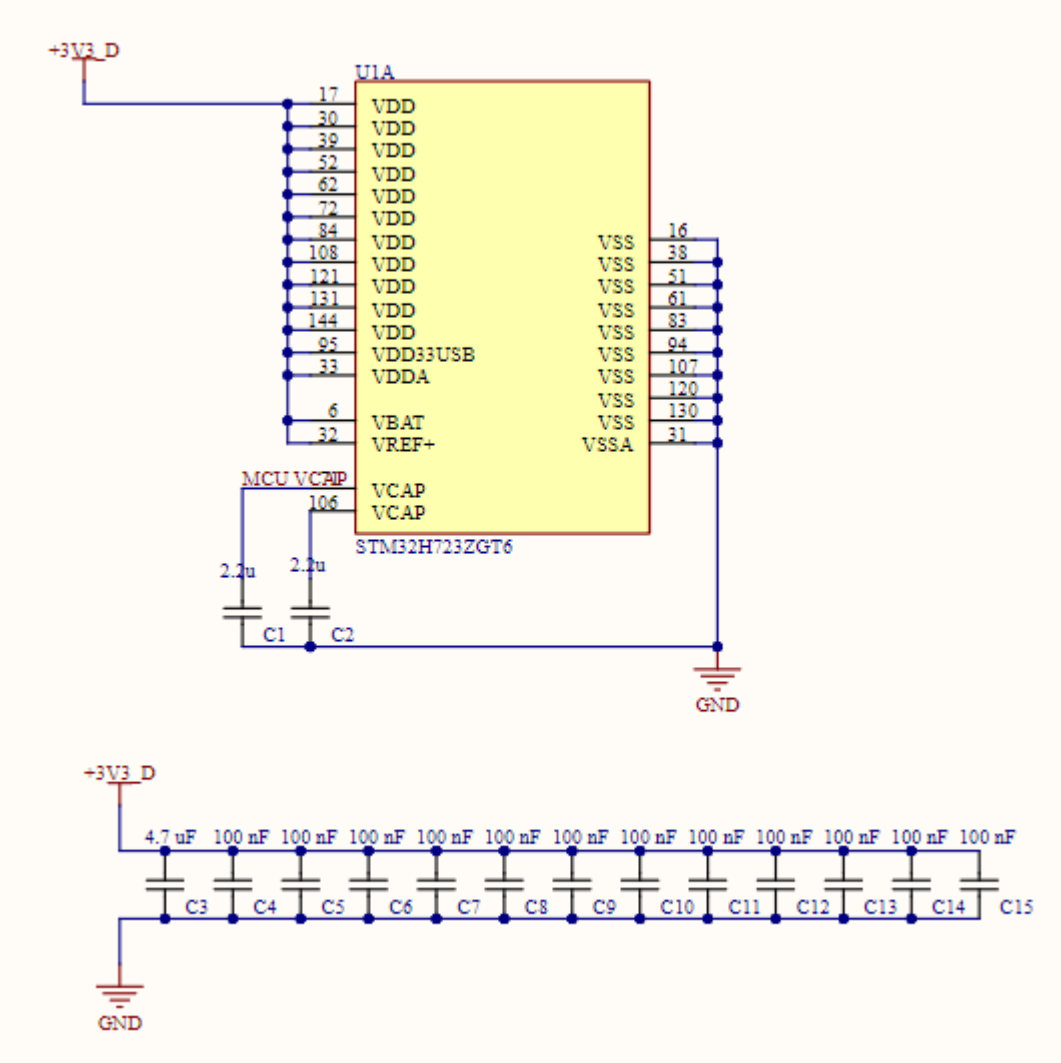


Figure 4.2: The power section of the controller

4.1.2 The TSR 1-2450

The TSR 1-2450 is a step down switching regulator with a high efficiency up to 96% [10]. This down switching regulator takes 12 V as its input and reduces it to 5 V.

- **In-house stock**
  ION Racing has its own stock of these components, and they're easy to solder.

- **Simplicity**
  The TSR 1-2450 is easy to place and use in electrical designs. This reduces complexity of the design.

- **Delivers enough current**
  The component has a max input current of 1000 mA.

The design is simple, input output. The TSR 1-2450 handles the rest by itself. Decoupling caps have been placed at the input and output to make sure the current is stable. A TVS-Diode has been placed to protect from overcurrent.

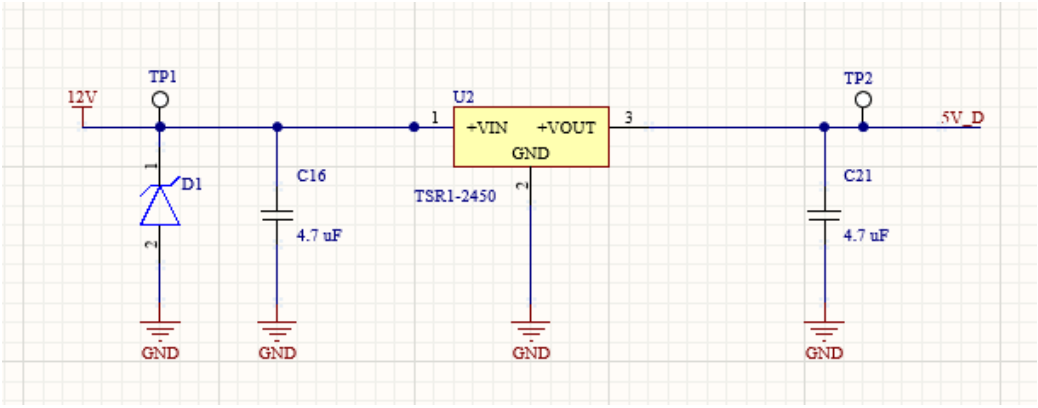


Figure 4.3: TSR in Altium Designer

4.1.3 The LP2989

The LP2989 is a fixed-output 500 mA precision LDO regulator designed for use with ceramic output capacitors. [6]

This regulator is used to convert 5 V to 3.3 V. It has a max output of 500 mA. This component is rated for temperatures from -40 to 125 °C. It comes with overtemperature and overcurrent protection. [6] Additionally one can use the following equation to determine if additional cooling is needed.

$$P_{MAX} = \frac{T_{J(MAX)} - T_A}{R_{R\Theta JA}} = \frac{125°C - 30°C}{156.5°C/W} = 607mW \tag{4.1}$$

For this equation $T_{J(MAX)}$ is = 125 °C which is the maximum temperature inside the LP2989 before the overtemperature protection shuts down. T_A = 30 °C is the maximum temperature in the air around the circuit. Due to the high temperatures at FSUK 2022 30 °C has been used as air temperature. The motor controller box is water cooled.

- **In house stock**
  ION Racing has its own stock of these components

- **Delivers 500 mA**
  According to the built in power consumption calculator in STM32Cube IDE this is sufficient to power the entire card and the resolvers.

Figure 4.4 shows the schematics for the regulator. There are decoupling capacitors connected to the input and output of the regulator, these make sure that the current in and out is stable. As this component has built in overcurrent protection there is no TVS-Diode connected to it.

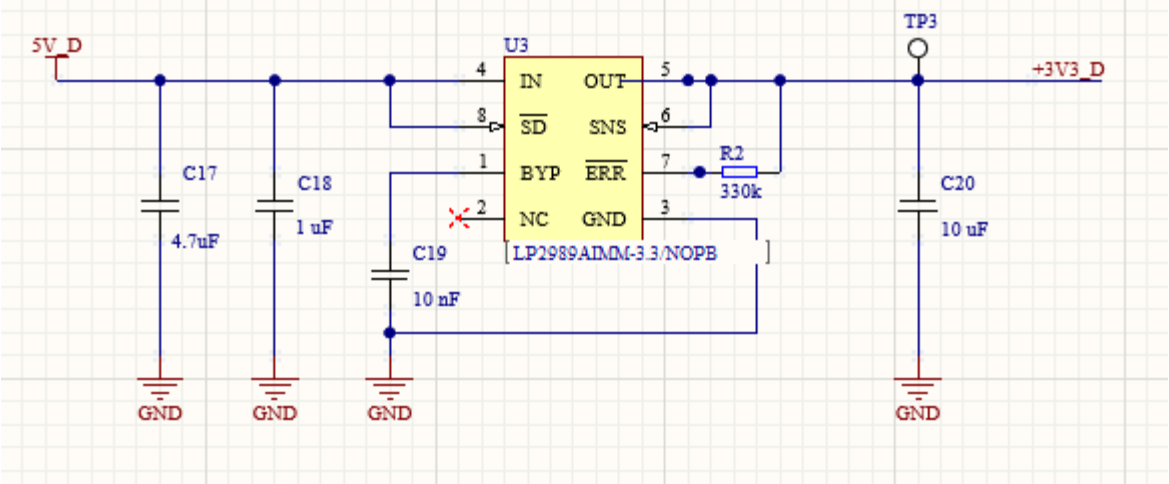


Figure 4.4: The LP2989 in Altium Designer

4.1.4 The MAX3051EKA+T

The MAX3051 is a Low-supply-Current CAN Transceiver that interfaces between the CAN protocol controller and the physical wires of the bus lines in a controller area network. [8]

The CAN-transceiver is used to transform the CAN-signals from the microcontroller into a differential signal to communicate with other electrical circuits on the car. Information about the CAN-protocol and why its chosen is found in a later chapter.

- **In house stock**
  ION Racing has its own stock of these components

- **Already used in ION Racing designs**
  Using already tested and completed designs allows for a faster testing phase when the card is produced.

A schematic of the CAN-Transceiver is shown in figure 4.5. A 100 nF decoupling capacitor is connected to VSS. This component has an adjustable maximum data rate. To select the data rate a resistance is placed between the RS-pin and ground. The equation for the resistance needed is taken from the data-sheet for the transceiver. The data rate is matched to the data rate of the ECU.

$$R_{RS}(k\Omega) = \frac{12000}{kbps} \tag{4.2}$$

There is placed a TVS Diode between CAN H and CAN L to protect the can transceiver from over current. To avoid noise and signal reflection, the transceiver is terminated with a 120 Ω resistor. There placed 4 test points to measure RX, TX, CAN-H and CAN-L.

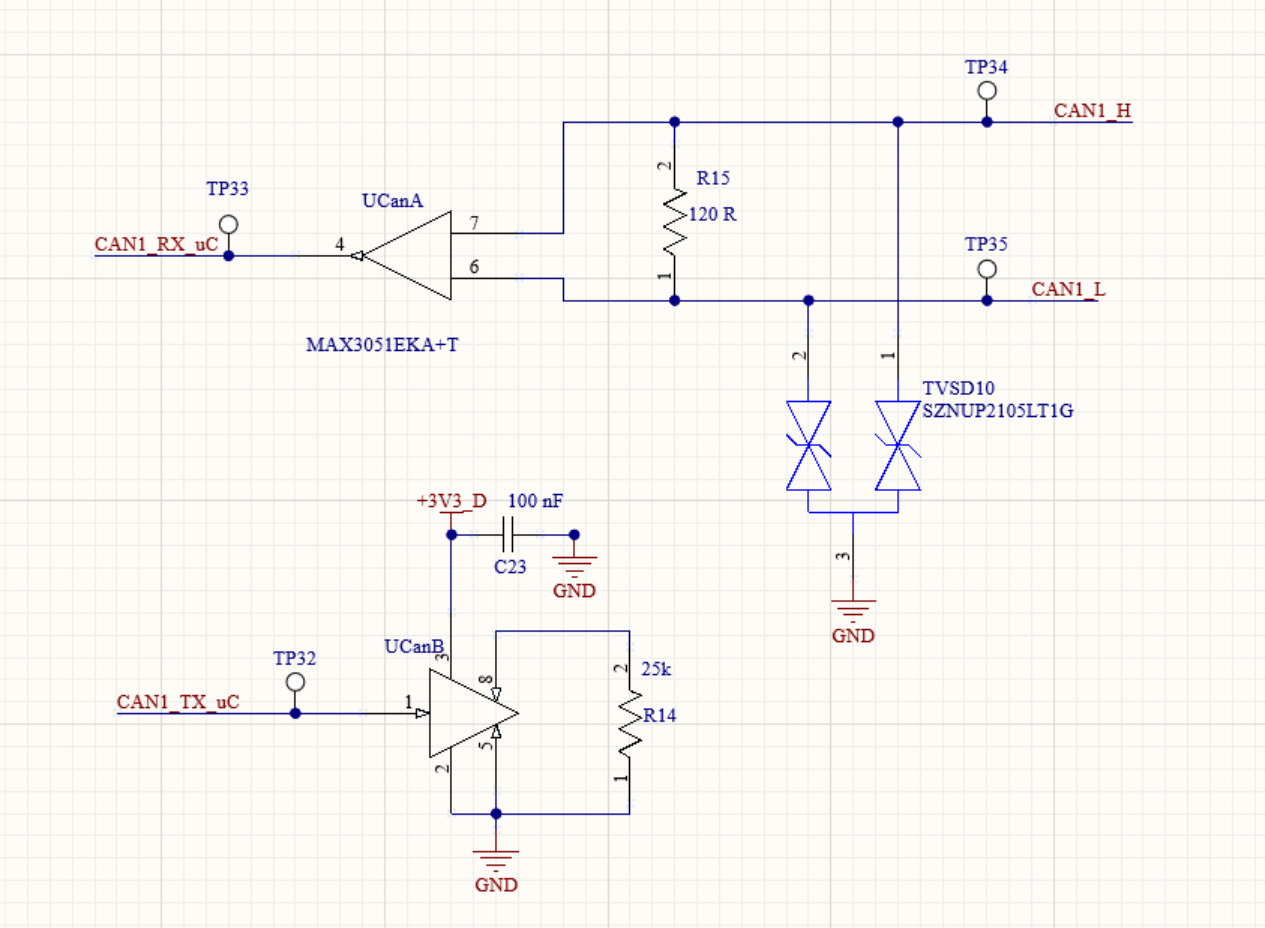


Figure 4.5: The MAX3051 in Altium Designer

4.1.5 The MLX90380

The MLX90380 is a monolithic contactless sensor IC sensitive to the flux density applied orthogonally and parallel to the IC surface. [9]

- **Easy to use**
  The MLX90380 outputs 2 signals that can easily be picked up by ADCs.

- **Small and Cheap**
  the MLX90380 comes is a SOIC-8 Package and costs only $6 USD, which compared to the previous resolver ION Racing used is a 94% reduction in costs.

For reference the previous resolver had a price of $100 USD

- **High Accuracy**
  The total angular error according to the data-sheet is ±1°.

- **Easy to place**
  The MLX90380 can be placed in multiple different configurations and positions which allows for more flexibilty during testing and production.

- **Covers the motors maximum RPM**
  The MLX90380 has a maximum Angular Speed of 25000 RPM

- **In stock at JLC PCB**
  JLC PCB keeps a stock.

This design is setup to match the recommended setup in the MLX30980 datasheet [9].

