Design of Test Bench for Electric Motors used in Electric Motorbikes

- Final Report -

Fredrik Lundeberg, Iñigo Alvarez
Talha Khan, Daniel Söderman
Moaaj Bhana, Sebastian Bülow
Sofia Olsson, Johan Blomqvist
Jennifer Tannfelt Wu

KTH
Mechatronics

Nomenclature

AC Alternating Current

BLDC Brushless Direct Current

CAD Computer Aided Design

CAN Controller Area Network

DC Direct Current

EMI Electromagnetic Interference

FEM Finite Element Method

FOC Field Oriented Control

GND Ground

Id Direct Current

Iq Quadrature Current

LED Light Emitting Diode

LM Load Motor

LMC Load Motor Control

MUT Motor Under Test

PCB Printed Circuit Board

PWM Pulse Width Modulation

SOTA State of the Art

SSR Solid State Relay

UI User Interface

UML Unified Modeling Language

Contents

1 - Introduction

1.1 Background

CAKE is a start-up company based in Stockholm whose goal is to produce high performance electric off-road motorcycles. They aim to launch their first model to the market in 2018, and are currently designing and building their first prototype. The company logo can be seen in Figure 1.1.

Figure 1.1: CAKE logo

The electric vehicle market has been steadily growing, however, is still far away from reaching maturity - especially for electric motorcycles [8]. Many suppliers and manufacturers for electric motorcycle specific power components, such as motors or batteries, still often fall under the hobbyist category and lack complete documentation and extensive testing. Therefore, there is a need from CAKE to be able to select components after having extensively testing them for performance and robustness.

1.2 The Project: Building a Test Rig

In order to help the company choose and test electrical power components, we have proposed to build a test rig that will be able to primarily test the performance of their motors but also their batteries, controller, and drivetrain configuration. The complete user requirements are presented in the next section.

1.3 User Requirements

After validating the initial requirements list with CAKE, here is the final user requirements list:

1. The test rig shall be able to test electric motors between 6 and 15 kw
2. The test rig shall be able to test brushed DC and BLDC axial electric motors
3. The test rig shall be able to run with various battery configurations
4. The test rig shall be able to run with various motor controllers
5. The test rig shall be able to run with various drivetrains
6. The test rig shall be able to simulate test tracks by providing varying loads to the MUT which corresponds to the effects of driver weight, motorcycle weight, and test track slopes

The refined user requirements and requirements developed subsequently from them can be seen in more detail in Section 1.5 and Appendix A respectively.

1.4 Scope

In the previous semester we investigated the SOTA for test rigs. Most industrial dynamometers use hysteresis, eddy current, or powder brakes to measure the torque of the MUT with the shafts of the brake and the MUT coupled directly together [6].

From user requirement 6, we see that that using a brake is not possible as it would be unable to simulate a downhill scenario in a test track. Therefore, we made the decision to use a BLDC motor rated higher than what was specified in user requirement 1. To satisfy user requirement 5, we designed the test rig with the capability of direct shaft coupling, or coupling through a belt drive with interchangeable gear ratios.

While CAKE wanted a lot of functionality from the test rig, we focused our efforts on primarily being able to simulate a test track and being able to test motor properties due to time and cost constraints. By allowing for various battery and motor controller configurations as specified in user requirements 3 and 4 and having as modular a design as possible, we have ensured that this project can be further expanded upon. An expanded project could eventually be used not only for testing motor properties and performance in specific scenarios, but also tuning controller parameters, and optimizing and tuning the system of motor, drive, batteries, and controller for performance in specific off-road applications (such as tuning for a sports mode vs. eco mode). A detailed explanation of our major design decisions can be seen in Section 3 and suggestions for future work can be found in Section 8.

1.5 Method

This section will detail our development methods and further detail our user requirements.

V-model

Our systems development was based on the V-model where we started with the User Requirements and further refined the requirements for subsystems. We broke down our project into 4 major categories: mechanical, electrical, motor control, and embedded systems. By initially considering the overall design, we were able to break up our project into subgroups for further development. We followed the V-model for each of these subsystems starting with the development and testing of individual units, which were integrated into subsystems and tested and then later in the development process, integrated into the overall system.

SysML/Model-Based Development

For organizing and documenting our work, we used SysML modeling language. SysML is very similar to UML but makes improvements to it so that it is more suited towards systems engineering. It supports specification, design, analysis, verification and validation of systems and so works well to accommodate the details of the V-model. The SysML language is quite extensive and due to time constraints, only the key features were taken advantage of. We used SysML exclusively to document all our requirements as it allowed us to take advantage of the ability to derive requirements from larger systems, break up larger requirements into subrequirements and provide traceability to hardware blocks. We also used block diagrams and detailed block diagrams extensively, however, these were implemented in Simulink. For details, see Appendix E. We also used state-charts to design the logic of our embedded systems, and visualize the possible test-cases and eventually test for model coverage. The details of our state-charts can be seen in Appendix G.

Requirements

From CAKE we derived our User Requirements which can be seen in Section 1.3. We further refined these user requirements, the details of which can be seen in Figure 1.2. The requirements were used to develop our subsystems and further refine them with requirements. The final requirements can be seen in Appendix A with the details of SysML also tabulated into a table for clarity.

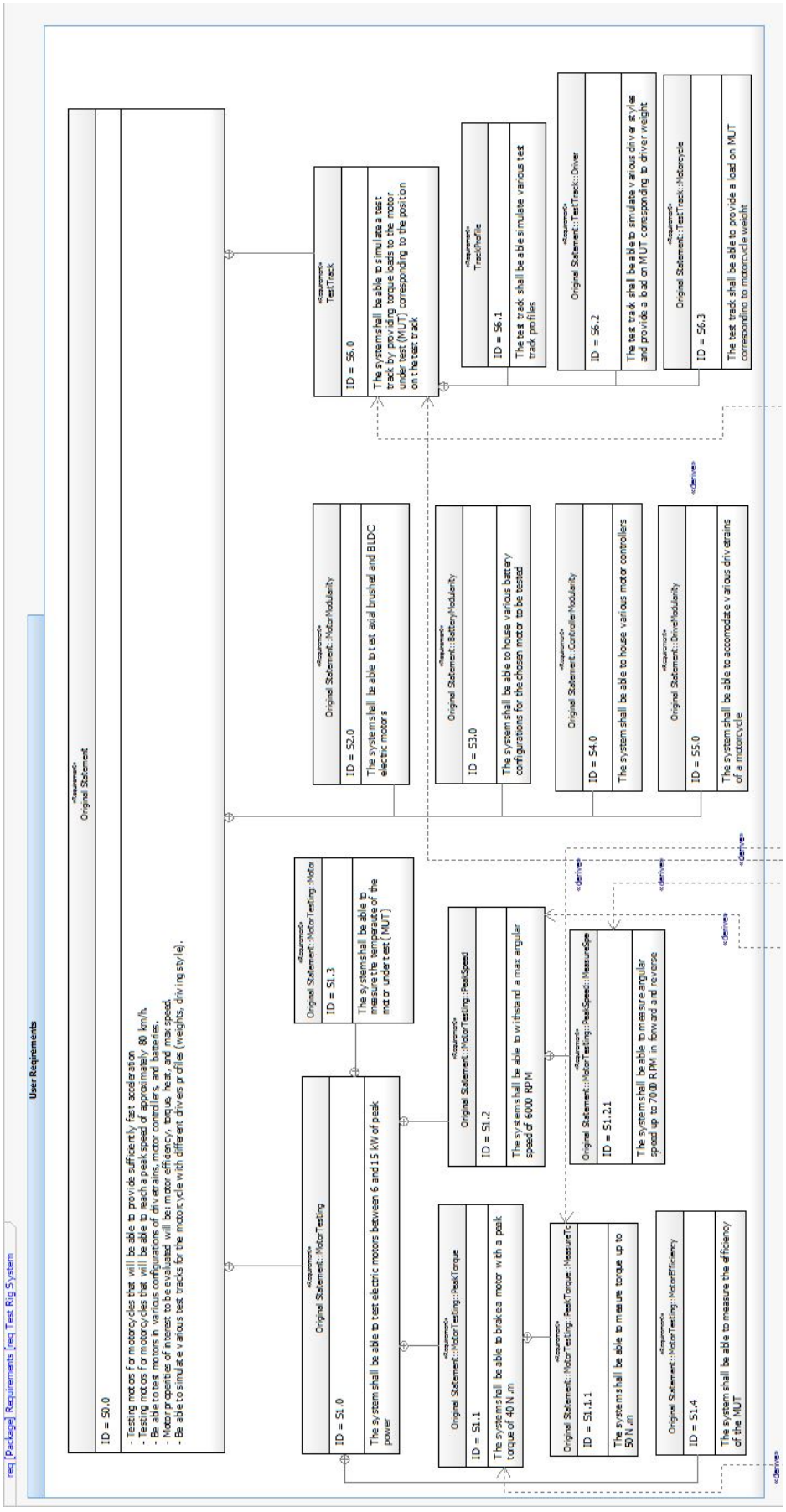

Figure 1.2: User Requirements

Analysis

For analysis and calculations we used Simulink. We developed a model for the BLDC motor we used as our load motor along with a controller for it based on FOC. The details of these can be seen in Section 5 and Appendix E. For electrical circuit analysis, most of the calculations were done by hand or in simple matlab scripts. For mechanical design, we created models in the CAD software SolidEdge and did FEM analysis for the critical components. The details of the FEM analysis can be found in Appendix C.

Testing, Verification, and Validation

Where testing was possible for individual units, we did testing and verified that the unit blocks worked before we integrated them into subsystems. For test cases, refer to Appendix B. When a subsystem was created, we created test cases to properly test those components as well before integrating them into the larger system as per the V-model. For our electrical components we primarily performed simple bench tests.

2 - Design Overview

This report follows as a continuation of the project's previous report from May 2017. In that report a state of the art is presented together with design concepts. In this chapter new major design decisions will be motivated.

2.1 Design Decisions

Many design decisions have been taken but the most important ones needs to be motivated since they form the base of the project.

2.1.1 Mechanical Design

A requirement from CAKE is to have a modular test rig that can test different motors and transmissions. The two previously developed concepts were to have a plate with several holes for motor mountings or to use rails on which the components could move. Sketches from those concepts are shown in Figure 2.1. The final design was to go with aluminum rails to ensure flexibility in the axis' positions.

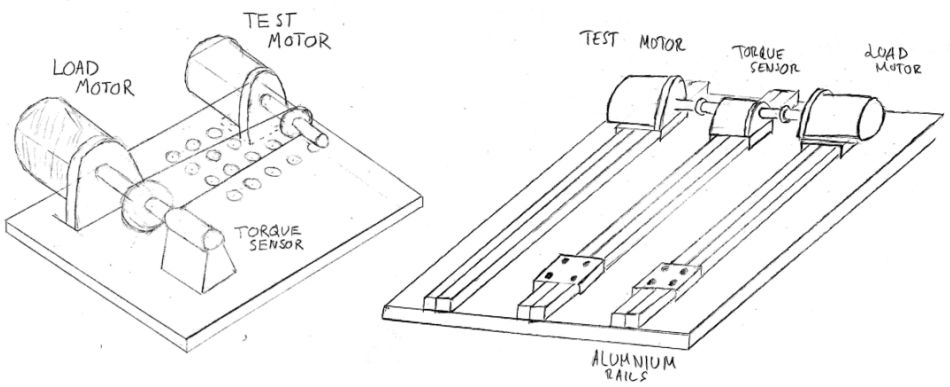

Figure 2.1: Two concepts of the mechanical design

The chosen transmission between the load motor and the MUT was a belt drive since this will introduce one extra degree of freedom and increase the modularity. The MUT does not have to be on the same height as the load motor and it is easy to test another motor, as only an extra pulley is required for the transmission. For details, please refer to Section 4.

2.1.2 Power Design

In the first designs of the test rig configuration all concepts were powered by the AC main, however this turned out not to be possible due to unavailability/functionality of the power mains in the Mechatronics lab hall. As seen in Figure 2.2 there have been considerations to have separate batteries for each motor for regeneration or a common battery for regeneration of the entire system. There have also been considerations to power both motors by the AC main or only the MUT. The final design was to power each motor with separate batteries.

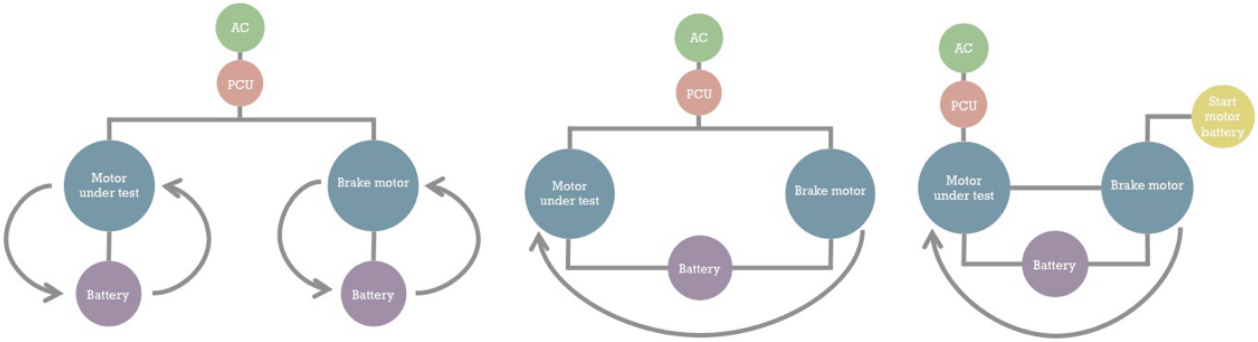

Figure 2.2: Concepts of the test rig configuration

The second major design decision came as a result of a requirement from CAKE (see: user requirement 3) who wanted to be able to test a full system and not just a motor. It should be possible to test different controllers and batteries for the MUT. Therefore, the powering of the MUT should be separated from the load motor and the regenerated energy from the load motor could not be used to power the MUT. The design decision made was to have two separate battery supplies for each motor.

2.1.3 Control Design

The motor and controller for the load motor was provided by CAKE. The controller supports torque control functionality, but since in practice it is current control, an outer control loop had to be designed. To implement this a torque sensor was mounted on the shaft so a torque could be fed back to the outer control loop.

2.1.4 Distributed System Design

In the old report the embedded system was the least developed part. The idea was to have a single micro controller connected to the computer with a user interface. The final design is to have a distributed system with connected nodes separated by functionality.

This allows less functions on each node and by this a higher execution frequency as well as parallel execution. The distributed nodes communicate with each other through the CAN protocol. The main reason to have more than one nodes is because of safety. If one system fails others subsystems can still be controlled.

2.2 Overall Design

In Figure 2.3 an overview of the whole system is depicted. On the left the MUT is connected to a controller and battery. The MUT is connected to the load motor via a torque sensor and an encoder. The purpose of the load motor is to test the motor properties of the MUT and simulate a test track and motorcycle dynamics. In most cases the load motor will work as a generator and energy will be generated. This will charge the battery until it's fully charged. In cases where the battery is fully charged, there will be a dump load that the current is diverted to to protect against overcharging the batteries. Two of the system's

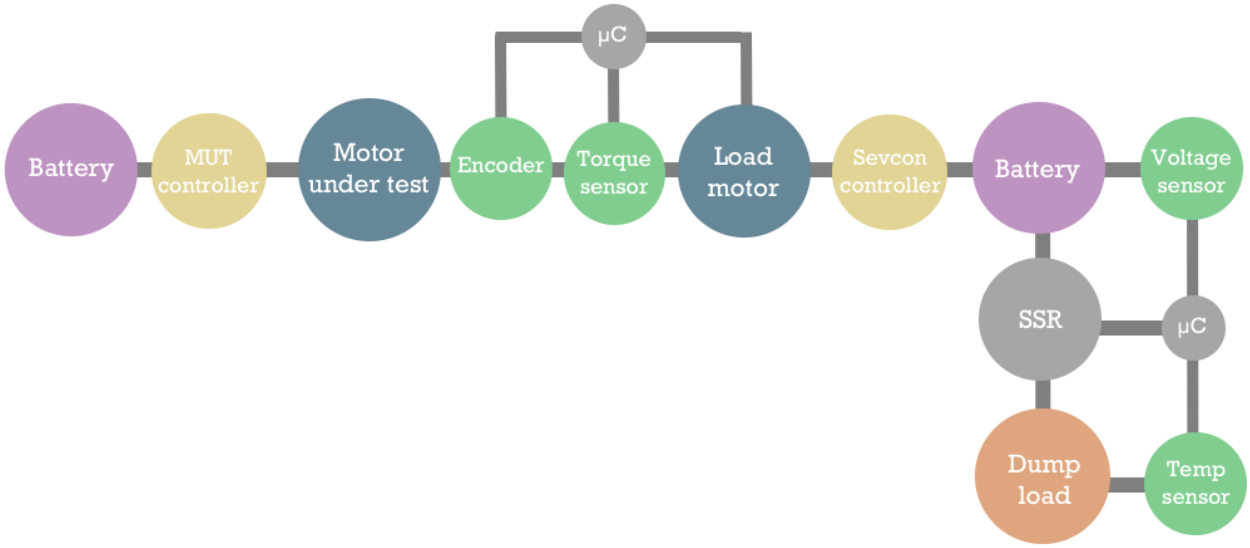

Figure 2.3: Simplified overview of final design

three micro controllers are included in the figure. The third one is used to display real time data for the user as well as monitor the system status by receiving status messages from the other nodes. The micro controller connected to the load motor takes care of the outer control loop. The micro controller connected to the voltage sensor and the temperature sensor handles redirection of generated energy.

2.3 Safety Considerations

As the test rig deals with high voltages and large torques, safety considerations are very important. Most of the safety features were built into the motor controller provided by CAKE. There is also inherent safety built into the BLDC motor as it cannot simply run when connected to a voltage source and so in most failure conditions, the motor will stop running. To protect against any other unexpected failure, the design intent was to have an emergency switch that could disable the entire system. For mechanical safety, components were designed to ensure that mechanical failure is unlikely, and static analysis of the load cases were performed for the load- and MUT modules.

3 - Electrical Design and Electronics

3.1 Overview

In this chapter all the non-mechanical components are described. For a simplified wiring diagram of the whole system, see Figure 3.1. In this wiring diagram, all the physical relationships between the different components are represented. However, circuits, sensors and controllers are represented as "black boxes" with some inputs and outputs, and will be further detailed in Section 3.3. The embedded microcontrollers will be further detailed in Chapter 6. In Section 3.4, an overview of the functionality of the entire rig is given before discussing the main operation modes of the load motor from an electrical point of view.

Most of the circuits in Figure 3.1 and the microcontrollers will be placed inside a metallic enclosure to protect these elements from EMI. For a detailed description of the enclosure, refer to Section 4.9 in Chapter 4. A schematic of the wiring going in and out of the enclosure is included in Appendix D.

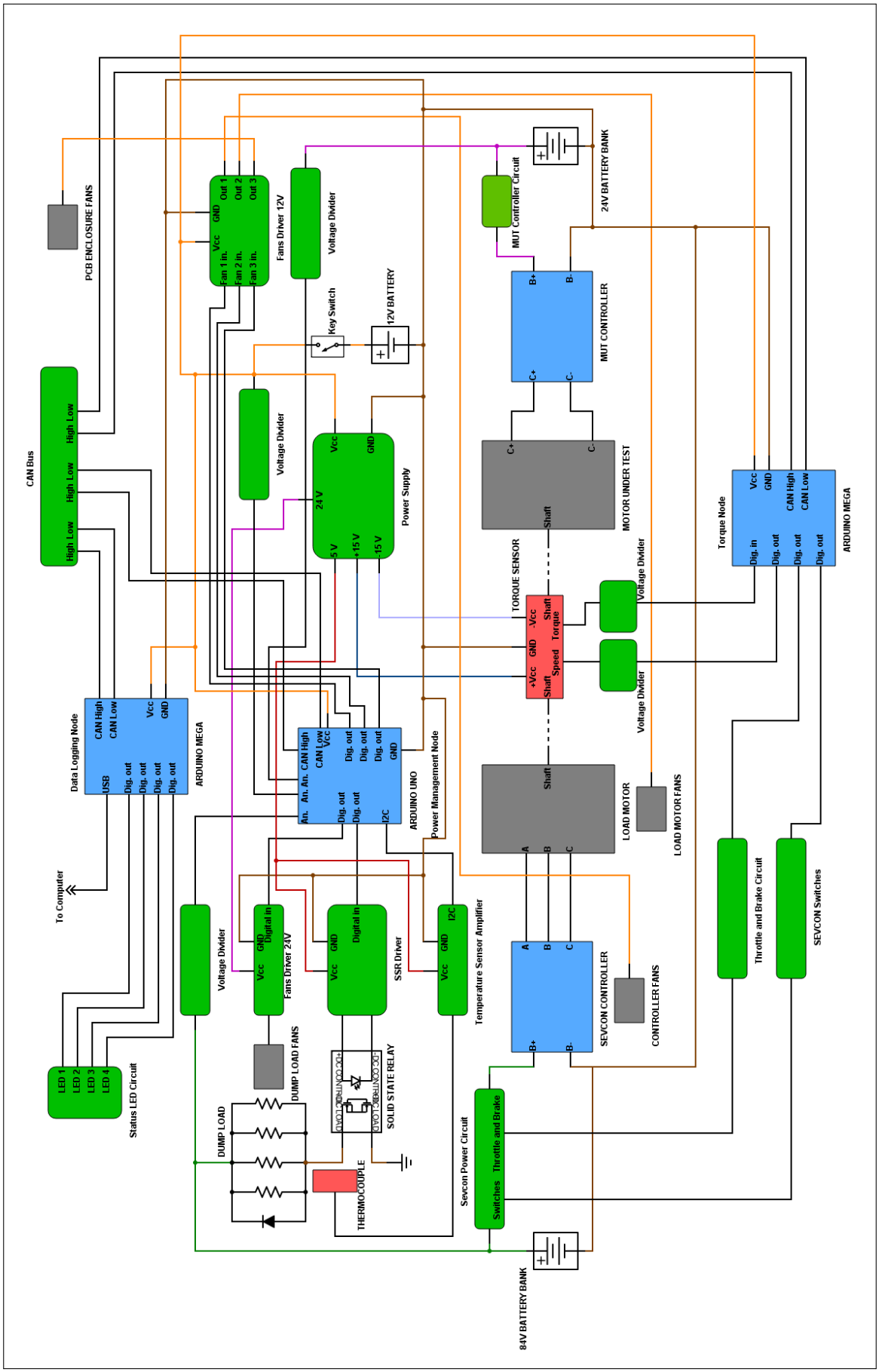

Figure 3.1: Wiring diagram.

3.2 Components

3.2.1 Load Motor Side

Load Motor

The motor used as a load for the test rig is the *Revolt* RV-160-Pro, 75 KV model. The RV-160-Pro comes with hall sensors, a temperature sensor and custom shaft, see Table 3.1 for the motor's main characteristics. For more detailed parameters, refer to Figure F.2 in Appendix F.

Parameter	**Value**
Operation voltage	24V-96V
Continuous power/ Peak power	12KW/20KW
Weight	9.0 kg

Table 3.1: RV-160-Pro characteristics.

A BLDC motor was chosen as the load motor over a brushed DC motor due to multiple factors. BLDC motors provide higher efficiency, have a longer life span, produce less electrical noise due to the lack of a commutation brush, and have higher torque output relative to size, and can provide more precise control. BLDC motors also were cheaper for power 3kW when compared to brushed DC motors.

Controller

The controller for the load motor is a *SEVCON* Gen4 Size 2 72/80V model. This BLDC controller is a four quadrant device, meaning that it can control the BLDC motor both when it is working as a motor or as a generator, which is needed for this application. The ratings for the controller are listed on Table 3.2.

Parameter	**Value**
Working voltage range	50.4V to 96V
Short term current rating	180A
Continuous current rating	75A

Table 3.2: SEVCON Gen4 Size 2 model specifications.

The current ratings for this controller are lower than what is required to run the motor at peak power. The *SEVCON* Gen4 Size 4 72/80V model is better suited to this application however the one provided was not operational and so a compromise had to be made to go with a lower current rating and sacrifice peak power, but still be able to demonstrate the overall concept.

Power

The load motor is going to be powered from an 84V battery bank. The operational range of the chosen motor is between 24 and 96 volts. The decision to go with 84V battery bank was due to the current constraints of the controller as well as the amount of current we wanted to draw from the battery bank to the power resistors (for more information on power resistors, see Section 3.2.3). Higher voltage means less current drawn for the same power output of the motor but also more power dissipated through the power resistors when dissipating the charge from the battery bank.

There were various options for battery choices. Lithium-ion batteries give high energy density however are expensive and also require a more complex battery management system to charge the individual lithium-ion cells. Therefore, we decided to go with 12V lead-acid car batteries connected in series as the power source. Lead-acid required a simpler battery management system and met our budget constraints. The battery model chosen is *Xtreme* E1. Each battery has a capacity of 50Ah.

Cooling

According to the manual, the short-term and continuous current ratings of the *SEVCON* controller have been rated with a $0.5°C/W$ finned heatsink attached to its surface with no forced-air cooling [7]. The controller has a built in mechanism to prevent overheating by derating the current it allows which leads to lower performance at unexpected times necessitating the need for cooling. Due to budget constraints, forced air cooling is used for cooling instead of the use of a heatsink [11]. The main parameter in the decision of fan sizing when there is direct exposure to the surface is the airflow of the fan determined by the equation.

$$Q(CFM) = \frac{1.76 \times P}{\Delta T} \tag{3.1}$$

Where Q is airflow, P is power and ΔT is the difference between the maximum allowable temperature and the ambient temperature. Based on this equation, the *Sunon* MEC0251V1-000U-A99 fans have been chosen to cool the Sevcon controller. In addition to fan cooling, the Sevcon controller is mounted to another metal plate which increases its thermal surface area allowing for additional heat dissipation. The controller and the plate are coupled with thermal paste to ensure that there is direct contact without air gaps.

An additional *Sunon* MEC0251V1-000U-A99 fan is used to cool the *Revolt* motor. The motor is an outrunner which has inherent cooling properties but for safety reasons, additional cooling is provided to the motor.

3.2.2 Motor Under Test (MUT) Side

The test rig was designed around *CAKE*'s specifications (See User Requirement 1). However, the company could not provide a set of motor, controller and battery pack for testing. This led the project team to decide on a brushed DC motor for testing, as they have a quicker configuration when compared to a BLDC motor.

The motor chosen is a *Prestolite Motors - Ametek* MUX-6302. The specifications of this DC series-wound brushed motor are shown in Table 3.3.

Parameter	**Value**
Operation voltage	24V
Peak power	3360W

Table 3.3: MUX motor characteristics.

The brushed DC MUT is controlled by a *Kelly* KDS24200E controller. This controller was chosen as its current ratings allowed for peak operation of the MUT. Some of its general specifications are listed on Table 3.4.

Parameter	**Value**
Working voltage range	12V to 24V
Short term current rating	200A
Continuous current rating	120A

Table 3.4: Kelly KDS24200E specifications.

The motor under test is powered by a 24V battery bank, consisting of two 12V batteries connected in series. The batteries used are the same as for the load motor: *Xtreme* E1 (50Ah).

3.2.3 Battery Management

The components of the Battery management system are explained in this section. For a detailed wiring diagram and further explanation, refer to Section 3.4.

Power Resistors

When the load motor is breaking the MUT, it is acting as a generator. The generated current is going to be used to charge the battery bank until it is fully charged. To prevent overvoltage, once that the battery bank is fully charged the generated power is diverted to

a power resistor bank that dissipates the energy in the form of heat. The power resistor bank or dump load consists of four power resistors, each one rated for 2.5kW and with a resistance of 3.9 ohm. Connected in parallel they are able to dissipate up to 10kW. The power resistors are manufactured by *TE Connectivity* and the model is TE2500B3R9J.

SSR

To switch the current towards the dump load in case the battery bank is fully loaded, the *Crydom* HDC200D160 DC-DC solid state relay has been chosen since it can switch at high frequency and is rated for 150 Amps and 150 VDC.

Freewheeling Diode

As the power resistor bank is also an inductive, switching with the SSR can lead to voltage spikes that would destroy the SSR. To protect the SSR, a *Semikron* SKR100/12 freewheeling diode connected is connected in parallel with the dump load.

