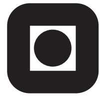

NTNU – Trondheim
Norwegian University of
Science and Technology

Design and Testing of Voltage Source Inverter and Motor Control System for Electric Vehicle

Lars Helge Opsahl

Master of Energy and Environmental Engineering
Submission date: July 2015
Supervisor: Lars Einar Norum, ELKRAFT

Norwegian University of Science and Technology
Department of Electric Power Engineering

Problem Description

The objective of this Master's thesis is to design and develop the voltage source inverter and motor control system for an electric racing car. The vehicle is meant to represent the team Revolve NTNU in the international racing competition Formula Student, and take part in the competitions at Silverstone in the UK and at Red Bull ring in Austria during the summer of 2015.

The design will be based on a 2-level voltage source inverter structure with insulated-gate bipolar transistors (IGBTs), including in-house designed control circuit board, program code, water-cooled heatsink, and carbon fiber casing. To enhance performance, the system will be optimized for the required operating conditions, while constructed as lightweight and compact as possible.

Relevant technologies and trends from literature and manufacturers will be explored. To find the best solutions, different alternatives will be investigated and discussed. In this process, simulation and modelling tools will be used to evaluate and test concepts. The optimum design will be considered a compromise between performances, low weight, robustness and security. Considering that the driver's safety is of high importance, extensive testing and verification of the design is required.

The thesis work will be the documentation of the design, manufacturing and testing of the inverter and motor control system. It will give information about how the system is operated, explain the design choices, and discuss alternative approaches.

Abstract

With fast rotating permanent-magnet motors and compact battery storage units, the high performance motorsport community has, in recent years, started to utilize the quick response and high torque of electrical drivetrain solutions. By recovering energy, braking into a corner, for then to utilize the same energy, boosting out again, the electrical machine gives increased performance even for the fastest racing cars. The purpose of the work presented in this Master's thesis is to develop and produce the power electronics and motor control system needed to control such an electric machine, where the final product is used to drive a fully electric prototype racing car.

The thesis presents the conceptual work, design choices, prototyping, testing and development of a 2-level voltage source inverter and motor control system. The voltage source is a battery accumulator energy storage unit, while the machine is a three-phase permanent-magnet motor. The voltage source inverter is based on insulated-gate bipolar transistors (IGBTs), controlled by a motor control system called field oriented control. On command from the driver's throttle, the motor control system generates the required stator currents, giving the wanted output torque, supplied from the voltage source of the battery accumulator.

The inverter is the connecting interface between the power source, and the rotating machinery. With the amount of energy and power needed to drive a racing car, safe and reliable operation is important. To ensure the safety of the driver and operating personnel, the system and its safety features have been through extensive testing in a laboratory setup and on the race tracks.

In the system development process, simulation and modelling tools have been applied to investigate different solutions and technologies. These tools have also been used to design the electric circuits, program code and mechanical design needed for the system. The final product is installed in a racing car and shows great performance.

Sammendrag

Utviklingen av små, raske permanent-magnet-motorer og kompakte batteriløsninger, har ført til økt interesse for elektriske drivsystemløsninger i racing-miljøet. Ved å regenerere energi under nedbremsingen inn mot en sving, for så å benytte den samme energien, til å aksellerere ut av svingen igjen, gir økt ytelsen for selv de sprekeste racerbilene. Hensikten med arbeidet som presenteres i denne masteroppgaven, er å utvikle kraftelektronikken og motorkontrollsystemet for å styre en slik elektrisk maskin. Det endelige motorkontrollsystemet vil bli benyttet til å drive en elektrisk prototype-racerbil.

I masteroppgaven presenteres konseptarbeid, designvalg, prototyping, testing og utvikling av en frekvensomformer og motorkontrollsystem. Spenningskilden til frekvensomformeren er en batteriakkumulator, mens maskinen er en tre-fase permanent-magnet-motor. Omformeren er basert på IGBT-transistorer, kontrollert av et feltorientert kontrollsystem. Når bilføreren trykker ned pådrags-pedalen, vil motorkontrollsystemet genererer de nødvendige statorstrømmene fra batteriakkumulatoren som gir det ønskede momentetpådraget.

Omformeren står koblet mellom spenningskilden og den roterende maskinen. Med nok energi og effekt til å drive en racerbil, er sikker og pålitelig drift høyt prioritert. For å ivareta sikkerheten til bilføreren og driftspersonell, er systemet og sikkerhetsfunksjonene vært gjennom omfattende testing på laboratoriet og på bane.

Gjennom utviklingsprosessen av systemet, har simulerings- og modellerings-verktøy blitt benyttet til å sammenligne og utforske forskjellige teknologier og løsninger. Disse verktøyene har også vært viktige i designprosessen av de elektriske kretsene, programkoden og det mekaniske designet i systemet. Sluttproduktet er nå installert i racerbil og viser gode kjøreegenskaper.

Acknowledgements

I would like to express my gratitude to Bård Almås, Aksel Hanssen and Svein Erling Norum from the department of electrical engineering at NTNU, for help and guidance on practical matters and for lending me the needed equipment for laboratory testing. In this matter, I would also like to thank Morten Flå and Dominik Häger from the department workshop for assistance on prototyping and manufacturing parts needed for testing.

Design and construction of an electric racing car is a team effort. Without a strong and well organized team like Revolve NTNU, this project could never have been realized. A special thanks goes to Kjetil Kjeka for circuit board design and programming. Also great thanks to Simen August Tinderholt and the rest of the team.

To be able to design and build a racing car, the team is dependent on sponsors for financial backup, specialized equipment, and technical knowhow. I would therefore like to thank all of Revolve NTNU's sponsors for their support. A special thanks to Kjell Ljøkelsøy from Sintef Energy for multiple consultations concerning circuit board design, component specifications and general troubleshooting.

Last but not least, I would like to thank my supervisor Lars Einar Norum, for guidance and for asking the critical questions during the initial design and testing of the system.

Preface

The goal of this Master's thesis has been to develop and produce the voltage source inverter and motor control system for a Formula Student electric racing car. The car is designed and built by a team of NTNU students and is meant to be raced in the annual Formula Student competitions. My responsibility in the team has been the design, manufacturing, testing and development of the power electronic system needed for driving the electrical vehicle. Because Formula Student is a racing competition, the system has been optimized for high performance at certain operating conditions, while also being constructed as lightweight as possible.

Writing my Master's thesis on a development project like this, brings in several crossing engineering fields, creative teamwork, and the possibility to bring a theoretical design into practice over a short period of time. The voltage source inverter is a complex system and critical for the operation of the vehicle. It is the connecting interface between a power source of 288 lithium-ion battery cells, and a high torque, fast rotating machine. The amount of energy and power, brings in the need for well-designed safety features to ensure the safety of the drivers and the operating personnel. This has implied extensive testing and design verification, in a laboratory environment, the manufacturing workshop and at the race circuit.

The thesis is meant to give insight into the conceptual work, design choices, prototyping and processes of testing and developing the complete motor control system. The underlying theory is explained, while the models, simulations, program code, mechanical and electrical designs presented. The results can be used as guidance for the next students interested in similar challenges.

30.07.2015

Lars Helge Opsahl

Abbreviations

4WD	:	Four wheel drive
AC	:	Alternating current
ADC	:	Analog-to-digital converter
AFEC	:	Analog front end controller
AIR	:	Accumulator isolation relay
BMS	:	Battery management system
BOT	:	Brake over-travel switch
BSPD	:	Brake system plausibility device
CAD	:	Computer aided design
CAN	:	Controller area network
CFD	:	Computational fluid dynamics
cpr	:	Counts per revolution
DC	:	Direct current
DSP	:	Digital signal processor
DTC	:	Direct torque control
ECU	:	Engine control unit
EEFC	:	Enhanced embedded flash controller
EMI	:	Electromagnetic interference
ERS	:	Energy recovery system
ESD	:	Electrostatic discharge
ESR	:	Equivalent series resistance
EV	:	Electric vehicle
FOC	:	Field oriented control
FPGA	:	Field programmable gate array
FPU	:	Floating point unit
FRAM	:	Ferroelectric random-access memory
GLV	:	Grounded low voltage
HVD	:	High voltage disconnector

IC	:	Internal combustion
IDP	:	Integrated development platform
IGBT	:	Insulated-gate bipolar transistor
IMD	:	Insulation monitoring device
IMU	:	Inertial measurement unit
KERS	:	Kinetic energy recovery system
LED	:	Light emitting diode
$LiCoO_2$	:	Lithium cobalt oxide
MCU	:	Microcontroller
MDF	:	Medium density fiberboard
MOSFET	:	Metal-oxide-semiconductor field-effect transistor
PCB	:	Printed circuit board
PIO	:	Parallel input output
PM	:	Permanent-magnet
PMC	:	Power management controller
ppr	:	Pulses per revolution
PWM	:	Pulse-width modulation
rpm	:	Revolutions per minute
SAE	:	Society of Automotive Engineers
Si	:	Silicon
SiC	:	Silicon carbide
SMD	:	Surface mount device
SoC	:	System-on-chip
SPI	:	Serial peripheral interface
SSI	:	Synchro-serial interface
SV	:	Space vector
TPS	:	Torque position sensor
VSI	:	Voltage source inverter

Table of Contents

Problem Description I

Abstract III

Sammendrag V

Acknowledgements VII

Preface IX

Abbreviations XI

Table of Contents XIII

List of Figures XVII

List of Tables XXIII

1. Introduction 1
   - 1.1. Background 1
     - 1.1.1. Formula Student 1
     - 1.1.2. Revolve NTNU 4
   - 1.2. Motivation 6
   - 1.3. The Electrical Drivetrain 7
     - 1.3.1. Permanent-Magnet Synchronous Motor 8
     - 1.3.2. Engine Control Unit and Traction Control 11
     - 1.3.3. Battery Accumulator and Safety Systems 12
     - 1.3.4. Voltage Source Inverter 14

2. Motor and Motor Control Theory 18
   - 2.1. Park and Clark Transformation 18
   - 2.2. Pulse Width Modulation 20
   - 2.3. Space Vector Modulation 21
   - 2.4. Field Oriented Control 22
   - 2.5. Direct Torque Control 24

3. Sensors and Measurements 28
   - 3.1. Closed Loop Current Transducer 28
   - 3.2. Magnetic Shaft Encoder 30

3.3. Voltage Measurement 33

4. Control System Simulation 34
4.1. Field Oriented Control Simulation 34
4.2. Direct Torque Control Simulation 37

5. Control System Hardware 39
5.1. Circuit Board Design 39
5.2. Data and Computational Processing Unit 41
5.3. Circuit Board Production 42

6. Control System Software 44
6.1. Motor Control Program Code 44
6.2. Field Oriented Control Code 46
6.3. CAN Messages and Error Status 47

7. Voltage Source Inverter Design 49
7.1. Power Semiconductor Transistors 49
7.2. Gate Drive Circuit 54
7.3. Input Capacitor Bank 56
7.4. Discharge Circuit 58
7.5. Heatsink Design 61
7.6. Voltage Source Inverter Casing Design 64
7.6.1. Placement in the Monocoque 65
7.6.2. DC Side Plate Conductors 66
7.7. Cables and Contacts 67

8. Assembly and Manufacturing 69
8.1. Prototype Voltage Source Inverter Assembly 69
8.2. Heatsink Manufacturing 70
8.3. Carbon Fiber Casing Manufacturing 71
8.4. Voltage Source Inverter Assembly 74
8.5. Assembly in the Monocoque 77

9. Motor and Motor Control Testing 79
9.1. Initial Laboratory Test Setup 79
9.2. Laboratory Test Setup with Load 81

9.3. Laboratory Test Setup with Load and Battery Accumulator 85
9.4. Testing in the Car 89
10. Discussion 93
11. Conclusion 95
12. Further Work 97
13. Bibliography 98
A. Problems and Issues During Testing 100
B. Program Code for Motor Control 107
B.1: inverter_control.c 108
B.2: fpu.h 114
B.3: InverterDrivers 115
B.4: Modules 127
B.5: Settings 133
B.6: RevolveDrivers 135
B.7: util 149
B.8: revolve_can_definitions.h 150
C. Inverter CAN Messages and Errors 153
D. Inverter Prototype Setup Manual 158
E. Simulink Model of the Battery Accumulator 164
F. Simulink Control System Models 169
G. Voltage Source Inverter Wiring Harness 173
H. Schematic Drawings 174

List of Figures

Figure 1.1: The competing teams from Formula Student UK at Silverstone, 2014. 2

Figure 1.2: Endurance course map at Silverstone. 3

Figure 1.3: Revolve team 2015 at Silverstone race circuit. 4

Figure 1.4: Revolve NTNU's cars after the unwailing of Vilje, April 2015. 5

Figure 1.5: Emrax 228 PM motor from Enstroj. 9

Figure 1.6: Emrax 228 Torque and power characteristics. 11

Figure 1.7: Overview of the electrical drivetrain with safety systems. 13

Figure 1.8: Overview of the electrical drivetrain. 14

Figure 1.9: Three-phase alternating current waveforms. 15

Figure 1.10: 2-level voltage source inverter structure. 15

Figure 1.11: Bamocar d3 voltage source inverter. 16

Figure 2.1: Stator current vector in the abc reference frame. 18

Figure 2.2: Park and Clark transformation. 20

Figure 2.3: Triangular carrier pulse width modulation. 21

Figure 2.4: Available voltage space vectors. 22

Figure 2.5: Field oriented control schematic structure. 23

Figure 2.6: Stator and rotor flux linkages. 24

Figure 2.7: Optimal selection of voltage space vectors. 25

Figure 2.8: Direct torque control schematic structure. 26

Figure 3.1: Closed loop current transducer functionality. 28

Figure 3.2: LEM LA 150-P current sensor. 29

Figure 3.3: LEM LF 205-S current sensor. 30

Figure 3.4: Incremental ABZ encoder functionality. 31

Figure 3.5: Magnetic shaft encoder from RLS 32

Figure 3.6: DC bus voltage measurement. 33

Figure 4.1: PM motor and IGBT Simulink models. 34

Figure 4.2: Simulink model of FOC with PWM speed control, speed plot. 35

Figure 4.3: Simulink model of FOC with PWM torque control, torque plot. 35

Figure 4.4: Simulink model of FOC with hysteresis speed control, torque plot. 36

Figure 4.5: Simulink model of FOC with SV PWM speed control, speed plot. 37

Figure 4.6: Simulink model with torque estimation. 38

Figure 4.7: Simulink model of stator flux drift. 38

Figure 5.1: Inverter control circuit board schematic drawing. 39

Figure 5.2: Inverter control circuit board rendering. 40

Figure 5.3: Control circuit board production. 42

Figure 5.4: X-ray image of the microcontroller on the control circuit board. 43

Figure 6.1: Control circuit board programming. 44

Figure 6.2: Inverter control program. 45

Figure 7.1: Inside of a power semiconductor module [13]. 49

Figure 7.2: Semitrans 3 power semiconductor module. 50

Figure 7.3: SEMiX power semiconductor module. 50

Figure 7.4: IGBT structure with silicon layers [5]. 52

Figure 7.5: Turn-on charcteristics for a power semiconductor transistor [13] 53

Figure 7.6: Gate driver functionality schematic. 54

Figure 7.7: SKHI 23/12 R gate driver from Semikron. 55

Figure 7.8: Cornell Dubilier 944U metallized polypropylene film capacitor. 58

Figure 7.9: Discharge voltage and current waveforms [1]. 59

Figure 7.10: Welwyn WH100 discharge resistor. 59

Figure 7.11: Discharge control circuit board. 60

Figure 7.12: Cynergy3 discharge reley. 60

Figure 7.13: Thermal heatsink model [13]. 61

Figure 7.14: Heat sink rendering, bottom side. 62

Figure 7.15: Heat sink rendering, top side, with IGBT module. 63

Figure 7.16: Heatsink simulation of temperature distribution. 63

Figure 7.17: Voltage source inverter casing rendering, front side. 64

Figure 7.18: Voltage source inverter casing rendering, rear side. 65

Figure 7.19: Placement in the monocoque [1]. 66

Figure 7.20: DC side plate conductors [18]. 67

Figure 8.1: Voltage source inverter prototype, right side. 69

Figure 8.2: Voltage source inverter, rear side. 69

Figure 8.3: Voltage source inverter prototype, left side. 70

Figure 8.4: Aluminum heatsink with base plate and nipples. 71

Figure 8.5: Carbon fiber casing mold. 72

Figure 8.6: Carbon fiber casing production. 72

Figure 8.7: Carbon fiber casing vacuum packing. 73

Figure 8.8: Carbon fiber casing production result. 73

Figure 8.9: Voltage source inverter assembly, tractive system section. 74

Figure 8.10: Voltage source inverter assembly wiring. 75

Figure 8.11: Positive locking of the AC terminal contacts. 75

Figure 8.12: Positive locking of the conductor plates. 76

Figure 8.13: Voltage source inverter assembly from above. 76

Figure 8.14: Voltage source inverter, front side. 77

Figure 8.15: Motor room assembly, voltage source inverter and PM motor. 78

Figure 8.16: Energy meter attached to the voltage source inverter. 78

Figure 9.1: Gate-emitter voltage measurement during initial testing. 79

Figure 9.2: Supply voltage source for initial test setup. 80

Figure 9.3: Initial test setup with static load. 80

Figure 9.4: Software PWM with static load. 81

Figure 9.5: Testing with a synchronous motor in laboratory test setup. 82

Figure 9.6: Initial PM motor test rig, front side. 82

Figure 9.7: Initial PM motor test rig, rear side. 83

Figure 9.8: Laboratory test setup schematic drawing. 84

Figure 9.9: Test setup with PM motor and DC machine load. 84

Figure 9.10: Battery accumulator in the laboratory test setup. 85

Figure 9.11: Test setup with torque pedal, ECU and dashboard. 86

Figure 9.12: Phase current and DC voltage waveforms under heavy load conditions. 87

Figure 9.13: Common-mode voltage measured in the battery accumulator container. 87

Figure 9.14: Voltage and phase current ripples from control loop frequency. 88

Figure 9.15: Current ripples from control loop frequency. 88

Figure 9.16: Test setup with battery accumulator and load. 89

Figure 9.17: Phase current during testing of the car on rig with no load. 89

Figure 9.18: Initial testing of the car on rig. 90

Figure 9.19: Initial testing of the car at parking lot at NTNU. 90

Figure 9.20: Revolve Analyze, dq currents and torque request. 91

Figure 9.21: Revolve Analyze, inverter plugin. 91

Figure 11.1: Final rendering of Vilje, front side. 95

Figure A.1: Control circuit board prototype fix to include PWM. 100

Figure A.2: ABZ encoder fix on synchronous motor. 101

Figure A.3: Current and voltage waveforms during sensor saturation. 102

Figure A.4: Loose cable lug on PM motor phase. 103

Figure A.5: PM motor assembly at Silverstone . 103

Figure A.6: IGBT and heatsink failure. 104

Figure A.7: Inside of failed IGBT. 105

Figure D.1: Voltage source inverter prototype setup. 158

Figure D.2: Pin connection on the IGBT modules. 159

Figure D.3: Gate driver connection. 160

Figure D.4: Control circuit board connections. 161

Figure D.5: Motor phases connection. 162

Figure D.6: Encoder connection. 163

Figure E.1: Battery accumulator model in Simulink. 164

Figure E.2: Battery module configuration in Simulink. 165

Figure E.3: Battery cell configuration in the battery module in Simulink. 166

Figure E.4: Li-ion battery cell model in Simulink. 167

Figure E.5: Bock parameter settings for Li-ion battery cell model in Simulink. 168

Figure F.1: Simulink model of field oriented sontrol with PWM... ……………………….169

Figure F.2: Simulink model of field oriented control with hysteresis. ……………………….170

Figure F.3: Simulink model of field oriented control with SV PWM. ……………………….171

Figure F.4: Simulink model for direct torque control flux estimation. ………………………172

Figure G.1: Voltage source inverter wiring harness. ………………………………………...173

Figure H.1: Schematic drawing of the inverter control board. ………………………………174

Figure H.2: Schematic drawing of the power modules. ……………………………………...175

Figure H.3: Schematic drawing of the 24V input filter. ……………………………………...176

Figure H.4: Schematic drawing of the 5V and 3V3 supply. …………………………………176

Figure H.5: Schematic drawing of the 5V ADC circuit. ……………………………………..177

Figure H.6: Schematic drawing of the 15V supply. ………………………………………….177

Figure H.7: Schematic drawing of the negative 15V supply. ………………………………..177

Figure H.8: Schematic drawing of the CAN transceiver circuit. ……………………………177

Figure H.9: Schematic drawing of the computational circuitry. ……………………………..178

Figure H.10: Schematic drawing of the gate driver interface circuit. ………………………179

Figure H.11: Schematic drawing of the encoder interface circuit. …………………………...179

Figure H.12: Schematic drawing of the encoder with ABZ interface circuit. ………………..179

Figure H.13: Schematic drawing of current sensor interface. ………………………………..180

Figure H.14: Schematic drawing of the voltage measurement circuit. ………………………180

Figure H.15: Schematic drawing of the temperature measurement circuit from heatsink. …..181

Figure H.16: Schematic drawing of the temperature measurement circuit from motor. ……..181

Figure H.17: Schematic drawing of the FRAM circuitry. …………………………………...181

Figure H.18: Schematic drawing of the discharge circuit. …………………………………..182

List of Tables

Table 1.1: Formula Student competition evaluation. 2

Table 1.2: Emrax 228 PM motor characteristics. 10

Table 1.3: Bamocar d3 specifications. 17

Table 2.1: Optimal selection of non-zero voltage space vectors. 27

Table 6.1: CMSIS-DSP software library functions. 47

Table 6.2: Status bits in Revolve Analyze. 48

Table 7.1: Comparison of power semiconductor module characteristics. 51

Table 7.2: Alternative capacitors for input capacitor bank. 57

Table 7.3: Voltage source inverter contacts. 68

Table C.1: CAN messages sent continusly during operation from the VSI. 153

Table C.2: CAN messages reporting the states of the VSI. 154

Table C.3: CAN messages with set points for the dq-current controllers. 154

Table C.4: CAN messages with tuning parameters for the current controllers in the VSI. 155

Table C.5: CAN messages for tuning voltage levels limiting torque request. 155

Table C.6: CAN messages for tuning power limiting of the torque request. 156

Table C.7: CAN messages for tuning speed limitation of the torque request. 156

Table C.8: CAN messages for tuning temperature limitation of the torque request. 157

Table D.1: Pin connection on the IGBT modules. 159

Table D.2: Pin connection for the encoder. 163

1. Introduction

1.1. Background

The electrical machine has for a long time been considered superior to the internal combustion engine in many applications due to its robust design, high efficiency and advantageous torque-weight characteristics. The electrical machine is the clear choice for most industrial applications with high torque requirements and possibility for grid connection. Lately, this has also been picked up in the high performance motorsport community.

In 2009, the use of electric drive systems were introduced in Formula 1 as kinetic energy recovery systems (KERS) were legalized. Because of this, teams started to develop drivetrains using electric machines and batteries to recover braking energy. In 2014, this system was redefined and reintroduced as the energy recovery system (ERS), allowing each vehicle to regenerate up to 4 MJ of energy per lap and return up to 120 kW to the drivetrain. In the last couple of years, this has driven Formula 1 teams to develop hybrid drivetrains supplied by high-speed permanent-magnet (PM) machines, lightweight lithium-ion battery accumulators and highly efficient motor control systems. With 2014 as the year where the first Formula E Championship was held, hybrid-electric and fully electric drivetrain solutions have come to stay in high performance motorsport.

This trend has also been picked up in the Formula Student community, where fully electric powered vehicles have had great success over the last couple of years. On short and fast tracks, as in Formula Student, the high torque and fast response of an electric vehicle (EV) has clear advantages to the internal combustion (IC) engines.

1.1.1. Formula Student

Formula Student is a series of annual competitions where engineering students from all over the world meet to compete with one-seat racing cars. The competitions started out as SAE (Society of Automotive Engineers) Mini Indy at the University of Houston in 1979, but has today expanded to include a number of spinoff events all over the world. For the season of 2015, Revolve NTNU will participate in the Formula Student events at Silverstone in the UK and at Red Bull ring in Austria. Both these events use the original rule set from SAE International [2] with individual supplementary regulations and rules.

Fully electric vehicles (EVs) were introduced for the first time in Formula Student in 2010 and has since that been competing in a separate class to the internal combustion (IC) vehicles. In Formula Student UK on the other hand, both EVs and IC vehicles compete in the same class. Over the past couple of years this has resulted in EV teams winning the event.

Figure 1.1: The competing teams from Formula Student UK at Silverstone, 2014.

The events are divided into eight different challenges, where a total of 1000 points can be obtained for each of the teams. The team with the overall highest score wins the event. As is shown in table 1.1, the total score is distributed between static and dynamic events. The static events are presentation, engineering design and cost analysis. The goal with these events are to differentiate teams on the basis of planning, conceptual work, design, cost and general good engineering practice.

Static Events:		
	Presentation	75
	Engineering Design	150
	Cost Analysis	100
Dynamic Events:		
	Acceleration	75
	Skid-Pad	50
	Autocross	150
	Efficiency	100
	Endurance	300
Total Score:		1000

Table 1.1: Formula Student competition evaluation.

The engineering design score is evaluated by well-known judges from the motorsport community, and therefore seen as one of the most important events from an engineer's perspective.

Figure 1.2: Endurance course map at Silverstone.

The dynamic events are acceleration, skid-pad, autocross, efficiency and endurance. In the acceleration event the goal is to drive 75 m as fast as possible. The objective of the skid-pad is to evaluate the vehicle's cornering ability. This is done by driving as fast as possible on a small and circular track. In autocross the maneuverability and handling are of the greatest importance. This event is held at a tight track with sharp turns, where the driver can drive as fast as possible without thinking about energy consumption.

The endurance event evaluates the overall performance and tests durability and reliability of the vehicle. It is a single heat of about 22 km (12 rounds on the course shown in figure 1.2) with driver change at 11 km. This is one of the most important events because it gives the most points, but also because it is the hardest test for the car and the driver. Score for efficiency is given in the same event, based on readings from an energy meter. The energy meter is supplied by the organizers. It calculates and logs the energy consumption and power usage from the battery accumulator, based on voltage and current measurements.

1.1.2. Revolve NTNU

Revolve NTNU is the racing team representing the Norwegian University of Science and Technology in Formula Student. It is an independent student organization who works on the project of building a new racing car from scratch every year. The 2015 team consists of about 50 students from more than ten different engineering disciplines, who all work on design, production and testing of different parts of the vehicle.

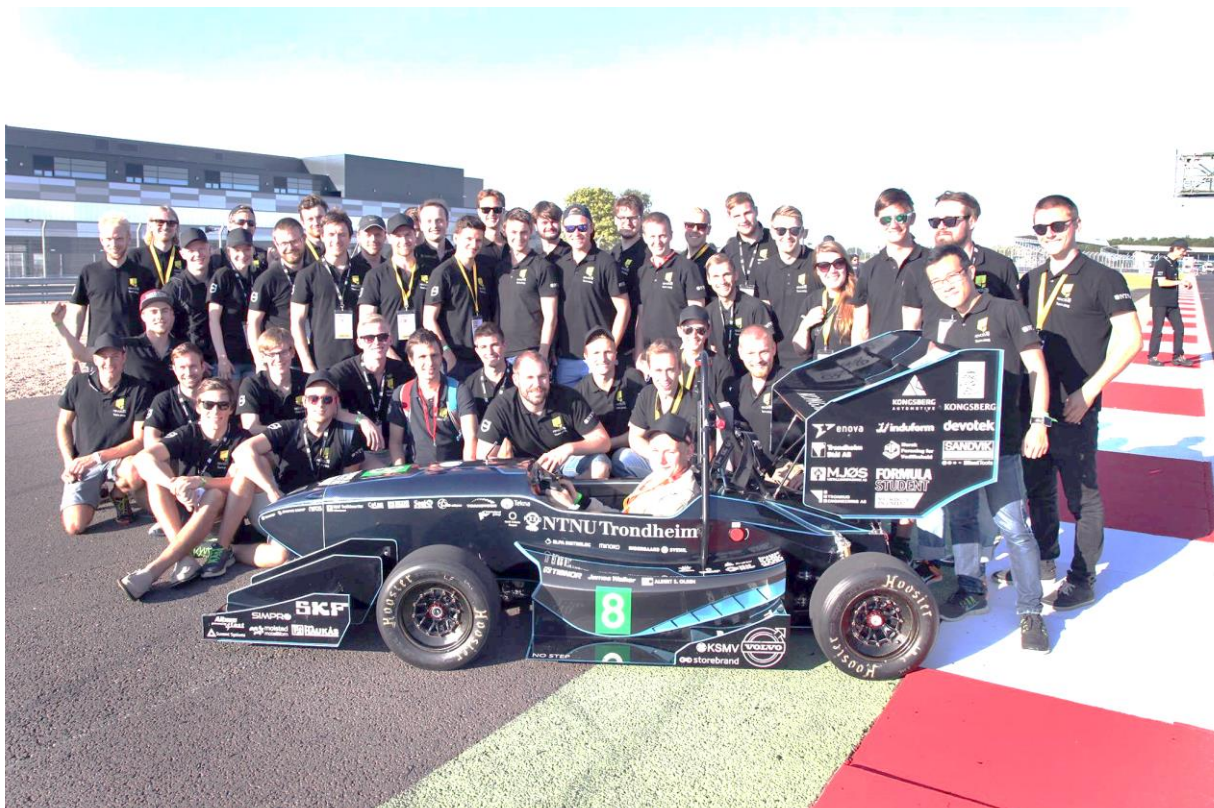

Figure 1.3: Revolve team 2015 at Silverstone race circuit.

Revolve was founded in 2010 and entered the Formula Student competitions for the first time in 2012 with the IC car KA Borealis R (second row, to the left in figure 1.4). The team was

given the “best newcomer award” and finished at a 17th place overall in the UK. The following season, the team made it to a 16th place in the same competition with the IC car KA Aquilo R (second row, in the middle in figure 1.4). For the 2014 season, the team decided to make Revolve’s first fully electric vehicle, which resulted in the car KOG Arctos R (second row, to the right in figure 1.4). With this car the team managed to finish at an 8th place in the UK. For the 2015 season, the team has once again decided to go for EV design. The goals are to reduce the weight and to develop more of the electronic systems in-house. The car, Vilje (first row in figure 1.4), is the newest car, representing Revolve NTNU in the 2015 season.

Figure 1.4: Revolve NTNU’s cars after the unveiling of Vilje, April 2015.

In the 2015 season, Revolve NTNU will bring Vilje to Silverstone in the UK in July and to Red Bull ring in Austria in August, to compete in the Formula Student competitions. The goal is to deliver a strong performance and to challenge the more well-established teams in the competitions.

1.2. Motivation

A fully electric drivetrain has clear advantages to IC engines in Formula Student. This is because the high torque and fast response from an EV fits well with the tracks in the competitions. The tracks are short, with tight and fast turns, where the maximum possible speed is in the range of 110 km/h. The most significant downsides are the need for a heavy battery accumulator, and the weight and complexity of the motor control system. By using lithium-ion battery cells, most battery accumulators in the competition end up weighing 45 – 80 kg. The high weight is unavoidable to obtain the needed energy storage to finish the endurance event. To get the best torque-weight ratio, most teams choose to go for three-phase permanent-magnet (PM) synchronous motors. To drive a motor like this from a battery accumulator, a voltage source inverter (VSI) with appropriate motor control system is needed. The VSI is the unit that converts the direct current (DC) from the battery accumulator to three-phase alternating currents (AC) for the motor. The motor controller is the part of the VSI that determines the amplitude and frequency of the current waveforms, according to the torque request from the driver. This is a system with high complexity, and a system few teams choose to design in-house. At the same time, it is normally also one of the systems with highest potential for weight reduction in the drivetrain of a Formula Student EV. In most commercial VSI systems, the weight is in the order of magnitude of the motor weight, or even higher. With a custom made carbon fiber casing and in-house designed heatsink, the weight is estimated to become as low as a quarter of the motor weight, and at least half the weight of the off-the-shelf VSI from last season.

As mentioned, the VSI is a complex system, and in-house design increases the risks of failure. A well-functioning VSI is crucial to the performance of the vehicle and to the safety of the driver. By designing a VSI that is compatible with the VSI from last season, this risk can be reduced, making the project possible to carry out. It is also considered important for the team to increase the knowledge level about power electronics and motor control. Together with the possibility for weight reduction, this is the main motivation for choosing an in-house design. Increased internal competence on power electronics and motor control theory is crucial for further developments in the electric drivetrain. This is seen as an important step to prepare the team for a possible transition to four wheel drive (4WD) with more sophisticated motor control systems. In-house design also makes it possible to custom fit the VSI to the car, instead of custom fitting the car to the VSI, like most teams do. This way the cables and cooling circuits can be made shorter for even higher weight reductions.

1.3. The Electrical Drivetrain

The Formula Student competitions allow the participating teams to choose between internal combustion and fully electric drivetrains, as long as the system is rule compliant and within a set of limitations. For internal combustion (IC) vehicles the engine must be a four-stroke piston engine with a limit on displacement per cycle of 610 cc. For electric vehicles (EV) the limitation is set to a maximum voltage of 600 V and a maximum power of 80 kW from the battery accumulator. Hybrid drivetrain solutions are prohibited in Formula Student.

For the 2015 season, the choice between IC and EV was based on experience from previous years of competition, and on the knowledge and expertise from the team members. Last season was Revolve's first season with EV design. Compared to the IC vehicle from the year before, this resulted in a considerable increase in the torque-weight ratio, with a weight reduction of more than 60 kg, and at the same time higher output torque. The low weight was a result of using a new construction technique for the chassis and due to the transition to a fully electric drivetrain. The new chassis was a carbon fiber monocoque. A monocoque is a chassis where the skin provides the main structural support of the vehicle and is considerably lighter than a steel frame structure. Except for the battery accumulator, a fully electric drivetrain is a lightweight and compact system. The main drawbacks of an EV are the heavy battery accumulator, a complex motor control system, together with the need for additional safety systems. Experience from previous Formula Student events and results from the last couple of years, still clearly indicated that a well-designed EV normally beats an IC vehicle. Revolve decided to go for EV design for the 2015 season because the team believed this would give the best performance, and because the expertise of the team members was seen suitable for the task.

To drive an EV, one or more electric motors are needed. In the last couple of years, a trend among the more well-established teams has been to use four wheel drive (4WD) systems, with wheel hub mounted permanent-magnet (PM) motors. These are normally custom designed, fast rotating motors, supplying output torque through planetary gear boxes with high gear ratios. The reasons for choosing 4WD systems are to maximize the total traction of the vehicle and to better utilize regenerative braking. Because the braking balance should be shifted towards the front wheels for best performance, 4WD is clearly advantageous to rear wheel drive systems in this matter. A 4WD solution adds weight to the vehicle due to the extra motors, power electronics and cooling circuitry. It also adds complexity to the control systems due to the need for multiple voltage source inverters (VSIs) to control the separate motors. On the other hand,

individual control of four motors makes it possible to implement torque vectoring. A good torque vectoring algorithm distributes the torque between the four wheels in a way that gives optimal traction and regenerative braking capability. This also eliminates the need for a mechanical differential, which reduces weight and increases controllability of the vehicle.

The reason why Revolve NTNU did not choose 4WD for the 2015 season, was because it was considered a large technological step and the risks associated with that, too high. The team also had trouble finding suitable off-the-shelf alternatives for motors. By increasing the competence on power electronics and motor control theory through in-house-designed VSI this season, the team will be better equipped for such a transition next year. The team will then also get more time to find suitable motor alternatives or plan in-house designed solutions.

1.3.1. Permanent-Magnet Synchronous Motor

The electric machine with the highest torque-weight ratio and most suitable for the Revolve design is the permanent-magnet (PM) synchronous motor. The closest alternatives are the DC motor or the asynchronous (induction) motor. The DC motor is the easiest to control, but also the alternative with the lowest torque-weight ratio and lowest robustness. Some of the first-year EV teams in the competitions use this alternative because of the simple motor control. The DC voltage applied to such a motor can be directly controlled by pulse-width modulation (PWM) of a semiconductor transistor, like a metal-oxide-semiconductor field-effect transistor (MOSFET) or an insulated-gate bipolar transistor (IGBT). The asynchronous motor is like the PM motor, more difficult to control due to the need for modulating and controlling three phase-shifted alternating current waveforms. For the asynchronous motor, the rotor is short-circuited and no permanent-magnets used to generate magnetic rotor flux. This makes the motor robust, but introduces a slip factor between the rotational speeds of the rotor and stator magnetic field. The slip factor should be accounted for in the motor control system. The PM motor has a higher torque-weight ratio than the asynchronous motor, but also higher cost. The low cost and high robustness, is the main reason why asynchronous motors are the most used machine in industry and used in the drivetrains of cars like the Tesla Model S and Mercedes B-Electric. For the application in a Formula Student EV, the choice is simple; the additional cost of a PM motor can easily be justified by the higher performance.

Last season, the PM motor Emrax 228 HV from Enstroj was used. This is an axial-flux machine with outer rotor. The outer rotor makes less room for the inner stator coils. This results in lower stator inductance or lower stator current capabilities. On the other side, it leaves more room for

additional pole pairs. As can be seen from equation 1.1, more pole pairs give lower mechanical rotational speed of the motor. For small, high-torque motors, it is normally an issue to keep the rotational speed down. Lower speed also makes it easier to design the gear box system. In equation 1.1, ω_e is the electrical angular speed of the permanent-magnet flux, ω_m the mechanical angular speed of the rotor, and P the number of pole pairs.

$$\omega_e = P \cdot \omega_m \tag{1.1}$$

As can be seen from table 1.2, the stator resistance and the stator inductances for the Emrax 228 are quite low. This suggests that the motor is designed for higher currents, rather than easy controllability. As is clear from equation 1.2, lower stator inductance gives higher current ripples for the same switching frequency. In equation 1.2, v_S is the stator coil voltage, R_S the stator resistance, i_S the stator current and L_S the stator inductance

$$v_S = R_S \cdot i_S + L_S \cdot \frac{d}{dt} i_S \tag{1.2}$$

Figure 1.5: Emrax 228 PM motor from Enstroj.

The Emrax 228 was originally designed to take off and land gliding planes, by a Slovenian engineer with a passion for flying. The motor was meant to make his hobby easier to do, and not for commercial applications. After some time, the motor was discovered by Formula Student teams who wanted to use it for driving their EVs. Today the Emrax design has won awards for highest power-weight ratio and is the preferred choice for most Formula Student EV teams with one motor.

Peak motor power	P_{peak}	100 kW
Continuous motor power	P_{cont}	40 kW
Maximum rotational speed	n_{max}	6000 rpm
Maximum motor torque	T_{max}	240 Nm
Maximum battery voltage	V_{DC}	600 V
Pole pairs	P	10
Stator resistance	R_S	0.018 Ω
Stator inductances (dq axis)	L_d / L_q	175 µH / 180 µH
Permanent-magnet flux	φ_m	0.0542 Wb
Weight	m	12.3 kg

Table 1.2: Emrax 228 PM motor characteristics.

As I clear from table 1.2, the output power of the Emrax 228 is appropriate for an application with a maximum power limit of 80 kW. The maximum rotor speed of 6000 rpm is relatively easy to shift down to a more suitable wheel speed, through a gear box system. 6000 rpm is also considered low compared to teams with in-house designed motors, running on speeds up to 20000 rpm. The low rotational speed is, as explained before, due to the high number of pole pairs. The motor also features a well-designed cooling system, with water-cooled stator and air-cooled rotor, built-in rotor position sensor, and temperature measurement. The drawback of using a motor with outer rotor is that the motor, for safety reasons, has to be built into a shielding housing. This can easily be solved in a lightweight manner by designing the motor shielding, transmission system and mechanical differential into one unit.

Based on experience from last season and from other teams, the Emrax 228 is a good choice for a one-motor drivetrain. The team has not been able to find any other good commercial alternatives. The best alternative to this would be to design and build the motor in-house. This alternative was disregarded due to the risks and the short time of the design period. In-house motor design is a more viable concept for the 2016 season.

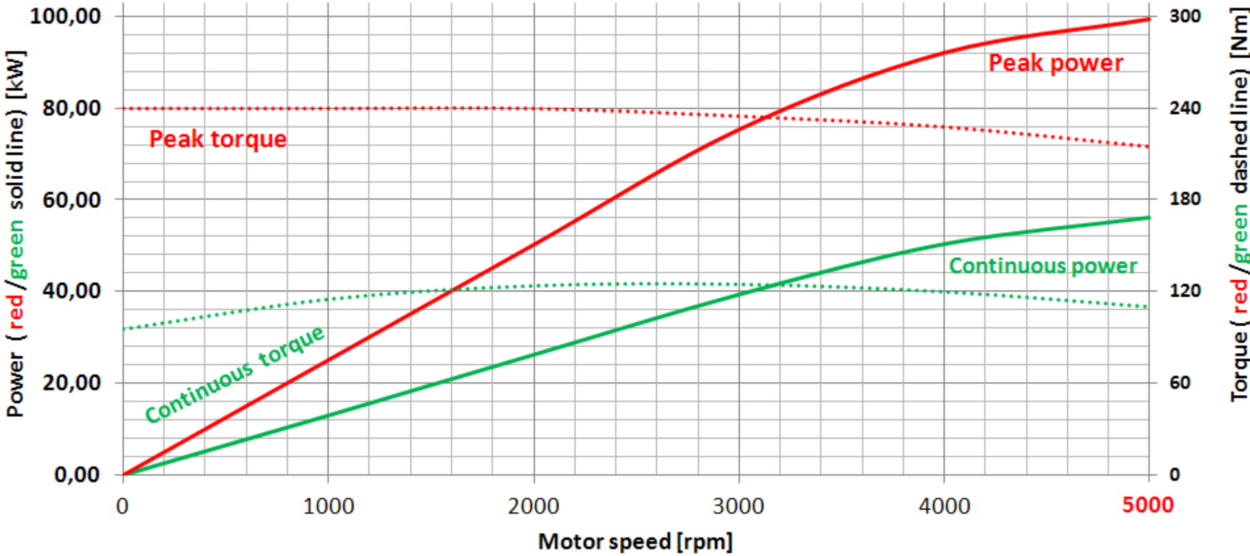

Figure 1.6: Emrax 228 Torque and power characteristics.

1.3.2. Engine Control Unit and Traction Control

The interface between the torque position sensors (TPS) on the torque pedal, and the VSI, is the engine control unit (ECU). The name ECU is misleading, because the unit is not really controlling an engine, but supplying the VSI and motor controller with torque request from the driver. The reason why the name "motor control unit" is not used, is due to the confusion of mixing it with the commonly used abbreviation for microcontrollers (MCU). As can be seen in figure 1.7, the ECU is placed in the front of the vehicle and connected to the VSI through a controller area network (CAN) bus. It is positioned in the front of the vehicle to be close to the torque pedal. On the torque pedal there are two TPS units who both communicate with the ECU on separate CAN busses. The ECU receives the separate TPS values, compares them, and translates the values to a torque request for the VSI. If it is a mismatch of more than ten percent between the TPS units, the ECU sends a torque request of zero, while giving an error message. If the values are within the range, the ECU performs launch control or traction control on the torque request from the driver. Launch control is ideally a function where the driver requests maximum torque for an acceleration run. The ECU will then request the perfect amount of

torque from the VSI. The perfect amount of torque is defined as the point of optimal wheel spin. Traction control has a similar functionality, but with higher time constants. In contrast to launch control, the higher time constants for traction control, makes it possible with feedback control. The inputs to the traction control algorithm are data from wheel speed sensors and from an inertial measurement unit (IMU). The IMU is a system containing four accelerometers, each giving the individual acceleration in three axes. The accelerometers are calibrated by GPS and capable of giving the acceleration, speed and position of the vehicle.

The ECU also provides the power limitation for the drivetrain. From the Formula Student rules, the vehicle is not allowed to use more than 80 kW from the battery accumulator for more then a moving average of 500 ms, or for a total of 100 ms. The ECU gets information about the DC bus current and voltage from measurements in the battery accumulator. Based on this information, it calculates the power, and controls the torque request to keep the power limit from is being exceeded.

1.3.3. Battery Accumulator and Safety Systems

The battery accumulator is built up by 288 lithium cobalt oxide ($LiCoO_2$) cells, where two cells are parallel connected, and a total of 144 parallel connections, connected in series. This gives a nominal accumulator voltage of 535 V and a capacity of about 7.5 kWh, [3]. The battery accumulator is the heaviest part of the vehicle and weighs about 50 kg.

In an EV, the battery accumulator is the most critical part when it comes to safety. The lithium-ion cells need constant supervision to make sure the individual cell voltages and temperatures are within the safe operating area. As can be seen in figure 1.7, the battery management system (BMS) is responsible for this. If the BMS detects a cell voltage or temperature outside of the safe operating area, it shuts down the tractive system through the accumulator isolation relays (AIRs). The AIRs are two DC relays, one connected in series with each of the battery poles, that opens on command from the BMS or if the shutdown circuit is opened. The shutdown circuit goes through most of the car and is opened if any of the safety systems detect an error or a fault. The break system plausibility device (BSPD) opens the shutdown circuit if a positive battery current is measured while the driver is pushing the brake pedal. The insulation monitoring device (IMD) opens the shutdown circuit if an insulation failure in the tractive system is detected. The shutdown circuit is also connected to a break over-travel switch (BOT) in case the brake pedal breaks down, to an inertia switch in case of crash, to a number of

shutdown buttons in the cockpit and outside monocoque, through all the tractive systems contacts, and to a tractive system master switch.

Figure 1.7: Overview of the electrical drivetrain with safety systems.

The high voltage disconnector (HVD) in figure 1.7 is a contact which physically opens the tractive system on the positive pole. This contact is always removed when the car is transported or in storage.

1.3.4. Voltage Source Inverter

As can be seen in figure 1.7 and figure 1.8, the voltage source inverter (VSI) connects the battery accumulator to the three-phase PM motor. The VSI modulates the three-phase current waveforms driving the PM motor in the car. The amplitude and frequency of these currents are determined by the motor controller, based on the torque request from the ECU. As can be seen from equation 1.3, the vehicle can be accelerated by applying an electromagnetic torque T_e that overcomes the load torque T_L from friction and aerodynamic drag, and the viscosity coefficient k_d from the air.

$$T_e = J_{eq} \cdot \frac{d}{dt}\omega_m + k_d \cdot \omega_m + T_L \tag{1.3}$$

Equation 1.3 is derived from Newton's law of motion, where J_{eq} is the equivalent rotor inertia.

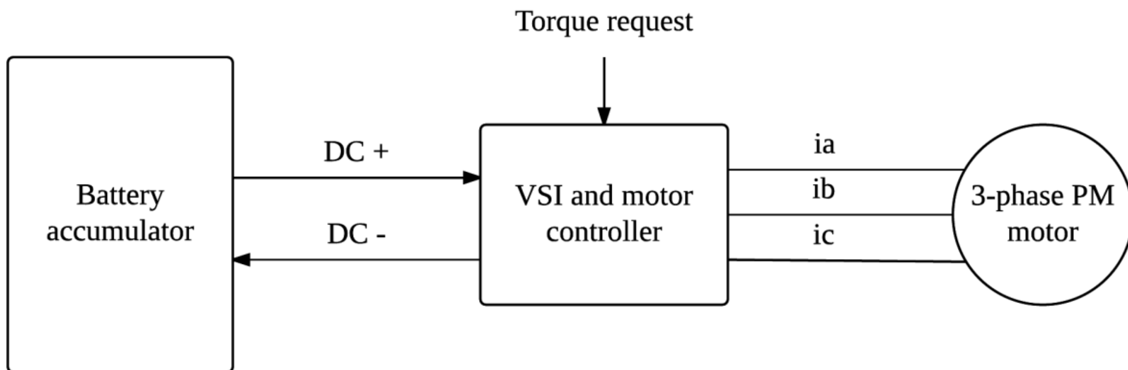

Figure 1.8: Overview of the electrical drivetrain.

As can be seen from equation 1.5, the electromagnetic torque is determined from the permanent-magnet flux, the number of pole pairs and the stator current amplitude I_s.

$$\begin{bmatrix} i_a(t) \\ i_b(t) \\ i_c(t) \end{bmatrix} = I_s \cdot \begin{bmatrix} \sin(\omega_e \cdot t) \\ \sin(\omega_e \cdot t - \frac{2\pi}{3}) \\ \sin(\omega_e \cdot t - \frac{4\pi}{3}) \end{bmatrix} \tag{1.4}$$

$$T_e = \frac{3}{2} P \cdot \varphi_m \cdot I_s \tag{1.5}$$

From equation 1.5, it is clear that for a sinusoidal three-phase PM motor, three phases of sinusoidal current waveforms, phase-shifted by 120 electrical degrees, will produce an output torque. For a PM motor, this torque can be controlled by changing the amplitude and frequency of the stator current waveforms.

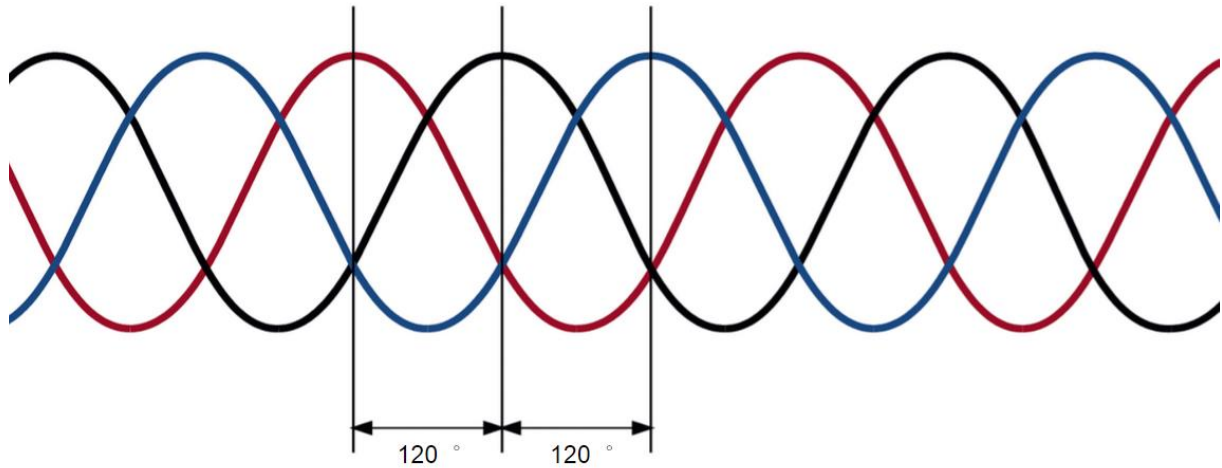

Figure 1.9: Three-phase alternating current waveforms.

Figure 1.10 shows a 2-level voltage source inverter structure. In this structure, the DC voltage of the battery accumulator can be converted to three-phase AC voltages, by modulating sinusoidal waveforms with high frequency switching of power semiconductors.

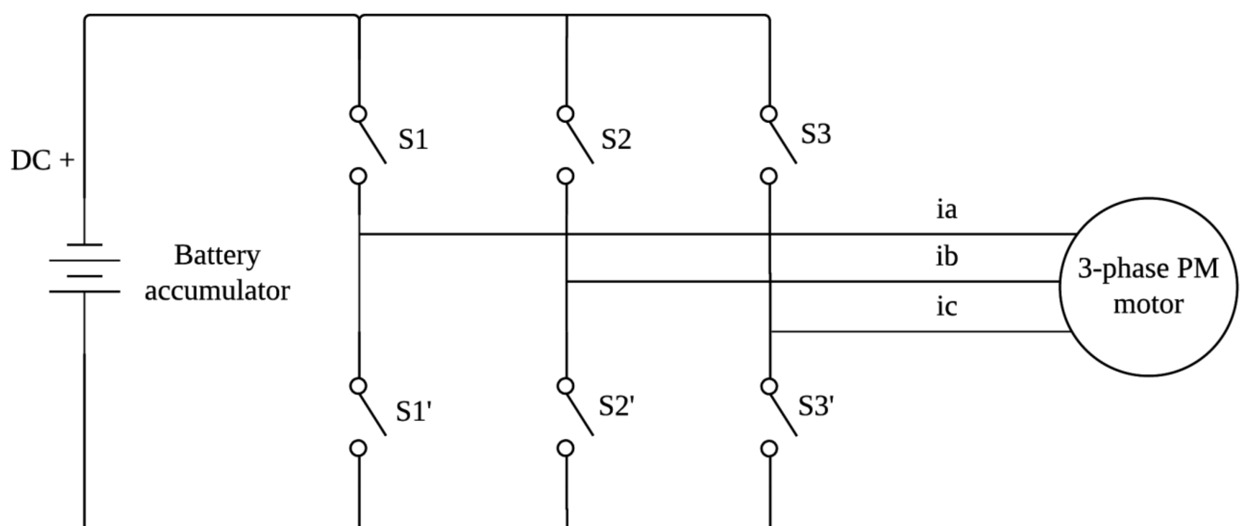

Figure 1.10: 2-level voltage source inverter structure.

By high frequency switching of the transistors in each of the phase legs in figure 1.10, waveforms with modified amplitude and frequency can be modulated. The amplitude determines the motor power, while the frequency is proportional to the wheel speed of the car. If the driver requests more torque, the motor controller needs to control the switching of the transistors in a way that gives the amplitude and frequency according to the required torque.

A 2-level VSI is the most normal structure to use in smaller scale systems. This is because it is compact, lightweight and have relatively good performance. If higher performance is wanted, multilevel VSIs can be an option. In a multilevel VSI, the input DC voltage is divided into multiple voltage levels. Each of these levels are switched on and off by transistors in a way that gives smoother output waveforms. This reduces the losses in the motor, but results in the VSI becoming bigger and heavier. For a 3-level VSI, 12 transistors are needed, instead of only 6 for a 2-level VSI. For the Revolve design, multiple levels was an initial design concept, but early disregarded due to the additional weight and size.

Last season, an off-the-shelf VSI from Eltek was used, the Bamocar d3, [4]. This is a 2-level insulated-gate bipolar transistor (IGBT)-based VSI with water-cooled heatsink. It features different control setting possibilities, with options for torque and speed control. It is compatible with different rotor position sensors and controlled over CAN bus.

Figure 1.11: Bamocar d3 voltage source inverter.

The Bamocar d3 is used by many Formula Student teams and comes recommended by the motor manufacturer. It is a robust and general purpose designed inverter, with an easy-to-use software for tuning and setting up the system functionalities. As can be seen in table 1.3, the characteristics of the Bamocar d3 are suitable for a Formula Student drivetrain with maximum output power of 80 kW. For the 2014 season, the Bamocar d3 was supplied from the 12 V grounded low voltage (GLV) system of the car, but can also be delivered for a 24 V supply system. Last season, it interfaced with a resolver for rotor position measurement from the motor, but can also be delivered to interface with an encoder or even run sensorless. The downsides with this inverter are the high weight, and the low level of flexibility when it comes to the input/output cables and cooling-circuit. As can be seen in figure 1.11, the phase current cables come out at the left side of the box, the nipples for the water-cooling comes out in the front, together with the power supply, CAN and other signal connections. The DC contacts cannot be seen in the figure, but are going in at the right side of the box. Because of this, the motor room and the car has to be designed to fit with the inverter inputs/outputs. By in-house design, it is possible to do this the other way around, designing the VSI, with its inputs and outputs, to fit well into the rest of the motor room assembly. Another disadvantage with using the Bamocar d3, is that it is designed for a general purpose. It lacks some of the required functionalities like discharge and precharge circuits for the internal input capacitance, and it needs to be controlled from an external unit. Last year the ECU did the communication and control of the Bamocar d3. The ideal case would be for the VSI to be more integrated into the vehicle system, and for the ECU to only work as the traction control unit, not the motor controller control unit.

Maximal DC voltage	700 V_{DC}
Continuous output current	200 A_{eff}
Peak output current	400 A_{peak}
Clock frequency	8 – 16 kHz
Power supply	12 V
Weight	8.5 kg
Price	$ 3500

Table 1.3: Bamocar d3 specifications.

2. Motor and Motor Control Theory

2.1. Park and Clark Transformation

The three-phase stator currents shown in figure 1.9 can also be expressed as a rotating vector. Equation 2.1 and figure 2.1 show this vector, rotating in the abc reference frame with the electrical angular speed of the permanent-magnet rotor flux.

$$\vec{i_s}(t) = i_a(t) + i_b(t) \cdot e^{j\frac{2\pi}{3}} + i_c(t) \cdot e^{j\frac{4\pi}{3}} \quad (2.1)$$

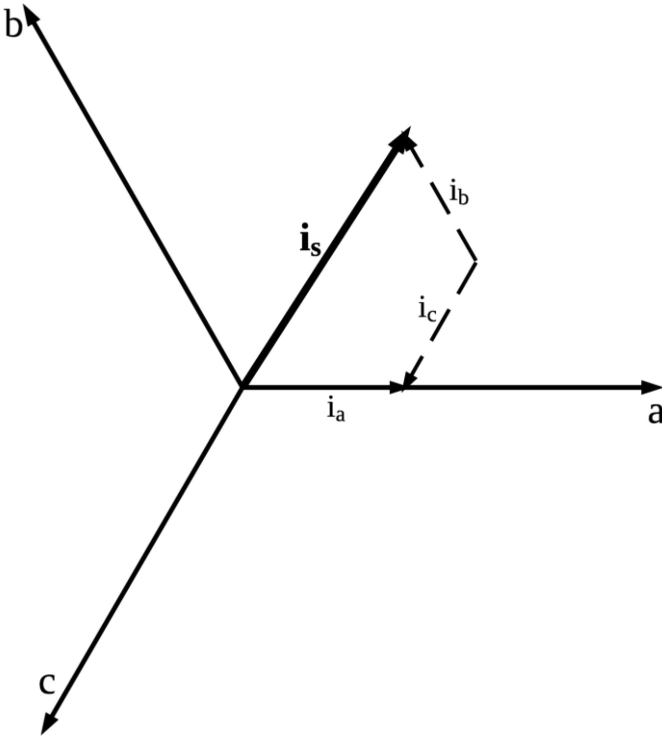

Figure 2.1: Stator current vector in the abc reference frame.

In a control system, it is difficult to work with sinusoidal values like for the stator current vector. The reason for this, is that the references and output values for the applied controllers then also have to be sinusoidal. To avoid this, it is normal to use coordinate system transformations to simplify the current representation. The Clark and Park transformations are commonly used in control systems for three-phase AC motors.

The idea behind the Clark transformation is to represent the three-phase currents in a two-phase coordinate system, with the fixed axes α and β. The stator currents will then become like $i_{s\alpha}$ and $i_{s\beta}$ in equation 2.2, and like equation 2.3 for the inverse case.

$$\begin{bmatrix} i_{s\alpha}(t) \\ i_{s\beta}(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} & 0 \end{bmatrix} \cdot \begin{bmatrix} i_a(t) \\ i_b(t) \\ i_c(t) \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} i_a(t) \\ i_b(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{\sqrt{3}} & \frac{\sqrt{3}}{2} & 0 \end{bmatrix} \cdot \begin{bmatrix} i_{s\alpha}(t) \\ i_{s\beta}(t) \end{bmatrix} \tag{2.3}$$

In the αβ reference frame, the expression for current is still dependent on the electrical angular speed of the permanent-magnet flux. To remove this dependency, the vector needs to be projected into the dq reference frame. The dq reference frame rotates with the same angular speed as the permanent-magnet flux, which makes the current expression independent of the electrical position θ_e. In this transformation, it is normal to align the d axis with the electrical angular position of the permanent-magnet flux. The stator currents i_{sd} and i_{sq} then become like in equation 2.4, and like in equation 2.5 for the inverse case.

$$\begin{bmatrix} i_{sd}(t) \\ i_{sq}(t) \end{bmatrix} = \begin{bmatrix} \cos(\theta_e) & \sin(\theta_e) \\ \cos(\theta_e) & -\sin(\theta_e) \end{bmatrix} \cdot \begin{bmatrix} i_\alpha(t) \\ \beta(t) \end{bmatrix} \tag{2.4}$$

$$\begin{bmatrix} i_\alpha(t) \\ \beta(t) \end{bmatrix} = \begin{bmatrix} \cos(\theta_e) & -\sin(\theta_e) \\ \cos(\theta_e) & \sin(\theta_e) \end{bmatrix} \cdot \begin{bmatrix} i_{sd}(t) \\ i_{sq}(t) \end{bmatrix} \tag{2.5}$$

The relationships between the stator current vector and the current representations in the αβ and dq reference frames are shown in figure 2.2. Because the whole dq reference frame is rotating with the same angular speed as the stator current vector, the dq currents become DC values instead of AC waveforms.

In the dq reference frame, the voltage equations and the torque equation become like in equations 2.6, 2.7 and 2.8.

$$v_{sd} = R_s \cdot i_{sd} + \frac{d}{dt}\varphi_{rd} - \omega_e \cdot \varphi_{\mathrm{rq}} \tag{2.6}$$

$$v_{sq} = R_s \cdot i_{sq} + \frac{d}{dt}\varphi_{\mathrm{rq}} + \omega_e \cdot \varphi_{\mathrm{rd}} \tag{2.7}$$

$$T_e = \frac{3}{2} P \cdot (\varphi_{rd} \cdot i_{sq} - \varphi_{\mathrm{rq}} \cdot i_{sd}) \tag{2.8}$$

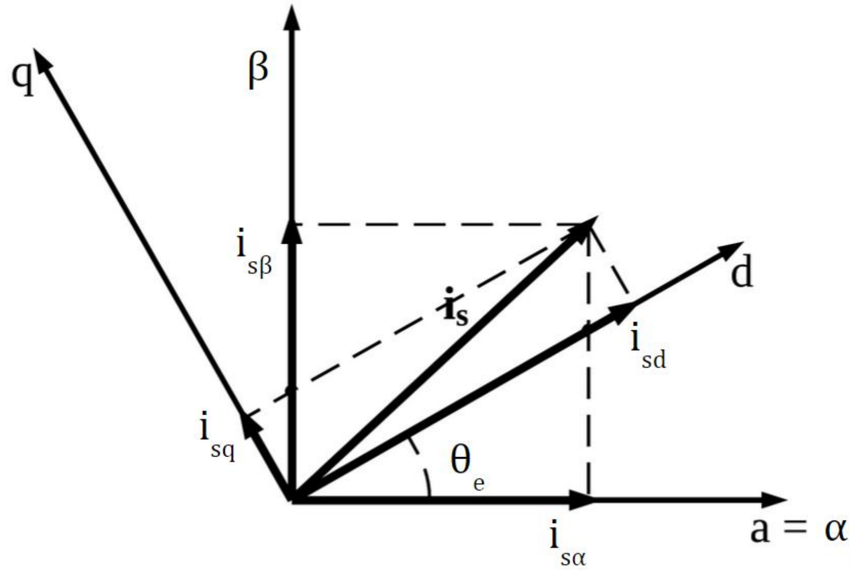

Figure 2.2: Park and Clark transformation.

Where the permanent-magnet rotor flux φ_m is divided into the dq components φ_{rd} and φ_{rq}. To optimize the electromagnetic torque, the appropriate strategy is normally to use a current reference in the d axis direction as zero. From equation 2.9, the stator current is then shifted onto the q axis, making the electromagnetic torque directly proportional to the q current i_{sq}.

$$i_s = \sqrt{i_{sd}^2 + i_{sq}^2} \tag{2.9}$$

$$T_e \propto \varphi_{rd} \cdot i_{sq} \tag{2.10}$$

If a non-zero reference value for the d current is used, the rotor speed can be increased beyond the nominal speed limitations for the motor. This phenomenon is called field weakening, and can be used to increase the speed range of the motor at expense of the applied output torque. A magnetic flux opposing the permanent-magnet flux can be damaging for the permanent-magnets if applied over a longer time period. It may lead to demagnetized permanent-magnets, which again leads to reduced torque production (equation 2.10), and heat generation.

2.2. Pulse Width Modulation

The most common way to control power semiconductor transistors, is by using pulse width modulation (PWM). This is done as shown in figure 2.3, with a triangular waveform compared to a reference signal, giving a duty cycle for the transistors to stay turned on. The carrier

waveform can either be saw tooth or centered triangular shaped. In the Revolve design, centered triangular carrier waveforms is preferred, to easier synchronize the control loop frequency with the PWM. In this case, the phase current measurements can be synchronized with the carrier frequency. By keeping the control loop frequency at twice the carrier frequency, the current measurements can be averaged for improved performance.