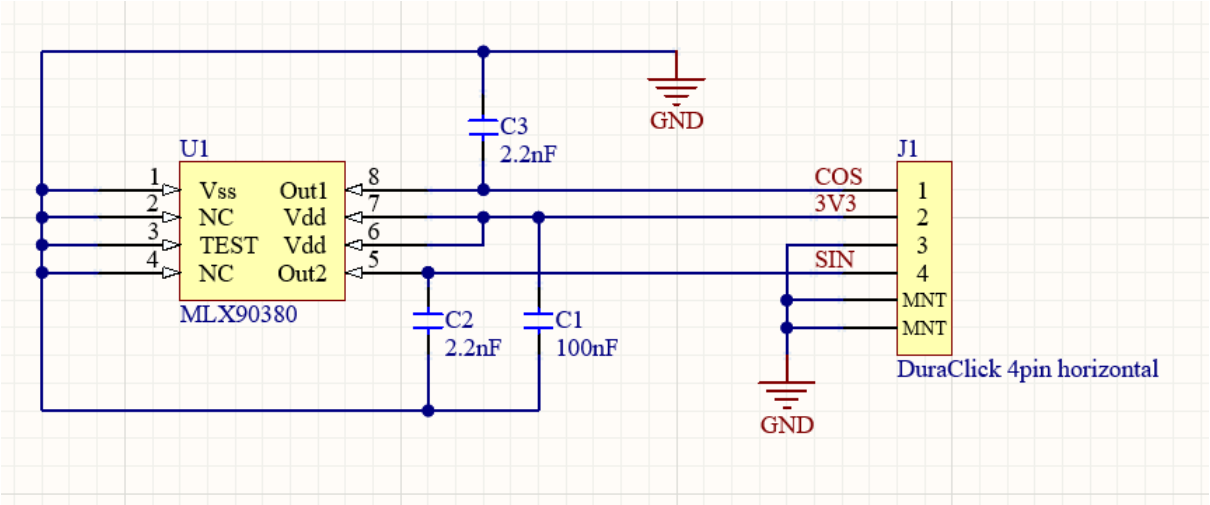


Figure 4.6: The MLX90380 in Altium Designer

4.1.6 Molex DuraClik

Most connectors on the board are of the DuraClik variant due to its simplicity.

- **Simple**
  The DuraClik is easy to use, and is modular as each pin gets its own cable from the card.

- **Modular**

The DuraCliks modularity makes it easy to swap out a cable if needed.

- **In House Stock**
  ION Racing has a wide selection of the Molex DuraClik connectors available, for example 2 pin, 4 pin, 6 pin, and 8 pin.

- **The usual standard used at ION Racing**
  Alumni have recommended this connector and in house experience shows these to be reliable and easy to work with.

4.2 Differential ADC Design

To reduce noise for the differential ADCs a common mode differential filter has been designed. To protect from ESD TVS-Diodes have been implemented into the design. The following equations were used to design this filter.

$$R_{flt} = \frac{V_{ov} - V_{esd}}{I_{max}} \tag{4.3}$$

$$C_{diff} = \frac{1}{2 \cdot \pi \cdot f_c \cdot (2 * R_{flt})} \tag{4.4}$$

$$C_{cm} = \frac{C_{diff}}{10} \tag{4.5}$$

The filter ends up looking like this in Altium [4.7], Where the differential pair input is put through a low pass filter to remove any high frequency noise, and then passed into the controller for processing. Because the ADC is set to differential mode no additional digital filter within the uC should be needed. TVS-Diodes are used to protect from overcurrent.

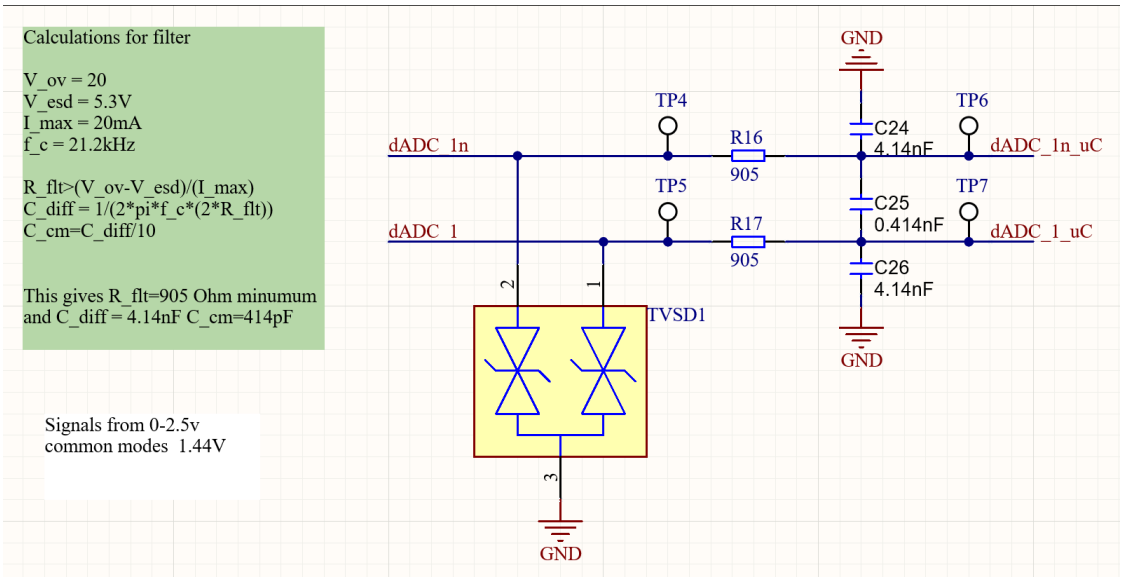


Figure 4.7: Common Mode Differential Filter Design in Altium

The value for resistance is the minimum required to limit the current into the ADC pins given the system parameters and can be quickly changed if it is shown to be insufficient.

4.3 Resolver ADC design

For the signals output by the MLX30980 2 ADC channels are employed. These are filtered through a low pass RC-filter and will be further filtered through a digital filter within the uC. Eventually if testing reveals too much noise, a filter on the resolver card or using a higher order filter on the MCU might be beneficial, there might be grounds to use buffers aswell. There is also connected a TVSD between the *sin* and *cos* lines to protect from overvoltage.

The following component values were used in the filter:

F_c	R	C
198.94 Hz	20k	40 nF

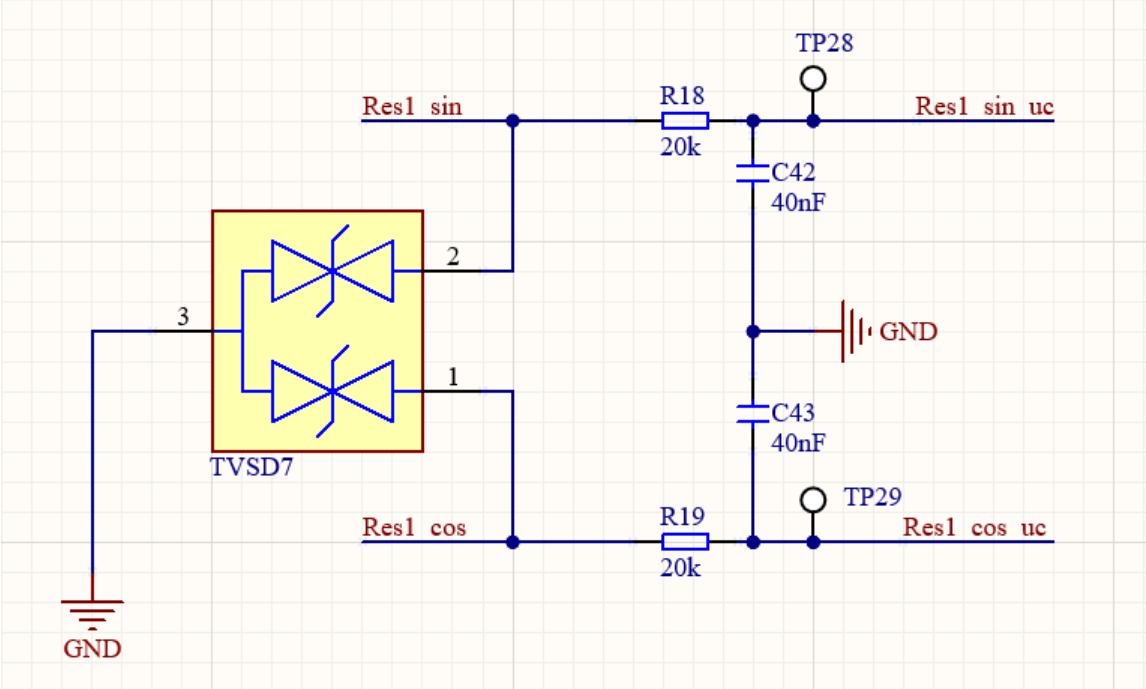


Figure 4.8: ADC circuit with *Cos* and *Sin*

This gives a $F_c = 198.94Hz$.

Because the signal measured is either a *cos* or *sin*, where the max frequency is controlled by how fast the motor can rotate, F_c has been chosen to be around 200 Hz. Maximum motor spin is 6500 RPM, which translated to Hz will be 108.33 Hz. This will remove most EMI, and additional filtration will be done in software.

4.4 Connectors

The following schematics are for the connectors used in this design. The TFML-112-02-L-D-LC is a 24-pin connector is a placeholder until the planned counterpart has been decided on. This is due to how there are plans to use a ribbon cable between the MCU and inverter drivers.