Fuse

The system is also protected against a surge current from the motor by a fuse rated for 100A.

Cooling

The power resistors generate heat but begin derating at an ambient temperature of $70^{\circ}C$ [10]. As the resistors are near each other, their ambient temperature rises quite quickly and therefore cooling is needed. The fans for cooling the power resistors were selected by their rated airflow according to equation (3.1).

The power resistors are going to be cooled down by six fans. The fan model chosen is a *Sanyo Denki* 109E5724K502. The specifications of each fan are detailed in Table 3.5.

Parameter	**Value**
Working voltage range	24VDC
Rated current	1.3A
Airflow	300.14 CFM

Table 3.5: Fan specifications.

3.2.4 Sensors

The rig contains four sensors: temperature sensor, torque sensor, speed sensor, voltage sensor. The temperature sensor is a K type thermocouple, with a detectable range of -50 to 600 ^{o}C. This temperature sensor measures the temperature of the power resistors, that have to be kept below a certain temperature value in order to keep its nominal resistance value.

The torque sensor is coupled between the load motor and the MUT. It uses strain gauge measuring technology and outputs a digital signal with frequency proportional to the torsional strain. The signal is then converted to the corresponding torque in a microcontroller and fed back to the load motor. Its range is $\pm$100 Nm and the frequency of the output signal varies between 5-15 kHz. 10 kHz is the offset which corresponds to 0 Nm. The power supply needed is $\pm$15V and roughly 75 mA from each source.

The torque sensor is also equipped with a speed sensor. It outputs a digital signal where the RPM of the shaft is derived from the frequency of the signal and the number of pulses per revolution. Information on the voltage sensors is provided in Section 3.3.3.

3.2.5 Battery for Power Supply

As seen in Figure 3.1, a 12V battery is used to power all the circuits. A power supply circuit will provide 5V, $\pm$15V and 24V outputs from this battery. The battery used is a lead-acid battery with a 7.2Ah capacity.

3.3 Circuits and PCB Design

3.3.1 SEVCON Controller Power Circuit

The load motor is powered by an 84V battery bank and controlled by a *SEVCON* controller, as defined in Section 3.2.1. The controller requires a specific circuit whose wiring diagram is depicted in Figure 3.2. This wiring diagram shows the detailed connections between the blocks *LOAD MOTOR*, *SEVCON CONTROLLER*, *Sevcon Power Circuit* and *84V BATTERY BANK* in Figure 3.1. The contactor used is an *Omron* G9EC1B24DC model, whose specifications are given in Table 3.6. Wires between the *SEVCON* controller, *Revolt* motor, battery bank and contactor are all 16 mm^2 section.

There are four drive switches. The FS1 switch is part of the throttle assembly. The throttle voltage is ignored until FS1 is closed. The seat switch indicates presence of operator on the vehicle. Drive was not allowed if this switch is open. Forward and reverse switches were used to indicate the direction that motor should rotate. We used a footbrake switch, which

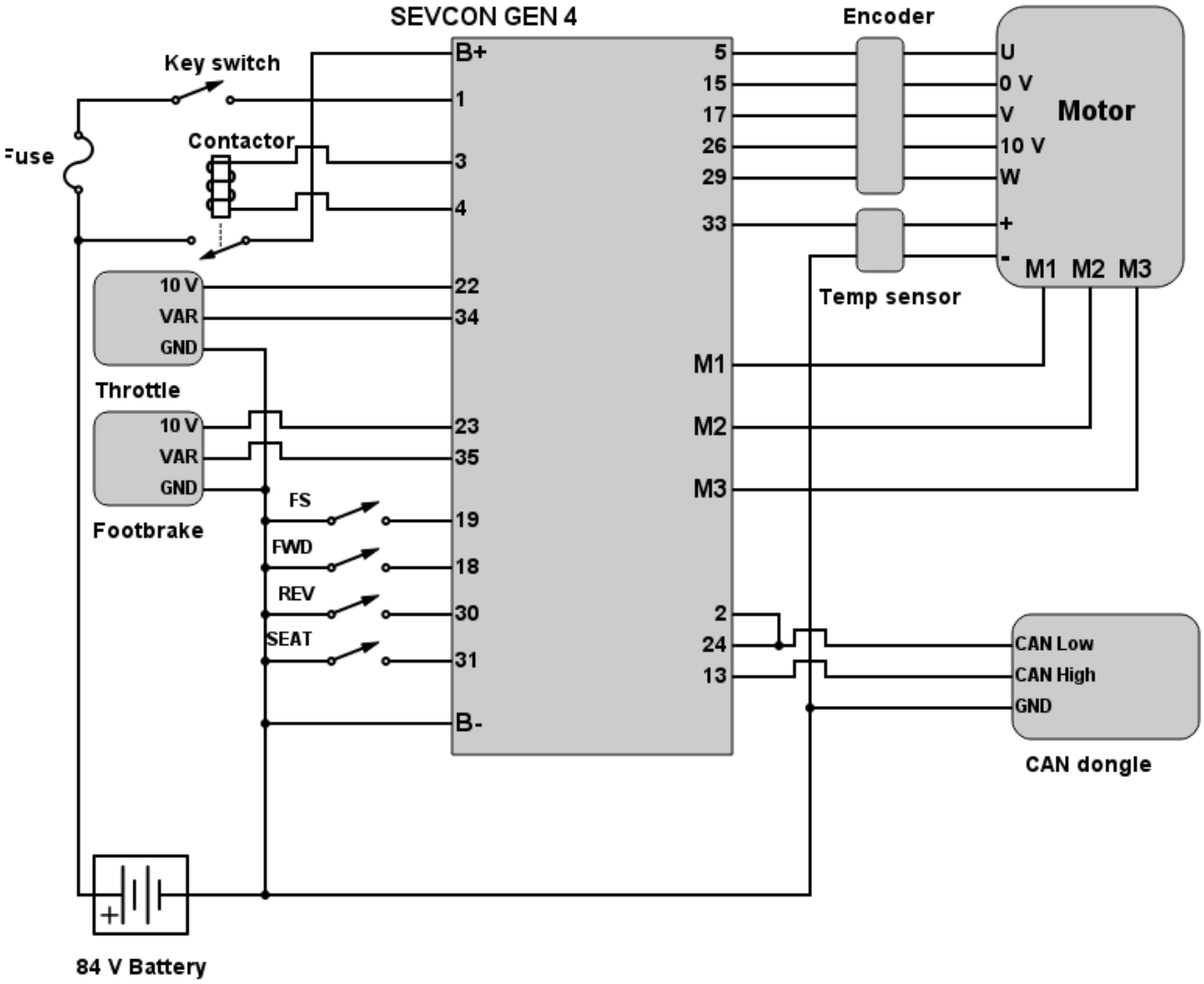

Figure 3.2: Load motor wiring diagram.

Parameter	Value
Coil voltage	24 VDC
Switching voltage	400 VDC
Switching current	200 A

Table 3.6: Load motor's contactor specifications.

is used to apply maximum foot braking when the switch is closed. It is used to reduce the speed of the motor when in operation. Footbrake is given higher priority when both footbrake and throttle are pressed at the same time.

Throttle input is configured so that a torque demand is continually calculated to take account of pre-set limits on the level and rate-of-change of torque. The torque demand is used to calculate current demands; that is, the controller calculates what current is required

within the motor to generate the required torque.

3.3.2 Kelly Controller Power Circuit

The motor under test is powered by a 24V battery bank and controlled by a *Kelly* controller, as defined in Section 3.2.2. The controller requires a specific circuit whose wiring diagram is depicted in Figure 3.3. This wiring diagram shows the detailed connections between the blocks *MOTOR UNDER TEST*, *MUT CONTROLLER*, *MUT Controller Circuit* and *24V BATTERY BANK* in Figure 3.1. The contactor used is an *Ametek Prestolite* JAC4603BB00 model, whose specifications are given in Table 3.7. Wires between the B+, B- and M terminals in the controller and the motor under test all have a 16 mm^2 section.

Parameter	Value
Coil voltage	24 VDC
Switching voltage	48 VDC
Switching current	100 A

Table 3.7: MUT's contactor specifications.

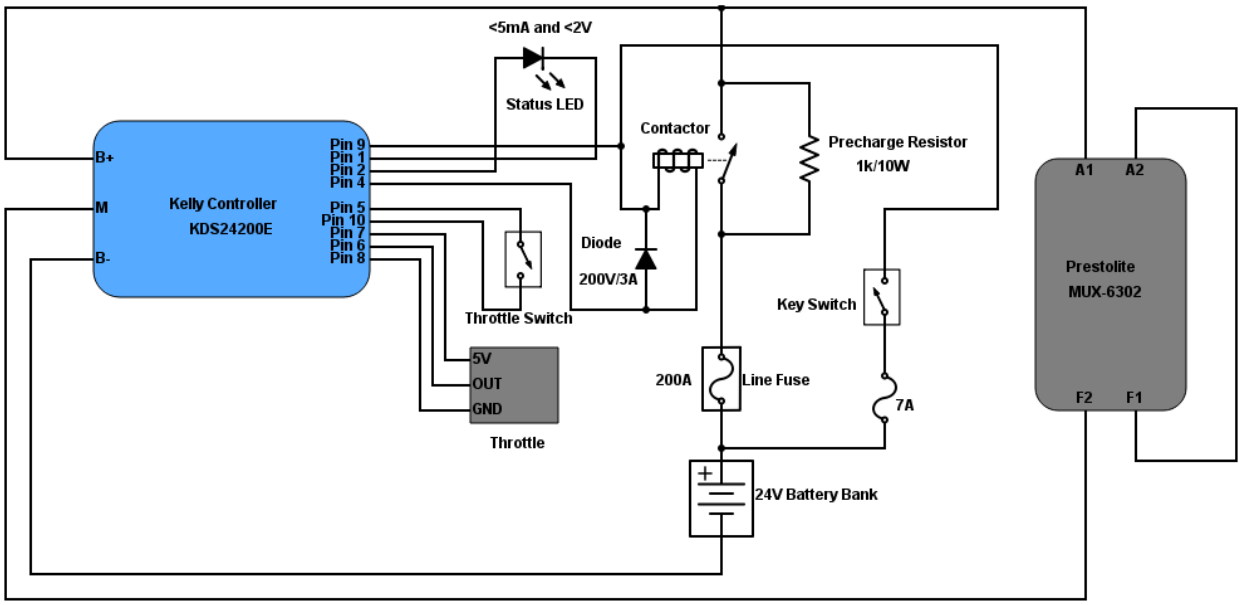

Figure 3.3: MUT wiring diagram.

3.3.3 Voltage Dividers

There are five voltage dividers in the test rig: one for the 84V battery bank on the load motor side, one for the 24V battery bank on the MUT side, one for the power supply's 12V battery and two for the torque sensor's torque and speed outputs.

For example, the first voltage divider is used to convert a 0 to 89 V range into a 0 to 4.72 V range in order to be able to measure the voltage with an Arduino Uno, whose analog input pin can only take up to 5 V. The circuit is completed with a capacitor on the low side of the voltage divider, for filtering purposes as well as to compensate for the high input impedance, which is necessary when R1 is higher than 10K ohm and Arduino's analog input pin is used. Also, a zener diode is added for protection to make sure that in case of the voltage exceeds 5 V, the Arduino is not damaged. See Figure 3.4 for a schematic of the circuit.

As mentioned, 89 V (100% charge) correspond to 4.72 V with this voltage divider configuration. The voltage should be kept between 89 V (100% charge) and 86.8 V (80% charge). This is due to the choice of batteries: lead-acid car batteries should be kept above 80% charge. A charge of 80% corresponds to 4.61 V over R2. This means that the measurements needs to be quite accurate. The A/D converter on the Arduino Uno is 10 bit which means that steps of 0.005V can be measured. It is considered enough in this case, since variations of approximately +/- 0.03 V will be measured. The power supply voltage of the Arduino has to be very stable and accurate since it is the reference voltage used internally in the Arduino when using the analog input pin for voltage measurement. Resistors with very good tolerances (+/- 0.1%) are used to ensure accurate measurements.

For the other two voltage dividers used for measuring battery voltages, the design is the same, and only the values of the resistors are changed. For the torque sensor the two voltage dividers used follow the same design. A voltage divider is used on the digital output signals of the torque sensor (speed and torque) to reduce the maximum output voltage from 13V to roughly 4V. In Table 3.8, the values for the resistors used for every circuit are specified.

Voltage divider	**R1**	**R2**
84V Battery	100 kΩ	5.6 kΩ
24V Battery	5.6 kΩ	1.3 kΩ
12V Battery	5.6 kΩ	3.3 kΩ
Torque sensor: Torque signal	2.2 kΩ	1.3 kΩ
Torque sensor: Speed signal	2.2 kΩ	1.3 kΩ

Table 3.8: Resistor values for the three voltage dividers.

3.3.4 Solid State Relay (SSR) Driver

The solid state relay requires an input voltage between 4.5 and 32 VDC on the logic side. The Arduino Uno can provide 5V from its digital output pins. However, the current is limited to 20mA in the digital pins of the Arduino Uno. A test was performed where the solid state relay's input was connected straight into a digital output in the Arduino. The SSR could not be switched in this configuration as it requires more current than the 20mA that the Arduino can supply. Therefore, the circuit shown in Figure 3.5 was created, where

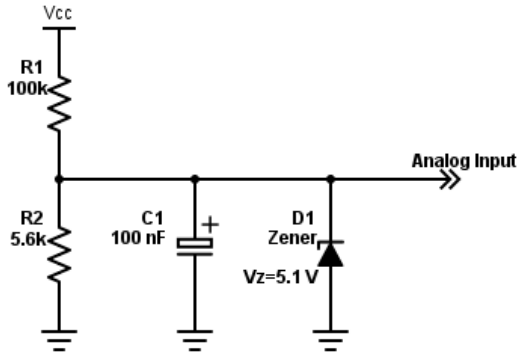

Figure 3.4: Voltage divider.

the Arduino switches a NPN transistor which switches 5V from a power supply. This way the input pins of the solid state relay will get enough current.

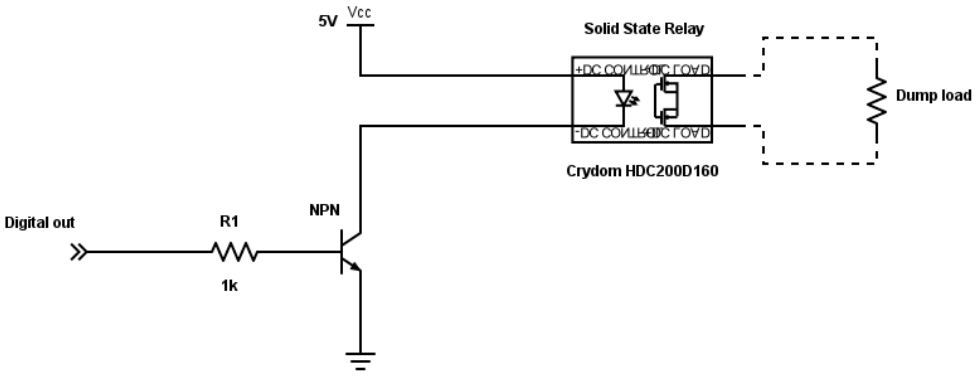

Figure 3.5: Solid state relay driver circuit.

3.3.5 Fan Drivers

There are four fan sets in the test rig, listed in Table 3.9.

Element to cool	Fan model	Driver circuit
Load motor	Sunon MEC0251V1-000U-A99	Fan circuit 1
Load motor controller	Sunon MEC0251V1-000U-A99	Fan circuit 1
Electronics enclosure box	Sunon EB60251S1-000U-999	Fan circuit 1
Dump load	Sanyo Denki 109E5724K502	Fan circuit 2

Table 3.9: Fan circuits.

There are two circuits to drive the fans, necessary since they are going to be controlled from an Arduino. *Fan circuit 1* drives the 12V fans and *Fan circuit 2* drives the 24V fans. For the

Fan circuit 1 , BU910 NPN darlington power transistors are used for switching the fans on and off (see Figure 3.6).

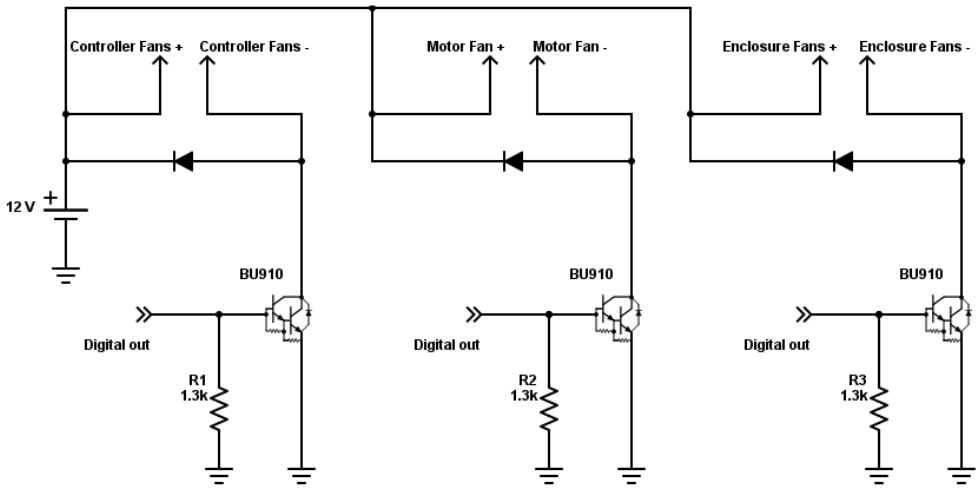

Figure 3.6: Fan circuit 1.

For the *Fan circuit 2*, a BUZ342 power mosfet is used for switching the fans on and off (see Figure 3.7).

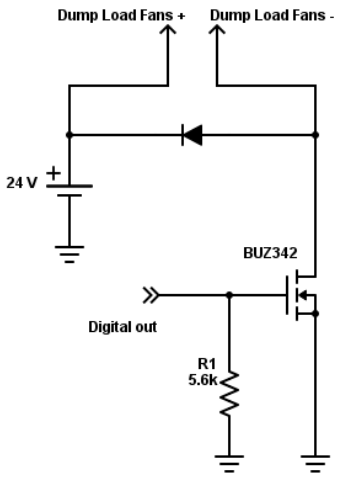

Figure 3.7: Fan circuit 2.

3.3.6 Voltage Regulators

All the PCBs and sensors are powered from a 12V battery. However, the torque sensor is powered with $\pm$15V, the fans for the dump load are powered by 24V and a 5V line is needed to power the digital side of the solid state relay and the temperature sensor amplifier.

It was decided to go for off the shelf voltage regulators. The regulators have to convert a 12V input into a 5V, ±15V or 24V. In the Table 3.10 the voltage regulators used are listed.

Input	Output	Model
6.5-32VDC	5VDC/1A	Traco Power TSR 1-2450
9-36VDC	±15VDC/330mA	Traco Power THM 10-2423WI
8-60VDC	10-120VDC/0-15A	900W DC-DC CNC boost module

Table 3.10: Voltage regulators.

3.3.7 Temperature Sensor Amplifier

The amplifier circuit for the thermocouple is manufactured by *seeed studio*. This "Grove - High Temperature Sensor" board amplifies the thermocouple signal and provides a cold - compensation reference.

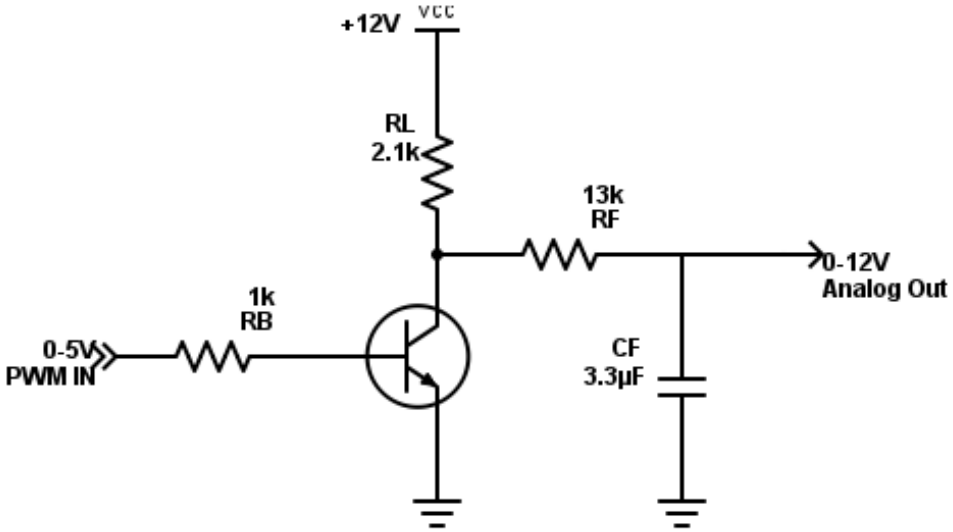

Figure 3.8: Throttle/Brake circuit.

3.3.8 Throttle and Brake circuit

The throttle and brake circuits scales the 0-5V PWM output from an Arduino into an analog voltage between 0 and 12V, thus allowing the throttle and brake signals to be controlled by an Arduino. This is done by first switching a 12V supply with a PWM signal from the Arduino and then using a low-pass filter to get a stable DC voltage. The two circuits, throttle and brake, are identical and a schematic can be seen Figure3.8.

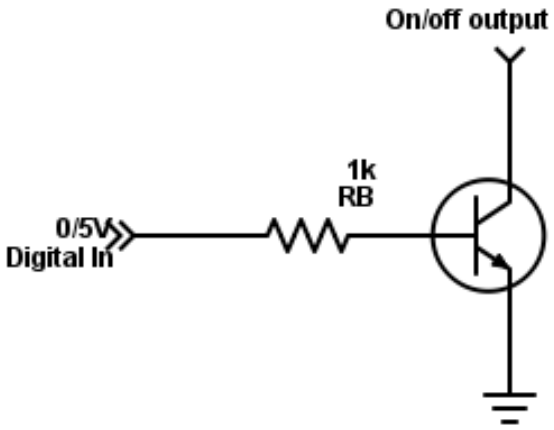

Figure 3.9: SEVCON Switch circuit.

3.3.9 SEVCON Switches Circuit

The different switches on the Sevcon controller need to be pulled to ground to be activated. The switching circuit does this with the on/off signal coming as a digital output from an Arduino. The schematic for this circuit can be seen in Figure3.9.

3.3.10 Status LED Circuit

This circuit includes four LEDs that are attached to the enclosure box and can be used to represent different states in our system.

3.4 Power Management

There are two main possible scenarios in the test rig regarding the load motor: it can either behave as a motor or as a generator. In this section both situations are going to be discussed.

One of the main features of this rig is the implementation of a dynamic motorbike model that can translate a height profile input into a torque output. This allows to input a track profile and simulate the torque that the MUT will take if it was mounted on a real motorbike. The load motor is going to get a torque reference from the dynamic motorbike model, corresponding to a specific track, and is going to apply this torque on the MUT's output shaft. The load motor can output a torque both in a clockwise and an anticlockwise direction. The possibility of applying a torque in both directions with our load motor means that we can both brake and drive the MUT. If an uphill or a flat section is going to be

simulated, the load motor will be breaking the MUT by applying a torque in the opposite direction to the MUT. On the contrary, if a downhill section is going to be simulated, the load motor will drive the MUT by providing a torque in the same direction. These possible test scenarios mean that both the load motor and the MUT can behave as a motor or as a generator during a test.

3.4.1 Load Motor Acting As a Motor

Whenever a downhill profile is being simulated, and the load motor is driving the MUT, or for very low braking torques, the load motor is going to behave as motor. This means that the load motor is going to be draining the batteries during this time. In Figure 3.10, the current path and direction when the load motor acts as a motor is depicted.

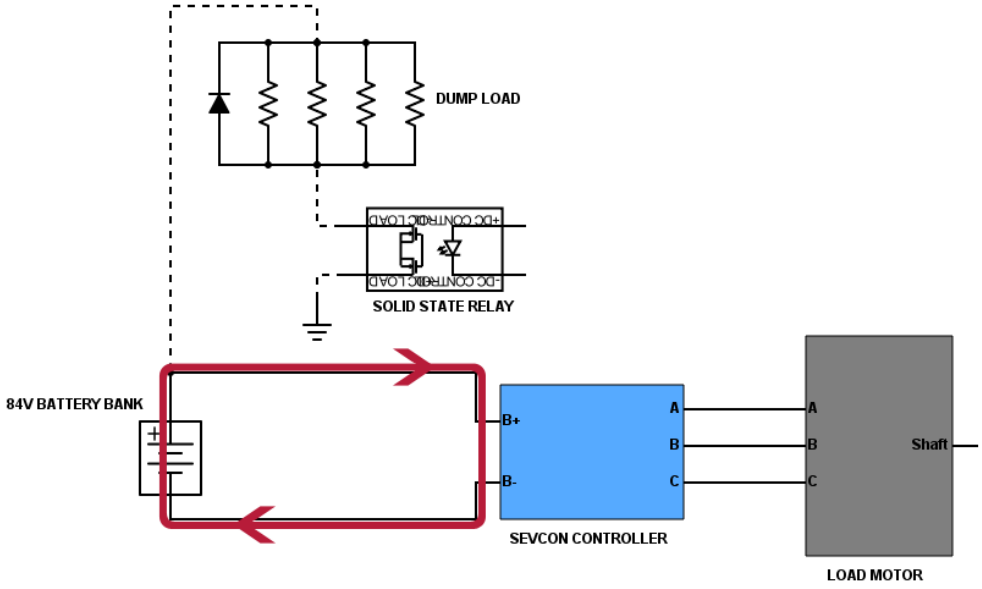

Figure 3.10: Load motor acting as a motor.

3.4.2 Load Motor Acting As a Generator

When flat or uphill profiles are simulated, the load motor is mostly going to be a generator. This situation requires a design that can manage the generated power. While the load motor is acting as a generator, the generated current is used to charge the 84V battery bank used to power the load motor (see Figure 3.11). The voltage of the battery bank is constantly monitored, and when it reaches its full level of charge, the solid state relay is turned on to drive the current through the power resistors. The power will then be converted to heat in the dump load until the battery bank discharges to a certain level (see Figure 3.12).

It is important to mention that while the generated power is burned in the resistor bank, there is power being drawn from the battery bank all the time. The reason for this is that the battery bank has to be connected to the *SEVCON* controller all the time in order to power it. When the power resistors are connected, they are in parallel with the battery bank and the *SEVCON* controller. Because all these elements are connected in parallel they are under the same voltage and the power resistors will consume a current proportional to this voltage all the time. Therefore, whatever current is not supplied by the generator has to be drawn from the battery bank. A simplified model to explain this behaviour is depicted in Figure 3.13.

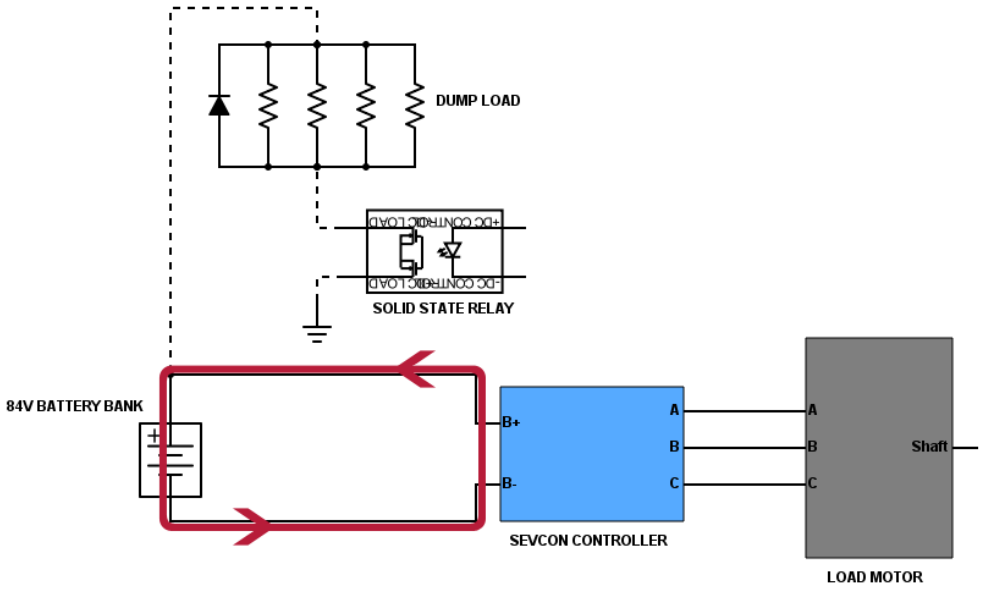

Figure 3.11: Load motor acting as a generator.