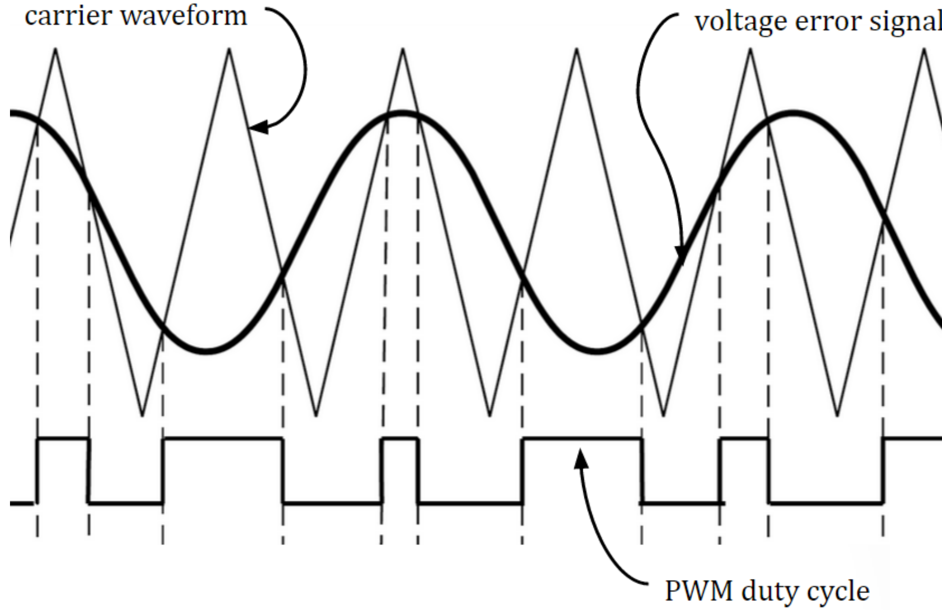

Figure 2.3: Triangular carrier pulse width modulation.

2.3. Space Vector Modulation

Space vector (SV) modulation is an alternative and more advanced modulation technique, where the modulated waveforms are given by applying specific voltage space vectors. By turning on and off (1 or 0) the phase legs S_1, S_1 and S_1 in equation 2.11, the eight voltage space vectors in figure 2.4 can be made.

$$\vec{v_s}(S_1, S_2, S_3) = \frac{\sqrt{3}}{2} \cdot V_{DC} \cdot \left[S_1 + S_2 \cdot e^{j\frac{2\pi}{3}} + S_3 \cdot e^{j\frac{4\pi}{3}}\right] \tag{2.11}$$

These are the available space vectors for a 2-level VSI. For a multilevel VSI, there are more possible combinations, giving the possibility to experiment and design selection algorithms to cancel out harmonic components. As is clear from equations 2.12 and 2.13 given in [6], the maximum line-to-line voltage $V_{LL\,max}^{SV\,PWM}$ for SV PWM utilizes more of the DC bus voltage V_{DC} than the maximum line-to-line voltage for the ordinary PWM $V_{LL\,max}^{PWM}$.

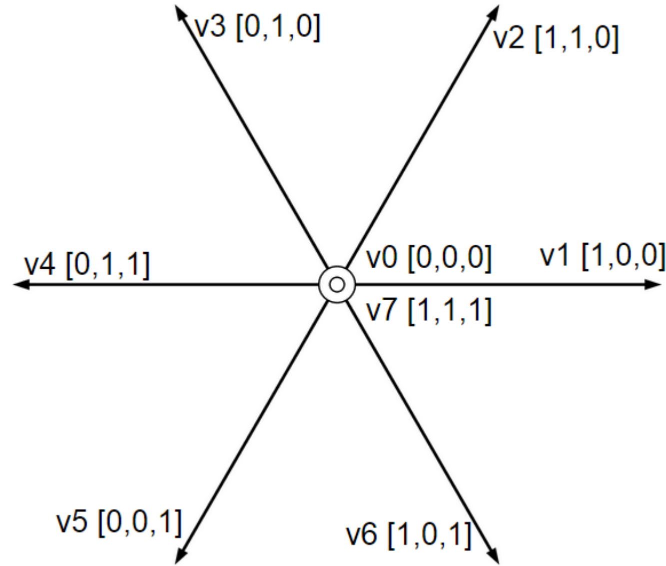

Figure 2.4: Available voltage space vectors.

$$V_{LL\,max}^{SV\,PWM} = \frac{1}{\sqrt{2}} \cdot V_{DC} = 0.707 \cdot V_{DC} \tag{2.12}$$

$$V_{LL\,max}^{PWM} = \frac{\sqrt{3}}{2\sqrt{2}} \cdot V_{DC} = 0.612 \cdot V_{DC} \tag{2.13}$$

By adjoining the different space vectors from figure 2.4 over a time period, an average voltage vector can be made to fit whatever voltage required to supply the motor with appropriate current. There are different tactics for choosing vectors to cancel out harmonics components, minimizing switching losses or to keep the switching frequency constant. As can be understood from this, SV modulation gives higher performance than ordinary PWM. On the other hand, it also brings more complexity into the system. Ordinary PWM was chosen for the Revolve design because of its simplicity and robustness. The implementation of PWM on a microcontroller is much easier than SV modulation, reducing the chance for mistakes and failure.

2.4. Field Oriented Control

To optimize the output torque production for a given stator current i_s, a common strategy is to set the current reference set point in the d direction $i_{sd\,set}$ to 0. From equation 2.9, the current controller will then shift the stator current onto the q axis, giving the relationship in equation

2.10. As explained before, for a PM motor with zero stator current in the d direction, the produced output torque will be directly proportional to the current in the q direction. Because the dq reference frame is rotating with the electrical angular speed of the permanent-magnet flux, the dq currents are DC values. This makes it possible to use simple PI controllers to control the currents and produced output torque. This control scheme is normally called field oriented control (FOC). The FOC system structure proposed in the Revolve design is shown in figure 2.5.

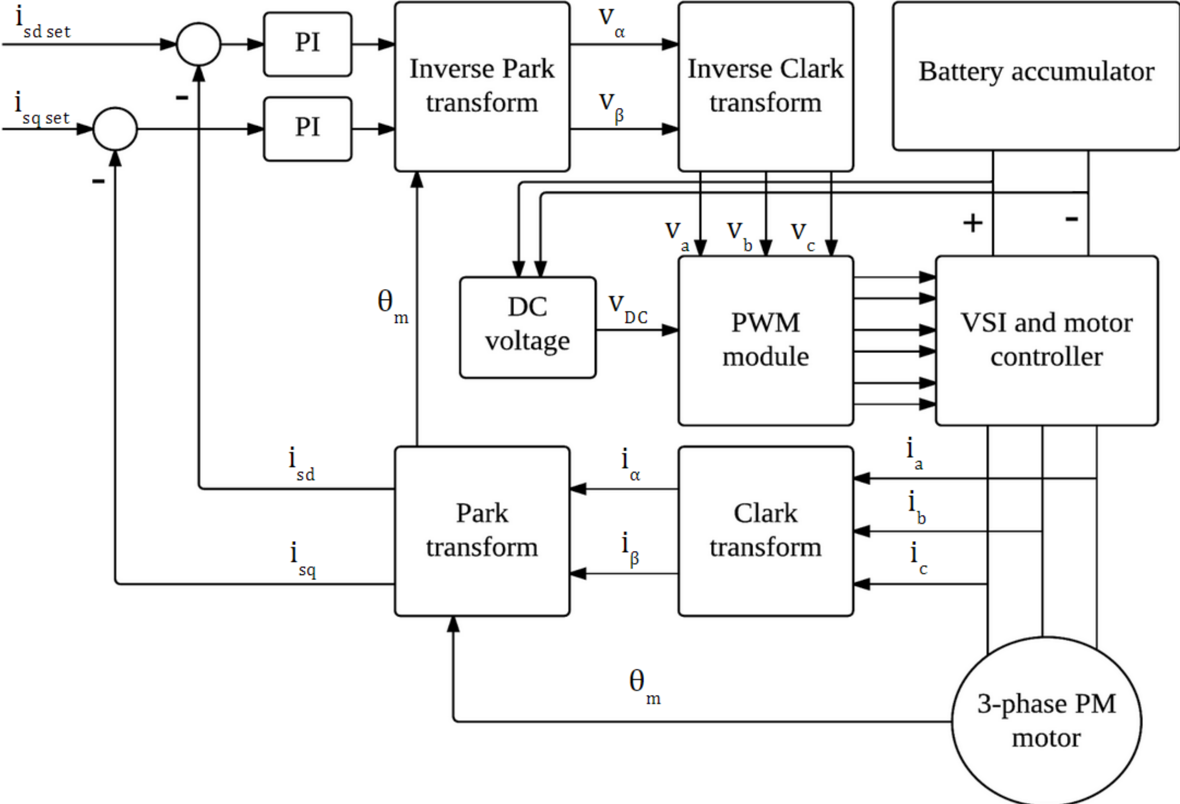

Figure 2.5: Field oriented control schematic structure.

In this system the ab current phases and the rotor position are measured and used to perform Clark and Park transformations into the rotating dq reference frame with equation 2.2 and 2.4. The currents are then compared to the reference set points $i_{sd\ set}$ and $i_{sq\ set}$ and corrected by the means of PI controllers. The output from these controllers are then transformed back into the αβ reference frame by inverse Park transformation (equation 2.5). The output phase voltages of the inverse Clark are fed into a PWM module, giving appropriate duty cycles. The DC bus voltage measurement is used to scale the duty cycles from the PWM module. This is to give appropriate duty cycle values independently of applied voltage level. This is done to give good

performance at both, higher and lower input DC voltage. This is a useful feature when using the system in a laboratory setup.

It is also worth mentioning that the outer control loop for rotor speed is not part of the system. This is because it is more natural for the driver of the vehicle to function as the outer speed loop.

2.5. Direct Torque Control

According to [8], the direct torque control system is the superior control structure for an AC motor drive. The basic idea of direct torque control (DTC), as further discussed in [9], [10] and [11], is to control the stator flux vector by estimation of electromagnetic output torque and stator flux. The angle between the stator flux and the permanent-magnet rotor flux, δ in figure 2.6, is called the torque angle (equation 2.12).

$$\delta = \arctan\left(\frac{L_\beta \cdot i_\beta}{L_\alpha \cdot i_\alpha + \varphi_m}\right) \tag{2.12}$$

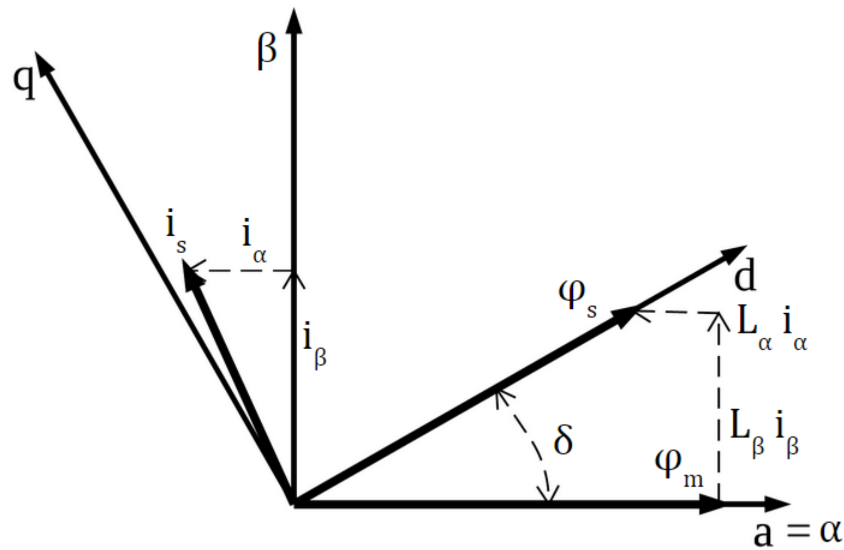

Figure 2.6: Stator and rotor flux linkages.

In steady state, δ will be constant corresponding to the load torque. Then, both the stator flux and the rotor flux will rotate with the electrical angular speed. In transient operation on the other hand, δ will vary, giving the stator and rotor fluxes different rotational speeds. The speed of the stator flux can easily be changed with respect to the rotor flux. This is due to the small electrical

time constants in relation to the mechanical time constants of the system. As given for surface mounted PM motors in [12], the output torque becomes like in equation 2.13.

$$T_e = \frac{3}{2} \cdot P \cdot \varphi_m \cdot i_s \cdot \sin(\delta) \tag{2.13}$$

As can be understood from equation 2.13 and figure 2.7, the torque angle and the stator flux is changed by applying different voltage space vectors. In figure 2.7, the stator flux vector is in sector 1 (S1) of the alpha-beta frame. By applying a voltage vector within one of the four quadrants (Q1, Q2, Q3 or Q4), different effects on the output torque and stator flux will result, as is shown in the figure and listed in table 2.1. The change in electromagnetic output torque is coming from a change in torque angle.

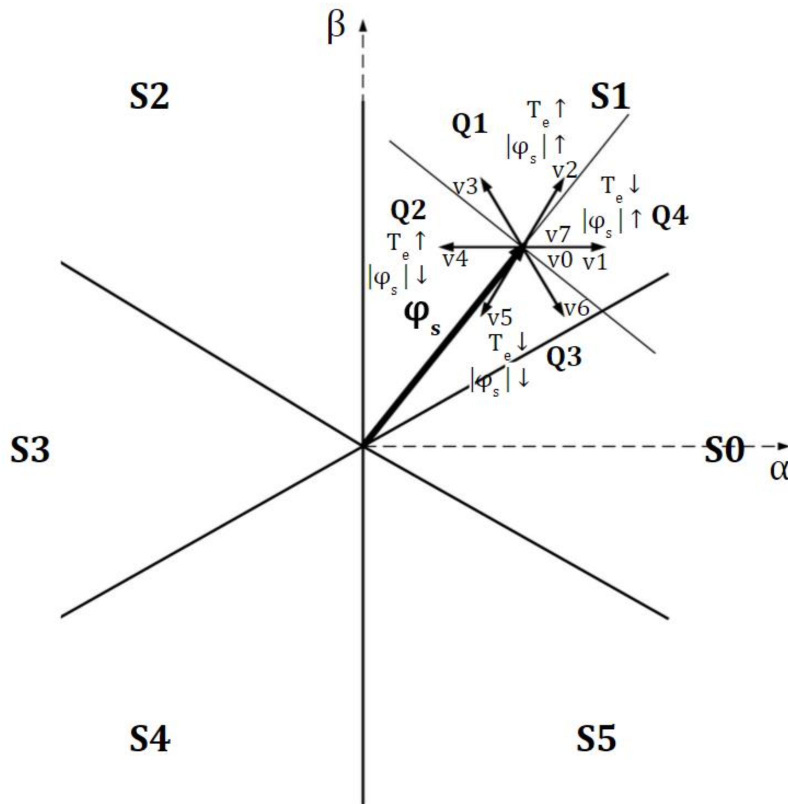

Figure 2.7: Optimal selection of voltage space vectors.

To estimate the stator flux, equation 2.14 and 2.15 are used. Then the electromagnetic output torque can be calculated from the estimated stator flux with equation 2.16. To be able to find and apply the optimal voltage space vector, equation 2.17 is used to get the absolute value and angle of the stator flux vector. The angle is used to locate what sector in figure 2.7 the vector is in, while the amplitude is used for the flux controller.

$$\varphi_\alpha = \int (v_\alpha - R_s \cdot i_\alpha) dt + \varphi_{\alpha 0} \tag{2.14}$$

$$\varphi_\beta = \int \left(v_\beta - R_s \cdot i_\beta\right) dt + \varphi_{\beta 0} \tag{2.15}$$

$$T_e = \frac{3}{2} \cdot P \cdot (\varphi_\alpha \cdot i_\beta - \varphi_\beta \cdot i_\alpha) \tag{2.16}$$

$$\varphi_s = \sqrt{\varphi_\alpha^2 + \varphi_\beta^2} < \arctan(\frac{\varphi_\alpha}{\varphi_\beta}) \tag{2.17}$$

In figure 2.8, a proposed DTC system is proposed. The phase currents are measured and used for flux and torque estimation with equation 2.14, 2.15 and 2.16. Equation 2.17 is used to find the stator flux amplitude and sector for the vector selection table and flux controller.

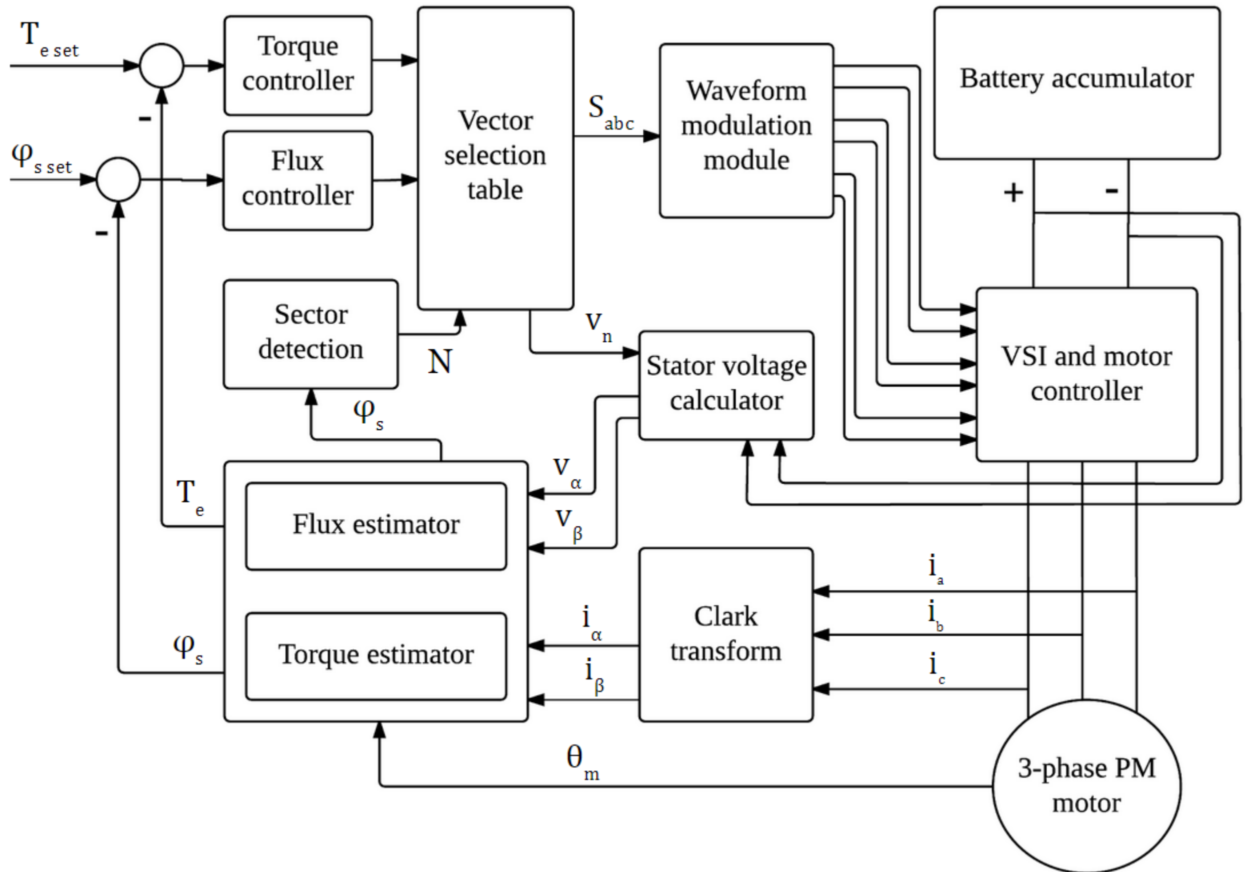

Figure 2.8: Direct torque control schematic structure.

In the torque controller, the estimated torque from equation 2.16 is compared to a torque set point. The vector selection table is basically like table 2.1, giving different voltage vectors based on if the torque and flux should be increased or decreased. The stator voltage calculator block in the figure is needed to get the stator voltages. To do this, it is more convenient to use the DC

bus voltage and calculate the stator voltages, than actually measuring it. The rotor position is also measured in figure 2.8. For the proposed system, this is not needed, as it is not included in any of the applied equations. The angular position can be applied if Park transformation is used in the DTC system.

Stator flux:	**Sector:** / **Output torque:**	S0	S1	S2	S3	S4	S5
$\varphi_s = 1$	$T_e = 1$	v2 [1,1,0]	v3 [0,1,0]	v4 [0,1,1]	v5 [0,0,1]	v6 [1,0,1]	v1 [1,0,0]
	$T_e = -1$	v6 [1,0,1]	v1 [1,0,0]	v2 [1,1,0]	v3 [0,1,0]	v4 [0,1,1]	v5 [0,0,1]
$\varphi_s = -1$	$T_e = 1$	v3 [0,1,0]	v4 [0,1,1]	v5 [0,0,1]	v6 [1,0,1]	v1 [1,0,0]	v2 [1,1,0]
	$T_e = -1$	v5 [0,0,1]	v6 [1,0,1]	v1 [1,0,0]	v2 [1,1,0]	v3 [0,1,0]	v4 [0,1,1]

Table 2.1: Optimal selection of non-zero voltage space vectors.

DTC is more complex than FOC. It is therefore more difficult to implement and requires more processing power. Because of the integrations in the flux estimations, proper filtering is needed to avoid drifting in the results. To keep it simple and avoid unnecessary risks, FOC was chosen for the Revolve design.

3. Sensors and Measurements

3.1. Closed Loop Current Transducer

A closed loop current transducer provides accurate current measurement for AC, DC and complex currents. It gives galvanic isolation, fast response and high accuracy for DC and for high frequency AC measurements. In other words, ideal for measuring the phase currents in a motor control system like FOC or DTC. The principle of operation is sketched in figure 3.1.

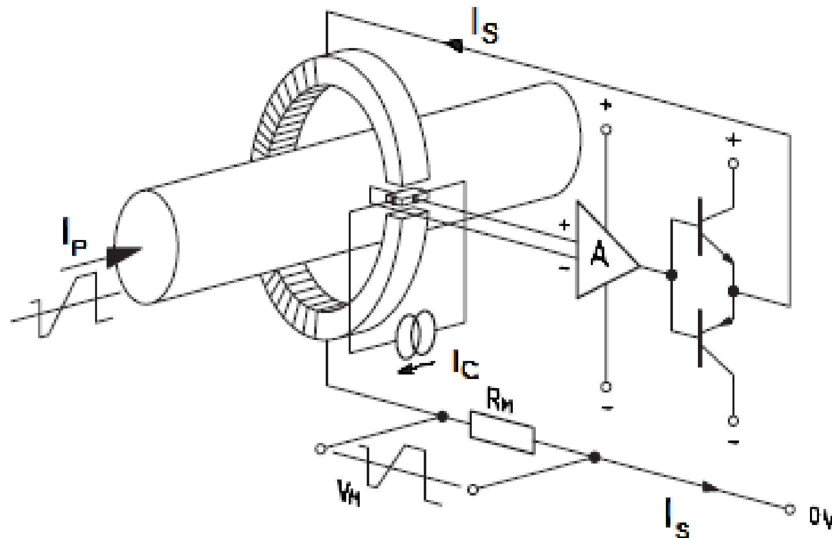

Figure 3.1: Closed loop current transducer functionality.

A closed loop current transducer uses the Hall effect to create a compensating current in the secondary coil, giving a total flux equal to zero. The secondary current I_S, creates a flux opposing and cancelling out the flux from the primary current I_P in figure 3.1. With the flux cancelled out, the magnetic potentials become identical, and the relationship between the secondary and primary currents constant, like in equation 3.1.

$$I_S = I_P \cdot \frac{N_P}{N_S} \tag{3.1}$$

To be able to cancel, or compensate, the primary flux, a Hall effect element is used. The Hall voltage from the Hall effect element is proportional to the product of the control current I_C and the magnetic field from the core. The Hall voltage is used to control the secondary current with the operational amplifier and transistor circuit shown in figure 3.1. The secondary current can

then easily be measured from the voltage V_M, over the measuring resistance R_M. In the datasheet for a current transducer, the relationship N_P/N_S is called the conversion ratio. The maximum measuring range is given by the magnetic properties of the core and the properties of the control circuitry. It can be scaled by changing the measuring resistance, where a lower resistance gives higher measuring range. With a supply voltage of ± 15 V for the control circuitry, the lowest measuring resistance value is given by the maximum allowable secondary current.

In the application of a VSI, the primary current would be the phase currents, while the ± 15 V supply voltage will be given from the control circuit board. Two different closed loop current transducers from LEM was used in the Revolve design. At first the LEM LA 150-P, as shown in figure 3.2. This is a through hole circuit board mounted sensor, with maximum measuring range of ± 212 A. It features high accuracy, good linearity and a wide frequency bandwidth. It is a compact and lightweight device, with a weight of less than 25 g.

Figure 3.2: LEM LA 150-P current sensor.

This was the preferred current sensor device until an incident happened during testing of the inverter. At the most extreme operating conditions, the maximum measuring range of ± 212 A was not enough, and a new current sensor was needed. A more detailed description of the incident and why the sensor was replaced, is given in appendix A.

The new current sensor, the LEM LF 205-S, is also a closed loop current transducer. The main differences from the LA 150-P are that it has a higher current measuring range, it is chassis mounted, and has a larger and heavier design. The LF 205-S has a measuring range of ± 420 A, it weighs 78 g, and is considerably larger. It is not meant to be mounted on a circuit board, but directly to the inverter casing. The additional weight and size was not ideal, but made it possible to drive the inverter at higher peak currents. Luckily, the larger sized device still fit into the inverter casing. The LF 205-S also has the same interface as the LA 150-P, with three pins; one for the measuring signal and two for the ± 15 V supply voltage. Because the sum of the three

phase currents add up to zero, only two sensors are needed for the phase current measurements. On the other hand, by using three sensors, possible offsets in the measurements can be accounted for. This was still not done due to the limited space and because the two-sensor setup gave good performance. The LEM LF 205-S sensor is shown in figure 3.3.

Figure 3.3: LEM LF 205-S current sensor.

3.2. Magnetic Shaft Encoder

An encoder is a sensor that generates a digital signal in response to movement. For rotational movement sensing, like for the rotor in a PM motor, a shaft encoder can be applied. Shaft encoders are available with both optical and magnetic sensing technologies. The magnetic encoders use permanent-magnets and Hall effect elements for sensing, while the optical use light emitting diodes (LEDs) and phototransistors. The magnetic technology is more resistant to thermal and mechanical shock, while the optical is more reliable for industrial applications. For use in automotive applications, it is important with a robust design that can handle vibrations. The magnetic sensing with Hall effect elements was therefore chosen for this design.

An incremental encoder with ABZ interface generates two pulses and one reference impulse, like illustrated in figure 3.4. The A and B pulses are phase shifted by 90 degrees and therefore called quadrature output pulses. The states of the two quadrature pulses can be decoded to counts, where the number of counts per revolution (cpr) gives the resolution of the encoder. To be able to measure position with an incremental ABZ encoder, the Z pulse is needed as

reference. The pulse occurs once per revolution and makes it possible to reference the encoder counts to an absolute value.

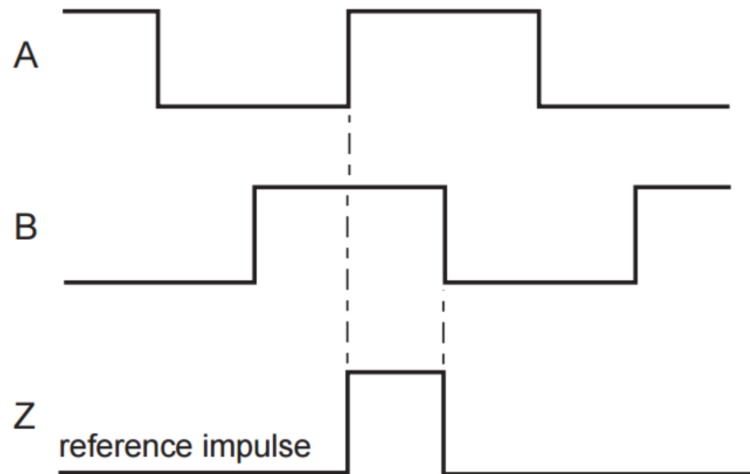

Figure 3.4: Incremental ABZ encoder functionality.

Because the Z pulse is needed for position measurement, the encoder needs to be initialized at startup. Meaning, before the ABZ encoder can be used for measuring absolute position, the Z pulse needs to occur at least once. For motor control systems where the rotor position is needed for Park transformation, this is important to bear in mind. For example, if the motor control system loses GLV power, the encoder needs to be initialized by turning the rotor until the Z pulse occurs. To avoid initialization, there are also multi-turn absolute encoders available on the market. Instead of giving incremental rotational information, an absolute encoder generates a digital representation of the absolute position.

At first, an absolute encoder with binary synchro-serial interface (SSI) was acquired for the Revolve design. SSI is a serial data communication where the transmission is synchronized at the receiving and sending ends by a common clock signal. The data sent from the encoder is binary codes representing sectors of one revolution. For an absolute binary encoder, the resolution is therefore given as the number of bits needed to represent all the sectors of one revolution. The acquired encoder was the RM44SC absolute binary SSI encoder from RLS. The encoder requires power supply of 5 V and features a resolution of 8192 pulses per revolution (ppr) (can be represented by 13 bits). As explained in more detail in appendix A, this encoder has a maximum speed rating of 2500 rpm. Because the PM motor, the Emrax 228, has a maximum rating of 6000 rpm, this encoder was useless for the application. It was therefore decided to acquire a new encoder from the same manufacturer. The RM44IC is an encoder from

RLS with the same magnetic encoder flange and supply power. This is an incremental ABZ encoder with a resolution of 2048 cpr and a maximum speed rating of 10000 rpm. Because it uses the same encoder flange and power supply, it was easily changed. The encoder flange is the part connected to the rotor of the motor and is shown as the bottom part in figure 3.5. As can be seen from the figure, the encoder base unit is attached to the flange with two screws. The control circuit board could use the same power supply, but the data interface had to be changed. By using an internal decoder in the microcontroller on the control circuit board, the ABZ pulses were easily translated into counts. By using this decoder, the need for computational power from the processor was reduced compared to using the binary SSI encoder.

Figure 3.5: Magnetic shaft encoder from RLS

The downside with using the ABZ encoder, is that it has to be initialized every time the GLV system is turned off. The easiest way of doing this, is to push the vehicle until the rotor has rotated a maximum of one revolution. Because the gear exchange ratio of the vehicle is 3.91:1, the driveshaft and wheels only have to be rotated a maximum of ¼ a revolution. The problem with this approach occurs if the vehicle stops during a race and the GLV needs to be reset. A solution to this problem would be to use a startup procedure without rotor position and feedback control until the Z pulse occurs.

3.3. Voltage Measurement

Like for the phase current measurement, a closed loop voltage transducers with Hall effect elements can also be used for measuring the DC bus voltage. LEM offer voltage transducers for similar power supply and output characteristics, as for the current transducers. Because the DC bus voltage is centered around a constant value with smaller deviations than the phase currents, and because the accuracy is less important for the applied control system, a simpler and more compact solution was chosen. As shown in figure 3.6, and in more detain given in the schematic drawing of the circuit in appendix H, a voltage divider circuit is used.

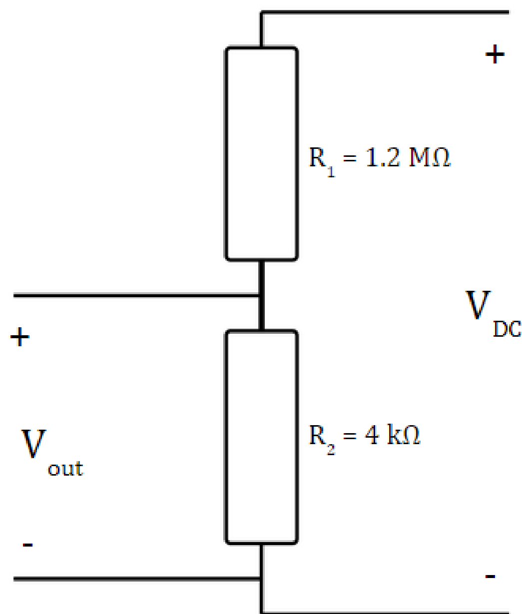

Figure 3.6: DC bus voltage measurement.

The DC bus voltage V_{DC} is measured by dividing it with a constant resistance value, like in figure 3.6 and equation 3.2. By using a large resistance R_1 and a smaller resistance R_2, the measurement voltage V_{out} becomes small enough for the anaolog-to-digital converter (ADC) in the microcontroller to handle.

$$V_{out} = V_{DC} \cdot \frac{R_2}{R_1+R_2} \tag{3.2}$$

To avoid large currents in this circuit, the total resistance should be large. With a total resistance of 1204 kΩ, large currents will not be a problem. This makes it possible to put the measurement circuit on the control circuit board. To make the circuitry compact, a number of small surface mount device (SMD) resistors in series are used, instead of one large 600 V rated resistor.

4. Control System Simulation

To simulate the motor control system, Simulink models were developed. The SimPowerSystems library was used for standard models for PM motor and IGBTs, as seen in figure 4.1.

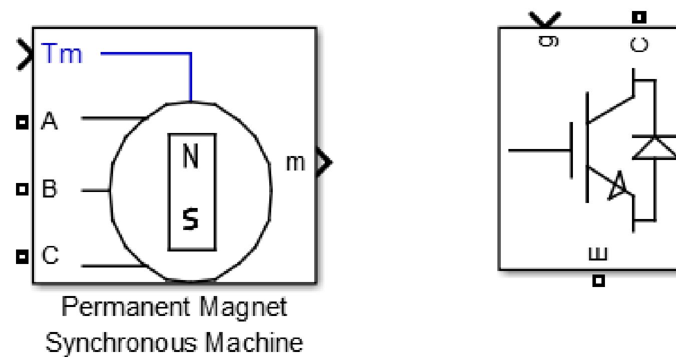

Figure 4.1: PM motor and IGBT Simulink models.

An ideal voltage source model was used to supply the 2-level VSI and PM motor. SimPowerSystem also has a lithium-ion battery cell model available. An attempt to use this was done at first. A battery accumulator model containing a total of 288 lithium-ion cell models were made, as can be seen in appendix E. This model turned out to be too complex to simulate, and was therefore dropped.

It was considered using the C code generator in Simulink to generate the program code for the motor control system directly. This concept was disregarded because the generated code was difficult to read and it was considered easier and more efficient to manually write it. The Simulink models were therefore mostly used to simulate different operating conditions, finding initial tuning parameters and required control loop frequencies.

4.1. Field Oriented Control Simulation

To simulate FOC, the models in appendix F were used. The models were run with a control loop frequency of 16 kHz and PM motor settings equal to the Emrax 228 characteristics. In the FOC PWM model, the standard Simulink models for Clark and Park transformation and PWM

generator were applied. With an outer speed control loop, the result was like in figure 4.2, while it with torque control ended up like in figure 4.3.

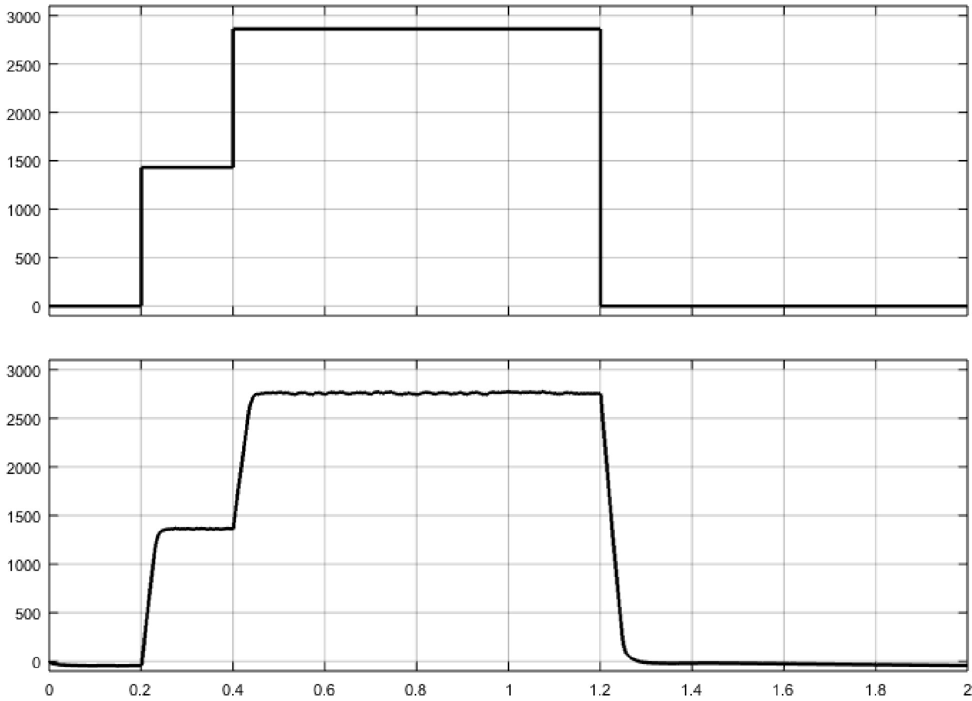

Figure 4.2: Simulink model of FOC with PWM speed control, speed plot.

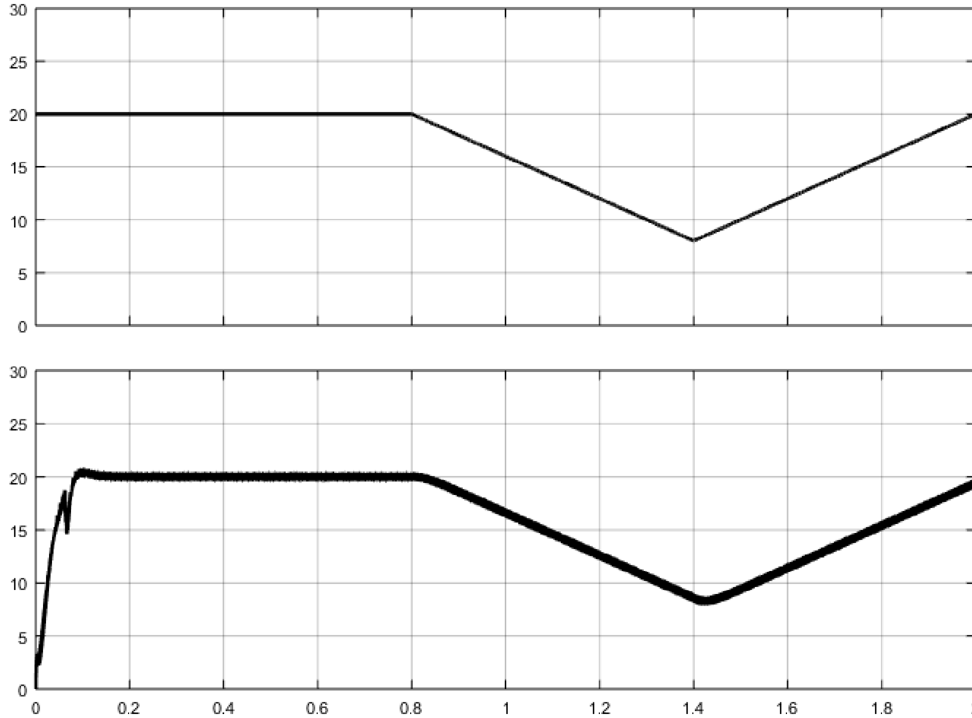

Figure 4.3: Simulink model of FOC with PWM torque control, torque plot.

The results with both, speed control and torque control were good. In figure 4.2, the upper waveform is the speed reference, while the lower one is the actual speed. The same applies for the torque plots, where the upper is the reference torque, and the lower the produced electromagnetic torque. The produced torque has some ripples, but still follows the reference value. These plots were made with a reference torque equal to the load torque applied to the motor, at lightly loaded operation and 530 V DC voltage supply from an ideal voltage source.

In the next model, hysteresis was used instead of PWM. In figure 4.4, the load torque is plotted against the produced torque. The hysteresis effect can clearly be seen in the plot. The way the hysteresis in this model works, is that the error from the PI controllers were compared to a hysteresis limit, giving the direct control of the IGBTs. The model could have been improved further, but the concept was early disregarded, and put aside.

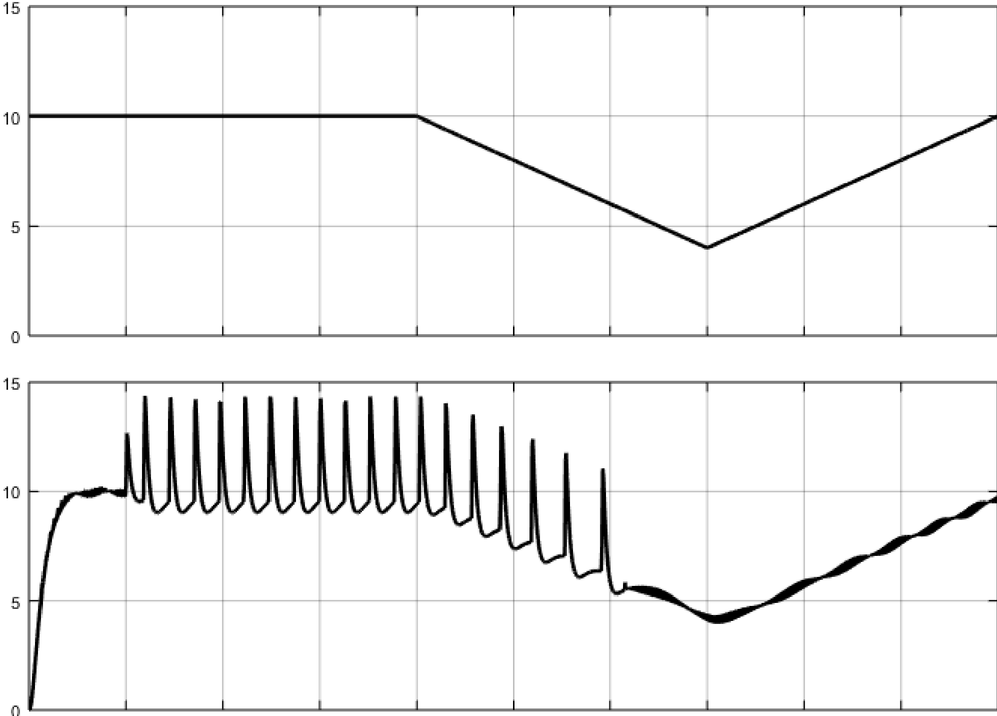

Figure 4.4: Simulink model of FOC with hysteresis speed control, torque plot.

In the following model, SV PWM was used. Just as with the hysteresis model, this model was not prioritized. The reason for this, was that it was decided to go for ordinary PWM in the actual VSI design. SV PWM could have given better performance, as discussed in chapter 2, but it is also more complex and needs more processing power. In figure 4.5, a speed plot with speed

control is shown. Like in the other plots, the upper is the reference, while the lower is the actual speed. The result could have been improved by tuning parameters more carefully.

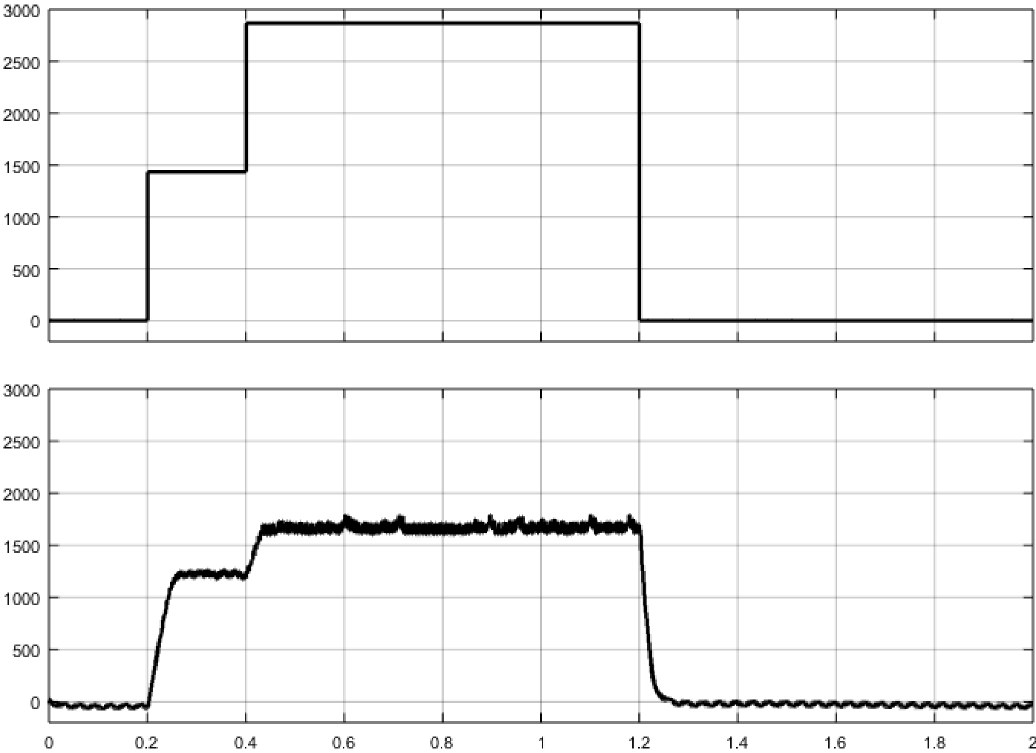

Figure 4.5: Simulink model of FOC with SV PWM speed control, speed plot.

While testing the actual inverter, the FOC Simulink models were used to explore the effects of using different control loop and switching frequencies. In the actual inverter, the control loop frequency ended up at the maximum frequency the microcontroller could handle, leaving the microcontroller as the limiting factor for system control speed.

4.2. Direct Torque Control Simulation

A successful Simulink model for DTC was never developed. The last model in appendix F is actually running on FOC, while the estimations for DTC are done in parallel. This was done to verify estimations, as a start of the development of the DTC model. In the model, the estimations from chapter 2 are done. The stator flux is estimated from the stator currents and voltages, while the torque is estimated from the stator flux and currents. In figure 4.6, the load torque, estimated torque and actual torque are plotted. The actual torque follows the load torque well, while the estimated torque curve is a little misshaped. The reason why the actual torque follows the load

torque so well, is because the control system is running FOC. In figure 4.7, one of the problems with DTC is shown. The flux estimation tends to drift because of the integration done in the equations. In this model, an attempt to filter and correct the flux was made. This improved the results, but more advanced filtering should be applied to solve the problem.

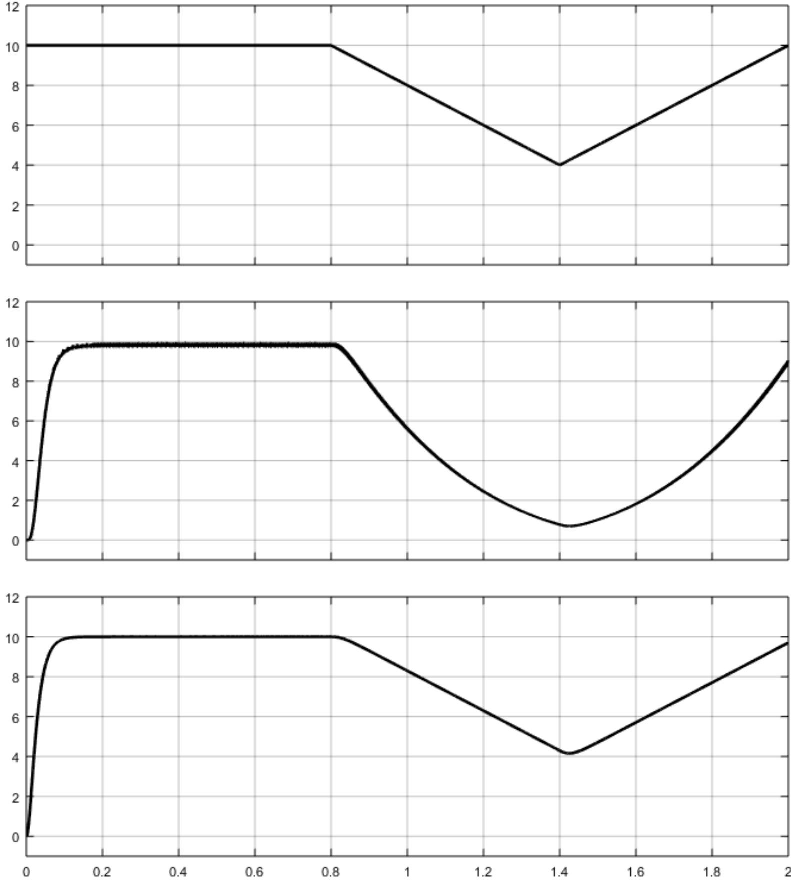

Figure 4.6: Simulink model with torque estimation.

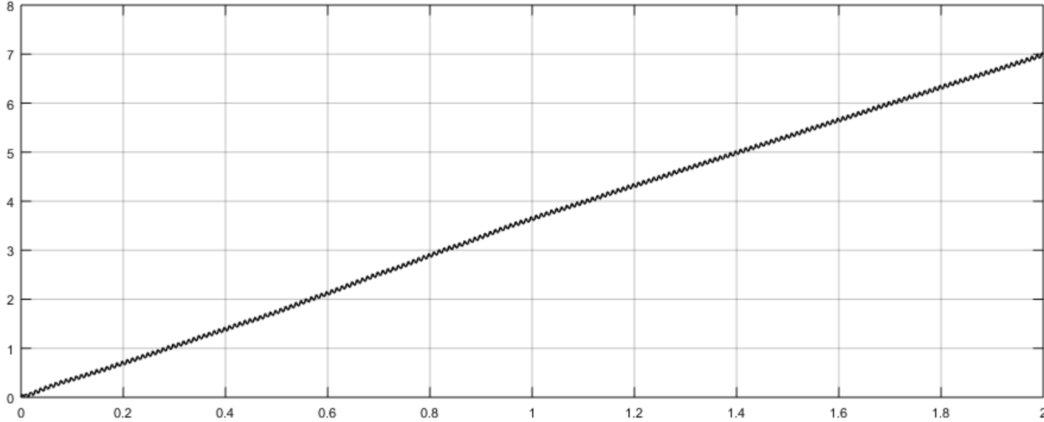

Figure 4.7: Simulink model of stator flux drift.

5. Control System Hardware

5.1. Circuit Board Design

To process the measurement data and execute the FOC algorithm, controlling the IGBTs, a control circuit board was designed. As sketched in figure 5.1, the board is supplied with power from the GLV system and is connected to both the CAN busses in the vehicle. A microcontroller receives measurement data from the encoder, the current sensor circuits, the DC bus voltage measurement circuit, and temperature measurements from the heatsink and PM motor.

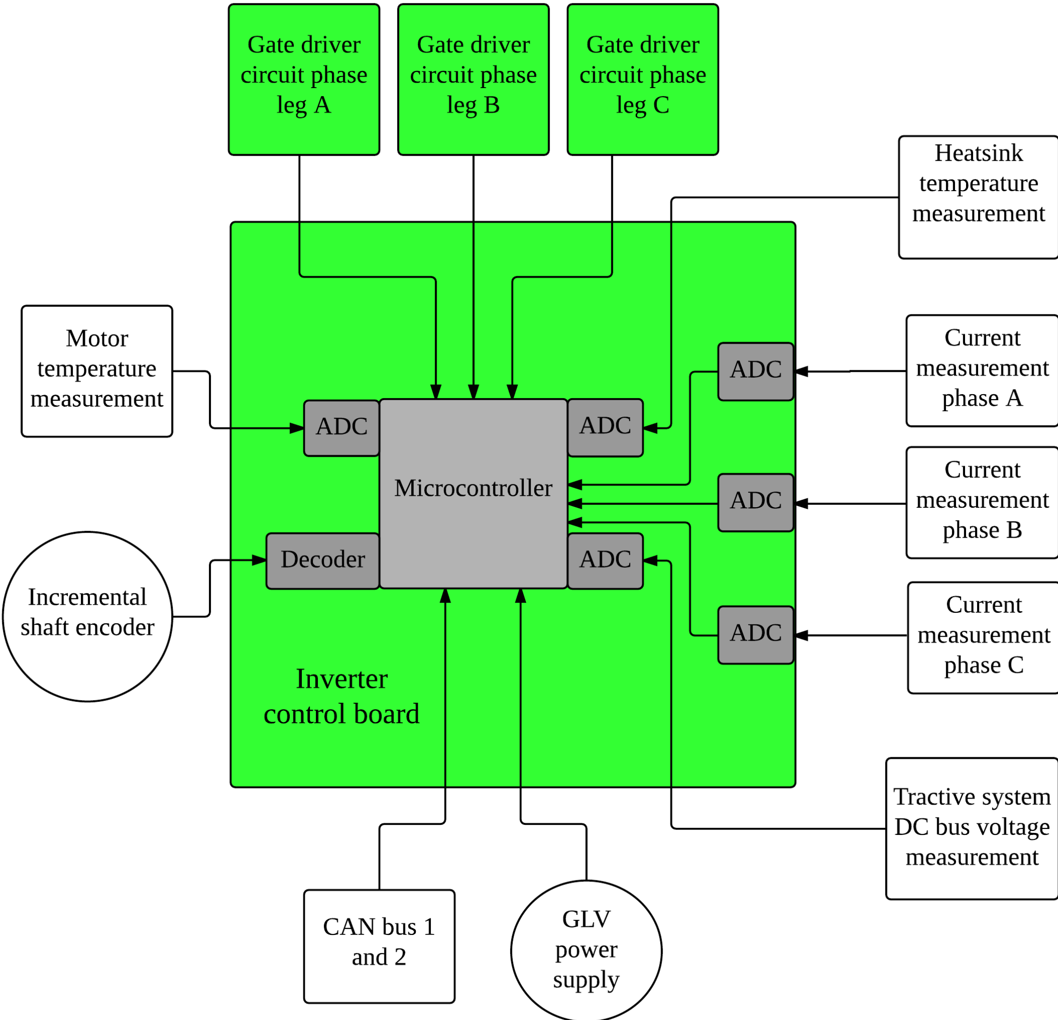

Figure 5.1: Inverter control circuit board schematic drawing.

The control circuit board was designed using the design tool Altium, with the version control program Vault. In figure 5.2, a rendering of the final circuit board is shown, while all the schematic drawings of the circuits are given in appendix H.

In the beginning of the design period, a component and schematic library in Vault was created. By using the library, and always uploading and updating the work to the server, it was made possible to reuse and modify circuit designs for other circuit boards in the vehicle. Most of the power supply circuits, the microcontroller circuit, the CAN transceiver circuit and temperature measurement circuits are examples of standardized designs.

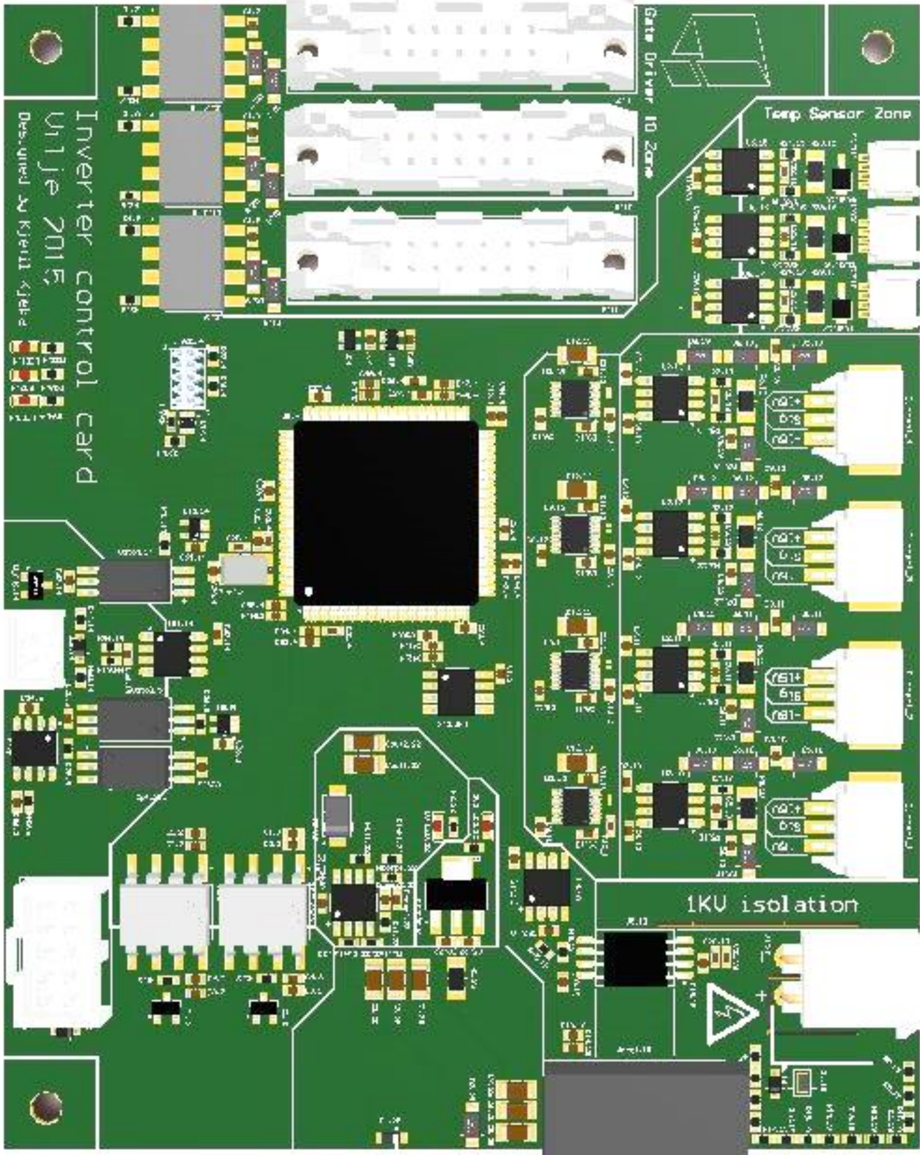

Figure 5.2: Inverter control circuit board rendering.

As mentioned, the board is supplied with 26 V from the GLV. Internally on the board, the different circuits need different power supplies. The microcontroller requires 3.3 V, the gate drivers 15 V, the current sensors ± 15 V, the encoder and DC bus voltage measurement circuits 5 V, and so on. The board has therefore been given internal power supplies for 15 V, 5 V and 3.3 V. An overview of the power supply circuits can be seen in appendix figure H.2, while the

GLV input filter, 3.3 V, 5 V, 15 V and negative 15 V supplies are given in the appendix figures H.3-H.7. Apart from the negative 15 V supply, these circuits are used on several other circuit boards in the vehicle. This also goes for the CAN transceiver and microcontroller circuit in the appendix figures H.8 and H.9. The schematic drawings that have been exclusively made for the inverter control circuit board, are the gate driver interface, encoder, current sensor, and voltage measurement circuits given in appendix figure H.10-H.14. The temperature measurement circuits and the FRAM (ferroelectric random-access memory) in appendix figure H.15-H.17 are circuits partly standardized for the rest of the circuit boards. The schematic drawing in appendix figure H.1 is an overview of the circuit board, in more detail than figure 5.1.

The schematic drawing in appendix figure H.18 is not part of the inverter control board. That is the schematic for the discharge circuit board. This is another circuit board in the VSI casing. Its functionality is discussed in more detail in chapter 7.

5.2. Data and Computational Processing Unit

To process the measurement data and perform the required calculations and operations for the motor control, a microcontroller (MCU) is used. In the initial design phase, it was considered to use a system-on-chip (SoC) solution. This is an integrated circuit, normally containing digital, analog and mixed-signal functions on a single chip. A SoC with a powerful MCU or digital signal processor (DSP) and a field programmable gate array (FPGA) was considered as a viable design concept. The FPGA contains programmable logic blocks, allowing fast data processing. For a motor control application, the measurement data could then be handled in parallel, making the control system very fast and accurate. Programming of FPGA is a complex process, more difficult and time consuming than only handling a MCU. The SoC concept was therefore set aside in favor for a MCU-based design.

To make the circuit board designs in the vehicle as standardized as possible, it was decided to only use two different MCU chips. These are MCUs of the Atmel-MEGA and Atmel-SAM series. The applied Atmel-MEGA MCU is an 8-bit microcontroller, while the applied Atmel-SAM is a 32-bit, more sophisticated MCU. Because of this, the SAM4E16E MCU ended up as the choice for the inverter control circuit. This is an ARM Cortex-M4 processor-based MCU with floating point unit and high data bandwidth. The chip has 144-pins and can be operated with a maximum frequency up to 120 MHz. It also features desirable functionalities like CAN

interface, serial peripheral interface (SPI), quadrature decoder, ADCs and PWM modules. The CAN interface is important for the communication with the rest of the vehicle. The SPI bus for receiving data from the external ADCs in the current sensor circuits. The quadrature decoder is required to decode the ABZ encoder pulses. The PWM modules to operate of the insulated-gate bipolar transistors (IGBTs). While the internal ADCs are used to convert the analog voltage and temperature measurement signals to digital representation.

5.3. Circuit Board Production

The control circuit board was designed in-house by the team, but produced and assembled at the workshop of a professional industry partner. The printed circuit boards (PCBs) were produced according to files generated from the control circuit board Altium design model, together with a stencil showing the component pads. During assembly, the stencil was used to apply solder paste to all the pads where components were to be placed. Then, component placement was then done by hand, as shown in figure 5.3. With all the components in the right place, the PCB was sent through a reflow oven.

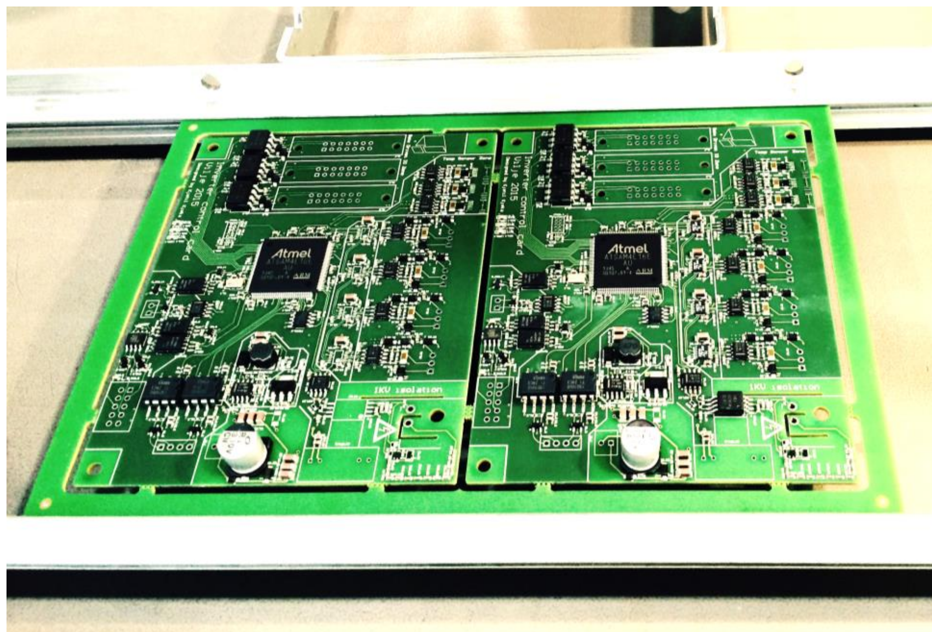

Figure 5.3: Control circuit board production.

With the components on both sides of the PCB soldered, the board was inspected with an x-ray machine to control the quality of the solders. In figure 5.4, an x-ray image of the MCU and some of the closest components is shown. In the image, the 144 pins and their paths into the

integrated circuit in the middle of the MCU are clearly seen. The close by components are small SMD capacitors connected to the MCU pins, while the component at the upper right side, is an external oscillator used for synchronization of the CAN busses for the MCUs in the vehicle network. The components overlapping with the MCU, in the middle, are in reality on the other side of the PCB.

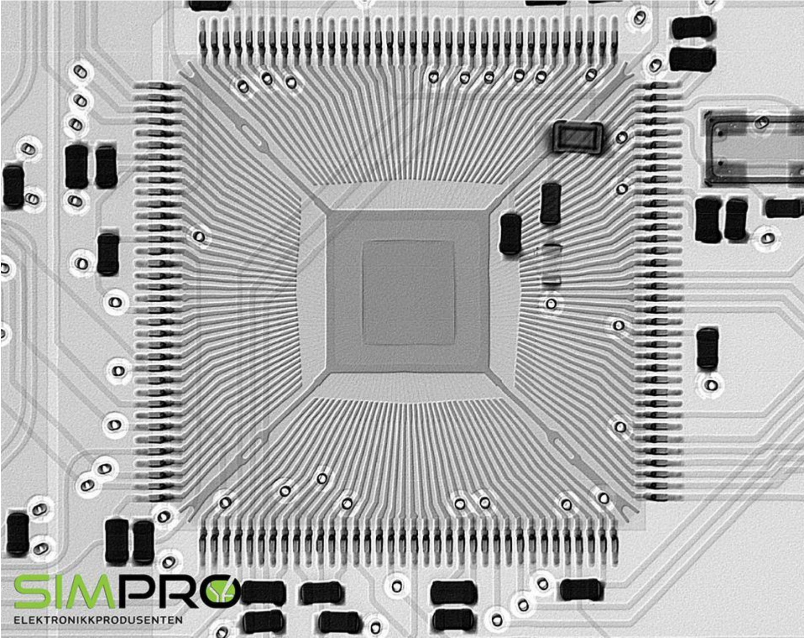

Figure 5.4: X-ray image of the microcontroller on the control circuit board.