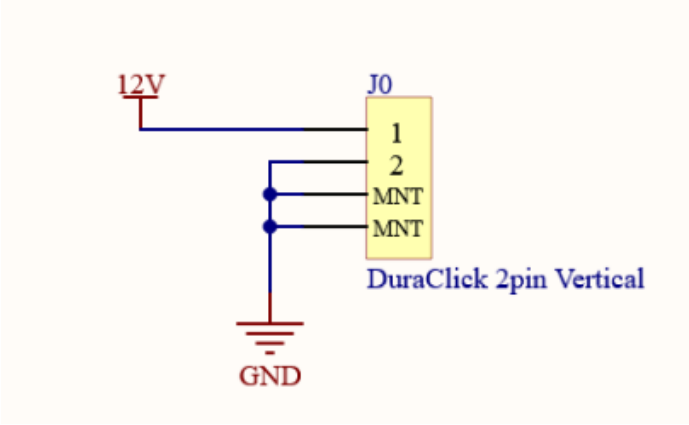


Figure 4.9: 12 Volt input

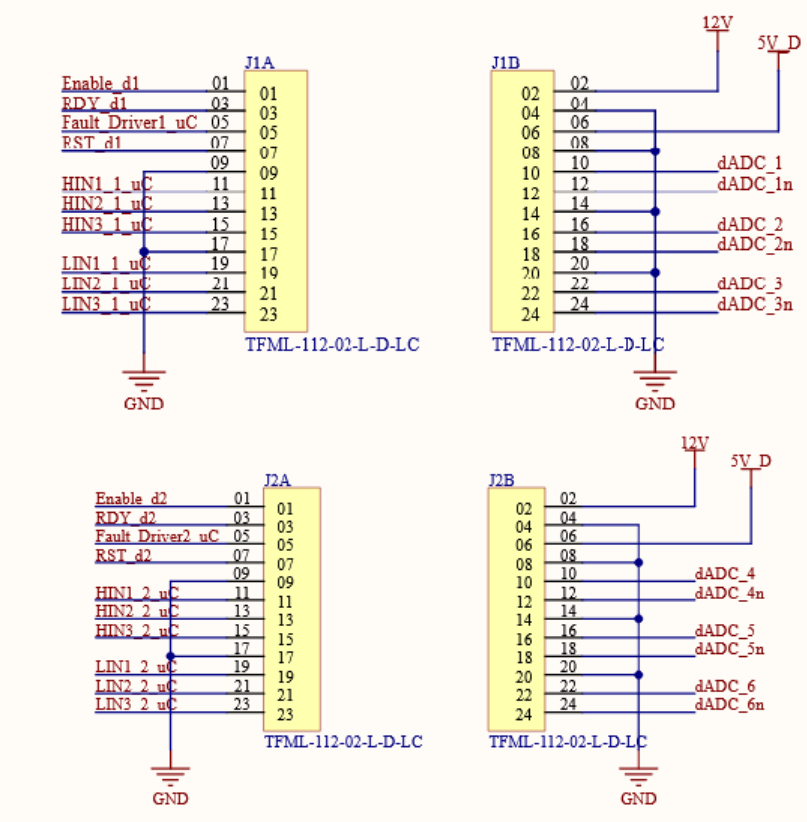


Figure 4.10: Connector to drivers

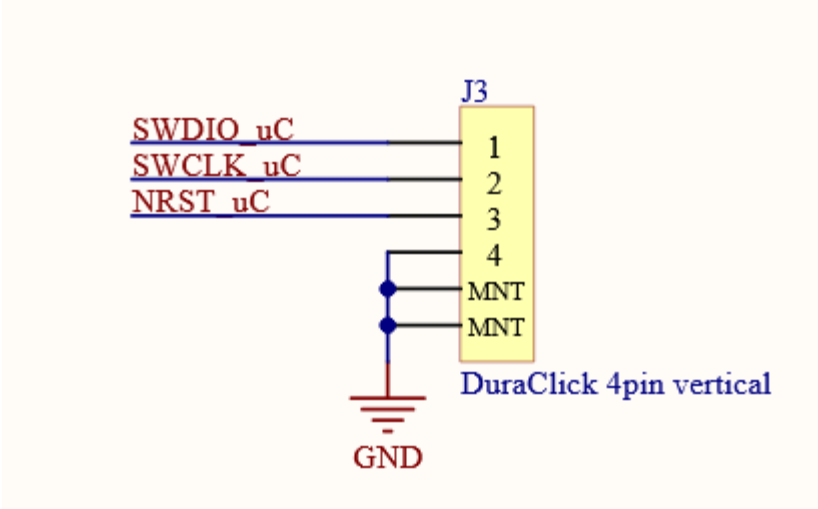


Figure 4.11: Debug connector

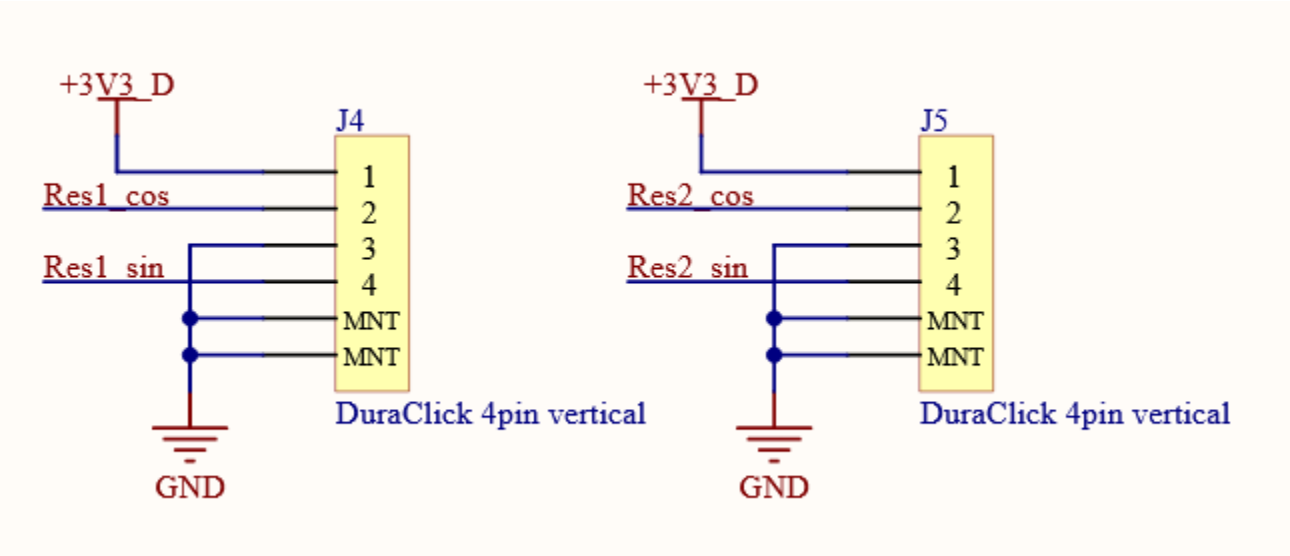


Figure 4.12: Resolver Connectors

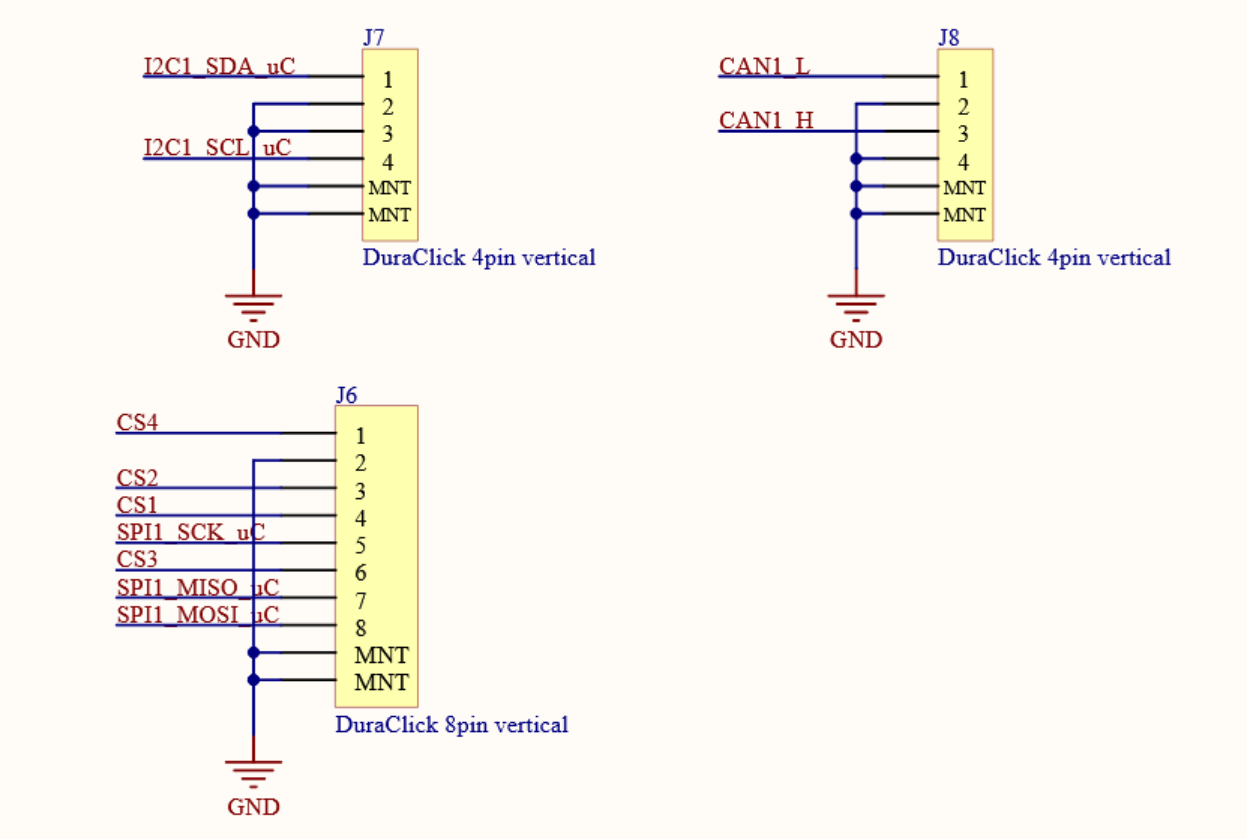


Figure 4.13: Connectivity: SPI, I^2C and CAN

4.5 Printed circuit board

During the controller's PCB design process a couple of critical points have been taken into consideration.

- **Board size**
  There isn't a specific size requirement. There should be enough room for the components to be somewhat separated to avoid noise or interference. As of the latest revision, the PCB has a size of 14.5 cm x 9.5 cm, but can be reduced in future revisions with a smarter routing strategy between the components.

- **Number of layers**
  Number of layers required depends on the size of the board, the routing connections and how many signal/net planes are desired. The current revision has 6 layers for ease of connection, however a 4 layered version would suffice.

- **Component placement**
  The placement of the different components is the most significant point. The layout of components can be seen in figure 4.15. Some parts of the controller need to be seperated, especially as the PWM and differential ADC signals can interfere with other signals and cause undesired noise. CAN, SPI and I^2C are kept apart from the PWM and differential ADC as they are vulnerable to noise. The decoupling capacitors have been placed near the μC for filtering out voltage spikes.

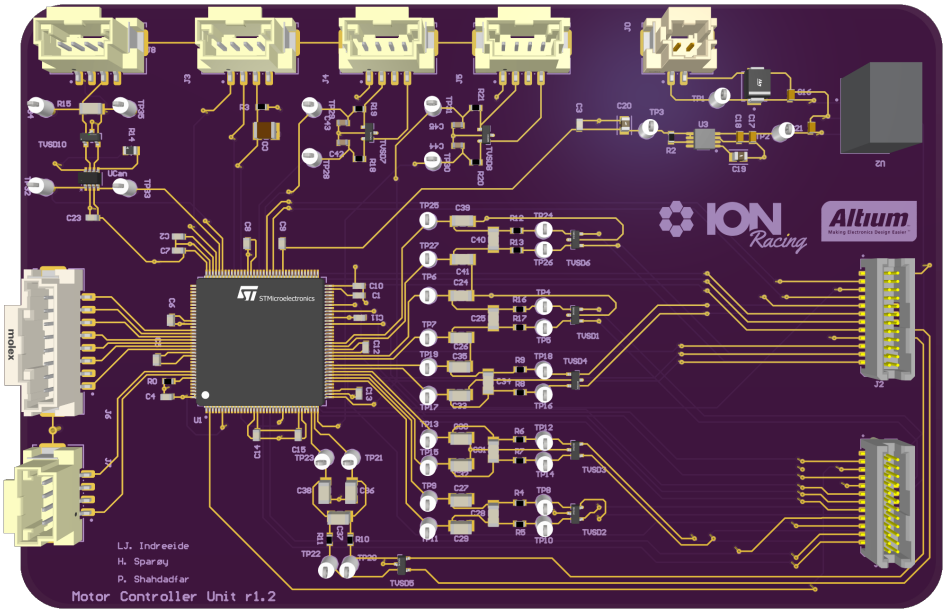


Figure 4.14: Top view of controller in Altium Designer

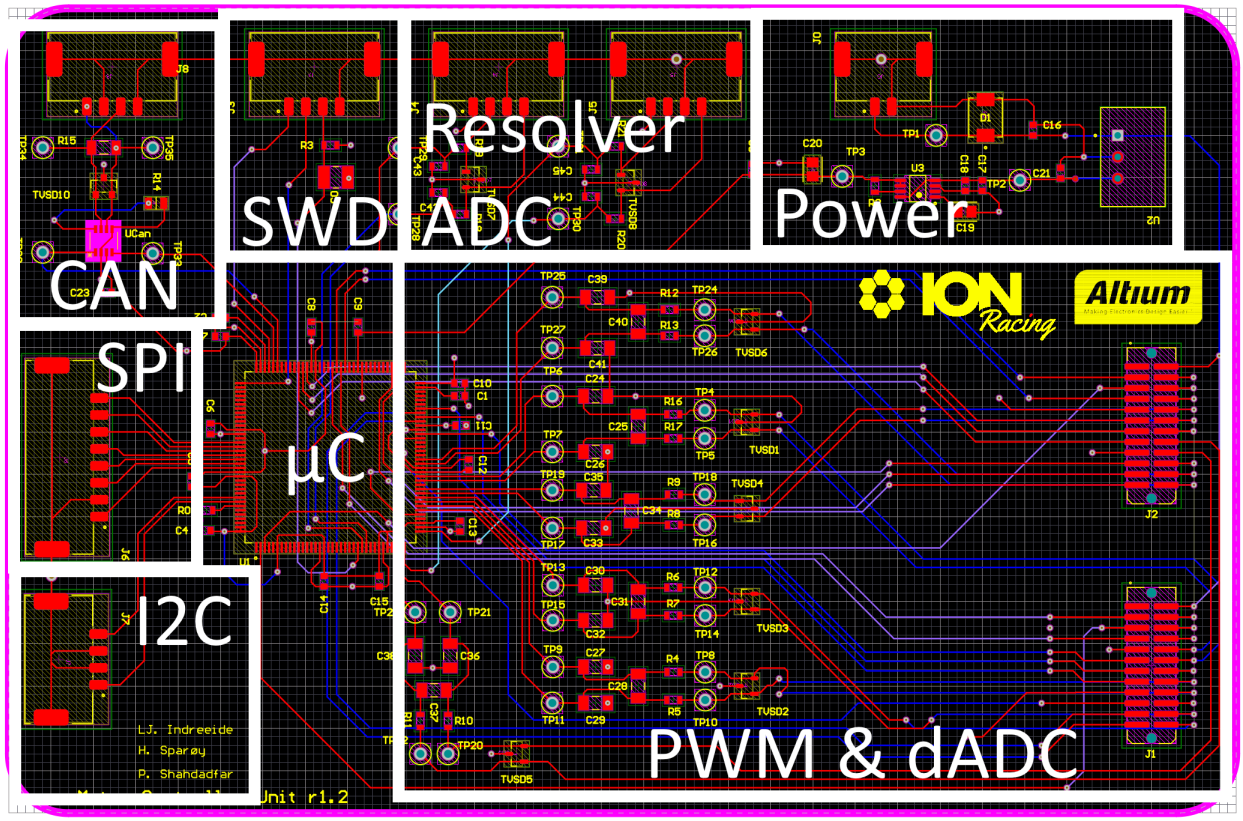


Figure 4.15: Placement of parts on the controller

4.5.1 Improvements which should be considered

Before sending the PCB design for production, several potential improvements need to be addressed. Firstly, better organization of the components can reduce the size of the board. Secondly, the signal routing between components requires a redesign, particularly for the improvement of differential paired routing to ensure balanced transmission between signals. This not only contributes to a more compact controller but also helps minimize noise interference. A 4-layered board instead of 6 layered is enough and is also cost efficient. By separating parts of each layer into zones with the desired net, a signal/net plane for the entire layer can be avoided. This approach also provides some immunity from external sources.

The controller couldn't be completed and produced due to time constraints, so it will need to be done at a later stage.

Chapter 5

Connectivity

This chapter is about the connectivity options chosen and why they were chosen.

5.1 CAN

Controller Area Network (CAN) is a communication protocol and bus standard that was originally developed for use in the automotive industry. It is a robust and widely used network protocol for connecting electronic control units (ECUs) within vehicles and other industrial applications.

CAN allows for multiple controller units to communicate with each other over a shared bus. It uses differential signals to transmit data, which provides noise immunity and allows for longer cable lengths. CAN is a message-oriented protocol which means data is transmitted in the form of messages, with each message consisting of an identifier and a payload. The identifier determines its priority and the content.

5.1.1 Why use CAN?

CAN is used because this is what the AMS and ECU is communicating with, and switching away from CAN will lead to several major changes being needed to made to those systems. Using CAN several electrical units can communicate over one network of cables. This leads to a simpler and smaller cable network compared to if every card needed to be connected individually. As all units connected have access to all communication on the network, it would be easy to expand systems with new units. New units would be able to retrieve and transmit without making changes to the existing units. CAN also has built in functionality to detect faults or errors in messages. This makes troubleshooting easier.

The STM32H723ZGx has support for FDCAN, which will not be used due to the rest of the units in the car still using normal CAN. FDCAN "Flexible Data-Rate Controller

Area Network" supports higher data rates compared to traditional CAN. While CAN typically supports up to 1 Mbps, FDCAN supports up to 8 Mbps allowing for faster communication. FDCAN is backwards compatible with CAN. This means FDCAN can communicate with existing CAN devices and networks using the CAN protocol.

The data rate chosen in this design is 500 kbps which is the same as the ECU and AMS. In a CAN network, all connected controllers should ideally operate at the same data rate to ensure reliable communication. If controllers within the network operate at different data rates, it can lead to communication issues such as message collisions and data corruption. This is also one of the reasons the MCU is using CAN instead of FDCAN, as the existing network is running on CAN.

5.1.2 Differential signaling

The CAN-transceiver transforms CAN-TX and CAN-RX into two differential signals that each consists of CAN-H and CAN-L. To send sizeable amounts of data without major interference from EMI the signals are made into differential signals. Differential signaling is a method used to transmit data reliably over a distance by utilizing the voltage differences between the two signal lines. It involves sending the same signal, but with opposite polarities, on two separate conductors or traces. The receiver then compares the voltage difference between the two lines to interpret the transmitted data. Since the receiver looks at the voltage difference, it can effectively reject common-mode noise which will affect both lines equally. This leads to any noise or interference that is common to both lines to be canceled out, resulting in improved signal integrity.

5.1.3 CAN Setup

The CAN setup can be divided into multiple sections as shown in the following table.

Start of frame	Arbitration	Control	Data	CRC	End of frame

- **Start of frame:** Is the start of the message.
- **Arbitration:** Contains the messages ID and priority.
- **Control:** States the length of the data.
- **Data:** Contains the data being sent.
- **CRC:** *Cyclic redundancy check*, This is used for detection of errors through calculating a checksum from the transmitted bits.
- **End of frame:** Is the end of the message.

5.1.4 Notes regarding further design with CAN

If a further design of the MCU is selected to be decentralized, using FDCAN between the parts could be beneficial due to the high speed and increased size of messages. FDCAN supports up to 64 bytes of data per message compared to CANs 8 bytes. This could be done using a separate FDCAN network isolated from the original CAN network that is only used within the different units produced in a decentralized design.

5.2 SPI

Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication. SPI allows for the exchange of data between a master device and one or more slave devices. The SPI communication follows a full-duplex data transfer mechanism, where data can be simultaneously transmitted and received. The master device controls the communication by generating the clock signal and initiating the data transfer. To select which slave device to communicate with, a specific SS/CS line will be pulled low, indicating that communication with the slave is desired. [2]

5.2.1 Why use SPI

During the design phase of this thesis, a member of ION Racing was designing the driver cards and inverters that are going to be produced and tested on the 2024 car. During their planning phase they requested the possibility for SPI. This is why the MCU has SPI in its design.

5.2.2 How SPI works

Here is a brief overview of how SPI works:[2]

- **SCK (Serial Clock):** This line carries the clock signal generated by the master device. This synchronizes the data transfer between the master and slaves.
- **MOSI (Master Out Slave In):** This line is used by the master to send data to the slaves.
- **MISO (Master In Slave Out):** This line is used by the slaves to send data to the master.
- **SS/CS (Slave Select/Chip Select):** This line is used by the master to select the specific slave device with which it wants to communicate. Each slave typically has its own SS/CS line.

During data transfer, the master sends data on the MOSI line while the slaves send data on the MISO line. The data is transmitted in a synchronized manner based on the clock

signal on the SCK line.