3.4.3 Motor Under Test As a Motor or Generator

The MUT can also behave both as a motor or as a generator. The current design does not include any solution to dissipate the energy generated by the MUT once that its battery bank is fully charged. However, the *Kelly* controller used for the current MUT can monitor the battery voltage of the MUT and an option can be set on the controller's software to shut down the main contactor when a certain voltage is reached in order to protect the batteries.

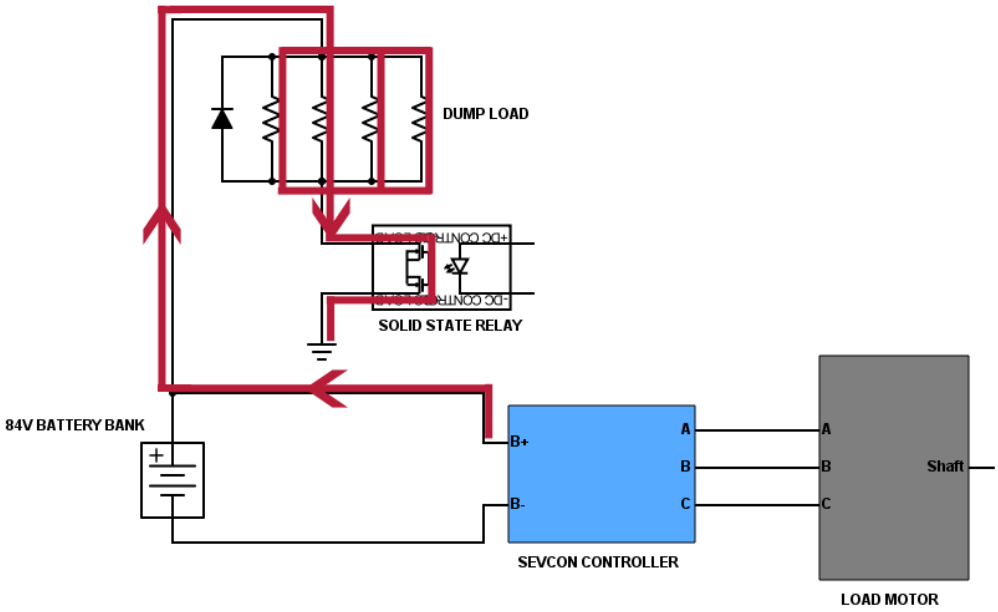

Figure 3.12: Load motor acting as a generator when the battery bank is fully charged.

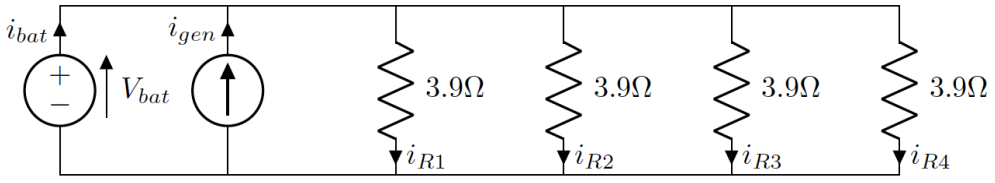

Figure 3.13: Simplified model of the battery bank in parallel with the SEVCON controller and power resistors.

4 - Mechanical Design

This chapter covers the mechanical design of the test rig and the considerations made during the design process. First there is an overview of the entire rig and then each part is covered in more detail.

4.1 Overview

For the mechanical design of the test rig, one of the most important requirement is the modular nature of the design. Since the purpose is to test different kinds of electric motors, the mechanical design has to provide the possibility to change both the mounts for the motor and the necessary support equipment, for example batteries and controllers. Because of this the design is based on mounts for the different components that are fastened together on a common base frame. An overview of the finished test rig is presented in Figure 4.1.

The outer dimensions of the base frame is made to fit a standard sized EU pallet, to make the test rig mobile. This led to size constraints for all mounts, which were designed to be as small as possible without disturbing the ability to reach specific components.

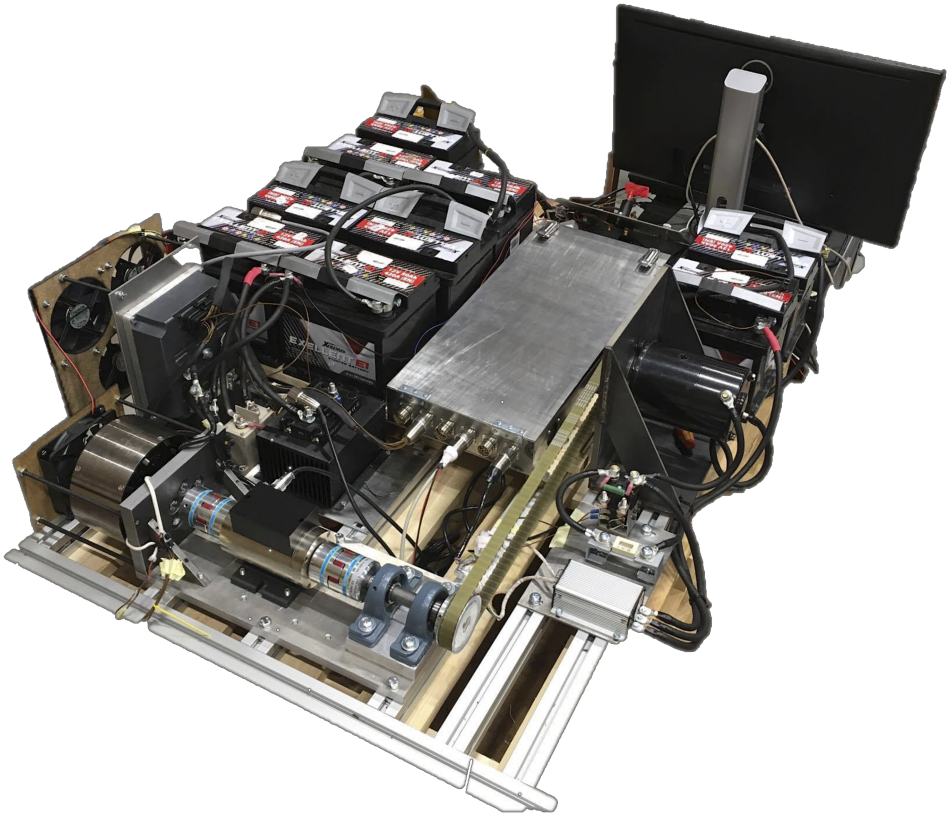

Figure 4.1: Overview of the entire assembled mechanical design.

4.2 Aluminum Base Frame

The base for the test rig was chosen as a frame built with 40x40 aluminum profiles, the frame can be seen in Figure 4.2. These aluminum profiles have T tracks where special nuts and bolts can be used for fastening wherever required on the frame. This allows for a strong base that has great capability for modularity and can be expanded whenever necessary. The base frame simplifies the fine-tuning of the placements of the components, as both the rails themselves and the attachment points inside the rails can be slid into the right position.

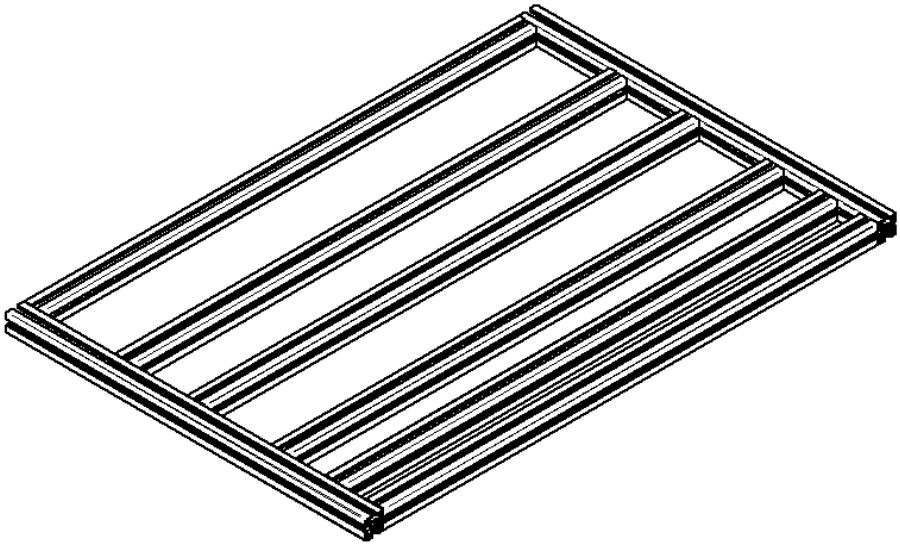

Figure 4.2: Drawing view of the aluminum base frame.

4.3 Load Module

The module where the load motor, torque sensor and belt pulley come together is referred to as the load module, see Figure 4.3. The design is aimed to allow for ease of manu-

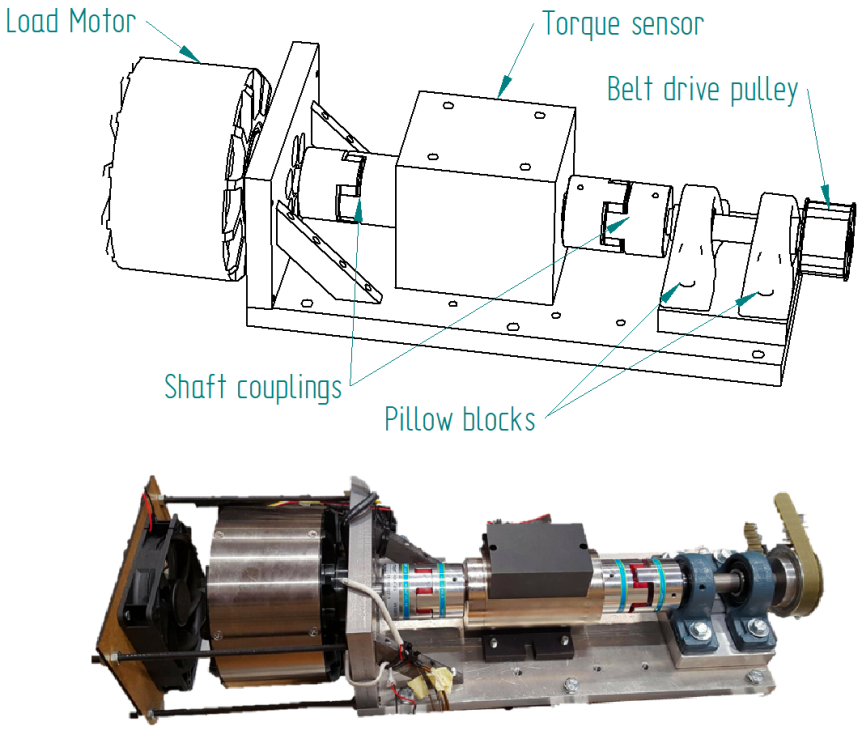

Figure 4.3: Drawing and picture of load module with fan mount attached.

facturing and assembly, while maintaining flexibility in mounting in order to be able to satisfy the tolerance requirements posed by the torque sensor. The three main components of the load module are the load motor, the torque sensor and the output shaft. The output shaft serves the purpose of keeping the design flexible and modular, making it possible to change shaft as desired to fit different transmissions etc. It is mounted on two pillow blocks which allows the shaft to absorb potential radial shock loads in the transmission which could otherwise damage the torque sensor.

The torque sensor requires the shafts of the power- and load equipment to be concentric with the shafts of the sensor within 0.05 mm. There are two ways for the design to guarantee the fulfillment of this requirement. One approach is to set tolerances for manufacturing such that mounting of parts is guaranteed to be centered within the posed limits. However, this requires knowledge on the mechanical tolerances of the parts to be mounted, and that the mounts of the parts are designed with proper alignment surfaces machined to high precision. As the mechanical documentation of the torque sensor was not sufficient for such purposes, an alternative approach was selected where the holes for the mounting bolts of the components were given enough clearance such that all components could be adjusted on assembly to be sufficiently aligned. The upper deviation for measurements determining the height of the shafts was set to 0 to guarantee that any misalignment can be corrected with shims in assembly.

In order to simplify the assembly process, a pair of flexible shaft couplings were selected which are depicted in Figure 4.4.

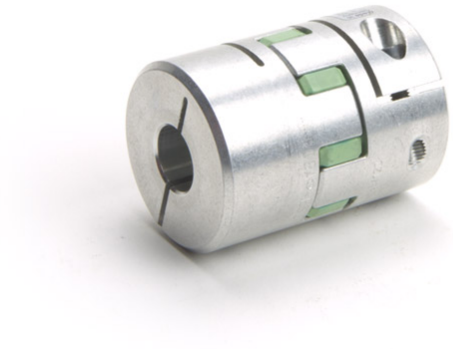

Figure 4.4: The flexible shaft coupling selected.

The hardness of the elastomeric element, together with the size determines the permitted misalignment values, transferable torque and spring rigidity of the coupling [4]. The size of the coupling was determined by selecting the largest one permitted by the rest of the design. The hardness of the elastomer were then selected to 98Sh A to allow sufficient torque to be transmitted, while maximizing the permitted misalignment. The properties of the selected coupling can be found in Table 4.1.

The finished design was manufactured by water cutting the base shape of the parts in the

Parameter	Value
Hardness	98 Sh A
T_{nom} [Nm]	60
T_{max} [Nm]	120
ΔK_r [mm]	0.1
ΔK_a [mm]	1.4
ΔK_w [deg]	0.9

Table 4.1: The parameters of the selected shaft coupling. T_{nom} and T_{max} is the permitted nominal and maximum torque. ΔK_r, ΔK_a and ΔK_w are the permitted radial, axial and angular misalignment of the in- and outgoing shafts [4].

selected material, which were machined according to the drawings found in Appendix C to achieve sufficient precision. The material chosen for the load motor plate was steel, to maximize the rigidity and strength of the mount.

4.4 MUT Module

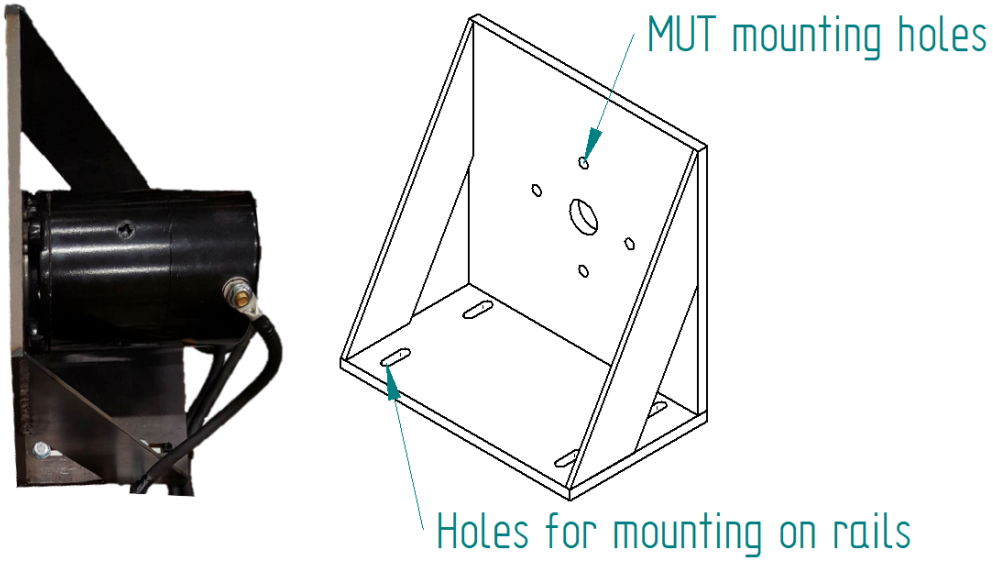

Figure 4.5: The manufactured MUT mount.

The module for mounting the motor under test is designed to be able to be adjusted to align and tension the belt drive. The slots for fastening the mount to the frame allows some movement in the axial direction. The mount can then be moved along the aluminium rail to tension the belt drive. The mount is made by two steel plates. One vertical, where the MUT is fastened with a bolt circle, and one horizontal, these are then supported by two side pieces in steel. The parts are watercut according to the drawings in Appendix C. These parts are then welded together to provide enough strength to handle the torque of both motors.

4.5 Static studies

To verify the strength requirement S1.1 found in Appendix A for both motor mounts of being able to withstand 40 Nm of torque, static studies were done in Solid Edge, as can be seen in Appendix C. These simulations are done with the load torque applied to the bolt circle where the motors would be attached. The load mount motor plate is at most subjected to 32.2 MPa of stress at the bolt circle, as seen in C.16, which is well below the yield point for steel which is at 262 MPa. The MUT mount is also subjected to a stress below the yield point, at 7.57 MPa, as seen in C.17.

4.6 Transmission

Different kinds of transmissions was considered for the power transfer between the load motor and the motor under test. Since the test rig is designed to be be modular, the idea is that the transmission type used can be changed depending on what motor it is used for. For example, gear and chain is often used on motorbikes but if a quieter transmission is desired, belt drives can be a better option. The fact that the mounts can be moved and adjusted means that these different types of power transmissions can be tested in the test rig. The shafts can also be connected directly to each other if information on the motor under test without a transmission is wanted. The transmission selected for this project is a timing belt drive, due to the simplicity in assembly and quieter drive. The chosen components are 31 AT10/22-2 belt drive pulleys and a 16mm wide polyurethane timing belt. The AT10 profile is a timing belt profile built for high pulling forces and high power. The requirement to withstand 40 Nm would mean a force of 1176 N with the chosen belt pulley with an effective diameter of 68.15 mm. The chosen profile with 16 mm belt width has a maximum pulling force of 2470 N [12], which makes it well dimensioned to handle the torque in the requirements.

4.7 Sevcon Controller Mount

The mount for the Sevcon controller, see Figure 4.6, is designed to simplify the cooling of the controller as it gets warm during operation. Therefore, the mount is made of steel which has a high thermal conductivity compared to for example wood, see [17]. The pieces is cut using a watercutter and welded together. The controller is mounted vertically to further improve heat transfer, with the metal base plate of the controller attached to the mount with a thin layer of thermal paste in between to minimize gaps between the surfaces. Fans are attached to the mount using threaded rods and a wooden fan mount, to help cool the controller. The fan mount is cut out in plywood using a lasercutter. The horizontal base part of the mount is attached to the rig. The heatsink for the solid state relay, the contactor

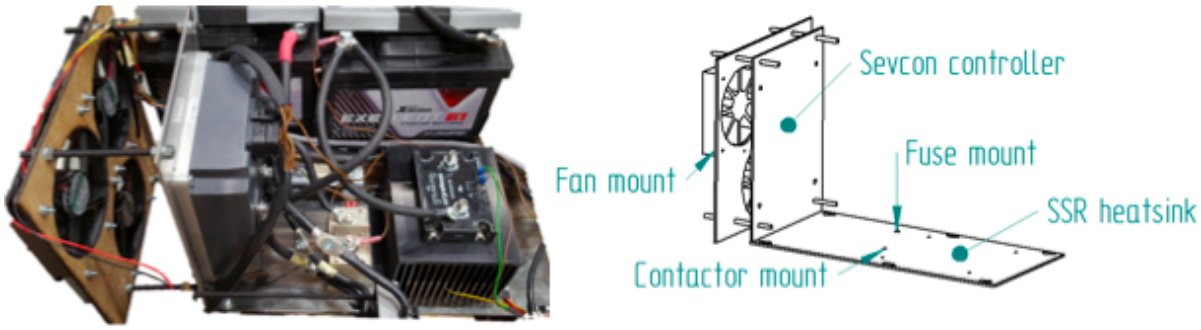

Figure 4.6: Drawing and picture of Sevcon controller mount with fuse mount, contacor and SSR heatsink.

for the controller and the fuse between the batteries and the solid state relay were placed on the base plate of the mount, as the components needed to be close to the batteries and controller. A mount for the fuse was 3D-printed in plastic to isolate the fuse against electric contact with the base plate, see Figure 4.7. The solid state relay was attached to the heatsink with a thin layer of thermal paste to increase thermal conductivity. Drawings for the controller mount, fan mount and fuse mount can be found in Appendix C.

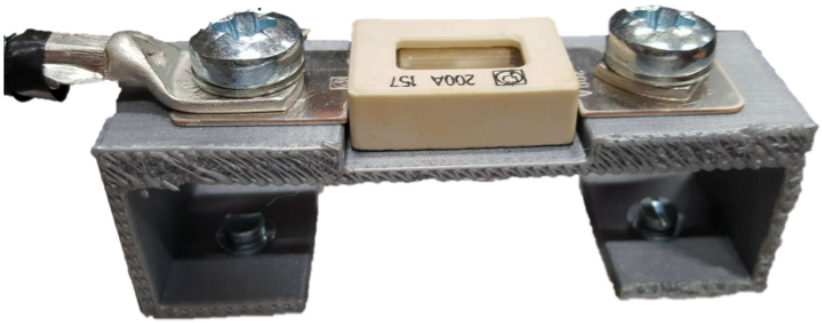

Figure 4.7: Fuse mount.

4.8 Kelly Controller Mount

The mount for the Kelly Controller, see Figure 4.8, is made of steel using the water cutter. The controller is attached with a thin layer of thermal paste to increase the thermal conductivity between the mount and the controller. The mounts also holds a contactor attached with a 3D-printed mount, and a fuse between the batteries and the contactor, see Figure 4.7.

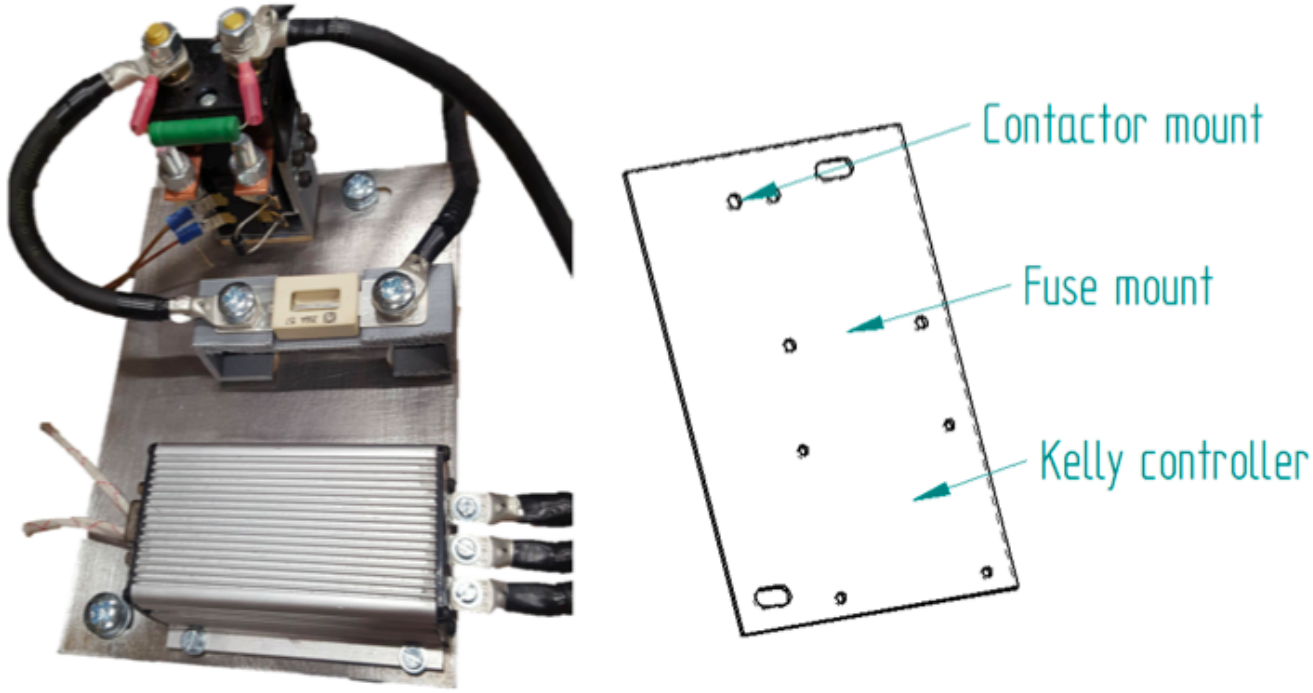

Figure 4.8: Drawing and picture of Kelly controller mount with line contactor and fuse.

4.9 PCB Enclosure

Since the motors are switching loads containing magnets, they produce electromagnetic interference, EMI, which interferes with surrounding electronics [15]. As the circuit boards and microcontrollers are planned to be placed in between the motors, a shielding enclosure is built. The enclosure is made in steel, as electromagnet radiation can not pass through metal [1]. All steel pieces were cut out using a watercutter according to the drawings in Appendix C, and then all pieces except the lid pieces were welded together. All openings to the connectors and fans are cut with as little tolerance as possible to avoid EMI inside the enclosure. A schematic and table of all wires going into and out of the enclosure through the connectors are shown in Appendix D. The lid is split in two, attached with hinges and it is tightly secured using latches. Four LED:s are attached to the lid. Figure 4.9 shows a picture and drawing of the assembled enclosure.

The inside of the enclosure consist of an additional bottom plate made of plywood to isolate against unwanted conductivity between electric circuits. It is cut out using a lasercutter. The wooden bottom plate has cutouts where connectors and fans are risking to interfere with it, to make it possible to take out without removing the connectors or fans. The placement of the electric circuits onto the wooden plate inside the enclosure can be seen in figure 4.10.

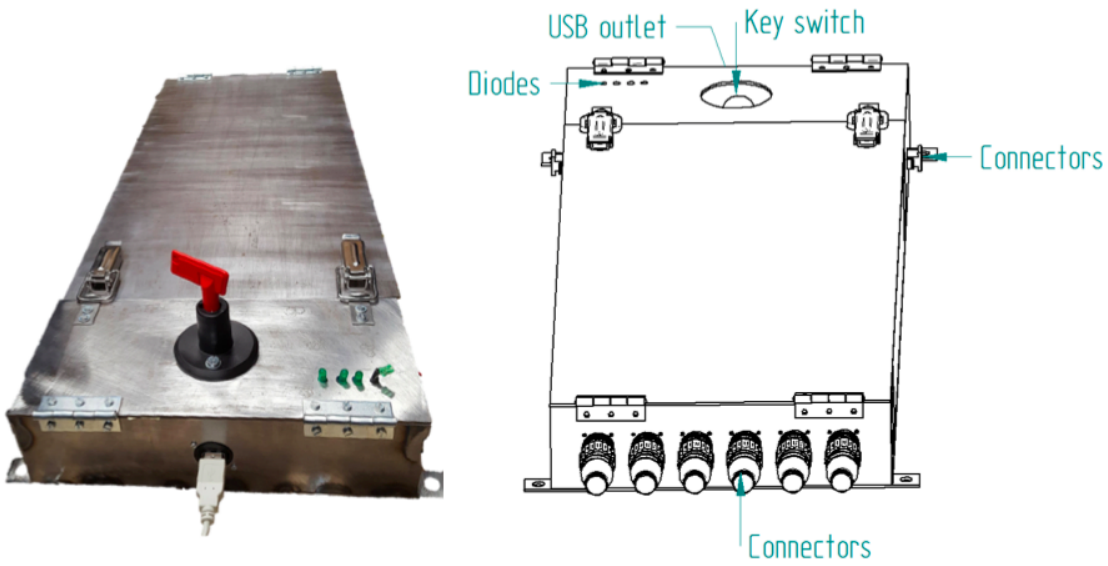

Figure 4.9: Front and back view of enclosure design

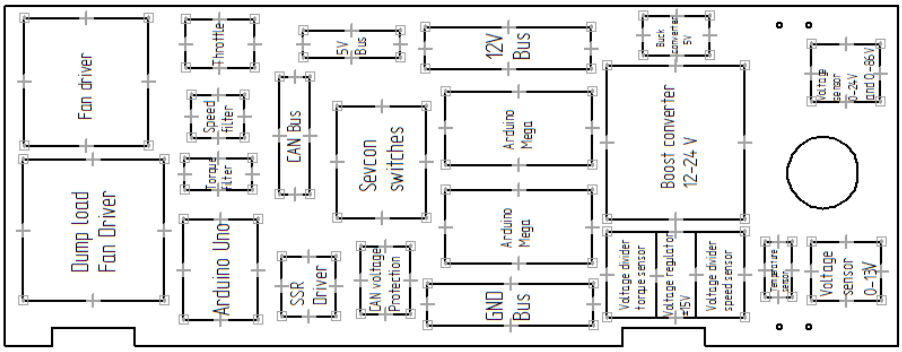

Figure 4.10: Schematic of the PCB:s in the enclosure.