6. Control System Software

6.1. Motor Control Program Code

The MCU program code is written in C, using the integrated development platform (IDP) Atmel Studio 6 and the version control system GitHub. Atmel Studio 6 provides a platform for writing, building and debugging C code for the MCU. For programming and debugging it, the Atmel-ICE debugger was used. This is the white box connected to the circuit board in figure 6.1.

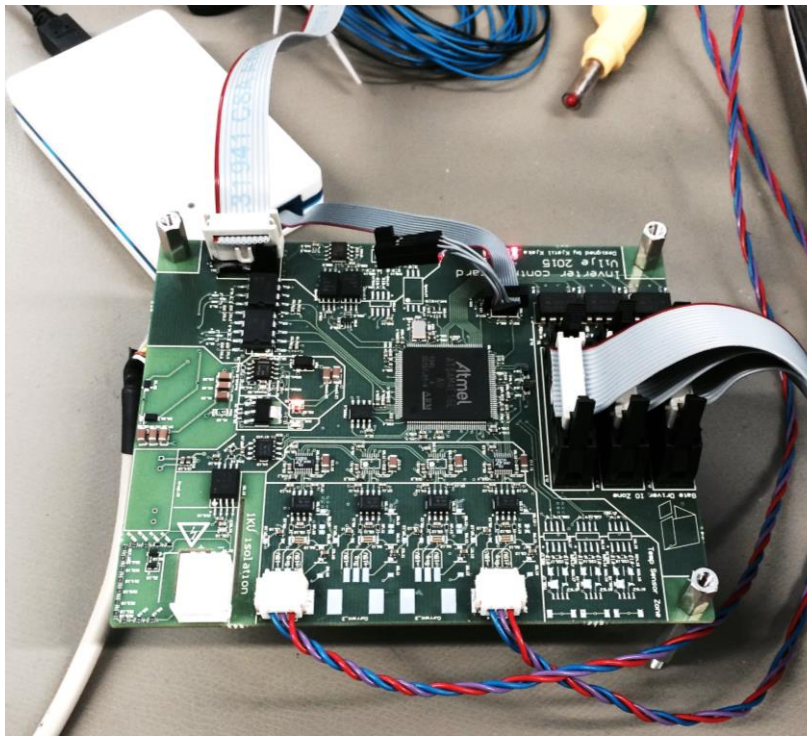

Figure 6.1: Control circuit board programming.

In figure 6.2, the inverter control program structure is shown. Before the system starts running, the first block in figure 6.2, initializes the different modules of the system. In this block, the initialization of flash, floating point unit (FPU), power management controller (PMC), light emitting diodes (LEDs), CAN, PWM and analog front end controller (AFEC) is done. Together with the initialization of the current data conversion and reading, encoder data reading, error status reading and sending, and the field oriented control calculations. After initialization is completed, an infinite while loop, containing the inverter control program, starts running.

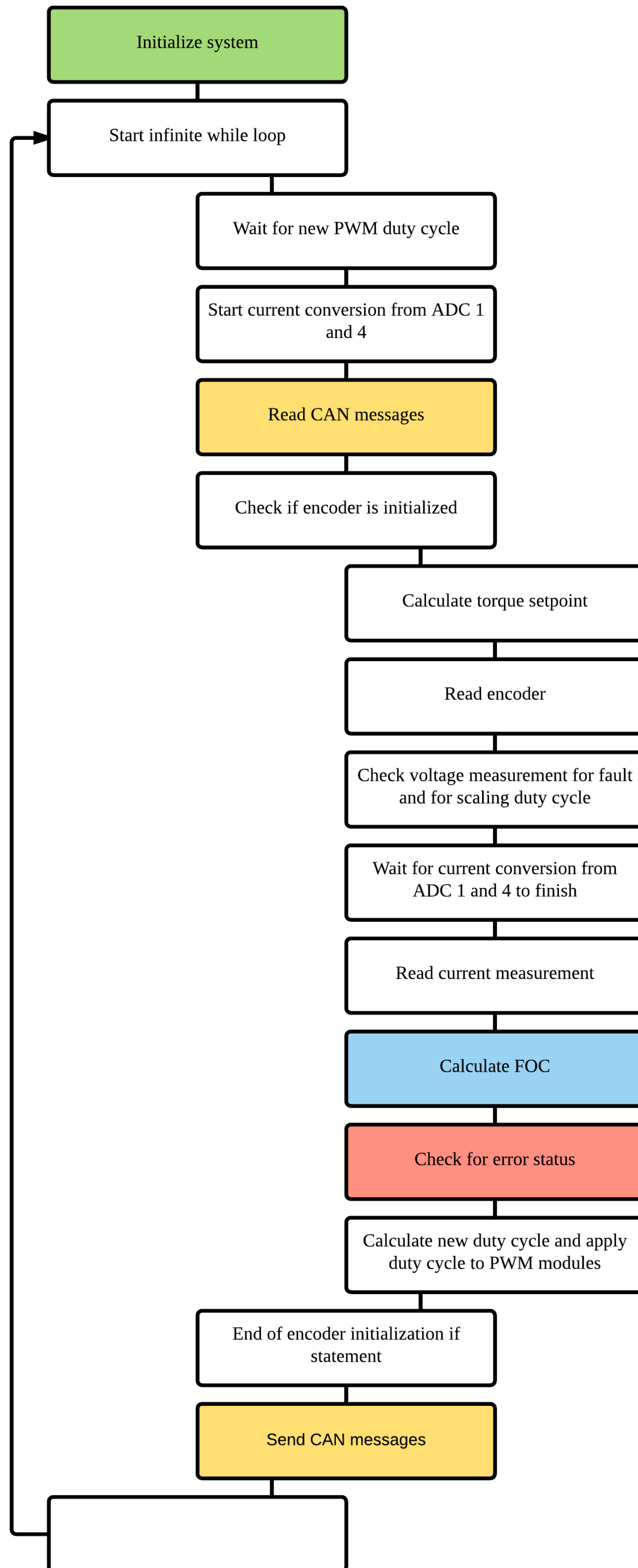

Figure 6.2: Inverter control program.

The first stop in the program loop, is to wait for new PWM duty cycle to be calculated. This is done to synchronize the control loop with the PWM duty cycle. Then, the current conversion from ADC 1 and ADC 4 is started. In figure 6.1, the two contacts with red-purple-blue wires are the applied current sensor circuits. Current sensor circuit 1 (ADC 1) is the left circuit, while current sensor circuit 4 (ADC 4) is right one. After this, the MCU reads the CAN messages relevant for the system. To continue at this point, the encoder initialization needs to be complete. If not, the program will never enter the FOC system part.

If the encoder is initialized (at least one Z pulse have occurred, as explained in chapter 3), the torque set point will be calculated. This is done by multiplying the torque request from the ECU with a motor constant. Then, the encoder and the DC bus voltage measurements will be read in the following two blocks. After this, the program will stop and wait until the current conversion is finished, before the current measurements are read. In the blue and red boxes in figure 6.2, the FOC algorithm is executed and the error status checked. After this, the new duty cycles are calculated and applied to the PWM modules commanding the switching of the IGBTs. Before starting the whole procedure again, the MCU sends its CAN messages.

Polling is used instead of interrupts in the program flow. At first a more interrupt-based program structure was used. In this system, the interrupts with highest priority cut in, when ready to execute. The interrupt-based program showed good performance, but polling was still considered better. This control structure was executed faster, allowing faster control loop speed. The system is also easier to follow and to modify without disturbing important functionalities.

The program code has been divided into folders and files to make it easier to follow. All the code is given in appendix B, where the specific folders and files have been divided into subchapters.

6.2. Field Oriented Control Code

As explained before, the initialization of the FOC system is done before the program loop starts running. In the initialization sequence, the control parameters for the PI controllers are set. In the blue box in figure 6.2, the FOC calculation block, is where all the operations involved in the control system are executed. First, the Clark and Park transformations are performed using the phase currents i_a and i_b, and the electrical position θ_e. Then, the error between the setpoint and actual dq currents are calculated, before the PI controllers are run. After this, reverse Park

and Clark transformations are performed, giving the voltages for the duty cycle calculations. To optimize these calculations for speed, the CMSIS DSP software library is used. This is a library of common signal processing functions optimized for the Cortex-M processor. These applied functions with inputs/outputs are shown in table 6.1.

	Input	Output
arm_sin_cos_f32	Electrical angular position	Equivalent sine and cosine values
arm_pid_f32	Pointers to control parameters	No output
arm_clarke_f32	ab phase currents	αβ axis currents
arm_inv_clarke_f32	αβ axis currents	ab phase currents
arm_park_f32	αβ axis currents	dq axis currents
arm_inv_park_f32	dq axis currents	αβ axis currents

Table 6.1: CMSIS-DSP software library functions.

The functions in table 6.1 require floating point inputs and generate floating point outputs. Because the Atmel SAM4E16E MCU have a FPU and because the CMSIS DSP library provide the needed functions for floating point representation, the use of fixed point integers has been avoided in the whole program.

6.3. CAN Messages and Error Status

In appendix B.8, the header file containing the identification and priority numbering for all the CAN messages, on both the CAN busses in the vehicle, are shown. In appendix C, lists of input and output data with specific priority are listed. All the receiving data in these lists are on the CAN bus, but some of the sending data had to be dropped. The reason for this was that the CAN busses got overloaded. The most significant receiving data are the dq current set points. The d current set point is set to zero (as long as field weakening is not desired), while the q current is given from the ECU. The most significant sending data are the rotor speed, DC bus voltage, dq currents, the fault vector, and the encoder initialization status. The dq currents are important

parameters for tuning the PI controllers. These values can be sent from the car via a telemetry system, and be represented in the in-house developed analysis software, Revolve Analyze. Revolve Analyze provides two-way telemetry and feature a function for sending back new tuning values for the PI controllers. The analysis software also provides information about the errors and status information for the VSI. This information is given as status bits in the header file for error codes in appendix B, and shown in table 6.2. The status bit 24 starts the VSI, while bit 25 allows the VSI to receive torque request from Revolve Analyze instead of the ECU. This was a convenient function during laboratory testing, but a dangerous feature when testing in the car. The rest of the bits are self-explanatory.

Bit:	Function:		
0	Permanent current limit exceeded	16	Encoder not initialized
1	Permanent voltage limit exceeded	17	
2		18	PID saturation phase A
3		19	PID saturation phase B
4		20	PID saturation phase C
5		21	Gate driver error phase A
6		22	Gate driver error phase B
7		23	Gate driver error phase C
8	Current limit phase A exceeded	24	Ready to drive
9	Current limit phase B exceeded	25	Accept torque request from telemetry
10	Current limit phase C exceeded	26	
11	Upper voltage limit exceeded	27	
12	Speed limit exceeded	28	
13	Gate driver error	29	
14	Lower voltage limit exceeded	30	
15	Lower torque request limit exceeded	31	

Table 6.2: Status bits in Revolve Analyze.

7. Voltage Source Inverter Design

7.1. Power Semiconductor Transistors

The most important components in a voltage source inverter are the power semiconductor transistors, switching the voltage source on and off. There are many different solutions available on the market. The most suitable solution for this design, is power semiconductor modules, either single modules, half-bridge modules or a full six-pack module. The reason for considering modules more suitable than discrete transistors, is because a semiconductor module is easier to cool and because the inductances are symmetrically distributed between the internal components. The reason why a module is easier to cool, is due to the copper base plate it is mounted to. This base plate can easily be mounted to a water-cooled or air-cooled heatsink. A half-bridge module is a module containing two IGBTs and two antiparallel diodes (as shown in figure 7.1). In other words, one phase of the VSI. For a three-phase VSI, three half-bridge modules are needed. This solution becomes more compact than using single modules, where a total of six modules are needed. On the other hand, it is less compact than using one six-pack module. The reason why a six-pack module is not used, is due to the required ratings for the system. According to Semikron [13], a voltage rating of 1200 V and a current rating of 300 A is suitable for this design.

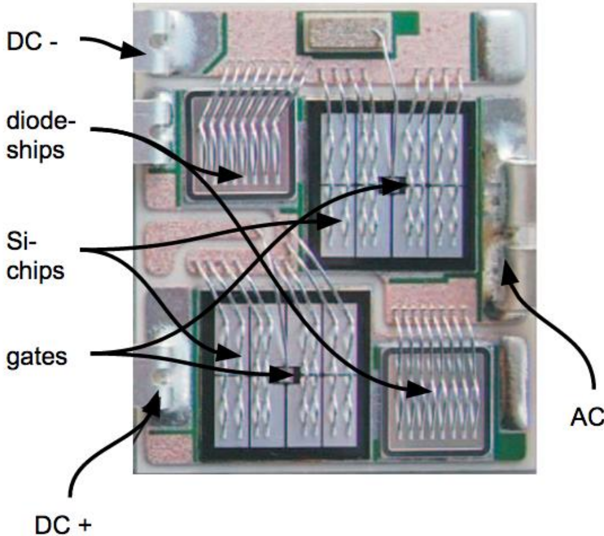

Figure 7.1: Inside of a power semiconductor module [13].

In figure 7.2 and 7.3, two alternative half-bridge module packages are shown. The Semitrans 3 package is used by many manufacturers and available for all types, ratings and generations of transistors. The SEMiX package is a newer design, not available for a selection like the Semitrans 3, but still widely used in industry. The main difference between the two, are where the connections are located. On the Semitrans 3 module, the three terminals on the top are for DC +/- and AC, while the four contacts on the left side are for controlling the IGBTs.

Figure 7.2: Semitrans 3 power semiconductor module.

On the SEMiX module, the DC +/- and AC terminals are at each of the sides of the module, while the contacts for controlling the IGBTs are at the top of the module. This way the control circuitry can be directly mounted on top of the module.

Figure 7.3: SEMiX power semiconductor module.

The Semitrans 3 module package was chosen for this design. The reason for this was because of the simple design and because of its large range of available alternatives. It is for example available with silicon carbide (SiC) metal-oxide-semiconductor field-effect transistors (MOSFETs) instead of IGBTs. In table 7.1, both the SKM300GB12V and the CAS300M12BM2 has Semitrans 3 module packages. The difference between the modules is that the SKM300GB12V is an IGBT half-bridge module from Semikron, while the CAS300M12BM2 is a SiC MOSFET half-bridge module from Cree. As is clear from the table, the Semikron half-bridge modules, SKM300GB12V and SEMiX303GB12Vs, have similar ratings, while the SiC MOSFET module from Cree is quite different.

	SKM300GB12V (IGBT)	SEMiX303GB12Vs (IGBT)	CAS300M12BM2 (MOSFET)
Maximum collector current @ T_c = 80 °C	319 A	342 A	300 A
On-state resistance @ T_j = 150 °C, 300 A	4.57 mΩ	4.4 mΩ	8.6 mΩ
Switching energy (E_{on} + E_{off}) @ T_j = 150 °C, 300 A	23 mJ + 33 mJ	27 mJ + 36 mJ	6 mJ + 6 mJ
Total gate charge @ 300 A	3310 nC	3300 nC	1025 nC
Unit price	$ 174	$ 146	$ 561

Table 7.1: Comparison of power semiconductor module characteristics.

Both IGBTs and MOSFETs are voltage controlled transistors. An IGBT is turned on by applying a positive voltage between the gate and emitter in figure 7.4. By doing this, the collector current i_C is allowed to flow from collector to emitter. A positive gate-emitter voltage induces positive charge on the gate conductor, attracting negative charge to the other side of the gate oxide. This opens a channel through the P-type and N-type silicon layers where the collector current can flow. The P-type silicon in figure 7.4 is silicon doped with boron and has electrons as majority carrier, while the N-type silicon is doped with phosphorous and have holes as majority carrier.

A MOSFET is controlled in the same way. The difference is that it lacks the P^+ collector silicon layer as is in the IGBT. For an IGBT during turn-on, the collector current causes hole injection

from the P^+ collector region into the N^- region, attracting electrons from the N^+ emitter region. This is called double injection and leads to a higher charge density, increasing the conductivity and reducing the on-state resistance. Double injection is the main difference between an IGBT and a MOSFET, and is responsible for the differences in table 7.1.

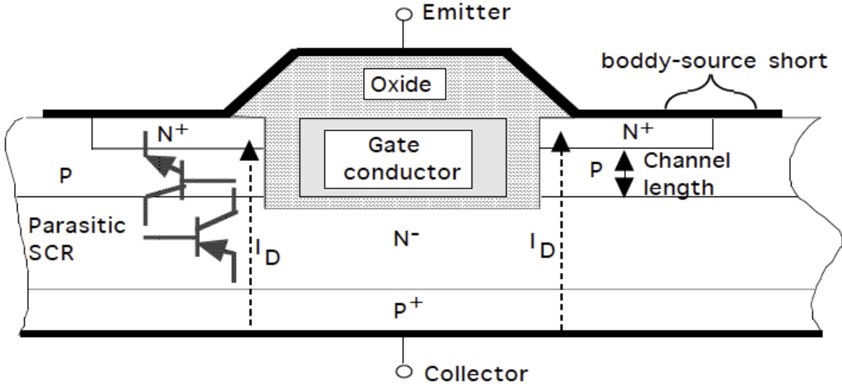

Figure 7.4: IGBT structure with silicon layers [5].

Because of double injection, an IGBT can be manufactured with more silicon and still have low ohmic losses due to the high conductivity of the silicon in the on-state. With more silicon, it can be designed for higher voltage levels and higher collector currents. On the other side, double injection also makes the transistor slower than a MOSFET. More energy is needed to turn it on and off, giving higher switching energies and needed gate charge.

For a normal silicon (Si) MOSFET, the ohmic losses will become too high if designed for higher voltages and currents. This is why Si MOSFETs only are used in low power electronic circuits. With SiC MOSFETs, on the other hand, the transistors can be designed to handle higher voltages and currents, while still having reasonably low on-state resistance. The SiC MOSFET is new technology compared to the IGBT and not yet taken up by the industry. It is promising technology for applications where high switching frequency and high power is wanted, [14] and [15]. By choosing the SKM300GB12V IGBT module from Semikron, the rest of the design is still compatible with the CAS300M12BM2 SiC MOSFET from Cree. This makes it possible to try out the SiC MOSFET modules in the same design at a later time. This is interesting for further research and development on the inverter.

In figure 7.5, the turn-on characteristics for a power semiconductor transistor is shown. At t_1 the gate-emitter voltage reaches the threshold voltage where the collector current i_C starts to flow. The voltage and current characteristics from t_1 to t_4 is of importance when estimating the

power dissipation of the transistor. In equation 7.1, the conduction losses of the transistor is given. The conduction losses can be calculated from the applied collector current and collector-emitter voltage or by the on-state resistance, when the transistor is in the on-state. In equation 7.2, the switching losses are given. This is the losses coming from the turn-on and turn-off characteristics of the transistor. The turn-on loss can be calculated by integrating the product of the collector current and collector-emitter voltage waveforms over the time t_1 to t_4 in figure 7.5. Normally the turn-on and turn-off energies are given in the datasheet of the transistor for different operating conditions.

$$P_{cond} = V_{CE} \cdot i_C \quad (7.1)$$

$$P_{sw} = (E_{on} + E_{off}) \cdot f_{sw} \quad (7.2)$$

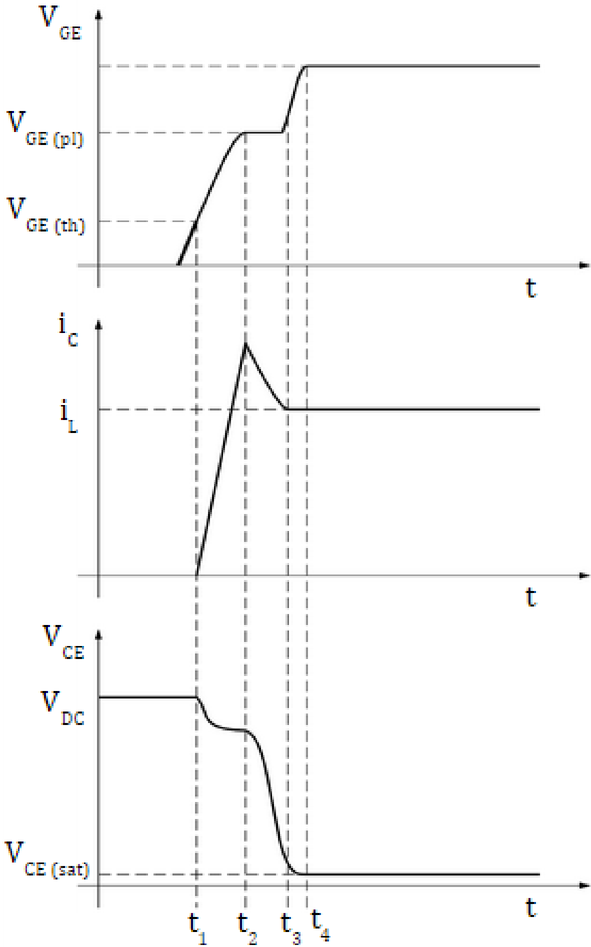

Figure 7.5: Turn-on characteristics for a power semiconductor transistor.

The total power dissipation of the transistor is an important design parameter when it comes to heatsink design. By averaging the power dissipation over a time of typical operation, an

appropriate estimation can be found. With an average battery accumulator power of 30 kW and a switching frequency of 8 kHz, a conservative power dissipation estimate is 100 W per IGBT module.

7.2. Gate Drive Circuit

A gate driver circuit is needed to charge and discharge the gate capacitances to switch a power semiconductor transistors on and off. The circuit is the interface between the low voltage control system and the voltage level of the tractive system, providing galvanic isolation between the systems. It also provides safety features like short circuit protection and interlocking between upper and lower transistor in a phase leg.

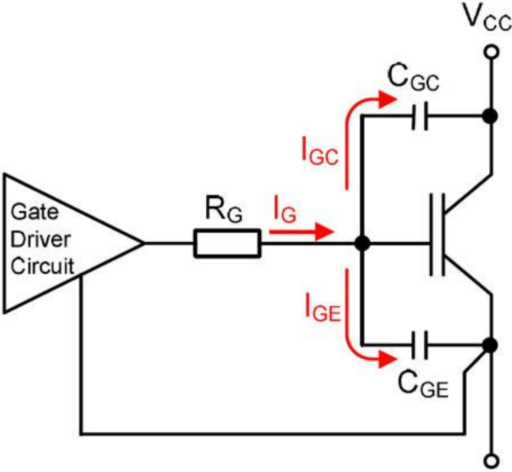

Figure 7.6: Gate driver functionality schematic.

It was decided to buy the gate drivers from Semikron, the same manufacturer as the IGBTs. The reason for not designing the gate driver circuit in-house was to reduce risks and to get a gate driver designed to work well with the chosen IGBT module. The SKHI 23/12 R driver from Semikron in figure 7.7 has all the features and functionalities required for such a circuit and is recommended by the manufacturer. One gate driver circuit board, controls the two transistors in one IGBT half-bridge module, requiring one board per phase leg. If the gate drivers had been designed in-house, the three gate driver circuit boards needed for a three phase system could have been put onto one circuit board, making the system more compact and lightweight. As previously mentioned this was not done due to the risks associated with designing the circuit in-house.

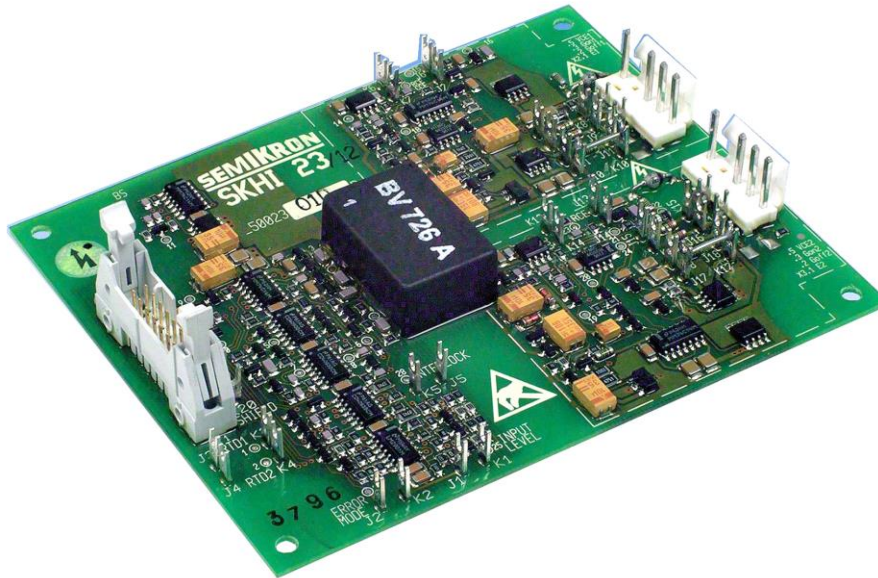

Figure 7.7: SKHI 23/12 R gate driver from Semikron.

As is clear from figure 7.5, the gate driver has to apply a voltage higher than the gate-emitter threshold value for the IGBT, before any collector current can flow. During the turn-on, the gate driver has to supply enough current to charge the gate-emitter and gate-collector (Miller capacitance) capacitances before the full collector current can flow with low conduction losses. In figure 7.5, the gate-emitter voltage reaches the threshold value at t_1. Due to the Miller capacitance, the gate-emitter voltage will be kept on a plateau from t_2 to t_3, before it reaches the voltage level which leads the collector-emitter voltage to saturation at t_4. As given in figure 7.5 and equations 7.1 and 7.2, the time between t_1 and t_4 represents the switching losses of the IGBT. The switching waveform of the IGBT can be modified on the gate drivers by changing the gate resistance values R_G in figure 7.6. By soldering on lower gate resistances, the IGBTs will be turned on and off faster. Faster switching reduces the switching losses, but leads to oscillations if set too fast.

The SKHI 23/12 R gate driver also features collector-emitter voltage monitoring, short circuit protection, soft short circuit protection, under voltage supply monitoring, interlock between upper and lower IGBT in a phase leg, internal isolated power supply and error message communication. If the control system tries to turn on both IGBTs in a phase leg at the same time, or if the supply voltage to the gate driver drops below 13 V, the gate driver will shut down the IGBT module and report an error message back to the control circuit board. In normal operation the gate driver should be supplied with 15 V from the control circuit board, together

with the switching signals for the IGBTs. How to correctly connect the driver to the IGBT module and to the control circuit board is explained in more detail in the “Inverter Prototype Setup Manual” in appendix D. This setup manual was made so that the VSI prototype easily could be modified and assembled during testing.

The functions mentioned can be modified and controlled by soldering different resistor values or strappings on the board. As already mentioned, the gate resistance can easily be modified in this way. The interlock dead time between upper and lower IGBT can also be modified in a similar way. The standard dead time for the board from the manufacturer is set to 10μs. This a conservative setup and should be modified according to the system. This is done by soldering on smaller resistors on the board. Finding the optimum dead time and gate resistor can only be done by testing and measuring the collector-emitter voltages in a test setup.

It is also worth mentioning that the standard setup for the gate driver is to control two IGBT modules in parallel. By soldering on a specific strapping on the board, this is easily changed to single IGBT module control.

7.3. Input Capacitor Bank

As explained in [1], a capacitor bank on the DC input side of the VSI is important to provide a low impedance path for ripple currents associated with the switching frequency and control loop frequency. The capacitor bank should be placed as close as possible to the terminals of the IGBT modules and be capable of absorbing enough energy to damp the DC side ripples. This is important to protect the battery accumulator against high frequency voltage ripples. On the other hand, it is also important that the stored energy can be discharged fast at shutdown of the system. As can be understood from equation 7.3, the stored energy E can become dangerous for a capacitor bank with high equivalent capacitance C_{eq} at a voltage level of 530 V. It is therefore important to find capacitors with appropriate ratings and capacitance.

$$E = \frac{1}{2} C_{eq} \cdot V_{DC}^2 \qquad (7.3)$$

The most typical types of capacitors for this application are film capacitors or electrolyte capacitors, [16]. In table 7.2, two suitable capacitor alternatives are listed. The PEH169UT433 is an aluminum electrolyte capacitor, while the 944U161K801ABM is a metallized polypropylene film capacitor. To obtain the required voltage rating and current ripple

capability, two capacitors in series and two in parallel are needed for the aluminum electrolyte capacitor alternative. While it for the film capacitor, only two in parallel are needed.

		PEH169UT433	944U161K801ABM
Voltage rating	V	350	800
Capacitance	μF	3300	160
Height	mm	105	52
Diameter	mm	75	84
Equivalent series resistance	mΩ	20	0.8
Maximum current ripple	A_{peak}^{RMS}	32.8	73

Table 7.2: Alternative capacitors for input capacitor bank.

There are also available electrolyte capacitors with higher voltage ratings, but then the size gets bigger. The capacitance and energy storage will then also become higher than desired. With the alternative with four of the electrolyte capacitors, the total capacitance becomes 1650 μF, which already is on the upper limit for this system. The VSI from last season, the Bamocar d3 from Unitek, had a DC side input capacitance of 320 μF [4], while the standard VSI from Sintef Energy [17] has a capacitance of 3300 μF. In the Bamocar d3, film capacitors are used, while there are electrolyte capacitors in the VSI from Sintef Energy. If the equivalent series resistance (ESR), maximum current ripple capability and physical size also are taken into consideration, the choice is leaning towards the film capacitor for this application. With two of the 944U161K801ABM capacitors from Cornell Dubilier, the total capacitance becomes 320 μF. This is the exact same as for the Bamocar d3, and considered as a suitable capacitance level. Because of the relatively low capacitance, the discharge circuitry also becomes smaller and simpler. On the other hand, higher capacitance could give higher performance, because of higher amount of stored energy at the inverter side of the energy meter. This will make it possible for the inverter to feed the PM motor with higher power transients, without the energy meter noticing.

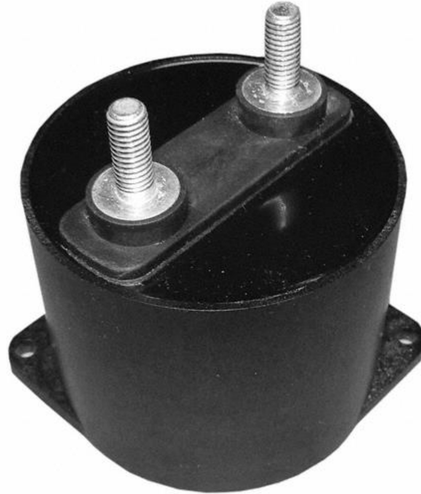

Figure 7.8: Cornell Dubilier 944U metallized polypropylene film capacitor.

Another point in this discussion is that film capacitors are bipolar. Unlike the electrolyte capacitor, the film capacitor can be connected both ways. If an electrolyte capacitor is connected the wrong way, it might blow up when energized. The choice finally fell on two of the film capacitor in table 7.2 (Cornell Dubilier 944U161K801ABM).

7.4. Discharge Circuit

According to the rules from SAE International [2], the input capacitor bank voltage of the VSI has to be discharged to a voltage level below 60 V_{DC} within five seconds after the shutdown circuit is opened. The DC side input voltage v_{dis} is given by equation 7.4, where V_0 is the initial capacitor bank voltage and R_{dis} the discharge resistance. Equation 7.5 gives the discharge current i_{dis}. The discharge resistance and current determines the power and heat production in the discharge resistor, and is thus important for dimensioning the circuitry.

$$v_{dis} = V_0 \cdot e^{-\frac{t}{R_{dis} \cdot C_{eq}}} \tag{7.4}$$

$$i_{dis} = \frac{v_{dis}}{R_{dis}} \tag{7.5}$$

Like shown in figure 7.9, the discharge resistor needs to be dimensioned so that it will discharge sufficient energy to get below 60 V within five seconds, without too high transients of power dissipation.

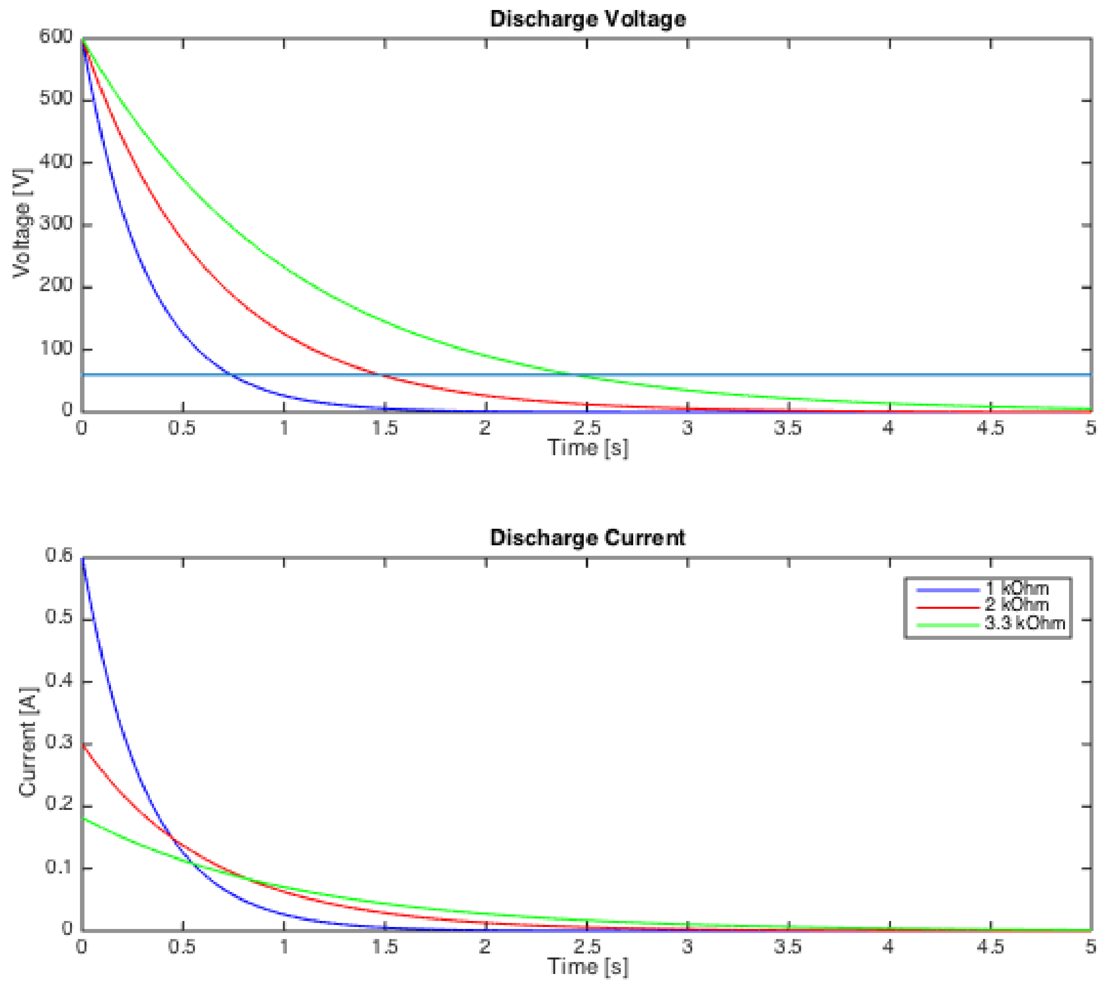

Figure 7.9: Discharge voltage and current waveforms [1].

In figure 7.9, the discharge voltages and currents for three alternative resistors of the type Welwyn WH100 (figure 7.10) are plotted. To keep the instantaneous power and heat production to a minimum and still stay within the five second rule, the 3.3 kΩ alternative was chosen.

Figure 7.10: Welwyn WH100 discharge resistor.

To control the discharging, a circuit board and two relays are used. Because the discharge circuit is a critical safety system, two discharge relays in series are used for increased redundancy. Temperature measurement on the discharge resistor is also an important safety feature.

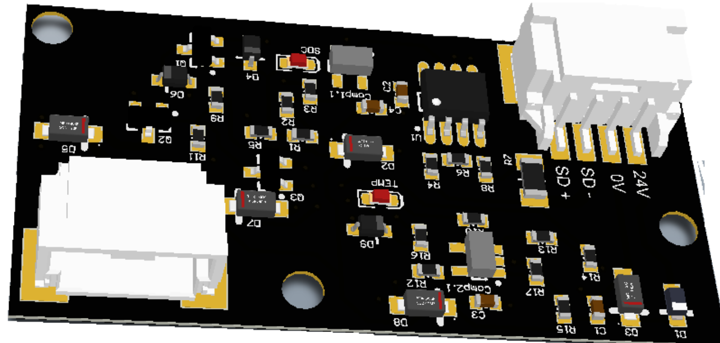

Figure 7.11: Discharge control circuit board.

The discharge control circuit board in figure 7.11 (circuit schematic drawing in appendix figure H.18) is supplied from the GLV system and controls the two discharge relays (figure 7.12) parallel. The control is based on the shutdown circuit state and the discharge resistor temperature. The Cynergy3 DBT72410P relay is a normally-closed relay and will close the discharge circuit whenever the shutdown circuit is opened. This means that if an error or fault is detected (shutdown circuit opens), or the tractive system is shut off due to transportation, storage or maintenance, the discharge circuit will be active. If the temperature measurement on the resistor crosses the limit of 80 ° C, the control circuit board will overrule the shutdown circuit and open the relays until the temperature has dropped to a safe level.

Figure 7.12: Cynergy3 discharge relay.

The reason for extra safety measures like multiple relays in series and temperature monitoring, is due to an unwanted incident with this circuit last season. Last season, one of the battery accumulators was destroyed because it was discharged to an unsafe voltage level over the discharge resistor.

7.5. Heatsink Design

As explained and discussed in [1], the VSI needs a heatsink to transport heat away from the IGBT modules. Because the PM motor is water-cooled, it is also convenient to use water-cooling in the VSI. Then the same cooling system, with radiator and hoses, can be used for both the VSI and the PM motor. When dimensioning the heatsink, the power losses from the IGBT modules need to be estimated, and an appropriate thermal model set up. The power losses P_{loss} from the IGBT modules are estimated in equation 7.1 and 7.2. The applied thermal model from [13], as shown in figure 7.13 and given in equation 7.6, is applied.

$$T_j - T_s = P_{loss} \cdot R_{th(j-s)} \tag{7.6}$$

T_j, T_c, T_s and T_a represent the temperatures at the junction of the semiconductor chips in the IGBT modules, at the module casings, the heatsink surface, and ambient.

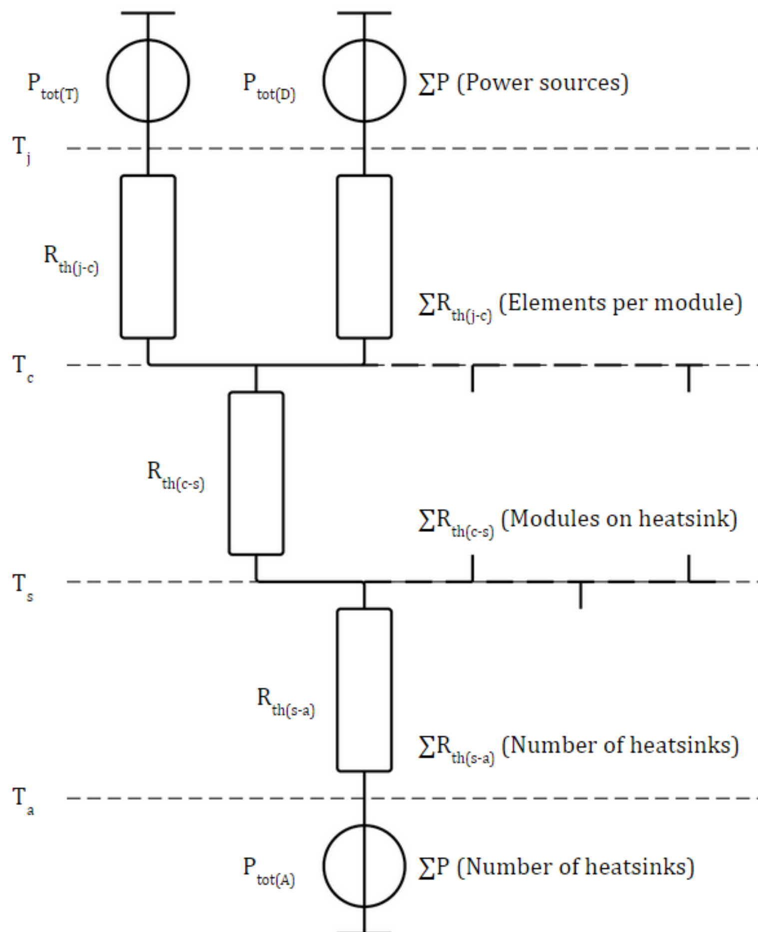

Figure 7.13: Thermal heatsink model [13].

The thermal resistances $R_{th(j-c)}$, $R_{th(c-s)}$ and $R_{th(s-a)}$ are values that is found in the respective datasheets or from simulation. They represent the thermal resistances from junction to case, from case to heatsink and from heatsink to ambient. In figure 7.14 and 7.15, renderings of the heatsink design are shown. The models are computer aided design (CAD) models made in SolidWorks. In figure 7.16, a computational fluid dynamics (CFD) analysis of the heatsink CAD model, is shown. The simulation is done using the CFD software STAR-CCM+, with a water flow rate of 0.04 kg/s, where only the heat transfer between IGBT module and heatsink is considered. In figure 7.16, each of the IGBT modules are modeled as constant power sources of 100 W, which is a conservative estimation of the normal operational condition. This value is based on the power dissipation estimations done before. With 100 W per IGBT module, the losses are still small compared to the estimated losses of the PM motor. These losses are estimated to be 2 kW under heavy load. The IGBT modules will break down due to heat at a junction temperature T_j of 175 °C. Based on equation 7.6, this is equivalent to a heatsink surface temperature of about 125 °C. With a maximum heatsink temperature of 75 °C in the CFD analysis, the design should be dimensioned appropriately.

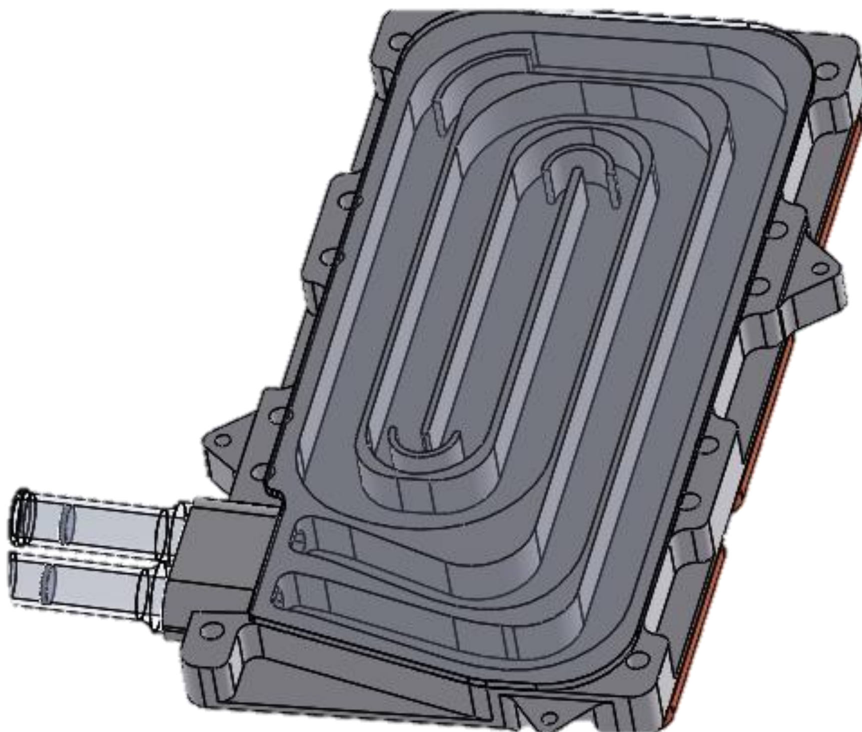

Figure 7.14: Heat sink rendering, bottom side.

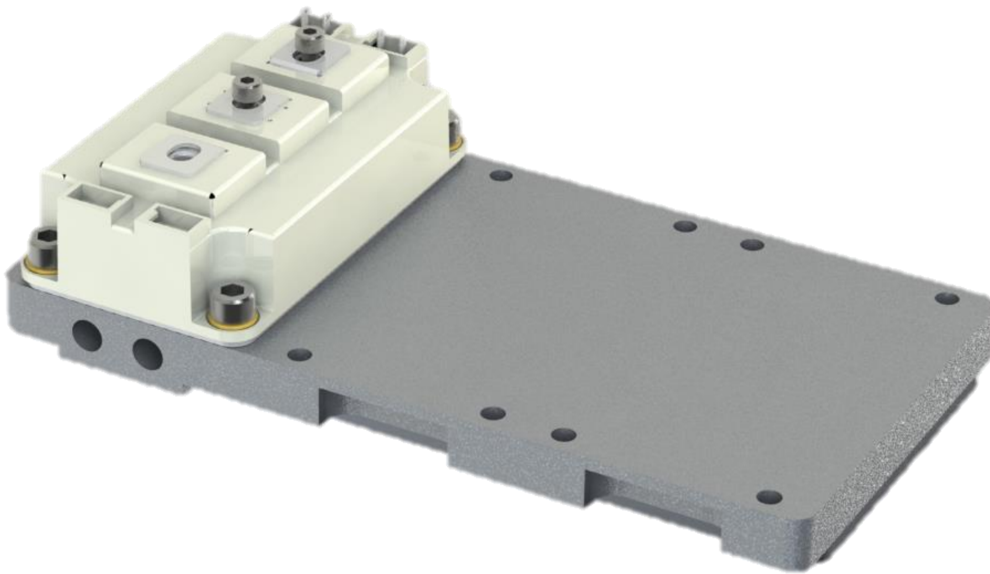

Figure 7.15: Heat sink rendering, top side, with IGBT module.

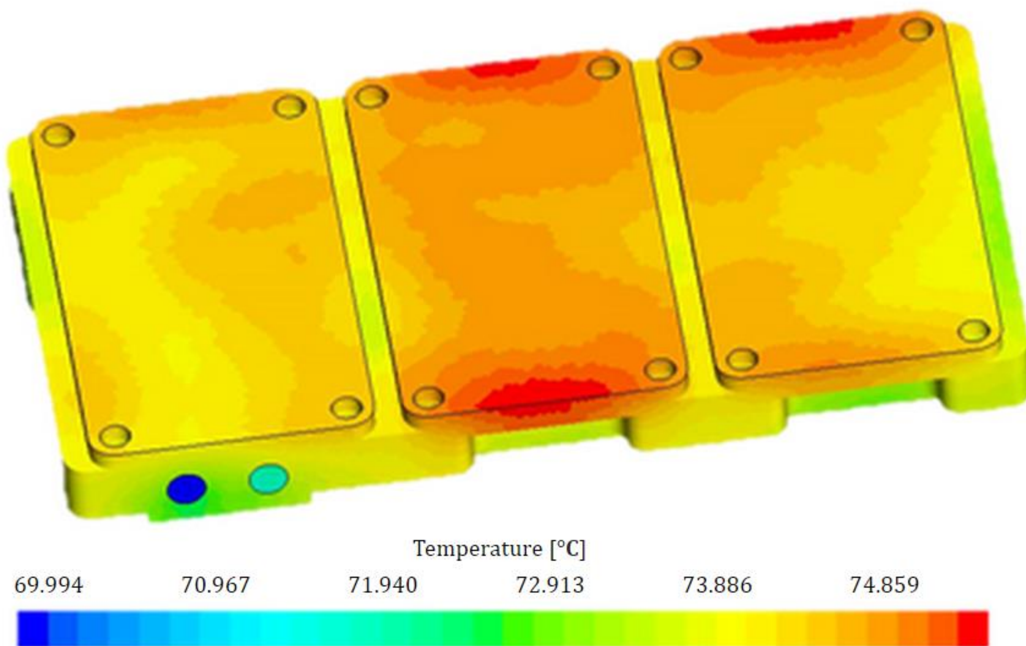

Figure 7.16: Heatsink simulation of temperature distribution.

7.6. Voltage Source Inverter Casing Design

As discussed and explained in [1], all tractive system components should be properly encapsulated and isolated from the GLV system and chassis. The components should also be able to withstand moisture in the form of rain or puddles. For the VSI, this means that the system have to be built into a sealed casing with waterproof contacts. The VSI from last season, the Bamocar d3, was delivered with a heavy aluminum casing and weighted a total of 8.5 kg. For this year's design, with a carbon fiber casing, it is estimated that the weight will be reduced to about 4 kg.

In figure 7.17 and 7.18, a CAD assembly model of the VSI casing is shown. The model is made with the modelling tool SolidWorks and the version control program PDM. It shows the casing, with IGBT modules, input capacitor bank, discharge circuit, gate drivers, control circuit board, current sensors and contacts. The wiring harness in the casing is left out of the model, but is given in detail in appendix G. The VSI wiring harness schematic drawing in appendix G gives a complete overview of the tractive system GLV wiring, and pin identifications for the contacts in the VSI casing.

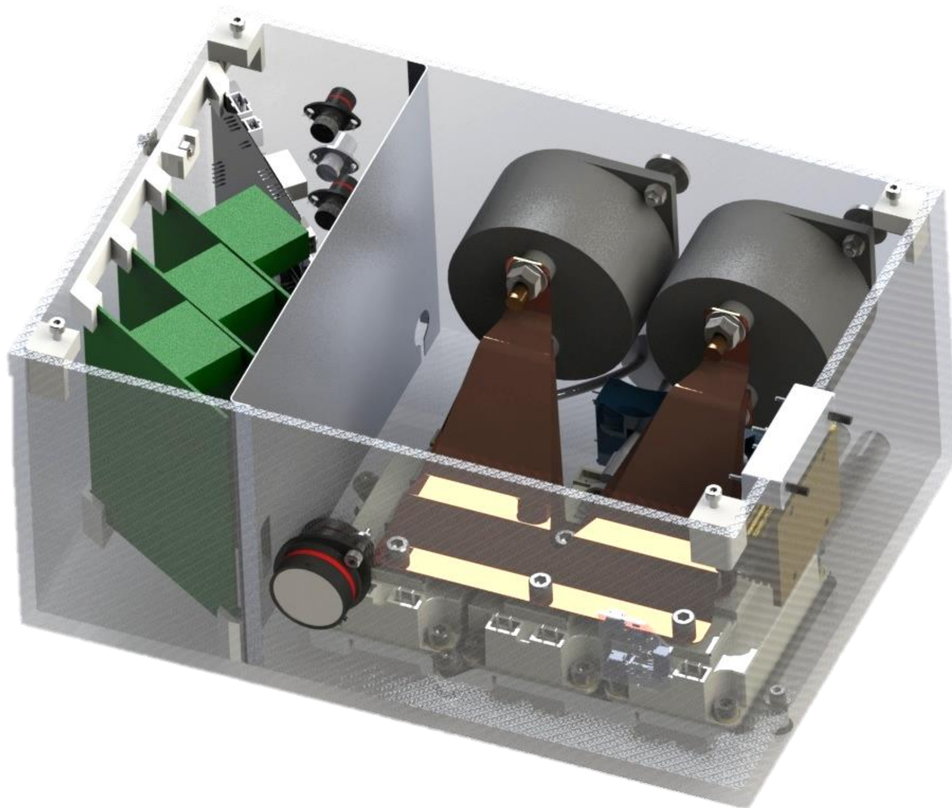

Figure 7.17: Voltage source inverter casing rendering, front side.

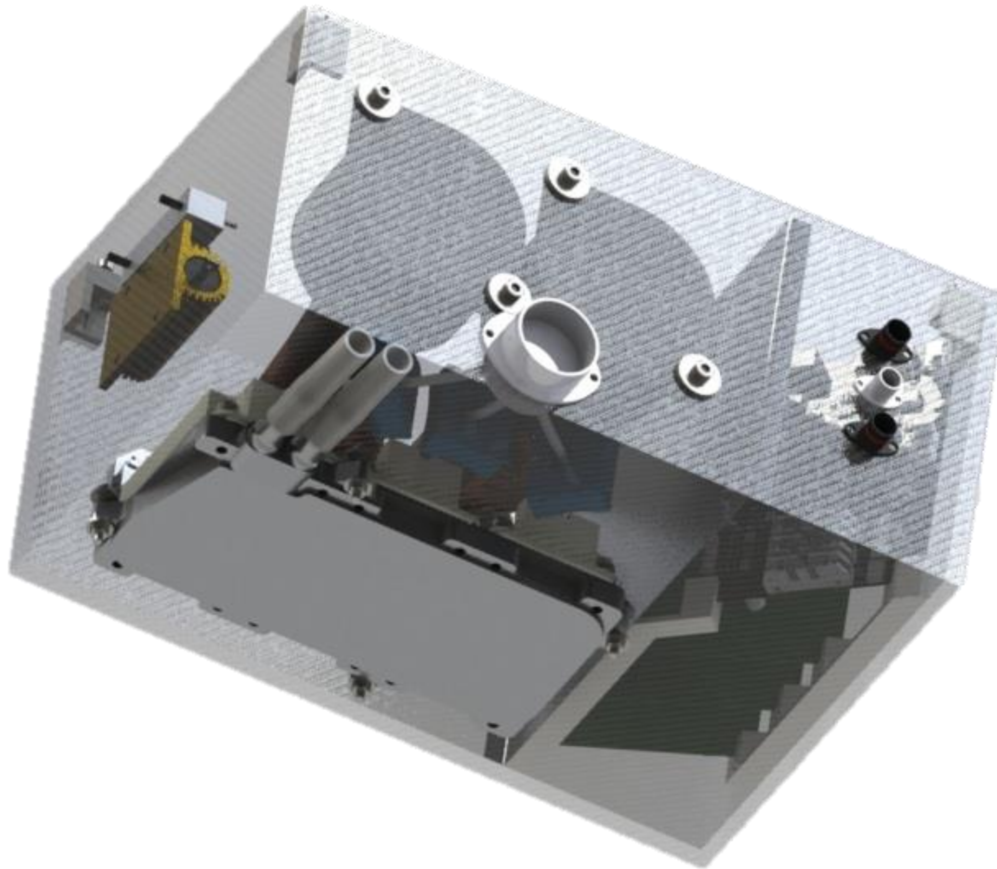

Figure 7.18: Voltage source inverter casing rendering, rear side.

7.6.1. Placement in the Monocoque

Figure 7.19 shows the placement of the VSI in the monocoque. The VSI and the energy meter are the green boxes in the figure, while the red box is the battery accumulator. The yellow structure is the motor shielding housing with differential and gear box, while the purple structure is the seat. The VSI is attached with suspension plates to the inside top of the monocoque. The suspension plates are not shown in figure 7.17, 7.18 or 7.19. One plate is glued to each side of VSI and one on the rear side, attaching the VSI to the ceiling with four bolts. The energy meter is attached to the VSI with clip bonds and designed with the same tractive system contacts, making it easy to install.

To remove the VSI, the seat and a fire wall has to be removed from the cockpit. The firewall is a thin Kapton tape-covered aluminum wall between the seat and the battery accumulator. With the seat and firewall removed, the battery accumulator can be taken out through the cockpit. Then the tractive system DC contact can be disconnected through the cockpit, and the tractive system AC contact, encoder and motor temperature contact, CAN bus and power contact and

water hoses disconnected through the rear service hatch. The rear service hatch is not shown in figure 7.19, but is located just above the motor shielding housing. Then, the four bolts attaching the VSI suspension plates to the monocoque can be loosened, and the VSI taken out through the cockpit. The VSI is not the easiest component to remove, but this was seen as the most compact way of packing the motor room. It is anyways required by the SAE International rules [2] that the battery accumulator is removed during charging. With the battery accumulator out, half of the works is already done.

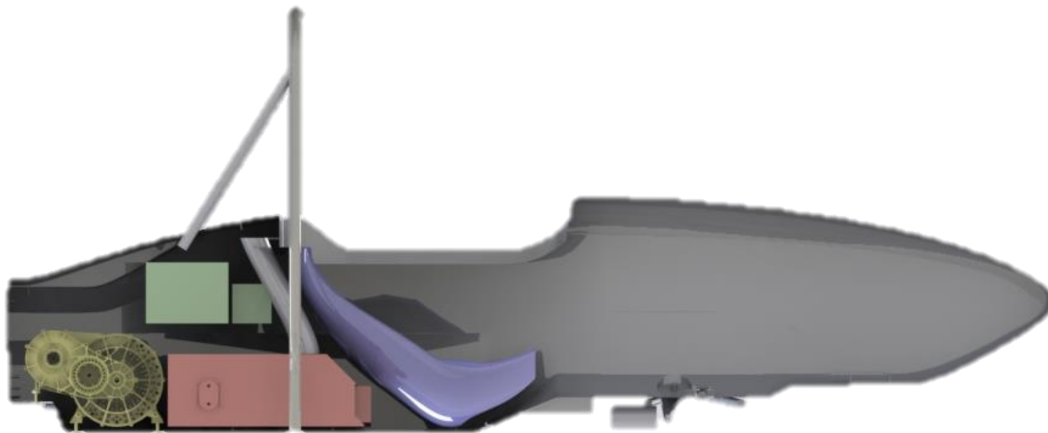

Figure 7.19: Placement in the monocoque [1].

7.6.2. DC Side Plate Conductors

According to [18], using parallel plate conductors from the capacitor bank to the IGBT modules on the input DC side of the VSI, is important for increased performance and safety. From equation 7.7, it is clear that increased inductance L leads to overvoltages V when the change in current is fast (di/dt). For a VSI, rapid change in the current is unavoidable. It then becomes important to keep the inductance as low as possible.

$$V = L \cdot \frac{di}{dt} \tag{7.7}$$

$$\frac{L}{m} \approx \mu \cdot \frac{h}{w} \tag{7.8}$$

In [18], the design of the plate conductors is done according to equation 7.8 and figure 7.20, where w is the width, h the space between the plates, m the length, μ the magnetic

permeability of the material, and L the resulting inductance. The formula in equation 7.8 is an approximation or a guidance to how the plates should be designed. To minimize the inductance, it is desirable to minimize the distance between the plates, minimize the length and maximize the width.

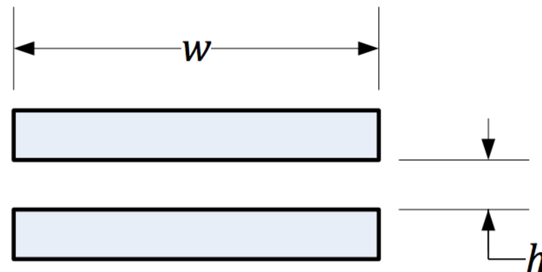

Figure 7.20: DC side plate conductors [18].

The final plate conductor design can be seen in the CAD model in figure 7.17. The DC cables are not included in this model, but are in reality connected on the middle of a copper plate between the capacitor terminals on each of the poles.

7.7. Cables and Contacts

The SAE International rules [2] require all tractive system cables to be rated for 90 °C, be colored orange, and have appropriate copper cross sectional area according to the current flow. Early in the design phase, it was decided to use 35mm^2 cables on the DC side and 16 mm^2 cables on the AC side. After realizing that the high frequency components from the VSI was contributing to more heat production on the AC side than the DC side, the cables were changed to 25 mm^2 on both the AC and DC side. The shielded cable Ölflex FD 90 CY was chosen because of its flexibility and robustness. The manufacturer recommends [20] it for motors driven by frequency converters and for automotive applications. As dictated from the rules, the cable is colored orange and rated for 90 °C. By using the cable shielding it is possible to include the grounding of the tractive system components in the cable.

All the contacts used in the VSI are Deutsch contacts from the autosport series. They feature a compact design, positive locking mechanism and a conductive aluminum alloy finish for grounding purposes. The contacts used in the VSI are listed in table 7.3. To avoid mixing up

the contacts with equal number of pins, one male and one female contact are used. The Deutsch AS Heavy Duty contacts for the tractive system also feature smaller pins for possibility to include signal cables. In table 7.3, this is shown with the number inside the parentheses. These pins are used to bring the shutdown circuit into the VSI. By letting the shutdown circuit through the tractive system contacts, the circuit will be opened whenever the contacts are disconnected.

	Pins:	Contact:
DC side tractive system	2 (3)	Deutsch ASHD 2-way (male)
AC side tractive system	3 (2)	Deutsch ASHD 3-way (female)
Encoder and motor temperature	13	Deutsch AS (female)
CAN bus 1 & 2 + GLV supply	6	Deutsch AS mini (female)
Programming	6	Deutsch AS mini (male)

Table 7.3: Voltage source inverter contacts.

8. Assembly and Manufacturing

8.1. Prototype Voltage Source Inverter Assembly

The prototype inverter casing was made by Lexan and aluminum. The walls and lid are Lexan, while the floor is a thin aluminum plate. The prototype was built larger than the original design, to make room for unforeseen design changes and to be able to fit measuring equipment. As can be seen in figure 8.1, 8.2 and 8.3, the prototype is much alike the design model from figure 7.17 and 7.18, with IGBT modules, capacitor bank, DC side plate conductors, gate drivers and aluminum shielding wall towards the low voltage section.

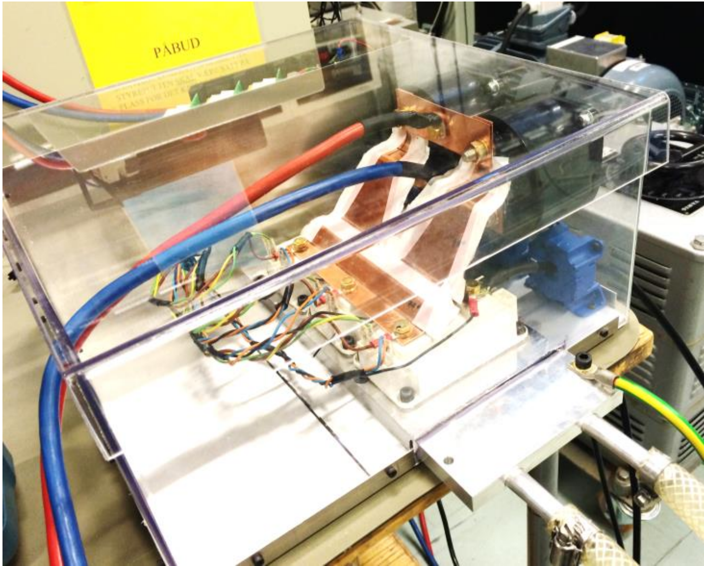

Figure 8.1: Voltage source inverter prototype, right side.

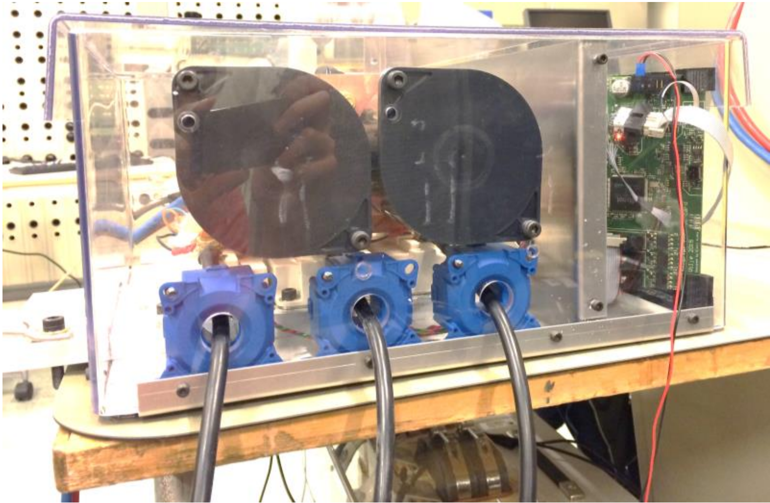

Figure 8.2: Voltage source inverter, rear side.

The water-cooled heatsink shown in figure 8.1 was a spare part from last season. In the prototype setup, it is meant to be cooled with tap water from the sink. The water hoses in figure 8.1 are connected to the nearest sink in the laboratory. The prototype VSI is grounded through the heatsink, with the green/yellow cable shown in figure 8.1.