5.3 I2C

Inter-Integrated Circuit (I^2C) is a popular synchronous serial communication protocol used to enable communication between integrated circuits. It allows multiple devices to communicate with each other using only two lines: a serial data line (SDA) and a serial clock line (SCL). The I2C protocol operates in a master-slave configuration, where one device acts as the master and initiates and controls the communication with one or more slave devices. The master device generates the clock signal on the SCL line to synchronize the data transfer. [7]

5.3.1 Why use I2C

The usage of I^2C has not been decided yet but it could be used to upgrade certain systems or parts in the future. One of the upgrades that could be made using I^2C would be to the resolver used. The MLX90381 is the same resolver as the MLX90380, except it has a lower output refresh rate of 2 μs while the MLX90380 has a refresh rate of 4 μs. The MLX90381 can also change its detection field based on the programming while the MLX90380 cannot due to its pre-programmed state. This would also allow for a more modular approach as one could change the orientation of the detectors on the fly as needed. This is why the MCU has I^2C in its design.

5.3.2 How I2C works

Here is a brief overview of how I^2C works: [7]

1. **Start Condition:** The master initiates communication by sending a specific sequence of signals on the SDA and SCL lines. This indicates the start of a transaction.

2. **Addressing:** The master sends the address of the slave device it wants to communicate with. Each slave has its own unique address assigned. The address includes a read/write bit, indicating which function the master would like to access.

3. **Data Transfer:** Once the slave device with the matching address is selected, data is transmitted in a series of 8-bit chunks. Each chunk is acknowledged by the receiving device.

4. **Stop Condition:** After the data has been transferred, the master sends a stop condition that indicates the end of the transaction.

Chapter 6

Discussion and further work

This thesis and project has proven to be quite a challenge for someone with close to no knowledge on controlling a three-phase motor. The magnitude of work and effort required to design and produce a motor controller was not anticipated. Only a few of the goals set for the testing phase were met and the remaining will need to be completed at at later point in time. A great portion of the entire project is incomplete with some errors that need to be reworked before a final version of the controller is ready for production.

6.1 The system

The system that was created for testing introduced a lot of time consuming issues. Mostly due to the lack of availability for both microcontrollers and resolvers. A microcontroller that did not meet all the requirements of the project had to be used, and without a resolver a sensorless algorithm had to be implemented. These two complications made the software a lot more challenging to create.

6.2 Software

The software outlined in figure 3.10 and 3.11 does not make the motor spin smoothly since it lacks a startup phase. Without the startup phase, the back EMF readings are not useful and the software can't use their readings for 6-step commutation. The motor has however been spun by removing the back EMF portion of the software and periodically iterating the commutation sequence. This method is highly inefficient, but demonstrates that the software could function with either a working resolver or a back EMF startup phase.

6.3 Design

The design created in this project would in theory work on 2 motors, but if attempted to scale up to four motors would be lacking. This is due to each differential ADC using 2 pins each and each timer requiring 6. Due to how not every pin can be an input for ADCs or output PWM, this means the design has to be updated to take into account the increased requirements. Eventually a decentralized design with a main controller and several sensor modules with communication through FDCAN might be the way to expand upon this design.

The design is also lacking LEDs, but the pins for LEDs have been selected and put into the design. The need for LEDs is questionable due to the placement of the card in a cramped closed box. An argument can be made for implementing LEDs as they could help with debugging.

Before production, potential improvements for the PCB design should be made. Changes such as smarter component organization, redesigned signal routing and additional noise reduction strategies will be beneficial to the final product.

6.4 Further work

As mentioned, the controller still requires a great deal of work. An objective of this project was to settle between controlling 2 or 4 motors. Only 1 motor control was accomplished to some degree and a desired PI-controller remains to be applied. In order to implement the controller in a electric race car, the following objectives need attending:

- Expand software
- Improved revision of the controller's PCB design
- Produce the controller
- Establish connection with the car's ECU
- Test the controller with the car's high voltage accumulator.
- Optimize the controller for Formula Student events

6.5 Conclusion

This thesis's primary objective has been to attempt to design and produce a motor controller for an electric race car. The thesis was more complicated than anticipated, and availability issues further added to the complexity. There is a lot of work to be done still to finish the motor controller and fit it to the car. A lot of progress was still made, and a good starting point for both the software and the design has been achieved.

Further development will be done for the next formula student season. Testing revealed some missteps made with the selection of the STM32H723ZG compared to other options. This thesis writing experience has been highly educational and informative, providing valuable insights into motor control techniques and the essential components involved.

Bibliography

[1] Altium. *Altium Designer*. 2023. URL: https://www.altium.com/altium-designer (visited on 06/02/2023).

[2] Analog Devices. *Introduction to SPI Interface*. 2018. URL: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf (visited on 05/28/2023).

[3] EMRAX. *EMRAX 228 Datasheet*. 2022. URL: https://emrax.com/wp-content/uploads/2022/11/EMRAX_228_datasheet_A00.pdf (visited on 03/20/2023).

[4] EMRAX. *User's Manual for Advanced Axial Flux Synchronous Motors and Generators, Version 5.1*. 2018. URL: https://emrax.com/wp-content/uploads/2017/10/user_manual_for_emrax_motors.pdf (visited on 03/20/2023).

[5] FSUK. *FSUK Rules*. Jan. 2023. URL: https://www.imeche.org/docs/default-source/1-oscar/formula-student/2023/rules/fsuk-2023-rules---v1-1-(released-version---dec-22)7f18038e54216d0c8310ff0100d05193.pdf?sfvrsn=2 (visited on 01/16/2023).

[6] Texas Instruments. *LP2989 Micropower and Low-Noise, 500-mA Ultra Low-Dropout Regulator for Use With Ceramic Output Capacitors*. 2015. URL: https://www.ti.com/lit/ds/symlink/lp2989.pdf?ts=1684821599879 (visited on 05/22/2023).

[7] Texas Instruments. *Understanding the I2C Bus*. 2015. URL: https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1685869110840 (visited on 05/29/2023).

[8] maxim integrated. *+3.3V, 1Mbps, Low-Supply-Current CAN Transceiver*. 2018. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX3051.pdf (visited on 05/22/2023).

[9] Melexis. *MLX90380 - Triaxis Resolver*. 2021. URL: https://media.melexis.com/-/media/files/documents/datasheets/mlx90380-datasheet-melexis.pdf (visited on 04/13/2023).

[10] Traco Power. *TSR 1 Datasheet*. 2022. URL: https://www.tracopower.com/sites/default/files/products/datasheets/tsr1_datasheet.pdf (visited on 05/22/2023).

[11] STMicroelectronics. *AN1946 Application Note: SENSORLESS BLDC MOTOR CONTROL AND BEMF SAMPLING METHODS WITH ST7MC*. 2007. URL: https://www.st.com/resource/en/application_note/cd00020086-sensorless-bldc-motor-control-and-bemf-sampling-methods-with--st7mc-stmicroelectronics.pdf (visited on 04/12/2023).

[12] STMicroelectronics. *STM32 Data Sheet*. 2021. URL: https://www.st.com/resource/en/datasheet/stm32h723zg.pdf (visited on 02/08/2023).