The placement is planned by characteristics of the circuits, example by placing all switching PCB:s like the voltage regulators or fan drivers close to the air flow of the fans as they generate heat, or by what circuits needs to be close to each other to use as short cables as possible. Cut outs were made in front of the fans to let air flow out of the enclosure. The wooden plate is put on a distance from the bottom plate to allow for a fixed placement of the circuits through drilled holes in the future, without the bolts being in contact with the steel bottom plate and risk short circuits and avoid making the enclosure conducting.

4.10 Power Resistor Mount

The power resistors are used to burn away excess energy when the batteries are fully charged, which results in a big temperature increase. To handle the temperature increase, the mount needed to hold both resistors and fans. The mount was made of steel to ensure

a robust construction and allow heat to transfer easily. The overall design is made using as little material between the resistors and fans as possible, to increase the air flow and thereby increase the cooling of the resistors. The mount allows the resistors to be mounted vertically, to further increase the air flow from the fans, see Figure 4.11 for drawing and 4.12 for pictures of the finalized design. All pieces are cut out using a water cutter and then welded according to the drawings in Appendix C. The supports were welded to the mount to make it more robust, because of the vertical design.

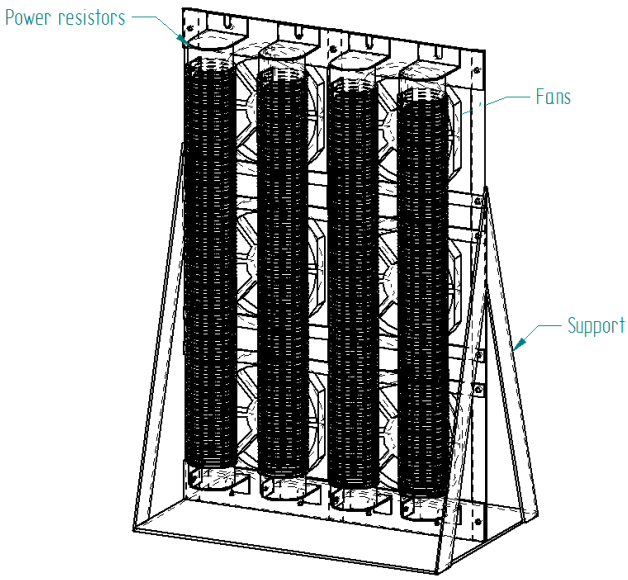

Figure 4.11: Drawinng of power resistor mount.

Figure 4.12: Picture of power resistor mount, from all three views.

5 - Control

This chapter first present a model of the test rig as well as the external and internal forces acting on a motor bike in motion. Simulated test cases are discussed and analyzed in terms of power used/generated in the motors as well as the induced torque from the load motor. The model was used to verify that the calculated speed and torque references from the dynamics profile worked as expected.

5.1 Modeling

A Matlab/Simulink model of the test rig was made with the purpose of obtaining information about the motors' behavior during different test cases. Another purpose is to be able to tune controller parameters which could then later be implemented on the practical controllers. Both the LM and the MUT are modeled as well as their respective controllers. A connecting shaft between the two motors is also modeled, as well as the dynamics of the motorcycle. A simplified schematic of the model can be seen in figure 5.1. Screenshots from the Simulink model can be found in appendix E

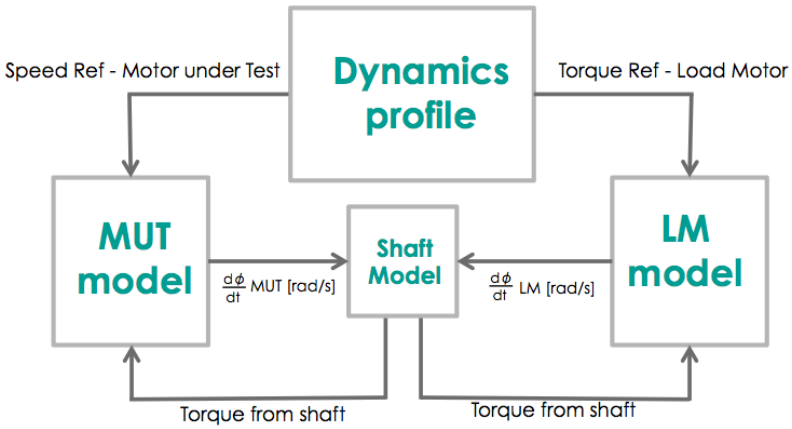

Figure 5.1: Model design for the test rig.

Input to the model is a profile of a test track based on an array containing height profile of the track as well as a fixed length between each height point. An array containing the desired velocity of the motor bike between each height point is also used as input to the model. The dynamic profile is then used to calculate the angular reference velocity to the MUT as well as the torque reference to the LM depending on the bikes current position on the simulated test track. A lot of information can be obtained from the simulation, however the main focus is to see how well the motors follows their respective references e.g. torque

induced from the LM and the angular velocity from the MUT. Other interesting aspects includes PWM signals from the controllers, currents and the external forces calculated in the dynamic profile.

5.1.1 The dynamics of the motorcycle

The motorcycle is modeled in order to generate a torque reference to the LM when the motorcycle constantly tries to reach its reference velocity throughout the test track. The model describes the internal and external forces acting on the motorcycle in motion. The test track is assumed two dimensional to simplify the model. The parameters used to simulate a motor bike are presented in table 5.1. It is important to note that some of the parameters such as bike dimensions, drive-train inertia and rider mass are estimated and could easily be changed to target another profile. Rolling friction coefficient can also be changed depending on the surface the bike is intended to drive on. The modularity of the dynamics profile makes it possible to not only simulate off road test tracks but also urban environments with paved roads.

Parameter	Value	Unit	Comment
Air density, ρ	1.225	$\frac{kg}{m^3}$	Physical constant
Gravity, g	9.82	$\frac{m}{s^2}$	Physical constant
Drag coefficient, C_d	0.7		Approximated
Rolling friction coefficient, C_r	0.02		Approximated
Frontal area, A	0.75	m^2	Approximated
Tire radius, R_w	0.3	m	Approximated
Front sprocket teeth number, n_1	12		Given
Back sprocket teeth number, n_2	80		Given
Motorcycle mass, m_m	60	kg	Given
Rider mass, m_r	80	kg	Approximated
Mass front sprocket, $m_{f\_sprocket}$	1	kg	Approximated
Mass back sprocket, $m_{b\_sprocket}$	3	kg	Approximated
Radius front sprocket, $r_{f\_sprocket}$	0.0225	m	Approximated
Radius back sprocket, $r_{b\_sprocket}$	0.15	m	Approximated
Length chain, l_{chain}	1	m	Approximated

Table 5.1: Table listing the parameters used in the dynamic model of the motorcycle.

Figure 5.2 shows the external forces acting on the motorcycle. The F_{drag} is the air resistance force, F_{grav} is the gravity force and F_{roll} is the rolling friction force (acting on both tires).

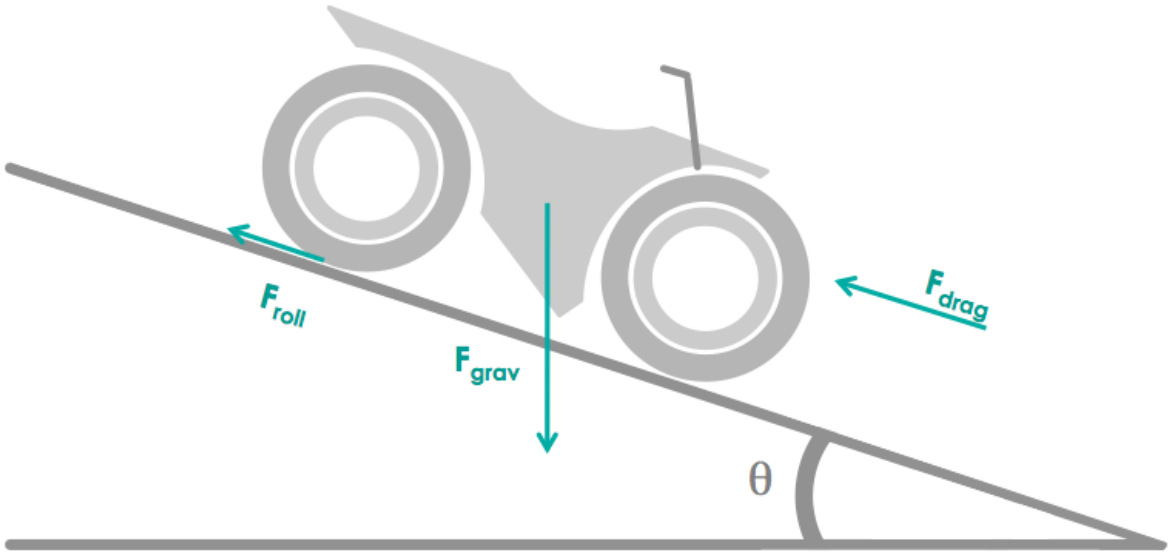

Figure 5.2: The external forces acting on the motorcycle.

The air resistance force F_{drag} is calculated according to

$$F_{drag} = \frac{v_{tire}^2 \cdot \rho \cdot A \cdot C_d}{2} \tag{5.1}$$

where v_{tire} is the tire velocity, ρ is the air density, A is the frontal area of the motorcycle and rider and C_d is the drag coefficient. The v_{tire} in is calculated according to

$$v_{tire} = \omega_{shaft} \cdot R_w \cdot \frac{n_1}{n_2} \tag{5.2}$$

where ω_{shaft} is the motor shaft velocity, R_w is the tire radius and $\frac{n_1}{n_2}$ is the transmission gear ratio. The gravity force F_{grav} is calculated according to

$$F_{grav} = mg \cdot \sin\theta \tag{5.3}$$

where m is the total mass of motorcycle and rider, g is the gravitational constant, and θ is the angle of the slope in radians.

The rolling friction force F_{roll} is calculated according to

$$F_{roll} = C_r \cdot mg \cdot \cos\theta \tag{5.4}$$

where C_r is the rolling friction coefficient between tire and ground. The inertial forces in the drivetrain due to acceleration reflected at the motor shaft is calculated according to

$$T_{drivetrain} = \alpha \cdot J_{drivetrain} \tag{5.5}$$

where α is the angular acceleration of the motor and $J_{drivetrain}$ is the total inertia of the drivetrain. The inertia of the drivetrain is calculated according to

$$J_{drivetrain} = J_{front\_sprocket} + J_{chain} + (\frac{n_1}{n_2})^2(J_{tire} + J_{back\_sprocket}) \tag{5.6}$$

where $J_{front\_sprocket}$ is the inertia of the front sprocket, J_{chain} is the inertia of the chain, J_{tire} is the inertia of the tire and $J_{back\_sprocket}$ is the inertia of the back sprocket. The inertia of the front and back sprocket $J_{sprocket}$ is approximated using the formula for cylinders [13], according to

$$J_{sprocket} = m_{sprocket} \cdot r_{sprocket}^2 \tag{5.7}$$

where $m_{sprocket}$ is the mass of the sprocket and $r_{sprocket}$ is the radius of the sprocket. The chain inertia J_{chain}, is approximated according to

$$J_{chain} = m_{chain} \cdot l_{chain} \cdot (\frac{n_1}{n_2})^2 \tag{5.8}$$

where m_{chain} is the mass of the chain and l_{chain} is the length of the chain. The tire inertia J_{tire}, is approximated according to

$$J_{tire} = m_{tire} R_w^2 \tag{5.9}$$

where m_{tire} is the mass of the tire.

To get the total torque T_{ref}, the external forces acting on the motorcycle are transformed to torque and then added to the internal torque according to

$$T_{ref} = R_w \cdot \frac{n_1}{n_2}(F_{drag} + F_{grav} + F_{roll}) + T_m \tag{5.10}$$

5.1.2 The Motors in The Model

As previously stated the LM is used to output a torque on the shaft, 5.10, and consequently the MUT to simulate a test track. To achieve this the LM uses a cascaded control loop where the inner loop regulates the currents using FOC and the outer loop controls the torque. FOC, Field-Oriented Control, is the controller method used in the Sevcon Gen 4 which converts the three phase stator currents into two orthogonal components: Iq and Id using Park and Clarke transformations. PI controllers are then used to regulate Iq and Id before they are converted back into phase voltages using inverse Park and inverse Clarke transform [16]. PWM signals are then generated to control the supplied phase voltages to the BLDC motor via the inverter, see figure 5.3.

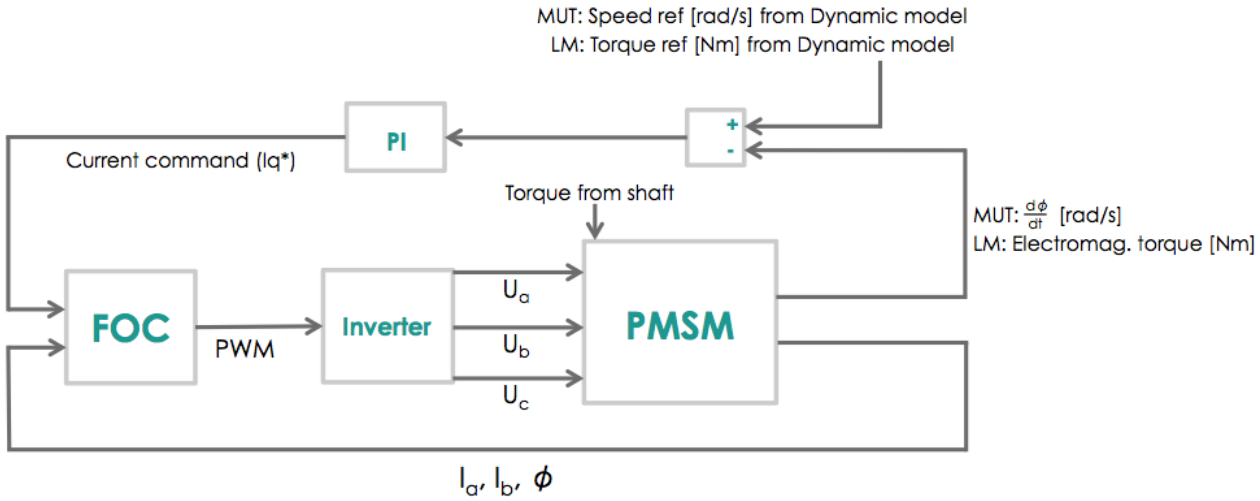

Figure 5.3: Cascaded control loops for MUT and LM

The advantage of using FOC is that the Iq current is directly proportional to the induced torque which helps facilitate torque control. However since the mathematical equations used to convert the signals are rather complex the decision was made to use an existing Simulink model of FOC and modify it to suit our purpose. Link to the original model can be found under bibliography [5].

The controller gains for the inner current loop can be calculated from the motors electrical parameters [2] according to

$$K_p = 2 \cdot \zeta \cdot \omega_0 \cdot L - R \tag{5.11}$$

and

$$K_i = \omega_0^2 \cdot L \tag{5.12}$$

where ω_0 is the natural frequency of the current loop [rad/s] and ζ is the damping of the loop. The PI-values for the cascaded torque were tuned manually to achieve a desired performance of the closed loop torque control. The idea is to scale the torque proportionally to the Iq current.

The MUT used in the model is based on the Revolt Pro 160 just as the LM and not the actual MUT used in the test rig. This was done to properly be able to simulate a motor bike since the actual MUT used in the rig is too weak and only used to test the functionality of the test rig. The MUT in the model uses FOC the same way as the LM to control the currents in the inner loop, see figure 5.3. The outer loop is a PI-controller to regulate the speed. The equations used for the speed control parameters can be tuned by taking the inertia into account [2] according to

$$K_{p\omega} = 2 \cdot \zeta_\omega \cdot \omega_{0\omega} \cdot J \tag{5.13}$$

and

$$K_{i\omega} = \omega_{0\omega}^2 \cdot J \tag{5.14}$$

where J is the total mechanical inertia of the drivetrain, $\omega_{0\omega}$ is the natural frequency of the outer velocity loop [rad/s] and ζ_ω is the damping of the outer loop.

5.1.3 The Shaft

The shaft model inputs the angular velocities from each motor. It outputs the shaft transmitted torque according to

$$T_{sh} = K_{sh} \cdot \int (\omega_m - \omega_l)dt + D_{sh} \cdot (\omega_m - \omega_l) \tag{5.15}$$

where T_{sh} is the output shaft torque, K_{sh} is the stiffness and D_{sh} is the damping of the shaft. The shaft torque is then fed back as load torque to each motor block with the sign inverted to the load motor.

5.1.4 Results From Model Simulation

In this section, results from different simulations tests are presented and analyzed. Initial simulations are used to study extreme scenarios such as running the bike at max speed, having a steep uphill track or a steep downhill track. For each test the following units are plotted: torque from the load motor, angular velocity from the MUT, the Iq and Id currents from LM. Plots from the first test, running the bike at max speed can be found below whereas plots from other tests can be found under Appendix B. Each test consists of two arrays of data and a step length. The first array is the height profile [m] of the simulated track where the inputted step length [m] is the distance between each height point in the x-range. The second array consists of the desired bike speed [km/h] between each height point. As previously stated this information is then used to calculate the reference torque from the load motor as well as the reference angular velocity for the MUT.

Simulation Test 1 - Max speed on flat surface:

- height_array = $[0, 0, 0]$
- speed_array = $[85, 85]$
- step_length = 50

Simulation Test 1 simulates a max bike speed of 85 km/h on a flat surface.

The MUT in the model accelerates up to 85 km/h, which corresponds to an angular velocity of approximately 5000 rpm, in less than 2 seconds, see Figure 5.4. During the acceleration phase the LM outputs a torque of approximately 15 Nm and is later reduced to around 10 Nm once the MUT reaches a constant velocity. At the same time it is noticed that the Iq current has an inverted curve compared to the torque and reaches its minimum around -125A, see Figure 5.5. This is due to the fact that the LM is acting as a generator and is actually being driven by the MUT. The Id current is controlled to be zero but starts to oscillate more once the MUT reaches a constant velocity. From test 3, which simulates a steep downhill, Figure B.3 in Appendix B shows that the LM only act as generator during the MUT:s acceleration phase and once it reaches a constant velocity it starts helping to drive the MUT. In this case the Iq current reaches 180A, see Figure B.4, which was set as the current limit according to the Sevcon Gen4. This corresponds to a negative torque output of approximately -22 Nm.

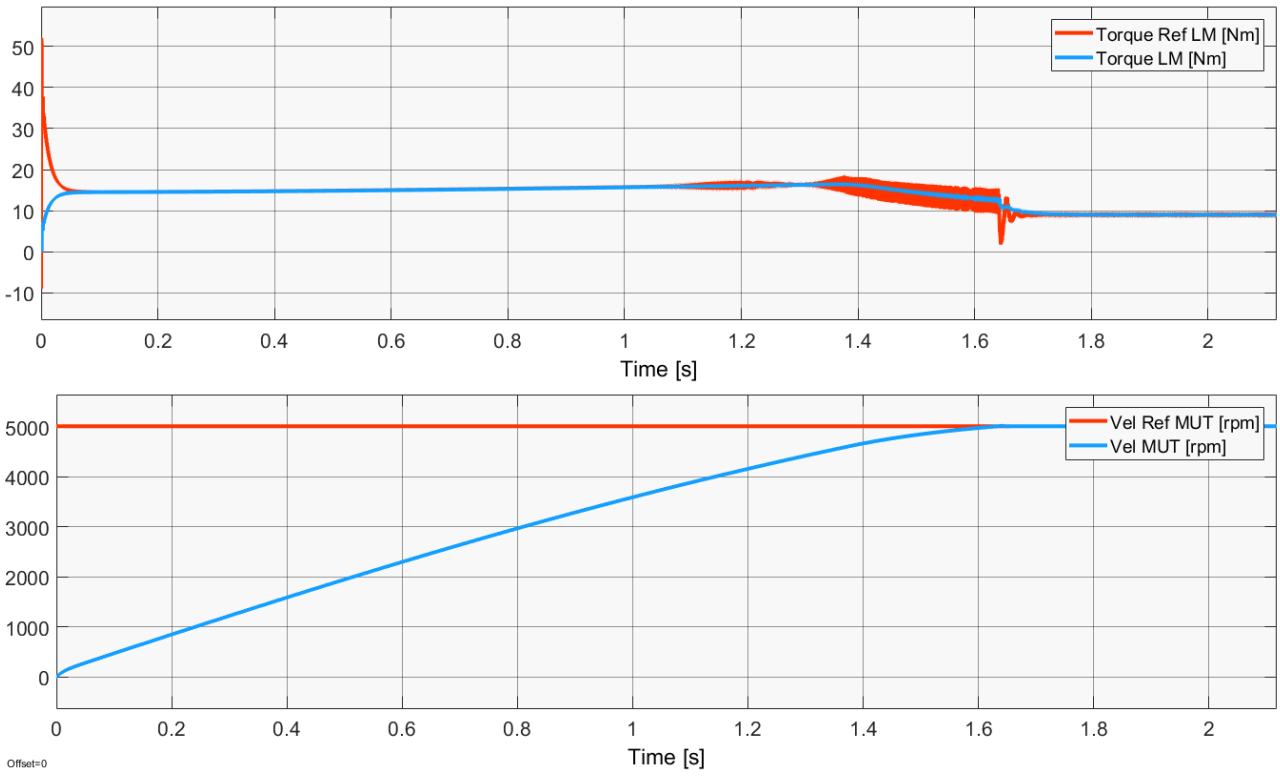

Figure 5.4: Simulation Test 1 - Torque from LM vs Speed from MUT

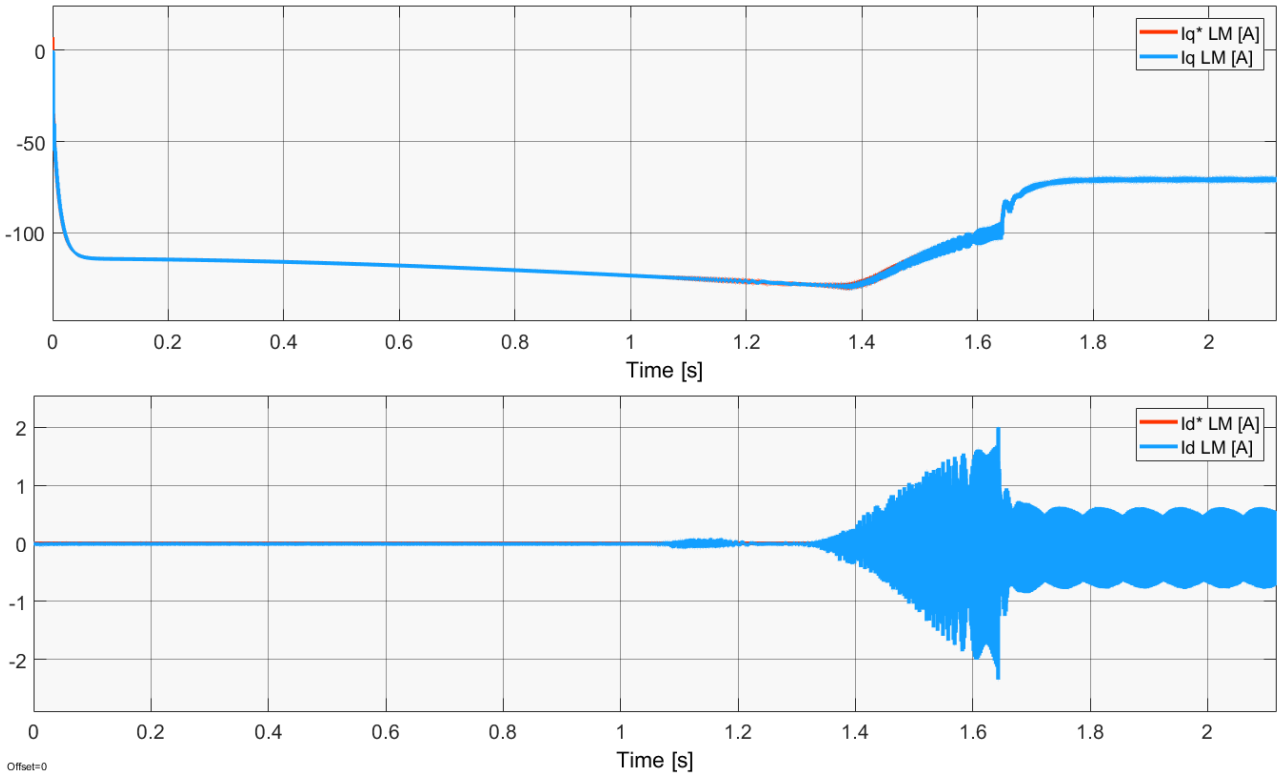

Figure 5.5: Simulation Test 1 - Iq and Id current on LM

Limitations

Controlling the BLDC in a cascaded loop using FOC is highly sensitive to motor parameters. If the motor is not correctly represented in the control scheme the system tends to become unstable. Since there was no data sheet for the motor with the motor parameters the manufacturer was contacted to provide the parameters. A few of the values were gathered this way, however a few had to be researched online or inferred through physical measurements. There are some uncertainties about the validity of the parameters used.

Some of the noticed errors in the model: the motors accelerates too fast, maximum torque output seems to be limited around +-20 Nm however it should be able to output up to +-40 Nm. Reasons for this could be as previously stated that the motor parameters aren't the completely right values. Other reason could be factors that not included in the model, e.g. friction from the belt drive.

Another limitation include computational power. The model uses a lot of computations which make it rather slow and time consuming. Therefore only shorter simulation tests have been made to see the behavior of the rig instead of simulating entire tracks. Increasing sample time is one way to make the simulations faster but at the same time it can decrease the performance of the controllers.

5.1.5 Sevcon Gen4

Sevcon Gen4 is used to control the LM whose specification are given in figure F.2 under Appendix F. Through an iterative process the PI-controller gains are chosen to make the motor stable. The PI parameters used are seen in figure F.4 under Appendix F. The maximum overspeed limit was set to 5000 *rpm* such that its within the limit of operation of the torque sensor.

6 - Embedded system

This chapter presents the functionality of the embedded system in the test rig, the network layout and protocols used, and a detailed description of the state model of each node on the network.

6.1 Overview

The distributed system of microcontrollers is responsible for monitoring the state of the test rig, power management, controlling the load of the MUT and recording data for logging purposes. This section will present an overview of this system.

As seen in the network topology presented in Figure 6.1 the system consists of three main nodes connected to a CAN bus, each node having a defined set of responsibilities which is presented in the sections below. The torque control- and data logging nodes are implemented on Arduino Mega 2560, and the power management node is implemented on an Arduino Uno. The source code for the different nodes can be found in Appendix I.

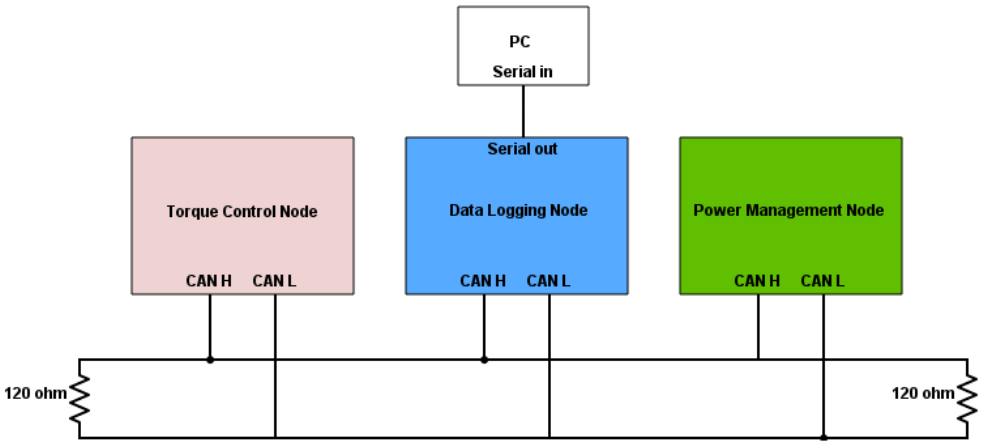

Figure 6.1: The network topology of the distributed system.