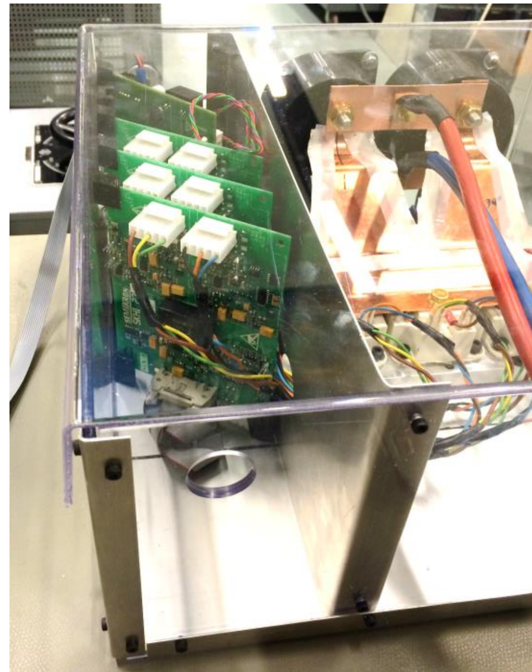

Figure 8.3: Voltage source inverter prototype, left side.

For the cabling and wiring, no contacts are used in the prototype. Using Deutsch contacts for this application would be too expensive. All the cables and wires are therefore led through holes in the casing. The holes are located at approximately the same position as the contacts in figure 7.17 and 7.18.

A more detailed description of the prototype VSI for setup and assembly is given in the "Inverter Prototype Setup Manual" in appendix D and in the "Voltage Source Inverter Wiring Harness" in appendix G.

8.2. Heatsink Manufacturing

The heatsink design was milled from an aluminum base piece as shown in figure 8.4, while the base plate was cut from a thin aluminum plate. The nipples for connecting the water hoses to the heatsink was turned in a lathe. The base plate and nipples were glued onto the heatsink with the glue Araldite AW4859. To increase the surface contact and thermal conductivity between

the IGBT modules and heatsink, thermal grease was used. The thermal grease was applied to the base plate of the IGBT modules before mounting them to the heatsink, like given in [21]. The total weight of the heatsink, with lid, nipples and glue ended up on 396 g.

Figure 8.4: Aluminum heatsink with base plate and nipples.

8.3. Carbon Fiber Casing Manufacturing

The inverter casing is made by four layers of TeXtreme carbon fiber with Rohacell IG31 core material in the middle. As shown in figure 8.5, a negative mold of MDF (medium density fiberboards) was made to get the wanted shape for the casing. The surfaces of the mold was covered with epoxy, before the carbon fiber and core was glued to the walls of mold, like shown in figure 8.6. Then, the mold was vacuum packed, like in figure 8.7, and put into an oven to cure for several hours.

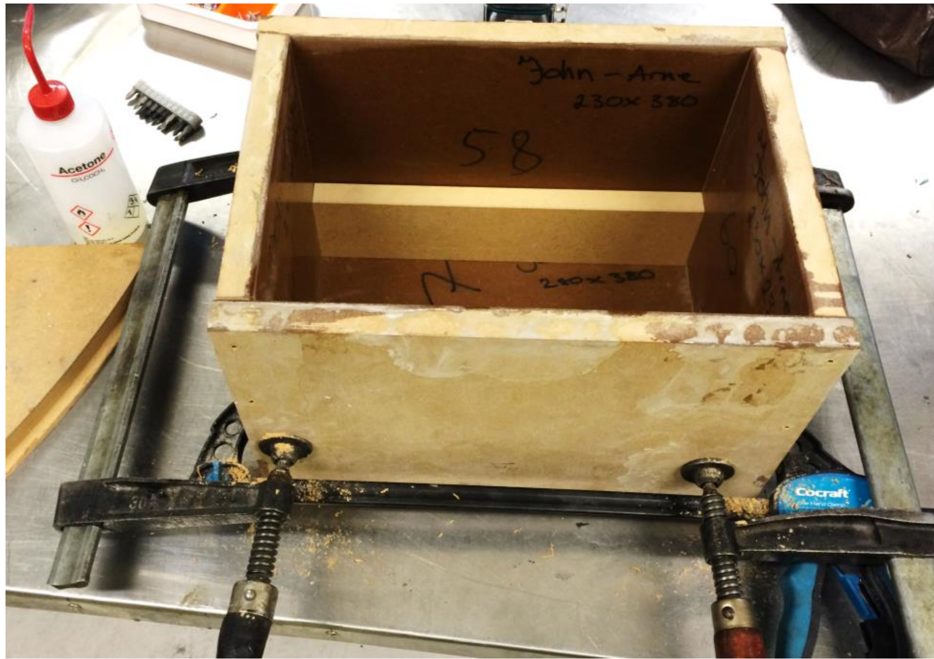

Figure 8.5: Carbon fiber casing mold.

Figure 8.6: Carbon fiber casing production.

Figure 8.7: Carbon fiber casing vacuum packing.

After detaching the mold, the result ended up like in figure 8.8. The outside finish could have been better, with rough surfaces and pieces of MDF from the mold stuck in between the fibers. The inside finish on the other hand, ended up nice and smooth. For better visual finish, a positive mold might have given a better result.

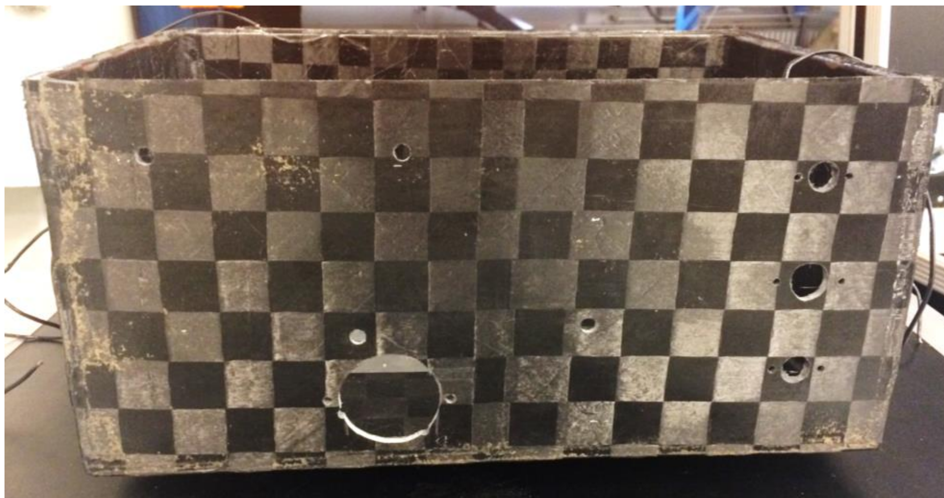

Figure 8.8: Carbon fiber casing production result.

Although the finish was not perfect, the casing is mechanically strong and extremely lightweight. The total weight of the casing, with lid, ended up on 320 g.

8.4. Voltage Source Inverter Assembly

As shown in figure 8.9 and 8.10, the input plate conductors are bended and covered with Kapton tape, connecting the IGBT module terminals to the input capacitors. Same as for the plate conductors, the internal shielding plate between the tractive system section and the low voltage section, is covered with Kapton tape. Kapton tape is used to isolate all grounded conductive parts located close to current conductive components. The DC cables are attached to the plate conductors and capacitor terminals with cable lugs and bolts to thin copper plates. As shown in the figures, all conductive bolts and nuts in the VSI that are not supposed to conduct current, are grounded with wires to the heatsink. This includes the bolts attaching the contacts, capacitors, lid, discharge resistor and internal shielding plate, to the casing. To the right in figure 8.9, the discharge resistor and discharge relays can be seen. The discharge circuit board is not shown in these pictures, but is located at the right side, next to the discharge relays. The purple wires in the pictures are the shutdown circuit, coming in through the DC contact, going through the discharge circuit board, through the AC contact, and back and out again through the DC contact.

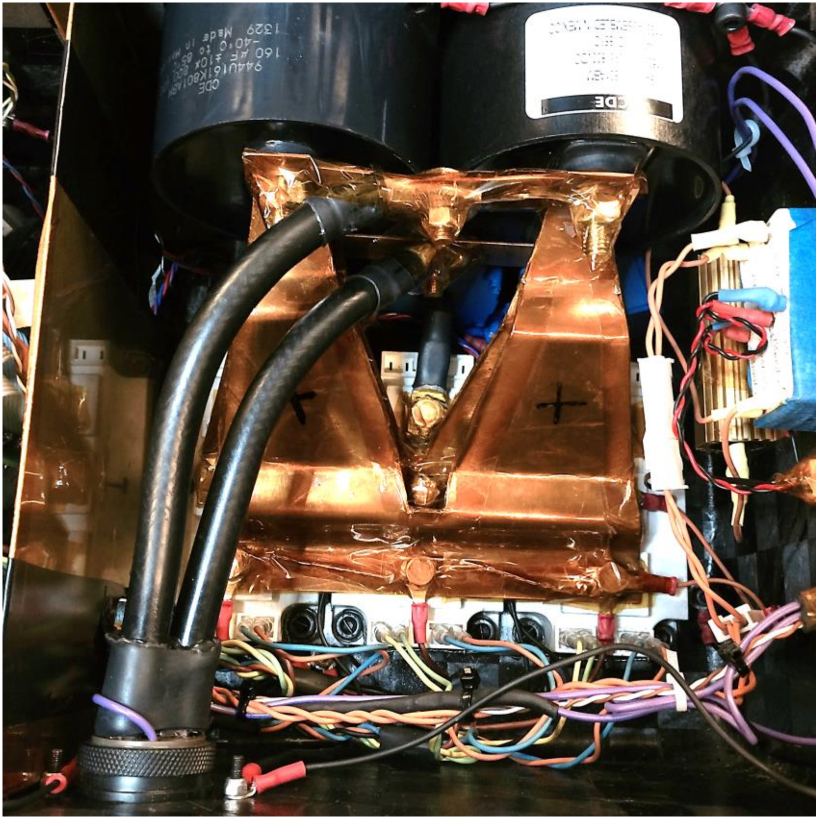

Figure 8.9: Voltage source inverter assembly, tractive system section.

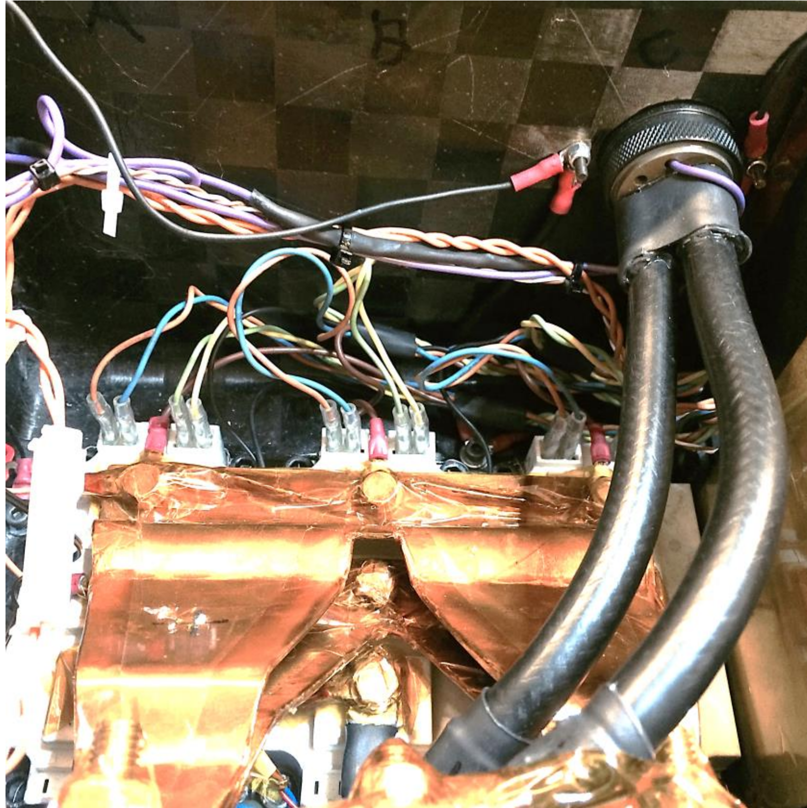

Figure 8.10: Voltage source inverter assembly wiring.

Figure 8.11: Positive locking of the AC terminal contacts.

From the SAE International rules [2], all tractive system components need to be locked together with positive locking. The Deutsch contacts already feature positive locking. For the connections to the DC and AC terminals on the IGBT modules, lock wire is used to lock the bolts, as shown in figure 8.11 and 8.12. For locking the bolts on the capacitors, elliptical offset nuts are used.

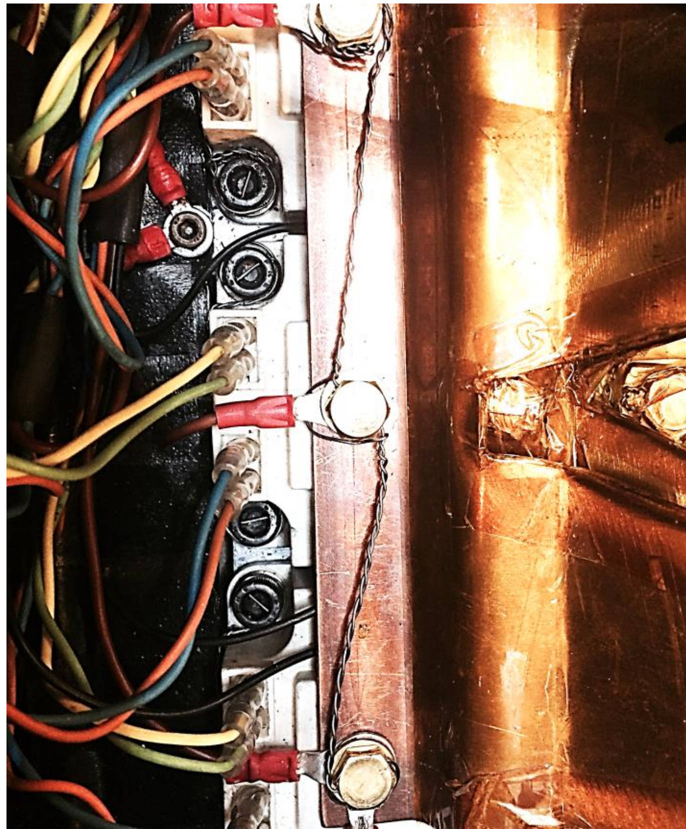

Figure 8.12: Positive locking of the conductor plates.

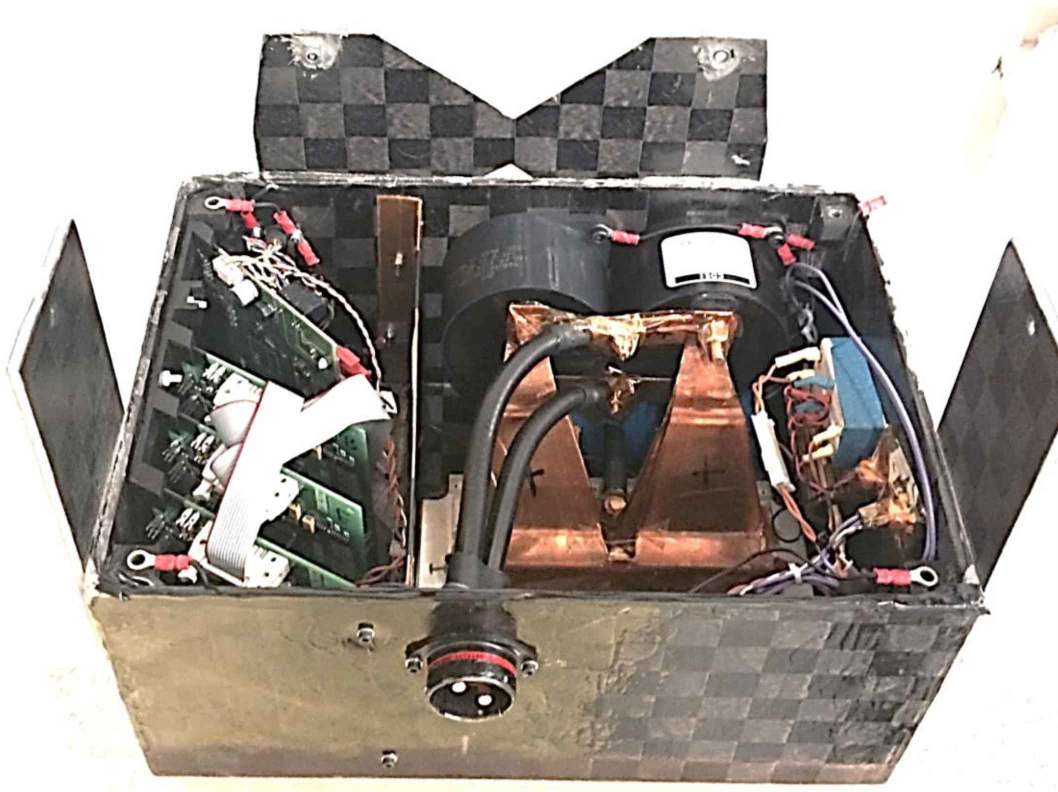

Figure 8.13: Voltage source inverter assembly from above.

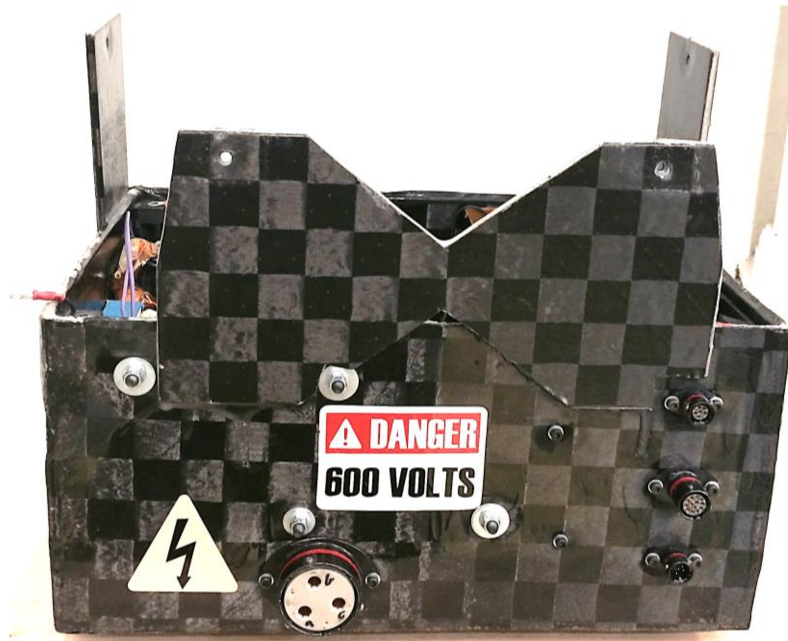

Figure 8.14: Voltage source inverter, front side.

In figure 8.13 and 8.14, the suspension plates discussed in chapter 7 are shown. They are glued to the inverter casing and attached to the monocoque with aluminum brackets and bolts. In the pictures, the Deutsch contacts discussed in chapter 7 can also be seen. The circuit board stacking solution barely shown in figure 8.13, are three 3D printed spacers. Two spacers are glued to the floor of the VSI, while one is attached to the upper left wall with nylon bolts. These three spacers make sure the circuit boards are hold in place.

As mentioned before the carbon fiber casing and lid ended up weighing 320 g, while the heatsink with base plate and nipples ended up on 396 g. The weight of the input plate conductors ended up on 230 g. Together with the weight of the IGBT modules, capacitors, contacts, current sensors, circuit boards, and so on, the total inverter weight ended up on 4.4 kg. This is almost half of the weight compared to last year's VSI of 8.5 kg.

8.5. Assembly in the Monocoque

As is shown in figure 8.15, the inverter fits well with the motor housing, in the motor room. In the picture, the three motor phases and the water hoses can clearly be seen, together with the encoder and motor temperature cables and contacts to the inverter. The contacts on the right side are, starting from above, GLV power and CAN, encoder and motor temperature, and contact for programming. All the contacts are easy to reach from the rear service hatch. The

programming contact makes it possible to reprogram the motor control circuit board without removing it from the monocoque. This is convenient when testing and debugging the system.

On the right side of the picture in figure 8.15, the GLV battery is barely shown. It is designed with the same battery cells as the tractive system battery accumulator and supplies the vehicle with GLV power of about 26 V. In figure 8.16, the inverter and energy meter is shown from the cockpit side, without the seat, firewall and battery accumulator. The energy meter is attached to the inverter with click bonds. It is easy to install and remove due to the use of the same Deutsch DC contact as on the inverter.

Figure 8.15: Motor room assembly, voltage source inverter and PM motor.

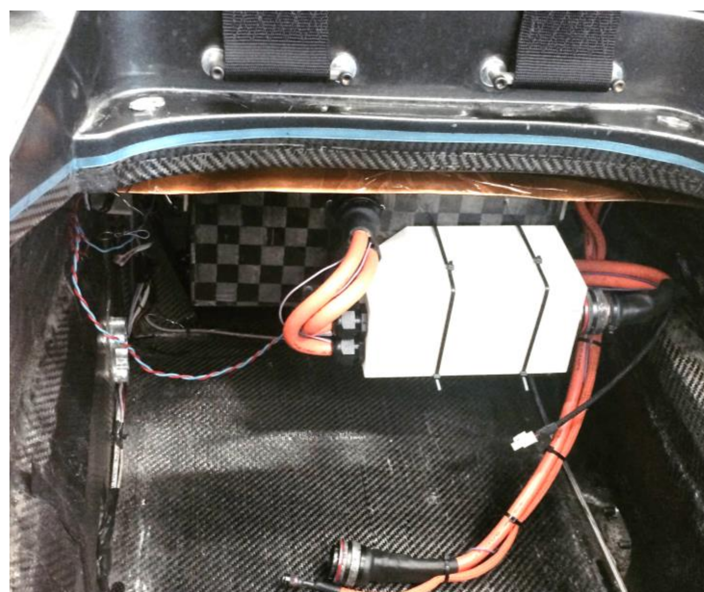

Figure 8.16: Energy meter attached to the voltage source inverter.

9. Motor and Motor Control Testing

9.1. Initial Laboratory Test Setup

The first testing was done with an IGBT module, gate driver, and a prototype control circuit board with only the gate driver interface. This circuit board was capable of turning on and off the IGBT and receiving error messages from the gate driver. This prototype was made to make sure the gate driver interface was designed correctly. In figure 9.1, the gate-emitter voltage applied in this test is shown.

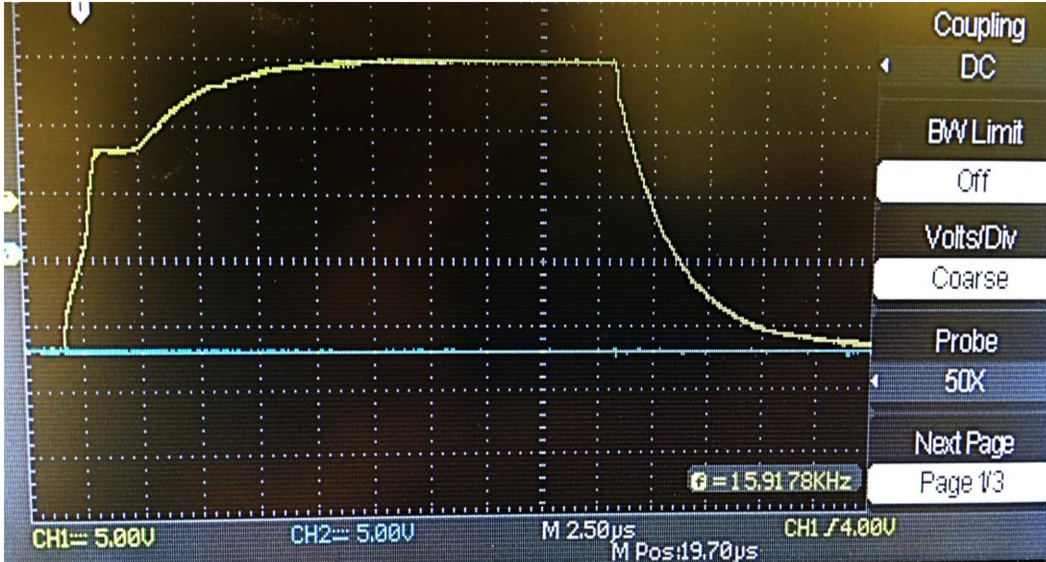

Figure 9.1: Gate-emitter voltage measurement during initial testing.

When the control circuit board design was finished and the circuit board produced and assembled, the initial laboratory test setup was started. As shown in figure 9.2 and 9.3, the DC voltage supply was an isolation transformer, autotransformer, rectifier and capacitor bank. The isolation transformer was used to bring the three-phase 230 V mains voltage up to 400 V with a ΔY-connection. The autotransformer was used to make it possible to run on different voltage levels and for safety reasons (internal fuses on the phases). The three-phase rectifier and the capacitor bank (the black boxes in figure 9.3) were used to rectify and smooth the AC voltage.

With the prototype inverter from chapter 8, the system was connected to a static load. This load was a three-phase 2000 μH inductor with a resistive load in series, as shown in figure 9.3. Initially the inverter was not cooled due to low power operation. The power dissipation could then be observed by feeling the heatsink temperature. Without cooling, the aluminum heatsink

was easily heated. When higher currents and voltages were applied, the cooling was properly installed. In the test setup, water hoses were supplying the heatsink with tap water from a nearby sink.

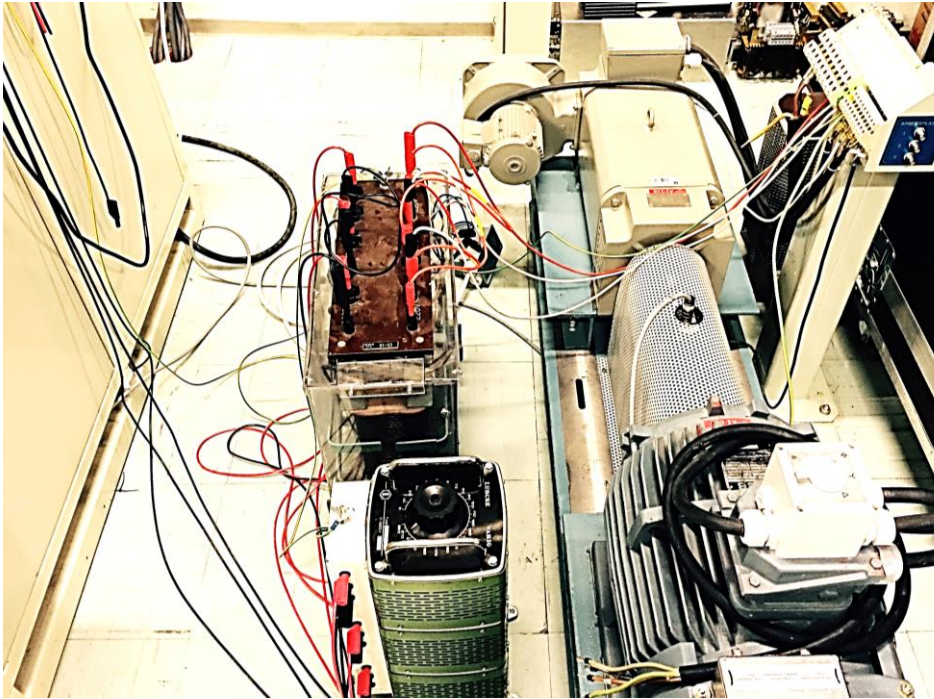

Figure 9.2: Supply voltage source for initial test setup.

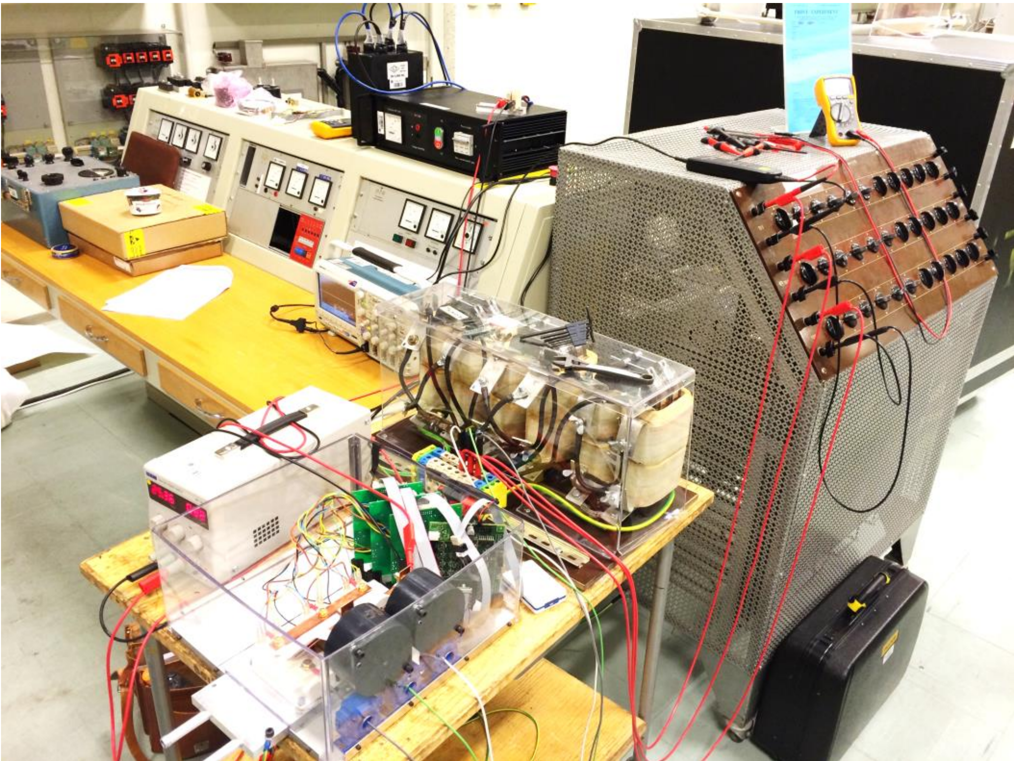

Figure 9.3: Initial test setup with static load.

In figure 9.4, the phase current waveforms with the static load is shown. The VSI was not running any control system at this point. The waveforms were generated by software PWM from the control circuit board. Software PWM was used because the PWM pins on the microcontroller was not yet connected. At first, the concept was to use hysteresis control and not PWM. This was later changed, and the PWM pins on the MCU connected to the gate drive interface.

As can be seen in figure 9.4, the waveforms are smooth and with small ripples. This is because of the high inductance of the load. With 2000 μH, this was a factor of 10 higher than the stator inductance of the PM motor in the car.

Figure 9.4: Software PWM with static load.

9.2. Laboratory Test Setup with Load

The first test with the VSI driving a motor, was done on an asynchronous motor without feedback control and with software PWM. The motor power and speed was controlled by adjusting the autotransformer voltage to the DC bus supply. This setup was used to start configuration the current measurement circuits and program code for ADCs and SPI bus.

The asynchronous motor was changed to a synchronous motor as the laboratory setup was moved to new motor lab. The new motor was the one shown in figure 9.5. This setup was used

to further develop the current and rotor position sensing circuits and code. As can be seen at the right side of the motor in the picture, a shaft encoder was attached to the rotor shaft.

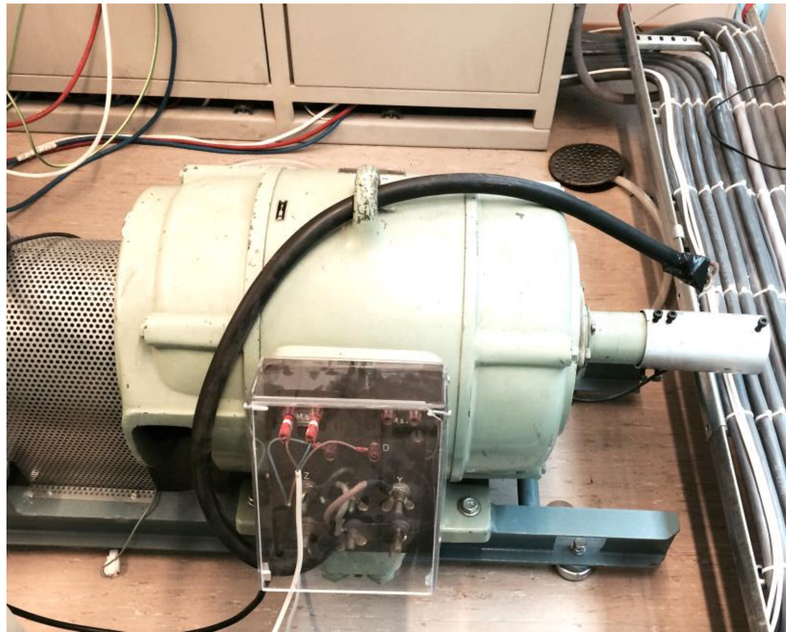

Figure 9.5: Testing with a synchronous motor in laboratory test setup.

The synchronous motor in figure 9.5 was used until the Emrax 228 PM motor, for the car, was ready and installed in the rig shown in figure 9.6 and 9.7.

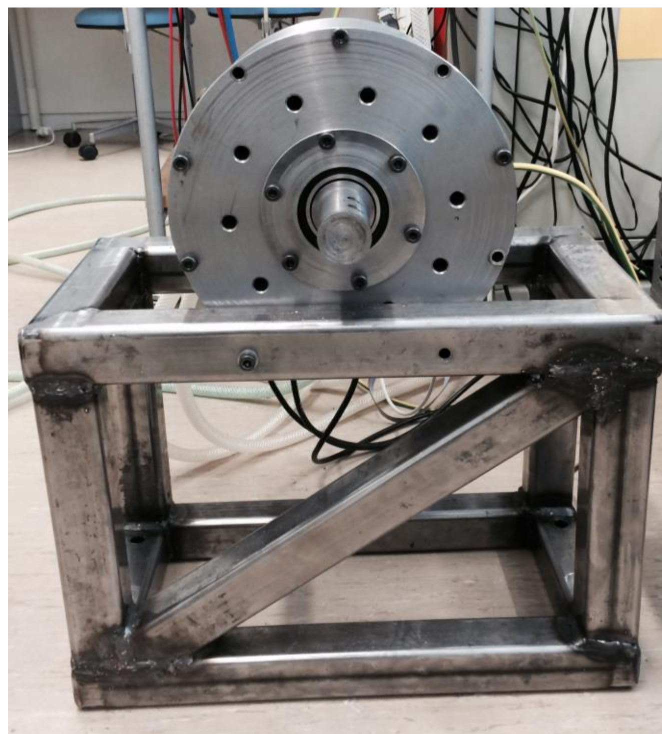

Figure 9.6: Initial PM motor test rig, front side.

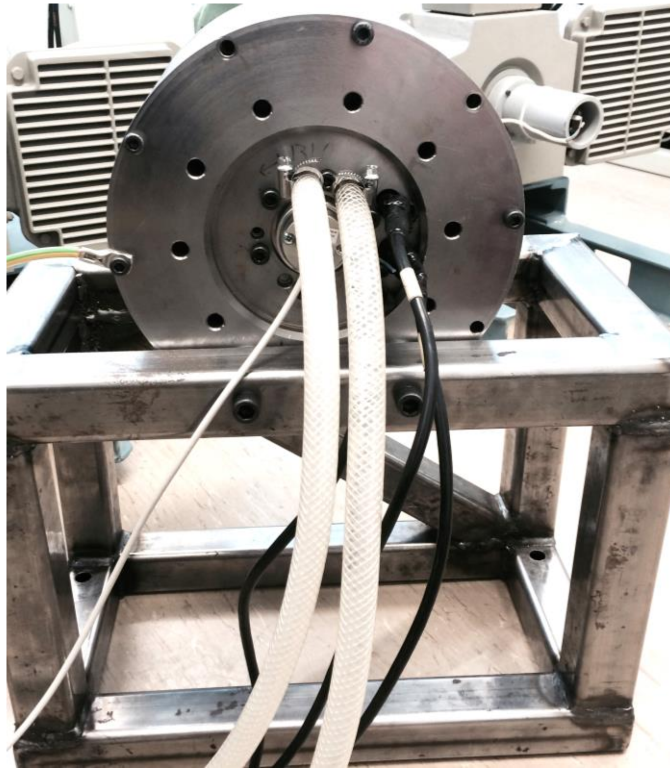

Figure 9.7: Initial PM motor test rig, rear side.

The Emrax 228 rig was initially run without load. The PM motor was connected on the same water-cooling circuit as the VSI heatsink, with tap water from a nearby sink, and the encoder and stator phase cables connected as shown in figure 9.7. The FOC system with current and rotor position sensing was developed in this setup. Effort was put into finding the optimum offset value for the encoder and to develop an initialization procedure for the ABZ encoder. The encoder offset value was initially found by trying out different values, listening to the sounds coming from the motor and inspecting the current waveforms. Later, the method proposed in [22] was used. This is a procedure where a fixed current vector is applied the motor. The rotor flux will then align itself with the q axis. Because the rotor is presently at standstill, the position of the rotor flux is known. This position can then be defined as the d axis. By doing this the permanent-magnet rotor flux (d axis) and the stator flux become phase shifted by 90 electrical degrees, which produces the output torque when the fluxes start rotating.

The next step was to connect the PM motor to a load. This was done as shown in figure 9.8 and 9.9. The rotor of the PM motor was connected to the rotor shaft of a DC machine with a claw connection as shown in figure 9.9. This way, the output torque from the PM motor could run the DC machine as a generator. The generated energy could then be sent back to the grid. By controlling the torque on the DC machine, the load could easily be changed.

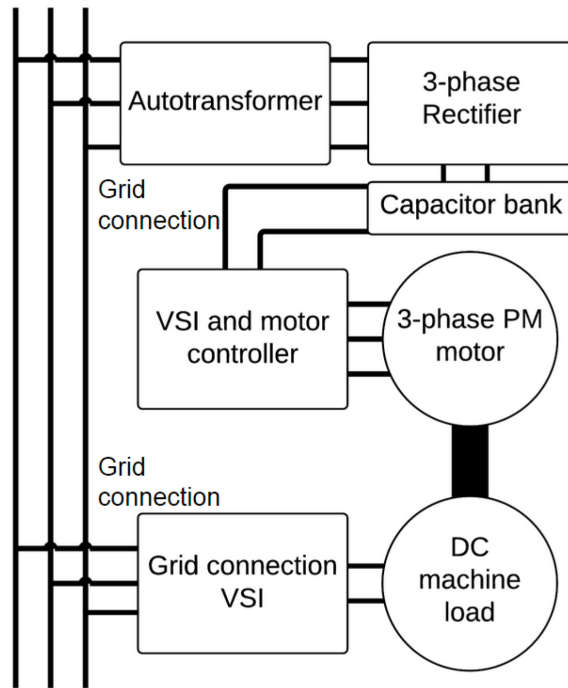

Figure 9.8: Laboratory test setup schematic drawing.

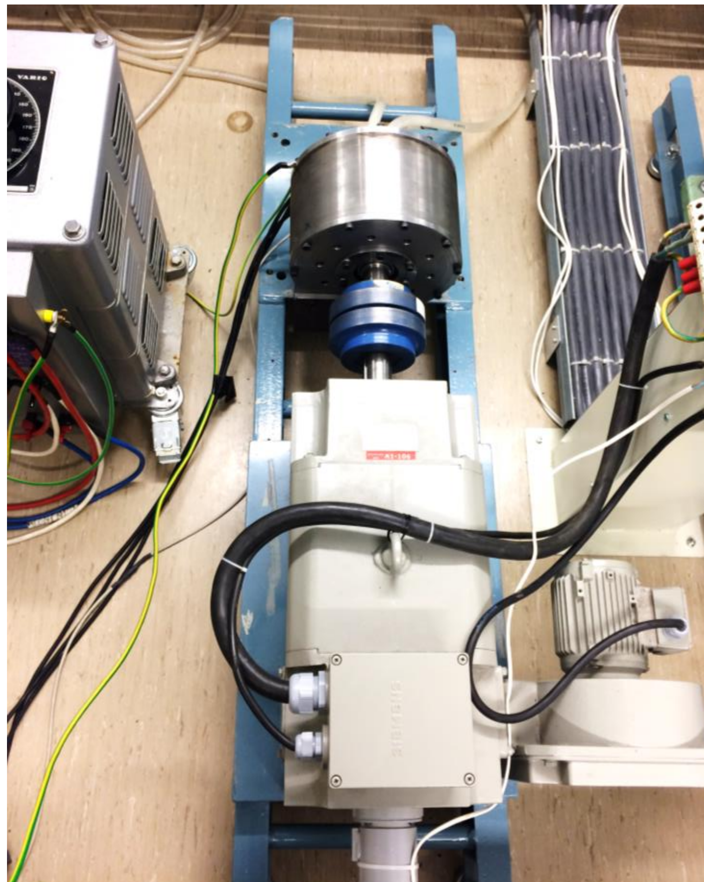

Figure 9.9: Test setup with PM motor and DC machine load.

At this point, the control circuit board design had been changed to include hardware PWM from the MCU and the SSI encoder had been changed to ABZ interface.

9.3. Laboratory Test Setup with Load and Battery Accumulator

The next step of the testing phase was to use the battery accumulator as voltage source, instead of the rectified grid voltage. The battery accumulator, with its eight battery modules, and 288 battery cells, is shown in figure 9.10. At the output of the accumulator, there are AIRs at each of the battery poles and a fuse in series with one of the poles. If the shutdown circuit is opened, the AIRs open both the poles. In the laboratory setup, the shutdown circuit was opened if the IMD or the BMS reported an error or fault. In addition to this, the precharge and discharge circuits had to be installed before the battery accumulator could be used. In the previous setup, the three-phase rectifier had provided the VSI with pre- and dis- charging. The precharge circuit was now located in the battery accumulator, while the discharge circuit in the VSI.

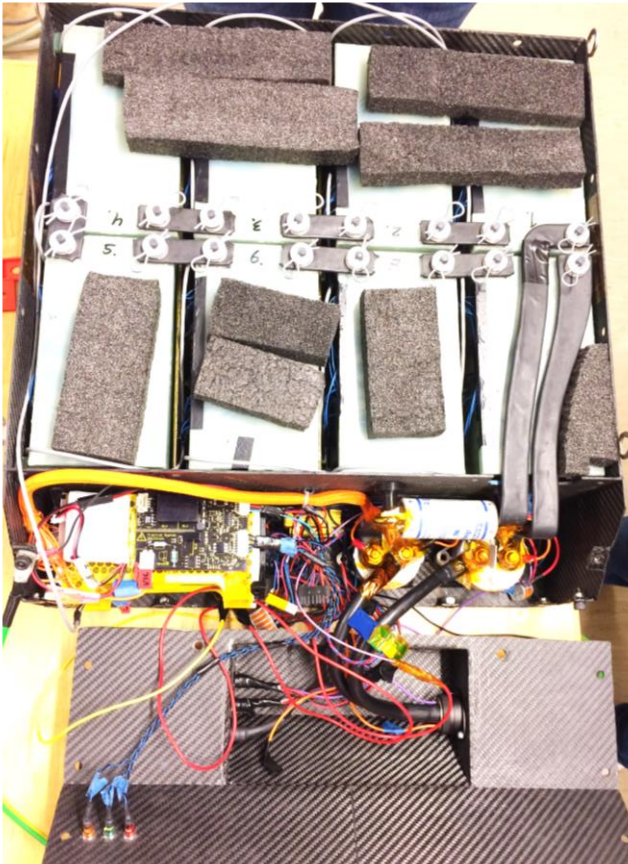

Figure 9.10: Battery accumulator in the laboratory test setup.

As can be seen in figure 9.11, the test setup was expanded to include parts of the vehicle communication network. The prototype inverter was connected to the dashboard, torque pedal with TPS units, ECU and a telemetry unit. Because both the TPS units are used in the torque pedal, the network was using both the CAN busses in the vehicle. With this setup, the VSI was driving the load-connected PM motor on command from the torque pedal, while the dashboard and telemetry unit provided information about the system states and errors.

Figure 9.11: Test setup with torque pedal, ECU and dashboard.

In figure 9.12, the phase currents and DC bus voltage waveforms are shown. The picture is an oscilloscope view with current probes (Fluke 80i-110s) on each of the phases, and differential voltage probe (Tektronix P5200A) on the DC bus battery voltage. As can be seen in figure 9.12, the waveforms are smooth and with low ripples. The picture shows the system under heavy load, with more than 200 A peak-peak phase currents and at very low speed. As is clear from the figure, the VSI was well tuned for low speed conditions at this point. The reason for this, was that the load machine was not rated for high speed operation. Because of this, all the testing at the time, was done at low speeds. Higher speeds also led to vibrations in the test rig, and was not considered safe. The vibrations were a result of a small deviation between the two rotor shaft connections, due to high tolerances in the in-house constructed motor rig.

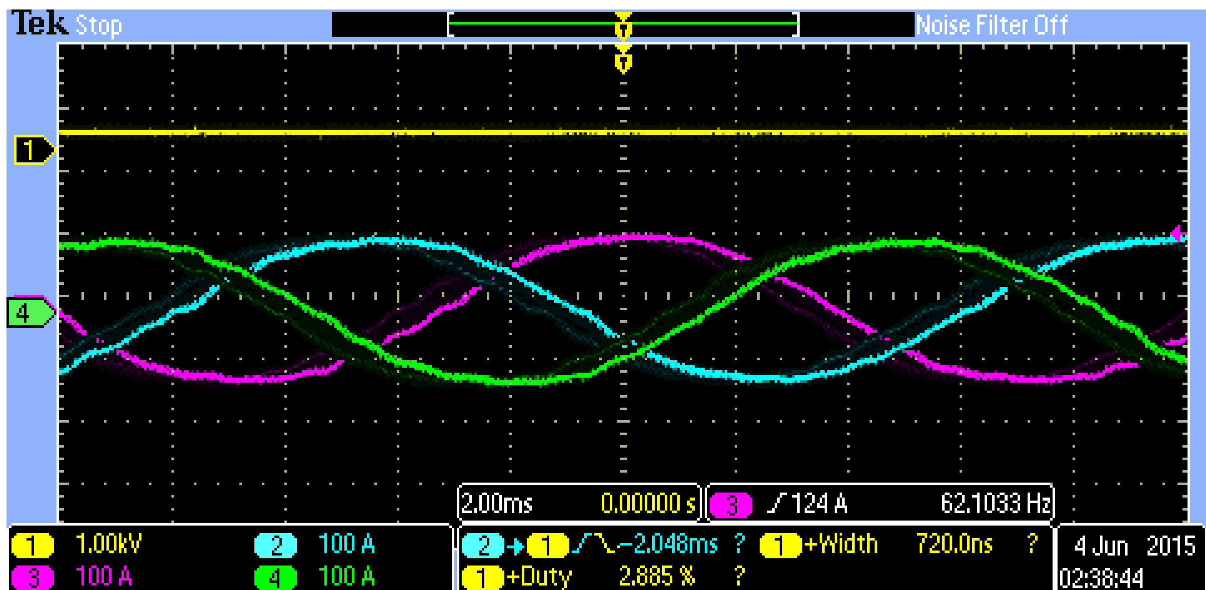

Figure 9.12: Phase current and DC voltage waveforms under heavy load conditions.

With the battery accumulator as voltage source for the VSI, issues with electromagnetic interference (EMI) were encountered. In figure 9.13, the common-mode voltage between the accumulator carbon fiber casing and the tractive system poles is shown. Because the IGBTs are switching between 0 V and 530 V at a high frequency, the heatsink-to-IGBTmodule connection becomes like a capacitor. With the accumulator casing grouned in the VSI heatsink, the resulting common-mode voltage became like in figure 9.13. The frequency of the waveform is the same as the switching frequency of the IGBTs.

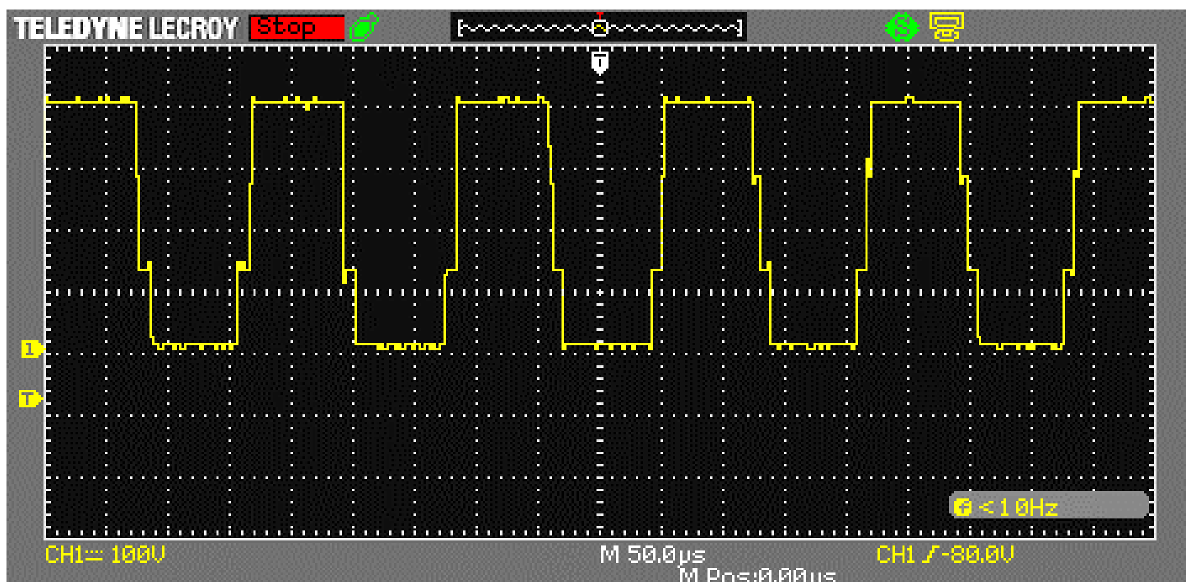

Figure 9.13: Common-mode voltage measured in the battery accumulator container.

In the beginning, the common-mode voltage caused trouble for the BMS in the battery accumulator. The BMS communication with the battery cells was interrupted by EMI, resulting in the BMS shutting down the system. The problem was solved by changing to shielded cables for the battery communication, and by proper grounding of the circuit boards.

In figure 9.14 and 9.15, the PM motor is driven at heavy load and at a faster speed than in figure 9.12. In the figures, ripples in the waveforms are clearly visible. As can be seen in figure 9.15, the frequency of the ripples coincides with the control loop frequency of 16 kHz.

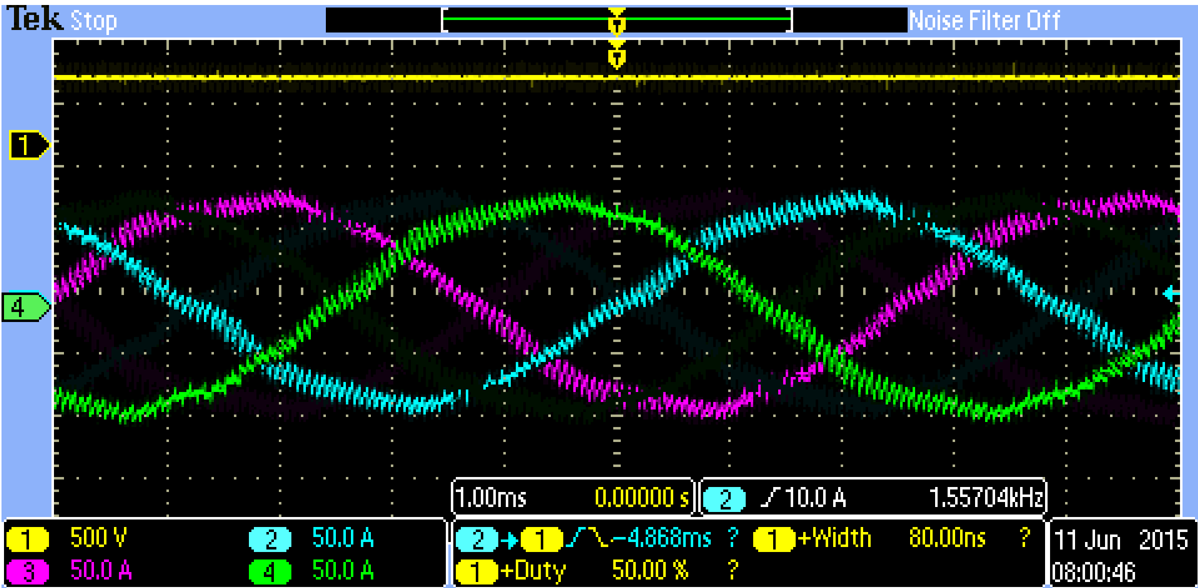

Figure 9.14: Voltage and phase current ripples from control loop frequency.

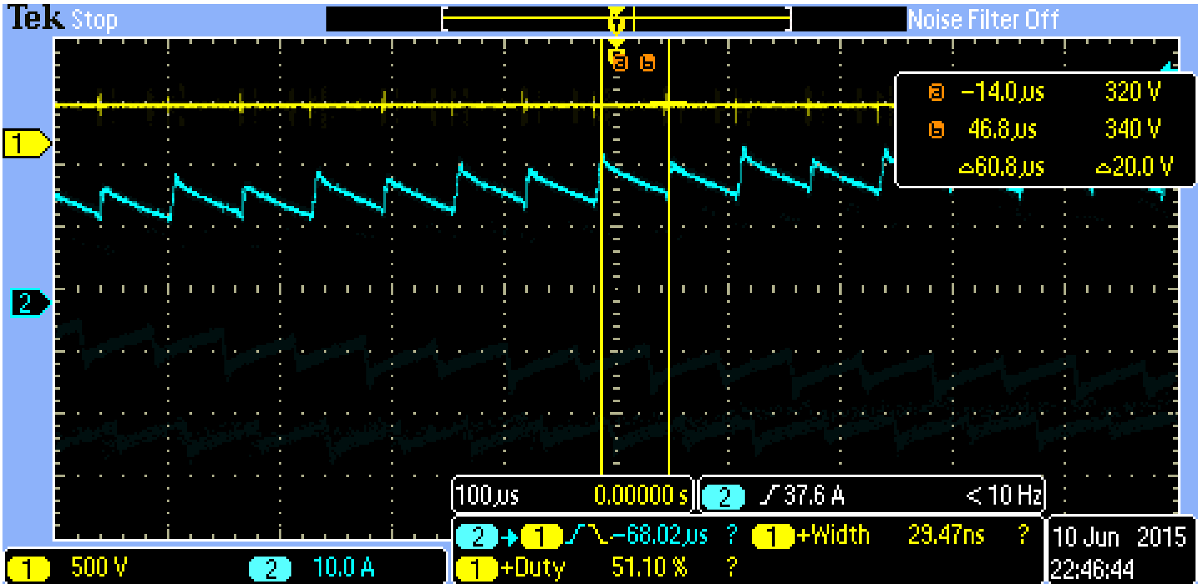

Figure 9.15: Current ripples from control loop frequency.

Figure 9.16 shows the test setup with PM motor, load, battery accumulator and final VSI design. This was the last test setup before the system was installed in the car.

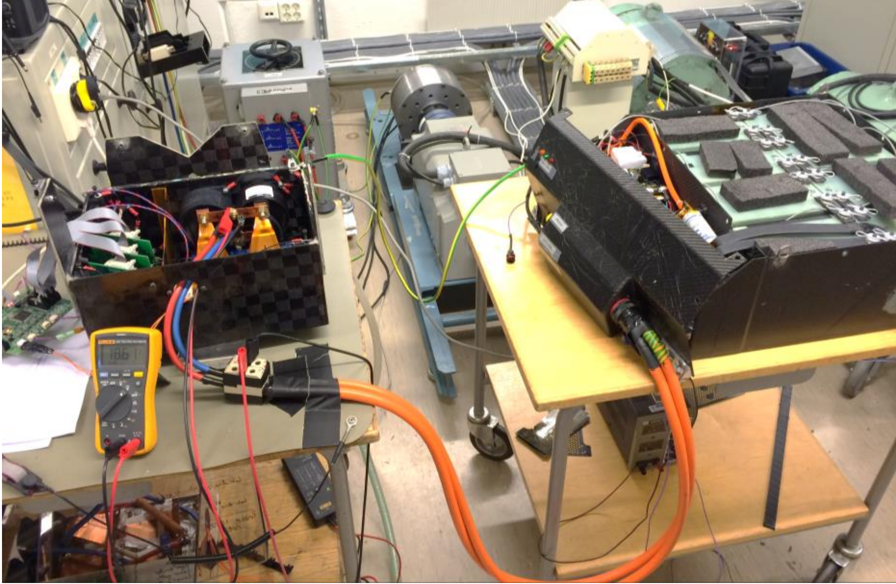

Figure 9.16: Test setup with battery accumulator and load.

9.4. Testing in the Car

The first tests of the drivetrain after installing it in the car, was done with the vehicle on rig, without wheels, like shown in figure 9.18. In figure 9.17, one of the phase current measurements for this test setup is shown.

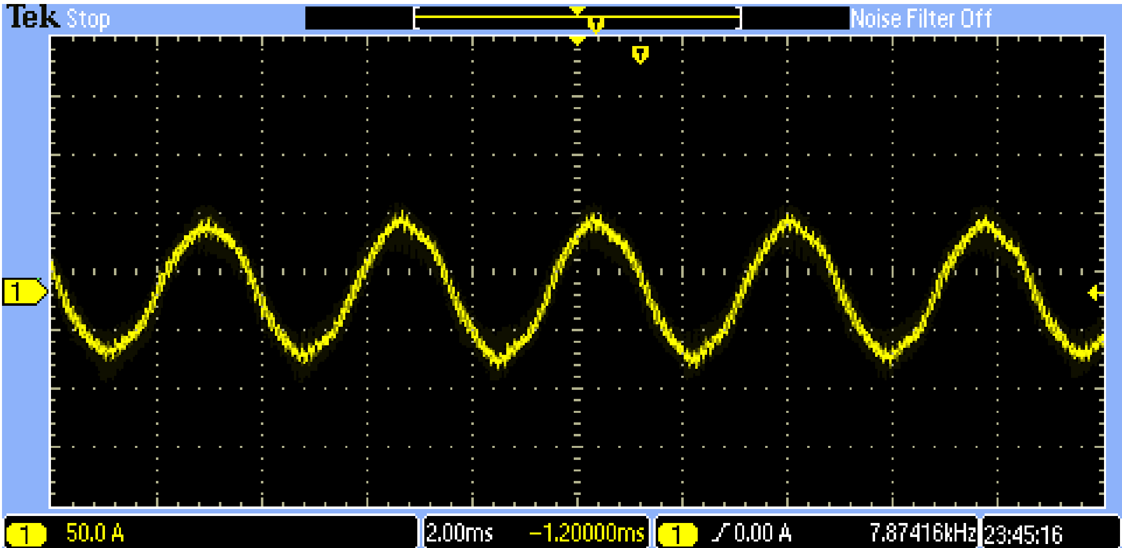

Figure 9.17: Phase current during testing of the car on rig with no load.

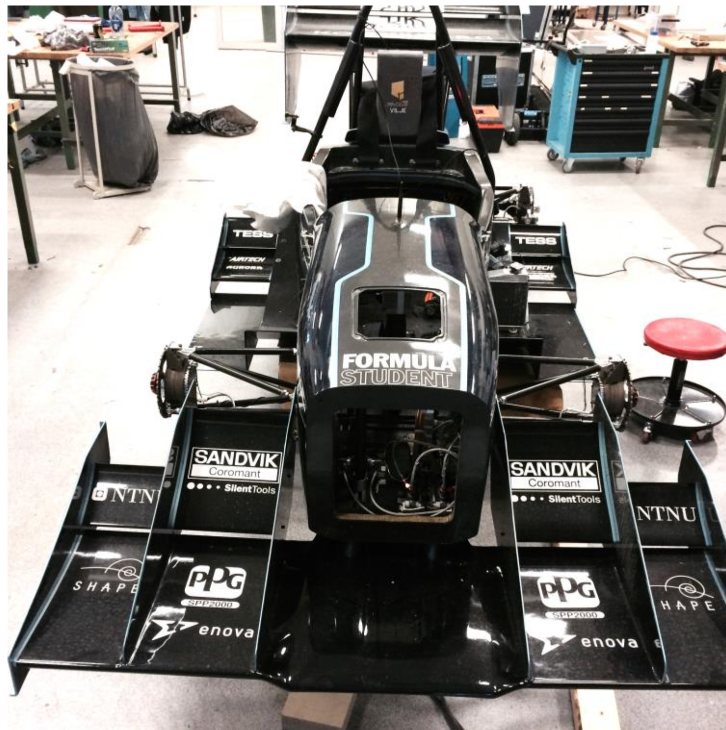

Figure 9.18: Initial testing of the car on rig.

Figure 9.19: Initial testing of the car at parking lot at NTNU.

After obtaining satisfactory operation of the vehicle drivetrain on rig, the car was brought out for testing on a parking lot at NTNU. In this test environment, oscilloscope and current probes

could not be used. For analysis and tuning, the in-house developed software Revolve Analyze, was applied. Revolve Analyze uses the telemetry system in the vehicle to analyze and represent data from the CAN busses in the car. In figure 9.20, the yellow line is the d current, the purple line the q current, and the green line the torque request from the driver. By using this information, the VSI could be tuned to follow the set points.

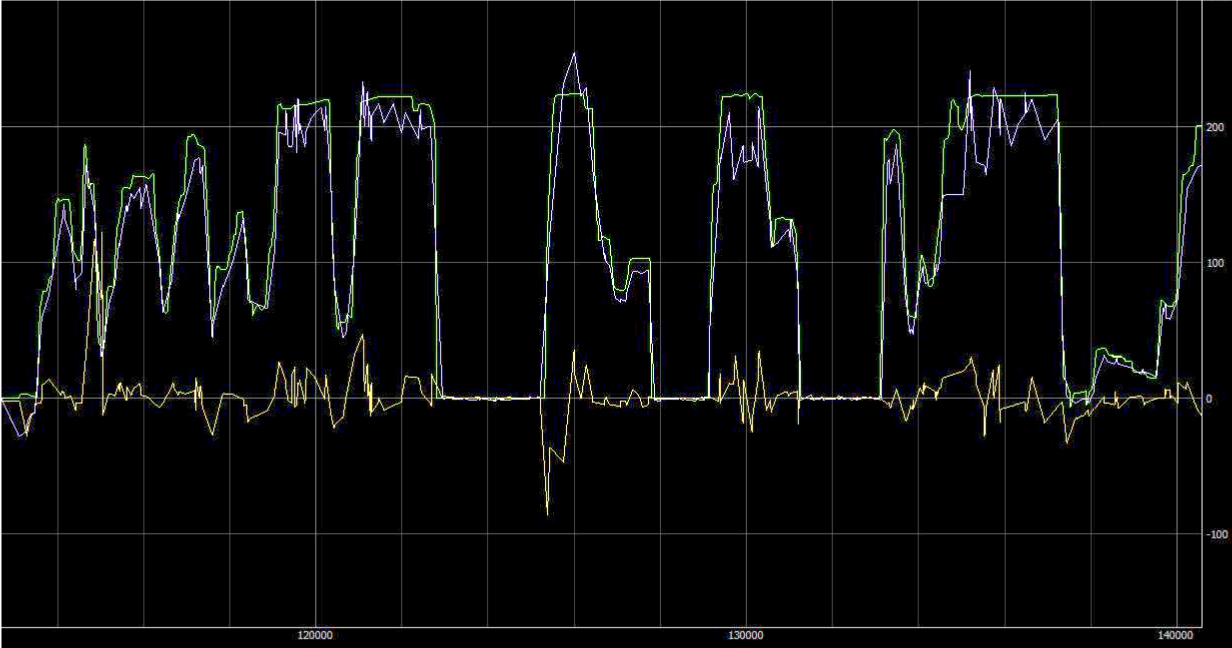

Figure 9.20: Revolve Analyze, dq currents and torque request.