Appendix A

Software code

A.1 main.c

```
1
2
3  /* USER CODE BEGIN Header */
4  /**
5    ******************************************************************************
6    * @file           : main.c
7    * @brief          : Main program body
8    ******************************************************************************
9    * @attention
10   *
11   * Copyright (c) 2023 STMicroelectronics.
12   * All rights reserved.
13   *
14   * This software is licensed under terms that can be found in the LICENSE file
15   * in the root directory of this software component.
16   * If no LICENSE file comes with this software, it is provided AS-IS.
17   *
18   ******************************************************************************
19   */
20 /* USER CODE END Header */
21 /* Includes ------------------------------------------------------------------*/
22 #include "main.h"
23
24 /* Private includes ----------------------------------------------------------*/
25 /* USER CODE BEGIN Includes */
26 #include "stm32h7xx_hal_gpio.h"
27 #include "stm32h7xx_hal_cortex.h" //la til
28 #include "stm32h7xx_hal_tim.h"
29 #include <stdbool.h>
30 #include <stdio.h>
31 //#include "stm32h7xx_hal.h"
32 /* USER CODE END Includes */
33
34 /* Private typedef -----------------------------------------------------------*/
35 /* USER CODE BEGIN PTD */
36
37 /* USER CODE END PTD */
38
39 /* Private define ------------------------------------------------------------*/
40 /* USER CODE BEGIN PD */
41
42 /* USER CODE END PD */
43
44 /* Private macro -------------------------------------------------------------*/
45 /* USER CODE BEGIN PM */
46
```

```
47 /* USER CODE END PM */
48
49 /* Private variables ---------------------------------------------------------*/
50 ADC_HandleTypeDef hadc1;
51 ADC_HandleTypeDef hadc2;
52 ADC_HandleTypeDef hadc3;
53 DMA_HandleTypeDef hdma_adc1;
54 DMA_HandleTypeDef hdma_adc2;
55
56 DAC_HandleTypeDef hdac1;
57
58 DTS_HandleTypeDef hdts;
59
60 TIM_HandleTypeDef htim1;
61 TIM_HandleTypeDef htim4;
62 TIM_HandleTypeDef htim8;
63
64 UART_HandleTypeDef huart3;
65
66 /* USER CODE BEGIN PV */
67 PWM_State_t Cur_PWM_State = PWM_ON;
68 Commutation_State_t Cur_Commutation_State = step_1;
69 Master_Count_Step_t Cur_Master_Count_Step = Step_1;
70 uint32_t Period_Timer = 65535;
71 uint16_t BEMF_ADC_Val[3];
72 uint32_t Pedal_ADC_Val;
73 uint16_t Pedal_Val = 0;
74 uint16_t BEMF_ZeroPoint = 64000;
75 uint16_t Prescaler_Value = 300;
76 bool LetsGo = false;
77 bool BEMF_A_ispositive = false;
78 bool BEMF_B_ispositive = false;
79 bool BEMF_C_ispositive = false;
80 bool Phase_A_Running = false;
81 bool Phase_AN_Running = false;
82 bool Phase_B_Running = false;
83 bool Phase_BN_Running = false;
84 bool Phase_C_Running = false;
85 bool Phase_CN_Running = false;
86 // testvariable
87 uint16_t Test = 0;
88
89
90 /* USER CODE END PV */
91
92 /* Private function prototypes -----------------------------------------------*/
93 void SystemClock_Config(void);
94 void PeriphCommonClock_Config(void);
95 static void MX_GPIO_Init(void);
96 static void MX_DMA_Init(void);
97 static void MX_ADC2_Init(void);
98 static void MX_ADC3_Init(void);
99 static void MX_DTS_Init(void);
100 static void MX_TIM1_Init(void);
101 static void MX_TIM4_Init(void);
102 static void MX_TIM8_Init(void);
103 static void MX_USART3_UART_Init(void);
104 static void MX_USB_OTG_HS_USB_Init(void);
105 static void MX_ADC1_Init(void);
106 static void MX_DAC1_Init(void);
107 /* USER CODE BEGIN PFP */
108 void Execute_Commutation(Commutation_State_t Cur_Commutation_State);
109 /* USER CODE END PFP */
110
111 /* Private user code ---------------------------------------------------------*/
112 /* USER CODE BEGIN 0 */
113
114
115
116 // Define function for calculating value for timers to count to
117
118 void Execute_Commutation(Commutation_State_t Cur_Commutation_State){
119   // Function for executing the correct step of the commutation sequence
120   // Get reading from pedal sensor
```

```
121   HAL_ADC_Start_DMA(&hadc1, &Pedal_ADC_Val, 1);
122   HAL_ADC_PollForConversion(&hadc1, 10);
123   Pedal_Val = HAL_ADC_GetValue(&hadc1);
124   // Switch on and off the correct outputs
125   switch(Cur_Commutation_State){
126     case step_1:
127       Phase_C_Running = false;
128       Phase_A_Running = true;
129       break;
130
131     case step_2:
132       Phase_BN_Running = false;
133       Phase_CN_Running = true;
134       break;
135
136     case step_3:
137       Phase_A_Running = false;
138       Phase_B_Running = true;
139       break;
140
141     case step_4:
142       Phase_CN_Running = false;
143       Phase_AN_Running = true;
144       break;
145
146     case step_5:
147       Phase_B_Running = false;
148       Phase_C_Running = true;
149       break;
150
151     case step_6:
152       Phase_AN_Running = false;
153       Phase_BN_Running = true;
154       break;
155   }
156 }
157
158 void Pulse(Master_Count_Step_t Cur_Master_Count_Step){
159   /* Function that outputs the phases in sequence to mimic phase shift */
160   // Check which step the master counter is at and output the appropriate phase
161   switch(Cur_Master_Count_Step){
162   case Step_1:
163     // Only output the phase when it is supposed to in the commutation sequence
164     if(Phase_A_Running){
165       TIM1->CCR1 = Pedal_Val;
166     }
167     if(Phase_AN_Running){
168       TIM8->CCR1 = Pedal_Val;
169     }
170     // Turn the other outputs off
171     TIM1->CCR2 = 0;
172     TIM8->CCR2 = 0;
173     TIM1->CCR3 = 0;
174     TIM8->CCR3 = 0;
175     break;
176   case Step_2:
177     // Only output the phase when it is supposed to in the commutation sequence
178     if(Phase_B_Running){
179       TIM1->CCR2 = Pedal_Val;
180     }
181     if(Phase_BN_Running){
182       TIM8->CCR2 = Pedal_Val;
183     }
184     // Turn the other outputs off
185     TIM1->CCR1 = 0;
186     TIM8->CCR1 = 0;
187     TIM1->CCR3 = 0;
188     TIM8->CCR3 = 0;
189     break;
190   case Step_3:
191     // Only output the phase when it is supposed to in the commutation sequence
192     if(Phase_C_Running){
193       TIM1->CCR3 = Pedal_Val;
194     }
```

```
195     if(Phase_CN_Running){
196       TIM8->CCR3 = Pedal_Val;
197     }
198     // Turn the other outputs off
199     TIM1->CCR1 = 0;
200     TIM8->CCR1 = 0;
201     TIM1->CCR2 = 0;
202     TIM8->CCR2 = 0;
203     break;
204   }
205 }
206 /* USER CODE END 0 */
207
208 /**
209   * @brief  The application entry point.
210   * @retval int
211   */
212 int main(void)
213 {
214   /* USER CODE BEGIN 1 */
215
216   /* USER CODE END 1 */
217
218   /* MCU Configuration--------------------------------------------------------*/
219
220   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
221   HAL_Init();
222
223   /* USER CODE BEGIN Init */
224
225   /* USER CODE END Init */
226
227   /* Configure the system clock */
228   SystemClock_Config();
229
230 /* Configure the peripherals common clocks */
231   PeriphCommonClock_Config();
232
233   /* USER CODE BEGIN SysInit */
234
235   /* USER CODE END SysInit */
236
237   /* Initialize all configured peripherals */
238   MX_GPIO_Init();
239   MX_DMA_Init();
240   MX_ADC2_Init();
241   MX_ADC3_Init();
242   MX_DTS_Init();
243   MX_TIM1_Init();
244   MX_TIM4_Init();
245   MX_TIM8_Init();
246   MX_USART3_UART_Init();
247   MX_USB_OTG_HS_USB_Init();
248   MX_ADC1_Init();
249   MX_DAC1_Init();
250   /* USER CODE BEGIN 2 */
251   // Turn all PWM outputs off at the start
252   TIM1->CCR1 = 0;
253   TIM1->CCR2 = 0;
254   TIM1->CCR3 = 0;
255   TIM8->CCR1 = 0;
256   TIM8->CCR2 = 0;
257   TIM8->CCR3 = 0;
258
259   /* USER CODE END 2 */
260
261   /* Infinite loop */
262   /* USER CODE BEGIN WHILE */
263   while (1)
264 {
265 // Check if PWM is supposed to be on
266 if(Cur_PWM_State == PWMLON){
267   // Get readings from the back EMF ADC
268   HAL_ADC_Start_DMA(&hadc2, (uint16_t*)&BEMF_ADC_Val[0], 3);
```

```
269   HAL_ADC_PollForConversion(&hadc2, 15);
270   // Check what commutation step the algorithm is in
271   switch(Cur_Commutation_State){
272     case step_1:
273       // If the back EMF reading from phase C has dropped below the zeropoint move on to the
      next commutation step
274       if(BEMF_ADC_Val[2] < BEMF_ZeroPoint){
275         Cur_Commutation_State = step_2;
276       }
277       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
278       Execute_Commutation(Cur_Commutation_State);
279       break;
280
281     case step_2:
282       // if the back EMF reading from phase B goes above the zeropoint move on to the next
      commutation step
283       if(BEMF_ADC_Val[1] > BEMF_ZeroPoint){
284         Cur_Commutation_State = step_3;
285       }
286       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
287       Execute_Commutation(Cur_Commutation_State);
288       break;
289
290     case step_3:
291       // If the back EMF reading from phase A has dropped below the zeropoint move on to the
      next commutation step
292       if(BEMF_ADC_Val[0] < BEMF_ZeroPoint){
293         Cur_Commutation_State = step_3;
294       }
295       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
296       Execute_Commutation(Cur_Commutation_State);
297       break;
298
299     case step_4:
300       // if the back EMF reading from phase C goes above the zeropoint move on to the next
      commutation step
301       if(BEMF_ADC_Val[2] > BEMF_ZeroPoint){
302         Cur_Commutation_State = step_5;
303       }
304       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
305       Execute_Commutation(Cur_Commutation_State);
306       break;
307
308     case step_5:
309       // If the back EMF reading from phase B has dropped below the zeropoint move on to the
      next commutation step
310       if(BEMF_ADC_Val[1] < BEMF_ZeroPoint){
311         Cur_Commutation_State = step_6;
312       }
313       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
314       Execute_Commutation(Cur_Commutation_State);
315       break;
316
317     case step_6:
318       // if the back EMF reading from phase A goes above the zeropoint move on to the next
      commutation step
319       if(BEMF_ADC_Val[0] > BEMF_ZeroPoint){
320         Cur_Commutation_State = step_1;
321       }
322       // Call to the Execute_Commutation function to update the duty cycle and potentially
      phase output
323       Execute_Commutation(Cur_Commutation_State);
324       break;
325     }
326 }
327
328 else{
329   // Turn off PWM output
330   HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);
```

```
331   HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
332   HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_3);
333   HAL_TIM_PWM_Stop(&htim8, TIM_CHANNEL_1);
334   HAL_TIM_PWM_Stop(&htim8, TIM_CHANNEL_2);
335   HAL_TIM_PWM_Stop(&htim8, TIM_CHANNEL_3);
336 }
337
338
339     /* USER CODE END WHILE */
340
341     /* USER CODE BEGIN 3 */
342   }
343   /* USER CODE END 3 */
344 }
345
346 /**
347   * @brief System Clock Configuration
348   * @retval None
349   */
350 void SystemClock_Config(void)
351 {
352   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
353   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
354
355   /** Supply configuration update enable
356   */
357   HAL_PWREx_ConfigSupply(PWR_LDO_SUPPLY);
358
359   /** Configure the main internal regulator output voltage
360   */
361   __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE0);
362
363   while(!__HAL_PWR_GET_FLAG(PWR_FLAG_VOSRDY)) {}
364
365   /** Initializes the RCC Oscillators according to the specified parameters
366   * in the RCC_OscInitTypeDef structure.
367   */
368   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI48|RCC_OSCILLATORTYPE_HSI;
369   RCC_OscInitStruct.HSIState = RCC_HSI_DIV1;
370   RCC_OscInitStruct.HSICalibrationValue = 64;
371   RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
372   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
373   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
374   RCC_OscInitStruct.PLL.PLLM = 4;
375   RCC_OscInitStruct.PLL.PLLN = 34;
376   RCC_OscInitStruct.PLL.PLLP = 1;
377   RCC_OscInitStruct.PLL.PLLQ = 4;
378   RCC_OscInitStruct.PLL.PLLR = 2;
379   RCC_OscInitStruct.PLL.PLLRGE = RCC_PLL1VCIRANGE_3;
380   RCC_OscInitStruct.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;
381   RCC_OscInitStruct.PLL.PLLFRACN = 3072;
382   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
383   {
384     Error_Handler();
385   }
386
387   /** Initializes the CPU, AHB and APB buses clocks
388   */
389   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
390                                 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2
391                                 |RCC_CLOCKTYPE_D3PCLK1|RCC_CLOCKTYPE_D1PCLK1;
392   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
393   RCC_ClkInitStruct.SYSCLKDivider = RCC_SYSCLK_DIV1;
394   RCC_ClkInitStruct.AHBCLKDivider = RCC_HCLK_DIV2;
395   RCC_ClkInitStruct.APB3CLKDivider = RCC_APB3_DIV2;
396   RCC_ClkInitStruct.APB1CLKDivider = RCC_APB1_DIV2;
397   RCC_ClkInitStruct.APB2CLKDivider = RCC_APB2_DIV2;
398   RCC_ClkInitStruct.APB4CLKDivider = RCC_APB4_DIV2;
399
400   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
401   {
402     Error_Handler();
403   }
404 }
```

```
405
406 /**
407   * @brief Peripherals Common Clock Configuration
408   * @retval None
409   */
410 void PeriphCommonClock_Config(void)
411 {
412   RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};
413
414   /** Initializes the peripherals clock
415   */
416   PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_ADC;
417   PeriphClkInitStruct.PLL2.PLL2M = 4;
418   PeriphClkInitStruct.PLL2.PLL2N = 12;
419   PeriphClkInitStruct.PLL2.PLL2P = 2;
420   PeriphClkInitStruct.PLL2.PLL2Q = 2;
421   PeriphClkInitStruct.PLL2.PLL2R = 2;
422   PeriphClkInitStruct.PLL2.PLL2RGE = RCC_PLL2VCIRANGE_3;
423   PeriphClkInitStruct.PLL2.PLL2VCOSEL = RCC_PLL2VCOWIDE;
424   PeriphClkInitStruct.PLL2.PLL2FRACN = 0;
425   PeriphClkInitStruct.AdcClockSelection = RCC_ADCCLKSOURCE_PLL2;
426   if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
427   {
428     Error_Handler();
429   }
430 }
431
432 /**
433   * @brief ADC1 Initialization Function
434   * @param None
435   * @retval None
436   */
437 static void MX_ADC1_Init(void)
438 {
439
440   /* USER CODE BEGIN ADC1_Init 0 */
441
442   /* USER CODE END ADC1_Init 0 */
443
444   ADC_MultiModeTypeDef multimode = {0};
445   ADC_ChannelConfTypeDef sConfig = {0};
446
447   /* USER CODE BEGIN ADC1_Init 1 */
448
449   /* USER CODE END ADC1_Init 1 */
450
451   /** Common config
452   */
453   hadc1.Instance = ADC1;
454   hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV2;
455   hadc1.Init.Resolution = ADC_RESOLUTION_16B;
456   hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
457   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
458   hadc1.Init.LowPowerAutoWait = DISABLE;
459   hadc1.Init.ContinuousConvMode = DISABLE;
460   hadc1.Init.NbrOfConversion = 1;
461   hadc1.Init.DiscontinuousConvMode = DISABLE;
462   hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
463   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
464   hadc1.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DMA_CIRCULAR;
465   hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
466   hadc1.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
467   hadc1.Init.OversamplingMode = DISABLE;
468   if (HAL_ADC_Init(&hadc1) != HAL_OK)
469   {
470     Error_Handler();
471   }
472
473   /** Configure the ADC multi-mode
474   */
475   multimode.Mode = ADC_MODE_INDEPENDENT;
476   if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
477   {
478     Error_Handler();
```

```
479   }
480
481   /** Configure Regular Channel
482   */
483   sConfig.Channel = ADC_CHANNEL_6;
484   sConfig.Rank = ADC_REGULAR_RANK_1;
485   sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
486   sConfig.SingleDiff = ADC_SINGLE_ENDED;
487   sConfig.OffsetNumber = ADC_OFFSET_NONE;
488   sConfig.Offset = 0;
489   sConfig.OffsetSignedSaturation = DISABLE;
490   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
491   {
492     Error_Handler();
493   }
494   /* USER CODE BEGIN ADC1_Init 2 */
495
496   /* USER CODE END ADC1_Init 2 */
497
498 }
499
500 /**
501   * @brief ADC2 Initialization Function
502   * @param None
503   * @retval None
504   */
505 static void MX_ADC2_Init(void)
506 {
507
508   /* USER CODE BEGIN ADC2_Init 0 */
509
510   /* USER CODE END ADC2_Init 0 */
511
512   ADC_ChannelConfTypeDef sConfig = {0};
513
514   /* USER CODE BEGIN ADC2_Init 1 */
515
516   /* USER CODE END ADC2_Init 1 */
517
518   /** Common config
519   */
520   hadc2.Instance = ADC2;
521   hadc2.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV2;
522   hadc2.Init.Resolution = ADC_RESOLUTION_16B;
523   hadc2.Init.ScanConvMode = ADC_SCAN_ENABLE;
524   hadc2.Init.EOCSelection = ADC_EOC_SEQ_CONV;
525   hadc2.Init.LowPowerAutoWait = DISABLE;
526   hadc2.Init.ContinuousConvMode = DISABLE;
527   hadc2.Init.NbrOfConversion = 3;
528   hadc2.Init.DiscontinuousConvMode = DISABLE;
529   hadc2.Init.ExternalTrigConv = ADC_SOFTWARE_START;
530   hadc2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
531   hadc2.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DMA_CIRCULAR;
532   hadc2.Init.Overrun = ADC_OVR_DATA_PRESERVED;
533   hadc2.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
534   hadc2.Init.OversamplingMode = DISABLE;
535   if (HAL_ADC_Init(&hadc2) != HAL_OK)
536   {
537     Error_Handler();
538   }
539
540   /** Configure Regular Channel
541   */
542   sConfig.Channel = ADC_CHANNEL_2;
543   sConfig.Rank = ADC_REGULAR_RANK_1;
544   sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
545   sConfig.SingleDiff = ADC_SINGLE_ENDED;
546   sConfig.OffsetNumber = ADC_OFFSET_NONE;
547   sConfig.Offset = 0;
548   sConfig.OffsetSignedSaturation = DISABLE;
549   if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
550   {
551     Error_Handler();
552   }
```

```
553
554   /** Configure Regular Channel
555   */
556   sConfig.Channel = ADC_CHANNEL_5;
557   sConfig.Rank = ADC_REGULAR_RANK_2;
558   if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
559   {
560     Error_Handler();
561   }
562
563   /** Configure Regular Channel
564   */
565   sConfig.Channel = ADC_CHANNEL_6;
566   sConfig.Rank = ADC_REGULAR_RANK_3;
567   if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
568   {
569     Error_Handler();
570   }
571   /* USER CODE BEGIN ADC2_Init 2 */
572
573   /* USER CODE END ADC2_Init 2 */
574
575 }
576
577 /**
578   * @brief ADC3 Initialization Function
579   * @param None
580   * @retval None
581   */
582 static void MX_ADC3_Init(void)
583 {
584
585   /* USER CODE BEGIN ADC3_Init 0 */
586
587   /* USER CODE END ADC3_Init 0 */
588
589   ADC_ChannelConfTypeDef sConfig = {0};
590
591   /* USER CODE BEGIN ADC3_Init 1 */
592
593   /* USER CODE END ADC3_Init 1 */
594
595   /** Common config
596   */
597   hadc3.Instance = ADC3;
598   hadc3.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
599   hadc3.Init.Resolution = ADC_RESOLUTION_12B;
600   hadc3.Init.DataAlign = ADC3_DATAALIGN_RIGHT;
601   hadc3.Init.ScanConvMode = ADC_SCAN_ENABLE;
602   hadc3.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
603   hadc3.Init.LowPowerAutoWait = DISABLE;
604   hadc3.Init.ContinuousConvMode = ENABLE;
605   hadc3.Init.NbrOfConversion = 3;
606   hadc3.Init.DiscontinuousConvMode = DISABLE;
607   hadc3.Init.ExternalTrigConv = ADC_SOFTWARE_START;
608   hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
609   hadc3.Init.DMAContinuousRequests = DISABLE;
610   hadc3.Init.SamplingMode = ADC_SAMPLING_MODE_NORMAL;
611   hadc3.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DR;
612   hadc3.Init.Overrun = ADC_OVR_DATA_PRESERVED;
613   hadc3.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
614   hadc3.Init.OversamplingMode = DISABLE;
615   if (HAL_ADC_Init(&hadc3) != HAL_OK)
616   {
617     Error_Handler();
618   }
619
620   /** Configure Regular Channel
621   */
622   sConfig.Channel = ADC_CHANNEL_0;
623   sConfig.Rank = ADC_REGULAR_RANK_1;
624   sConfig.SamplingTime = ADC3_SAMPLETIME_2CYCLES_5;
625   sConfig.SingleDiff = ADC_SINGLE_ENDED;
626   sConfig.OffsetNumber = ADC_OFFSET_NONE;
```

```
627   sConfig.Offset = 0;
628   sConfig.OffsetSign = ADC3_OFFSET_SIGN_NEGATIVE;
629   if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
630   {
631     Error_Handler();
632   }
633
634   /** Configure Regular Channel
635   */
636   sConfig.Channel = ADC_CHANNEL_1;
637   sConfig.Rank = ADC_REGULAR_RANK_2;
638   if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
639   {
640     Error_Handler();
641   }
642
643   /** Configure Regular Channel
644   */
645   sConfig.Channel = ADC_CHANNEL_2;
646   sConfig.Rank = ADC_REGULAR_RANK_3;
647   if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
648   {
649     Error_Handler();
650   }
651   /* USER CODE BEGIN ADC3_Init 2 */
652
653   /* USER CODE END ADC3_Init 2 */
654
655 }
656
657 /**
658   * @brief DAC1 Initialization Function
659   * @param None
660   * @retval None
661   */
662 static void MX_DAC1_Init(void)
663 {
664
665   /* USER CODE BEGIN DAC1_Init 0 */
666
667   /* USER CODE END DAC1_Init 0 */
668
669   DAC_ChannelConfTypeDef sConfig = {0};
670
671   /* USER CODE BEGIN DAC1_Init 1 */
672
673   /* USER CODE END DAC1_Init 1 */
674
675   /** DAC Initialization
676   */
677   hdac1.Instance = DAC1;
678   if (HAL_DAC_Init(&hdac1) != HAL_OK)
679   {
680     Error_Handler();
681   }
682
683   /** DAC channel OUT1 config
684   */
685   sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
686   sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
687   sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
688   sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
689   sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
690   if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
691   {
692     Error_Handler();
693   }
694   /* USER CODE BEGIN DAC1_Init 2 */
695
696   /* USER CODE END DAC1_Init 2 */
697
698 }
699
700 /**
```

```
701   * @brief DTS Initialization Function
702   * @param None
703   * @retval None
704   */
705 static void MX_DTS_Init(void)
706 {
707
708   /* USER CODE BEGIN DTS_Init 0 */
709
710   /* USER CODE END DTS_Init 0 */
711
712   /* USER CODE BEGIN DTS_Init 1 */
713
714   /* USER CODE END DTS_Init 1 */
715   hdts.Instance = DTS;
716   hdts.Init.QuickMeasure = DTS_QUICKMEAS_DISABLE;
717   hdts.Init.RefClock = DTS_REFCLKSEL_PCLK;
718   hdts.Init.TriggerInput = DTS_TRIGGER_HW_NONE;
719   hdts.Init.SamplingTime = DTS_SMP_TIME_15_CYCLE;
720   hdts.Init.Divider = 0;
721   hdts.Init.HighThreshold = 0x0;
722   hdts.Init.LowThreshold = 0x0;
723   if (HAL_DTS_Init(&hdts) != HAL_OK)
724   {
725     Error_Handler();
726   }
727   /* USER CODE BEGIN DTS_Init 2 */
728
729   /* USER CODE END DTS_Init 2 */
730
731 }
732
733 /**
734   * @brief TIM1 Initialization Function
735   * @param None
736   * @retval None
737   */
738 static void MX_TIM1_Init(void)
739 {
740
741   /* USER CODE BEGIN TIM1_Init 0 */
742
743   /* USER CODE END TIM1_Init 0 */
744
745   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
746   TIM_MasterConfigTypeDef sMasterConfig = {0};
747   TIMEx_BreakInputConfigTypeDef sBreakInputConfig = {0};
748   TIM_OC_InitTypeDef sConfigOC = {0};
749   TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};
750
751   /* USER CODE BEGIN TIM1_Init 1 */
752
753   /* USER CODE END TIM1_Init 1 */
754   htim1.Instance = TIM1;
755   htim1.Init.Prescaler = Prescaler_Value - 1;
756   htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
757   htim1.Init.Period = 65535;
758   htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
759   htim1.Init.RepetitionCounter = 0;
760   htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
761   if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
762   {
763     Error_Handler();
764   }
765   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
766   if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
767   {
768     Error_Handler();
769   }
770   if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
771   {
772     Error_Handler();
773   }
774   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
```

```
775   sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
776   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
777   if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
778   {
779     Error_Handler();
780   }
781   sBreakInputConfig.Source = TIM_BREAKINPUTSOURCE_BKIN;
782   sBreakInputConfig.Enable = TIM_BREAKINPUTSOURCE_ENABLE;
783   sBreakInputConfig.Polarity = TIM_BREAKINPUTSOURCE_POLARITY_HIGH;
784   if (HAL_TIMEx_ConfigBreakInput(&htim1, TIM_BREAKINPUT_BRK, &sBreakInputConfig) != HAL_OK)
785   {
786     Error_Handler();
787   }
788   sConfigOC.OCMode = TIM_OCMODE_PWM2;
789   sConfigOC.Pulse = 20000;
790   sConfigOC.OCPolarity = TIM_OCPOLARITY_LOW;
791   sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;
792   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
793   sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
794   sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
795   if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
796   {
797     Error_Handler();
798   }
799   if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
800   {
801     Error_Handler();
802   }
803   if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
804   {
805     Error_Handler();
806   }
807   sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
808   sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
809   sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
810   sBreakDeadTimeConfig.DeadTime = 0;
811   sBreakDeadTimeConfig.BreakState = TIM_BREAK_ENABLE;
812   sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
813   sBreakDeadTimeConfig.BreakFilter = 0;
814   sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
815   sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
816   sBreakDeadTimeConfig.Break2Filter = 0;
817   sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
818   if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
819   {
820     Error_Handler();
821   }
822   /* USER CODE BEGIN TIM1_Init 2 */
823   // Start the PWM counter for interrupts
824   HAL_TIM_PWM_Start_IT(&htim1, TIM_CHANNEL_1);
825   HAL_TIM_PWM_Start_IT(&htim1, TIM_CHANNEL_2);
826   HAL_TIM_PWM_Start_IT(&htim1, TIM_CHANNEL_3);
827   // Start the output compare
828   HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_1);
829   HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_2);
830   HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_3);
831   HAL_TIMEx_OCN_Start(&htim1, TIM_CHANNEL_1);
832   HAL_TIMEx_OCN_Start(&htim1, TIM_CHANNEL_2);
833   HAL_TIMEx_OCN_Start(&htim1, TIM_CHANNEL_3);
834   /* USER CODE END TIM1_Init 2 */
835   HAL_TIM_MspPostInit(&htim1);
836
837 }
838
839 /**
840   * @brief TIM4 Initialization Function
841   * @param None
842   * @retval None
843   */
844 static void MX_TIM4_Init(void)
845 {
846
847   /* USER CODE BEGIN TIM4_Init 0 */
848
```

```
849   /* USER CODE END TIM4_Init 0 */
850
851   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
852   TIM_MasterConfigTypeDef sMasterConfig = {0};
853   TIM_OC_InitTypeDef sConfigOC = {0};
854
855   /* USER CODE BEGIN TIM4_Init 1 */
856
857   /* USER CODE END TIM4_Init 1 */
858   htim4.Instance = TIM4;
859   htim4.Init.Prescaler = Prescaler_Value - 1;
860   htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
861   htim4.Init.Period = 65535;
862   htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
863   htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
864   if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
865   {
866     Error_Handler();
867   }
868   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
869   if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
870   {
871     Error_Handler();
872   }
873   if (HAL_TIM_OC_Init(&htim4) != HAL_OK)
874   {
875     Error_Handler();
876   }
877   sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
878   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
879   if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
880   {
881     Error_Handler();
882   }
883   sConfigOC.OCMode = TIM_OCMODE_TIMING;
884   sConfigOC.Pulse = 0;
885   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
886   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
887   if (HAL_TIM_OC_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
888   {
889     Error_Handler();
890   }
891   /* USER CODE BEGIN TIM4_Init 2 */
892   // Start the timer counters
893   __HAL_TIM_ENABLE_IT(&htim4, TIM_IT_UPDATE );
894   HAL_TIM_Base_Start(&htim4);
895   HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
896   HAL_TIM_Base_MspInit(&htim4);
897   // Enable and set priority of the timer interrupt
898   HAL_NVIC_SetPriority(TIM4_IRQHandler(), 0, 0);
899   HAL_NVIC_EnableIRQ(TIM4_IRQHandler());
900   HAL_TIM_Base_Start_IT(&htim4);
901   /* USER CODE END TIM4_Init 2 */
902
903 }
904
905 /**
906   * @brief TIM8 Initialization Function
907   * @param None
908   * @retval None
909   */
910 static void MX_TIM8_Init(void)
911 {
912
913   /* USER CODE BEGIN TIM8_Init 0 */
914
915   /* USER CODE END TIM8_Init 0 */
916
917   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
918   TIM_MasterConfigTypeDef sMasterConfig = {0};
919   TIM_OC_InitTypeDef sConfigOC = {0};
920   TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};
921
922   /* USER CODE BEGIN TIM8_Init 1 */
```

```
923
924   /* USER CODE END TIM8_Init 1 */
925   htim8.Instance = TIM8;
926   htim8.Init.Prescaler = Prescaler_Value - 1;
927   htim8.Init.CounterMode = TIM_COUNTERMODE_UP;
928   htim8.Init.Period = 65535;
929   htim8.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
930   htim8.Init.RepetitionCounter = 0;
931   htim8.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
932   if (HAL_TIM_Base_Init(&htim8) != HAL_OK)
933   {
934     Error_Handler();
935   }
936   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
937   if (HAL_TIM_ConfigClockSource(&htim8, &sClockSourceConfig) != HAL_OK)
938   {
939     Error_Handler();
940   }
941   if (HAL_TIM_PWM_Init(&htim8) != HAL_OK)
942   {
943     Error_Handler();
944   }
945   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
946   sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
947   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
948   if (HAL_TIMEx_MasterConfigSynchronization(&htim8, &sMasterConfig) != HAL_OK)
949   {
950     Error_Handler();
951   }
952   sConfigOC.OCMode = TIM_OCMODE_PWM2;
953   sConfigOC.Pulse = 20000;
954   sConfigOC.OCPolarity = TIM_OCPOLARITY_LOW;
955   sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;
956   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
957   sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
958   sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
959   if (HAL_TIM_PWM_ConfigChannel(&htim8, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
960   {
961     Error_Handler();
962   }
963   if (HAL_TIM_PWM_ConfigChannel(&htim8, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
964   {
965     Error_Handler();
966   }
967   if (HAL_TIM_PWM_ConfigChannel(&htim8, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
968   {
969     Error_Handler();
970   }
971   sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
972   sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
973   sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
974   sBreakDeadTimeConfig.DeadTime = 0;
975   sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
976   sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
977   sBreakDeadTimeConfig.BreakFilter = 0;
978   sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
979   sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
980   sBreakDeadTimeConfig.Break2Filter = 0;
981   sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
982   if (HAL_TIMEx_ConfigBreakDeadTime(&htim8, &sBreakDeadTimeConfig) != HAL_OK)
983   {
984     Error_Handler();
985   }
986   /* USER CODE BEGIN TIM8_Init 2 */
987   // Start the PWM counters for interrupts
988   HAL_TIM_PWM_Start_IT(&htim8, TIM_CHANNEL_1);
989   HAL_TIM_PWM_Start_IT(&htim8, TIM_CHANNEL_2);
990   HAL_TIM_PWM_Start_IT(&htim8, TIM_CHANNEL_3);
991   // Start the output compare
992   HAL_TIM_OC_Start(&htim8, TIM_CHANNEL_1);
993   HAL_TIM_OC_Start(&htim8, TIM_CHANNEL_2);
994   HAL_TIM_OC_Start(&htim8, TIM_CHANNEL_3);
995   HAL_TIMEx_OCN_Start(&htim8, TIM_CHANNEL_1);
996   HAL_TIMEx_OCN_Start(&htim8, TIM_CHANNEL_2);
```

```
997   HAL_TIMEx_OCN_Start(&htim8, TIM_CHANNEL_3);
998
999   /* USER CODE END TIM8_Init 2 */
1000  HAL_TIM_MspPostInit(&htim8);
1001
1002 }
1003
1004 /**
1005   * @brief USART3 Initialization Function
1006   * @param None
1007   * @retval None
1008   */
1009 static void MX_USART3_UART_Init(void)
1010 {
1011
1012   /* USER CODE BEGIN USART3_Init 0 */
1013
1014   /* USER CODE END USART3_Init 0 */
1015
1016   /* USER CODE BEGIN USART3_Init 1 */
1017
1018   /* USER CODE END USART3_Init 1 */
1019   huart3.Instance = USART3;
1020   huart3.Init.BaudRate = 115200;
1021   huart3.Init.WordLength = UART_WORDLENGTH_8B;
1022   huart3.Init.StopBits = UART_STOPBITS_1;
1023   huart3.Init.Parity = UART_PARITY_NONE;
1024   huart3.Init.Mode = UART_MODE_TX_RX;
1025   huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
1026   huart3.Init.OverSampling = UART_OVERSAMPLING_16;
1027   huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
1028   huart3.Init.ClockPrescaler = UART_PRESCALER_DIV1;
1029   huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
1030   if (HAL_UART_Init(&huart3) != HAL_OK)
1031   {
1032     Error_Handler();
1033   }
1034   if (HAL_UARTEx_SetTxFifoThreshold(&huart3, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
1035   {
1036     Error_Handler();
1037   }
1038   if (HAL_UARTEx_SetRxFifoThreshold(&huart3, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
1039   {
1040     Error_Handler();
1041   }
1042   if (HAL_UARTEx_DisableFifoMode(&huart3) != HAL_OK)
1043   {
1044     Error_Handler();
1045   }
1046   /* USER CODE BEGIN USART3_Init 2 */
1047
1048   /* USER CODE END USART3_Init 2 */
1049
1050 }
1051
1052 /**
1053   * @brief USB_OTG_HS Initialization Function
1054   * @param None
1055   * @retval None
1056   */
1057 static void MX_USB_OTG_HS_USB_Init(void)
1058 {
1059
1060   /* USER CODE BEGIN USB_OTG_HS_Init 0 */
1061
1062   /* USER CODE END USB_OTG_HS_Init 0 */
1063
1064   /* USER CODE BEGIN USB_OTG_HS_Init 1 */
1065
1066   /* USER CODE END USB_OTG_HS_Init 1 */
1067   /* USER CODE BEGIN USB_OTG_HS_Init 2 */
1068
1069   /* USER CODE END USB_OTG_HS_Init 2 */
1070
```

```
1071 }
1072
1073 /**
1074   * Enable DMA controller clock
1075   */
1076 static void MX_DMA_Init(void)
1077 {
1078
1079   /* DMA controller clock enable */
1080   __HAL_RCC_DMA1_CLK_ENABLE();
1081
1082   /* DMA interrupt init */
1083   /* DMA1_Stream0_IRQn interrupt configuration */
1084   HAL_NVIC_SetPriority(DMA1_Stream0_IRQn, 0, 0);
1085   HAL_NVIC_EnableIRQ(DMA1_Stream0_IRQn);
1086   /* DMA1_Stream1_IRQn interrupt configuration */
1087   HAL_NVIC_SetPriority(DMA1_Stream1_IRQn, 0, 0);
1088   HAL_NVIC_EnableIRQ(DMA1_Stream1_IRQn);
1089
1090 }
1091
1092 /**
1093   * @brief GPIO Initialization Function
1094   * @param None
1095   * @retval None
1096   */
1097 static void MX_GPIO_Init(void)
1098 {
1099   GPIO_InitTypeDef GPIO_InitStruct = {0};
1100 /* USER CODE BEGIN MX_GPIO_Init_1 */
1101 /* USER CODE END MX_GPIO_Init_1 */
1102
1103   /* GPIO Ports Clock Enable */
1104   __HAL_RCC_GPIOC_CLK_ENABLE();
1105   __HAL_RCC_GPIOF_CLK_ENABLE();
1106   __HAL_RCC_GPIOH_CLK_ENABLE();
1107   __HAL_RCC_GPIOA_CLK_ENABLE();
1108   __HAL_RCC_GPIOB_CLK_ENABLE();
1109   __HAL_RCC_GPIOE_CLK_ENABLE();
1110   __HAL_RCC_GPIOD_CLK_ENABLE();
1111   __HAL_RCC_GPIOG_CLK_ENABLE();
1112
1113   /*Configure GPIO pin Output Level */
1114   HAL_GPIO_WritePin(BEMF_DIVIDER_ENABLE_GPIO_Port, BEMF_DIVIDER_ENABLE_Pin, GPIO_PIN_SET);
1115
1116   /*Configure GPIO pin Output Level */
1117   HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);
1118
1119   /*Configure GPIO pin Output Level */
1120   HAL_GPIO_WritePin(Driver_Card_Enable_GPIO_Port, Driver_Card_Enable_Pin, GPIO_PIN_SET);
1121
1122   /*Configure GPIO pin Output Level */
1123   HAL_GPIO_WritePin(GPIOD, GPIO_PIN_10, GPIO_PIN_RESET);
1124
1125   /*Configure GPIO pin Output Level */
1126   HAL_GPIO_WritePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin, GPIO_PIN_RESET);
1127
1128   /*Configure GPIO pin : B1_Pin */
1129   GPIO_InitStruct.Pin = B1_Pin;
1130   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
1131   GPIO_InitStruct.Pull = GPIO_NOPULL;
1132   HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
1133
1134   /*Configure GPIO pin : Driver_Card_Fault_Pin */
1135   GPIO_InitStruct.Pin = Driver_Card_Fault_Pin;
1136   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
1137   GPIO_InitStruct.Pull = GPIO_NOPULL;
1138   HAL_GPIO_Init(Driver_Card_Fault_GPIO_Port, &GPIO_InitStruct);
1139
1140   /*Configure GPIO pins : RMII_MDC_Pin RMII_RXD0_Pin RMII_RXD1_Pin */
1141   GPIO_InitStruct.Pin = RMII_MDC_Pin|RMII_RXD0_Pin|RMII_RXD1_Pin;
1142   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1143   GPIO_InitStruct.Pull = GPIO_NOPULL;
1144   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
```

```
1145   GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1146   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
1147
1148   /*Configure GPIO pins : RMII_REF_CLK_Pin RMII_MDIO_Pin */
1149   GPIO_InitStruct.Pin = RMII_REF_CLK_Pin|RMII_MDIO_Pin;
1150   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1151   GPIO_InitStruct.Pull = GPIO_NOPULL;
1152   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1153   GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1154   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
1155
1156   /*Configure GPIO pin : BEMF_DIVIDER_ENABLE_Pin */
1157   GPIO_InitStruct.Pin = BEMF_DIVIDER_ENABLE_Pin;
1158   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1159   GPIO_InitStruct.Pull = GPIO_NOPULL;
1160   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1161   HAL_GPIO_Init(BEMF_DIVIDER_ENABLE_GPIO_Port, &GPIO_InitStruct);
1162
1163   /*Configure GPIO pins : LED_GREEN_Pin Driver_Card_Enable_Pin */
1164   GPIO_InitStruct.Pin = LED_GREEN_Pin|Driver_Card_Enable_Pin;
1165   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1166   GPIO_InitStruct.Pull = GPIO_NOPULL;
1167   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1168   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
1169
1170   /*Configure GPIO pin : RMII_TXD1_Pin */
1171   GPIO_InitStruct.Pin = RMII_TXD1_Pin;
1172   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1173   GPIO_InitStruct.Pull = GPIO_NOPULL;
1174   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1175   GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1176   HAL_GPIO_Init(RMII_TXD1_GPIO_Port, &GPIO_InitStruct);
1177
1178   /*Configure GPIO pin : PD10 */
1179   GPIO_InitStruct.Pin = GPIO_PIN_10;
1180   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1181   GPIO_InitStruct.Pull = GPIO_NOPULL;
1182   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1183   HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
1184
1185   /*Configure GPIO pin : USB_FS_OVCR_Pin */
1186   GPIO_InitStruct.Pin = USB_FS_OVCR_Pin;
1187   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
1188   GPIO_InitStruct.Pull = GPIO_NOPULL;
1189   HAL_GPIO_Init(USB_FS_OVCR_GPIO_Port, &GPIO_InitStruct);
1190
1191   /*Configure GPIO pin : USB_FS_VBUS_Pin */
1192   GPIO_InitStruct.Pin = USB_FS_VBUS_Pin;
1193   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
1194   GPIO_InitStruct.Pull = GPIO_NOPULL;
1195   HAL_GPIO_Init(USB_FS_VBUS_GPIO_Port, &GPIO_InitStruct);
1196
1197   /*Configure GPIO pin : USB_FS_ID_Pin */
1198   GPIO_InitStruct.Pin = USB_FS_ID_Pin;
1199   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1200   GPIO_InitStruct.Pull = GPIO_NOPULL;
1201   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1202   GPIO_InitStruct.Alternate = GPIO_AF10_OTG1_HS;
1203   HAL_GPIO_Init(USB_FS_ID_GPIO_Port, &GPIO_InitStruct);
1204
1205   /*Configure GPIO pins : RMII_TX_EN_Pin RMII_TXD0_Pin */
1206   GPIO_InitStruct.Pin = RMII_TX_EN_Pin|RMII_TXD0_Pin;
1207   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1208   GPIO_InitStruct.Pull = GPIO_NOPULL;
1209   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1210   GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1211   HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
1212
1213   /*Configure GPIO pin : LED_YELLOW_Pin */
1214   GPIO_InitStruct.Pin = LED_YELLOW_Pin;
1215   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1216   GPIO_InitStruct.Pull = GPIO_NOPULL;
1217   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1218   HAL_GPIO_Init(LED_YELLOW_GPIO_Port, &GPIO_InitStruct);
```

```
1219
1220   /* EXTI interrupt init*/
1221   HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
1222   HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
1223
1224 /* USER CODE BEGIN MX_GPIO_Init_2 */
1225   // Enable EXTI interrupt for B1_Pin
1226   HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
1227   HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
1228 /* USER CODE END MX_GPIO_Init_2 */
1229 }
1230
1231 /* USER CODE BEGIN 4 */
1232 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc){
1233   // Stop the ADC's after a reading has been completed
1234   HAL_ADC_Stop_DMA(&hadc1);
1235   HAL_ADC_Stop_DMA(&hadc2);
1236 }
1237
1238 // Needed for logging of data during testing
1239 int _write(int file, char *ptr, int len) {
1240     int DataIdx;
1241
1242     for (DataIdx = 0; DataIdx < len; DataIdx++) {
1243         ITM_SendChar(*(ptr + DataIdx));
1244     }
1245
1246     return len;
1247 }
1248 /* USER CODE END 4 */
1249
1250 /**
1251   * @brief  This function is executed in case of error occurrence.
1252   * @retval None
1253   */
1254 void Error_Handler(void)
1255 {
1256   /* USER CODE BEGIN Error_Handler_Debug */
1257   /* User can add his own implementation to report the HAL error return state */
1258   __disable_irq();
1259   while (1)
1260   {
1261   }
1262   /* USER CODE END Error_Handler_Debug */
1263 }
1264
1265 #ifdef  USE_FULL_ASSERT
1266 /**
1267   * @brief  Reports the name of the source file and the source line number
1268   *         where the assert_param error has occurred.
1269   * @param  file: pointer to the source file name
1270   * @param  line: assert_param error line source number
1271   * @retval None
1272   */
1273 void assert_failed(uint8_t *file, uint32_t line)
1274 {
1275   /* USER CODE BEGIN 6 */
1276   /* User can add his own implementation to report the file name and line number,
1277     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
1278   /* USER CODE END 6 */
1279 }
1280 #endif /* USE_FULL_ASSERT */
```