6.1.1 Torque Control Node

The torque control node is responsible for the outer control loop of the load torque. It keeps the height profile of the track to be simulated, as well as the data relevant to compute the bike dynamics such as mass, gear ratio, wheel radius etc. found in Table 5.1.

In order to keep the latency of the control system as low as possible this node does all the sampling of sensor data necessary for control purposes, namely the velocity and torque of the shaft, which is read from the torque sensor. Details on the how data is converted from the sensor output can be found in Section 3.2.4. This data is used in the computation of the torque reference based on the motor cycle dynamics, which is converted to an analog output to set the torque reference of the load module controller.

Figure 6.2 shows an overview of the torque control node setup.

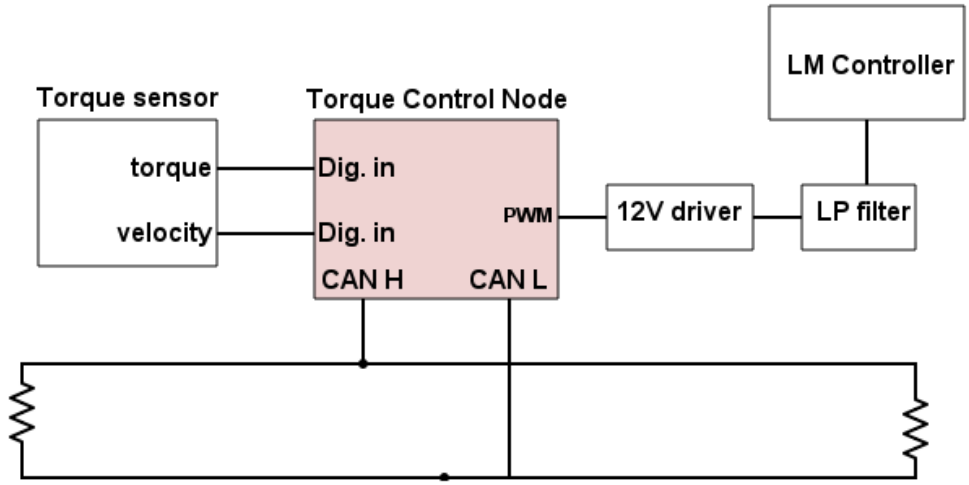

Figure 6.2: Overview of the torque control node showing the connections to peripheral components.

6.1.2 Power Management Node

The power management node is responsible for monitoring the state of the battery banks and temperatures for controlling the cooling system of the rig. It has the following functionality:

- Monitor:
  - The voltage level of the battery bank used to power the load motor.
  - The voltage level of the power supply battery.
  - The voltage level of the MUT battery.
  - The temperature of the load motor.
  - The temperature of the load motor controller.
  - The temperature of the dump load.

- Control the solid state relay to route the generated current from the load motor as described in section 3.2.3.
- Control the fans for cooling the load motor, the load motor controller and the dump load.
- Send the measured voltages and the dump load temperature on the CAN bus for data logging.

The node has three voltage sensors and a temperature sensor connected to the analog input pins of the controller. The voltage sensors are used to measure the voltage of the batteries, and the temperature sensor is used for measuring the temperature of the dump load. The temperatures of the load motor and controller are received on the CAN bus.
For fan control, the node is connected to the fan circuits described in section 3.3.5. Figure 6.3 presents an overview of the power management node and the external components connected to it.

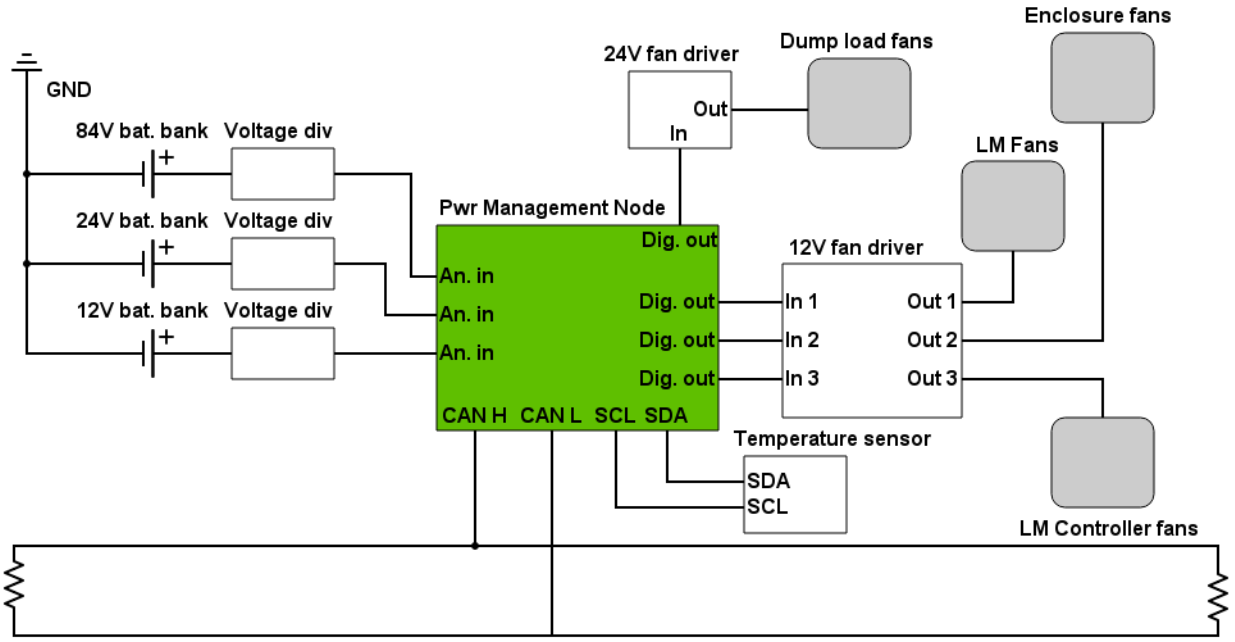

Figure 6.3: Overview of the power management node showing the connections to peripheral components.

6.1.3 Data Logging Node

The data logging node is the interface between the CAN network and the computer node where the log data is stored and plotted. It monitors the CAN bus for log data and sends it via serial to the PC node. It is also responsible for receiving the track profile from the PC and transmitting it via CAN to the torque control node on system startup.

6.1.4 Track profile

The height profile of the track to be simulated is generated on the PC node. Details on track generation can be found in Section 6.2. If the code for the data logging- and torque control node is compiled with *GET_TRACK_EN* set to 1, the nodes will enter a initialization state waiting to receive a track profile. The profile is transmitted to the data logging node via serial, and when the transmission is completed, the data logging node proceeds to send the track via the CAN bus to the torque node.

6.1.5 State Chart

The system state flow model is developed to serve as a basis for embedded coding and to clearly visualize test- and use cases. In the design phase it also serves as an effective tool to regulate behavior in both nominal and non-nominal conditions. The software utilized for development is Simulink. Initial work is pertained to defining node functionality, that is, exploring concurrent and exclusive operating modes. The models abstraction level is then chosen and relevant input signals and output states at said level are found. Each node sets an internal operating state based on sensor readings, functionality and user commands. Permutations of the node-level states are then evaluated at a central hub to set a system wide state for the test rig, which is broadcasted to the node network in its entirety.

General Node Functionality

The following states are present in all nodes.

- Base - This controls which state the node is in (init, normal, warning or critical). The state change is made after evaluation of a local truth table, informed by the sensor values and operational modes of the peripherals. When a change of state is triggered, the node in question relays this event to the logging node.
- Communication - This controls the reception and sending of messages from and to the node. The sending in the model is executed in a frequency of 10Hz.

Power Management Node

The power management node has three unique and concurrent state archetypes.

- Voltage Sensor - This classifies the voltages dependent on sensor readings. All the voltages have the classes normal and under. For the 84 Volt Battery there is also an

over voltage sub-state as we expect the load motor to run in regenerative mode for the majority of testing.

- Temperature Sensor - This classifies the temperatures into classes low, normal, warm and hot, with different thresholds given the context of sensing.
- Peripheral Driver - Handles operation of the solid state relay and the fans. These peripherals can all be active, in which state they can either be turned on or off, or disabled, due to some error.

Figure G.1 in Appendix G presents an overview of the state flow in the power management node.

Data Logging Node

The data logging node has two unique and concurrent state archetypes:

- Node Monitor - This concurrent state interprets the the collected states of all the nodes in the system, and decides how this should affect system operations.
- System - If the node monitor concludes that operation must be halted. The system is turned off.

Figure G.2 in Appendix G presents an overview of the state flow in the power management node.

Torque Control Node

The torque control node has one unique and concurrent state archetypes:

- Control - Gets torque on shaft via sensor. Calculates reference torque given speed of the shaft. The angular velocity is related to a linear velocity given simulated bike parameters and is then integrated and derived to produce a reference torque taking external and internal forces into account. The difference of the sensor read torque and the calculated reference torque is used to compute the control signal and to actuate the load motor.

Figure G.3 in Appendix G presents an overview of the state flow in the power management node.

6.1.6 Network

The three nodes mentioned above are together with the Sevcon connected on a CAN network. CAN, Controller Area Network, is a communication protocol which allows microcontroller nodes to communicate without a governing computer. It is a fast and secure way of sending messages often used in the automotive industry and for control applications. The nodes are broadcasting messages on the networking, meaning there are no direct messages sent between two nodes. Instead, the messages are marked with an identifier to indicate on the content of the message, to which node it was sent to or by which node it was sent from[3].

The speed of network, called the baud rate was set to 500 kbps. The power management node and the torque control node are sending data to the data logging node every 100 ms. The table below list the different messages sent to the data logging node.

From node	ID	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Power management	0x1	Status	Dumpload temp [°C]						
Power management	0x1	Status	12V Battery [V]						
Power management	0x1	Status	24V Battery [V]						
Power management	0x1	Status	84V Battery [V]						
Torque control	0x2	Staus	Velocity [rpm]						
Torque control	0x2	Status	Torque [Nm]						

Table 6.1: Messages sent to the data logging node

A message sent on the bus can contain up to 8 bytes of data. For the receiving node to be able to identify the content of the message, all messages sent on the bus must follow a defined format, where the first byte of the data field contains information on the data in the remaining 7 bytes. The format of this header byte is presented in Table 6.2. By parsing the header byte, the receiving node gets information on what data the message contains, who sent it, and the state of the sender.

bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
Node ID		Signal Type				Node State	

Table 6.2: Status byte structure

By formatting the messages this way, no processing needs to be performed on the data in the data logging node when transmitting the data received on the CAN bus on the serial bus to the PC. This means that the processing power of the PC can be utilized for decoding the messages for data logging, for visualization and storing purposes.

CANopen

The Sevcon controller communicates using CANopen which builds on CAN but implements a network handling system[7]. It supports updates of node status through heartbeats, an object dictionary with indices and subindices of stored data and protocols for communicating with other nodes in the network.

PDO, process data object, are used to share real time data between nodes. PDO messages can be sent out asynchronously or synchronously. Asynchronous messages are sent out on events. For an example on updates in input data that are mapped to a PDO. Synchronous messages are sent out upon a request by a SYNC message from a sync-producing node[14]. This allows updates of the entire system at the same time.

For this project the PDOs are the only necessary functionality of CAN open during runtime. Sevcon allows for data transmission of five different PDOs with different identifiers. The messages are sent out synchronously on an internal event timer. The timer was set to send out messages every 20 ms. The chosen data to be sent from Sevcon are sorted by the node it is going to. All data required by the one node are sent in one message so in total three messages are sent out on the network by Sevcon every 20 ms. An example message is listed in table 6.3.

ID	length [bytes]	Byte 0	Byte 1	Byte 2	Byte 3
0x300	3	Heatsink [°C]		Motor temp [°C]	
0x301	4	Velocity [rpm]			
0x302	4	Iq [A]	Uq [A]	Current [A]	

Table 6.3: PDOs sent out from Sevcon

As the messages from the Sevcon do not adhere to the same format as the messages sent by the microcontroller nodes, any node on the CAN bus that receives a message must check the CAN id of the sender in order to know to decode the data. As seen in Table 6.3, the CAN id in the messages transmitted by the Sevcon contains information on the content of the message.

6.2 Desktop UI

The user interface utilizes the Matlab environment. If the system is in the initializing state, a track profile can be uploaded through an m-script, see code in Appendix H, which is done by sending a 1024-byte array containing 1023 points of elevation data and one integer value denoting the resolution of the given track. That is, the last 8-bit integer informs the distance in meters between the prior values. As such, tracks ranging from $1 * 1022 = 1022$ meters to $255 * 1022 = 260610$ meters with a maximum height difference of 255 meters between the highest and lowest point can be modeled. The output of the system (node

states, sensor values and calculated values such as distance traveled and torque reference) are communicated from the logging node to Simulink via serial communication. The data sent over this channel corresponds to bytes 0 through 7 in table 6.1. Runtime interactivity between the Desktop UI and the distributed system is as of now limited.

6.2.1 Simulink

Using Simulinks serial communication library we receive the raw data coming in from the logging node in chunks of 8 bytes. Byte 0 is parsed for node ID, node state and signal type. Bytes 1 through 4 are combined into a 32-bit floating point number using bit-shift and bitwise *OR* operations. This number is then labeled and routed according to its signal type as defined in byte 0 using an if-action-subsystem. A figure showing the layout of the user interface can be seen in Appendix H.

7 - Results and Discussion

A functional and modular mechanical structure was designed and constructed to allow testing of different drive-trains. Sensor circuits for diversion load temperature, battery voltages, shaft speed and shaft torque were designed and unit-tested. Means of accessing additional data points via the Sevcon Gen4 controller over CANOpen (e.g. load motor and controller temperature, load motor current) were explored and verified. A distributed system of Arduino platforms was designed by means of state flow and implemented in the system. A message format atop the CAN and serial protocols was constructed and message handling functionality thereof was verified. Communication between PC and the system as well as a basic user interface in Simulink was implemented. A framework for state handling was authored but not utilized. Circuits for outer loop control of torque (including low pass filtering) were designed and implemented. Power systems were dimensioned and designed to safely allow for different modes of operation of the load motor (driving and regenerative braking), encompassing the drive of the solid state relay to divert excess energy to the dump load and control of fans to dissipate heat build up.

Most issues became apparent and exacerbated during and post the integration phase. Due to timing and a poorly set up environment to test the load motor independently the realization of how unstable it was controlled came late in the development process. It seems that, in general, a codified road map that better organized what work could be done in parallel and what had to be done in series would have served the project greatly. As the project was executed a lot of unforeseen interdependencies between groups caused competing needs for specific hardware and bottlenecks in work-flow. This has resulted in a poorly verified system since some subsystem functionality is now gated by the incorrect drive of the load motor. Lessons can be learned with regard to assuming the difficulty of tasks as well. In hindsight, configuration of the Sevcon-controller did not receive the amount of attention it warranted, especially considering that the main functionality of the rig is to put out a given torque on the shaft. Cross-referencing the behavior of the motor to the motor model in Simulink would also help in further development of outer control loops.

Evaluating the health of the project as it is, the missing piece of the puzzle seems to be verified motor parameters and a tuned controller. In theory, validation of system functionality and further development of user interface could commence immediately following effective torque control of the load motor.

With regard to what may seem as a counter intuitive design choice, that in not utilizing a regenerative cycle and simply dumping excess energy, it serves to facilitate evaluation of the entire drive train including battery performance. In any case, if a complete regenerative cycle is sought after, the dump load mechanic implemented in the rig could act like a buffer in spikes at points where the rated charging current is exceeded.

8 - Future Work

This chapter presents suggested areas for future development that has been identified during the course of the project, as well as planned functionality that was designed but not implemented.

8.1 Electrical

One of the main issues of the electrical system in the test rig is pertaining to the PCB enclosure box that is described in Chapter 4 Section 4.9. Going forward with the project, it is suggested that effort is put into securing the circuits of the test rig to a non conducting panel, and that wiring inside the box is made more robust. This will serve to reduce the risk of short circuits between PCB's and enclosure, and simplify the process of verifying functionality of each circuit. To aid in this work, a complete representation of the pinout of the connectors attached the enclosure is represented in Appendix D.

Another point of improvement would be to reduce the number of circuit boards, by grouping them together on larger boards, which would reduce the wiring needed inside the enclosure and reduce the complexity of the electronic system. During the design and implementation phase of the project it has been beneficial in terms of unit testing functionality to design each circuit on an individual board. For a final design however the complexity of setting up the system gets quite cumbersome and prone to error.

The functionality of the dump load described in Chapter 3 Section 3.2.3 needs to be tested and verified. The suggested way would be to run the system with the load motor in generation mode, and verify that the current flows to the power resistors when the battery bank is in a fully charged state. As the Power Management Node of the embedded system is involved in this functionality this will also serve to verify that functionality. In an initial test phase it may be good to consider independent testing of the functionality electrical- and embedded system.

8.2 Embedded

The code base for the three nodes found in Appendix I is thought to serve as a platform for further development. Care has been taken to create a maintainable structure, and shared interfaces and information between nodes is kept in a Node library to avoid duplication of

information.

8.2.1 Torque Control Node

The main area for further development in the Torque Control Node is to implement outer closed control loop for the LM torque. The Torque Control Node currently computes the torque reference based on the track profile and motor cycle dynamics. What needs to be further developed, tested and verified is the computation of control signal to set the break and throttle inputs of the LMC. The details of how the outer control loop should be implemented is at this point somewhat unclear, as more insight is needed into how the LMC interprets the throttle and break inputs in terms of resulting torque at the shaft.

Regarding the sampled values from the torque sensor, the shaft torque has been verified to be measured accurately as presented in Appendix B. What could be improved upon is determining the direction of shaft velocity. As the output from the torque sensor does not convey directional information, this is currently inferred from the direction of the torque. This is not an accurate way of determining direction as a negative torque only implies that the rate of change in velocity is negative. It is recommended to consider adding a separate quadrature encoder to the system, to be able to accurately determine shaft speed. There is also a possibility of reading the velocity from the LMC which can broadcast directional velocity of the LM on the CAN bus.

Further development also includes evaluation of the base state as a function of concurrent states described in Chapter 6 Section 6.1.5 and developing the handling of the track profile. Functionality should be implemented such that the track can be stored on the EEPROM, such that a new track need not be received each time the system is rebooted. An intuitive way for the user to select the startup mode of the system should be implemented to improve the usability of the system.

8.2.2 Power Management Node

For the Power Management Node, effort should be put into verifying the functionality of power routing between the LM battery bank and the dump load as mentioned in Section 8.1. Beyond that, it currently lacks the functionality of reading the temperatures of the LM and LMC from the CAN bus, which is meant to be used for controlling the cooling of these components.

8.2.3 Data Logging Node

The Data Logging Node has implemented functionality for filtering out the LMC messages sent on the bus, to be able to assure proper formatting of all messages sent on the serial connection to the Desktop UI. This has not been verified to work, but it should be a simple matter of making sure that the definitions for the CAN id's in the IO module of the Node library is set according to the configuration of the LMC.

Further suggestions for implementation is to work on the master control structure of the distributed system. The Data Logging Node should be able to, based on the base states of the slave nodes, issue commands on the CAN bus, to at least be able to shut the system down if a critical state is detected in the system.

8.2.4 Desktop UI

The Desktop UI could mainly be improved upon in terms of runtime interactivity with the embedded system. Ideally, it should function as a control panel for the test rig, with the ability to initiate and stop a track simulation, load new track profiles to the system in an intuitive and seamless way, and retrieve and analyze data from tests. This kind of activity is now limited, and the flow of information is one way. It could be useful to look into other development platforms than Matlab Simulink for this functionality, such as Python or Java which could simplify the serial communication with the Data Logging Node.

Some effort should also be put into developing a fluid user interface for generation of track profiles from geographical data. The Google Maps Evaluation API [9] could be useful for this.

8.3 LMC

Future work on the LMC of the test rigs is one of the most critical points for continued development, as it plays an important role in the overall functionality. A lot of the functionality in the other parts of the system is dependent on load control, both in terms of being able to test and verify behavior, or for development purposes. As such, this should be considered a high priority for any continuation of this project. The main points of improvement is to get stable torque control by controlling the throttle- and break inputs to the controller fully functional and verified. It is suspected that one of the reasons for the unstable behavior observed in the torque control is related to a poor representation of the motor parameters. As such, determining these parameters accurately should be made a priority.

For the embedded system of the rig to function properly, the controller should also be configured to send the data needed for logging and control and control purposes on the CAN Open interface described in Chapter 6.

8.4 Modeling

Future work pertaining to modeling should focus on verify the accuracy of the results produced by the model. This includes researching the reason for the unreasonably high acceleration produced by the motor models. As stated in chapter 5, under limitations of modeling, it is critical that the motor parameters are correct when modeling a BLDC motor using FOC. Therefore verifying and validating that the motor parameters are correct will enhance the performance of the model. It is also possible that the current model does not take into account all sources of losses in the system, such as the friction in the belt drive which if added would dampen the dynamics of the system.

Beyond that, an investigation on how different time steps affect the model could speed up the simulation without reducing the performance of the controllers and motors would be beneficial. Investigating the bandwidth (natural frequency) of the inner and outer control loops could improve the ability of tuning the controller gains.

Finally the Simulink model should be verified and validated by comparing the results from tests on the real system performance with simulated data.

8.5 Mechanical

The mechanical design has a few areas which could be improved upon. Due to the length of the transmission belt, there is some unwanted vibrations in the rig at higher rotational speeds. This could be solved by adding an idler pulley between the two driving pulley to serve as a tensioner for the belt.

Another problem encountered is that in the current design, there is no simple way to fix the output shaft of the load module in place. This is something that would be useful when for example working on the torque control of the load motor. One possible solution for this would be to extend the output shaft, adding another keyed joint, which would create a simple solution for attaching a locking mechanism to the end of the shaft, without the need for removing the pulley for the belt drive.

Finally, the mechanical design could be improved by simplifying the process of connecting the shafts of the load module and the MUT together, for running tests without a belt drive transmission.

Bibliography

[1] *A Primer on Electromagnetic Shielding*. http://www.dtbtest.com/electromagnetic-shielding-primer.aspx.

[2] Viktor Bobek. *PMSM Electrical Parameters Measurement*. http://cache.freescale.com/files/microcontrollers/doc/app_note/AN4680.pdf.

[3] *Controller Area Network (CAN) Overview*. http://www.ni.com/white-paper/2732/en/. Accessed: November, 2017.

[4] *Datasheet for mayr ROBA-ES, K.940.V13.EN*. https://www.mayr.com/synchronisation/documentations/k_940_v13_en_08_08_2013.pdf.

[5] *Field-Oriented Control of Permanent Magnet Synchronous Machine*. https://se.mathworks.com/help/ecoder/examples/field-oriented-control-of-permanent-magnet-synchronous-machine.html.

[6] Talha Khan Daniel Söderman Mooaj Bhana Sebastian Bulow Sofia Olsson Johan Blomqvist Jennifer Tannfelt Wu Fredrik Lundeberg Iñigo Alvarez. *CAKE - Final Report SOTA*. 2017.

[7] *Gen4 Applications Reference Manual*. https://www.thunderstruck-ev.com/images/Gen4ProductManualV34.pdf. Accessed: September, 2017.

[8] *Global EV Outlook 2017*. https://www.iea.org/publications/freepublications/publication/GlobalEVOutlook2017.pdf.

[9] *Google Maps Elevation API*. https://developers.google.com/maps/documentation/elevation/start.

[10] *High Power Wire Wound Resistor - Type TE Series*. http://www.mouser.com/ds/2/418/NG_DS_9-1773453-2_C-734895.pdf.

[11] *How to Select the Right Fan or Blower*. http://www.sunon.com/uFiles/file/03_products/07-Technology/004.pdf.

[12] *Mekanex Belt Drive Technical Information*. https://www.mekanex.se/produkter/komponenter/kuggremsdrifter/.

[13] *Moment of Inertia for Uniform Objects*. http://www.livephysics.com/physical-constants/mechanics-pc/moment-inertia-uniform-objects/.

[14] *Process data object (PDO)*. https://www.can-cia.org/can-knowledge/canopen/pdo-protocol, note = Accessed: November, 2017.

[15] *Reduction EMI of BLDC Motor Drive Based on Software Analysis*. https://www.hindawi.com/journals/amse/2016/1497360/.

[16] *Sensorless PMSM Field-Oriented Control*. https://www.nxp.com/docs/en/reference-manual/DRM148.pdf.

[17] *Thermal Conductivity of Common Materials and Gases*. https://www.engineeringtoolbox.com/thermal-conductivity-d_429.html.

Appendices

A - Requirements

The requirements were written in SysML modeling language following the standard requirement relationships as detailed in its specifications. The overall requirements block is presented in Figure A.1.

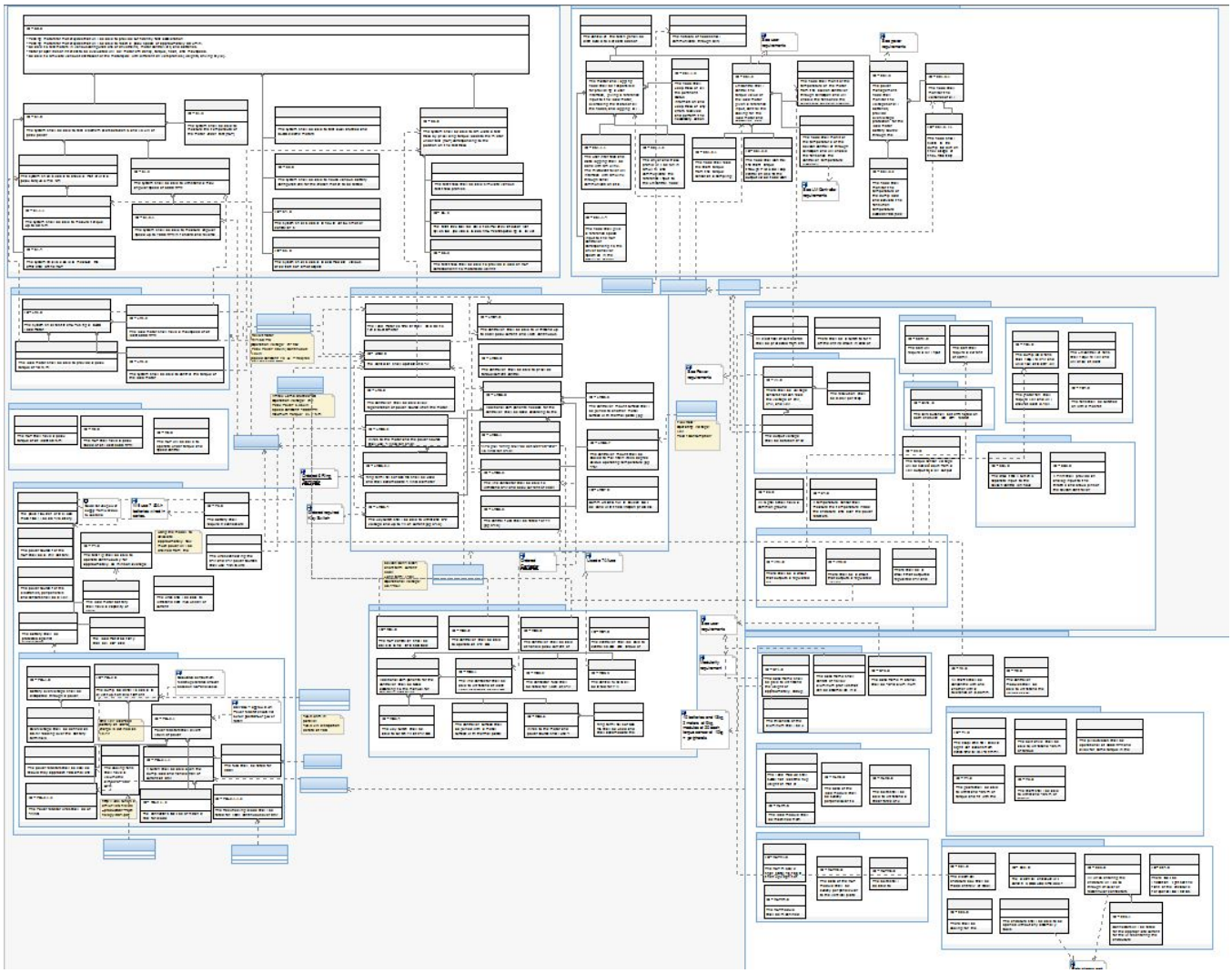

Figure A.1: Overall Requirements

For the sake of clarity, these requirements have also been presented in tabular form in Table A.1 below.