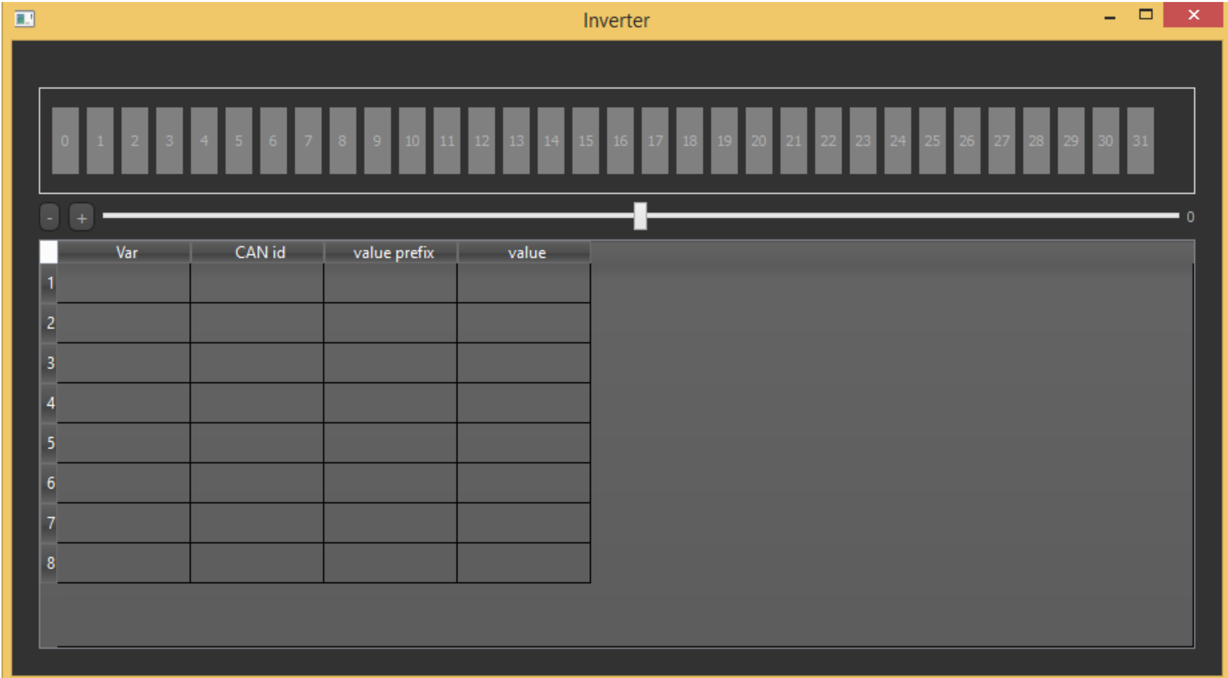

Figure 9.21: Revolve Analyze, inverter plugin.

In figure 9.21, the inverter plugin in Revolve Analyze is shown. As was discussed in chapter 6, this plugin system has been used to control the VSI during laboratory testing. The 32 status bits show the errors and the system status, according to table 6.2. During the laboratory testing, the sliding bar below the status bits was used to control the torque request to the VSI. The table below the sliding bar was used to send new parameters to the motor control system. It was for instance used to feed the control system with new control parameters for the PI controllers and to try different offset values for the encoder.

The final testing before the Formula Student competition at Silverstone, was done at the race circuit at Lånke at Stjørdal. This is a larger track, where the maximum speed and power limitations of the drivetrain can be tested properly.

The final test of the voltage source inverter design, was the Formula Student competition at Silverstone. The only problem with the motor control system during the competition, was that the power limiting function in the ECU was not set up right. In the acceleration event, this resulted in a power overshoot, with a moving average of 84 kW over 500 ms. Because the limit is 80 kW, the team was disqualified. In the endurance event, the car managed to get one of the fastest lap times in the competition! Unfortunately, the car was not able to finish the race due to a fault in the brake system. Apart from this, the rest of the events went well, and the team managed to get a third place in the design event.

10. Discussion

The testing of the voltage source inverter in the car has given good results. After changing the current sensors, no problems have been encountered at the required operating conditions. The system seems robust and the cooling effective. For higher performance and lower weight, the margins for the system need to be pushed further or different solutions applied.

By designing gate driver in-house, the three driver circuit boards in this design could be replaced with one, saving weight and space. The board could then be custom fit to the applied IGBT modules and casing geometry. The functionalities on the board can then be decided by the designer, giving more possibilities and flexibility. Along with in-house designed gate driver, the possibilities to use different semiconductor transistor technologies or designs, may give a boost in performance. For example, by using SiC MOSFETs the switching frequency can be increased significantly. With higher switching frequency the motor losses associated with the ripples as in figure 9.14, will be reduced. These rippels would also have been lower if a motor with higher stator inductance had been used. Because of the outer rotor, the Emrax 228 has room for many pole pairs, but not that much room for the stator coil conductors. The motor is designed for lower rotational speeds and higher currents, not to be simple to control.

For faster switching, the losses are shifted from the motor to the VSI, as the current ripples are smoothed. With the capacity of the present heatsink, shifting some of the losses to the VSI, could be overall beneficial. With the present MCU, this is not possible. With the applied control system, the maximum control loop frequency is 16 kHz and maximum switching frequency 8 kHz. To be able to switch faster, a new MCU or a SoC solution with FPGA is required. With more processing power, the possibility to use more complex operations and systems are also made possible. As discussed in chapter 2, SV PWM utilizes more of the DC bus voltage. It would also be possible to implement advanced filtering or compensation for the flux estimation in DTC.

A way to reduce the power dissipation in the VSI, even at high switching frequencies, is to optimize the gate resistor. The smaller the resistor value, the faster the capacitances in the IGBT will be charged and discharged. The SKM300GB12V IGBT module used in the Revolve design, actually features internal gate resistance. In the laboratory test setup, because of this, the IGBT module were able to operate with a gate resistance equal to zero. At this point the waveforms were starting to oscillate, meaning that the operating point was close to its limit.

If the optimal power dissipation distribution between VSI and motor is found, weight can be saved on the whole cooling system. Weight can also be saved on the sensors. Instead of using current transducers, it is also possible to use simpler, more lightweight sensors. If DTC is applied, it is also possible to skip the encoder and run sensorless drive. A way to simulate this, is to include it in the Simulink model for the system. By looking at different operating conditions, the ripples and the power dissipation can be roughly estimated. The best way to find out, would be in a test setup with temperature measurements.

Weight and performance are two of the most important parameters in Formula Student. At the same time it is also important to keep safety margins and to not take unnecessary risks in the design. By keeping things simple, the development time is decreased, giving more time for testing.

11. Conclusion

Compared to the voltage source inverter from last season, this design ended up more than 4 kg lighter, going from 8.5 kg to 4.4 kg. This is the largest weight reduction of any system in the vehicle compared to last year. Most of the reduction has come from using a carbon fiber casing, lightweight input capacitors, and an in-house designed heatsink.

The VSI casing fits well into the monocoque and motor room. Because of this, the length of cables and cooling circuits could be shortened. All the contacts are easy to reach from the rear service hatch or from the cockpit. The control circuit board can easily be reprogrammed from the service hatch, without needing to disassemble the casing. The system is well integrated into the vehicle and the communication network, giving direct information to the dashboard and two-ways communication over the vehicle's telemetry system. The telemetry system can easily be used for tuning the controller, making testing and development easier.

Figure 11.1: Final rendering of Vilje.

Based on extensive testing in a laboratory environment and actual driving of the car on track, the VSI design has been verified. The applied control system, field oriented control, works as expected. Same applies for the sensor devices, IGBT modules, discharge circuit and the other hardware components. The control circuit board and program code operate with high performance, while the safety functions contribute to the driver's safety. Through the design

process, different technologies and solutions have been investigated. This have been done through literature, simulation and laboratory testing. To model and design the system, several software tools have been applied. For the control system modelling, a Simulink model was developed. For the mechanical design and electrical design, 3D modelling tools were applied. These tools and models helped making design choices and were of great assistance in the manufacturing process.

A large part of the work have been laboratory testing and setup. Because of all the test and development work, few errors and faults have occurred in the later testing period. The VSI system performed well at the Formula Student competition at Silverstone and no design errors or flaws have been discovered.

12. Further Work

As discussed in chapter 10, there are ways of improving the system performance and lowering its weight. To be able to speed up the control loop and the switching, other alternatives for processing units should be investigated. Then, space vector modulation, direct torque control and flux estimation become more interesting concepts. For weight reduction, in-house designed gate driver or a more lightweight heatsink design may work.

Next season, 4WD will probably be one of the big concepts. 4WD with four individual PM motors, require four individual VSIs for motor control. To make such a system compact and lightweight, there are several alternative approaches. The VSIs can either be distributed as close as possible to the respective motors. They can be centralized to one unit, sharing a common heatsink. Or the compromise solution, where they are divided into two units, one on each side of the battery accumulator.

In a 4WD system, the ECU becomes more complex. Instead of supplying torque request to one VSI, it now has to handle four individual VSIs. To obtain good cornering abilities and regenerative braking, an algorithm for torque vectoring must be developed. In a system like this, the actual speed of the vehicle compared to the ground becomes important measurement data. In a rear wheel driven car, the front wheels can be used to get the actual speed, because the front wheels will never spin. With 4WD, on the other hand, an IMU or a laser is needed to find the true vehicle speed.

13. Bibliography

[1] L. H. Opsahl, "Design of Voltage Source Inverter for Electric Vehicle", Norwegian University of Science and Technology, 2014.

[2] SAE International, "2015 Formula SAE Rules", Published online, September 2014.

[3] E. Børsheim, M. S. Skåravik, "Energy storage, and design of tractive system for EV", Master's thesis NTNU, June 2014.

[4] Unitek Industrie Elektronik GmbH, "Bamocar d3 Manual", Nellmersback, 2012.

[5] N. Mohan, T. Undeland, W. Robbins, "Power Electronics – converters, applications and design", John Wileys & Sons inc., 2003.

[6] N. Mohan, "Advanced Electric Drives: Analysis, Control, and Modeling Using MATLAB/Simulink", Published online, 2014.

[7] P. Krause, O. Wasynczuk, S. Sudhoff, S. Pekarek, "Analysis of Electric Machinery and Drive Systems, Third Edition", Published online, 2013.

[8] ABB Drives, "Technical guide No. 1 Direct torque control – the world's most advanced AC drive technology", Rev. C, Published online 2011.

[9] M. F. Rahman, E. Haque, L. Tang, L. Zhong, "Problems associated with the direct torque control of an interior permanent-magnet synchronous motor drive and their remedies", IEEE Transaction on industrial electronics, vol. 51, pp 799-809, August 2004.

[10] L. Tang, M. F. Rahman, Y. Hu, "A novel direct torque control for interior permanent-magnet synchronous machine drive with low ripple in torque and flux – a speed-sensorless approach", IEEE Transaction on industry applications, vol. 39, pp. 1748-1756, November/December 2003.

[11] G. Buja, M Kazmierkowski, "Direct Torque Control of PWM Inverter-Fed AC Motors – A Survey", IEEE Transaction on industrial electronics, vol. 51, pp 744-757, August 2004.

[12] M. F. Rahman, L. Zhong, K. W. Lim, "A direct torque-controlled interior permanent magnet synchronous motor drive incorporating field weakening", IEEE Transaction on industry applications, vol. 34, pp. 1246-1253, November/December 1998.

[13] A. Wintrich, U. Nicolai, W. Tursky, T. Reiman, "Application Manual Power Semiconductors", Semikron International GmbH, Nuremberg 2011.

[14] R. Togashi, Y. Inoue, S. Morimoto, M. Sanda, "Performance improvement of ultra-high-speed PMSM drive system based on DTC by using SiC Inverter", International power electronics conference, pp. 356-362, May 2014.

[15] T. Zhao, J. Wang, A. Q. Huang, "Comparison of SiC MOSFET and Si IGBT based motor drive system", IEEE Transaction on industry applications, pp. 331-335, September 2007.

[16] M. Salcone, J. Bond, “Selecting film bus link capacitors for high performance inverter applications”, IEEE International electric machines and drives conference, pp. 1693-1699, May 2009.

[17] H. Kolstad, K. Ljøkelsøy, “20 kW IGBT omformer. Beskrivelse, 3. utgave”, SINTEF Energiforskning AS, September 2002.

[18] “Design Considerations for Designing with Cree SiC Modules Part 2. Techniques for Minimizing Parasitic Inductance”, Cree Inc., Durham 2013.

[19] M. Felden, P. Bütterling, P Jeck, L. Eckstein, K. Hameyer, “Electrical vehicle drive trains: From the specification sheet to the drive-train concept”, International power electronics and motion control conference, vol. 11, pp. 9-16, September 2010.

[20] “Lapp Cable Guide”, U. I . Lapp GmbH, Stuttgart 2011.

[21] Infineon, “AN2012-05 - 62mm Modues Application and Assembly Notes”, v2.0, published online February 2013.

[22] E. Simon, “Implementation of a Speed Field Oriented Control of 3-phase PMSM Motor using TMS320F240 – Application report SPRA588”, Texas Instruments, published online, September 1999.

[23] J. Catsoulis, “Designing Embedded Hardware”, O’Reilly, May 2005.

[24] B. Wu, “High-Power Converters and AC Drives”, IEEE Press, John Wiley & Sons inc., 2006.

[25] A. Emadi, Y. J. Lee, K. Rajashekara, “Power Electtronics and Motor Drives in Electric, Hybrid Electric, and Plug-In Hybrid Electric Vehicles”, IEEE Transaction on industrial electronics, vol. 55, pp 2237-2245, June 2008.

[26] Hydro-Quebec, TransEnergie Technologies inc., “SimPowerSystems User’s Guide ”, Revised version 3, published online, September 2003.

[27] LEM International, “Industry Current & Voltage Transducers – Industry Catalogue”, published online, April 2008.

[28] Atmel, “SAM4E Series datasheet”, published online, April 2015.

A. Problems and Issues During Testing

Control Circuit Board Fix

As can be seen in figure A.1, in the first design iteration of the control circuit board, the PWM pins on the MCU was not connected. At first the plan was to run the control system with hysteresis control or with software PWM, not using the PWM modules in the MC. This plan was later disregarded and a temporary fix was made before the next production run.

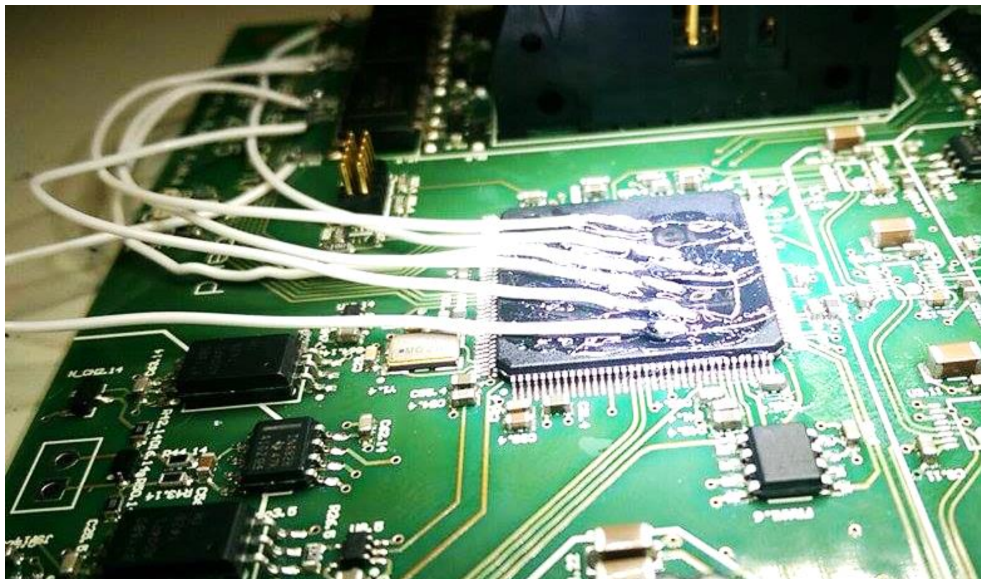

Figure A.1: Control circuit board prototype fix to include PWM.

Encoder Interface and Rating

At first, an absolute binary SSI encoder was used. This was an encoder with a resolution of 8192 ppr and only rated for a maximum rotational speed of 2500 rpm. Because the Emrax 228 has a maximum rotational speed of 6000 rpm, this encoder was more or less useless. The high resolution was also more than required, and took up to more processing power than needed. Instead of using this, the inverter control circuit board was redesigned to interface with an incremental ABZ encoder with a resolution of 2048 cpr and rated for a maximum rotational speed of 10000 rpm. Because of the lower resolution and because the internal decoder in the microcontroller could be used, this encoder required less computational power. This was partly due to lower resolution, but also because the synchronizing clock required for the SSI, is not required with ABZ interface.

In the meantime, while waiting for the new encoder and for the new control circuit board, a temporary fix was made. This was an ABZ encoder borrowed from the university, mounted to the rotor of a synchronous motor, as shown in figure A.2. Because the present control board was designed for SSI interface, a MCU was set to translate the ABZ signals to SSI signals. This was not a very good solution, but kept things going for a while.

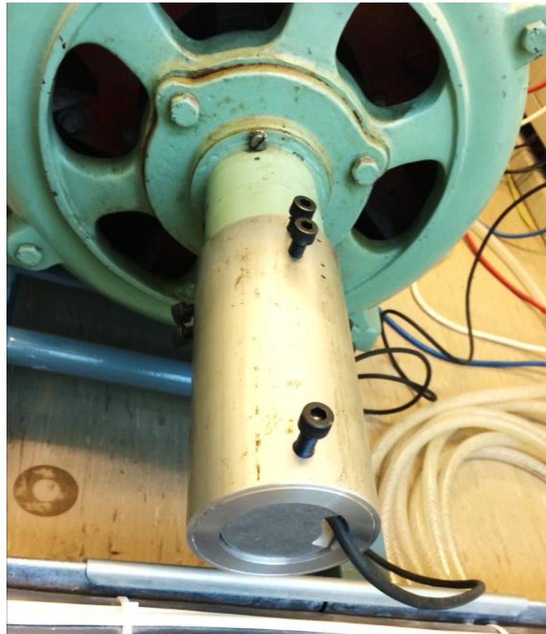

Figure A.2: ABZ encoder fix on synchronous motor.

Current Sensor Saturation in Test Setup

In figure A.3, the phase currents, DC current and DC voltage waveforms are captured on an oscilloscope during saturation of the current sensors. The yellow line (line 1) is the DC voltage (too high resolution to observe the waveform), the cyan (line 2) line the DC current, the pink (line 3) and green (line 4) lines the phase currents and the white lines the PWM signals (too high resolution to observe the signals). In this test, the water-cooling temperature was set as cold as possible and the VSI shut down shortly after the incident, resulting in no damaged equipment. As is clear from the oscilloscope view, the motor control loses control over the phase currents and large spikes appear. It can also be seen that the DC current is varying a lot, in both positive and negative directions. During the negative DC current spikes, the battery accumulator is charging. High charging currents like this might damage the battery cells. Luckily no damage was done to the battery cells this time.

The current sensor saturation occurred at high phase currents, not at normal operating conditions. The problem still had to be fixed to prevent it from happening under more extreme operating conditions in the car. The solution was to change current sensors to sensors with higher current measuring range.

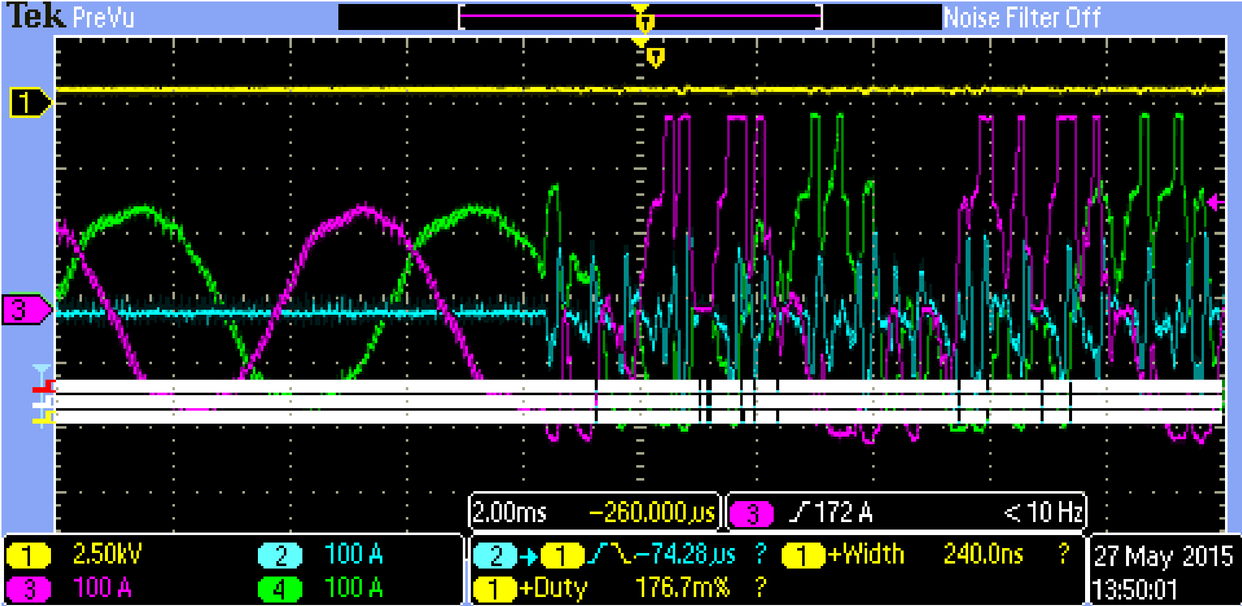

Figure A.3: Current and voltage waveforms during current sensor saturation.

With new current sensors and by increasing the transfer rate of the ADCs in the current sensor circuit, the problem was solved.

PM motor Problems and Solutions

During testing and assembly, two major issues occurred with the PM motor. The first issue was that one of the cable lugs for one of the stator phases came loose during assembly. The other issue was that the permanent-magnets got displaced during testing in the car.

As can be seen in figure A.4, the lowest cable lug is missing. It came loose during assembly of the motor in the test rig. The fix for this, was to use a powerful soldering iron and try to reconnect the lug with enough soldering tin. It was difficult to get enough heat and the quality of the soldering is unknown. Regardless of this, the result was good enough as long as the cable lugs were treated with care.

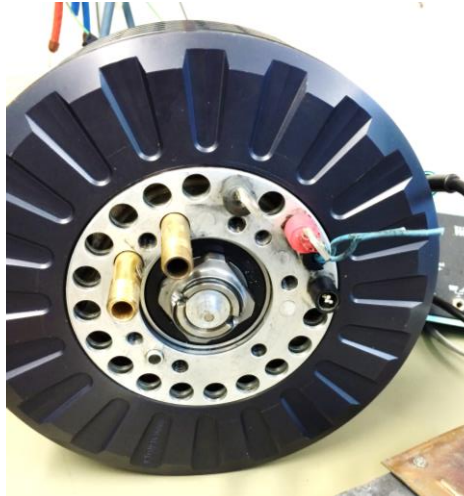

Figure A.4: Loose cable lug on PM motor phase.

The more critical issue was the permanent-magnet displacement. The problem was discovered during one of the last test runs at Lånke before the Formula Student competition at Silverstone. At the time, the reason for the problem was unknown. The symptoms were reduced output torque and unusual heat production in the motor.

Figure A.5: PM motor assembly at Silverstone.

The solution for the problem was to send the motor back to the manufacturer in Slovenia for service. Because this was one of the last days of testing before departure for UK, it was no time for doing this. The car was therefore sent to the UK without motor, while one of the team members travelled to Slovenia with the motor in his luggage. In Slovenia, the problem was

quickly identified at the manufacturer workshop. Due to heat, the glue, holding the permanent-magnets in place, had loosened. This was considered a warranty issue, so a new motor was given free of charge. And luckily the next flight from Ljubljana to London was just in time to reach Silverstone before the car and the rest of the team arrived. Figure A.5 shows the assembly of the new motor, in its motor housing, only hours before moving the car into the pits at Silverstone race circuit.

IGBT and Heatsink Failure

During testing in the car, problems with overheating and overcurrents in the IGBT modules occured a couple of times. The lower IGBT module in figure A.6 was destoyed due to overcurrents during a test run. In this test run, the safety function for overcurrents had been pushed to the limit, and the incident happened before the new current sensors (see chapter above) with higher measuring range had been installed. The high currents lead to current sensor saturation and loss of control.

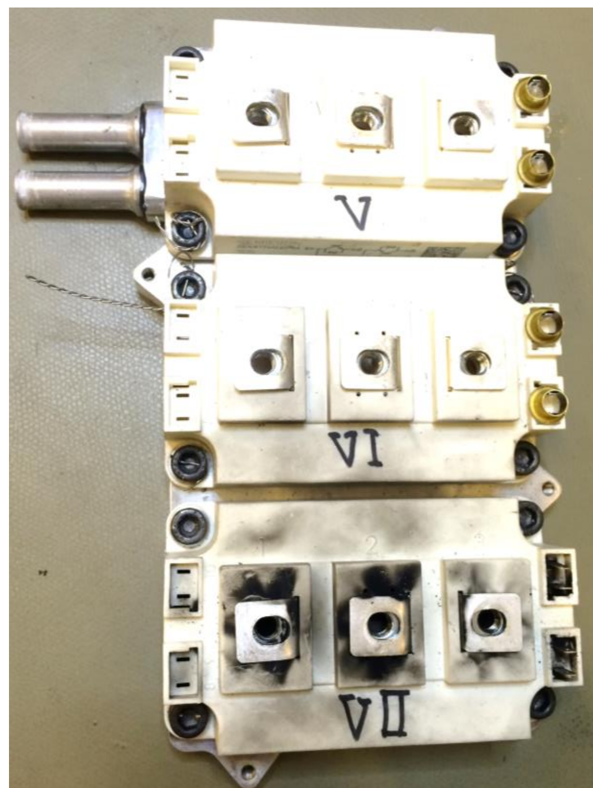

Figure A.6: IGBT and heatsink failure.

The lower IGBT module in figure A.6 was destroyed, while the two other modules survived. The symptoms were an unpleasant sound coming from the motor, together with significantly

reduced output torque. The VSI could still make the motor spin slowly with two phases, but not well enough to drive. The destroyed IGBT module was cut open for internal inspection, as shown in figure A.7. As can be seen, the IGBTs and terminals are totally burned out.

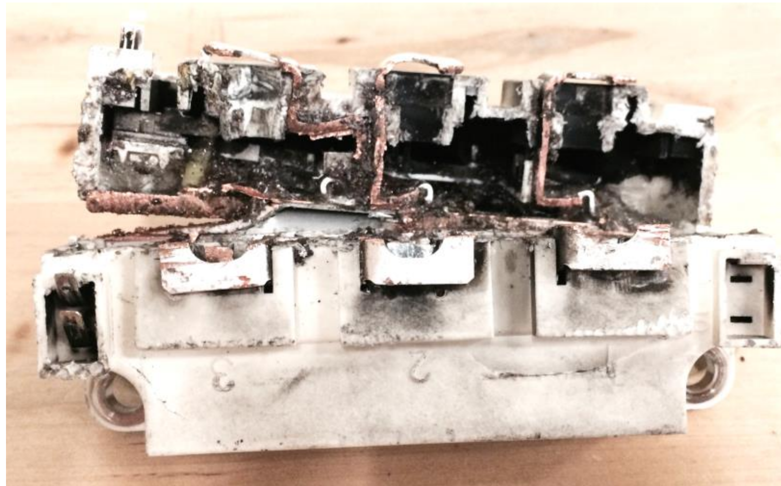

Figure A.7: Inside of failed IGBT.

The main issues with overheating during testing have been the test responsible forgetting to turn on the radiator pump and fan. One time, this resulted in the glue holding together the heatsink baseplate coming loose. After this, a new baseplate was made. This time, attached to the heatsink with both glue and bolts.

It is also worth mentioning that the gate-oxide in the IGBTs is very sensitive to electrostatic discharge (ESD). Two of the IGBT modules were destroyed due to ESD during assembly. To avoid this, the gate-emitter contacts should have been short circuited and the modules be carried in ESD approved containers or be connected to the gate drivers.

Control Circuit Board Issues

During the test period, a number of issues on the control circuit board has occurred. Most of the problems have been manufacturing errors or short circuits in power supply circuits. The manufacturing errors have normally been that a wrong resistor or a wrong capacitor has been picked during the circuit board assembly. When a wrong component is used in a power supply circuit, the components might get overheated and destroyed. This has happened a number of times and resulted in the death of a couple of control circuit boards. The issue with short circuits has normally been a result of measuring probes shorting two pads on the circuit board. This has

happened a couple of times when trying to measure the current sensor signals. The measurement probe has then short circuited the measurement pin to the negative 15 V supply. The negative 15 V supply circuit has then been overloaded and destroyed.

The best solution for this problem would be to design better short circuit protection for the power supply circuits. This is too late to do at this stage, but should be done in the next design iteration.

B. Program Code for Motor Control

The motor control code is written in C, using the version control system GitHub. The main file is inverter_control.c. This file is calling functions from other files and folders.

In the folder InverterDrivers, there are files with functions and code for blinking LEDs, PWM, individual IGBT control, SPI and current sensing and encoder reading.

In the folder Modules, there are files with functions and code for ADC, error handling, initialization of systems, and the field oriented control calculation function.

In the folder Settings, header files with parameters and error states are found.

The code in these folders is developed specifically for the motor control system. The rest of the folders and files used in the system are developed as standardized code for all MCU-based systems in the vehicle. This code is found in the folder RevolveDrivers.

The file revolve_can_definitions.h shows all the CAN messages and ID numbers for the two CAN busses in the vehicle.

B.1: inverter_control.c

```
#include <stdbool.h>
#include <sam.h>
#include <stdlib.h>

#include "fpu.h"

#include "InverterDrivers/tc.h"
#include "InverterDrivers/LED.h"
#include "sam4e-base/RevolveDrivers/pmc.h"
#include "sam4e-base/RevolveDrivers/eefc.h"
#include "sam4e-base/RevolveDrivers/can.h"
#include "sam4e-base/RevolveDrivers/afec.h"
#include "InverterDrivers/abz.h"
#include "InverterDrivers/pwm.h"
#include "InverterDrivers/CurrentSensor.h"
#include "InverterDrivers/Switch.h"
#include "InverterDrivers/spi.h"

#include "sam4e-base/RevolveDrivers/pio.h"

#include "Modules/angleConversions.h"
#include "Modules/adcConversion.h"
#include "Modules/error.h"

#include "Settings/ControlSettings.h"
#include "Settings/boardHardware.h"
#include "Settings/errorCodes.h"
#include "Settings/motorConstants.h"

#include "Modules/foc.h"
#include "Modules/commutation.h"
#include "Modules/voltageModulation.h"

#include "can-definitions/revolve_can_definitions.h"

#include "sam4e-base/util/delay.h"

static const uint32_t CALIBRATION_TIME_MS = 5000;

static struct Tc_abz_param encoder_settings = {
		.enabled = true,
		.measure_position = true,
		//.auto_correct = true,
		//.swap_inputs = true
		.count_phB = true,
		//.invert_phZ = true,
		.filter_max_ticks =  10
};

void inverter_init()
{
		WDT->WDT_MR &= ~(1<<13);
		init_flash();
		fpu_enable();

		struct PmcInit pmcInit = {
				.freq = INTERNAL_12MHZ,
				.css = PLLA_CLOCK,
				.pres = CLK_1,
				.divide = 1,
				.multiply = 9
		};

		pmc_init(pmcInit);

		led_init();

		struct TcInit tcInit = {
				.tcChannel = (&(TC1->TC_CHANNEL[0])),
				.timerClockSource = TIMER_CLOCK1,
				.timerOperatingMode = WAVEFORM_MODE,
				.rcCompareStopClock = false,
				.rcCompareDisableClock = false,
				.timerWaveformSelection = UP_RC,
		};

		// Initalization of controllers

		commutation_init(KP_BASE, KI_BASE, KD_BASE);
		foc_init(KP_BASE, KI_BASE, KD_BASE);

		// CAN

		uint32_t acceptence_mask[7] = {0};
		uint32_t id_masks[7] = {0};


		can_init(CAN0,
					120000000,
					CAN_BPS_1000K,
					acceptence_mask,
					id_masks,
					NULL);

		// Current sensing initialization
```

```
		spi_init(120000000, 40000000);
		current_init();

		// Enable encoder

		tc_abz_init(&encoder_settings, 0, TC0);

		// Status/error reigstry

		error_init(TRANSIENT_ERRORS|STATUS_BITS);
		error_signal(ENCODER_ZERO_NOT_PASSED);

		// PWM initialization

		pwm_init(120000000, PWM_FREQUENCY, DEAD_TIME_FREQUENCY);
		switch_init();

		// Initalization of afec
		afec_init(HV_VOLTAGE_MEASURE_ADC_INSTANCE, CPU_FREQ, (1<<HV_VOLTAGE_MEASURE_ADC_NUM));


}

int main()
{
		delay_init(CPU_FREQ);
		delay_ms(50);
		inverter_init();

		while(1)
		{
		// Variables needed to run the control loop, made static for performance
			static float currentA, currentB, currentC;
			static float torqueRequest;
			static float previousCurrentA, previousCurrentB, previousCurrentC;
			static float eletricalDegree, eletricalDegreeCompensated;
			static int32_t encoderValue, lastEncoderValue, compensatedEncoderValue, numRevolutions;
			static bool autoCalibrate = false;
			static uint16_t encoderOffset = ENCODER_OFFSET;
			static float dcVoltageFiltered;
			static float violationCurrentA, violationCurrentB, violationCurrentC;
			static float rpm;
			static float setpointQ, setpointD;
			static float KP = KP_BASE;
			static float KI = KI_BASE;
			static float KD = KD_BASE;
			static float currentD, currentQ;

			// Wait for PWM duty cycle

			while(!(PWM->PWM_ISR1 & PWM_ISR1_CHID0)); // running control loop synchronous with PWM

			led_enable(3);

			// Start current conversion
			current_startConversion(1);
			current_startConversion(4);

			// Read events from CAN-bus
			struct CanMessage receiveMessage;

			can_popMessage(CAN0, &receiveMessage);
			switch(receiveMessage.messageID){
				case TELEMETRY_TO_INVERTER_DATA_ID:
					switch(receiveMessage.data.u8[0]){
						case TORQUE_REQUEST_COMMAND:
							if(error_check(ACCEPT_TORQUE_REQUEST_FROM_TELEMETRY_NOT_ECU))
								torqueRequest = *( (float *) ( receiveMessage.data.u8 + 1 ) );
						break;
						case STATUS_BIT_FLIP_REQUEST_COMMAND:
							if(error_check( (1<<receiveMessage.data.u8[1]) ))
								error_clear( (1<<receiveMessage.data.u8[1]) );
							else
								error_signal( (1<<receiveMessage.data.u8[1]) );
						break;
						case CONTROLLER_PARAMETER_P_SET_REQUEST_COMMAND:
							KP = *( (float *) ( receiveMessage.data.u8 + 1 ) );
							foc_init(KP, KI, KD);
						break;
						case CONTROLLER_PARAMETER_I_SET_REQUEST_COMMAND:
							KI = *( (float *) ( receiveMessage.data.u8 + 1 ) );
							foc_init(KP, KI, KD);
						break;
						case CONTROLLER_PARAMETER_D_SET_REQUEST_COMMAND:
							KD = *( (float *) ( receiveMessage.data.u8 + 1 ) );
							foc_init(KP, KI, KD);
						break;
					}
				break;
				case ECU_TORQUE_REQUEST_ID:
					if(!error_check(ACCEPT_TORQUE_REQUEST_FROM_TELEMETRY_NOT_ECU))
						torqueRequest = receiveMessage.data.f[0];
				break;
			}


			if(tc_abz_is_initialized())
				error_clear(ENCODER_ZERO_NOT_PASSED);
```

```
		// Calculate set points
		setpointQ = torqueRequest * (1.0 / MOTOR_TORQUE_PER_CURRENT);
		setpointD = 0;

		// Read encoder

		encoderValue = abz_getCount(TC0);
		if( (encoderValue < 512) && (lastEncoderValue > 1536) )
			numRevolutions++;

		if( (encoderValue > -512) && (lastEncoderValue < -1536) )
			numRevolutions--;

		compensatedEncoderValue = encoderValue - encoderOffset;

//		if(compensatedEncoderValue < 0)
//			compensatedEncoderValue += 2048;

		rpm = angleConversions_encoderToRPM(encoderValue, numRevolutions, (PWM_FREQUENCY*2) );

		eletricalDegree = angleConversions_encoderToEletricalDegrees(11, compensatedEncoderValue, 10);



		// Check voltage

		uint16_t adcVoltageReading = afec_read_last_conversion(HV_VOLTAGE_MEASURE_ADC_INSTANCE, HV_VOLTAGE_MEASURE_ADC_NUM);

		float dcVoltage = adcConversion_linearConversion(adcVoltageReading,

		HV_VOLTAGE_MEASURE_ADC_OFFSET,

		HV_VOLTAGE_MEASURE_ADC_VALUE_AT_MAX,

		HV_VOLTAGE_MEASURE_VOLTAGE_MIN,

		HV_VOLTAGE_MEASURE_VOLTAGE_MAX);

		dcVoltageFiltered = 0.95 * dcVoltageFiltered + 0.05 * dcVoltage;

		// Wait untill conversion is done

		while(pio_readPin(CURRENT_1_INT_PIO, CURRENT_1_INT_PIN));
		while(pio_readPin(CURRENT_4_INT_PIO, CURRENT_4_INT_PIN));

		// Set low conversion pin

		pio_setOutput(CURRENT_1_CONV_PIO, CURRENT_1_CONV_PIN, LOW);
		pio_setOutput(CURRENT_4_CONV_PIO, CURRENT_4_CONV_PIN, LOW);

		// Read conversion over SPI bus

		int16_t adcValueCurrentA = (0xffff)&spi_tranceive(0x0000, CURRENT_A_SPI_CS);
		int16_t adcValueCurrentB = (0xffff)&spi_tranceive(0x0000, CURRENT_B_SPI_CS);

		currentA = current_calculateCurrent(adcValueCurrentA, 1);
		currentB = current_calculateCurrent(adcValueCurrentB, 4);
		currentC = -currentB - currentA;

		// Blink LED
		static bool ledState = false;

		led_set(1, ledState);

		ledState = ! ledState;

		// Calculate FOC


		// Error/status checking

		static int errorCheckIterator = 4;
		const int NUMBER_OF_ERRORS_CHECKS = 11;
		uint8_t errorMask = switch_getError();
		static uint8_t errorCounter_phaseA = 0;
		static uint8_t errorCounter_phaseB = 0;
		static uint8_t errorCounter_phaseC = 0;
		static uint8_t errorCounter_voltage = 0;

		switch(errorCheckIterator){
			case 0:
				// Check for permanent current limits
				if( ( (currentA + previousCurrentA)/2.0 > CURRENT_PHASE_LIMIT_PERMANENT ) || ( ( -(currentA +
previousCurrentA)/2.0 < -CURRENT_PHASE_LIMIT_PERMANENT) ) ) {
					error_signal(CURRENT_LIMIT_EXCEEDED_PERMANENT);
					violationCurrentA = currentA;
				}
			break;
			case 1:
				if( ( (currentB + previousCurrentB)/2.0 > CURRENT_PHASE_LIMIT_PERMANENT ) || ( ( - (currentB +
previousCurrentB)/2.0 < -CURRENT_PHASE_LIMIT_PERMANENT) ) ) {
					error_signal(CURRENT_LIMIT_EXCEEDED_PERMANENT);
					violationCurrentB = currentB;
				}
			break;
			case 2:
				if( ( (currentC + previousCurrentC)/2.0 > CURRENT_PHASE_LIMIT_PERMANENT ) || ( ( - (currentC +
previousCurrentC)/2.0 < -CURRENT_PHASE_LIMIT_PERMANENT) ) ) {
```

```
                        error_signal(CURRENT_LIMIT_EXCEEDED_PERMANENT);
                        violationCurrentC = currentC;
                }
        break;
        case 3:
                // Check for transient current limits
                if( ( (currentA + previousCurrentA)/2.0 > CURRENT_PHASE_LIMIT_TRANSIENT ) || ( ( -(currentA +
previousCurrentA)/2.0 < -CURRENT_PHASE_LIMIT_TRANSIENT) ) ) {
                        errorCounter_phaseA++;
                        if(errorCounter_phaseA > MAX_TRANSIENT_CURRENT_ERRORS){
                                error_signal(CURRENT_LIMIT_EXCEEDED_PHASE_A_TRANSIENT);
                                violationCurrentA = currentA;
                                errorCounter_phaseA = 0;
                        }
                } else {
                        errorCounter_phaseA = 0;
                        error_clear(CURRENT_LIMIT_EXCEEDED_PHASE_A_TRANSIENT);
                }
        break;
        case 4:

                 if( ( (currentB + previousCurrentB)/2.0 > CURRENT_PHASE_LIMIT_TRANSIENT ) || ( ( - (currentB +
previousCurrentB)/2.0 < -CURRENT_PHASE_LIMIT_TRANSIENT) ) ) {

                        errorCounter_phaseB++;
                        if(errorCounter_phaseB > MAX_TRANSIENT_CURRENT_ERRORS){
                                error_signal(CURRENT_LIMIT_EXCEEDED_PHASE_B_TRANSIENT);
                                violationCurrentB = currentB;
                                errorCounter_phaseB = 0;
                        }
                } else {
                        errorCounter_phaseB = 0;
                        error_clear(CURRENT_LIMIT_EXCEEDED_PHASE_B_TRANSIENT);
                }
        break;
        case 5:

                if( ( (currentC + previousCurrentC)/2.0 > CURRENT_PHASE_LIMIT_TRANSIENT ) || ( ( - (currentC +
previousCurrentC)/2.0 < -CURRENT_PHASE_LIMIT_TRANSIENT) ) ) {
                        errorCounter_phaseC++;
                        if(errorCounter_phaseC > MAX_TRANSIENT_CURRENT_ERRORS){
                                error_signal(CURRENT_LIMIT_EXCEEDED_PHASE_C_TRANSIENT);
                                violationCurrentC = currentC;
                                errorCounter_phaseC = 0;
                        }
                } else {
                        errorCounter_phaseC = 0;
                        error_clear(CURRENT_LIMIT_EXCEEDED_PHASE_C_TRANSIENT);
                }
        break;
        case 6:
                // Check for permanent DC voltage limit
                if (dcVoltageFiltered > VOLTAGE_DC_MAX_LIMIT_PERMANENT)
                        error_signal(VOLTAGE_LIMIT_EXCEEDED_PERMANENT);

                // Check for transient DC voltage limit
                if (dcVoltageFiltered > VOLTAGE_DC_MAX_LIMIT_TRANSIENT)
                        errorCounter_voltage++;
                        if(errorCounter_voltage > MAX_TRANSIENT_CURRENT_ERRORS){
                                error_signal(VOLTAGE_LIMIT_EXCEEDED_TRANSIENT);
                                errorCounter_voltage = 0;
                        }
                else{
                        errorCounter_voltage = 0;
                        error_clear(VOLTAGE_LIMIT_EXCEEDED_TRANSIENT);
                }
        break;
        case 7:
                if(rpm > RPM_LIMIT_TRANSIENT)
                        error_signal(SPEED_LIMIT_EXCEEDED_TRANSIENT);
                else if (rpm < (RPM_LIMIT_TRANSIENT - RPM_HYSTERESIS))
                        error_clear(SPEED_LIMIT_EXCEEDED_TRANSIENT);
        case 8:
                if(errorMask != 0)
                        error_signal(GATE_DRIVER_ERROR);
                else
                        error_clear(GATE_DRIVER_ERROR);

                if(errorMask & (1<<0))
                        error_signal(GATE_DRIVER_ERROR_A);
                else
                        error_clear(GATE_DRIVER_ERROR_A);

                if(errorMask & (1<<0))
                        error_signal(GATE_DRIVER_ERROR_B);
                else
                        error_clear(GATE_DRIVER_ERROR_B);

                if(errorMask & (1<<0))
                        error_signal(GATE_DRIVER_ERROR_C);
                else
                        error_clear(GATE_DRIVER_ERROR_C);
        break;
        case 9:
                if(dcVoltageFiltered < VOLTAGE_DC_MIN_LIMIT_TRANSIENT)
                        error_signal(VOLTAGE_TO_LOW);
                else
                        error_clear(VOLTAGE_TO_LOW);
        break;
```

```
			case 10:
				if(fabs(torqueRequest) < TORQUE_REQUEST_LOWER_LIMIT_TRANSIENT)
					error_signal(TORQUE_REQUEST_TOO_LOW_TRANSIENT);
				else
					error_clear(TORQUE_REQUEST_TOO_LOW_TRANSIENT);

		}

		errorCheckIterator++;

		if (errorCheckIterator >= NUMBER_OF_ERRORS_CHECKS)
			errorCheckIterator = 0;

		static float voltageA;
		static float voltageB;
		static float voltageC;


		if ( error_check(TRANSIENT_ERRORS|PERMANENT_ERRORS|ENCODER_ZERO_NOT_PASSED|TORQUE_REQUEST_TOO_LOW_TRANSIENT) | (
!error_check(READY_TO_DRIVE))){
			foc_calculate(
				(previousCurrentA + currentA)/2,
				(previousCurrentB + currentB)/2,
				eletricalDegree,
				setpointQ,
				setpointD,
				&currentQ,
				&currentD,
				&voltageA,
				&voltageB,
				&voltageC,
				false
			);
			pwm_setDrivers(false);
			foc_antiWindup(true);
		} else {

			foc_calculate(
				(previousCurrentA + currentA)/2,
				(previousCurrentB + currentB)/2,
				eletricalDegree,
				setpointQ,
				setpointD,
				&currentQ,
				&currentD,
				&voltageA,
				&voltageB,
				&voltageC,
				true
			);

			// Calculate duty cycles

			bool saturationA, saturationB, saturationC;

			float dutyA = voltageModulation_voltageToPwm(voltageA, dcVoltageFiltered, 0.0, 1.0, WINDUP_ALLOWANCE, &saturationA);
			float dutyB = voltageModulation_voltageToPwm(voltageB, dcVoltageFiltered, 0.0, 1.0, WINDUP_ALLOWANCE, &saturationB);
			float dutyC = voltageModulation_voltageToPwm(voltageC, dcVoltageFiltered, 0.0, 1.0, WINDUP_ALLOWANCE, &saturationC);

			foc_antiWindup(saturationA|saturationB|saturationC);


			error_signal((saturationA << PID_SATURATION_A) | (saturationB << PID_SATURATION_B) | (saturationC << PID_SATURATION_C));
			error_clear((!saturationA << PID_SATURATION_A) | (!saturationB << PID_SATURATION_B) | (!saturationC << PID_SATURATION_C));

			pwm_setDrivers(true);


			// Apply voltage through PWM

			pwm_updateDutyCycles(dutyA, dutyB, dutyC);
		}

		if(error_check(PERMANENT_ERRORS|ENCODER_ZERO_NOT_PASSED|VOLTAGE_TO_LOW)||!error_check(READY_TO_DRIVE)){
			foc_reset_regulator();
		}

		// Send CAN messages with data

		uint32_t MESSAGE_SENDING_PERIOD = 30000;
		static uint32_t messageSendingCounter;
		int sendNum = messageSendingCounter % 300;

		struct CanMessage sendMessage;

		if (messageSendingCounter == 0)
		{
			sendMessage.dataLength = CANR_ALIVE_MSG_DLC;
			sendMessage.messageID = CAN_ALIVE_MSG_ID;
			sendMessage.data.u8[0] = ALIVE_INVERTER;
			can_sendMessage(CAN0, sendMessage);
		}

		switch(sendNum){
			case 10:
				sendMessage.dataLength = CAN_INVERTER_DATA_CURRENT_DLC;
				sendMessage.messageID = CAN_INVERTER_DATA_CURRENT_ID;
```

```
					sendMessage.data.f[0] = currentQ;
					sendMessage.data.f[1] = currentD;
					can_sendMessage(CAN0, sendMessage);
				break;
				case 20:
					sendMessage.dataLength = CAN_INVERTER_DATA_ENCODER_POS_DLC;
					sendMessage.messageID = CAN_INVERTER_DATA_ENCODER_POS_ID;
					sendMessage.data.i32[0] = encoderValue;
					sendMessage.data.i32[1] = numRevolutions;
					can_sendMessage(CAN0, sendMessage);
				break;
				case 30:
					sendMessage.dataLength = CAN_INVERTER_DATA_RPM_DLC;
					sendMessage.messageID = CAN_INVERTER_DATA_RPM_ID;
					sendMessage.data.f[0] = rpm;
					can_sendMessage(CAN0, sendMessage);
				break;
				case 40:
					sendMessage.dataLength = CAN_INVERTER_DATA_VOLTAGE_DLC;
					sendMessage.messageID = CAN_INVERTER_DATA_VOLTAGE_ID;
					sendMessage.data.f[0] = dcVoltageFiltered;
					can_sendMessage(CAN0, sendMessage);
				break;
				case 50:
					sendMessage.dataLength = CAN_INVERTER_DATA_STATUS_DLC;
					sendMessage.messageID = CAN_INVERTER_DATA_STATUS_ID;
					sendMessage.data.u32[0] = error_getVector();
					can_sendMessage(CAN0, sendMessage);
				break;
				case 60:
					sendMessage.dataLength = CAN_INVERTER_DATA_ACCUMULATED_STATUS_DLC;
					sendMessage.messageID = CAN_INVERTER_DATA_ACCUMULATED_STATUS_ID;
					sendMessage.data.u32[0] = error_getLastVector();
					can_sendMessage(CAN0, sendMessage);
				break;
				/*
				case 70:
					sendMessage.dataLength = 6;
					sendMessage.messageID = 106;
					sendMessage.data.i16[0] = ((int16_t)KP);
					sendMessage.data.i16[1] = ((int16_t)KI);
					sendMessage.data.i16[2] = ((int16_t)KD);
					can_sendMessage(CAN0, sendMessage);
				break;
				case 80:
					sendMessage.dataLength = 6;
					sendMessage.messageID = 107;
					sendMessage.data.i16[0] = ((int16_t)rpm);
					can_sendMessage(CAN0, sendMessage);
				break;
				case 90:
					sendMessage.dataLength = 2;
					sendMessage.messageID = 108;
					sendMessage.data.i16[0] = ((int16_t)encoderOffset);
					can_sendMessage(CAN0, sendMessage);
				break;
				case 100:
					sendMessage.dataLength = 4;
					sendMessage.messageID = 109;
					sendMessage.data.u32[0] = error_getLastVector();
					can_sendMessage(CAN0, sendMessage);
				break;
				case 110:
					sendMessage.dataLength = 4;
					sendMessage.messageID = 110;
					sendMessage.data.u32[0] = error_getCount();
					can_sendMessage(CAN0, sendMessage);
				break;
				case 120:
					sendMessage.dataLength = 8;
					sendMessage.messageID = 111;
					sendMessage.data.i32[0] = encoderValue;
					sendMessage.data.i32[1] = numRevolutions;
					can_sendMessage(CAN0, sendMessage);
				break;*/

			}

			messageSendingCounter++;

			if(messageSendingCounter >= MESSAGE_SENDING_PERIOD)
				messageSendingCounter = 0;

			previousCurrentA = currentA;
			previousCurrentB = currentB;
			previousCurrentC = currentC;

			lastEncoderValue = encoderValue;

			led_disable(3);

		}
}
```

B.2: fpu.h

```
#ifndef FPU_H_
#define FPU_H_

#include <stdbool.h>
#include <stdint.h>

/** Address for ARM CPACR */
#define ADDR_CPACR 0xE000ED88

/** CPACR Register */
#define REG_CPACR (*((volatile uint32_t *)ADDR_CPACR))

/** Enable FPU */
static void fpu_enable(void)
{
		REG_CPACR |= (0xFu << 20);
}

/** Disable FPU */
static void fpu_disable(void)
{
 REG_CPACR &= ~(0xFu << 20);
}

/** Check if FPU is enabled */
static bool fpu_is_enabled(void)
{
 return (REG_CPACR & (0xFu << 20));
}

#endif /* FPU_H_ */
```

B.3: InverterDrivers

tc.c

```
#include <sam.h>
#include <stdbool.h>

#include "../sam4e-base/RevolveDrivers/pmc.h"

#include "tc.h"

#ifndef TC_WPMR_WPKEY_PASSWD
#define TC_WPMR_WPKEY_PASSWD TC_WPMR_WPKEY((uint32_t)0x54494D)
#endif

void tc_enable_writeprotect(Tc * tc)
{
		tc->TC_WPMR = TC_WPMR_WPKEY_PASSWD | TC_WPMR_WPEN;
}

void tc_disable_writeprotect(Tc * tc)
{
		tc->TC_WPMR = TC_WPMR_WPKEY_PASSWD;
}

void tc_clockEnable(TcChannel * tcChannel)
{
		tcChannel->TC_CCR = TC_CCR_CLKEN;
}

void tc_clockDisable(TcChannel * tcChannel)
{
		tcChannel->TC_CCR = TC_CCR_CLKDIS;
}

void tc_resetAndStart(TcChannel * tcChannel)
{
		tcChannel->TC_CCR = TC_CCR_CLKEN|TC_CCR_SWTRG;
}

void tc_setMode(TcChannel * tcChannel, enum TimerOperatingMode timerOperatingMode, enum TimerClockSource timerClockSource, enum TimerWaveformSelection
timerWaveformSelection)
{
		uint32_t bitMask = 0;

		switch(timerOperatingMode){
				case WAVEFORM_MODE:
						bitMask += timerClockSource;
						bitMask += timerWaveformSelection;
						bitMask += TC_CMR_WAVE;
		}


		tcChannel->TC_CMR = bitMask;
}

void tc_init(struct TcInit tcInit)
{

		// Add switch case
		pmc_enable_periph_clk(TC0_IRQn);
		tc_disable_writeprotect(TC0);
		pmc_enable_periph_clk(TC3_IRQn);
		tc_disable_writeprotect(TC1);
		pmc_enable_periph_clk(TC6_IRQn);
		tc_disable_writeprotect(TC2);

		/*  Disable TC clock. */
		tc_clockDisable(tcInit.tcChannel);

		/*  Disable interrupts. */
		tcInit.tcChannel->TC_IDR = 0xFFFFFFFF;

		/*  Clear status register. */
		tcInit.tcChannel->TC_SR;

		/*  Set mode. */
		tc_setMode(tcInit.tcChannel, tcInit.timerOperatingMode, tcInit.timerClockSource, tcInit.timerWaveformSelection);

}

void tc_sync(Tc * tc)
{
		tc->TC_BCR = TC_BCR_SYNC;
}

uint32_t tc_readStatus(TcChannel * tcChannel)
{
		return tcChannel->TC_SR;
}

void tc_setRA(TcChannel * tcChannel, uint32_t value)
{
		tcChannel->TC_RA = value;
```

```
}

void tc_setRB(TcChannel * tcChannel, uint32_t value)
{
		tcChannel->TC_RB = value;
}

void tc_setRC(TcChannel * tcChannel, uint32_t value)
{
		tcChannel->TC_RC = value;
}
```

tc.h

```
#ifndef TC_H_
#define TC_H_

#include <sam.h>
#include <stdbool.h>

enum TimerClockSource {TIMER_CLOCK1 = TC_CMR_TCCLKS_TIMER_CLOCK1,
		TIMER_CLOCK2 = TC_CMR_TCCLKS_TIMER_CLOCK2,
		TIMER_CLOCK3 = TC_CMR_TCCLKS_TIMER_CLOCK3,
		TIMER_CLOCK4 = TC_CMR_TCCLKS_TIMER_CLOCK4,
		TIMER_CLOCK5 = TC_CMR_TCCLKS_TIMER_CLOCK5,
		XC0 = TC_CMR_TCCLKS_XC0,
		XC1 = TC_CMR_TCCLKS_XC1,
		XC2 = TC_CMR_TCCLKS_XC2
};
enum TimerOperatingMode {WAVEFORM_MODE = 1};
enum TimerWaveformSelection {
		UP = TC_CMR_WAVSEL_UP,
		UPDOWN = TC_CMR_WAVSEL_UPDOWN,
		UP_RC = TC_CMR_WAVSEL_UP_RC,
		UPDOWN_RC = TC_CMR_WAVSEL_UPDOWN_RC
};
//enum TimerRcCompareEffect {NONE = 0, SET = 1, CLEAR = 2, TOGGLE = 3};

struct TcInit {
		TcChannel * tcChannel;
		enum TimerClockSource timerClockSource;
		enum TimerOperatingMode timerOperatingMode;
		bool rcCompareStopClock;
		bool rcCompareDisableClock;
		enum TimerWaveformSelection timerWaveformSelection;
};


void tc_init(struct TcInit tcInit);
void tc_setRA(TcChannel * tcChannel, uint32_t value);
void tc_setRC(TcChannel * tcChannel, uint32_t value);
uint32_t tc_readStatus(TcChannel * tcChannel);
void tc_resetAndStart(TcChannel * tcChannel);
void tc_sync(Tc * tc);

void tc_enable_writeprotect(Tc * tc);
void tc_disable_writeprotect(Tc * tc);


#endif /* TC_H_INCLUDED */
```

LED.c

```
#include <stdbool.h>
#include "LED.h"
#include "../sam4e-base/RevolveDrivers/pio.h"


void led_init()
{
		pio_enableOutput(PIOLED1, LED1);
		pio_enableOutput(PIOLED2, LED2);
		pio_enableOutput(PIOLED3, LED3);
}

void led_enable(int ledNum)
{
		switch(ledNum){
				case 1:
						pio_setOutput(PIOLED1, LED1, HIGH);
						break;
				case 2:
						pio_setOutput(PIOLED2, LED2, HIGH);
						break;
				case 3:
```

```
			pio_setOutput(PIOLED3, LED3, HIGH);
			break;
	}
}

void led_disable(int ledNum)
{
	switch(ledNum){
		case 1:
			pio_setOutput(PIOLED1, LED1, LOW);
			break;
		case 2:
			pio_setOutput(PIOLED2, LED2, LOW);
			break;
		case 3:
			pio_setOutput(PIOLED3, LED3, LOW);
			break;
	}
}

void led_set(int ledNum, bool value){
	if(value){
		led_enable(ledNum);
	}
	else{
		led_disable(ledNum);
	}
}
```