A.2 stm32h7xx_it.c

```
1
2 /* USER CODE BEGIN Header */
3 /**
4   ******************************************************************************
5   * @file    stm32h7xx_it.c
6   * @brief   Interrupt Service Routines.
7   ******************************************************************************
```

```
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32h7xx_it.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "stm32h7xx_hal_gpio.h"
#include "stm32h7xx_hal_cortex.h" //la til
#include "stm32h7xx_hal_tim.h"
#include <stdbool.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN TD */

/* USER CODE END TD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/* External variables --------------------------------------------------------*/
extern DMA_HandleTypeDef hdma_adc1;
extern DMA_HandleTypeDef hdma_adc2;
extern TIM_HandleTypeDef htim1;
extern TIM_HandleTypeDef htim4;
/* USER CODE BEGIN EV */

/* USER CODE END EV */

/******************************************************************************/
/*           Cortex Processor Interruption and Exception Handlers          */
/******************************************************************************/
/**
  * @brief This function handles Non maskable interrupt.
  */
void NMI_Handler(void)
{
  /* USER CODE BEGIN NonMaskableInt_IRQn 0 */

  /* USER CODE END NonMaskableInt_IRQn 0 */
```

```
82   /* USER CODE BEGIN NonMaskableInt_IRQn 1 */
83   while (1)
84   {
85   }
86   /* USER CODE END NonMaskableInt_IRQn 1 */
87 }
88
89 /**
90   * @brief This function handles Hard fault interrupt.
91   */
92 void HardFault_Handler(void)
93 {
94   /* USER CODE BEGIN HardFault_IRQn 0 */
95
96   /* USER CODE END HardFault_IRQn 0 */
97   while (1)
98   {
99     /* USER CODE BEGIN W1_HardFault_IRQn 0 */
100    /* USER CODE END W1_HardFault_IRQn 0 */
101  }
102 }
103
104 /**
105   * @brief This function handles Memory management fault.
106   */
107 void MemManage_Handler(void)
108 {
109   /* USER CODE BEGIN MemoryManagement_IRQn 0 */
110
111   /* USER CODE END MemoryManagement_IRQn 0 */
112   while (1)
113   {
114     /* USER CODE BEGIN W1_MemoryManagement_IRQn 0 */
115     /* USER CODE END W1_MemoryManagement_IRQn 0 */
116   }
117 }
118
119 /**
120   * @brief This function handles Pre-fetch fault, memory access fault.
121   */
122 void BusFault_Handler(void)
123 {
124   /* USER CODE BEGIN BusFault_IRQn 0 */
125
126   /* USER CODE END BusFault_IRQn 0 */
127   while (1)
128   {
129     /* USER CODE BEGIN W1_BusFault_IRQn 0 */
130     /* USER CODE END W1_BusFault_IRQn 0 */
131   }
132 }
133
134 /**
135   * @brief This function handles Undefined instruction or illegal state.
136   */
137 void UsageFault_Handler(void)
138 {
139   /* USER CODE BEGIN UsageFault_IRQn 0 */
140
141   /* USER CODE END UsageFault_IRQn 0 */
142   while (1)
143   {
144     /* USER CODE BEGIN W1_UsageFault_IRQn 0 */
145     /* USER CODE END W1_UsageFault_IRQn 0 */
146   }
147 }
148
149 /**
150   * @brief This function handles System service call via SWI instruction.
151   */
152 void SVC_Handler(void)
153 {
154   /* USER CODE BEGIN SVCall_IRQn 0 */
155
```