ID	Name	Specification
BF1.0	Weight	The base frame shall be able to withstand the weight of approximately 250kg

BF2.0	Modularity	The base frame shall consist of hollow aluminum profiles that can be assembled in a grid pattern
BF3.0	Material	The base frame material shall be 4040 aluminum
CAN1.0	CAN Bus	The CAN bus shall be terminated on both ends with 120 ohm resistor
E1.0	EMC	All electrical circuit boards shall be protected from EMI
E2.0	Switch	There shall be a switch to turn off the entire circuit in case of emergency
E3.0	Grounding	All signals shall have a common ground
E4.0	Temperature sensor	A temperature sensor shall measure the temperature inside the enclosure and over the power resistors
E5.0	Torque Sensor Divider	The torque sensor voltage will be scaled down from a 12V output to a 5V output
EE1.0	Material	The electrical enclosure box shall be made entirely of steel
EE2.0	DC Busses	The electrical enclosure will contain 3 DC busses made of copper metal
EE3.0	Connectors	All wires entering the enclosure will do so through circular or rectangular connectors
EE3.1	Current Requirements	Connectors will be rated for the appropriate current for the wires entering the enclousure
EE4.0	LEDs	There shall be indication lights at the front of the enclosure for operational status
EE5.0	Cooling	There shall be cooling for the enclosure
EE6.0	Opening	The enclosure shall be able to be opened without any assembly tools
ES1.0	Distributed System	The control of the test rig shall be distributed to 3 areas/nodes of control
ES1.1	Master and Logging	The master and logging node shall be responsible for providing a user interface, overseeing the status of all the nodes, and logging all the pertinent data
ES1.1.1	Serial	The user interface and data logging shall be done with simulink. The microcontroller will interface with simulink via a serial communication interface at a baud rate of 115200

ES1.1.3	Monitoring	The node shall keep track of all the pertinent status information and keep track of any errors received and perform the neccessary action as specified in the state charts
ES1.1.4	Speed control	The node shall give a reference speed input to the MUT controller corresponding to the driver behavior specified in the simulink model
ES1.2	LM Control	LM Control shall control the torque value of the load motor given a reference input, control the cooling for the load motor and controller, and monitor its status
ES1.2.1	Torque	The node shall read the shaft torque from the torque sensor at a sampling rate of 20 Hz
ES1.2.2	Control	The node shall control the shaft torque through PI closed loop control and send the output value to Sevcon
ES1.2.3.0	Track	The track profile shall be received from the data logging node via CAN on system start up.
ES1.2.3.1	Track	The track profile will be kept on the torque control node which will compute the torque reference for the LMC.
ES1.3	Power Management	The power management node shall monitor the voltages of all batteries, provide overvoltage protection for the load motor battery source through the dump load, control the cooling for the dump load, and monitor its status
ES1.3.1	Voltage Monitoring	The node shall monitor the voltages of all the battery sources
ES1.3.1.1	LM Battery	The node shall switch on the dump load when the voltage of the LM battery exceeds 89V
ES1.3.2	Dump Load Cooling	The node shall monitor the temperature of the dump load and activate the fans when temperature exceeds 70C (see dump load requirements)
ES1.3.3	Controller Cooling	The node shall monitor the temperature of the Sevcon controller through CANOpen and will enable the fans once the controller temperature exceeds critical levels
ES1.3.4	Motor Cooling	The node shall monitor the temperature of the motor from the Sevcon controller through CANOpen and will enable the fans once the motor temperature exceeds critical levels

ES2.0	CAN	The network of nodes shall communicate through CAN
FD1.0	Dump load fans	The dump load fans shall require 24V and will draw at least 7.2A
FD2.0	LM controller Fan	The LM controller fans shall require 12V and will draw at least 1.35A
FD3.0	Motor Fan	The motor fan shall require 12V and will draw at least 0.45A
FD4.0	Switch	The fans shall be switched on with a mosfet
LM1.0	LoadMotor	The system shall test the MUT using a BLDC load motor
LM1.1	LMTorque	The load motor shall be able to provide a peak torque of 40 N.m
LM1.2	LMSpeed	The load motor shall have a max speed of at least 6000 RPM
LM1.3	LMControl	The system shall be able to control the torque of the load motor
LM3.0	LMC Regeneration	The controller shall be able allow regeneration of power source when the motor is acting as a generator
LMC1.0	LMC BLDC	The load motor controller shall be able to run a BLDC motor
LMC2.0	LMC Voltage	The controller shall operate at 84V
LMC4.0	LMC Current	The controller shall be able to withstand up to 250A peak current and 150A continuous
LMC5.0	LMC Control	The controller shall be able to provide torque/current control
LMC6.0	LMC Signal Wiring	All signal wires shall be between 20 AWG and 16 AWG
LMC6.0	LMC Components	Additional components needed for the controller shall be sized according to the Sevcon Gen4 User Manual
LMC6.1	LMC Signal Wiring	All signal wiring shall be between 20 and 16 AWG (pg 3-13)
LMC6.2	Power Source and Motor Wires	Wires to the motor and the power source shall use 4 AWG (pg 3-10)
LMC6.2.1	Terminals	Ring terminals of size M8 shall be used and shall accomodate 4 AWG diameter
LMC6.3	Contactor	The line contactor shall be able to withstand 84V and peak current of 250A
LMC6.4	Key Switch	The keyswitch shall be able to withstand 84V voltage and up to 7A of current (pg 3-15)

LMC6.5	Control Fuse	The control fuse shall be rated for 7A (pg 3-15)
LMC6.6	Thermal Paste	The controller mount surface shall be joined to another metal surface with thermal paste (pg 3-3)
LMC6.7	Cooling	The controller mount shall be cooled to maintain its 25 degree celcius operating temperature (pg 4-6)
LMC7.0	CANopen	Communication with Sevcon shall be done with the CANOpen protocol
M1.0	MUTTorque	The MUT shall have a peak torque of at least 20 N.m
M1.0	Concentricity	All shafts shall be concentric with one another with a tolerance of 0.05mm
M2.0	MUTSpeed	The MUT shall have a peak speed of at least 3500 RPM
M2.0	Controller module	The controller modules shall be able to withstand the weight of the controller if upright
M3.0	MUTControl	The MUT will be able to operate under torque and speed control
MC1.0	MC DC	The MUT controller shall be able to run a series brushed DC motor
MC2.0	MC Voltage	The controller shall be able to operate at 24V DC
MC3.0	MC Current	The controller shall be able ot handle peak current of 140A
MC4.0	MC Control	The controller shall be able to control speed and torque of the motor
MC5.0	MC Components	Additional components for the controller shall be sized according to the manual for the Kelly controller
MC5.1	Contactor	The line contactor shall be able to withstand at least 140A of current at 24V DC
MC5.2	Contactor Fuse	The contactor fuse shall be rated for 150A at 24V DC
MC5.3	Control Fuse	The control fuse shall be rated for 7A
MC5.4	Key Switch	The key switch shall be able to switch 7A at 24V DC
MC5.5	Thermal Paste	The controller surface shall be joined with a metal surface with thermal paste
MC5.6	Power Cables	Wires to the motor and power source shall use 4 AWG

MC5.6.1	Terminals	Ring terminals of size M6 shall be used and shall accomodate the 4AWG diameter
MLM1.0	Load Motor	The load module shall sustain at least the 9kg weight of motor
MLM2.0	90 degree angle	The base of the load module shall be perpendicular to the vertical plate within 0.1 deg
MLM2.1	Shaft Alignment	The load motor shaft and output shaft shall be concentric to the torque sensor shafts within 0.15 mm.
MLM3.0	Bolts	The bolts used for fastening the LM plate shall be able to balance the torque produced by the load motor.
MLM4.0	Material+thickness	The load module shall be machined from 20mm steel
MUTM1.0	MUT	The MUT module shall sustain at least the 9kg weight of motor
MUTM2.0	90degrees	The base of the MUT module shall be perpendicular to the vertical plate within a tolerance of 0.1 deg
MUTM3.0	Material+thickness	The MUT module shall be machined from 10mm steel
P1.0	Load Motor Battery	The power source for the load motor shall be a 84V battery
P1.1	Capacity	The load motor battery shall have a capacity of 50Ah
P1.2	Type	The load motor battery shall be lead-acid
P1.3	Protection	The battery shall be protected against overvoltage
P2.0	MUT Battery	The power source for the MUT shall be a 24V battery
P3.0	Peripherals Battery	The power source for the electronics, peripeherals and sensors shall be a 12V battery
P4.0	Duration	The test rig shall be able to operate continuously for approximately 30 mins on average
P5.0	Motor Battery Wires	The wires connecting the 84V and 24V power sources shall use 4AWG wire
P6.0	Peripheral Battery Wires	The wires shall be able to withstand continuous 20A of current
PDL1.0	Dump	Battery overvoltage shall be dissipated through a power resistor load

PDL1.1	Overvoltage measure	Overvoltage shall be defined as 88.9V reading over the battery terminals
PDL2.0	Diversion	The dump load shall be able to divert up to 90A of current
PDL2.1	Power resistors	Power resistors shall divert 10kW of power
PDL2.1.1	Switch	A switch shall be able open the dump load and handle 90A of current at 89V
PDL2.1.1.1	Fuse	The fuse shall be rated for 200A
PDL2.1.1.2	Freewheeling diode	The freewheeling diode shall be rated for 100A continuous over 89V
PDL2.1.2	Wires	The Power resistor wires shall be of 4AWG
PDL2.1.3	Connectors	The connectors shall be of M8 and M10 for diode
PDL3.0	Cooling	The power resistors shall be cooled should they approach 70C ambient
PDL4.0	Cooling Fans	The cooling fans shall have a volumetric airflow of 285 CFM
S0.0	Original Statement	- Testing motors for motorcycles that will be able to provide sufficiently fast acceleration - Testing motors for motorcycles that will be able to reach a peak speed of approximately 80 km/h. - Be able to test motors in various configurations of drivetrains, motor controllers, and batteries. - Motor properities of interest to be evaluated will be: motor efficiency, torque, heat, and max speed. - Be able to simulate various test tracks for the motorcycle with different drivers profiles (weights, driving style).
S1.0	MotorTesting	The system shall be able to test electric motors between 6 and 15 kW of peak power
S1.1	PeakTorque	The system shall be able to brake a motor with a peak torque of 40 N.m
S1.1.1	MeasureTorque	The system shall be able to measure torque up to 50 N.m
S1.2	PeakSpeed	The system shall be able to withstand a max angular speed of 6000 RPM
S1.2.1	MeasureSpeed	The system shall be able to measure angular speed up to 7000 RPM in forward and reverse direction
S1.3	MotorTemp	The system shall be able to measure the temperaute of the motor under test (MUT)

S1.4	MotorEfficiency	The system shall be able to measure the efficiency of the MUT
S2.0	MotorModularity	The system shall be able to test axial brushed and BLDC electric motors
S3.0	BatteryModularity	The system shall be able to house various battery configurations for the chosen motor to be tested
S4.0	ControllerModularity	The system shall be able to house various motor controllers
S5.0	DriveModularity	The system shall be able to accomodate various drivetrains of a motorcycle
S6.0	TestTrack	The system shall be able to simulate a test track by providing torque loads to the motor under test (MUT) corresponding to the position on the test track
S6.1	TrackProfile	The test track shall be able simulate various test track profiles
S6.2	Driver	The test track shall be able to simulate various driver styles and provide a load on MUT corresponding to driver weight
S6.3	Motorcycle	The test track shall be able to provide a load on MUT corresponding to motorcycle weight
SC1.0	Switching	The circuit shall switch 3 separate input to the sevcon controller input pins
SC2.0	Throttle and Brake	A PWM shall provide an analog input to the throttle and brake pins of the sevcon controller
SSR1.0	Input	The SSR will require a 5V input
SSR2.0	Current	The SSR shall require a current of 30mA
T1.0	Couplers	The couplers shall allow a slight deflections from concentricity (0.1-0.4mm)
T2.0	Belt Drive	The belt drive shall be able to withstand 40N.m of torque
T3.0	Pillow Block	The pillowblocks shall be operational at 6000 RPM and allow for some torque in the lateral direction
T4.0	Gears	The gears shall be able to withstand 40N.m of torque and fit with the belt drive
T5.0	Shaft	The shaft shall be able to withstand 40N.m of torque
V1.0	Sensor	There shall be voltage sensors that can read the voltage of 84V, 24V, and 12V
V1.1	Output	The output voltage shall be between 0-5V

V1.2	Resolution	The resolution shall be 0.03V per step
VR1.0	5V	There shall be a circuit that outputs a regulated 5V
VR2.0	15V	There shall be a circuit that outputs a regulated +/- 15V
VR3.0	24V	There shall be a circuit that outputs a regualted 24V and supplies at least 10A

Table A.1: Requirements

B - Test Cases

Electrical

The functionality of each electronic board (with the corresponding microcontroller) was tested before connecting all the electronic boards together. In Table B.1, the test cases that were carried out are listed.

Test	Code	Hardware	Input	Expected result	Output	Status
Read 84V from power supply	Power management node	84V volt. divider, power supply	Analog input from votlage divider	Same as measured with multimeter (89V and 87.1V)	89.03 and 87.33 V	Pass
Read 24V battery bank voltage	Power management node	24V volt. divider, battery	Analog input from votlage divider	Same as measured with multimeter (25.4V)	25.35 V	Pass
Read 12V battery voltage	Power management node	12V volt. divider, battery	Analog input from votlage divider	Same as measured with multimeter (11.73V)	11.73 V	Pass
State of 84V Battery	Power management node	84V volt. divider, battery	Analog input from votlage divider	Same as measured with multimeter (87.7) and corresponding state (normal)	87.67 and normal	Pass
State of 24V Battery	Power management node	24V volt. divider, battery	Analog input from votlage divider	Same as measured with multimeter (25.3) and corresponding state (normal)	25.27 and normal	Pass
State and voltage of 12V Battery	Power management node	12V volt. divider, battery	Analog input from votlage divider	Same as measured with multimeter (8.77) and corresponding state (under)	8.77 and under	Pass
SSR on/off	Power management node	12V battery, 5V regulator, SSR driver, SSR	Force "on" state in code	SSR turns on	SSR on	Pass
Fans on/off	Power management node	12V battery, 24V battery, fans driver, fans	Force "on" state in code	Fans turn on	Fans on	Pass

Table B.1: Test cases for the electronic circuits.

Torque Sensor

A test was made to see if the measurements, read from the microcontroller, was of acceptable accuracy. 0.5 m levers were made and attached to one side of the shaft whereas the other side was held fixed. By adding weights to the levers the theoretical torque was easily calculated and could then be compared to the measured values. The results can be found in Table B.2 below. The torque sensor was slightly off calibrated with offset being at -0.01 Nm when the test was conducted.

$$Torque_{theoretical} = mass_{weights} \cdot 9.81 \cdot 0.5[Nm] \tag{B.1}$$

The torque induced from the levers, without any attached weights, were initially measured and then subtracted from all the tests to be able to compare the measured values with

Weight [Kg]	Positive direction [Nm]	Negative direction [Nm]	Theoretical value [Nm]
No weights	0.55	-0.53	-
0.5	2.50	-2.51	+-2.45
1	4.96	-5.01	+-4.91
1.5	7.41	-7.40	+-7.36
2	9.97	-9.98	+-9.81
2.5	12.32	-12.38	+-12.26

Table B.2: Results from torque test

the theoretical values, calculated in Equation B.1. The measured values were all slightly higher compared to their theoretical values. One possible explanation could be that the weights used could have been slightly heavier than 0.5 kg. The test gave a satisfactory result showing that the values from the microcontroller were all within the margin of error.

Simulation

Here plots from simulation tests are presented.

Simulation Test 2 - Steep uphill (see Figure B.1 and B.2):

- height_array = $[0, 15, 30]$
- speed_array = $[30, 30]$
- step_length = 50

Simulation test 3 - Steep downhill (see Figure B.3 and B.4):

- height_array = $[100, 50, 0]$
- speed_array = $[85, 40]$
- step_length = 100

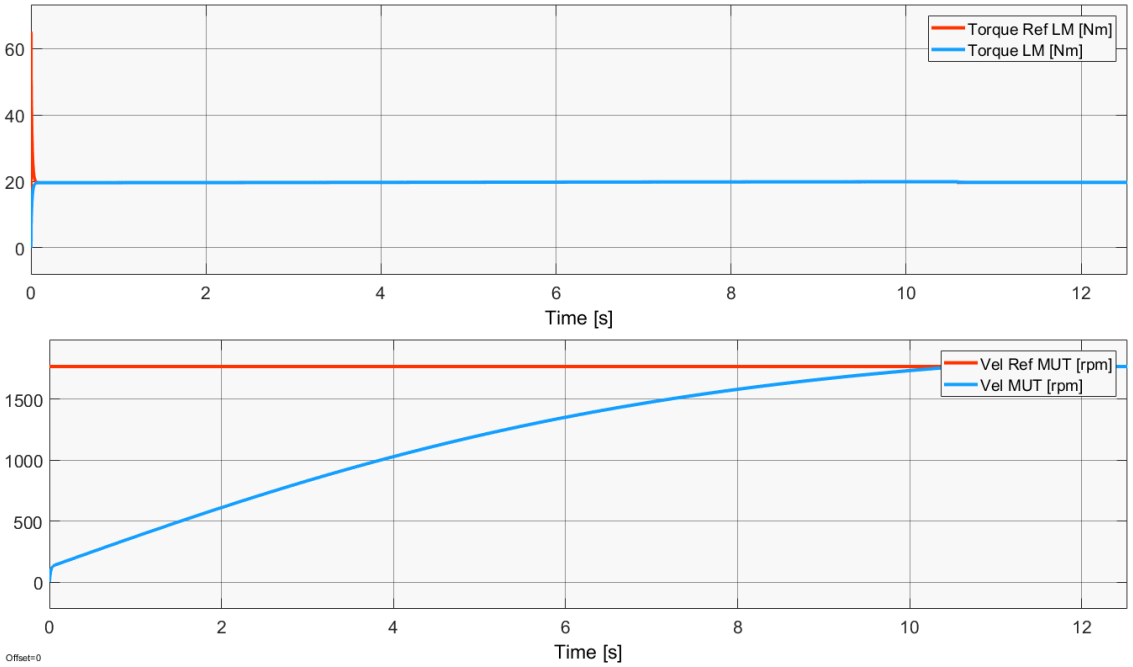

Figure B.1: Simulation Test 2 - Torque from LM vs Speed from MUT

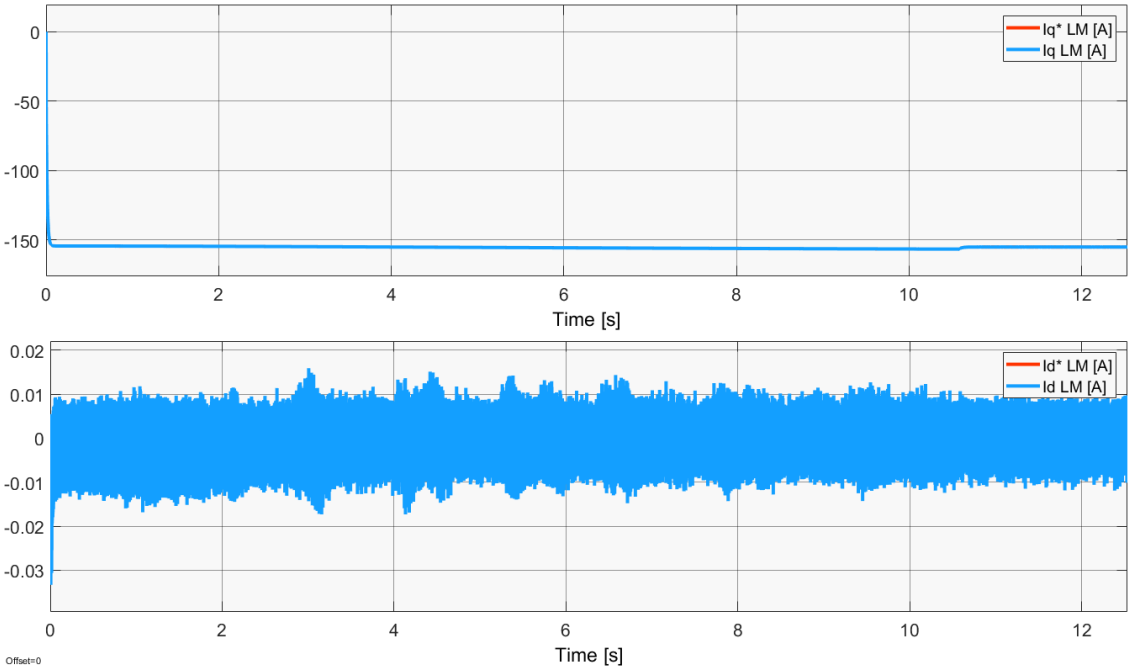

Figure B.2: Simulation Test 2 - Iq and Id current on LM

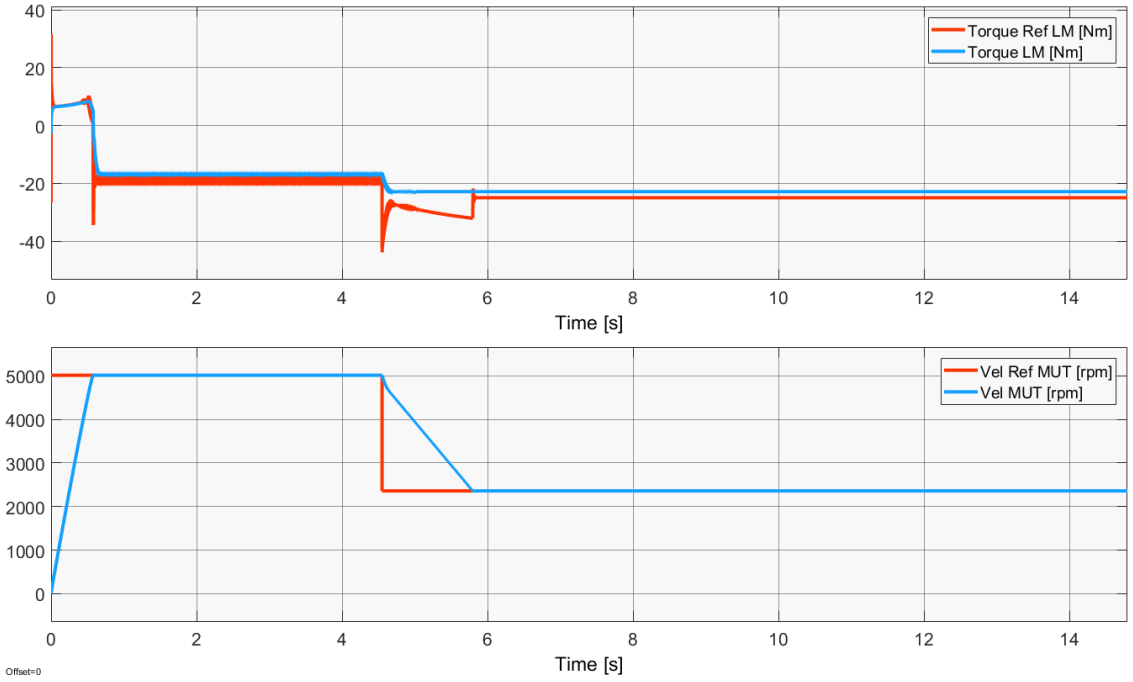

Figure B.3: Test Simulation 3 - Torque from LM vs Speed from MUT

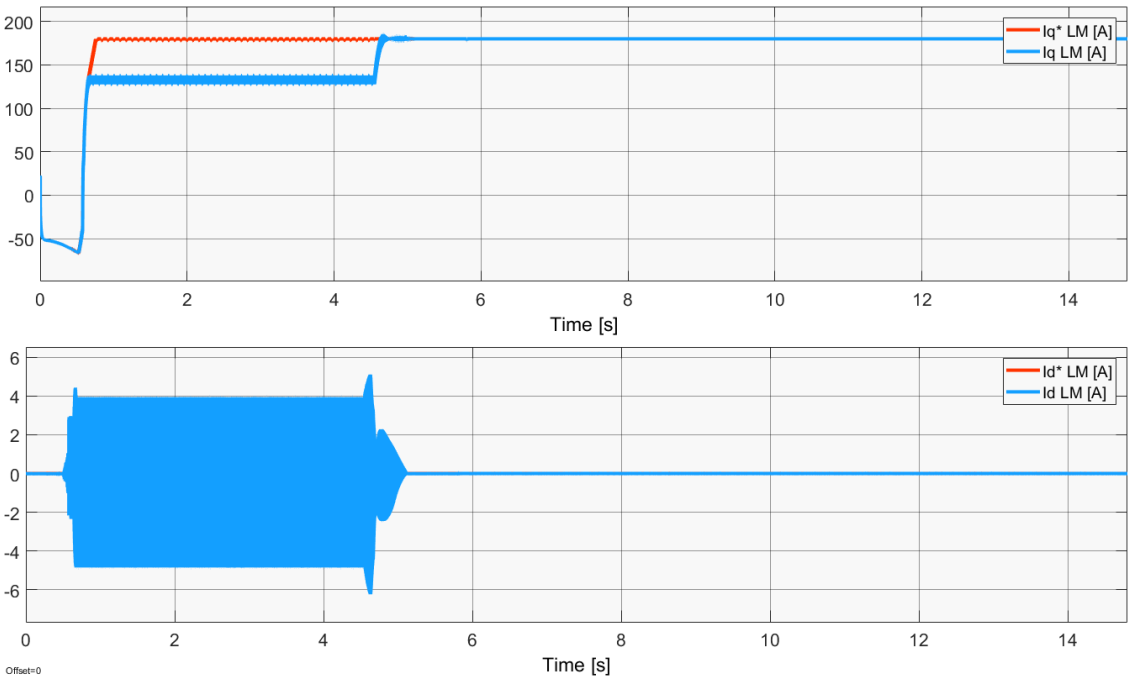

Figure B.4: Test Simulation 3 - Iq and Id current on LM

C - Drawings and Manufacturing

This appendix includes drawings of all manufactured components and static study simulations for LM and MUT (see Figure C.16 and C.17). Manufacturing method, material and other properties of each component are specified on the drawing.