LED.h

```
#ifndef LED_H_
#define LED_H_

#include "sam.h"

void led_init();
void led_enable(int ledNum);
void led_disable(int ledNum);
void led_set(int ledNum, bool value);

#define PIOLED1 PIOD
#define PIOLED2 PIOD
#define PIOLED3 PIOC

#define LED1 5
#define LED2 4
#define LED3 23


#endif /* LED_H_ */
```

abz.c

```
#include "tc.h"
#include "../sam4e-base/RevolveDrivers/pmc.h"
#include "../sam4e-base/RevolveDrivers/pio.h"
#include "abz.h"

const uint16_t encoder_max = 2048;
volatile bool  encoder_initialized = false;
volatile bool encoder_index_reached = false;
// Private tc functions:
void tc_enable_writeprotect(Tc * tc);
void tc_disable_writeprotect(Tc * tc);

void tc_abz_init(struct Tc_abz_param* settings, uint32_t timeBase, Tc *tc){

	if(tc == TC0){

			pio_setPull(PIOA, 0, NOPULL);
			pio_setPull(PIOA, 1, NOPULL);
			pio_setPull(PIOA, 16, NOPULL);
			pmc_enable_periph_clk(TC0_IRQn);
			pmc_enable_periph_clk(TC1_IRQn);
			pmc_enable_periph_clk(TC2_IRQn);
			pio_setMux(PIOA, 0 , B);
			pio_setMux(PIOA, 1 , B);
			pio_setMux(PIOA, 16, B);
			//TC0->TC_QIER |= TC_QIER_IDX;

			/*NVIC_ClearPendingIRQ(TC0_IRQn);
			NVIC_EnableIRQ(TC0_IRQn);*/
	}
	else if(tc == TC1){
```

```
//				pio_setMux(PIOA, 0 , B);
//				pio_setMux(PIOA, 1 , B);
//				pio_setMux(PIOA, 16, B);
				pmc_enable_periph_clk(TC3_IRQn);
				pmc_enable_periph_clk(TC4_IRQn);
				pmc_enable_periph_clk(TC5_IRQn);

				TC1->TC_QIER |= TC_QIER_IDX;

				NVIC_ClearPendingIRQ(TC1_IRQn);
				NVIC_EnableIRQ(TC1_IRQn);
		}
		else if(tc == TC2){
//				pio_setMux(PIOA, 0 , B);
//				pio_setMux(PIOA, 1 , B);
//				pio_setMux(PIOA, 16, B);
				pmc_enable_periph_clk(TC6_IRQn);
				pmc_enable_periph_clk(TC7_IRQn);
				pmc_enable_periph_clk(TC8_IRQn);

				TC2->TC_QIER |= TC_QIER_IDX;

				NVIC_ClearPendingIRQ(TC2_IRQn);
				NVIC_EnableIRQ(TC2_IRQn);
		}

		tc_disable_writeprotect(tc);

		tc->TC_BMR = 0x0000;

		tc->TC_CHANNEL[0].TC_CMR |= XC0;
		tc->TC_CHANNEL[1].TC_CMR |= XC0;
		tc->TC_CHANNEL[2].TC_CMR |= XC0;

		//tc_enable_writeprotect(tc);

		tc->TC_CHANNEL[0].TC_CMR &= ~TC_CMR_WAVE;
		tc->TC_CHANNEL[1].TC_CMR &= ~TC_CMR_WAVE;
		tc->TC_CHANNEL[2].TC_CMR |= TC_CMR_WAVE;
		uint32_t reg = 0;

		if(settings->enabled){
				reg |= TC_BMR_QDEN;
		}
		if(settings->no_dir_sense){
				reg |= TC_BMR_QDTRANS;
		}
		if(settings->count_phB){
				reg |= TC_BMR_EDGPHA;
		}
		if(settings->invert_phA){
				reg |= TC_BMR_INVA;
		}
		if(settings->invert_phB){
				reg |= TC_BMR_INVB;
		}
		if(settings->invert_phZ){
				reg |= TC_BMR_INVIDX;
		}
		if(settings->auto_correct){
				reg |= TC_BMR_AUTOC;
		}
		reg |= settings->filter_max_ticks << TC_BMR_MAXFILT_Pos;

		//tc_disable_writeprotect(tc);

		tc->TC_BMR |= reg;

		if(settings->measure_position){
				reg |= TC_BMR_POSEN;
				//Setup input capture
				tc->TC_CHANNEL[1].TC_CMR |= TC_CMR_ABETRG;
				tc->TC_CHANNEL[0].TC_CMR |= TC_CMR_ABETRG| TC_CMR_ETRGEDG_RISING;
		}
		if(settings->measure_speed){
				reg |= TC_BMR_SPEEDEN;
				//Setup Timebase
				tc->TC_CHANNEL[2].TC_CMR |= (TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC|TC_CMR_ACPC_TOGGLE);
				tc->TC_CHANNEL[2].TC_RC = timeBase;
				tc->TC_CHANNEL[0].TC_CMR |= TC_CMR_ABETRG;
				tc->TC_CHANNEL[2].TC_CMR |= TC_CMR_ETRGEDG_RISING|TC_CMR_LDRA_RISING|TC_CMR_LDRB_RISING;
				tc->TC_CHANNEL[2].TC_CCR |= TC_CCR_CLKEN|TC_CCR_SWTRG;
		}

		volatile uint32_t temp = tc->TC_QISR;

		tc->TC_BMR |= reg;

		tc_resetAndStart(&(tc->TC_CHANNEL[0]));
		tc_resetAndStart(&(tc->TC_CHANNEL[1]));
		tc_resetAndStart(&(tc->TC_CHANNEL[2]));

		tc_enable_writeprotect(tc);
}

enum Tc_abz_direction tc_abz_get_direction(Tc *tc){
		return !!(tc->TC_QISR & TC_QISR_DIR);
}
```

```
bool tc_abz_is_initialized(){
		return !!(TC0->TC_QISR & TC_QISR_IDX);
}


int16_t abz_getCount(Tc * tc)
{
		return  -tc->TC_CHANNEL[0].TC_CV;
}

void abz_getPosition(Tc * tc, int32_t * encoderPosPointer, int32_t * numRevolutionsPointer)
{
		int16_t firstRevolutionsSample = - tc->TC_CHANNEL[1].TC_CV;
		int16_t firstPosSample = - tc->TC_CHANNEL[0].TC_CV;
		int16_t secondRevoltionsSample = - tc->TC_CHANNEL[1].TC_CV;

		if(secondRevoltionsSample != firstRevolutionsSample){
			*encoderPosPointer = - tc->TC_CHANNEL[0].TC_CV;
			*numRevolutionsPointer = secondRevoltionsSample;
		} else if (secondRevoltionsSample == firstRevolutionsSample) {
			*encoderPosPointer = firstPosSample;
			*numRevolutionsPointer = firstRevolutionsSample;
		}

}
```

abz.h

```
#ifndef ABZ_H_
#define ABZ_H_

#include <stdbool.h>

enum Tc_abz_direction{
		ABZ_DIR_POSITIVE = false,
		ABZ_DIR_NEGATIVE = true
};

struct Tc_abz_param{
		bool enabled;
		bool measure_position;
		bool measure_speed;
		bool no_dir_sense;
		bool count_phB;
		bool invert_phA;
		bool invert_phB;
		bool invert_phZ;
		bool swap_inputs;
		bool phB_as_phZ;
		bool auto_correct;
		uint8_t filter_max_ticks;
};

struct Tc_abz_result{
		int32_t position_raw;
		int32_t rotations_raw;
		uint16_t position;
		int32_t rotations;
		int32_t speed;
		enum Tc_abz_direction direction;
};

void tc_abz_init(struct Tc_abz_param* settings, uint32_t timeBase, Tc *tc);
enum Tc_abz_direction tc_abz_get_direction(Tc *tc);
bool tc_abz_is_initialized();
int16_t abz_getCount(Tc * tc);
void abz_getPosition(Tc * tc, int32_t * encoderPosPointer, int32_t * numRevolutionsPointer);

extern const uint16_t encoder_max;


#endif /* ABZ_H_ */
```

pwm.c

```c
#include <sam.h>
#include <stdbool.h>

#include "sam4e-base/RevolveDrivers/pio.h"
#include "sam4e-base/RevolveDrivers/pmc.h"

// Private functions prototypes
int pwm_selectChannelPrescaler(uint8_t channel, uint8_t channelPrescalerBM);
void pwm_enableChannels(uint8_t channelBitmap);
void pwm_setChannelPeriodRegister(uint8_t channel, uint32_t counterTopValue);
void pwm_setWavePolarity(uint8_t channel, bool startHighNotLow);
void pwm_setWaveAlignment(uint8_t channel, bool centerNotLeftAlignment);
int pwm_selectChannelPrescaler(uint8_t channel, uint8_t channelPrescalerBM);
void pwm_setChannelCounterEventSelection(uint8_t channel, bool eventAtHalfPeriod);
void pwm_setChannelUpdateSelection(uint8_t channel, bool updateAtHalfPeriod);
void pwm_setOutputOverrideValue(bool highNotLow);
void pwm_disableWriteprotect();
void pwm_enableWriteprotect();

static uint32_t periodRegisterValue;

// API functions
void pwm_init(uint32_t peripheralClockFrequency, uint32_t pwmFrequency, uint32_t deadTimeFrequency)
{
		pmc_enable_periph_clk(PWM_IRQn);

		pio_setMux(PIOD, 21, A);
		pio_setMux(PIOD, 25, A);

		pio_setMux(PIOD, 22, A);
		pio_setMux(PIOD, 26, A);

		pio_setMux(PIOD, 23, A);
		pio_setMux(PIOD, 27, A);

		pwm_disableWriteprotect();

		PWM->PWM_SCM |= PWM_SCM_SYNC0 | PWM_SCM_SYNC1 | PWM_SCM_SYNC2 | PWM_SCM_SYNC3; // set all channels in sync mode

		pwm_setDeadTime(1, peripheralClockFrequency/deadTimeFrequency);
		pwm_setDeadTime(2, peripheralClockFrequency/deadTimeFrequency);
		pwm_setDeadTime(3, peripheralClockFrequency/deadTimeFrequency);

		pwm_selectChannelPrescaler(0, PWM_CMR_CPRE_MCK);
		periodRegisterValue = peripheralClockFrequency/pwmFrequency/2;

		pwm_setChannelPeriodRegister(0, periodRegisterValue);
		pwm_setChannelCounterEventSelection(0, true);

		pwm_setChannelUpdateSelection(0, true);
		pwm_setChannelUpdateSelection(1, true);
		pwm_setChannelUpdateSelection(2, true);
		pwm_setChannelUpdateSelection(3, true);

		pwm_setWavePolarity(1, false);
		pwm_setWavePolarity(2, false);
		pwm_setWavePolarity(3, false);

		pwm_setWaveAlignment(0, true);

		pwm_setOutputOverrideValue(false);

		pwm_enableChannels(PWM_ENA_CHID0 | PWM_ENA_CHID1 | PWM_ENA_CHID2 | PWM_ENA_CHID3);

		pwm_enableWriteprotect();
}

void pwm_updateDutyCycles(float duty1, float duty2, float duty3)
{
		PWM->PWM_CH_NUM[1].PWM_CDTYUPD = duty1 * periodRegisterValue;
		PWM->PWM_CH_NUM[2].PWM_CDTYUPD = duty2 * periodRegisterValue;
		PWM->PWM_CH_NUM[3].PWM_CDTYUPD = duty3 * periodRegisterValue;
		PWM->PWM_SCUC = PWM_SCUC_UPDULOCK;
}

void pwm_setDrivers(bool enableDrivers){
		if(enableDrivers)
				PWM->PWM_OSC =
(PWM_OS_OSH0|PWM_OS_OSH1|PWM_OS_OSH2|PWM_OS_OSH3|PWM_OS_OSL0|PWM_OS_OSL1|PWM_OS_OSL2|PWM_OS_OSL3);
		else
				PWM->PWM_OSS =
(PWM_OS_OSH0|PWM_OS_OSH1|PWM_OS_OSH2|PWM_OS_OSH3|PWM_OS_OSL0|PWM_OS_OSL1|PWM_OS_OSL2|PWM_OS_OSL3);
}

// Helper functions
void pwm_setChannelUpdateSelection(uint8_t channel, bool updateAtHalfPeriod)
{
		#ifndef PWM_CMR_UPDS
				#define PWM_CMR_UPDS (1<<11)
		#endif
		PWM->PWM_CH_NUM[channel].PWM_CMR &= ~ PWM_CMR_UPDS;
		if (updateAtHalfPeriod)
				PWM->PWM_CH_NUM[channel].PWM_CMR |= PWM_CMR_UPDS;
}

void pwm_setOutputOverrideValue(bool highNotLow)
```

```
{
		if(highNotLow)
			PWM->PWM_OOV |=
(PWM_OOV_OOVH0|PWM_OOV_OOVH1|PWM_OOV_OOVH2|PWM_OOV_OOVH3|PWM_OOV_OOVL0|PWM_OOV_OOVL1|PWM_OOV_OOVL2|PWM_OOV_OOVL3);
		else
			PWM->PWM_OOV &=
~(PWM_OOV_OOVH0|PWM_OOV_OOVH1|PWM_OOV_OOVH2|PWM_OOV_OOVH3|PWM_OOV_OOVL0|PWM_OOV_OOVL1|PWM_OOV_OOVL2|PWM_OOV_OOVL3);
}

void pwm_setChannelCounterEventSelection(uint8_t channel, bool eventAtHalfPeriod)
{
		PWM->PWM_CH_NUM[channel].PWM_CMR &= ~PWM_CMR_CES;
		if (eventAtHalfPeriod)
			PWM->PWM_CH_NUM[channel].PWM_CMR |= PWM_CMR_CES;
}

void pwm_setDeadTime(uint8_t channel, uint32_t deadTimeCycles)
{
		PWM->PWM_CH_NUM[channel].PWM_CMR |= PWM_CMR_DTE|PWM_CMR_DTHI|PWM_CMR_DTLI; // enable dead time generator

		PWM->PWM_CH_NUM[channel].PWM_DT = 0x00;
		PWM->PWM_CH_NUM[channel].PWM_DT |= PWM_DT_DTH(deadTimeCycles); // set deadTimeHigh
		PWM->PWM_CH_NUM[channel].PWM_DT |= PWM_DT_DTL(deadTimeCycles); // set deadTimeLow
}


void pwm_enableChannels(uint8_t channelBitmap)
{
		uint8_t channelMask = PWM_ENA_CHID0 | PWM_ENA_CHID1 | PWM_ENA_CHID2 | PWM_ENA_CHID3;
		PWM->PWM_ENA = channelMask & channelBitmap;
}

// pwm_setChannelPeriodRegister requires peripheral clock for PWM to be enabled
void pwm_setChannelPeriodRegister(uint8_t channel, uint32_t counterTopValue)
{
		PWM->PWM_CH_NUM[channel].PWM_CPRD = (counterTopValue & PWM_CPRD_CPRD_Msk);
}

// pwm_setWavePolarity requires peripheral clock for PWM to be enabled
void pwm_setWavePolarity(uint8_t channel, bool startHighNotLow){
		PWM->PWM_CH_NUM[channel].PWM_CMR &=		~PWM_CMR_CPOL;
		if (startHighNotLow)
			PWM->PWM_CH_NUM[channel].PWM_CMR |= PWM_CMR_CPOL;
}

void pwm_setWaveAlignment(uint8_t channel, bool centerNotLeftAlignment){
		PWM->PWM_CH_NUM[channel].PWM_CMR &=		~PWM_CMR_CALG;
		if (centerNotLeftAlignment)
			PWM->PWM_CH_NUM[channel].PWM_CMR |= PWM_CMR_CALG;
}

// This function is not very intuitive. channelPrescalerBM values can only be what's described in switch - case. Consider making it an enumerator.
int pwm_selectChannelPrescaler(uint8_t channel, uint8_t channelPrescalerBM)
{
		if (channel > 3 && channel < 0)
			return -1;

		switch (channelPrescalerBM){
			case PWM_CMR_CPRE_CLKA:
			case PWM_CMR_CPRE_CLKB:
			case PWM_CMR_CPRE_MCK_DIV_1024:
			case PWM_CMR_CPRE_MCK_DIV_512:
			case PWM_CMR_CPRE_MCK_DIV_256:
			case PWM_CMR_CPRE_MCK_DIV_128:
			case PWM_CMR_CPRE_MCK_DIV_64:
			case PWM_CMR_CPRE_MCK_DIV_32:
			case PWM_CMR_CPRE_MCK_DIV_16:
			case PWM_CMR_CPRE_MCK_DIV_8:
			case PWM_CMR_CPRE_MCK_DIV_4:
			case PWM_CMR_CPRE_MCK_DIV_2:
			case PWM_CMR_CPRE_MCK:
				break;
			default:
				return -1;
		}

		PWM->PWM_CH_NUM[channel].PWM_CMR &= ~PWM_CMR_CPRE_Msk;
		PWM->PWM_CH_NUM[channel].PWM_CMR |= channelPrescalerBM;

		return 0;
}

void pwm_disableWriteprotect(){
		PWM->PWM_WPCR = PWM_WPCR_WPKEY_PASSWD | PWM_WPCR_WPCMD_DISABLE_SW_PROT | PWM_WPCR_WPRG0 | PWM_WPCR_WPRG1 |
PWM_WPCR_WPRG2 | PWM_WPCR_WPRG3 | PWM_WPCR_WPRG4 | PWM_WPCR_WPRG5;
}

void pwm_enableWriteprotect(){
		PWM->PWM_WPCR = PWM_WPCR_WPKEY_PASSWD | PWM_WPCR_WPCMD_ENABLE_SW_PROT | PWM_WPCR_WPRG0 | PWM_WPCR_WPRG1 |
PWM_WPCR_WPRG2 | PWM_WPCR_WPRG3 | PWM_WPCR_WPRG4 | PWM_WPCR_WPRG5;
}
```

pwm.h

```
#ifndef PWM_H_
#define PWM_H_

#include <stdint.h>
#include <stdbool.h>

void pwm_init(uint32_t peripheralClockFrequency, uint32_t pwmFrequency, uint32_t deadTimeFrequency);
void pwm_updateDutyCycles(float duty1, float duty2, float duty3);
void pwm_setDrivers(bool enableDrivers);

#endif /* PWM_H_ */
```

CurrentSensor.c

```
#include "CurrentSensor.h"

#define ARM_MATH_CM4
#include <CMSIS/Include/arm_math.h>
#include "sam4e-base/RevolveDrivers/can.h"

void current_init()
{
		pio_init();

		pio_setOutput(CURRENT_1_CONV_PIO, CURRENT_1_CONV_PIN, LOW);
		pio_enableOutput(CURRENT_1_CONV_PIO, CURRENT_1_CONV_PIN);

		pio_setOutput(CURRENT_2_CONV_PIO, CURRENT_2_CONV_PIN, LOW);
		pio_enableOutput(CURRENT_2_CONV_PIO, CURRENT_2_CONV_PIN);

		pio_setOutput(CURRENT_3_CONV_PIO, CURRENT_3_CONV_PIN, LOW);
		pio_enableOutput(CURRENT_3_CONV_PIO, CURRENT_3_CONV_PIN);

		pio_setOutput(CURRENT_4_CONV_PIO, CURRENT_4_CONV_PIN, LOW);
		pio_enableOutput(CURRENT_4_CONV_PIO, CURRENT_4_CONV_PIN);

		pio_disableOutput(CURRENT_1_INT_PIO, CURRENT_1_INT_PIN);
		pio_disableOutput(CURRENT_2_INT_PIO, CURRENT_2_INT_PIN);
		pio_disableOutput(CURRENT_3_INT_PIO, CURRENT_3_INT_PIN);
		pio_disableOutput(CURRENT_4_INT_PIO, CURRENT_4_INT_PIN);
}

void current_startConversion(uint8_t sensorNum)
{
		switch(sensorNum){
				case 1:
				pio_setOutput(CURRENT_1_CONV_PIO, CURRENT_1_CONV_PIN, HIGH);
				break;
		case 2:
				pio_setOutput(CURRENT_2_CONV_PIO, CURRENT_2_CONV_PIN, HIGH);
				break;
		case 3:
				pio_setOutput(CURRENT_3_CONV_PIO, CURRENT_3_CONV_PIN, HIGH);
				break;
		case 4:
				pio_setOutput(CURRENT_4_CONV_PIO, CURRENT_4_CONV_PIN, HIGH);
				break;
		}
}

float current_calculateCurrent(int16_t adcValue, int sensorNum)
{
		float32_t ratio = ((float32_t)adcValue)/((float32_t)MAX_ADC_VALUE);

		float32_t voltage = ratio * ((float32_t)MAX_VOLTAGE);

		float32_t current = 0.0;

		switch(sensorNum){
				case 1:
						current = voltage/((float32_t) R1);
				break;
				case 2:
						current = voltage/((float32_t) R2);
				break;
				case 3:
						current = voltage/((float32_t) R3);
				break;
				case 4:
						current = voltage/((float32_t) R4);
				break;
		}

		current = current*LEM_CURRENT_RATIO;

		return current;
}
```

CurrentSensor.h

```
#ifndef CURRENTSENSOR_H_
#define CURRENTSENSOR_H_

#include "sam4e-base/RevolveDrivers/pio.h"

#define CURRENT_1_CONV_PIO PIOA
#define CURRENT_1_CONV_PIN 20

#define CURRENT_1_INT_PIO PIOC
#define CURRENT_1_INT_PIN 13

#define CURRENT_2_CONV_PIO PIOC
#define CURRENT_2_CONV_PIN 3

#define CURRENT_2_INT_PIO PIOC
#define CURRENT_2_INT_PIN 2

#define CURRENT_3_CONV_PIO PIOD
#define CURRENT_3_CONV_PIN 28

#define CURRENT_3_INT_PIO PIOA
#define CURRENT_3_INT_PIN 27

#define CURRENT_4_CONV_PIO PIOD
#define CURRENT_4_CONV_PIN 17

#define CURRENT_4_INT_PIO PIOA
#define CURRENT_4_INT_PIN 5

#define MAX_ADC_VALUE 0x7fff
#define MAX_VOLTAGE 10.24

#define R1 25
#define R2 50
#define R3 50
#define R4 25

#define LEM_CURRENT_RATIO 2000

void current_init();
void current_startConversion(uint8_t sensorNum);
float current_calculateCurrent(int16_t adcValue, int sensorNum);
#endif /* CURRENTSENSOR_H_ */
```

Switch.c

```
#include <stdbool.h>

#include "Switch.h"
#include "../sam4e-base/RevolveDrivers/pio.h"
#include "../sam4e-base/RevolveDrivers/pmc.h"


void switch_init()
{
/*		pio_setOutput(SWITCH_BOT_1_PIO, SWITCH_BOT_1_PIN, LOW);
		pio_setOutput(SWITCH_TOP_1_PIO, SWITCH_TOP_1_PIN, LOW);
		pio_setOutput(SWITCH_BOT_2_PIO, SWITCH_BOT_2_PIN, LOW);
		pio_setOutput(SWITCH_TOP_2_PIO, SWITCH_TOP_2_PIN, LOW);
		pio_setOutput(SWITCH_BOT_3_PIO, SWITCH_BOT_3_PIN, LOW);
		pio_setOutput(SWITCH_TOP_3_PIO, SWITCH_TOP_3_PIN, LOW);

		pio_enableOutput(SWITCH_BOT_1_PIO, SWITCH_BOT_1_PIN);
		pio_enableOutput(SWITCH_TOP_1_PIO, SWITCH_TOP_1_PIN);
		pio_enableOutput(SWITCH_BOT_2_PIO, SWITCH_BOT_2_PIN);
		pio_enableOutput(SWITCH_TOP_2_PIO, SWITCH_TOP_2_PIN);
		pio_enableOutput(SWITCH_BOT_3_PIO, SWITCH_BOT_3_PIN);
		pio_enableOutput(SWITCH_TOP_3_PIO, SWITCH_TOP_3_PIN);
*/
		pio_disableOutput(PIOD, 21);
		pio_disableOutput(PIOD, 25);
		pio_disableOutput(PIOD, 22);
		pio_disableOutput(PIOD, 26);
		pio_disableOutput(PIOD, 23);
		pio_disableOutput(PIOD, 27);

		pmc_enable_periph_clk(PIOA_IRQn);
		pmc_enable_periph_clk(PIOB_IRQn);
		pmc_enable_periph_clk(PIOC_IRQn);
		pmc_enable_periph_clk(PIOD_IRQn);
		pmc_enable_periph_clk(PIOE_IRQn);
}


void switch_disable()
{
		pio_setOutput(SWITCH_BOT_1_PIO, SWITCH_BOT_1_PIN, LOW);
		pio_setOutput(SWITCH_TOP_1_PIO, SWITCH_TOP_1_PIN, LOW);
```

```
		pio_setOutput(SWITCH_BOT_2_PIO, SWITCH_BOT_2_PIN, LOW);
		pio_setOutput(SWITCH_TOP_2_PIO, SWITCH_TOP_2_PIN, LOW);
		pio_setOutput(SWITCH_BOT_3_PIO, SWITCH_BOT_3_PIN, LOW);
		pio_setOutput(SWITCH_TOP_3_PIO, SWITCH_TOP_3_PIN, LOW);
}

// Return binary number where bits represents which switches that has error
int switch_getError()
{
		int error = 0;
		if(pio_readPin(ERROR_1_PIO, ERROR_1_PIN))
			error += (1<<0);

		if(pio_readPin(ERROR_2_PIO, ERROR_2_PIN))
			error += (1<<1);

		if(pio_readPin(ERROR_3_PIO, ERROR_3_PIN))
			error += (1<<2);

		return error;
}


void switch_set(int switchNum, bool topNotBot)
{
		switch(switchNum){
			case(1):
				if(topNotBot){
					pio_setOutput(SWITCH_BOT_1_PIO, SWITCH_BOT_1_PIN, LOW);
					pio_setOutput(SWITCH_TOP_1_PIO, SWITCH_TOP_1_PIN, HIGH);
				} else {
					pio_setOutput(SWITCH_TOP_1_PIO, SWITCH_TOP_1_PIN, LOW);
					pio_setOutput(SWITCH_BOT_1_PIO, SWITCH_BOT_1_PIN, HIGH);
				}
				break;
			case(2):
				if(topNotBot){
					pio_setOutput(SWITCH_BOT_2_PIO, SWITCH_BOT_2_PIN, LOW);
					pio_setOutput(SWITCH_TOP_2_PIO, SWITCH_TOP_2_PIN, HIGH);
					} else {
					pio_setOutput(SWITCH_TOP_2_PIO, SWITCH_TOP_2_PIN, LOW);
					pio_setOutput(SWITCH_BOT_2_PIO, SWITCH_BOT_2_PIN, HIGH);
				}
				break;
			case(3):
				if(topNotBot){
					pio_setOutput(SWITCH_BOT_3_PIO, SWITCH_BOT_3_PIN, LOW);
					pio_setOutput(SWITCH_TOP_3_PIO, SWITCH_TOP_3_PIN, HIGH);
				} else {
					pio_setOutput(SWITCH_TOP_3_PIO, SWITCH_TOP_3_PIN, LOW);
					pio_setOutput(SWITCH_BOT_3_PIO, SWITCH_BOT_3_PIN, HIGH);
				}
				break;
		}
}


void switch_applyVector(int vector)
{
		switch(vector){
			case 0:
				switch_set(1, false);
				switch_set(2, false);
				switch_set(3, false);
				break;
			case 1:
				switch_set(1, true);
				switch_set(2, false);
				switch_set(3, false);
				break;
			case 2:
				switch_set(1, false);
				switch_set(2, true);
				switch_set(3, false);
				break;
			case 3:
				switch_set(1, true);
				switch_set(2, true);
				switch_set(3, false);
				break;
			case 4:
				switch_set(1, false);
				switch_set(2, false);
				switch_set(3, true);
				break;
			case 5:
				switch_set(1, true);
				switch_set(2, false);
				switch_set(3, true);
				break;
			case 6:
				switch_set(1, false);
				switch_set(2, true);
				switch_set(3, true);
				break;
			case 7:
				switch_set(1, true);
				switch_set(2, true);
				switch_set(3, true);
```

```
                        break;
                default:
                        break;
        }
}
```

Switch.h

```
#ifndef SWITCH_H_
#define SWITCH_H_

#include "../sam4e-base/RevolveDrivers/pio.h"
#include <stdbool.h>

int switch_getError();
void switch_set(int switchNum, bool topNotBot);
void switch_applyVector(int vector);
int switch_getError();
void switch_init();
void switch_disable();

#define SWITCH_BOT_1_PIO PIOA
#define SWITCH_BOT_1_PIN 6

#define SWITCH_TOP_1_PIO PIOA
#define SWITCH_TOP_1_PIN 30

#define ERROR_1_PIO PIOD
#define ERROR_1_PIN 10

#define SWITCH_BOT_2_PIO PIOA
#define SWITCH_BOT_2_PIN 28

#define SWITCH_TOP_2_PIO PIOD
#define SWITCH_TOP_2_PIN 8

#define ERROR_2_PIO PIOC
#define ERROR_2_PIN 16

#define SWITCH_BOT_3_PIO PIOD
#define SWITCH_BOT_3_PIN 9

#define SWITCH_TOP_3_PIO PIOC
#define SWITCH_TOP_3_PIN 18

#define ERROR_3_PIO PIOD
#define ERROR_3_PIN 11

#endif /* SWITCH_H_ */
```

spi.c

```
#include "sam4e-base/RevolveDrivers/pio.h"
#include "sam4e-base/RevolveDrivers/pmc.h"

void spi_disableWriteProtetction()
{
        SPI->SPI_WPMR = SPI_WPMR_WPKEY_PASSWD;
}

void spi_enableWriteProtetction()
{
        SPI->SPI_WPMR = SPI_WPMR_WPKEY_PASSWD|SPI_WPMR_WPEN;
}

void spi_init(uint32_t peripheralClockFrequency, uint32_t spiClockFrequency)
{
        spi_disableWriteProtetction();

        pio_setMux (PIOA, 12, A);   //MISO
        pio_setMux (PIOA, 13, A);   //MOSI

        pio_setMux (PIOA, 11, A);   //cs0
        pio_setMux (PIOA, 9, B);    //cs1
        pio_setMux (PIOA, 10, B);   //cs2
        pio_setMux (PIOA, 3, B);    //cs3

        pio_setMux (PIOA, 14, A);   //SPCK

        pmc_enable_periph_clk(SPI_IRQn);

//      NVIC_ClearPendingIRQ(SPI_IRQn);
//      NVIC_EnableIRQ(SPI_IRQn);

        #define DELAY_BETWEEN_CS 10
```

```
	SPI->SPI_CR = SPI_CR_SPIEN;			// enable SPI
	SPI->SPI_MR = SPI_MR_MSTR|SPI_MR_PS|(DELAY_BETWEEN_CS<<SPI_MR_DLYBCS_Pos);

	/*SPI->SPI_IER = SPI_IER_RXBUFF;*/

	uint32_t scbr = peripheralClockFrequency/spiClockFrequency;

	for(int i = 0; i < 4; i++)
	{
		SPI->SPI_CSR[i] = SPI_CSR_CSNAAT|SPI_CSR_BITS_16_BIT|(scbr<<SPI_CSR_SCBR_Pos)|SPI_CSR_NCPHA|SPI_CSR_CPOL|SPI_CSR_DLYBS(3);
	}
}

uint32_t spi_tranceive(uint16_t transmit, uint8_t cs)
{
	SPI->SPI_TDR = ( (0xf & ~(1<<cs) ) << SPI_TDR_PCS_Pos)|SPI_TDR_LASTXFER|transmit;

	while(!(SPI->SPI_SR & SPI_SR_RDRF)); // block untill new data is received

	return SPI->SPI_RDR;
}
```

spi.h

```
#ifndef SPI_H_
#define SPI_H_

uint32_t spi_tranceive(uint16_t transmit, uint8_t cs);
void spi_init(uint32_t peripheralClockFrequency, uint32_t spiClockFrequency);

#endif /* SPI_H_ */
```

B.4: Modules

angleConversion.c

```
#include <stdint.h>

float angleConversions_encoderToDegrees(uint8_t mostSignificantBitNumber, uint32_t encoderValue)
{
	uint32_t encoderMaxValue = ( 1 << mostSignificantBitNumber ) - 1;
	return ((float)(encoderValue * 360))/((float) encoderMaxValue);
}

float angleConversions_encoderToEletricalDegrees(uint8_t mostSignificantBitNumber, int32_t encoderValue, uint8_t polePairs)
{
	int32_t encoderMaxValue = ((1 << mostSignificantBitNumber) - 1);
	return ((float) ( ((encoderValue * polePairs) % encoderMaxValue) * 360 ) )/((float) encoderMaxValue);
}

float angleConversions_encoderToRPM(int32_t encoderValue, int32_t numRevolutions, int32_t sampleFrequency)
{
	static float rpmEstimate;
	static int32_t lastNumRevolutions;
	static int32_t lastEncoderValue;
	static int32_t callsSinceLastChange;

	if((encoderValue == lastEncoderValue) && (numRevolutions == lastNumRevolutions)){
		callsSinceLastChange++;
	} else {
		int32_t deltaEncoderValue = (numRevolutions - lastNumRevolutions)*2048 + (encoderValue - lastEncoderValue);
		float rawRpmEstimate = ((float)deltaEncoderValue) * ((float)sampleFrequency) * 60.0 / ((float)(callsSinceLastChange)) / 2048.0;
		rpmEstimate = rpmEstimate * 0.9 + 0.1*rawRpmEstimate; // needs tuning
		callsSinceLastChange = 1;
	}

	lastNumRevolutions = numRevolutions;
	lastEncoderValue = encoderValue;

	return rpmEstimate;
}
```

angleConversion.h

```
#include <stdint.h>

#ifndef ANGLECONVERSIONS_H_
#define ANGLECONVERSIONS_H_

float angleConversions_encoderToDegrees(uint8_t mostSignificantBitNumber, uint32_t encoderValue);
float angleConversions_encoderToEletricalDegrees(uint8_t mostSignificantBitNumber, uint32_t encoderValue, uint8_t polePairs);
float angleConversions_encoderToRPM(int32_t encoderValue, int32_t numRevolutions, int32_t sampleFrequency);

#endif /* ANGLECONVERSIONS_H_ */
```

adcConversion.c

```
#include <stdint.h>

float adcConversion_linearConversion(int32_t rawValue,
								int32_t rawValueMin,
								int32_t rawValueMax,
								float conversionValueMin,
								float conversionValueMax)
{
	int32_t rawRangeSize = rawValueMax - rawValueMin;
	int32_t compensatedRawValue = rawValue - rawValueMin;

	float conversionRangeSize = conversionValueMax - conversionValueMin;

	return conversionValueMin + ((float) compensatedRawValue)*conversionRangeSize/( (float) rawRangeSize);
}
```

adcConversion.h

```
#ifndef ADCCONVERSION_H_
#define ADCCONVERSION_H_


#include <stdint.h>

float adcConversion_linearConversion(int32_t rawValue,
                                     int32_t rawValueMin,
                                     int32_t rawValueMax,
                                     float conversionValueMin,
                                     float conversionValueMax);

#endif /* ADCCONVERSION_H_ */
```

error.c

```
#include <stdbool.h>
#include <stdint.h>


#include "error.h"

static bool initiated = false;
static uint32_t errorCounter;
static uint32_t recoverableErrorVector;
static uint32_t errorVector, accumulatedErrorVector; // assumingnever more than 32 errors

int error_init(uint32_t recoverableErrorBitMask)
{
   if(initiated)
                                    return -1;
   recoverableErrorVector = recoverableErrorBitMask;
   errorVector = 0;
                  accumulatedErrorVector = 0;
   initiated = true;
   return 0;
}

void error_signal(uint32_t errorBitMask)
{
   errorVector |= errorBitMask;
                  accumulatedErrorVector |= errorVector;
                  errorCounter++;
}

void error_clear(uint32_t errorBitMask)
{
   uint32_t recoverVector = errorBitMask & recoverableErrorVector;
   errorVector &= ~(recoverVector);
}

bool error_check(uint32_t errorBitMask)
{
   return ((bool)(errorVector & errorBitMask));
}

uint32_t error_getVector()
{
                  return errorVector;
}

uint32_t error_getLastVector()
{
                  return accumulatedErrorVector;
}

uint32_t error_getCount()
{
                  return errorCounter;
}
```

error.h

```
#ifndef ERROR_H_
#define ERROR_H_


int error_init(uint32_t recoverableErrorBitMask);
void error_signal(uint32_t errorBitMask);
void error_clear(uint32_t errorBitMask);
bool error_check(uint32_t errorBitMask);
uint32_t error_getVector();
```

```
uint32_t error_getLastVector();
uint32_t error_getCount();

#endif /* ERROR_H_ */
```

foc.c

```
#include <sam.h>
#include <stdbool.h>

#define ARM_MATH_CM4
#include <CMSIS/Include/arm_math.h>
#include <math.h>

#include "foc.h"

// The controller tuning should be moved to config file and made agnostic due to to loop frequency
static arm_pid_instance_f32 qPid;
static arm_pid_instance_f32 dPid;
static float qPid_Ki, dPid_Ki; // Used in integral anti windup

void foc_init(float K_p, float K_i, float K_d)
{
		dPid.Kp = K_p;
		dPid.Kd = K_d;
		dPid.Ki = K_i;
		dPid_Ki = K_i;

		qPid.Kp = K_p;
		qPid.Kd = K_d;
		qPid.Ki = K_i;
		qPid_Ki = K_i;

		arm_pid_init_f32(&qPid, 1);
		arm_pid_init_f32(&dPid, 1);
}

void foc_antiWindup(bool enable)
{
		if(enable){
				qPid.Ki = 0.0;
				dPid.Ki = 0.0;
				arm_pid_init_f32(&qPid, 0);
				arm_pid_init_f32(&dPid, 0);
		} else {
				qPid.Ki = qPid_Ki;
				dPid.Ki = dPid_Ki;
				arm_pid_init_f32(&qPid, 0);
				arm_pid_init_f32(&dPid, 0);
		}
}

void foc_reset_regulator(){
		arm_pid_init_f32(&qPid, 1);
		arm_pid_init_f32(&dPid, 1);
}

void foc_calculate(
							// Inputs
							float currentA,
							float currentB,
							float eletricalDegrees,
							float setPointCurrentQ,
							float setPointCurrentD,
							// Outputs
							float * currentQPointer,
							float * currentDPointer,
							float * voltageAPointer,
							float * voltageBPointer,
							float * voltageCPointer,
							bool runPid
							)
{
		// Variables made static to increase performance
		static float32_t currentAlpha, currentBeta;
		static float32_t currentD, currentQ;
		static float32_t sinVal, cosVal;

		static float32_t deltaCurrentQ, deltaCurrentD;

		static float32_t voltageQ, voltageD;
		static float32_t voltageAlpha, voltageBeta;


		arm_sin_cos_f32(eletricalDegrees, &sinVal, &cosVal);
		arm_clarke_f32(currentA, currentB, &currentAlpha, &currentBeta);
		arm_park_f32(currentAlpha, currentBeta, &currentD, &currentQ, sinVal, cosVal);

		deltaCurrentQ = setPointCurrentQ - currentQ;
		deltaCurrentD = setPointCurrentD - currentD;

		if(runPid){
```

```
			voltageQ = arm_pid_f32(&qPid, deltaCurrentQ);
			voltageD = arm_pid_f32(&dPid, deltaCurrentD);

		}
		else{
			voltageQ = 0;
			voltageD = 0;
		}

		arm_inv_park_f32(voltageD, voltageQ, &voltageAlpha, &voltageBeta, sinVal, cosVal);
		arm_inv_clarke_f32(voltageAlpha, voltageBeta, voltageAPointer, voltageBPointer);

		*currentDPointer = currentD;
		*currentQPointer = currentQ;
		*voltageCPointer = - *voltageAPointer - *voltageBPointer;
}
```

foc.h

```
#ifndef FOC_H_
#define FOC_H_

#include <stdbool.h>

void foc_init(float K_p, float K_i, float K_d);

void foc_calculate(
		// Inputs
		float currentA,
		float currentB,
		float eletricalDegrees,
		float setPointCurrentQ,
		float setPointCurrentD,
		// Outputs
		float * currentQPointer,
		float * currentDPointer,
		float * voltageAPointer,
		float * voltageBPointer,
		float * voltageCPointer,
		bool runPid
);

void foc_reset_regulator();

void foc_antiWindup(bool enable);

#endif /* FOC_H_ */
```

commutation.c

```
#include <sam.h>
#include <stdbool.h>

#define ARM_MATH_CM4
#include <CMSIS/Include/arm_math.h>
#include <math.h>

static arm_pid_instance_f32 alphaPid;
static arm_pid_instance_f32 betaPid;

void commutation_init(float K_p, float K_i, float K_d)
{
		alphaPid.Kp = K_p;
		alphaPid.Kd = K_d;
		alphaPid.Ki = K_i;

		betaPid.Kp = K_p;
		betaPid.Kd = K_d;
		betaPid.Ki = K_i;

		arm_pid_init_f32(&alphaPid, 1);
		arm_pid_init_f32(&betaPid, 1);
}

void commutation_sinusodial(
							float currentA,
							float currentB,
							float eletricalDegreeSetpoint,
							float currentSetpoint,
							float * voltageAPointer,
							float * voltageBPointer,
							float * voltageCPointer)
{
		static float32_t sinVal, cosVal;
```

```
	static float32_t currentAlpha, currentBeta;
	static float32_t voltageAlpha, voltageBeta;
	static float32_t setpointCurrentAlpha, setpointCurrentBeta;
	static float32_t deltaCurrentAlpha, deltaCurrentBeta;

	arm_sin_cos_f32(eletricalDegreeSetpoint, &sinVal, &cosVal);
	arm_clarke_f32(currentA, currentB, &currentAlpha, &currentBeta);

	setpointCurrentAlpha = cosVal * currentSetpoint;
	setpointCurrentBeta = sinVal * currentSetpoint;

	deltaCurrentAlpha = setpointCurrentAlpha - currentAlpha;
	deltaCurrentBeta = setpointCurrentBeta - currentBeta;

	voltageAlpha = arm_pid_f32(&alphaPid, deltaCurrentAlpha);
	voltageBeta = arm_pid_f32(&betaPid, deltaCurrentBeta);

	arm_inv_clarke_f32(voltageAlpha, voltageBeta, voltageAPointer, voltageBPointer);

	*voltageCPointer = - *voltageAPointer - *voltageBPointer;
}
```

commutation.h

```
#ifndef COMMUTATION_H_
#define COMMUTATION_H_

void commutation_init(float K_p, float K_i, float K_d);

void commutation_sinusodial(
		float currentA,
		float currentB,
		float eletricalDegreeSetpoint,
		float currentSetpoint,
		float * voltageAPointer,
		float * voltageBPointer,
		float * voltageCPointer);
#endif /* COMMUTATION_H_ */
```

voltageModulation.c

```
#include <stdbool.h>

// Gives duty cycle between 0.0 and 1.0 centered around 0.5 (modulatedVoltages = 0 -> dutyCycle 0.5)
float voltageModulation_voltageToPwm(float modulatedVoltage, float dcVoltage, const float minDutyCycle, const float maxDutyCycle, const float saturation_overflow_allowed,
bool * saturation)
{
	float dutyCycle = 0.5 + (modulatedVoltage/dcVoltage)*0.5;

	if((dutyCycle > (maxDutyCycle + saturation_overflow_allowed)) || (dutyCycle < (minDutyCycle - saturation_overflow_allowed))){
		*saturation = true;
	}
	else{
		*saturation = false;
	}

	if(dutyCycle > maxDutyCycle)
	{
		dutyCycle = maxDutyCycle;
	} else if(dutyCycle < minDutyCycle) {
		dutyCycle = minDutyCycle;
	}

	return dutyCycle;
}
```

voltageModulation.h

```
#ifndef VOLTAGEMODULATION_H_
#define VOLTAGEMODULATION_H_

float voltageModulation_voltageToPwm(float modulatedVoltage, float dcVoltage, const float minDutyCycle, const float maxDutyCycle, const float saturation_overflow_allowed,
bool * saturation);

#endif /* VOLTAGEMODULATION_H_ */
```

B.5: Settings

ControlSettings.h

```
#ifndef CONTROLFLOW_H_
#define CONTROLFLOW_H_

/* General timing settings */
#define CPU_FREQ 120000000

#define PWM_FREQUENCY 8000
#define DEAD_TIME_FREQUENCY 500000


/* Controller tuning settings */
#define KP_BASE 2.05
#define KI_BASE 0.001
#define KD_BASE 0.0

#define WINDUP_ALLOWANCE 0.2

#endif /* CONTROLFLOW_H_ */
```

boardHardware.h

```
#include <sam.h>

#ifndef BOARDHARDWARE_H_
#define BOARDHARDWARE_H_

// HV voltage measure definition
#define HV_VOLTAGE_MEASURE_ADC_INSTANCE AFEC1
#define HV_VOLTAGE_MEASURE_ADC_NUM 4
#define HV_VOLTAGE_MEASURE_ADC_OFFSET (683 - 9)
#define HV_VOLTAGE_MEASURE_ADC_VALUE_AT_MAX (HV_VOLTAGE_MEASURE_ADC_OFFSET + 2482)

#define HV_VOLTAGE_MEASURE_VOLTAGE_MIN 0.0
#define HV_VOLTAGE_MEASURE_VOLTAGE_MAX 800.0

// Current measurement definitions
#define CURRENT_1_SPI_CS 0
#define CURRENT_2_SPI_CS 1
#define CURRENT_3_SPI_CS 2
#define CURRENT_4_SPI_CS 3

#define CURRENT_A_SPI_CS CURRENT_1_SPI_CS
#define CURRENT_B_SPI_CS CURRENT_4_SPI_CS

// ENCODER settings
#define ENCODER_OFFSET 0x426

#endif /* BOARDHARDWARE_H_ */
```

errorCodes.h

```
#ifndef STATUSBITS_H_
#define STATUSBITS_H_

// List of permanent shutdown limits
#define CURRENT_PHASE_LIMIT_PERMANENT 400.0
#define VOLTAGE_DC_MAX_LIMIT_PERMANENT 700.0

#define MAX_TRANSIENT_CURRENT_ERRORS 3

// List of transient shutdown limits
#define CURRENT_PHASE_LIMIT_TRANSIENT 420.0
#define VOLTAGE_DC_MAX_LIMIT_TRANSIENT 600.0
#define VOLTAGE_DC_MIN_LIMIT_TRANSIENT 15.0
#define RPM_LIMIT_TRANSIENT 5000.0
#define TORQUE_REQUEST_LOWER_LIMIT_TRANSIENT 0.1
#define RPM_HYSTERESIS 100.0

// List of all possible errors
#define CURRENT_LIMIT_EXCEEDED_PERMANENT 0    // (1<<0)
#define VOLTAGE_LIMIT_EXCEEDED_PERMANENT (1<<1)

#define CURRENT_LIMIT_EXCEEDED_PHASE_A_TRANSIENT (1<<8)
#define CURRENT_LIMIT_EXCEEDED_PHASE_B_TRANSIENT (1<<9)
#define CURRENT_LIMIT_EXCEEDED_PHASE_C_TRANSIENT (1<<10)
#define VOLTAGE_LIMIT_EXCEEDED_TRANSIENT (1<<11)
```

```
#define SPEED_LIMIT_EXCEEDED_TRANSIENT (1<<12)
#define GATE_DRIVER_ERROR (1<<13)
#define VOLTAGE_TO_LOW (1<<14)
#define TORQUE_REQUEST_TOO_LOW_TRANSIENT (1 << 15)

// Status bits
#define ENCODER_ZERO_NOT_PASSED (1<<16)

// Gate driver fault add hoc
#define GATE_DRIVER_ERROR_A (1<<21)
#define GATE_DRIVER_ERROR_B (1<<22)
#define GATE_DRIVER_ERROR_C (1<<23)

// Integrator saturation
#define PID_SATURATION_A            (1 << 18)
#define PID_SATURATION_B            (1 << 19)
#define PID_SATURATION_C            (1 << 20)

// Switchable status bits
#define READY_TO_DRIVE (1<<24)
#define ACCEPT_TORQUE_REQUEST_FROM_TELEMETRY_NOT_ECU (1<<25)

// Permanent (unrecoverable) errors
#define PERMANENT_ERRORS (CURRENT_LIMIT_EXCEEDED_PERMANENT|VOLTAGE_LIMIT_EXCEEDED_PERMANENT)

// Transient errors
#define TRANSIENT_ERRORS (SPEED_LIMIT_EXCEEDED_TRANSIENT \
                          |CURRENT_LIMIT_EXCEEDED_PHASE_A_TRANSIENT \
                          |CURRENT_LIMIT_EXCEEDED_PHASE_B_TRANSIENT \
                          |CURRENT_LIMIT_EXCEEDED_PHASE_C_TRANSIENT \
                          |VOLTAGE_LIMIT_EXCEEDED_TRANSIENT \
                          |GATE_DRIVER_ERROR \
                          |VOLTAGE_TO_LOW)

// Status, not errors
#define STATUS_BITS (ENCODER_ZERO_NOT_PASSED \
                     |READY_TO_DRIVE \
                     |ACCEPT_TORQUE_REQUEST_FROM_TELEMETRY_NOT_ECU\
                     |TORQUE_REQUEST_TOO_LOW_TRANSIENT\
                     |PID_SATURATION_A\
                     |PID_SATURATION_B\
                     |PID_SATURATION_C)

#endif /* STATUSBITS_H_ */
```

motorConstants.h

```
#ifndef MOTORCONSTANTS_H_
#define MOTORCONSTANTS_H_

#define MOTOR_TORQUE_PER_CURRENT 1.1 // 1.1 NM / Arms

#endif /* MOTORCONSTANTS_H_ */
```

B.6: RevolveDrivers

pmc.c

```
#include "pmc.h"
#include <sam.h>

#define MAX_PERIPH_ID    47

#ifdef __cplusplus
extern "C" {
#endif

void pmc_enable_writeprotect(void)
{
		PMC->PMC_WPMR = PMC_WPMR_WPKEY_PASSWD | PMC_WPMR_WPEN;
}

void pmc_disable_writeprotect(void)
{
		PMC->PMC_WPMR = PMC_WPMR_WPKEY_PASSWD;
}

void pmc_mck_set_prescaler(uint32_t pres)
{
		PMC->PMC_MCKR = (PMC->PMC_MCKR & (~PMC_MCKR_PRES_Msk)) | pres;
		while (!(PMC->PMC_SR & PMC_SR_MCKRDY));
}

void pmc_mck_set_source(uint32_t css)
{
		PMC->PMC_MCKR = (PMC->PMC_MCKR & (~PMC_MCKR_CSS_Msk)) | css;
		while (!(PMC->PMC_SR & PMC_SR_MCKRDY));
}

void pmc_switch_mainck_to_fastrc(void)
{
		PMC->CKGR_MOR = (PMC->CKGR_MOR & ~CKGR_MOR_MOSCSEL) | CKGR_MOR_KEY_PASSWD;
}

void pmc_enable_fastrc(void)
{
		PMC->CKGR_MOR |= (CKGR_MOR_KEY_PASSWD | CKGR_MOR_MOSCRCEN);
		while (!(PMC->PMC_SR & PMC_SR_MOSCRCS));		/* Wait the Fast RC to stabilize */
}

void pmc_set_fastrc_frequency(uint32_t moscrcf)
{
		PMC->CKGR_MOR = (PMC->CKGR_MOR & ~CKGR_MOR_MOSCRCF_Msk) | CKGR_MOR_KEY_PASSWD | moscrcf;
		while (!(PMC->PMC_SR & PMC_SR_MOSCRCS));		/* Wait the Fast RC to stabilize */
}

void pmc_disable_fastrc(void)
{
		PMC->CKGR_MOR = (PMC->CKGR_MOR & ~CKGR_MOR_MOSCRCEN & ~CKGR_MOR_MOSCRCF_Msk) | CKGR_MOR_KEY_PASSWD;
}

uint32_t pmc_osc_is_ready_fastrc(void)
{
		return (PMC->PMC_SR & PMC_SR_MOSCRCS);
}

uint32_t pmc_xtal_ready(void)
{
		return (PMC->PMC_SR & PMC_SR_MOSCXTS);
}

void pmc_enable_main_xtal(uint32_t xtalStartupTime)
{
		uint32_t mor = PMC->CKGR_MOR;
		mor &= ~(CKGR_MOR_MOSCXTBY|CKGR_MOR_MOSCXTEN);
		mor |= CKGR_MOR_KEY_PASSWD | CKGR_MOR_MOSCXTEN |
							CKGR_MOR_MOSCXTST(xtalStartupTime);
		PMC->CKGR_MOR = mor;
		/* Wait the main Xtal to stabilize */
		while (!pmc_xtal_ready());
}

void pmc_switch_mainck_to_xtal(void)
{
		PMC->CKGR_MOR |= CKGR_MOR_KEY_PASSWD | CKGR_MOR_MOSCSEL;
		while(!pmc_xtal_ready());
}

uint32_t pmc_mainck_ready(void)
{
		return PMC->PMC_SR & PMC_SR_MOSCSELS;
}

void pmc_disable_pllack(void)
{
		PMC->CKGR_PLLAR = CKGR_PLLAR_ONE | CKGR_PLLAR_MULA(0);
}
```

```
uint32_t pmc_plla_is_locked(void)
{
		return (PMC->PMC_SR & PMC_SR_LOCKA);
}

void pmc_enable_pllack(uint32_t mula, uint32_t pllacount, uint32_t diva)
{
		/* first disable the PLL to unlock the lock */
		pmc_disable_pllack();

		PMC->CKGR_PLLAR = CKGR_PLLAR_ONE | CKGR_PLLAR_DIVA(diva) | CKGR_PLLAR_PLLACOUNT(pllacount) | CKGR_PLLAR_MULA(mula);
		if(diva == 0 || mula == 0)
			return;

		while (!pmc_plla_is_locked());
}

void pmc_disable_usb_clock(void)
{
		PMC->PMC_SCDR = PMC_SCDR_UDP;
}

void pmc_enable_interrupt(uint32_t ul_sources)
{
		PMC->PMC_IER = ul_sources;
}

void pmc_disable_interrupt(uint32_t ul_sources)
{
		PMC->PMC_IDR = ul_sources;
}

uint32_t pmc_get_interrupt_mask(void)
{
		return PMC->PMC_IMR;
}

void pmc_enable_clock_failure_detector(void)
{
		PMC->CKGR_MOR |= CKGR_MOR_KEY_PASSWD | CKGR_MOR_CFDEN;
}

void pmc_disable_clock_failure_detector(void)
{
		PMC->CKGR_MOR = (PMC->CKGR_MOR & (~CKGR_MOR_CFDEN)) | CKGR_MOR_KEY_PASSWD;
}

void pmc_select_main_clock(MainClockFrequency freq)
{
		switch(freq){
			case INTERNAL_4MHZ:
			case INTERNAL_8MHZ:
			case INTERNAL_12MHZ:

			pmc_enable_fastrc();
			pmc_set_fastrc_frequency(freq);
			pmc_switch_mainck_to_fastrc();

			break;

			case EXTERNAL:
			pmc_enable_main_xtal(0xff);
			pmc_switch_mainck_to_xtal();
			pmc_disable_fastrc();

			break;
		}
}

void pmc_select_master_clock(MasterClockSource css, ProcessorClockPrescaler pres)
{
		if(css == PLLA_CLOCK){
			pmc_mck_set_prescaler(pres);
			pmc_mck_set_source(css);
			} else {
			pmc_mck_set_source(css);
			pmc_mck_set_prescaler(pres);
		}
}

uint32_t pmc_disable_periph_clk(uint32_t irqnNumber)
{
		if (irqnNumber > MAX_PERIPH_ID)
		return 1;

		if (irqnNumber < 32) {
			if ((PMC->PMC_PCSR0 & (1u << irqnNumber)) == (1u << irqnNumber))
			PMC->PMC_PCDR0 = 1 << irqnNumber;
			} else {
			irqnNumber -= 32;
			if ((PMC->PMC_PCSR1 & (1u << irqnNumber)) == (1u << irqnNumber))
			PMC->PMC_PCDR1 = 1 << irqnNumber;
		}
		return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////

void pmc_enable_usb_clock(int divide)
```

```
{
		if(divide > 16 && divide < 1)
			return;

		pmc_enable_periph_clk(UDP_IRQn);

		pmc_disable_writeprotect();

		PMC->PMC_USB = PMC_USB_USBDIV(divide-1);
		PMC->PMC_SCER = PMC_SCER_UDP;

		pmc_enable_writeprotect();
}

uint32_t pmc_enable_periph_clk(uint32_t irqnNumber)
{
		if (irqnNumber > MAX_PERIPH_ID)
		return 1;

		pmc_disable_writeprotect();
		if (irqnNumber < 32) {
			if ((PMC->PMC_PCSR0 & (1 << irqnNumber)) != (1 << irqnNumber))
			PMC->PMC_PCER0 = (1 << irqnNumber);
			} else {
			irqnNumber -= 32;
			if ((PMC->PMC_PCSR1 & (1 << irqnNumber)) != (1 << irqnNumber))
			PMC->PMC_PCER1 = (1 << irqnNumber);
		}
		pmc_enable_writeprotect();

		return 0;
}

uint32_t pmc_init(struct PmcInit pmc_init_struct)
{
		pmc_disable_writeprotect();
		pmc_select_main_clock(pmc_init_struct.freq);
		pmc_enable_pllack(pmc_init_struct.multiply, 0x3f, pmc_init_struct.divide);
		pmc_select_master_clock(pmc_init_struct.css, pmc_init_struct.pres);
		pmc_enable_writeprotect();

		return 0;
}

#ifdef __cplusplus
}
#endif
```

pmc.h

```
#ifndef PMC_H_INCLUDED
#define PMC_H_INCLUDED

#include "sam.h"

/// @cond 0
/**INDENT-OFF**/
#ifdef __cplusplus
extern "C" {
#endif
/**INDENT-ON**/
/// @endcond

/** Bit mask for peripheral clocks (PCER0) */
#define PMC_MASK_STATUS0	(0xFFFFFFFC)

/** Bit mask for peripheral clocks (PCER1) */
#define PMC_MASK_STATUS1	(0xFFFFFFFF)

/** Loop counter timeout value */
#define PMC_TIMEOUT		(2048)

/** Key to unlock CKGR_MOR register */
#ifndef CKGR_MOR_KEY_PASSWD
#define CKGR_MOR_KEY_PASSWD    CKGR_MOR_KEY(0x37U)
#endif

/** Key used to write SUPC registers */
#ifndef SUPC_CR_KEY_PASSWD
#define SUPC_CR_KEY_PASSWD    SUPC_CR_KEY(0xA5U)
#endif

#ifndef SUPC_MR_KEY_PASSWD
#define SUPC_MR_KEY_PASSWD    SUPC_MR_KEY(0xA5U)
#endif
//
// /** Mask to access fast startup input */
// #define PMC_FAST_STARTUP_Msk    (0x7FFFFu)

/** PMC_WPMR Write Protect KEY, unlock it */
#ifndef PMC_WPMR_WPKEY_PASSWD
#define PMC_WPMR_WPKEY_PASSWD    PMC_WPMR_WPKEY((uint32_t) 0x504D43)
```

```
#endif

// /** Using external oscillator */
// #define PMC_OSC_XTAL          0
//
// /** Oscillator in bypass mode */
// #define PMC_OSC_BYPASS        1
//
// #define PMC_PCK_0             0 /* PCK0 ID */
// #define PMC_PCK_1             1 /* PCK1 ID */
// #define PMC_PCK_2             2 /* PCK2 ID */
//
// /** Flash state in Wait Mode */
// #define PMC_WAIT_MODE_FLASH_STANDBY         PMC_FSMR_FLPM_FLASH_STANDBY
// #define PMC_WAIT_MODE_FLASH_DEEP_POWERDOWN  PMC_FSMR_FLPM_FLASH_DEEP_POWERDOWN
// #define PMC_WAIT_MODE_FLASH_IDLE            PMC_FSMR_FLPM_FLASH_IDLE

//
// /** Convert startup time from us to MOSCXTST */
// #define pmc_us_to_moscxtst(startup_us, slowck_freq)      \
//              ((startup_us * slowck_freq / 8 / 1000000) < 0x100 ?  \
//                              (startup_us * slowck_freq / 8 / 1000000) : 0xFF)


typedef enum {INTERNAL_4MHZ = CKGR_MOR_MOSCRCF_4_MHz,
              INTERNAL_8MHZ = CKGR_MOR_MOSCRCF_8_MHz,
              INTERNAL_12MHZ = CKGR_MOR_MOSCRCF_12_MHz,
              EXTERNAL
              } MainClockFrequency;

typedef enum {CLK_1 = PMC_MCKR_PRES_CLK_1,
              CLK_2 = PMC_MCKR_PRES_CLK_2,
              CLK_4 = PMC_MCKR_PRES_CLK_4,
              CLK_8 = PMC_MCKR_PRES_CLK_8,
              CLK_16 = PMC_MCKR_PRES_CLK_16,
              CLK_32 = PMC_MCKR_PRES_CLK_32,
              CLK_64 = PMC_MCKR_PRES_CLK_64,
              CLK_3 = PMC_MCKR_PRES_CLK_3
              } ProcessorClockPrescaler;

typedef enum {SLOW_CLOCK = PMC_MCKR_CSS_SLOW_CLK,
              MAIN_CLOCK = PMC_MCKR_CSS_MAIN_CLK,
              PLLA_CLOCK = PMC_MCKR_CSS_PLLA_CLK
              } MasterClockSource;

struct PmcInit {
              MainClockFrequency freq;
              MasterClockSource css;
              ProcessorClockPrescaler pres;
              uint8_t divide;
              uint8_t multiply;
};

void pmc_enable_usb_clock(int divide);
uint32_t pmc_enable_periph_clk(uint32_t irqnNumber);
uint32_t pmc_init(struct PmcInit pmc_init_struct);

#endif /* PMC_H_INCLUDED */
```

eefc.c

```
#define MASTER_CLOCK_FREQ 120000000

#include "sam.h"

void init_flash(void)
{
        /* Set FWS for embedded Flash access according to operating frequency */
        if ( MASTER_CLOCK_FREQ < CHIP_FREQ_FWS_0 ) {
                EFC->EEFC_FMR = EEFC_FMR_FWS(0)|EEFC_FMR_CLOE;
                } else {
                if (MASTER_CLOCK_FREQ < CHIP_FREQ_FWS_1) {
                        EFC->EEFC_FMR = EEFC_FMR_FWS(1)|EEFC_FMR_CLOE;
                        } else {
                        if (MASTER_CLOCK_FREQ < CHIP_FREQ_FWS_2) {
                                EFC->EEFC_FMR = EEFC_FMR_FWS(2)|EEFC_FMR_CLOE;
                                } else {
                                if ( MASTER_CLOCK_FREQ < CHIP_FREQ_FWS_3 ) {
                                        EFC->EEFC_FMR = EEFC_FMR_FWS(3)|EEFC_FMR_CLOE;
                                        } else {
                                        if ( MASTER_CLOCK_FREQ < CHIP_FREQ_FWS_4 ) {
                                                EFC->EEFC_FMR = EEFC_FMR_FWS(4)|EEFC_FMR_CLOE;
                                                } else {
                                                EFC->EEFC_FMR = EEFC_FMR_FWS(5)|EEFC_FMR_CLOE;
                                        }
                                }
                        }
                }
        }
}
```

eefc.h

```c
#ifndef EEFC_H_
#define EEFC_H_

void init_flash(void);

#endif /* EEFC_H_ */
```

can.c

```c
#include <stdlib.h>
#include "can.h"
#include "pio.h"
#include "pmc.h"

/** The max value for CAN baudrate prescale. */
#define CAN_BAUDRATE_MAX_DIV       128

/** Define the scope for TQ (time quantum). */
#define CAN_MIN_TQ_NUM            8
#define CAN_MAX_TQ_NUM            25

/** Define the mailbox mode. */
#define CAN_MB_DISABLE_MODE         0
#define CAN_MB_RX_MODE              1
#define CAN_MB_RX_OVER_WR_MODE       2
#define CAN_MB_TX_MODE              3
#define CAN_MB_CONSUMER_MODE         4
#define CAN_MB_PRODUCER_MODE         5

#define TX_BOX_ID 0
#define NUMBER_OF_MAILBOXES 8

#define CAN_WPMR_WP

/* Init help functions and data structures */

//ASF-stuff
typedef struct {
		uint8_t tq;      /**< CAN_BIT_SYNC + uc_prog + uc_phase1 + uc_phase2
				= uc_tq, 8 <= uc_tq <= 25. */
		uint8_t prog;    /**< Propagation segment, (3-bits + 1), 1~8; */
		uint8_t phase1;  /**< Phase segment 1, (3-bits + 1), 1~8; */
		uint8_t phase2;  /**< Phase segment 2, (3-bits + 1), 1~8, CAN_BIT_IPT
				<= uc_phase2; */
		uint8_t sjw;     /**< Resynchronization jump width, (2-bits + 1),
				min(uc_phase1, 4); */
		uint8_t sp;      /**< Sample point value, 0~100 in percent. */
} can_bit_timing_t;

/** Values of bit time register for different baudrates, Sample point =
 * ((1 + uc_prog + uc_phase1) / uc_tq) * 100%. */
const can_bit_timing_t can_bit_time[] = {
		{ 8, (2 + 1), (1 + 1), (1 + 1), (2 + 1), 75},
		{ 9, (1 + 1), (2 + 1), (2 + 1), (1 + 1), 67},
		{10, (2 + 1), (2 + 1), (2 + 1), (2 + 1), 70},
		{11, (3 + 1), (2 + 1), (2 + 1), (3 + 1), 72},
		{12, (2 + 1), (3 + 1), (3 + 1), (3 + 1), 67},
		{13, (3 + 1), (3 + 1), (3 + 1), (3 + 1), 77},
		{14, (3 + 1), (3 + 1), (4 + 1), (3 + 1), 64},
		{15, (3 + 1), (4 + 1), (4 + 1), (3 + 1), 67},
		{16, (4 + 1), (4 + 1), (4 + 1), (3 + 1), 69},
		{17, (5 + 1), (4 + 1), (4 + 1), (3 + 1), 71},
		{18, (4 + 1), (5 + 1), (5 + 1), (3 + 1), 67},
		{19, (5 + 1), (5 + 1), (5 + 1), (3 + 1), 68},
		{20, (6 + 1), (5 + 1), (5 + 1), (3 + 1), 70},
		{21, (7 + 1), (5 + 1), (5 + 1), (3 + 1), 71},
		{22, (6 + 1), (6 + 1), (6 + 1), (3 + 1), 68},
		{23, (7 + 1), (7 + 1), (6 + 1), (3 + 1), 70},
		{24, (6 + 1), (7 + 1), (7 + 1), (3 + 1), 67},
		{25, (7 + 1), (7 + 1), (7 + 1), (3 + 1), 68}
};

void (*can_callback_function[2])(void);

void CAN0_Handler(){
		if(can_callback_function[0] != NULL){
				(*can_callback_function[0])();
		}
}
void CAN1_Handler(){
		if(can_callback_function[1] != NULL){
				(*can_callback_function[1])();
		}
}
```

```
void can_enableRXInterrupt(Can *can, void *(callback_function)(void));

void can_writeProtectionEnable(Can *can){
		can->CAN_WPMR = CAN_WPMR_WPKEY_PASSWD | CAN_WPMR_WPEN;
}

void can_writeProtectionDisable(Can *can){
		can->CAN_WPMR = CAN_WPMR_WPKEY_PASSWD;
}

void can_enable(Can *can)
{
		can_writeProtectionDisable(can);
		can->CAN_MR |= CAN_MR_CANEN;
		can_writeProtectionEnable(can);
}

void can_disable(Can *can)
{
		can_writeProtectionEnable(can);
		can->CAN_MR &= ~CAN_MR_CANEN;
		can_writeProtectionDisable(can);
}

// ASF
static uint32_t can_setBaudrate(Can *can, uint32_t mck, uint32_t baudrate)
{
		can_writeProtectionDisable(can);
		uint8_t tq;
		uint8_t prescale;
		uint32_t mod;
		uint32_t cur_mod;
		can_bit_timing_t *bitTime;

		/* Check whether the baudrate prescale will be greater than the max divide value. */
		if (((mck + (baudrate * CAN_MAX_TQ_NUM * 1000 - 1)) /
						(baudrate * CAN_MAX_TQ_NUM * 1000)) >
						CAN_BAUDRATE_MAX_DIV) {
				return 0;
		}

		/* Check whether the input MCK is too small. */
		if ((mck / 2)  < baudrate * CAN_MIN_TQ_NUM * 1000) {
				return 0;
		}

		/* Initialize it as the minimum Time Quantum. */
		tq = CAN_MIN_TQ_NUM;

		/* Initialize the remainder as the max value. When the remainder is 0, get the right TQ number. */
		mod = 0xffffffff;
		/* Find out the approximate Time Quantum according to the baudrate. */
		for (uint8_t i = CAN_MIN_TQ_NUM; i <= CAN_MAX_TQ_NUM; i++) {
				if ((mck / (baudrate * i * 1000)) <=
								CAN_BAUDRATE_MAX_DIV) {
						cur_mod = mck % (baudrate * i * 1000);
						if (cur_mod < mod) {
								mod = cur_mod;
								tq = i;
								if (!mod) {
										break;
								}
						}
				}
		}

		/* Calculate the baudrate prescale value. */
		prescale = mck / (baudrate * tq * 1000);
		if (prescale < 2) {
				return 0;
		}

		/* Get the right CAN BIT Timing group. */
		bitTime = (can_bit_timing_t *)&can_bit_time[tq - CAN_MIN_TQ_NUM];

		/* Before modifying the CANBR register, disable the CAN controller. */
		can_disable(can);

		/* Write into the CAN baudrate register. */
		can->CAN_BR = CAN_BR_PHASE2(bitTime->phase2 - 1) |
						CAN_BR_PHASE1(bitTime->phase1 - 1) |
						CAN_BR_PROPAG(bitTime->prog - 1) |
						CAN_BR_SJW(bitTime->sjw - 1) |
						CAN_BR_BRP(prescale - 1);
		can_writeProtectionEnable(can);
		return 1;
}

void can_setPeripheralMux(Can *can){
		if(can == CAN0){
				pio_setMux(PIOB, 3, A);
				pio_setMux(PIOB, 2, A);
		}
		else if(can == CAN1){
				pio_setMux(PIOC, 12, C);
				pio_setMux(PIOC, 15, C);
		}
}
```

```
void can_enablePMC(Can *can){
		can_writeProtectionDisable(can);
		if(can == CAN0){
			pmc_enable_periph_clk(37);
			//PMC->PMC_PCER1 |= 1<<5;				//ENABLE CLOCK, PID = 37
		}
		else if(can == CAN1){
			pmc_enable_periph_clk(38);
			//PMC->PMC_PCER1 |= 1<<6;				//ENABLE CLOCK, PID = 38
		}
		can_writeProtectionEnable(can);
}

void can_setupTXMailbox(Can *can, uint8_t mailbox_id, uint8_t priority){
		can_writeProtectionDisable(can);
		/* Set the priority in Transmit mode. */
		can->CAN_MB[mailbox_id].CAN_MMR =
			(can->CAN_MB[mailbox_id].CAN_MMR & ~CAN_MMR_PRIOR_Msk) |
			(priority << CAN_MMR_PRIOR_Pos);

		/* Set box into TX mode*/
		can->CAN_MB[mailbox_id].CAN_MMR =
			(can->CAN_MB[mailbox_id].CAN_MMR & ~CAN_MMR_MOT_Msk) |
			(CAN_MB_TX_MODE << CAN_MMR_MOT_Pos);
		can_writeProtectionEnable(can);
}

void can_setupRXMailbox(Can *can, uint8_t mailbox_id, uint32_t acceptence_mask, uint32_t id_mask){
		can_writeProtectionDisable(can);
		/* Set acceptance mask  */
		can->CAN_MB[mailbox_id].CAN_MAM = CAN_MAM_MIDvA(acceptence_mask);
		/* Set message ID mask */
		can->CAN_MB[mailbox_id].CAN_MID = CAN_MID_MIDvA(id_mask);
		/* Set box into RX mode */
		can->CAN_MB[mailbox_id].CAN_MMR =
			(can->CAN_MB[mailbox_id].CAN_MMR & ~CAN_MMR_MOT_Msk) |
			(CAN_MB_RX_MODE << CAN_MMR_MOT_Pos);
		can_writeProtectionEnable(can);
}

void can_disableMailbox(Can *can, uint8_t mailbox_id){
		can_writeProtectionDisable(can);
		can->CAN_MB[mailbox_id].CAN_MMR = 0;
		can->CAN_MB[mailbox_id].CAN_MAM = 0;
		can->CAN_MB[mailbox_id].CAN_MID = 0;
		can->CAN_MB[mailbox_id].CAN_MDL = 0;
		can->CAN_MB[mailbox_id].CAN_MDH = 0;
		can->CAN_MB[mailbox_id].CAN_MCR = 0;
		can_writeProtectionEnable(can);
}

void can_setupFilters(Can *can, uint32_t acceptence_masks[7], uint32_t id_masks[7]){
		can_writeProtectionDisable(can);
		for (int mailboxID = 0; mailboxID < NUMBER_OF_MAILBOXES; mailboxID++)
		{
			if ( mailboxID != TX_BOX_ID)
			{
				can_setupRXMailbox(can, mailboxID, acceptence_masks[mailboxID], id_masks[mailboxID]);
			}
		}
		can_writeProtectionEnable(can);
}

void can_init(Can *can, uint32_t peripheral_clock_hz, uint32_t baudrate_kbps, uint32_t acceptence_masks[7], uint32_t id_masks[7], void *(callback_function)(void)){
		can_writeProtectionDisable(can);
		can_disable(can);
		can_setPeripheralMux(can);
		can_enablePMC(can);
		if(!can_setBaudrate(can, peripheral_clock_hz, baudrate_kbps)){
			return; // Illegal combination of peripheral_clock_hz and baudrate_kbps
		}

		// Reset 8 all mailboxes
		for (int i = 0; i<8; i++)
		{
			can_disableMailbox(can, i);
		}
		can_enable(can);
		/* Wait until the CAN is synchronized with the bus activity. */
		while(!(can->CAN_SR & CAN_SR_WAKEUP));

		//init TX mailbox:
		can_setupTXMailbox(can, TX_BOX_ID, 0);
		can_setupFilters(can, acceptence_masks, id_masks);
		can_enableRXInterrupt(can, callback_function);
		can_writeProtectionEnable(can);
}

enum CanTXstatus can_sendMessage(Can *can, struct CanMessage message){
		//check if mailbox is ready
		if ( !(can->CAN_MB[TX_BOX_ID].CAN_MSR & CAN_MSR_MRDY) )
		{
			return TRANSFER_BUSY;
		}
		can_writeProtectionDisable(can);
		// Write message id to mailboxz
		can->CAN_MB[TX_BOX_ID].CAN_MID = CAN_MID_MIDvA(message.messageID);
```

```
		// Write data
		can->CAN_MB[TX_BOX_ID].CAN_MDL = message.data.u32[0];
		if (message.dataLength > 4)
		{
			can->CAN_MB[TX_BOX_ID].CAN_MDH = message.data.u32[1];
		}

		// Write data length and initate transfer
		can->CAN_MB[TX_BOX_ID].CAN_MCR = CAN_MCR_MTCR | CAN_MCR_MDLC(message.dataLength);
		can_writeProtectionEnable(can);
		return TRANSFER_OK;
}

void can_read_message(Can *can, uint8_t mailboxID, struct CanMessage *message){
		// Get data length
		message->dataLength = (can->CAN_MB[mailboxID].CAN_MSR & CAN_MSR_MDLC_Msk) >> CAN_MSR_MDLC_Pos;

		// Get the message id
		message->messageID = (can->CAN_MB[mailboxID].CAN_MID & CAN_MID_MIDvA_Msk) >> CAN_MID_MIDvA_Pos;

		// Get the data
		message->data.u32[0] =  can->CAN_MB[mailboxID].CAN_MDL;
		if (message->dataLength > 4)
		{
			message->data.u32[1] =  can->CAN_MB[mailboxID].CAN_MDH;
		}

		// Allow new incomming message
		can->CAN_MB[mailboxID].CAN_MCR = CAN_MCR_MTCR;
}

void can_enableRXInterrupt(Can *can, void *(callback_function)(void)){

		if(callback_function == NULL){
			return;
		}
		else{
			if (can == CAN0){
				can_callback_function[0] = callback_function;
				NVIC_DisableIRQ(CAN0_IRQn);
				NVIC_ClearPendingIRQ(CAN0_IRQn);
				NVIC_SetPriority(CAN0_IRQn,5); // must be >= 5
				NVIC_EnableIRQ(CAN0_IRQn);
			}else if(can == CAN1){
				can_callback_function[1] = callback_function;
				NVIC_DisableIRQ(CAN1_IRQn);
				NVIC_ClearPendingIRQ(CAN1_IRQn);
				NVIC_SetPriority(CAN1_IRQn,5); // must be >= 5
				NVIC_EnableIRQ(CAN1_IRQn);
			}
			can_writeProtectionDisable(can);
			// Enable all interrupts on mailboxes that are in RX mode
			can->CAN_IER = 0xff & ~(1<<TX_BOX_ID);
			can_writeProtectionEnable(can);
		}
}

void can_disableRXInterrupt(Can *can){
		if (can == CAN0){
			can_callback_function[0] = NULL;
			NVIC_DisableIRQ(CAN0_IRQn);
			NVIC_ClearPendingIRQ(CAN0_IRQn);
		}else if(can == CAN1){
			can_callback_function[1] = NULL;
			NVIC_DisableIRQ(CAN1_IRQn);
			NVIC_ClearPendingIRQ(CAN1_IRQn);
		}
}

enum CANReceiveStatus can_popMessage(Can *can, struct CanMessage *message){

		for (int i = 0; i < NUMBER_OF_MAILBOXES; i++)
		{
			if (i != TX_BOX_ID && (can->CAN_MB[i].CAN_MSR & CAN_MSR_MRDY))
			{
				can_read_message(can, i, message);
				return GOT_NEW_MESSAGE;
			}
		}
		return NO_NEW_MESSAGE;
}
```