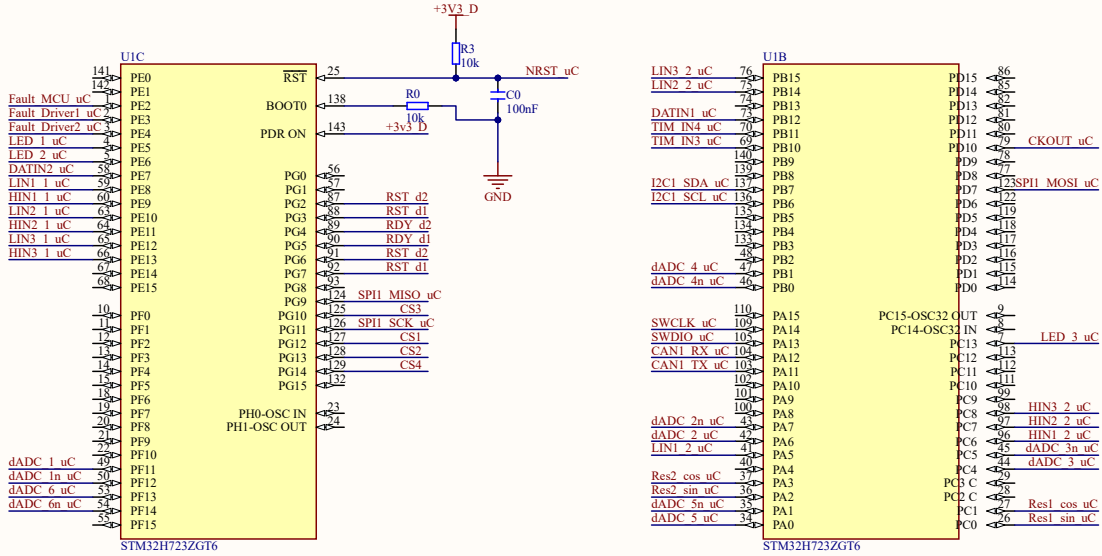
```
156   /* USER CODE END SVCall_IRQn 0 */
157   /* USER CODE BEGIN SVCall_IRQn 1 */
158
159   /* USER CODE END SVCall_IRQn 1 */
160 }
161
162 /**
163   * @brief This function handles Debug monitor.
164   */
165 void DebugMon_Handler(void)
166 {
167   /* USER CODE BEGIN DebugMonitor_IRQn 0 */
168
169   /* USER CODE END DebugMonitor_IRQn 0 */
170   /* USER CODE BEGIN DebugMonitor_IRQn 1 */
171
172   /* USER CODE END DebugMonitor_IRQn 1 */
173 }
174
175 /**
176   * @brief This function handles Pendable request for system service.
177   */
178 void PendSV_Handler(void)
179 {
180   /* USER CODE BEGIN PendSV_IRQn 0 */
181
182   /* USER CODE END PendSV_IRQn 0 */
183   /* USER CODE BEGIN PendSV_IRQn 1 */
184
185   /* USER CODE END PendSV_IRQn 1 */
186 }
187
188 /**
189   * @brief This function handles System tick timer.
190   */
191 void SysTick_Handler(void)
192 {
193   /* USER CODE BEGIN SysTick_IRQn 0 */
194
195   /* USER CODE END SysTick_IRQn 0 */
196   HAL_IncTick();
197   /* USER CODE BEGIN SysTick_IRQn 1 */
198
199   /* USER CODE END SysTick_IRQn 1 */
200 }
201
202 /******************************************************************************/
203 /* STM32H7xx Peripheral Interrupt Handlers                                    */
204 /* Add here the Interrupt Handlers for the used peripherals.                  */
205 /* For the available peripheral interrupt handler names,                      */
206 /* please refer to the startup file (startup_stm32h7xx.s).                    */
207 /******************************************************************************/
208
209 /**
210   * @brief This function handles DMA1 stream0 global interrupt.
211   */
212 void DMA1_Stream0_IRQHandler(void)
213 {
214   /* USER CODE BEGIN DMA1_Stream0_IRQn 0 */
215   //HAL_ADC_Stop_DMA(&hadc1);
216   /* USER CODE END DMA1_Stream0_IRQn 0 */
217   HAL_DMA_IRQHandler(&hdma_adc1);
218   /* USER CODE BEGIN DMA1_Stream0_IRQn 1 */
219
220   /* USER CODE END DMA1_Stream0_IRQn 1 */
221 }
222
223 /**
224   * @brief This function handles DMA1 stream1 global interrupt.
225   */
226 void DMA1_Stream1_IRQHandler(void)
227 {
228   /* USER CODE BEGIN DMA1_Stream1_IRQn 0 */
229   //HAL_ADC_Stop_DMA(&hadc2);
```

```
230   /* USER CODE END DMA1_Stream1_IRQn 0 */
231   HAL_DMA_IRQHandler(&hdma_adc2);
232   /* USER CODE BEGIN DMA1_Stream1_IRQn 1 */
233
234   /* USER CODE END DMA1_Stream1_IRQn 1 */
235 }
236
237 /**
238   * @brief This function handles EXTI line[9:5] interrupts.
239   */
240 void EXTI9_5_IRQHandler(void)
241 {
242   /* USER CODE BEGIN EXTI9_5_IRQn 0 */
243   // Blue user button changes the PWM State of the system
244   if (HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET)
245     {
246     if (Cur_PWM_State == PWM_ON){
247       Cur_PWM_State = PWM_OFF;
248     }
249     else{
250       Cur_PWM_State = PWM_ON;
251     }
252     }
253   __HAL_GPIO_EXTI_CLEAR_IT(B1_Pin);
254   /* USER CODE END EXTI9_5_IRQn 0 */
255   HAL_GPIO_EXTI_IRQHandler(USB_FS_OVCR_Pin);
256   /* USER CODE BEGIN EXTI9_5_IRQn 1 */
257
258   /* USER CODE END EXTI9_5_IRQn 1 */
259 }
260
261 /**
262   * @brief This function handles TIM1 update interrupt.
263   */
264 void TIM1_UP_IRQHandler(void)
265 {
266   /* USER CODE BEGIN TIM1_UP_IRQn 0 */
267
268   /* USER CODE END TIM1_UP_IRQn 0 */
269   HAL_TIM_IRQHandler(&htim1);
270   /* USER CODE BEGIN TIM1_UP_IRQn 1 */
271
272   /* USER CODE END TIM1_UP_IRQn 1 */
273 }
274
275 /**
276   * @brief This function handles TIM4 global interrupt.
277   */
278 void TIM4_IRQHandler(void)
279 {
280   /* USER CODE BEGIN TIM4_IRQn 0 */
281   // call the pulse function to phase shift the outputs
282   Pulse(Cur_Master_Count_Step);
283   // iterate the counter
284   if(Cur_Master_Count_Step == Step_3){
285     Cur_Master_Count_Step = Step_1;
286   }
287   else{
288     Cur_Master_Count_Step++;
289   }
290
291   // Let the interrupt iterate the commutation sequence for testing without back EMF
292   /*
293   Test++;
294   if(Test>5){
295     if(Cur_Commutation_State >= step_6){
296       Cur_Commutation_State = step_1;
297     }
298     else{
299       Cur_Commutation_State++;
300     }
301     Execute_Commutation(Cur_Commutation_State);
302     Test = 0;
303   }
```