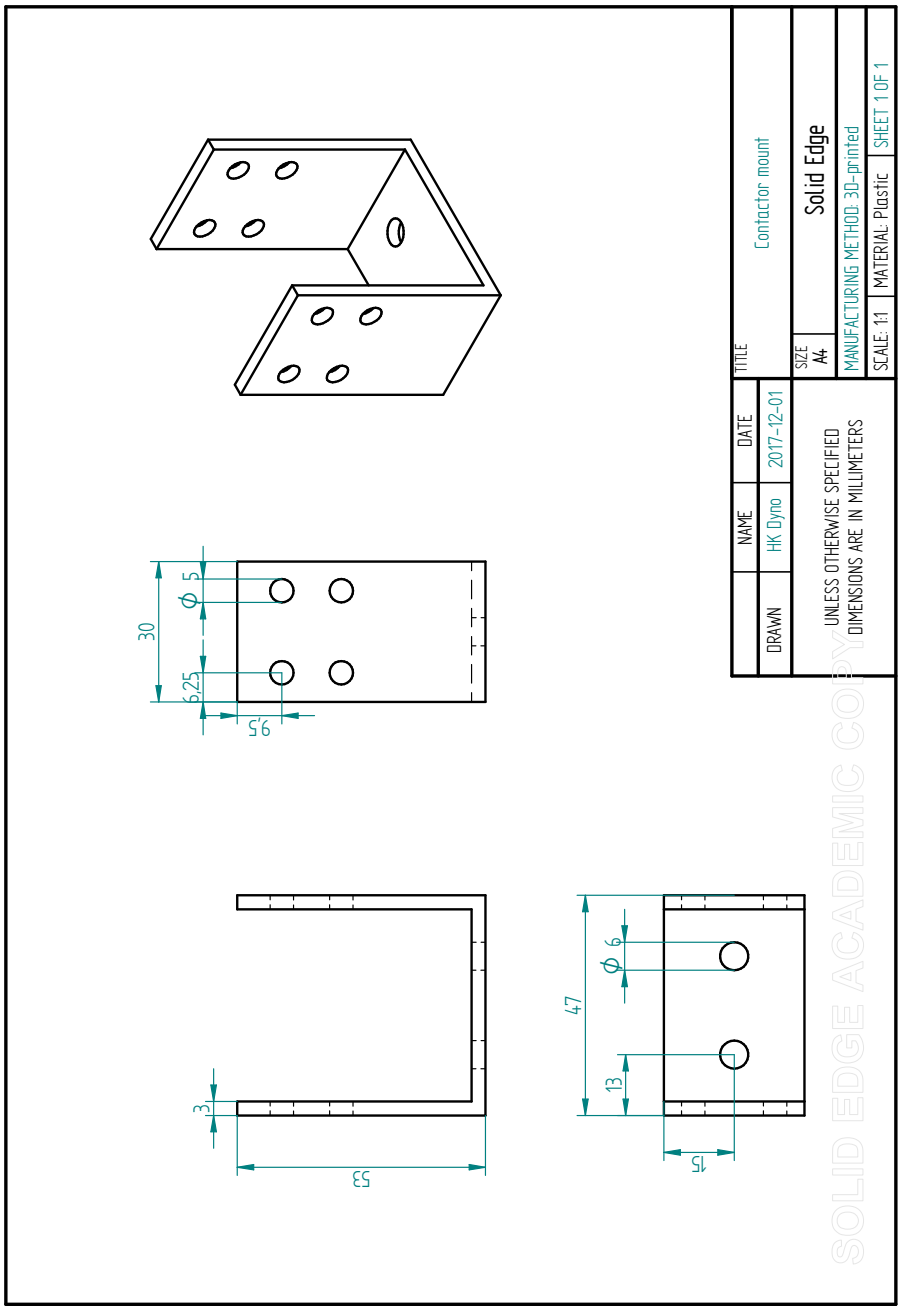

Figure C.1: Kelly contactor mount.

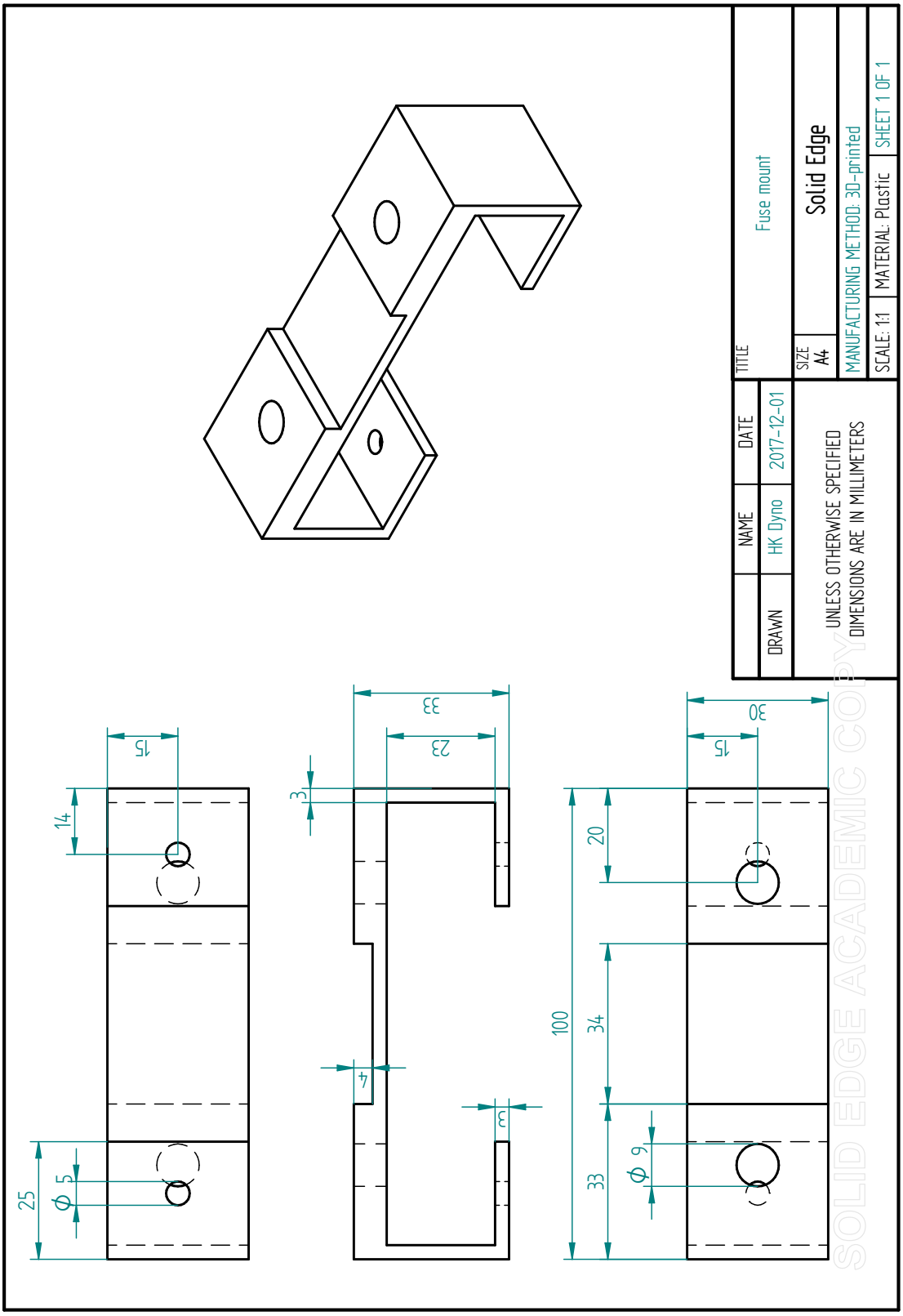

Figure C.2: Fuse mount.

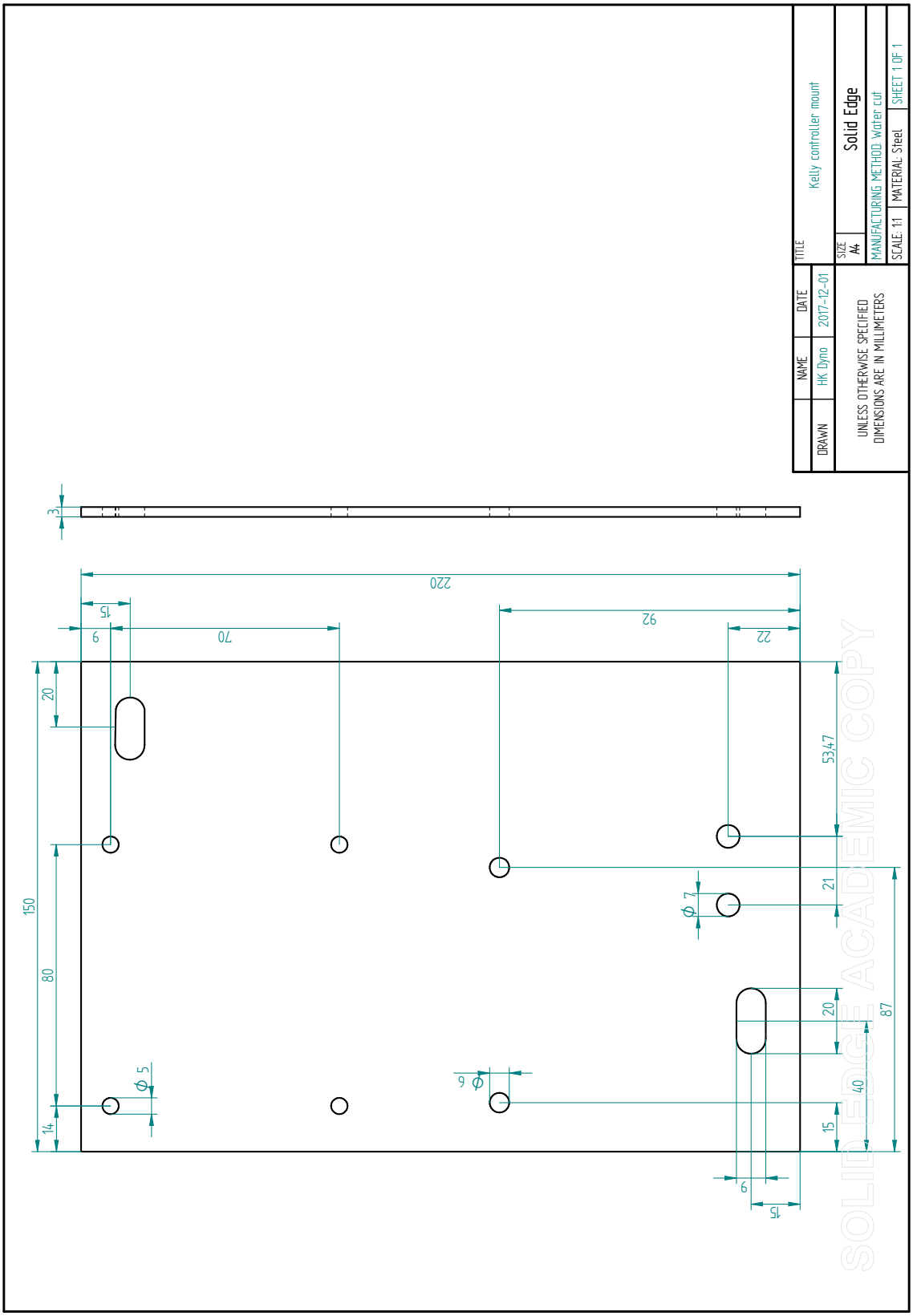

Figure C.3: Mount for Kelly controller, fuse and contactor.

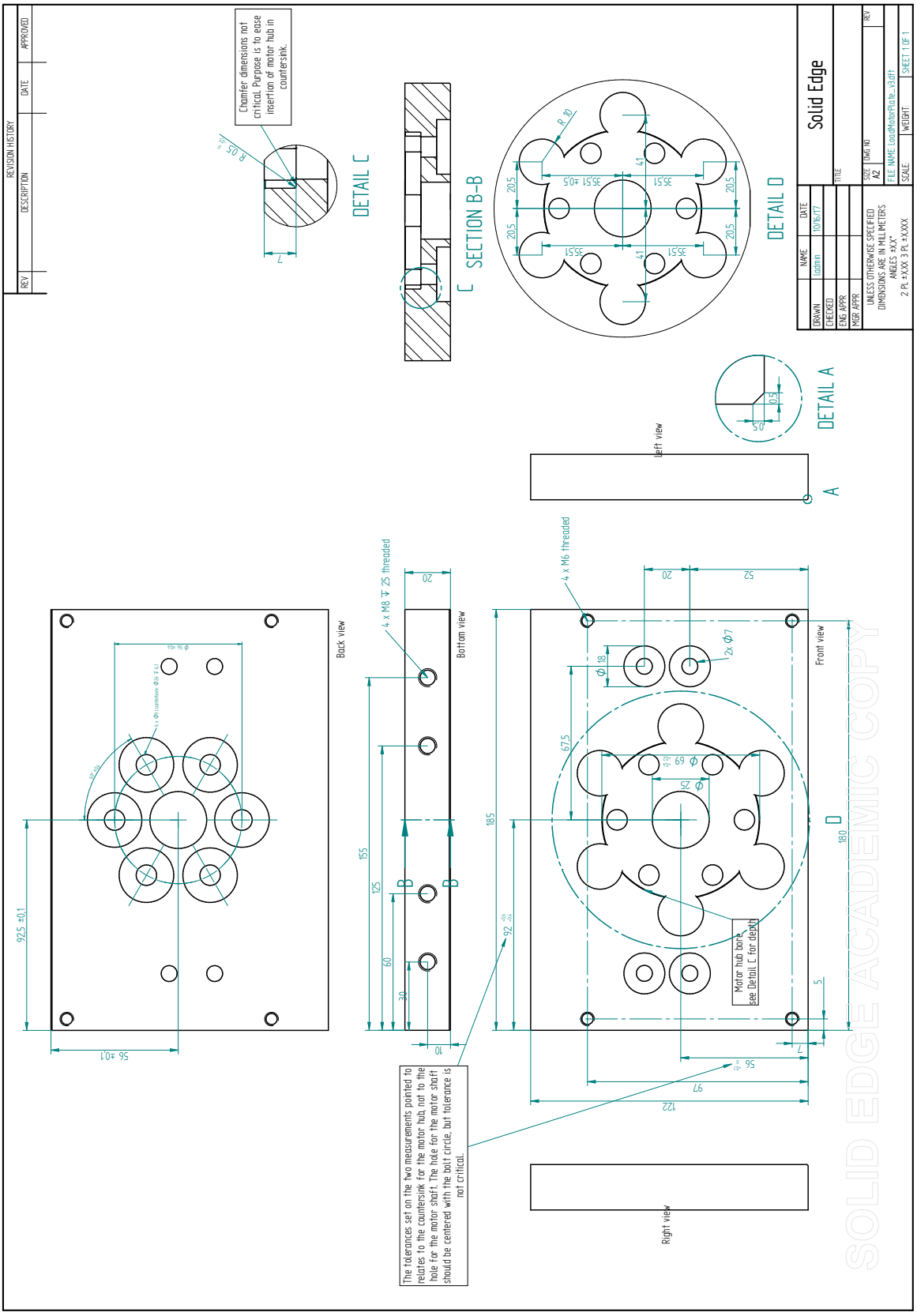

Figure C.4: Drawing of load module vertical plate.

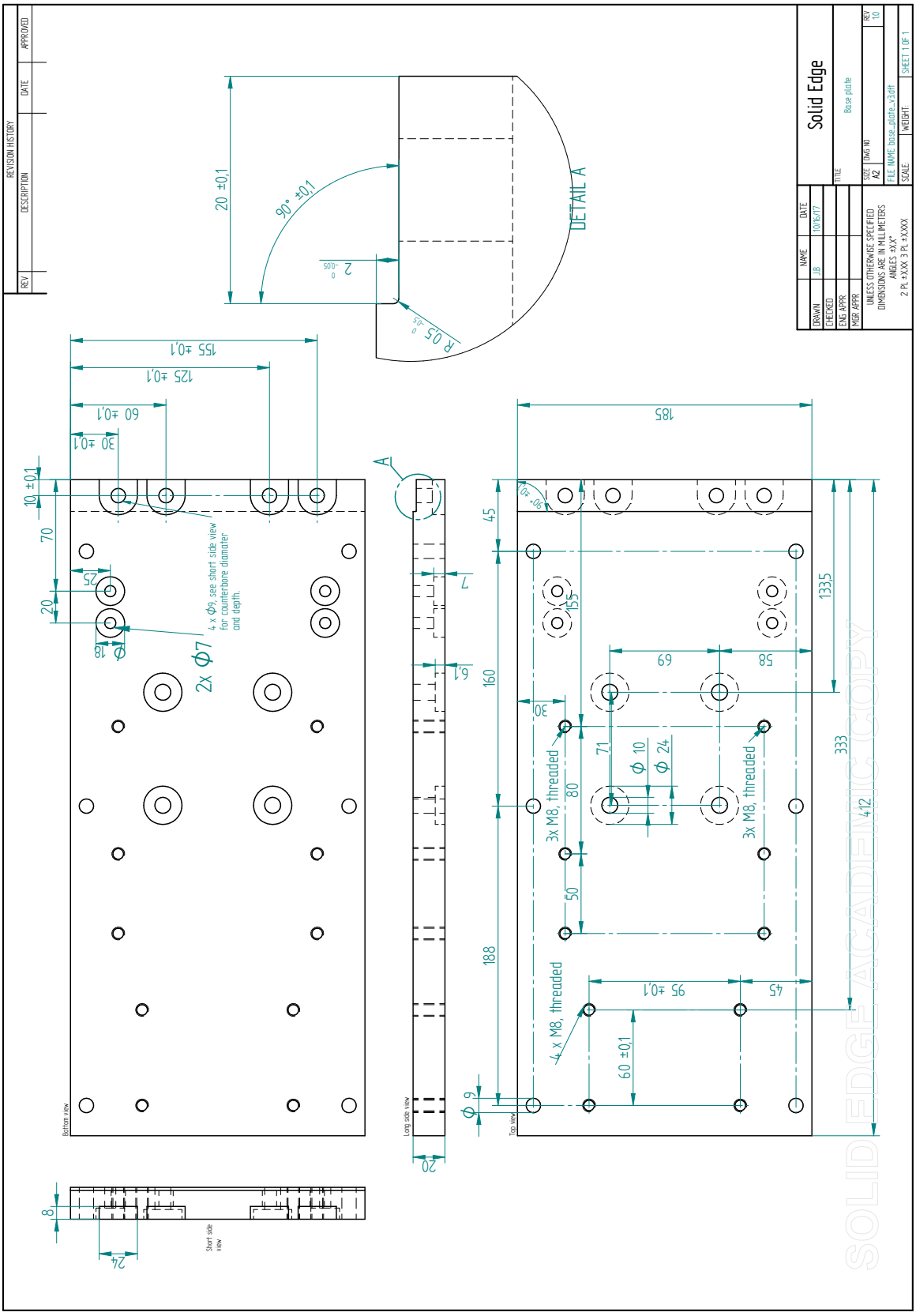

Figure C.5: Drawing of load module base plate.

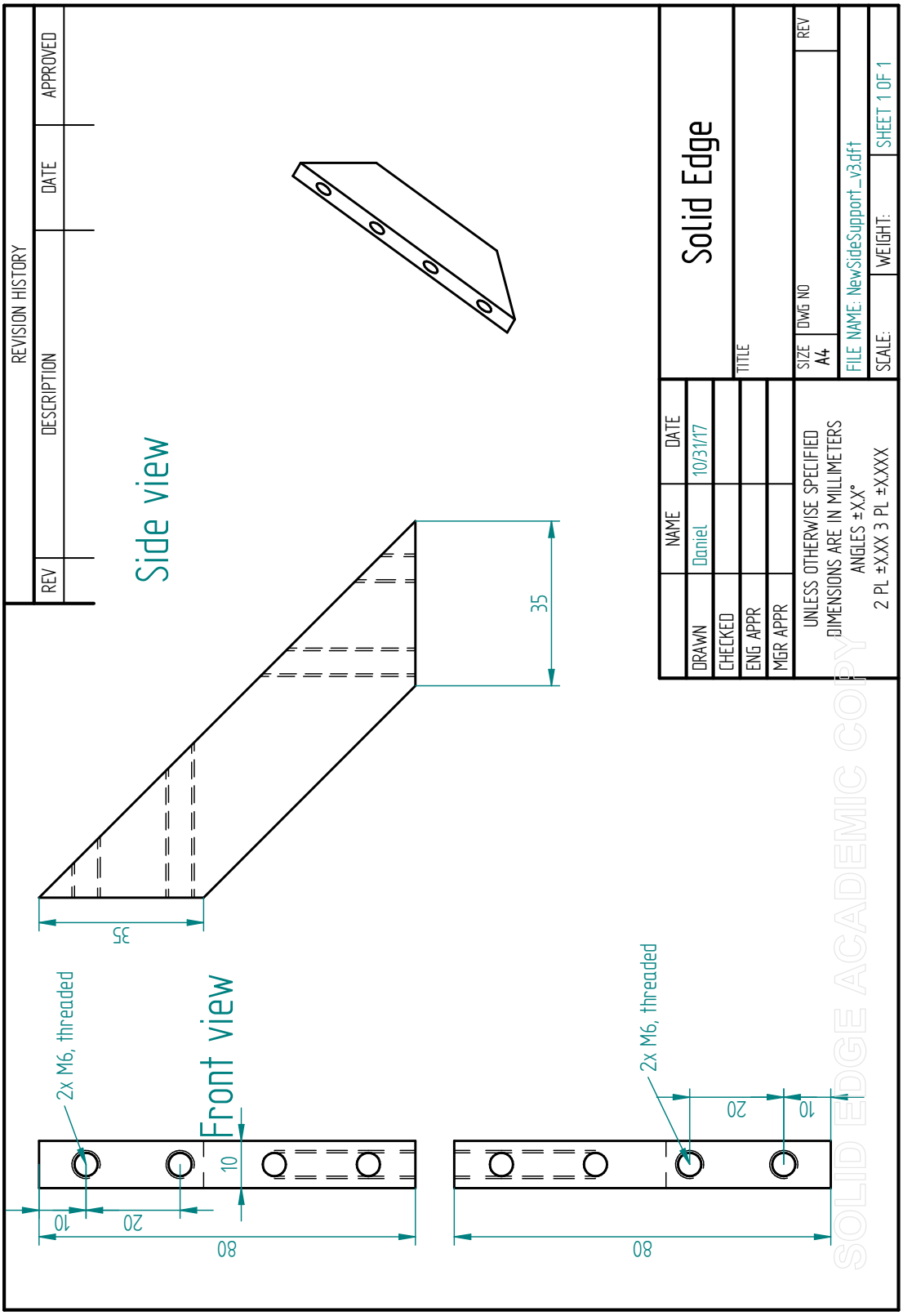

Figure C.6: Drawing of the supports for the vertical plate of the load module.

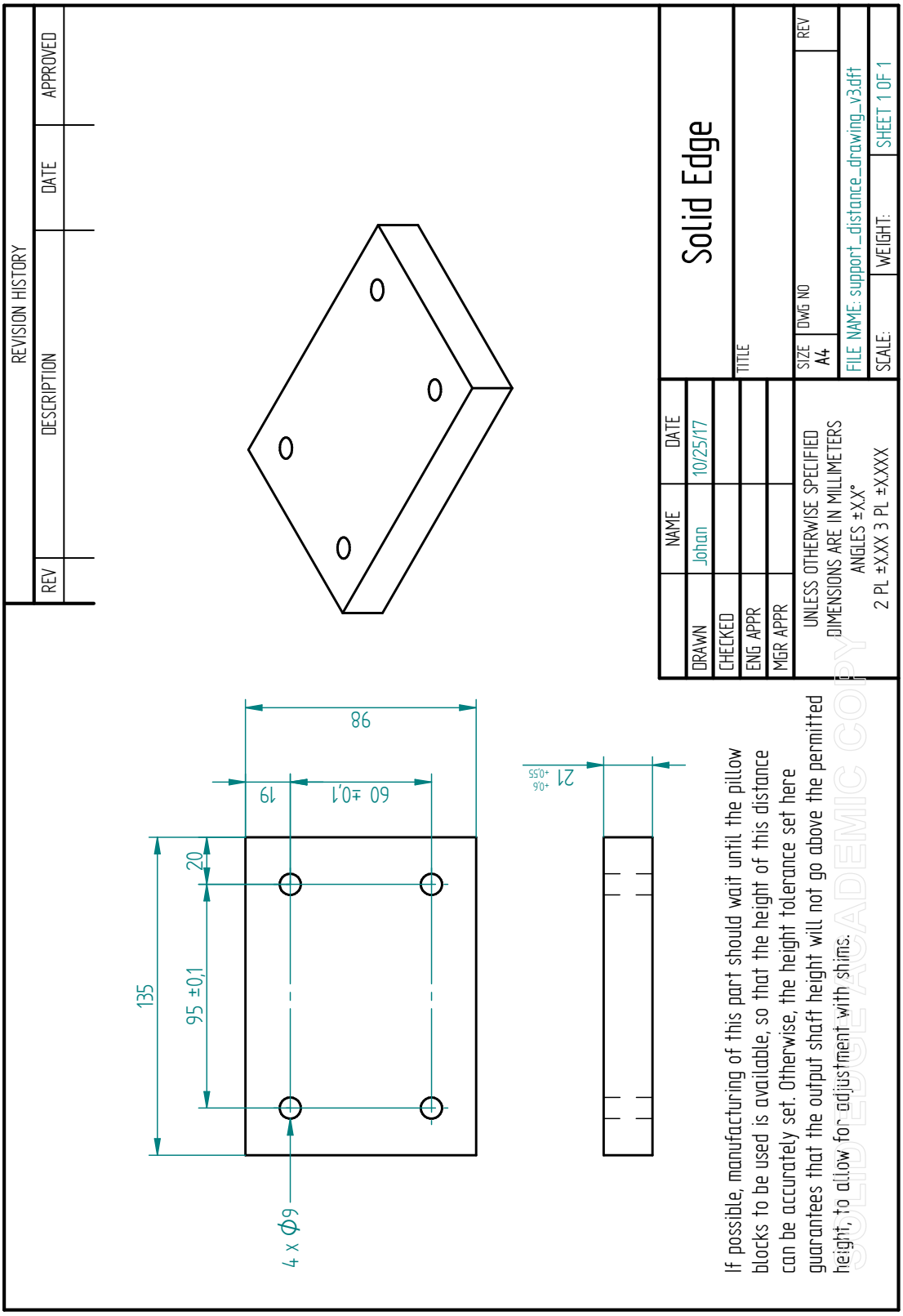

Figure C.7: Drawing of the distance for the pillow blocks in the load module.

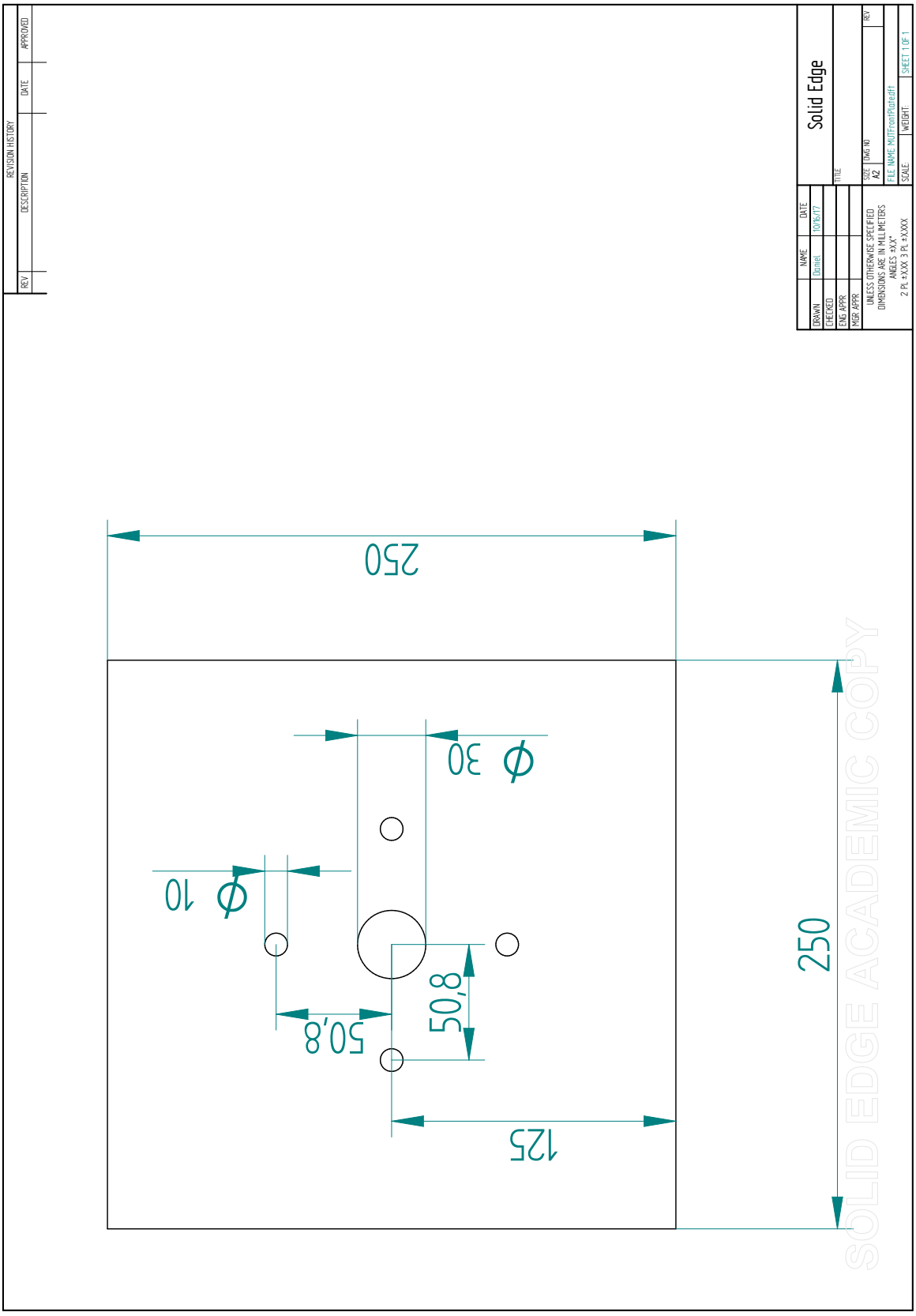

Figure C.8: Drawing of the MUT mount front plate, where the MUT is fastened.