can.h

```
#ifndef _CAN_H_
#define _CAN_H_

#include "sam.h"

#define CAN_BPS_1000K          1000
#define CAN_BPS_800K           800
#define CAN_BPS_500K           500
#define CAN_BPS_250K           250
#define CAN_BPS_125K           125
#define CAN_BPS_50K            50
#define CAN_BPS_25K            25
#define CAN_BPS_10K            10
#define CAN_BPS_5K             5

enum CanTXstatus {TRANSFER_OK = 0, TRANSFER_BUSY = 1};
enum CANReceiveStatus {GOT_NEW_MESSAGE = 0, NO_NEW_MESSAGE = 1};

typedef union can_data_t{
		// INTEGERS
		uint64_t u64;
		int64_t  i64;
		uint32_t u32[2];
		int32_t  i32[2];
		uint16_t u16[4];
		int16_t  i16[4];
		uint8_t  u8[8];
		int8_t   i8[8];

		// FLOAT
		float f[2];
		double db;
}CanData ;

struct CanMessage{
		CanData data;
		uint8_t dataLength;
		uint32_t messageID;
};

// All setup is now handled in can_init. Callback_function is called on interrupt.
void can_init(Can *can, uint32_t peripheral_clock_hz, uint32_t baudrate_kbps, uint32_t acceptence_masks[7], uint32_t id_masks[7], void *(callback_function)(void));
// When using can_setupFilters, CAN1_handler or/and CAN0_handler must be defined by user
// void can_setupFilters(Can *can, uint32_t acceptence_masks[7], uint32_t id_masks[7]);
enum CanTXstatus can_sendMessage(Can *can, struct CanMessage message);          //NOT THREADSAFE
enum CANReceiveStatus can_popMessage(Can *can, struct CanMessage *message);            //NOT THREADSAFE

#endif /* _CAN_H_ */
```

afec.c

```
#include <sam.h>

void afec_set_offset(Afec * AFEC, uint16_t offset);
void afec_set_tracktime_in_clock_periods(Afec * AFEC, uint32_t clockPeriods);
void afec_set_prescaler(Afec * AFEC, uint32_t peripheralClockFrequency, uint32_t afecClockFrequency);
void afec_enable_free_run_mode(Afec * AFEC);
void afec_enable_free_run_mode(Afec * AFEC);
void afec_enable_write_protect(Afec * AFEC);
void afec_disable_write_protect(Afec * AFEC);

void afec_init(Afec * AFEC, uint32_t peripheralClockFrequency, uint16_t activeChannelsBitmap)
{
		if (AFEC == AFEC0)
				pmc_enable_periph_clk(AFEC0_IRQn);
		else if(AFEC == AFEC1)
				pmc_enable_periph_clk(AFEC1_IRQn);

		afec_disable_write_protect(AFEC);
		AFEC0->AFEC_MR |= AFEC_MR_STARTUP_SUT640;
		afec_set_offset(AFEC, 2048);
		AFEC->AFEC_CHER = activeChannelsBitmap;
		afec_set_tracktime_in_clock_periods(AFEC, 4);
		afec_set_prescaler(AFEC, peripheralClockFrequency, 20000000);
		afec_enable_free_run_mode(AFEC);
		afec_enable_write_protect(AFEC);

}

uint16_t afec_read_last_conversion(Afec * AFEC, uint8_t channel)
{
		AFEC->AFEC_CSELR = channel;
		return AFEC->AFEC_CDR;
}

/* Helper/convinience functions */
void afec_set_offset(Afec * AFEC, uint16_t offset)
{
```

```
		AFEC->AFEC_COCR = AFEC_COCR_AOFF(offset);
}

void afec_set_tracktime_in_clock_periods(Afec * AFEC, uint32_t clockPeriods)
{
		AFEC->AFEC_MR &= ~AFEC_MR_TRACKTIM_Msk;
		AFEC->AFEC_MR |= AFEC_MR_TRACKTIM(clockPeriods-1);
}

void afec_set_prescaler(Afec * AFEC, uint32_t peripheralClockFrequency, uint32_t afecClockFrequency)
{
		AFEC->AFEC_MR &= ~AFEC_MR_PRESCAL_Msk;
		AFEC->AFEC_MR |= AFEC_MR_PRESCAL(peripheralClockFrequency / (afecClockFrequency * 2) - 1);
}

void afec_enable_free_run_mode(Afec * AFEC)
{
		AFEC->AFEC_MR |= AFEC_MR_FREERUN;
}

void afec_disable_free_run_mode(Afec * AFEC)
{
		AFEC->AFEC_MR &= ~AFEC_MR_FREERUN;
}

void afec_enable_write_protect(Afec * AFEC)
{
		AFEC->AFEC_WPMR = AFEC_WPMR_WPKEY_PASSWD | AFEC_WPMR_WPEN;
}

void afec_disable_write_protect(Afec * AFEC)
{
		AFEC->AFEC_WPMR = AFEC_WPMR_WPKEY_PASSWD;
}
```

afec.h

```
#include <sam.h>

#ifndef AFEC_H_
#define AFEC_H_

int16_t afec_read_last_conversion(Afec * AFEC, uint8_t channel);
void afec_init(Afec * AFEC, uint32_t peripheralClockFrequency, uint16_t activeChannelsBitmap);

#endif /* AFEC_H_ */
```

pio.c

```
#include "pio.h"
#include "sam.h"
#include "pmc.h"
#include <stdbool.h>

static void pio_enablePullup					( Pio * pio, uint8_t pin);
static void pio_enablePulldown					( Pio * pio, uint8_t pin);
static void pio_disablePull					( Pio * pio, uint8_t pin);
static void pio_enableWriteProtection	( Pio * pio );
static void pio_disableWriteProtection	( Pio * pio );

#define WPKEY_ENABLE	0x50494F01
#define WPKEY_DISABLE	0x50494F00

static void (*pioaInterruptHooks[32])();
static void (*piobInterruptHooks[32])();
static void (*piocInterruptHooks[32])();
static void (*piodInterruptHooks[32])();
static void (*pioeInterruptHooks[32])();

void pio_init()
{
		pmc_enable_periph_clk(PIOA_IRQn);
		pmc_enable_periph_clk(PIOB_IRQn);
		pmc_enable_periph_clk(PIOC_IRQn);
		pmc_enable_periph_clk(PIOD_IRQn);
		pmc_enable_periph_clk(PIOE_IRQn);

		NVIC_EnableIRQ(PIOA_IRQn);
		NVIC_EnableIRQ(PIOB_IRQn);
		NVIC_EnableIRQ(PIOC_IRQn);
		NVIC_EnableIRQ(PIOD_IRQn);
		NVIC_EnableIRQ(PIOE_IRQn);
}

void pio_enableOutput(Pio * pio, uint8_t pin ){
```

```
		pio_disableWriteProtection(pio);
		pio->PIO_OER = ( 1 << pin );
		pio_enableWriteProtection(pio);
}

void pio_disableOutput(Pio * pio, uint8_t pin ){

		pio_disableWriteProtection(pio);
		pio->PIO_ODR = ( 1 << pin );
		pio_enableWriteProtection(pio);
}

void pio_setOutput(Pio * pio, uint8_t pin, enum PinLevel pinlevel ){

		switch(pinlevel){
			case(LOW):
				pio->PIO_CODR = (1<<pin);
				break;
			case(HIGH):
				pio->PIO_SODR = (1<<pin);
				break;
		}
}


void pio_setMux(Pio * pio, uint8_t pin, enum Peripheral mux)
{
		pio_disableWriteProtection(pio);
		switch(mux){
			case PIO:
				pio->PIO_PER = (1<<pin);
				break;
			case A:
				pio->PIO_PDR = (1<<pin);
				pio->PIO_ABCDSR[0] &= ~(1<<pin);
				pio->PIO_ABCDSR[1] &= ~(1<<pin);
				break;
			case B:
				pio->PIO_PDR = (1<<pin);
				pio->PIO_ABCDSR[0] |= (1<<pin);
				pio->PIO_ABCDSR[1] &= ~(1<<pin);
				break;
			case C:
				pio->PIO_PDR = (1<<pin);
				pio->PIO_ABCDSR[0] &= ~(1<<pin);
				pio->PIO_ABCDSR[1] |= (1<<pin);
				break;
			case D:
				pio->PIO_PDR = (1<<pin);
				pio->PIO_ABCDSR[0] |= (1<<pin);
				pio->PIO_ABCDSR[1] |= (1<<pin);
				break;
		}
		pio_enableWriteProtection(pio);
}

void pio_setPull ( Pio * pio, uint8_t pin, enum PullType pulltype ){

			switch(pulltype){
				case PULLUP:
					pio_enablePullup(pio,pin);
					break;

				case PULLDOWN:
					pio_enablePulldown(pio,pin);
					break;

				case NOPULL:
					pio_disablePull(pio,pin);
					break;

				default:
					//do nothing
					break;
			}
};

bool pio_readPin(Pio * pio, uint8_t pin)
{
		if(pio->PIO_PDSR & (1<<pin))
			return true;
		else
			return false;
}

void pio_setFilter(Pio * pio, uint8_t pin, enum FilterType filter)
{
		pio_disableWriteProtection(pio);
		switch(filter){
			case NONE:
				pio->PIO_IFDR = (1<<pin);
				break;
			case GLITCH:
				pio->PIO_IFER = (1<<pin);
				pio->PIO_IFSCDR = (1<<pin);
				break;
			case DEBOUNCE:
				pio->PIO_IFER = (1<<pin);
```

```c
						pio->PIO_IFSCER = (1<<pin);
						break;
		}
		pio_enableWriteProtection(pio);
}

void pio_setFallingEdgeInterrupt(Pio * pio, uint8_t pin)
{
		pio->PIO_AIMER = (1<<pin);
		pio->PIO_ESR = (1<<pin);
		pio->PIO_FELLSR = (1<<pin);
}

void pio_setRisingEdgeInterrupt(Pio * pio, uint8_t pin)
{
		pio->PIO_AIMER = (1<<pin);
		pio->PIO_ESR = (1<<pin);
		pio->PIO_REHLSR = (1<<pin);
}

void pio_setLowLevelInterrupt(Pio * pio, uint8_t pin)
{
		pio->PIO_AIMER = (1<<pin);
		pio->PIO_LSR = (1<<pin);
		pio->PIO_FELLSR = (1<<pin);
}

void pio_setHighLevelInterrupt(Pio * pio, uint8_t pin)
{
		pio->PIO_AIMER = (1<<pin);
		pio->PIO_LSR = (1<<pin);
		pio->PIO_REHLSR = (1<<pin);
}

void pio_enableInterrupt(Pio * pio, uint8_t pin, enum InterruptType interruptType, void (*interruptFunction)(void))
{
		pio_disableWriteProtection(pio);

		if(pio == PIOA){
				pioaInterruptHooks[pin] = interruptFunction;
		} else if(pio == PIOB){
				piobInterruptHooks[pin] = interruptFunction;
		} else if(pio == PIOC){
				piocInterruptHooks[pin] = interruptFunction;
		} else if(pio == PIOD){
				piodInterruptHooks[pin] = interruptFunction;
		} else if(pio == PIOE){
				pioeInterruptHooks[pin] = interruptFunction;
		}

		switch(interruptType){
				case FALLING_EDGE:
						pio_setFallingEdgeInterrupt(pio, pin);
						break;
				case RISING_EDGE:
						pio_setRisingEdgeInterrupt(pio, pin);
						break;
				case LOW_LEVEL:
						pio_setLowLevelInterrupt(pio, pin);
						break;
				case HIGH_LEVEL:
						pio_setHighLevelInterrupt(pio, pin);
						break;
		}

		volatile uint32_t deleteInterrupts = pio->PIO_ISR;

		pio->PIO_IER =  ( 1 << pin );

		pio_enableWriteProtection(pio);
}

void pio_disableInterrupt( Pio * pio, uint8_t pin)
{
		pio_disableWriteProtection(pio);
		pio->PIO_IDR = ( 1 << pin );
		pio_enableWriteProtection(pio);
}

void pio_enablePullup(Pio * pio, uint8_t pin)
{
		pio_disableWriteProtection(pio);
		pio->PIO_PPDDR = (1<<pin);		// disable pulldown
		pio->PIO_PUER = (1<<pin);		// enable pullup
		pio_enableWriteProtection(pio);
}

void pio_enablePulldown(Pio * pio, uint8_t pin)
{
		pio_disableWriteProtection(pio);
		pio->PIO_PUDR = (1<<pin);		// disable pullup
		pio->PIO_PPDER = (1<<pin);		// enable pulldown
		pio_enableWriteProtection(pio);
}

void pio_disablePull(Pio * pio, uint8_t pin)
{
		pio_disableWriteProtection(pio);
		pio->PIO_PUDR = (1<<pin);		// disable pullup
```

```
		pio->PIO_PPDDR = (1<<pin);		// disable pulldown
		pio_enableWriteProtection(pio);
}

void pio_enableWriteProtection( Pio * pio ){
		pio->PIO_WPMR = WPKEY_ENABLE;
}

void pio_disableWriteProtection( Pio * pio ){
		pio->PIO_WPMR = WPKEY_DISABLE;
}


void PIOA_Handler()
{
		uint32_t interruptMask = PIOA->PIO_ISR & PIOA->PIO_IMR;

		for(int i = 0; i < 32; i++){
			if(interruptMask & (1<<i)){
				(*pioaInterruptHooks[i])();
			}
		}
}

void PIOB_Handler()
{
		uint32_t interruptMask = PIOB->PIO_ISR & PIOB->PIO_IMR;

		for(int i = 0; i < 32; i++){
			if(interruptMask & (1<<i)){
				(*piobInterruptHooks[i])();
			}
		}
}

void PIOC_Handler()
{
		uint32_t interruptMask = PIOC->PIO_ISR & PIOC->PIO_IMR;

		for(int i = 0; i < 32; i++){
			if(interruptMask & (1<<i)){
				(*piocInterruptHooks[i])();
			}
		}
}

void PIOD_Handler()
{
		uint32_t interruptMask = PIOD->PIO_ISR & PIOD->PIO_IMR;

		for(int i = 0; i < 32; i++){
			if(interruptMask & (1<<i)){
				(*piodInterruptHooks[i])();
			}
		}
}

void PIOE_Handler()
{
		uint32_t interruptMask = PIOE->PIO_ISR & PIOE->PIO_IMR;

		for(int i = 0; i < 32; i++){
			if(interruptMask & (1<<i)){
				(*pioeInterruptHooks[i])();
			}
		}
}
```

pio.h

```
#ifndef PIO_H_
#define PIO_H_

#include "sam.h"

#include <stdbool.h>

enum Peripheral			{ PIO,		A, B, C, D		};
enum FilterType			{ NONE,		DEBOUNCE, GLITCH	};
enum PinLevel			{ LOW,		HIGH				};
enum PullType			{ PULLUP, PULLDOWN, NOPULL		};
enum InterruptType		{FALLING_EDGE, RISING_EDGE, LOW_LEVEL, HIGH_LEVEL};



void pio_init				();
void pio_enableOutput		( Pio * pio, uint8_t pin );
void pio_disableOutput		( Pio * pio, uint8_t pin );
void pio_setOutput			( Pio * pio, uint8_t pin, enum PinLevel setState);
void pio_setMux				( Pio * pio, uint8_t pin, enum Peripheral mux		);
void pio_setPull			( Pio * pio, uint8_t pin, enum PullType pull		);
```

```
bool pio_readPin                              ( Pio * pio, uint8_t pin );
void pio_setFilter                            ( Pio * pio, uint8_t pin, enum FilterType filter);
void pio_enableInterrupt(Pio * pio, uint8_t pin, enum InterruptType interruptType, void (*interruptFunction)(void));
void pio_disableInterrupt( Pio * pio, uint8_t pin);

#endif
```

B.7: util

delay.c

```
#include <sam.h>
#include "delay.h"

static uint32_t freq;

void delay_init(uint32_t frequency)
{
		freq = frequency;
}

void delay_clk(volatile uint32_t cycles)
{
		// R0 and R1 can be used freely inside the function, referance ARM calling convetion
		__asm("MOV R1, #3");		// 3 clock cycles per loop
		__asm("UDIV R0, R1"); // unsigned division
		__asm("loop: SUBS R0, R0, #1");
		__asm("BNE loop");
}

void delay_ms(uint32_t ms)
{
		delay_clk(ms*(freq/1000));
}

void delay_us(uint32_t us)
{
		delay_clk(us*(freq/1000000));
}
```

delay.h

```
#ifndef DELAY_H_
#define DELAY_H_

#include <sam.h>

void delay_init(uint32_t frequency);
void delay_ms(uint32_t ms);
void delay_us(uint32_t us);

#endif /* DELAY_H_ */
```

B.8: revolve_can_definitions.h

```
#ifndef REVOLVE_CAN_DEFINITIONS_H_
#define REVOLVE_CAN_DEFINITIONS_H_

//ID
//Module ID
//Bytemapping

//------------------------------------------------------------------------------
// EXAMPLE: function = command, group = dash, module ID = 0
/* .id = CANR_FCN_CMD_ID | CANR_GRP_DASH_ID | CANR_MODULE_ID0 */
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
/*			-------		ARBITRATION ID DEFINITIONS ---------------------------------------*/
//------------------------------------------------------------------------------

//- ID[10..8]
#define CANR_FCN_PRI_ID					0x000			// Most significant
#define CANR_FCN_BOOT_ID				0x200			// Priority data
#define CANR_FCN_CMD_ID						0x400			// Module_id control
#define CANR_FCN_DATA_ID				0x600			// Data

//- ID[7..3]
#define CANR_GRP_SENS_ROTARY_ID			0x10			// Torque and steering encoder
#define CANR_GRP_SENS_BRK_ID			0x18			// Brake encoder
#define CANR_GRP_SENS_SPEED_ID			0x20			// Wheel speed
#define CANR_GRP_BMS_ID						0x28			// BMS-data
#define CANR_GRP_ECU_ID					0x30			// ECU
#define CANR_GRP_INVERTER_ID			0x38			// Inverter data
#define CANR_GRP_SUSP_ID				0x40			// Suspension
#define CANR_GRP_SENS_IMU_ID			0x48			// IMU master
#define CANR_GRP_SENS_DAMPER_ID			0x50			// Damper position
#define CANR_GRP_SENS_TEMP_ID			0x56			// Gearbox and water temp
#define CANR_GRP_SENS_GLVBMS_ID			0x58			// GLV BMS
#define CANR_GRP_DASH_ID				0x60			// Dashboard
#define CANR_GRP_SENS_BSPD_ID			0x68			// BSPD
#define CANR_GRP_SENS_IMD_ID			0x70			// Laptimer
#define CANR_GRP_TELEMETRY_ID			0x78			// Telemetry
#define CANR_GRP_FAN_CTRL_ID			0x80			// Fan control
//#define CANR_GRP_						0x88			//
//#define CANR_GRP_						0x90			//

//- ID[2..0]
#define CANR_MODULE_ID0_ID				0x0
#define CANR_MODULE_ID1_ID				0x1
#define CANR_MODULE_ID2_ID				0x2
#define CANR_MODULE_ID3_ID				0x3
#define CANR_MODULE_ID4_ID				0x4
#define CANR_MODULE_ID5_ID				0x5
#define CANR_MODULE_ID6_ID				0x6
#define CANR_MODULE_ID7_ID				0x7


//------------------------------------------------------------------------------
/*			-------		MASKS ------------------------------------------------------*/
//------------------------------------------------------------------------------
/*MASK function:
			ID = ID in CAN controller filter
			MASK = Mask in CAN controller filter
			ID_msg = ID from incoming message

			Accept = (ID xor ID_msg) | (~MASK);
			if(Accept == 0xFF)  Message recieved
			else						Message rejected
*/

//Mask for recieving one group specifically,
//Group together to add more messages
//High bit means bit is significant in ID_ext
#define MASK_RECIEVE_FCN			(0x600)
#define MASK_RECIEVE_GRP			(0x1F8)
#define MASK_RECIEVE_MODULE			(0x007)
#define MASK_RECIEVE_ALL			(MASK_RECIEVE_FCN|MASK_RECIEVE_GRP|MASK_RECIEVE_MODULE)


//------------------------------------------------------------------------------
/*			-------		ALIVE		-------------------------------------------------------*/
//------------------------------------------------------------------------------

// ---------------------------------------------------------------------//
// Functionality description:
			//
// Each module sends out a message with the following properties:			//
// ID = CANR_FCN_DATA_ID | CANR_GRP_DASH_ID	| CANR_MODULE_ID7_ID			//
// DATA[0] = MODULE ID
					//
// Frequency = 1 Hz
							//
// ---------------------------------------------------------------------//

//-ALIVE ID
#define CAN_ALIVE_MSG_ID				(CANR_FCN_DATA_ID	|CANR_GRP_DASH_ID|CANR_MODULE_ID7_ID)

//- ALIVE BYTES
#define CANR_ALIVE_MSG_DLC				1
```

```
#define CANR_ALIVE_MODULE_B                     0

//Module Alive IDs
#define ALIVE_ECU                               0x00
#define ALIVE_BSPD                              0x01
#define ALIVE_TELEMETRY                         0x02
#define ALIVE_DASH                              0x03
#define ALIVE_ADC_FR                            0x04
#define ALIVE_ADC_FL                            0x05
#define ALIVE_ADC_RR                            0x06
#define ALIVE_ADC_RL                            0x07
#define ALIVE_INVERTER                          0x08
#define ALIVE_FAN                               0x09
#define ALIVE_BMS                               0x0A
#define ALIVE_GLVBMS                            0x0B
#define ALIVE_IMU                               0x0C
#define ALIVE_IMD                               0x0D
#define ALIVE_STEER_POS                         0x0E
#define ALIVE_STEER_POS_UNINIT                  0x0F
#define ALIVE_TRQ                               0x10
#define ALIVE_TRQ_UNINIT                        0x11

//-----------------------------------------------------------------------------
/*          ------- Module ID overrides   ---------------------------------------------*/
/*  If you want to make your own macro to replace module_id_x, make it here   */
//-----------------------------------------------------------------------------

/* BMS ID OVERRIDES */

#define BMS_I_AM_ALIVE_ID               (CANR_FCN_DATA_ID       |CANR_GRP_DASH_ID|CANR_MODULE_ID7_ID)



#define BMS_COMMFAULTS_MSG_ID       (CANR_FCN_DATA_ID    |CANR_GRP_BMS_ID|CANR_MODULE_ID0_ID )

#define BMS_TOTVTG_ID                   (CANR_FCN_DATA_ID       |CANR_GRP_BMS_ID | CANR_MODULE_ID2_ID )
#define BMS_MAXMIN_TEMP_ID              (CANR_FCN_DATA_ID       |CANR_GRP_BMS_ID | CANR_MODULE_ID3_ID )
#define BMS_CURRENT_ID                  (CANR_FCN_DATA_ID   |CANR_GRP_BMS_ID | CANR_MODULE_ID4_ID )
#define BMS_MAXMIN_VTG_ID               (CANR_FCN_DATA_ID   |CANR_GRP_BMS_ID | CANR_MODULE_ID5_ID )
#define BMS_TEMP_ID                             (CANR_FCN_DATA_ID   |CANR_GRP_BMS_ID | CANR_MODULE_ID6_ID )
#define BMS_VTG_ID                              (CANR_FCN_DATA_ID   |CANR_GRP_BMS_ID | CANR_MODULE_ID7_ID )


#define BMS_ANALYZE_CMD_ID              (CANR_FCN_CMD_ID        |CANR_GRP_BMS_ID |CANR_MODULE_ID0_ID)
#define BMS_ANALYZE_ACK_ID              (CANR_FCN_CMD_ID    |CANR_GRP_BMS_ID |CANR_MODULE_ID1_ID)
#define BMS_STATE_MSG_ID                (CANR_FCN_CMD_ID        |CANR_GRP_BMS_ID|CANR_MODULE_ID2_ID )

/* GLVBMS ID OVERRIDES */

#define GLVBMS_VTG_ID                   (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID0_ID )  // 0x658
#define GLVBMS_TEMP_ID                  (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID1_ID )  // 0x659
#define GLVBMS_CURRENT_ID               (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID2_ID )  // 0x65A
#define GLVBMS_MAXMIN_VAL_ID        (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID4_ID )  // 0x65C
#define GLVBMS_STATE_MSG_ID         (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID5_ID )  // 0x65D
#define GLVBMS_TOTVTG_ID                (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID6_ID )  // 0x65E
#define GLVBMS_ERROR_MSG_ID             (CANR_FCN_DATA_ID | CANR_GRP_SENS_GLVBMS_ID | CANR_MODULE_ID7_ID )  // 0x65F

/* Inverter ID OVERRIDES */

#define INVERTER_RESERVED0                      (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID7_ID)
#define INTERTER_RESERVED1                      (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID6_ID)

#define CAN_INVERTER_MOTOR_TEMP_ID              (CANR_FCN_DATA_ID | CANR_GRP_INVERTER_ID | CANR_MODULE_ID6_ID)
#define CAN_INVERTER_MOTOR_TEMP_DLC             2

#define CAN_INVERTER_DATA_ACCUMULATED_STATUS_ID         (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID5_ID)
#define CAN_INVERTER_DATA_ACCUMULATED_STATUS_DLC    4

#define CAN_INVERTER_DATA_STATUS_ID             (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID4_ID)
#define CAN_INVERTER_DATA_STATUS_DLC        4

#define CAN_INVERTER_DATA_ENCODER_POS_ID    (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID3_ID)
#define CAN_INVERTER_DATA_ENCODER_POS_DLC   8

#define CAN_INVERTER_DATA_RPM_ID            (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID2_ID)
#define CAN_INVERTER_DATA_RPM_DLC                   4

#define CAN_INVERTER_DATA_CURRENT_ID        (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID1_ID)
#define CAN_INVERTER_DATA_CURRENT_DLC       8

#define CAN_INVERTER_DATA_VOLTAGE_ID        (CANR_FCN_DATA_ID       |CANR_GRP_INVERTER_ID | CANR_MODULE_ID0_ID)
#define CAN_INVERTER_DATA_VOLTAGE_DLC       4


/* Telemetry ID OVERRIDES */

#define TELEMETRY_TO_INVERTER_DATA_ID (CANR_FCN_DATA_ID | CANR_GRP_TELEMETRY_ID | CANR_MODULE_ID3_ID)

/* ECU OVERRIDES */

#define ECU_ALIVE_ID                                                                    CANR_FCN_DATA_ID |
CANR_GRP_DASH_ID | CANR_MODULE_ID7_ID

#define ECU_PLAY_RTDS_DATA                                                                      0x1
#define ECU_PLAY_RTDS_ID                                                                        CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID0_ID
#define ECU_PLAY_RTDS_DLC                                                                       1

#define ECU_DRIVE_ENABLED_DATA                                                          0x2
```

```
#define ECU_DRIVE_ENABLED_ID                                         CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID0_ID
#define ECU_DRIVE_ENABLED_DLC                                        1

#define ECU_DRIVE_DISABLED_DATA                                      0x3
#define ECU_DRIVE_DISABLED_ID                                        CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID0_ID
#define ECU_DRIVE_DISABLED_DLC                                       1

#define ECU_PRAMETER_ADJUSTMENT_CONFIRMED_DATA              0x0
#define ECU_PRAMETER_ADJUSTMENT_CONFIRMED_ID                CANR_FCN_BOOT_ID | CANR_GRP_ECU_ID |
CANR_MODULE_ID1_ID
#define ECU_PRAMETER_ADJUSTMENT_CONFIRMED_DLC               0

#define ECU_LAUNCH_CONTROL_REQUEST_CONFIRMED_DATA       0x0F
#define ECU_LAUNCH_CONTROL_REQUEST_CONFIRMED_ID             CANR_FCN_BOOT_ID | CANR_GRP_ECU_ID |
CANR_MODULE_ID2_ID
#define ECU_LAUNCH_CONTROL_REQUEST_CONFIRMED_DLC        1

#define ECU_LAUNCH_CONTROL_ARMED_DATA                           0xF0
#define ECU_LAUNCH_CONTROL_ARMED_ID                                  CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID2_ID
#define ECU_LAUNCH_CONTROL_ARMED_DLC                            1

#define ECU_LAUNCH_CONTROL_DISARMED_DATA                    0x00
#define ECU_LAUNCH_CONTROL_DISARMED_ID                          CANR_FCN_BOOT_ID | CANR_GRP_ECU_ID |
CANR_MODULE_ID2_ID
#define ECU_LAUNCH_CONTROL_DISARMED_DLC                         1

#define ECU_TORQUE_REQUEST_DATA                                 0x0
#define ECU_TORQUE_REQUEST_ID                                        CANR_FCN_PRI_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID3_ID
#define ECU_TORQUE_REQUEST_DLC                                       4

#define ECU_TRACTION_CONTROL_ON_DATA                            0x0F
#define ECU_TRACTION_CONTROL_ON_ID                                   CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID4_ID
#define ECU_TRACTION_CONTROL_ON_DLC                                  1


#define ECU_TRACTION_CONTROL_OFF_DATA                           0xF0
#define ECU_TRACTION_CONTROL_OFF_ID                                  CANR_FCN_BOOT_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID4_ID
#define ECU_TRACTION_CONTROL_OFF_DLC                            1

#define ECU_CURRENT_MAX_DATA                                         0
#define ECU_CURRENT_MAX_ID                                                CANR_FCN_DATA_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID0_ID
#define ECU_CURRENT_MAX_DLC                                               4

#define ECU_CURRENT_ERROR_DATA                                       0
#define ECU_CURRENT_ERROR_ID                                         CANR_FCN_DATA_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID1_ID
#define ECU_CURRENT_ERROR_DLC                                        4

#define ECU_CURRENT_OUTPUT_DATA                                      0
#define ECU_CURRENT_OUTPUT_ID                                        CANR_FCN_DATA_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID2_ID
#define ECU_CURRENT_OUTPUT_DLC                                       4

#define ECU_CURRENT_INTEGRAL_DATA                                    0
#define ECU_CURRENT_INTEGRAL_ID                                      CANR_FCN_DATA_ID |
CANR_GRP_ECU_ID | CANR_MODULE_ID3_ID
#define ECU_CURRENT_INTEGRAL_DLC                                4

#define ECU_CURRENT_CONTROLLER_STATE_DATA                   0
#define ECU_CURRENT_CONTROLLER_STATE_ID                         CANR_FCN_DATA_ID | CANR_GRP_ECU_ID |
CANR_MODULE_ID4_ID
#define ECU_CURRENT_CONTROLLER_STATE_DLC                    1

#define ECU_CURRENT_IMPLAUSIBILITY_DATA                         0
#define ECU_CURRENT_IMPLAUSIBILITY_ID                           CANR_FCN_DATA_ID | CANR_GRP_ECU_ID |
CANR_MODULE_ID5_ID
#define ECU_CURRENT_IMPLAUSIBILITY_DLC                          1


//------------------------------------------------------------------------------
/*                  ------- Byte mapping                                        -------------------------------*/
/* If you want to describe the mapping of your can bus data, make macros here  */
//------------------------------------------------------------------------------

/* Inverter data, sent from telemetry (analyze) */
#define STATUS_BIT_FLIP_REQUEST_COMMAND 0x01
#define CONTROLLER_PARAMETER_P_SET_REQUEST_COMMAND 0x11
#define CONTROLLER_PARAMETER_I_SET_REQUEST_COMMAND 0x12
#define CONTROLLER_PARAMETER_D_SET_REQUEST_COMMAND 0x13
#define TORQUE_REQUEST_COMMAND 0x41

//GLVBMS ERROR DATA
#define GLVBMS_ERROR_NOERROR                    0x00
#define GLVBMS_ERROR_OVERCURRENT                0x01
#define GLVBMS_ERROR_OVERVOLTAGE0x02
#define GLVBMS_ERROR_OVERTEMP                   0x03
#define GLVBMS_ERROR_REQUESTED                  0x04

//-----------------------------------------------------------------------------------

#endif /* REVOLVE_CAN_DEFINITIONS_H_ */
```

C. Inverter CAN Messages and Errors

In table C.1 the CAN messages that are sent continuously from the VSI during operation are listed. These are sensor data reporting internal variables from the motor controller. These messages contain the calculated rotor speed, the measured phase currents, the calculated dq-currents and the measured dc-bus voltage.

	Direction:	Data type:	Size [bits]:	Priority:
Speed_RPM	out	float	32	medium
Current_A	out	float	32	medium
Current_B	out	float	32	medium
Current_C	out	float	32	medium
Current_Q	out	float	32	medium
Current_D	out	float	32	medium
Voltage_DC	out	uint	16	high

Table C.1: CAN messages sent continuously during operation from the VSI.

In table C.2 the CAN messages that reports the states of the VSI are listed. The fault vector is a vector containing information about the different possible faults in the VSI. This is information like permanent phase current limits exceeded, permanent upper voltage limit exceeded, transient phase current limits exceeded, transient upper voltage limit exceeded, speed limit exceeded, gate driver fault, lower voltage limit exceeded, lower torque request limit exceeded, motor temperature limit exceeded or heat sink temperature limit exceeded. The initialization complete message tells if the encoder is initialized (the Z-line is found) and the shutdown circuit message tells if the shutdown circuit is active.

	Direction:	Data type:	Size [bits]:	Priority:
Fault vector	out	uint	40	high
Initialization complete	out		0	high
Active shutdown circuit	out		0	high

Table C.2: CAN messages reporting the states of the VSI.

Table C.3 shows the CAN messages giving the set points for the dq-current controllers in the VSI. In field oriented control without field weakening the d-current set point will always be zero, while the d-current set point will be proportional to the requested electromagnetic output torque.

	Direction:	Data type:	Size [bits]:	Priority:
Setpoint_Q	in	float	32	high
Setpoint_D	in	float	32	high

Table C.3: CAN messages with set points for the dq-current controllers.

In table C.4 the input CAN messages giving the PID controller values are shown. By sending CAN messages with new values for the p (proportional), i (integral), d (derivative) values, the dq-current controllers can be tuned. The student developed telemetry software Revolve Analyze is used to send new tuning parameters to the current controllers during testing and setup.

	Direction:	Data type:	Size [bits]:	Priority:
PID_P_Q	in	float	32	low
PID_I_Q	in	float	32	low
PID_D_Q	in	float	32	low
PID_P_D	in	float	32	low
PID_I_D	in	float	32	low
PID_D_D	in	float	32	low

Table C.4: CAN messages with tuning parameters for the current controllers in the VSI.

The parameters in table C.5 gives the voltage levels where the torque request will be limited. The parameters are used to tune the torque limitation based on the transient voltage limits so that the behavior of the motor controller gets smooth for the driver. When the motor is generating power during braking, the DC bus voltage tends to increase. It is therefore important to limit the torque request so that the voltage limit of 600 V is never exceeded for more than a moving average of 500 ms or a total of 100 ms. If this happens in one of the Formula Student competitions, the team will be disqualified.

	Direction:	Data type:	Size [bits]:	Priority:
VDC_H_LIM_START	in	float	32	low
VDC_H_LIM_END	in	float	32	low
VDC_L_LIM_START	in	float	32	low
VDC_L_LIM_END	in	float	32	low
VDC_HYSTERESIS	in	float	32	low

Table C.5: CAN messages for tuning voltage levels limiting torque request.

The parameters in table C.6 are used to tune the torque request based on power limits, in the same way as the voltage levels from table C.5. If more than 80 kW of power from the battery accumulator is spent for more than a moving average of 500 ms or for a total of 100 ms, the team will be disqualified from the Formula Student competitions. The organizers of the competitions supply the competing teams with an energy meter to measure this.

	Direction:	Data type:	Size [bits]:	Priority:
P_H_LIM_START	in	float	32	low
P_H_LIM_END	in	float	32	low
P_L_LIM_START	in	float	32	low
P_L_LIM_END	in	float	32	low
P_HYSTERESIS	in	float	32	low

Table C.6: CAN messages for tuning power limiting of the torque request.

The parameters in table C.7 are used to tune the torque request based on speed limits of the motor, in the same way as the voltage levels from table C.5 and the power limits from table C.6. The main reason why this is done is to make sure the rotational speed of the motor never exceeds the maximum rpm-rating for the motor.

	Direction:	Data type:	Size [bits]:	Priority:
N_H_LIM_START	in	float	32	low
N_H_LIM_END	in	float	32	low
N_L_LIM_START	in	float	32	low
N_L_LIM_END	in	float	32	low
N_HYSTERESIS	in	float	32	low

Table C.7: CAN messages for tuning speed limitation of the torque request.

The parameters in table C.8 are used to tune the torque request based on temperature limits from the heat sink temperature measurements and from the motor temperature measurements, in the same way as before. The main reason for doing this is to make sure the motor and the IGBT modules never overheats.

	Direction:	Data type:	Size [bits]:	Priority:
T_H_LIM_START	in	float	32	low
T_H_LIM_END	in	float	32	low
T_L_LIM_START	in	float	32	low
T_L_LIM_END	in	float	32	low
T_HYSTERESIS	in	float	32	low

Table C.8: CAN messages for tuning temperature limitation of the torque request.

D. Inverter Prototype Setup Manual

IGBTs to gate drivers:

IGBT A should be connected with gate driver A, IGBT B with gate driver B and so on. Looking from above with the capacitors on top, IGBT A is the one to the right, IGBT B the one in the middle and IGBT C the one to the left, while gate driver A is the first one, gate driver B the one in the middle and gate driver C the one on top. This can be seen in figure D.1.

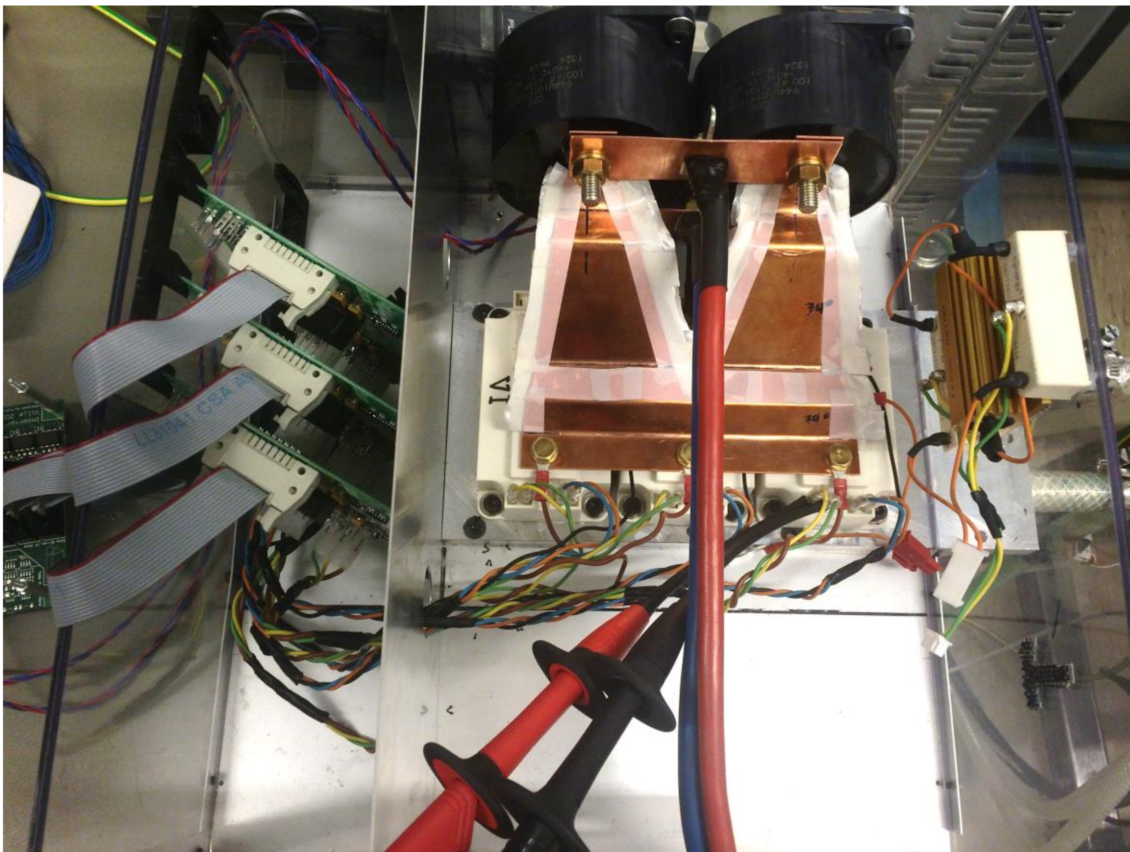

Figure D.1: Voltage source inverter prototype setup.

Next, it is important to make sure the contacts are connected correctly to the IGBT modules. As is shown in figure D.2, yellow cable is the first (pin 4 on the IGBT), green the second (pin 5 on the IGBT), blue the third (pin 7 on the IGBT), orange the fourth (pin 6 on the IGBT).

Brown cable should be connected to DC+ (pin 3 on the IGBT) and black cable to AC (pin 1 on the IGBT).

Pin number:	Function:
1	AC
2	DC-
3	DC+
4	Upper gate
5	Upper emitter
6	Lower gate
7	Lower emitter

Table D.1: Pin connection on the IGBT modules.

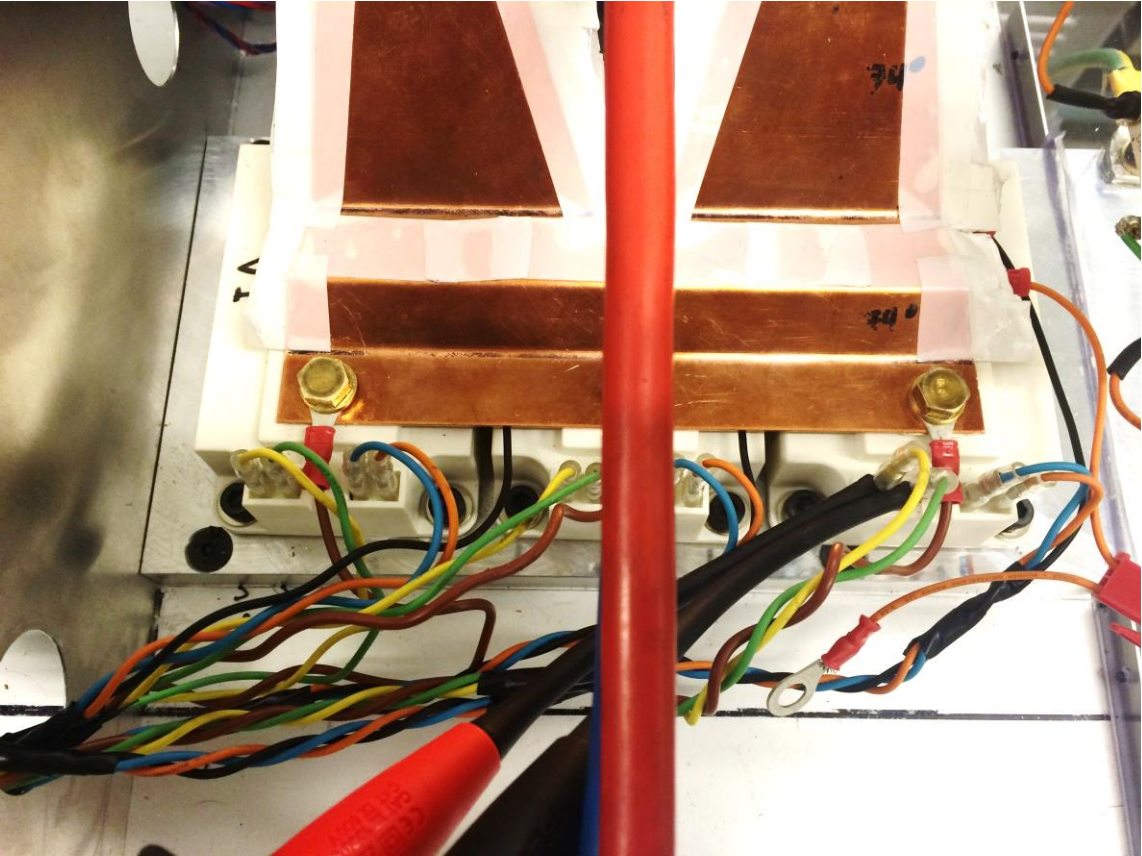

Figure D.2: Pin connections on the IGBT modules.

On the gate driver, the upper contact (to the right in figure D.3) is the one with the yellow, green and brown cables, while the lower contact is the one with the orange, blue and black cables.

Figure D.3: Gate driver connections.

Control board:

First, the gate drivers should be connected to the right headers on the control board. Gate driver A should be connected to the first header, gate driver B to the middle and gate driver C to the last one. In figure D.4 these contacts are shown, where the first one is the one to the outer left in the figure.

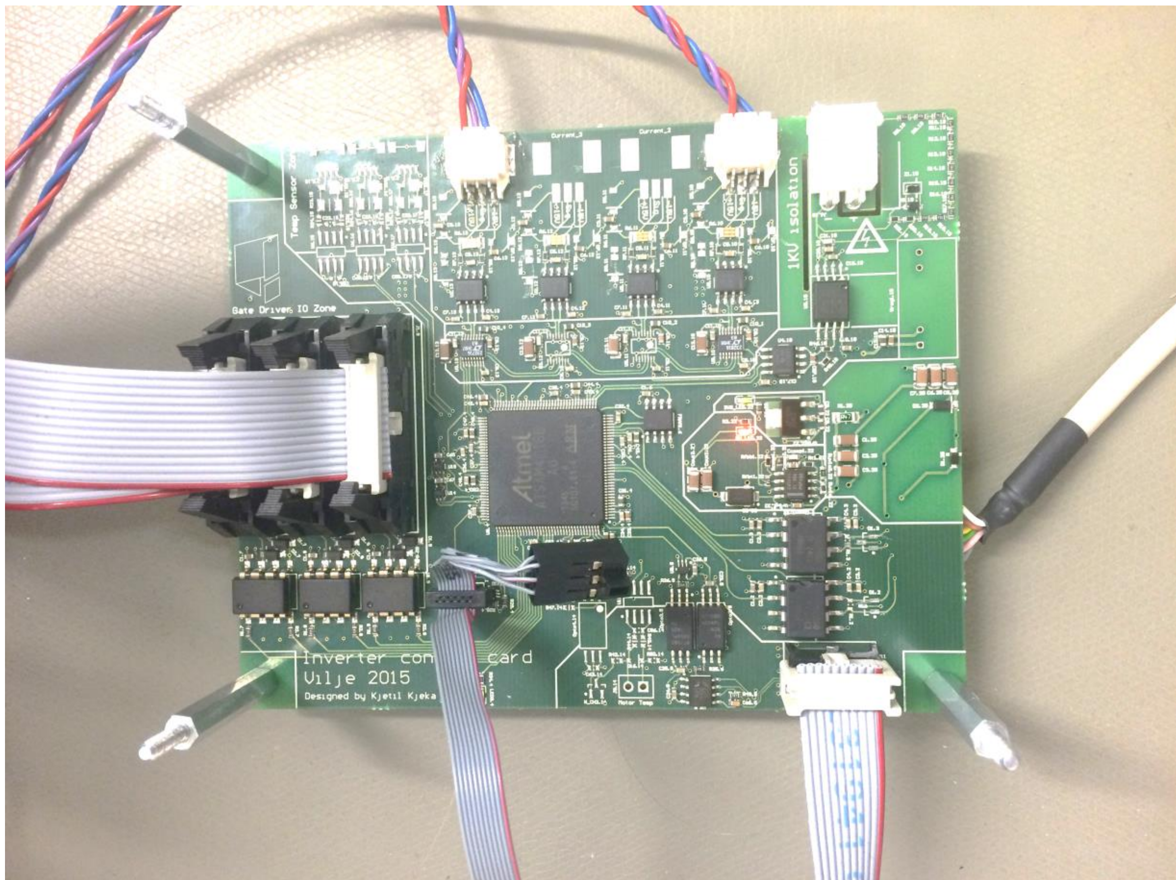

Figure D.4: Control circuit board connections.

Then, the current sensors should be connected. The current sensor on phase A should go to current sensor header in position 1, while the current sensor on phase B to position 4 (this is labeled on the PCB).

Next, the encoder, the programming contact and CAN/power should be connected. CAN/power is easy (no wrong way to put it), but the programming contact and encoder should be connected as shown in figure D.4.

Motor:

It is important to connect the motor phases correctly! In figure D.5, phase A is the upper one (parallel to the water inlet/outlet), phase B in the middle and phase C the lower one.

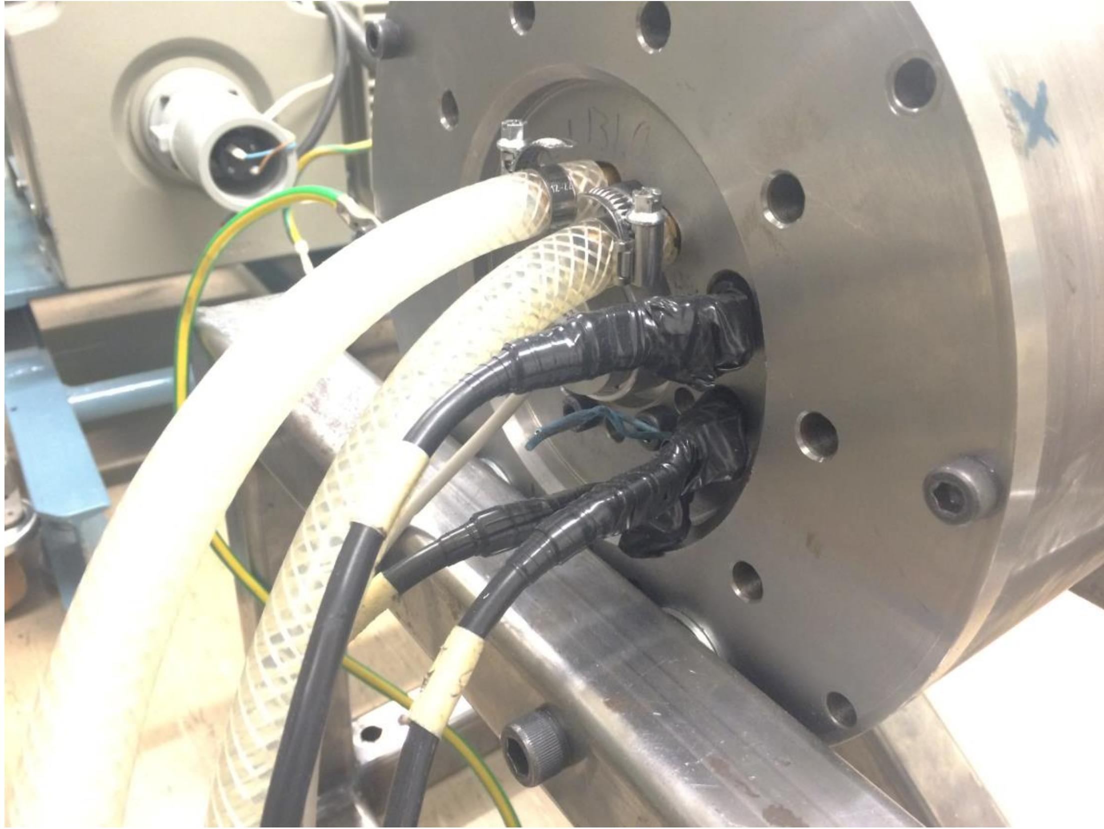

Figure D.5: Motor phases connection.

Encoder:

To connect the encoder correctly, the color labeled wires in the encoder cable should be connected to the right pins on the control board. In the following image these wires have been connected to a ribbon cable leading to the control board.

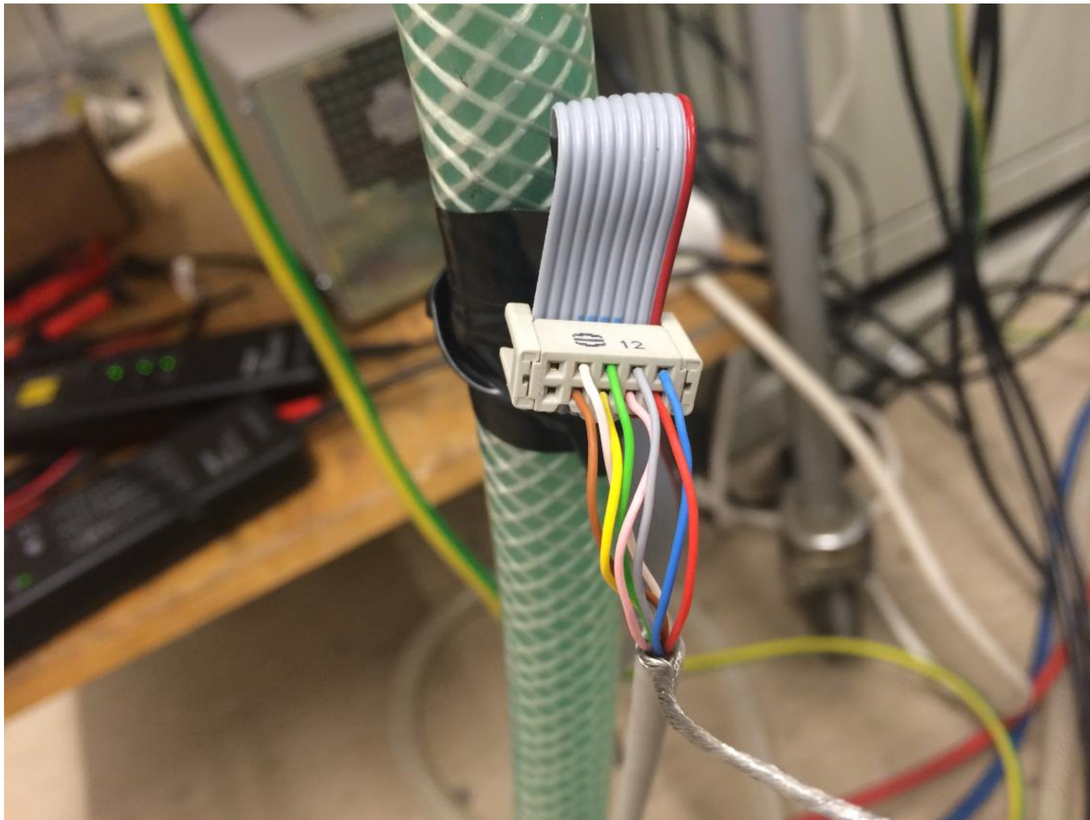

Figure D.6: Encoder connection.

Pin number:	Color coding:	Function:
1	Blue	0V
2	Red	5V
3	Grey	A+
4	Pink	A-
5	Green	B+
6	Yellow	B-
7	White	Data+
8	Brown	Data-

Table D.2: Pin connection for the encoder.

E. Simulink Model of the Battery Accumulator

In figure E.1 a Simulink model of the battery accumulator is shown. It contains the eight sub blocks shown in figure E.2, which represent the eight battery modules in the accumulator. Each of these modules contain the battery cell configuration shown in figure E.3, with 18 series connected, parallel connected battery cells. In each of the sub blocks in figure E.3, two $LiCoO_2$ cells are connected in parallel as shown in figure E.4. The battery cell models in figure E.4 are models from the SimPowerSystems library in Simulink. The applied block parameter settings for the models are shown in figure E.5.

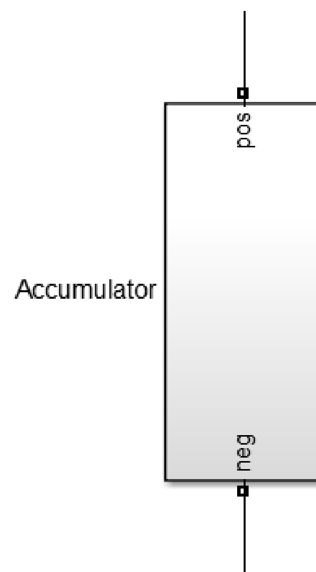

Figure E.1: Battery accumulator model in Simulink.

Figure E.2: Battery module configuration in Simulink.

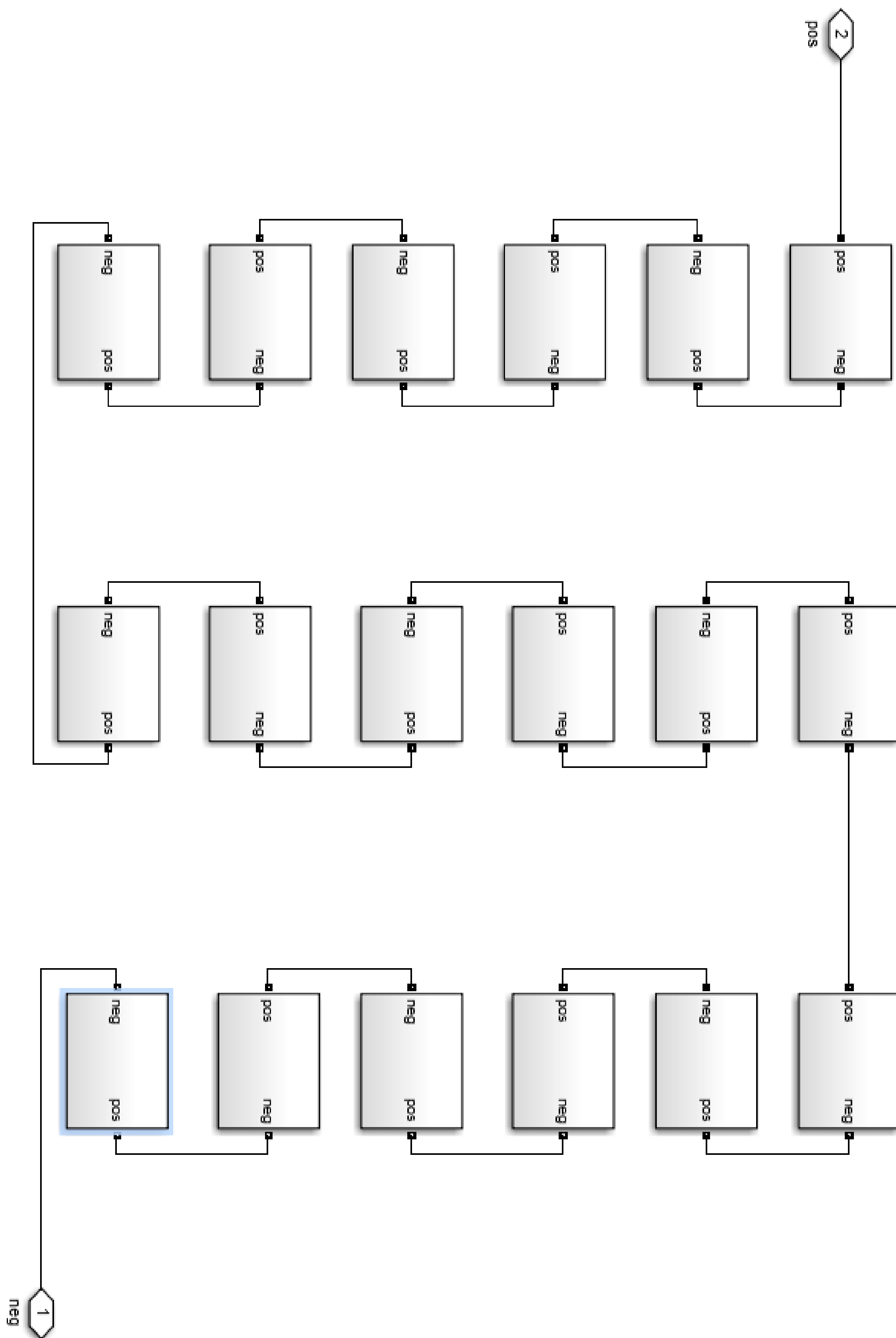

Figure E.3: Battery cell configuration in a battery module in Simulink.