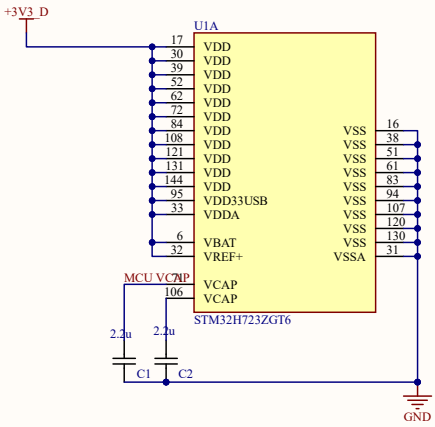
```
304   */
305
306
307
308   /* USER CODE END TIM4_IRQn 0 */
309   HAL_TIM_IRQHandler(&htim4);
310   /* USER CODE BEGIN TIM4_IRQn 1 */
311
312   /* USER CODE END TIM4_IRQn 1 */
313 }
314
315 /* USER CODE BEGIN 1 */
316 /*
317 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
318   if(htim->Instance == TIM4){
319     Pulse(Cur_Master_Count_Step);
320     Cur_Master_Count_Step++;
321   }
322 }
323 */
324 // button interrupt function
325 void EXTI15_10_IRQHandler(void)
326 {
327   // Blue user button changes the PWM State of the system
328   if (HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET)
329     {
330     if (Cur_PWM_State == PWM_ON){
331       Cur_PWM_State = PWM_OFF;
332     }
333     else{
334       Cur_PWM_State = PWM_ON;
335     }
336     }
337   __HAL_GPIO_EXTI_CLEAR_IT(B1_Pin);
338 }
339 /* USER CODE END 1 */
```

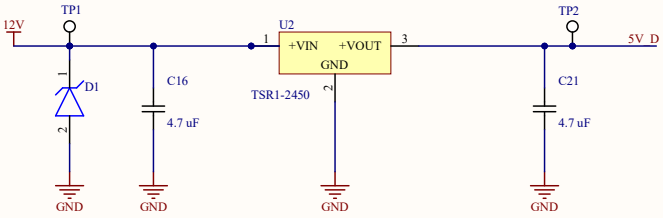
Appendix B

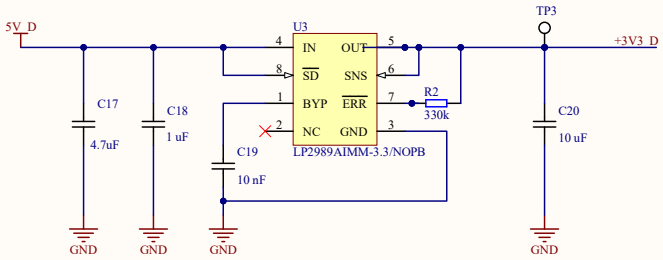
Altium Schematics

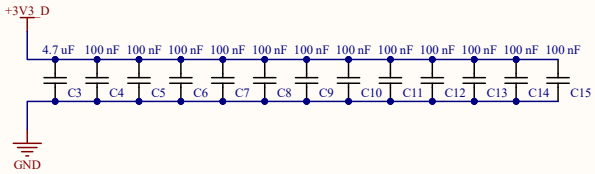


Title		
Size: A4	Number	Revision
Date: 6.04.2023		Sheet of
File: C:\Users\..\uC.SchDoc		Drawn By: Lars Johann Indreeide

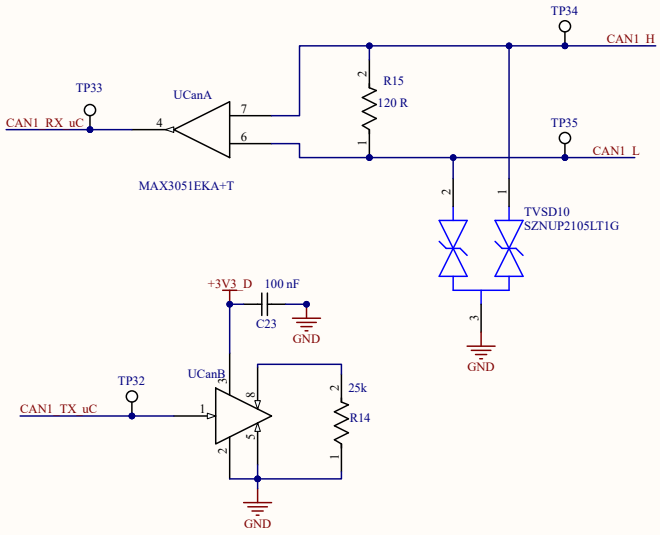








Title		
Size A4	Number	Revision
Date: 6.04.2023		Sheet of
File: C:\Users\..\MCU_Power.SchDoc		Drawn By: Lars Johann Indreeide



Title			
Size: A4	Number		Revision
Date: 6.04.2023		Sheet of	
File: C:\Users\..\CAN.SchDoc		Drawn By: Lars Johann Indreeide	

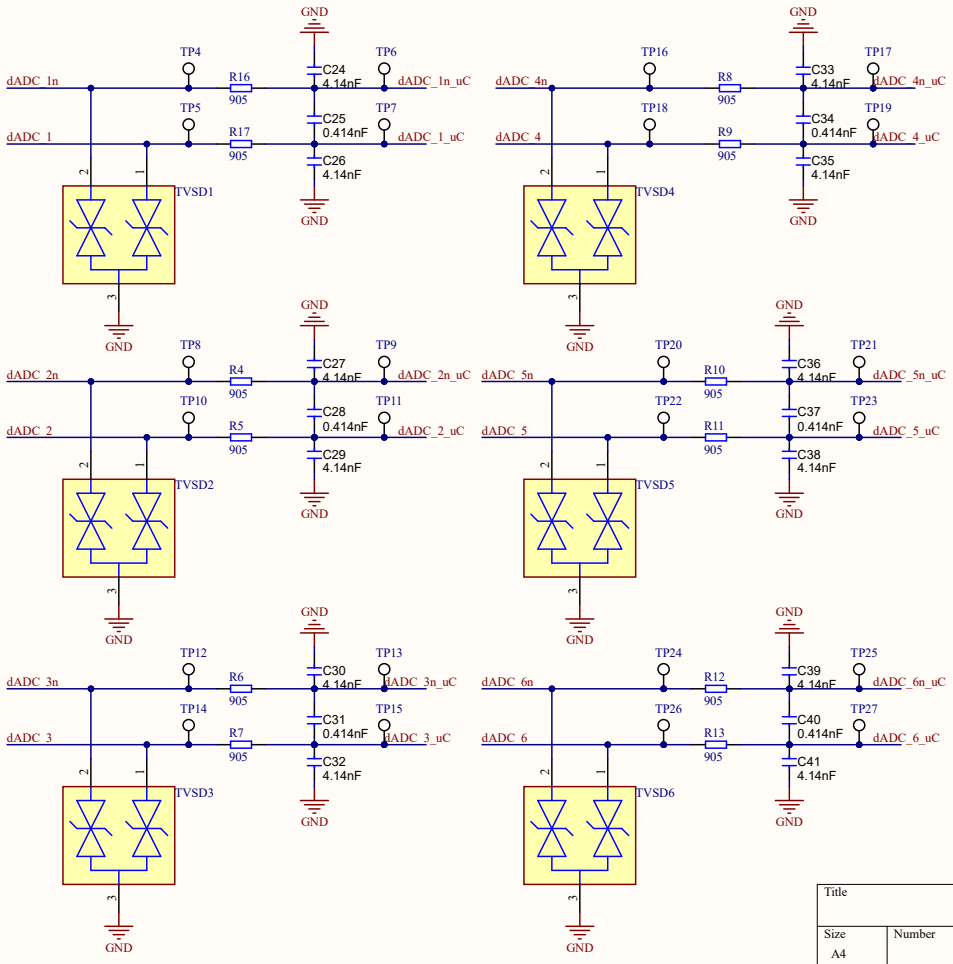
Calculations for filter

V_ov = 20
V_esd = 5.3V
I_max = 20mA
f_c = 21.2kHz

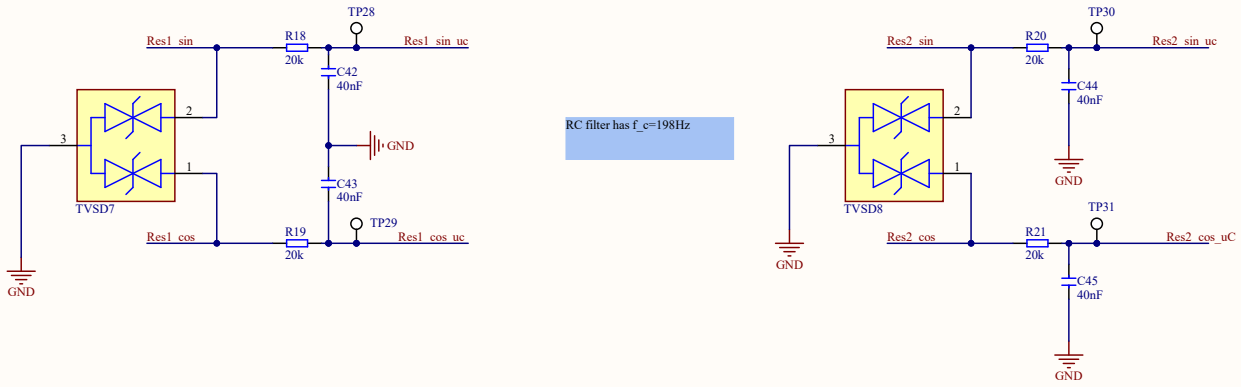
R_flt>(V_ov-V_esd)/(I_max)
C_diff = 1/(2*pi*f_c*(2*R_flt))
C_cm=C_diff/10

This gives R_flt=905 Ohm minumum and C_diff = 4.14nF C_cm=414pF

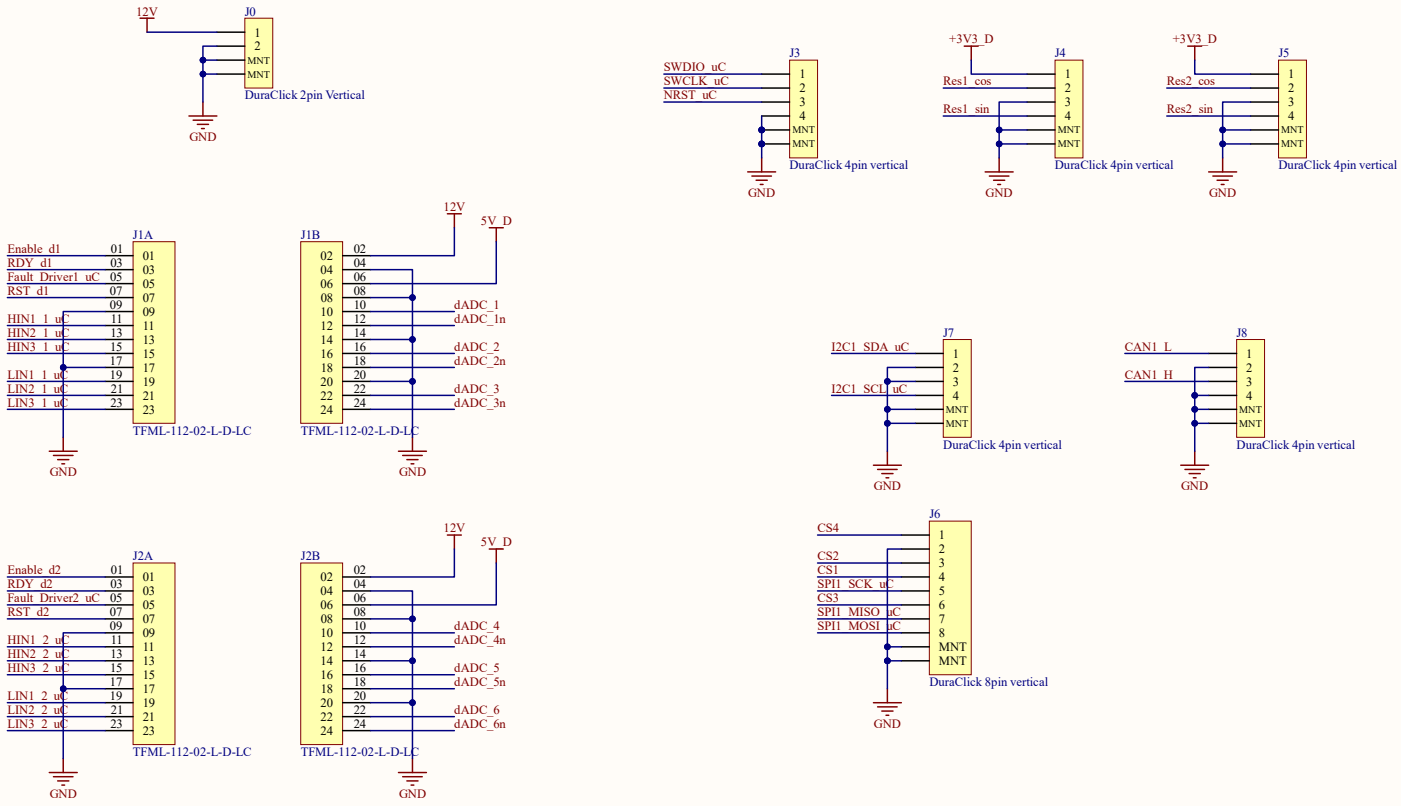
Signals from 0-2.5v
common modes 1.44V



Title		
Size: A4	Number	Revision
Date: 6.04.2023		Sheet of
File: C:\Users\..\dADC.SchDoc		Drawn By: Lars Johann Indreeide



Title		
Size: A4	Number	Revision
Date: 6.04.2023		Sheet of
File: C:\Users\..\Res.SchDoc		Drawn By: Lars Johann Indreeide



Title		
Size: A4	Number	Revision
Date: 6.04.2023		Sheet of
File: C:\Users\..\Connectors.SchDoc		Drawn By: Lars Johann Indreeide