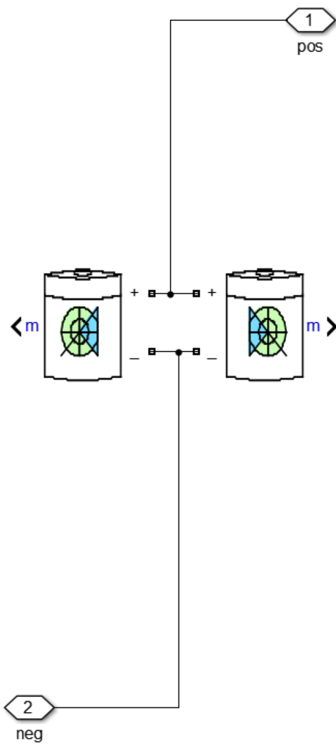

Figure E.4: Li-ion battery cell model in Simulink.

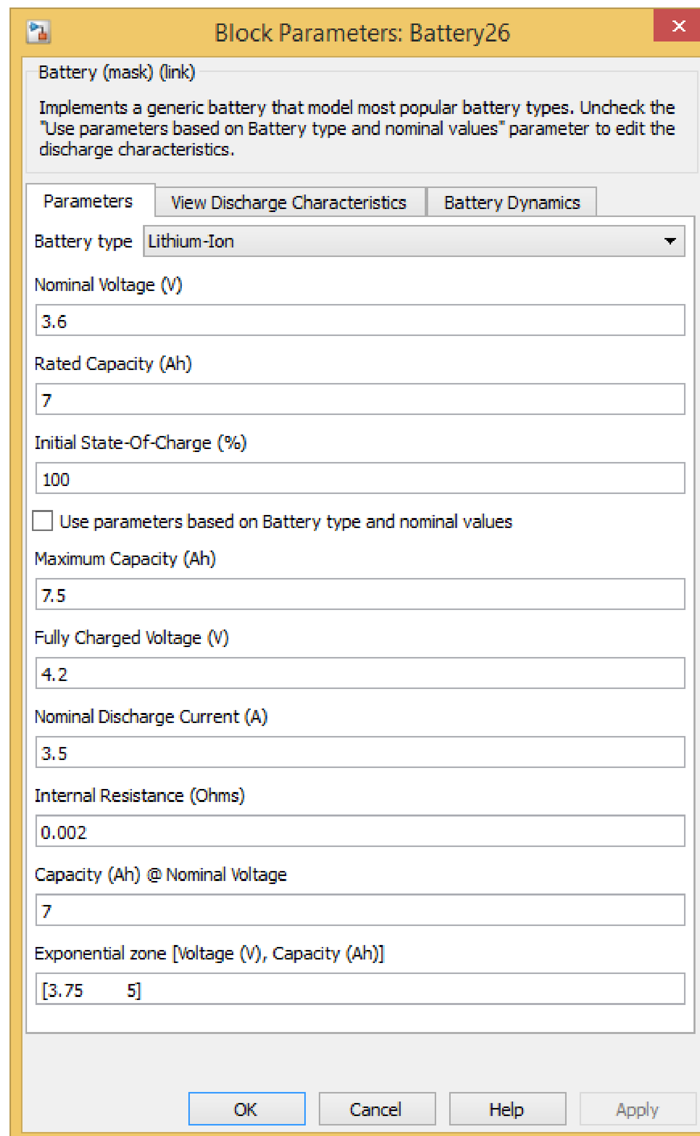

Figure E.5: Block parameter settings for Li-ion battery cell model in Simulink.

The use of this battery accumulator in the VSI model, made the overall model to large and complex to simulate. The model was therefore not used in this thesis work.

F. Simulink Control System Models

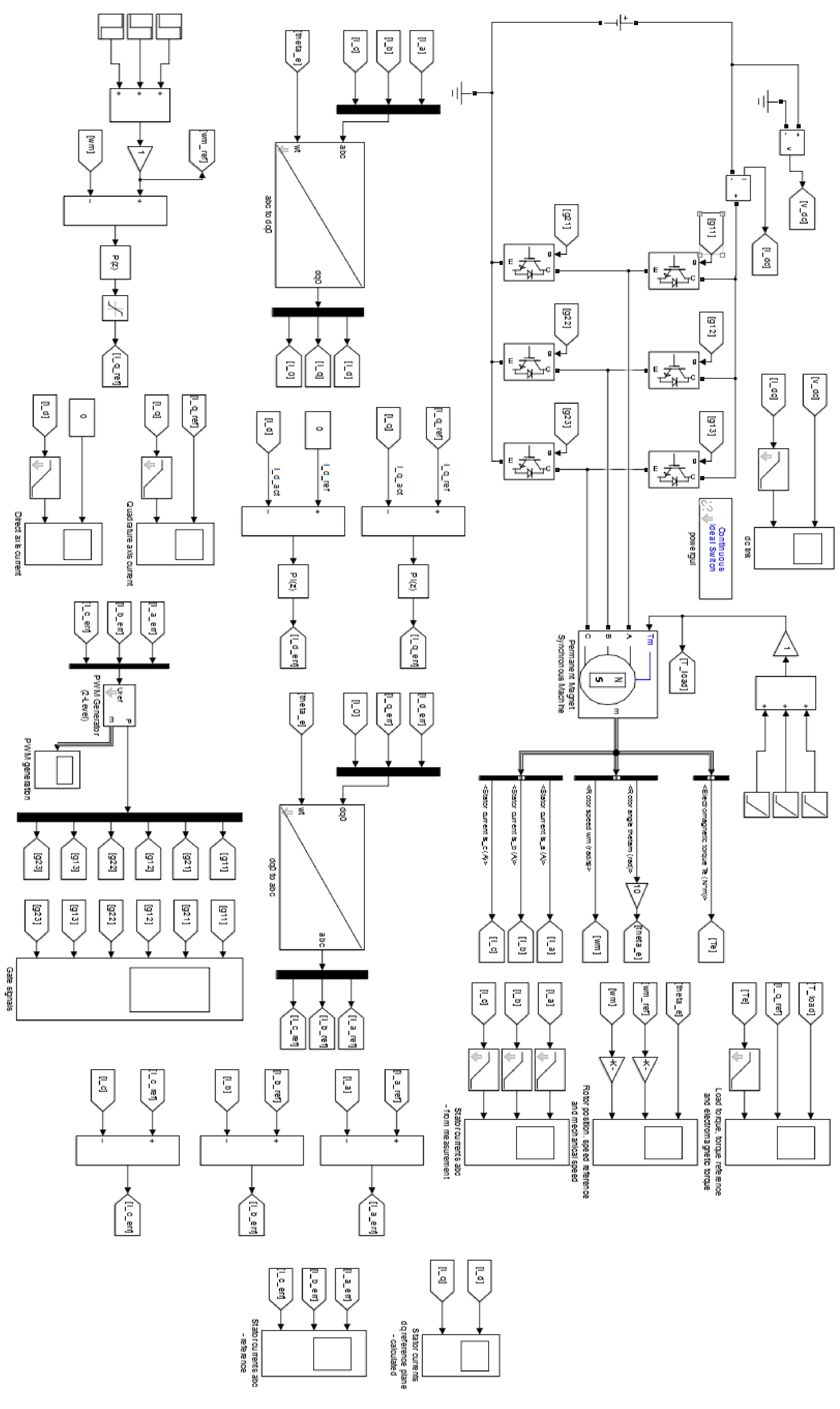

Figure F.1: Simulink model of field oriented control with PWM.

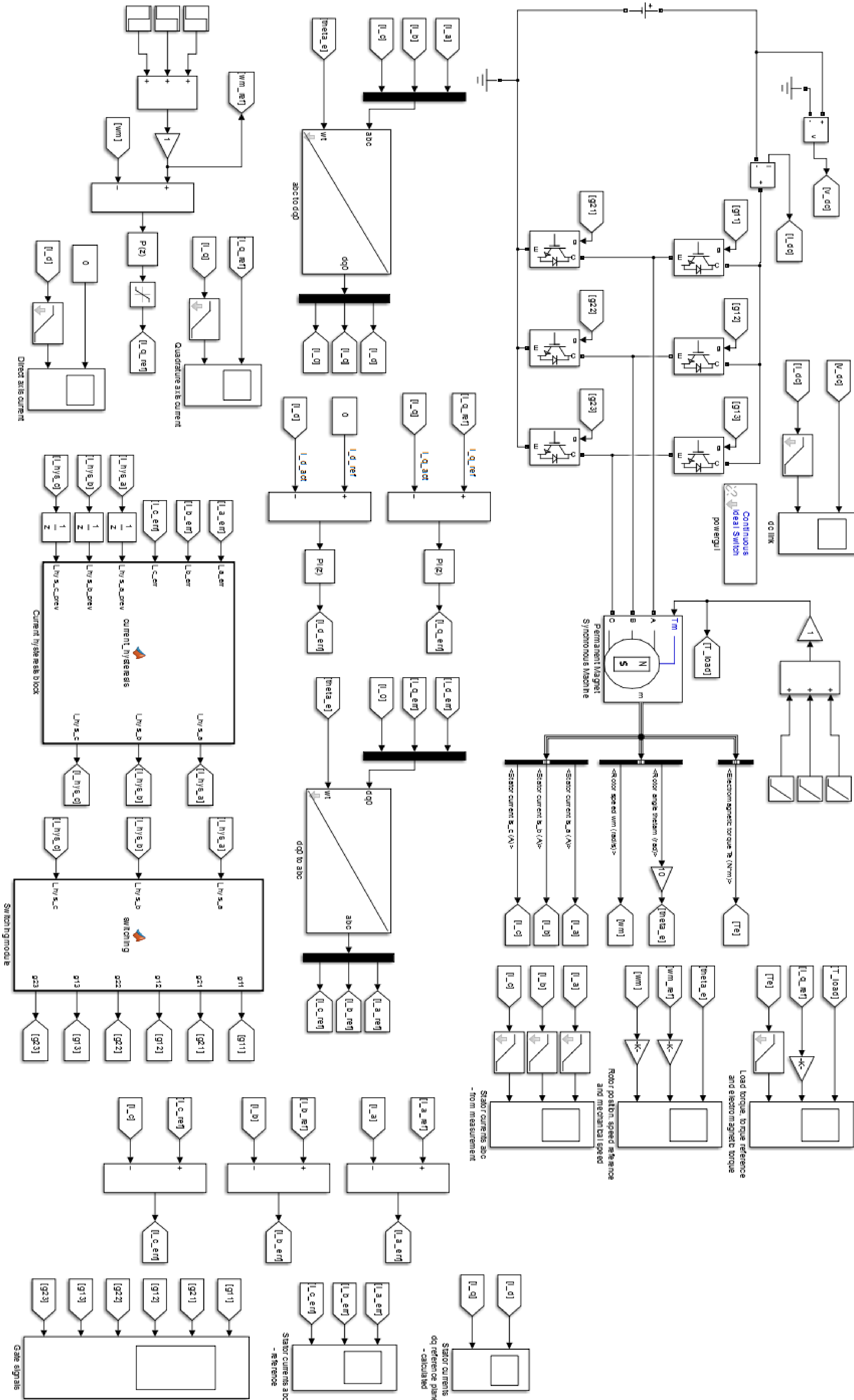

Figure F.2: Simulink model of field oriented control with hysteresis.

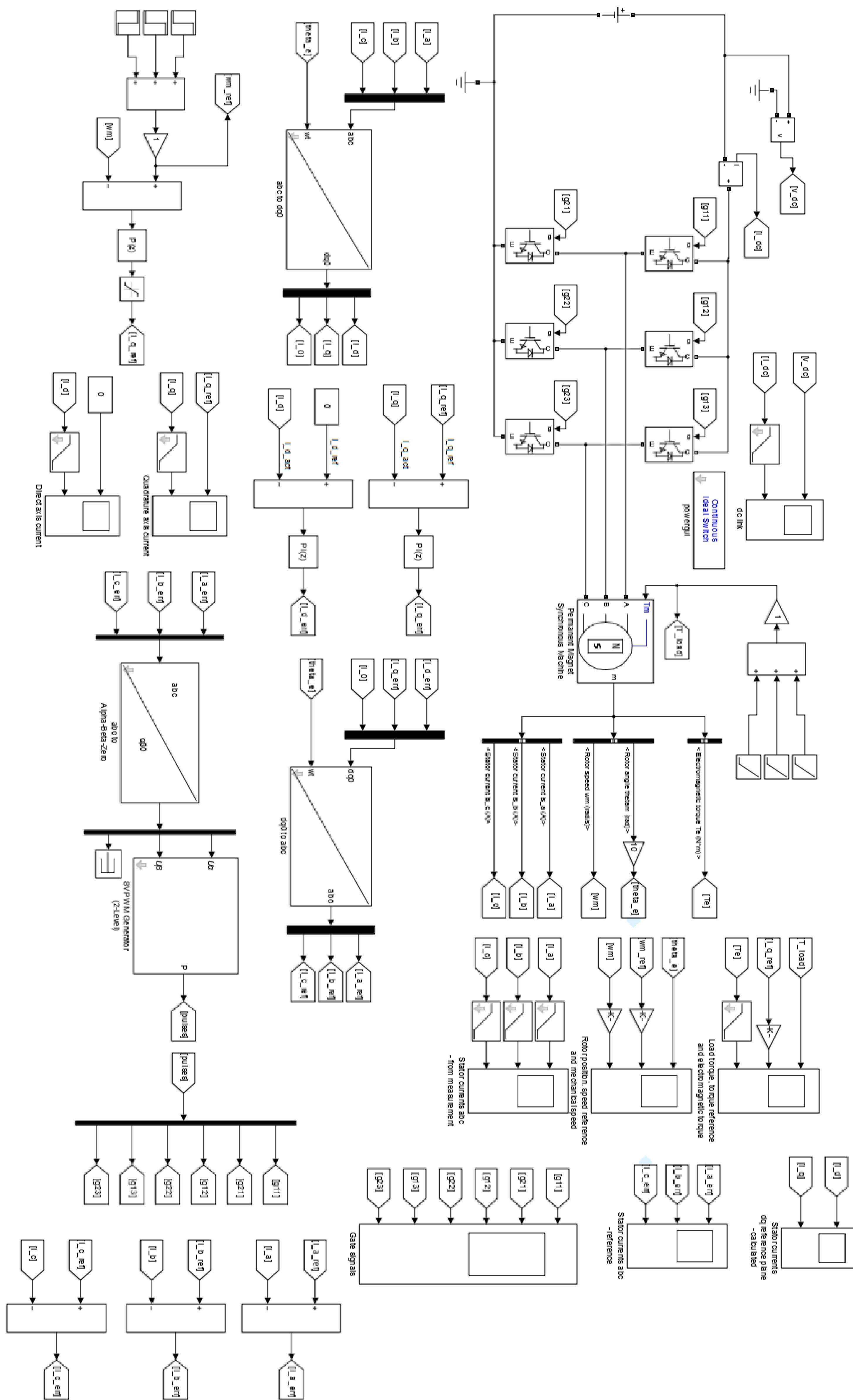

Figure F.3: Simulink model of field oriented control with SV PWM.

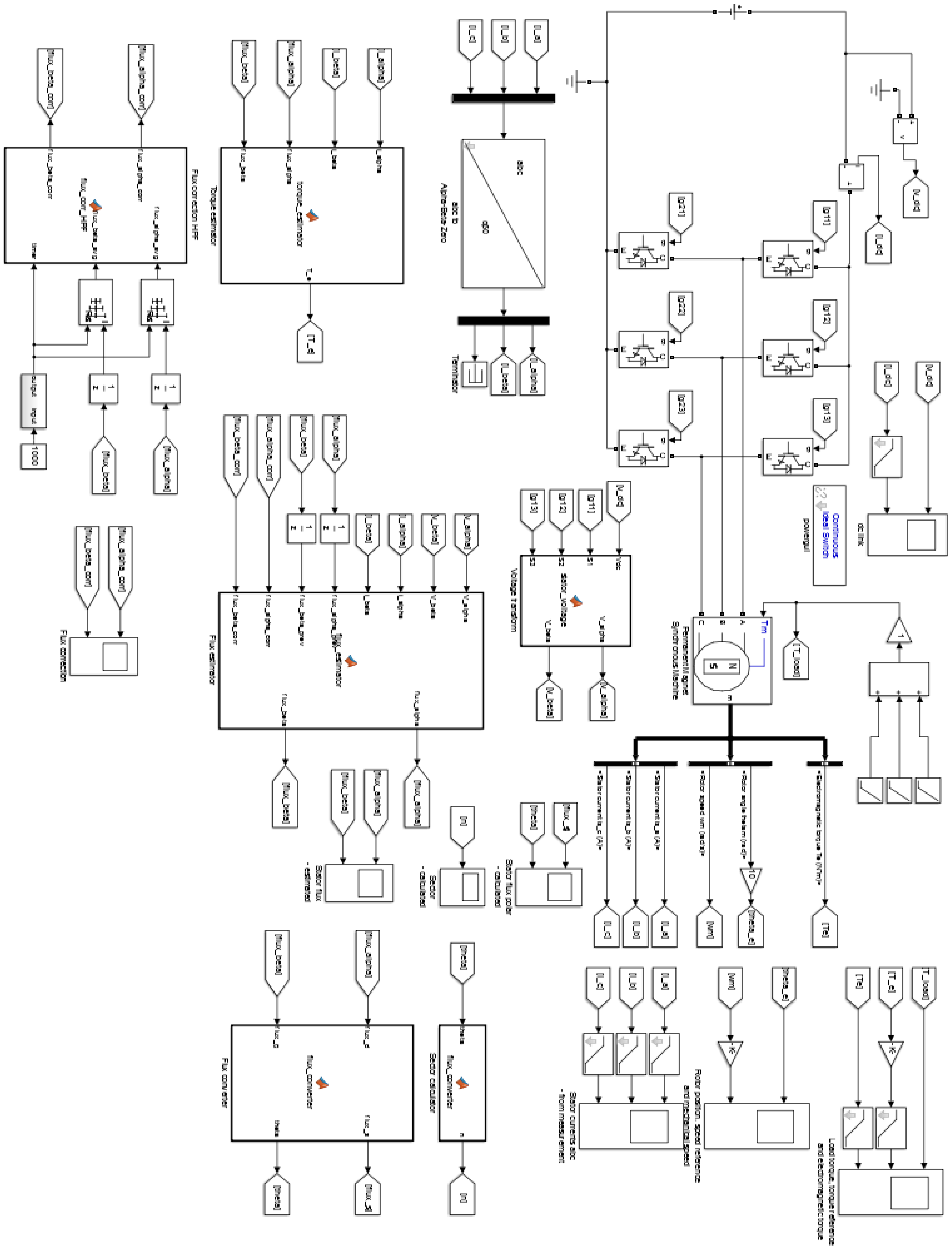

Figure F.4: Simulink model for direct torque control flux estimation.

G. Voltage Source Inverter Wiring Harness

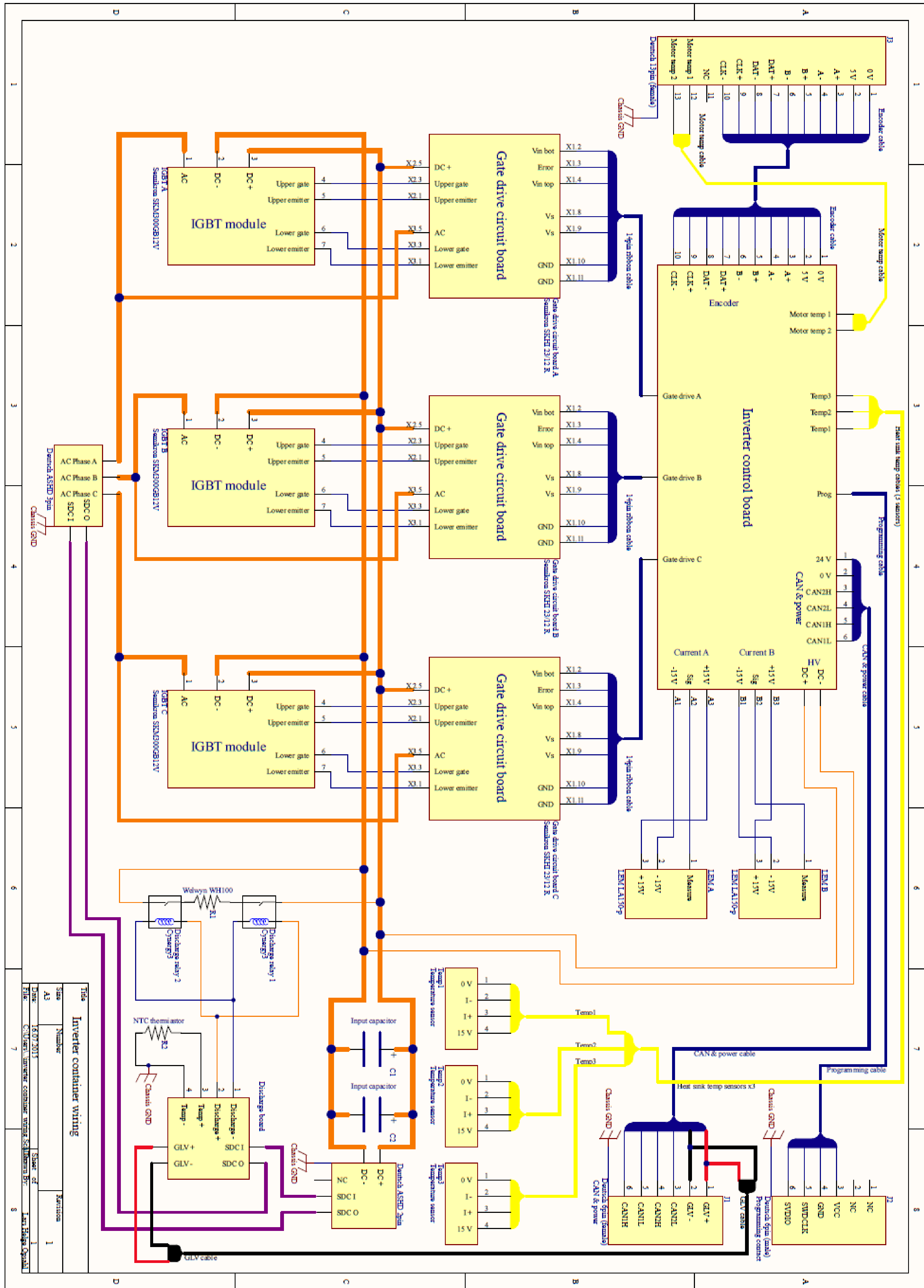

Figure G.1: Voltage source inverter wiring harness.

H. Schematic Drawings

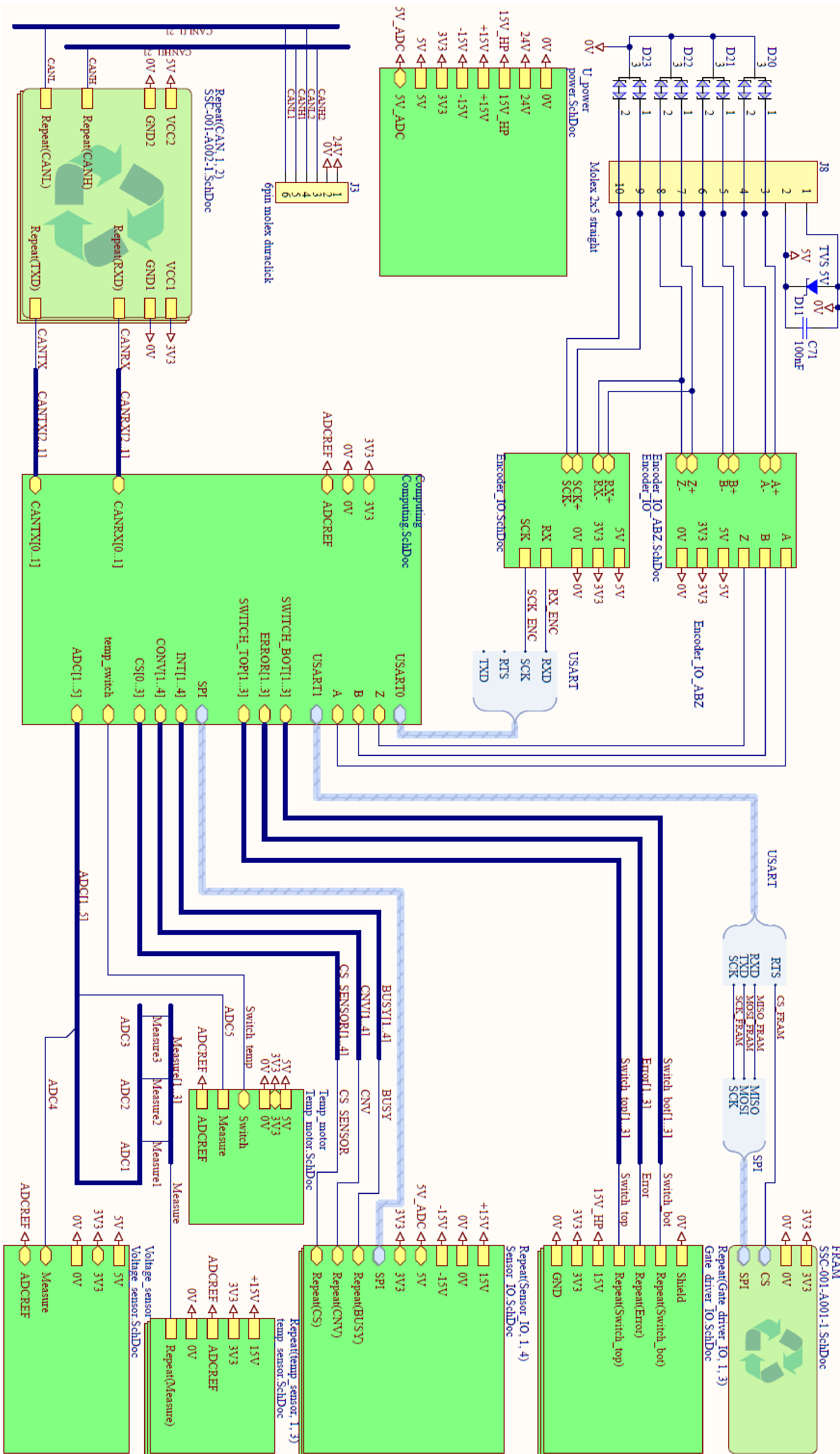

Figure H.1: Schematic drawing of the inverter control board.

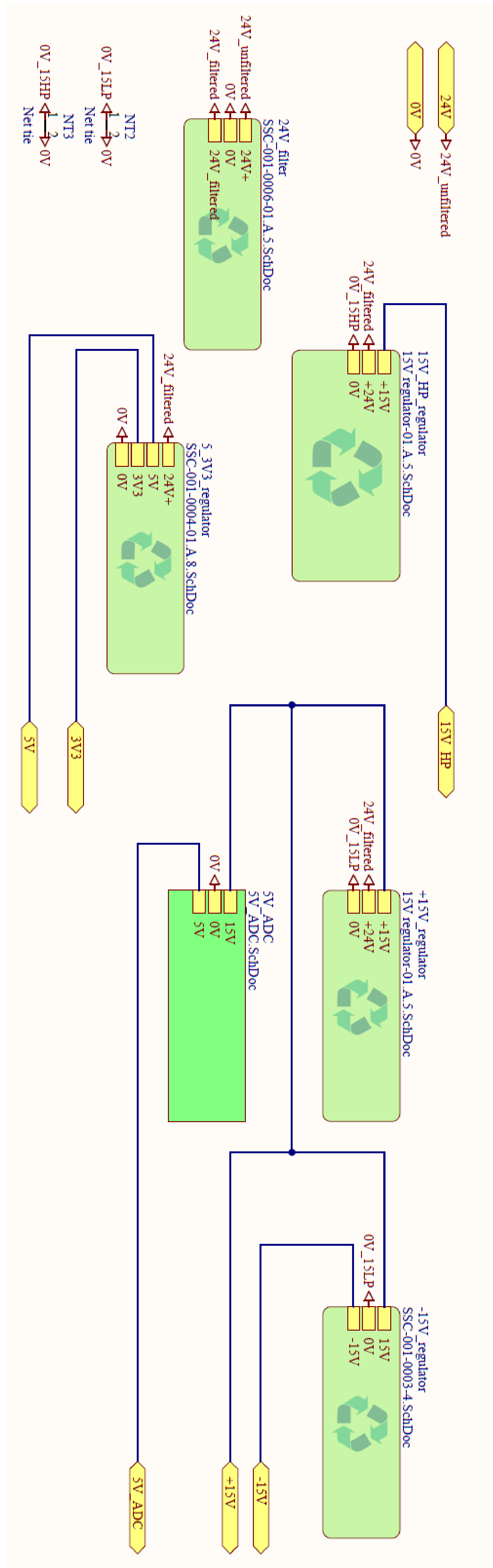

Figure H.2: Schematic drawing of the power modules.

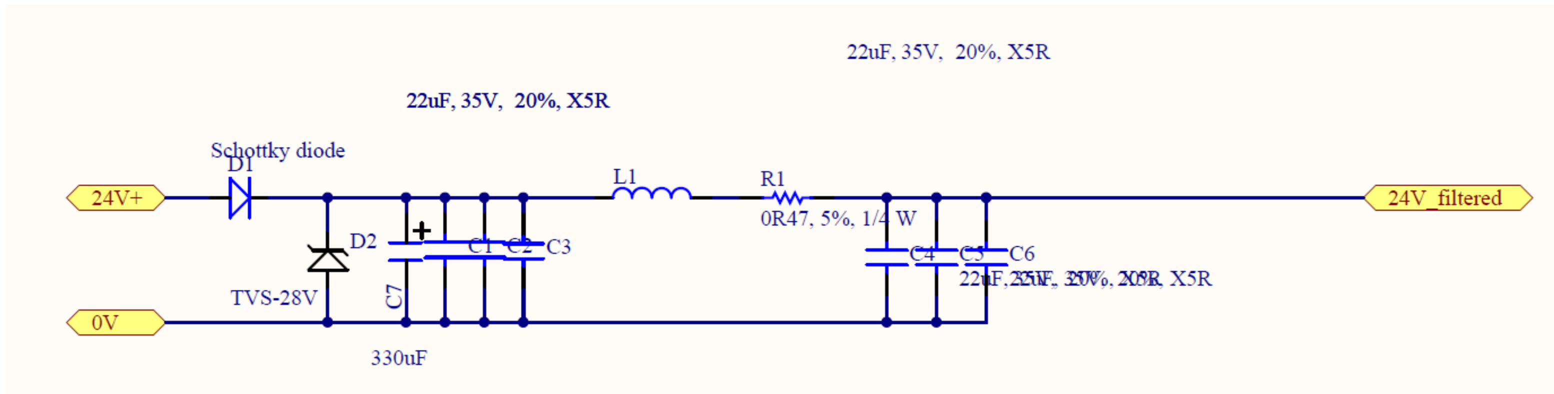

Figure H.3: Schematic drawing of the 24V input filter.

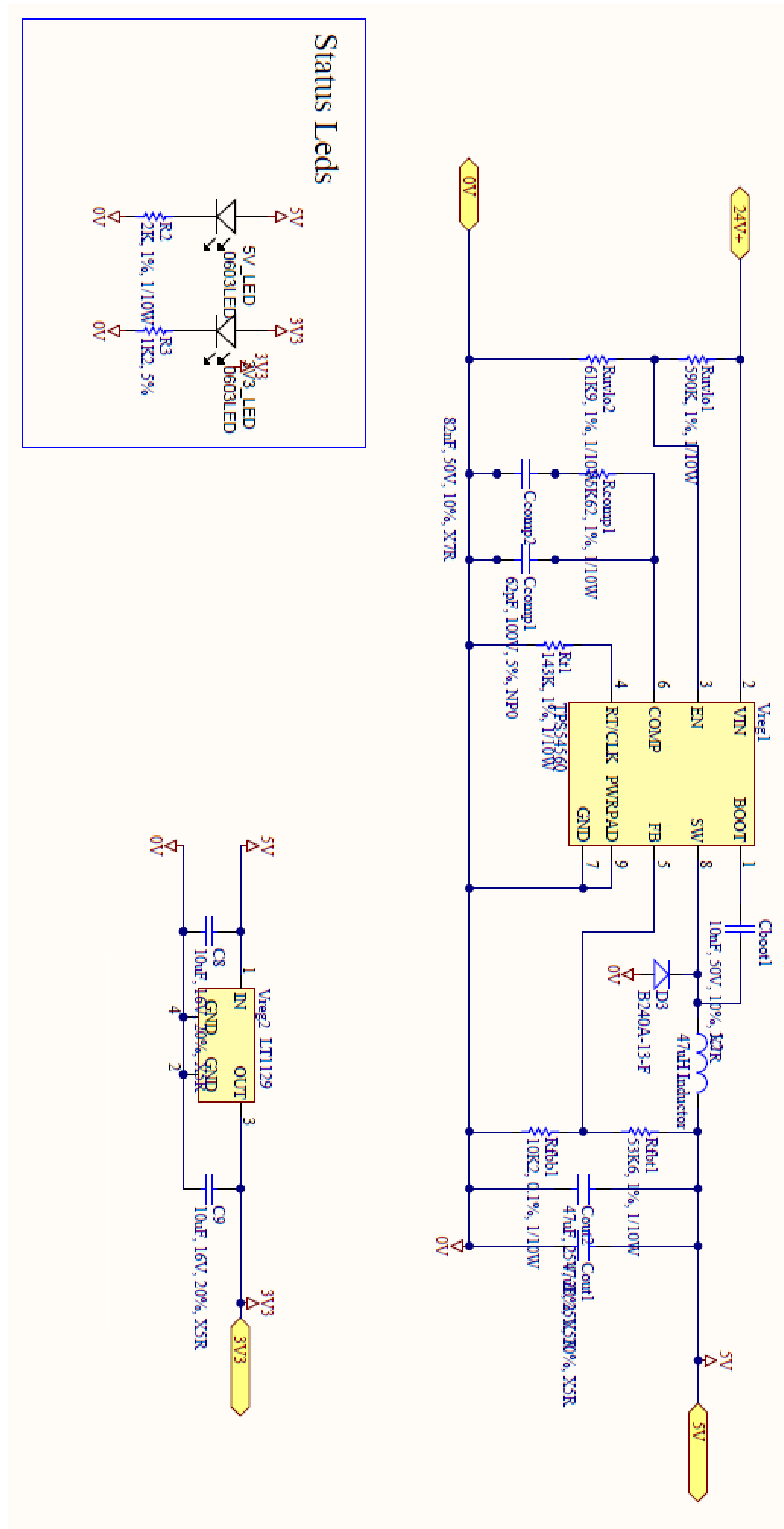

Figure H.4: Schematic drawing of the 5V and 3V3 supply.

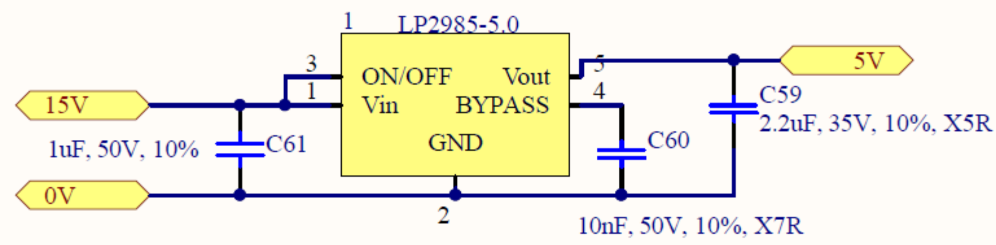

Figure H.5: Schematic drawing of the 5V ADC circuit.

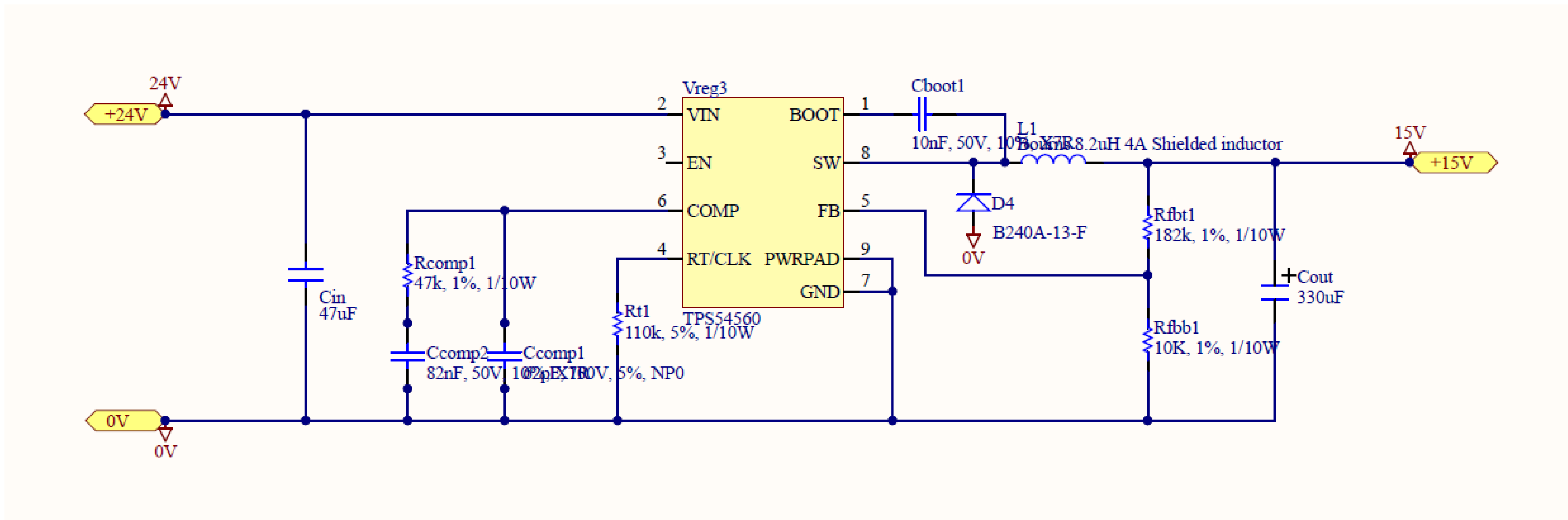

Figure H.6: Schematic drawing of the 15V supply.

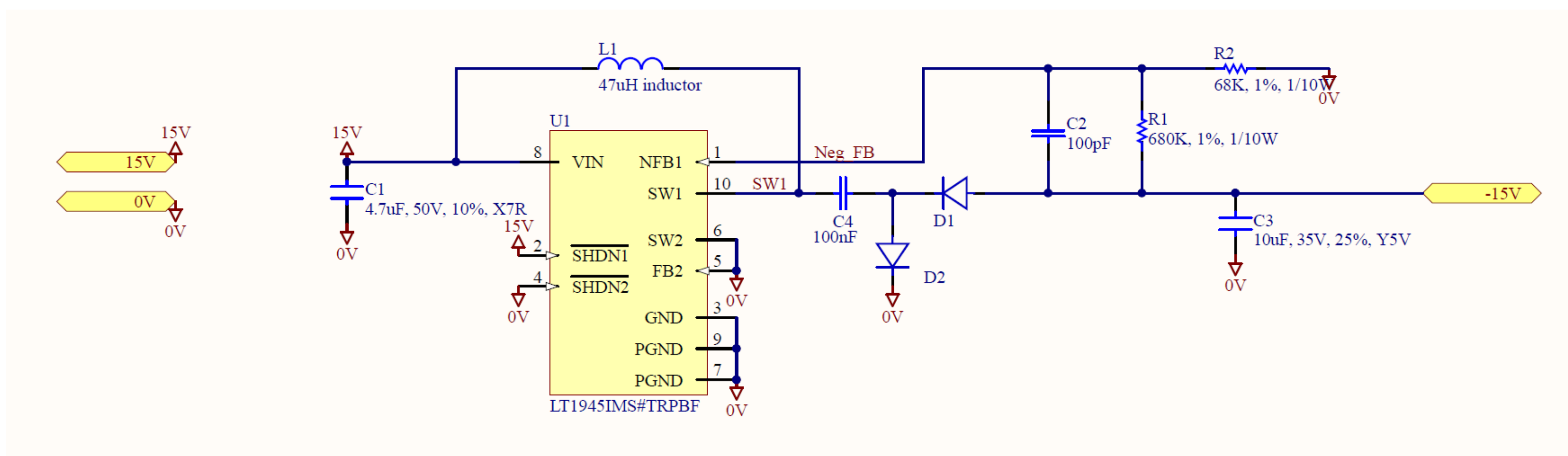

Figure H.7: Schematic drawing of the negative 15V supply.

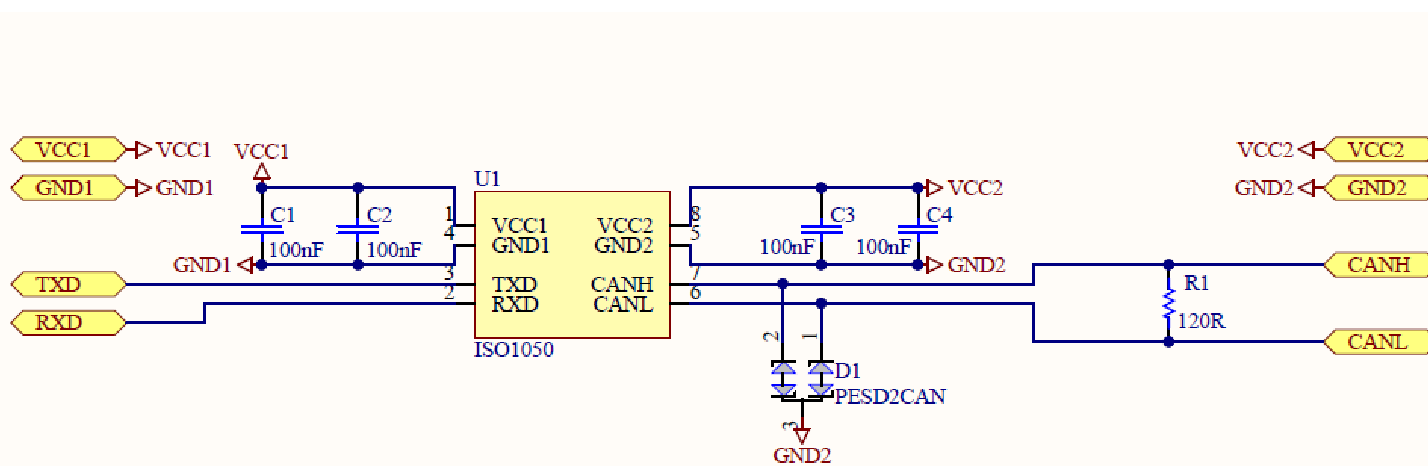

Figure H.8: Schematic drawing of the CAN transceiver circuit.

Figure H.9: Schematic drawing of the computational circuitry.

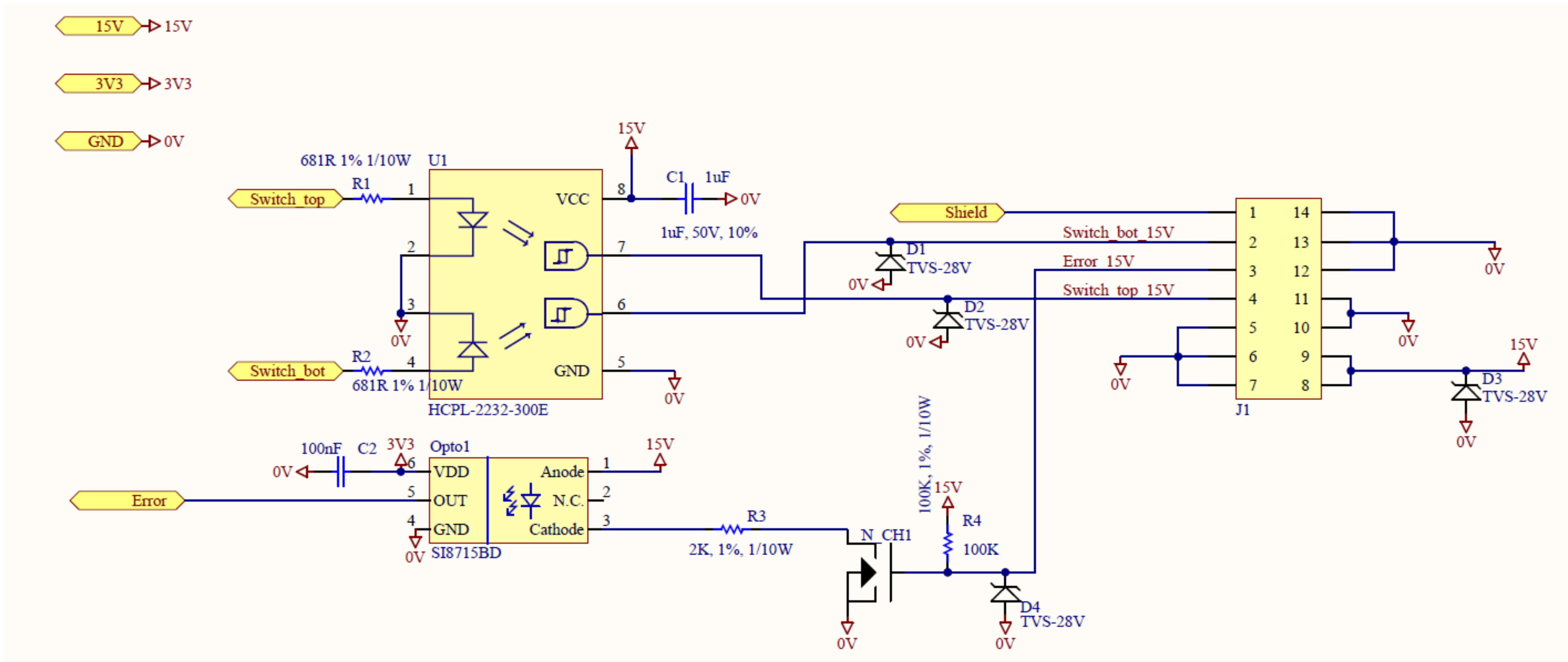

Figure H.10: Schematic drawing of the gate driver interface circuit.

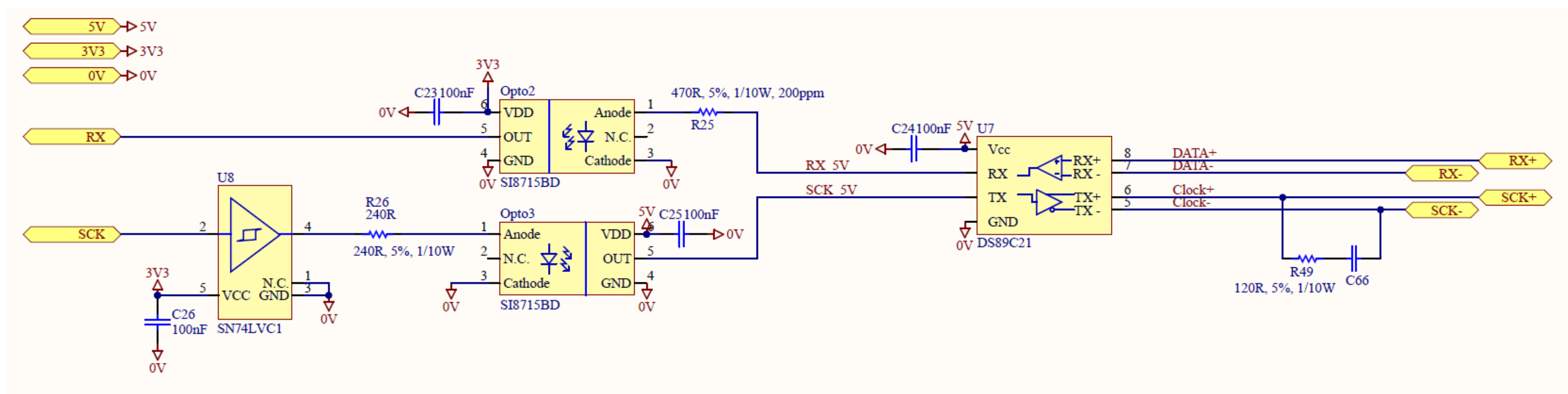

Figure H.11: Schematic drawing of the encoder interface circuit.

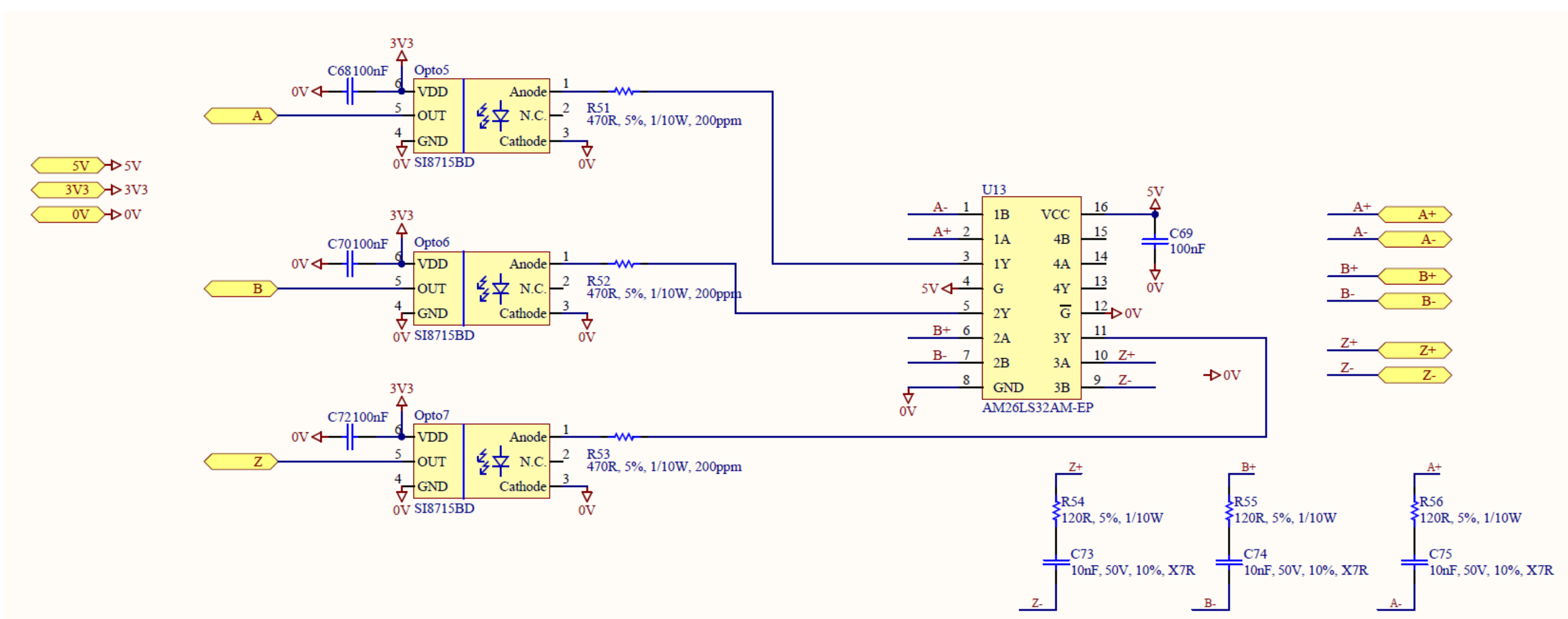

Figure H.12: Schematic drawing of the encoder with ABZ interface circuit.

Figure H.13: Schematic drawing of the current sensor interface.

Figure H.14: Schematic drawing of the voltage measurement circuit.

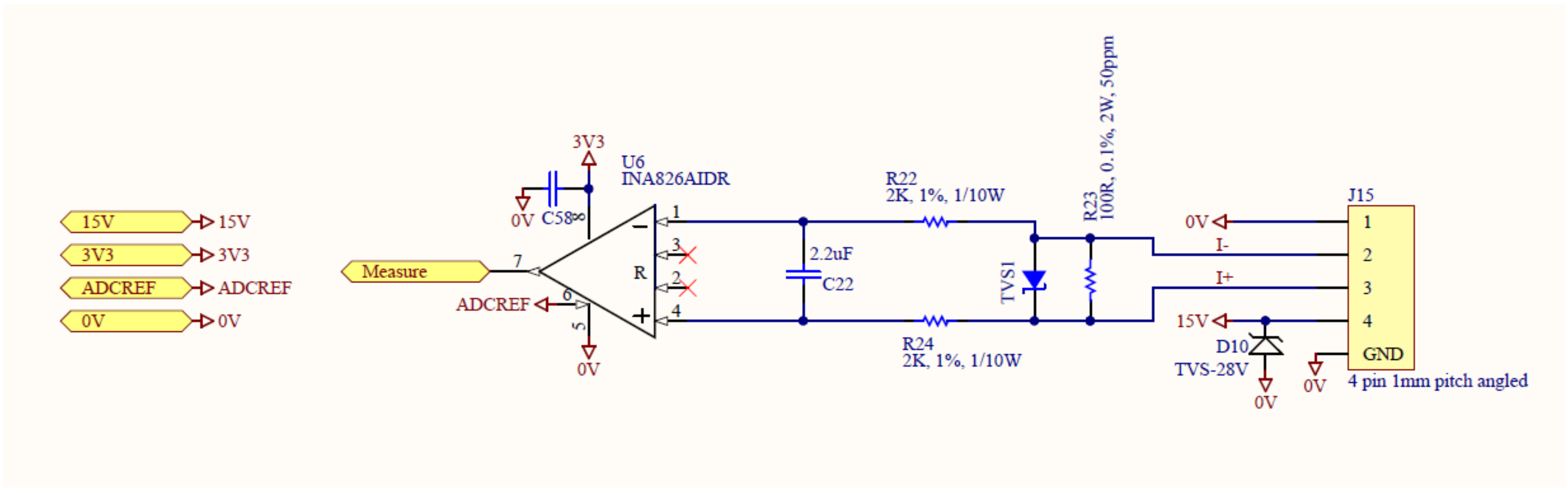

Figure H.15: Schematic drawing of the temperature measurement circuit from heatsink.

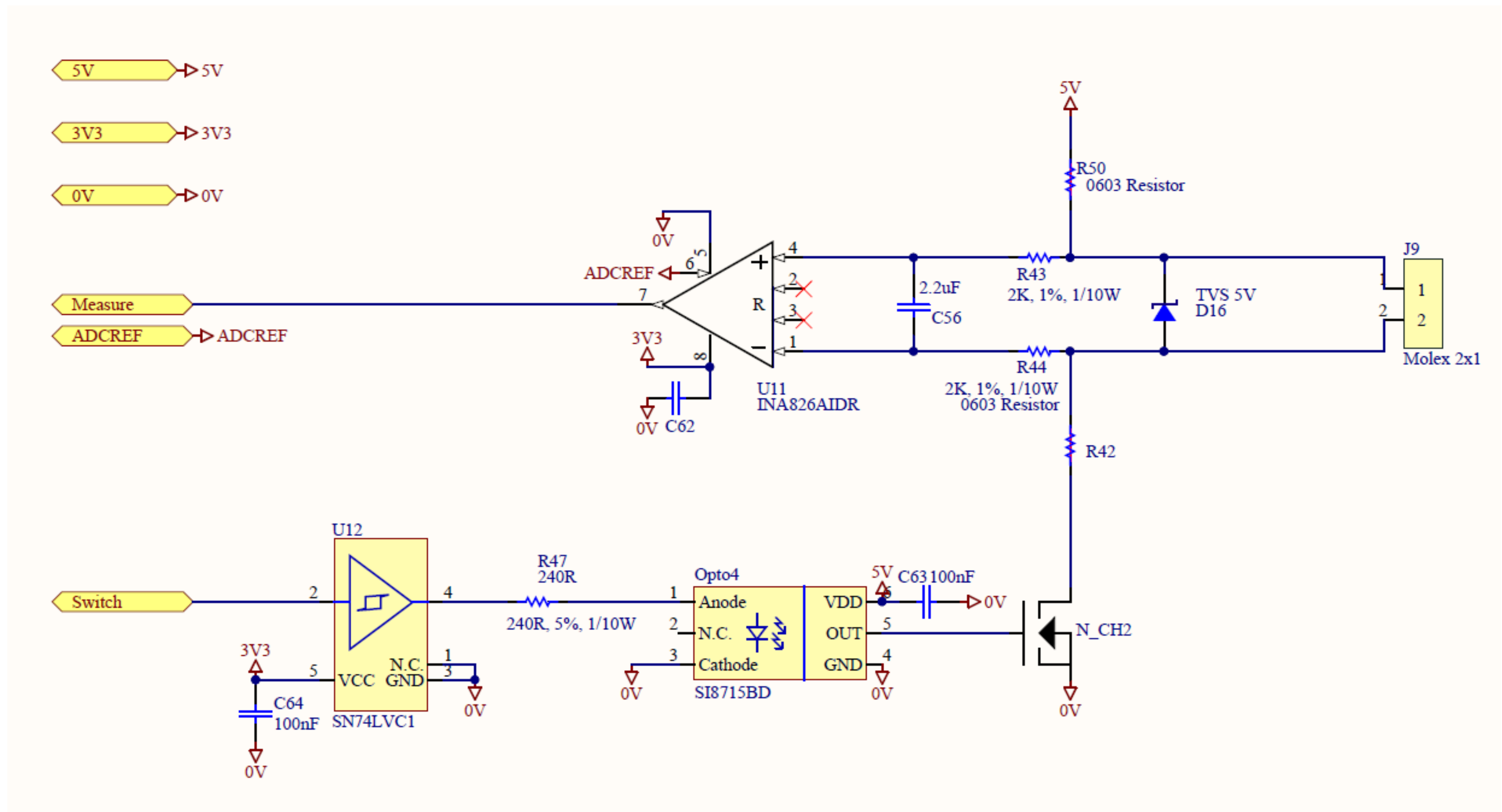

Figure H.16: Schematic drawing of the temperature measurement circuit from motor.

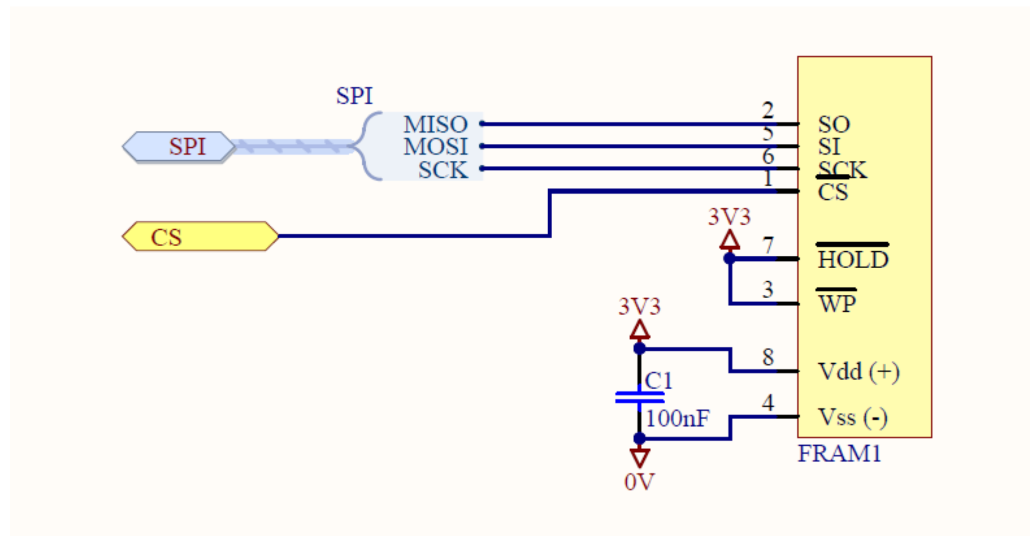

Figure H.17: Schematic drawing of the FRAM circuitry.

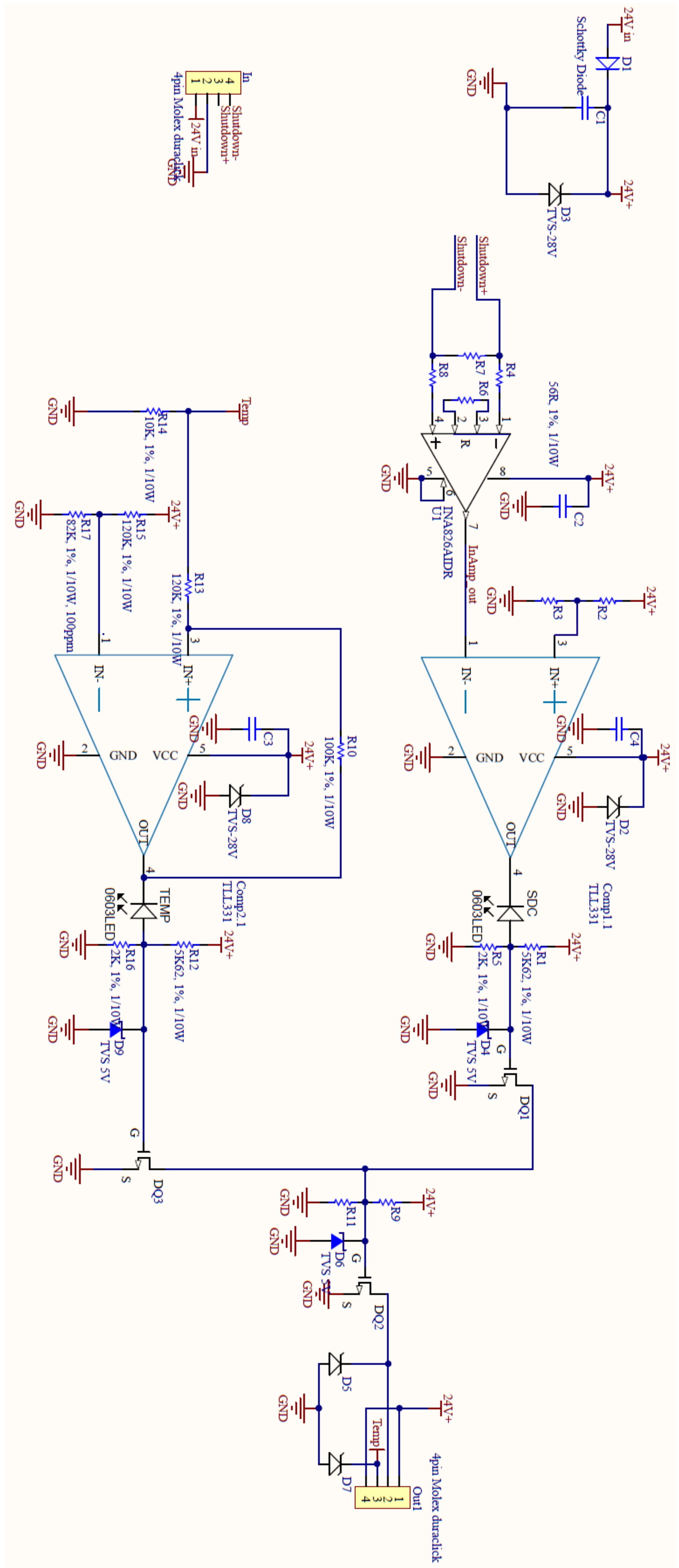

Figure H.18: Schematic drawing of the discharge circuit.