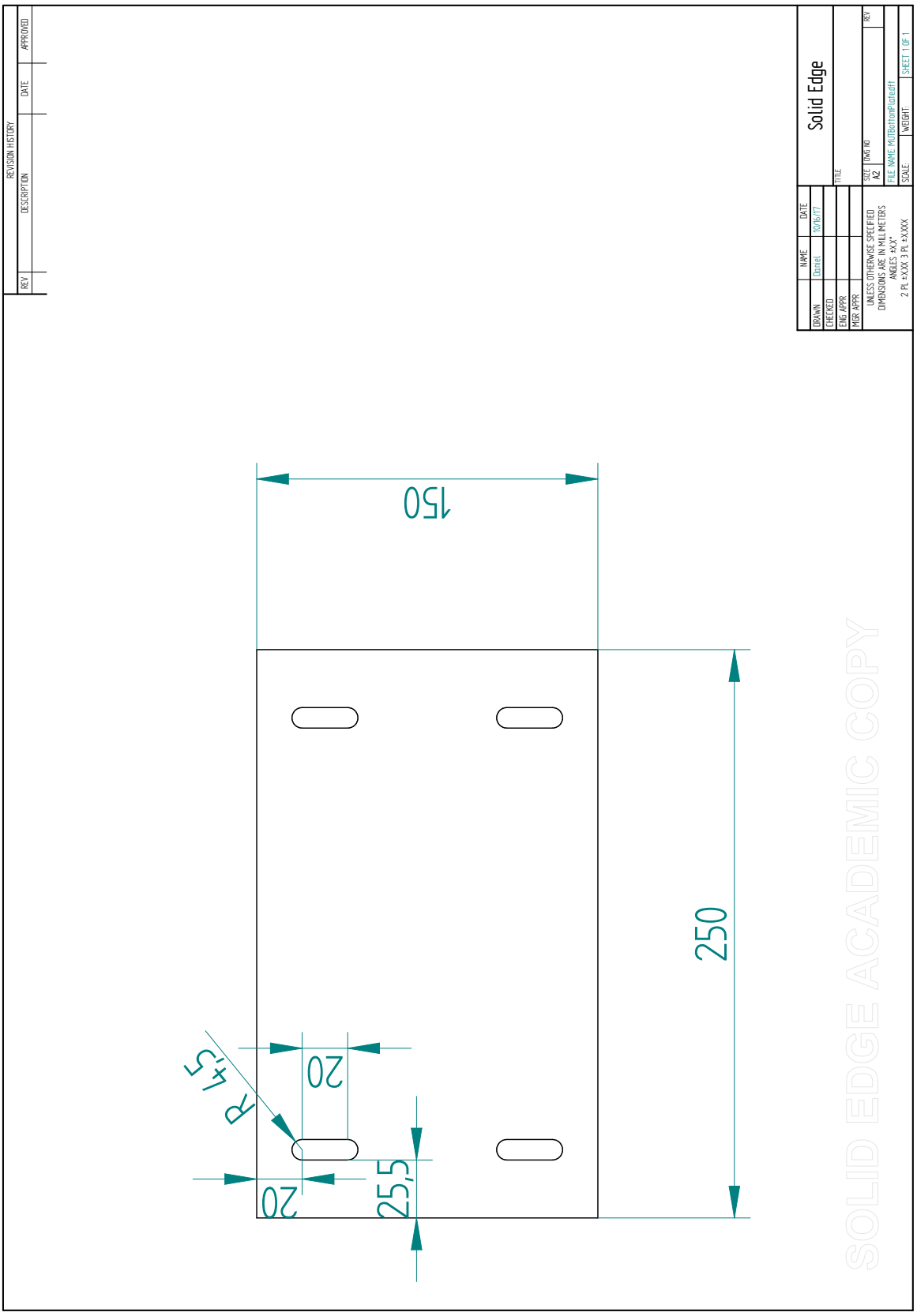

Figure C.9: Drawing of the bottom plate of the MUT mount where the mount is fastened to the rails.

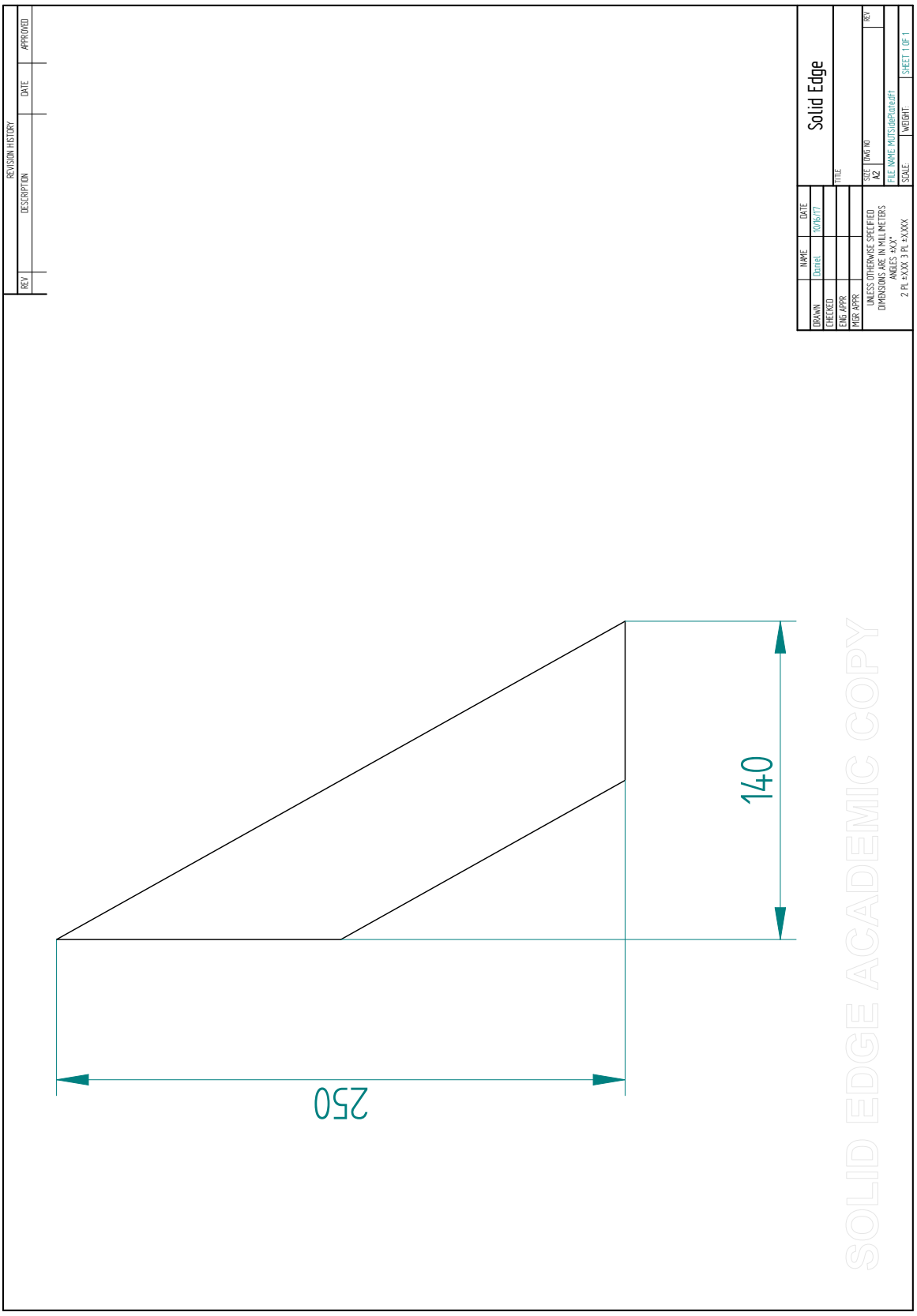

Figure C.10: Drawing of the MUT mount side plates that function as a strengthening support.

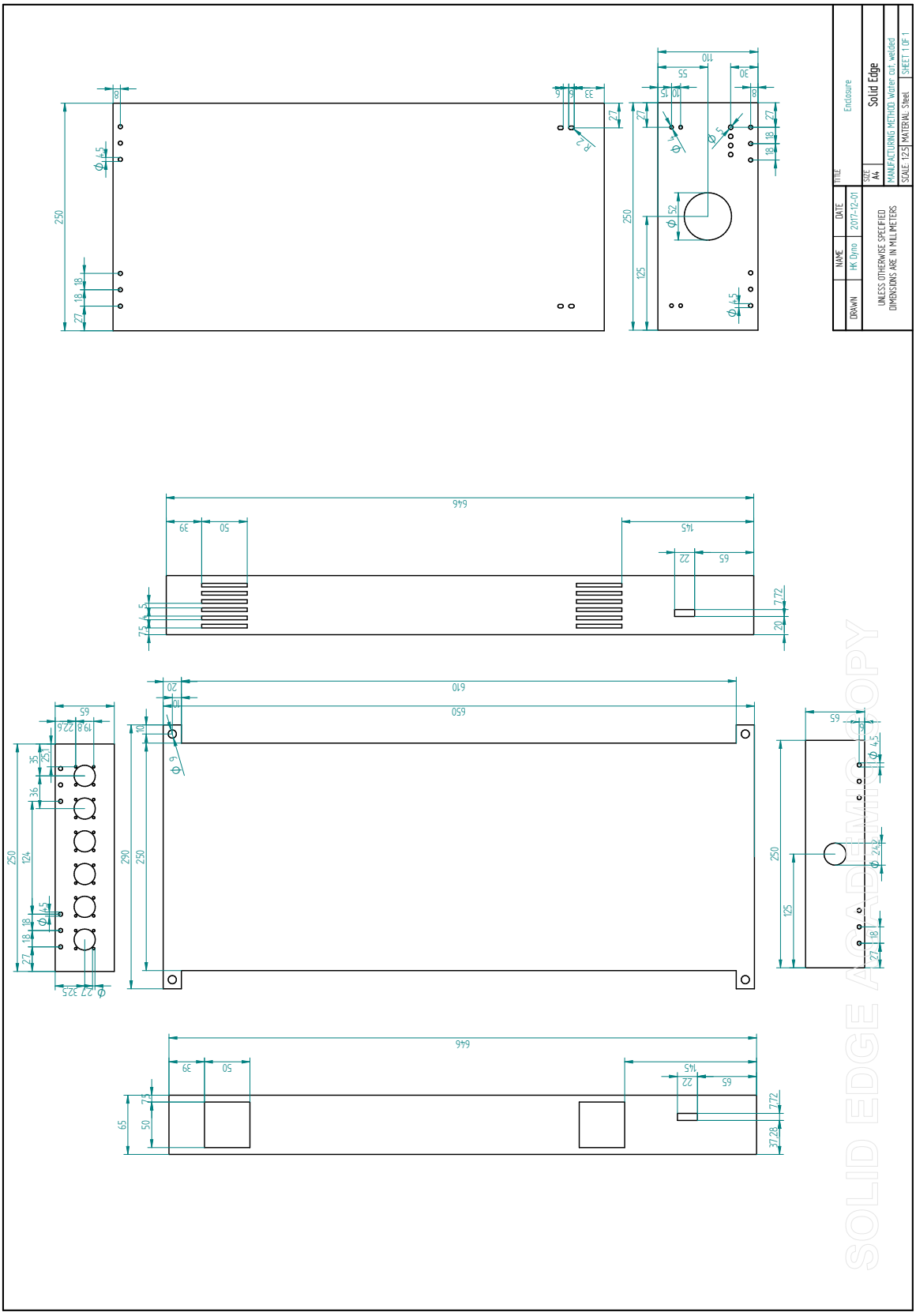

Figure C.11: Drawing of the enclosure.

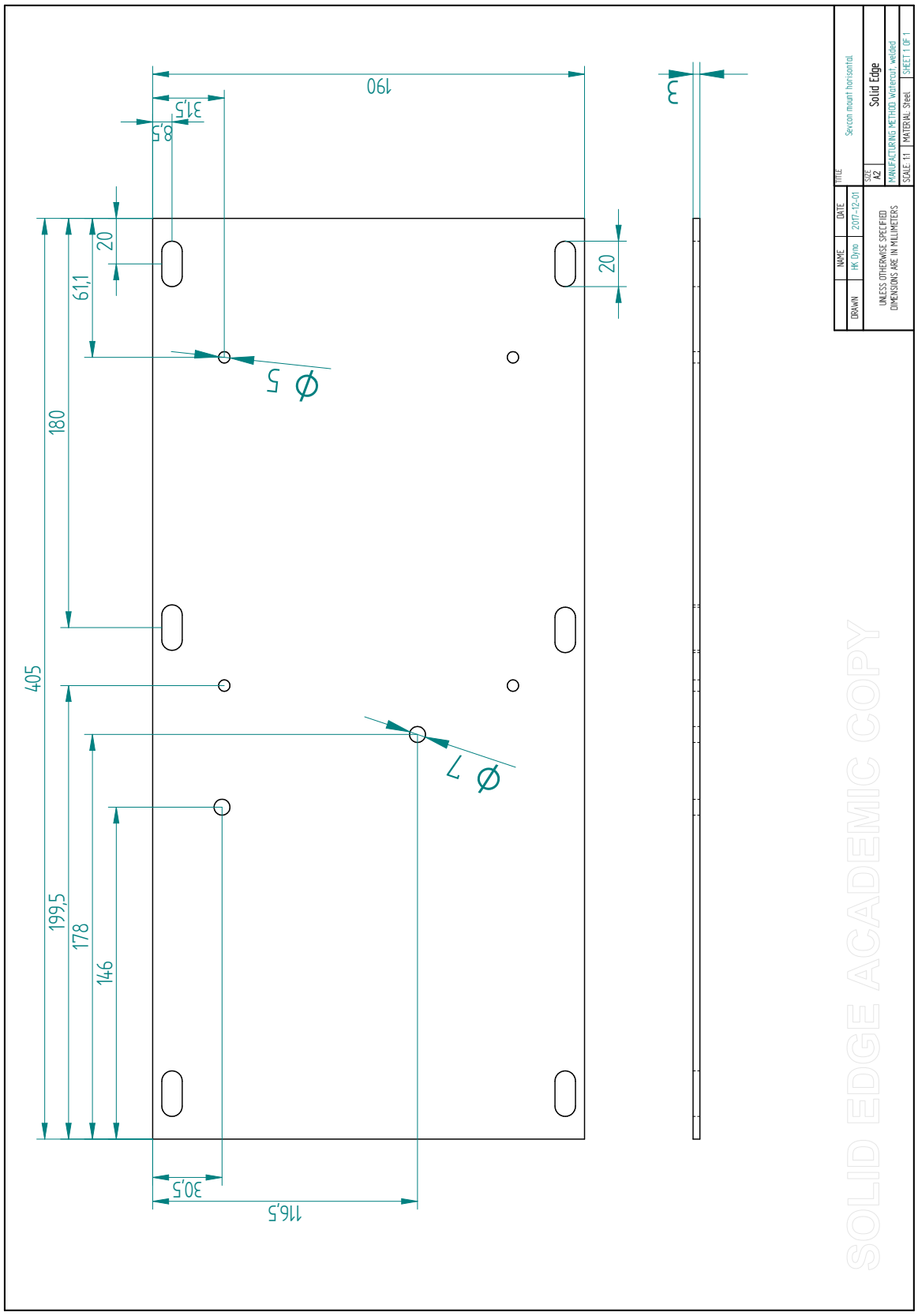

Figure C.12: Drawing of the horisontal part of the Sevcon controller mount.

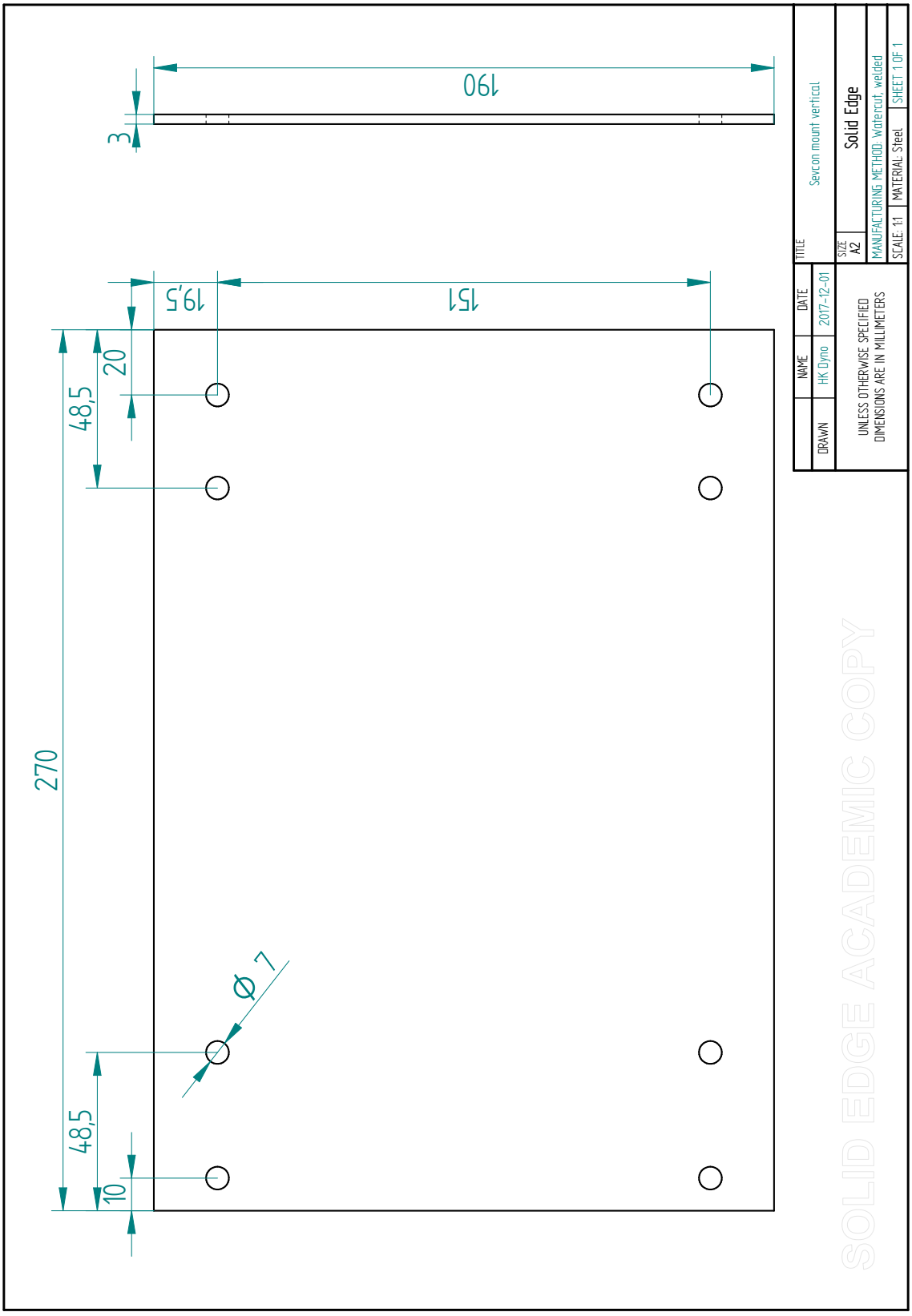

Figure C.13: Drawing of the vertical part of the Sevcon controller mount.

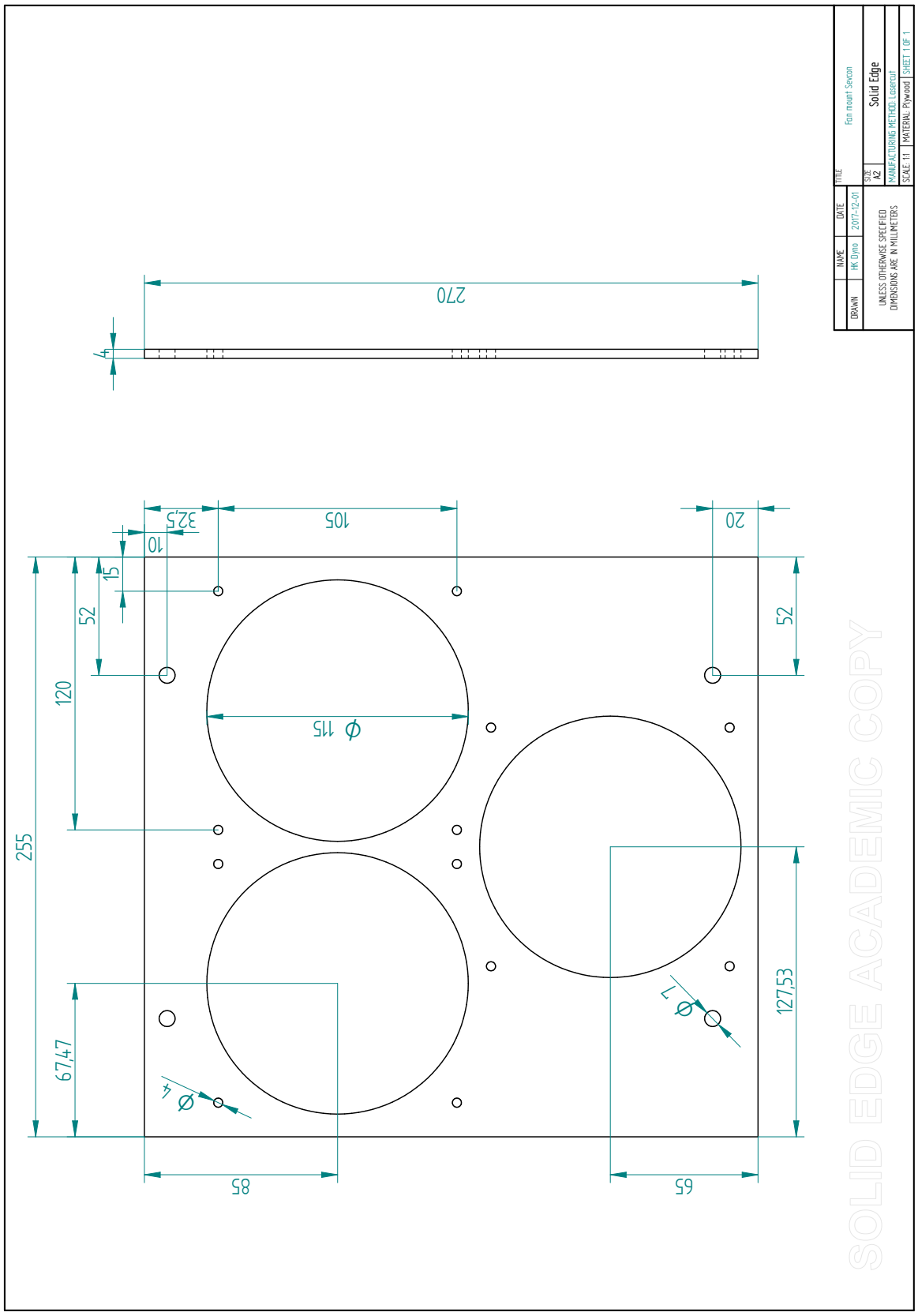

Figure C.14: Drawing of the fan mount to the Sevcon controller.

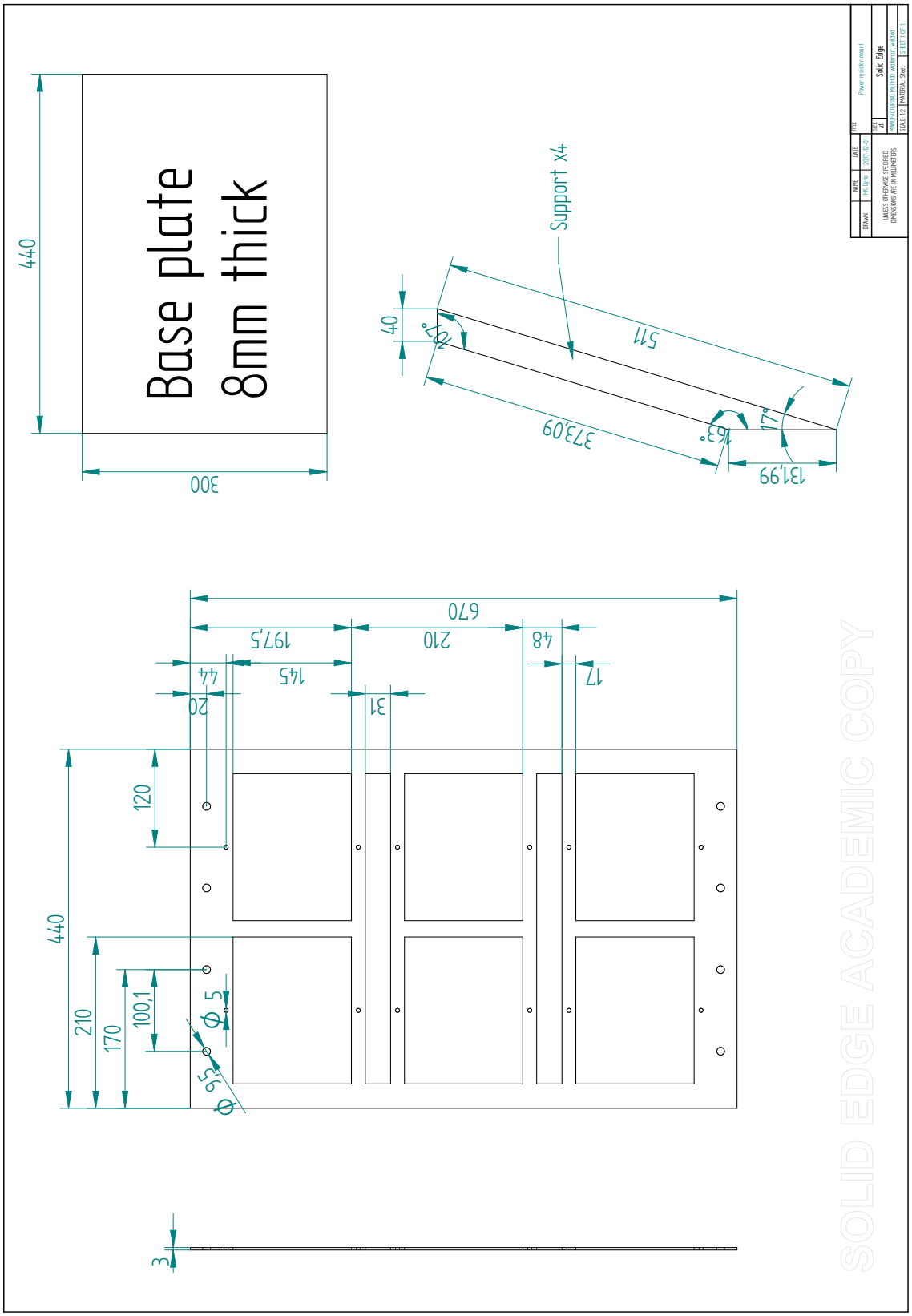

Figure C.15: Drawing of the power resistor and fans mount.

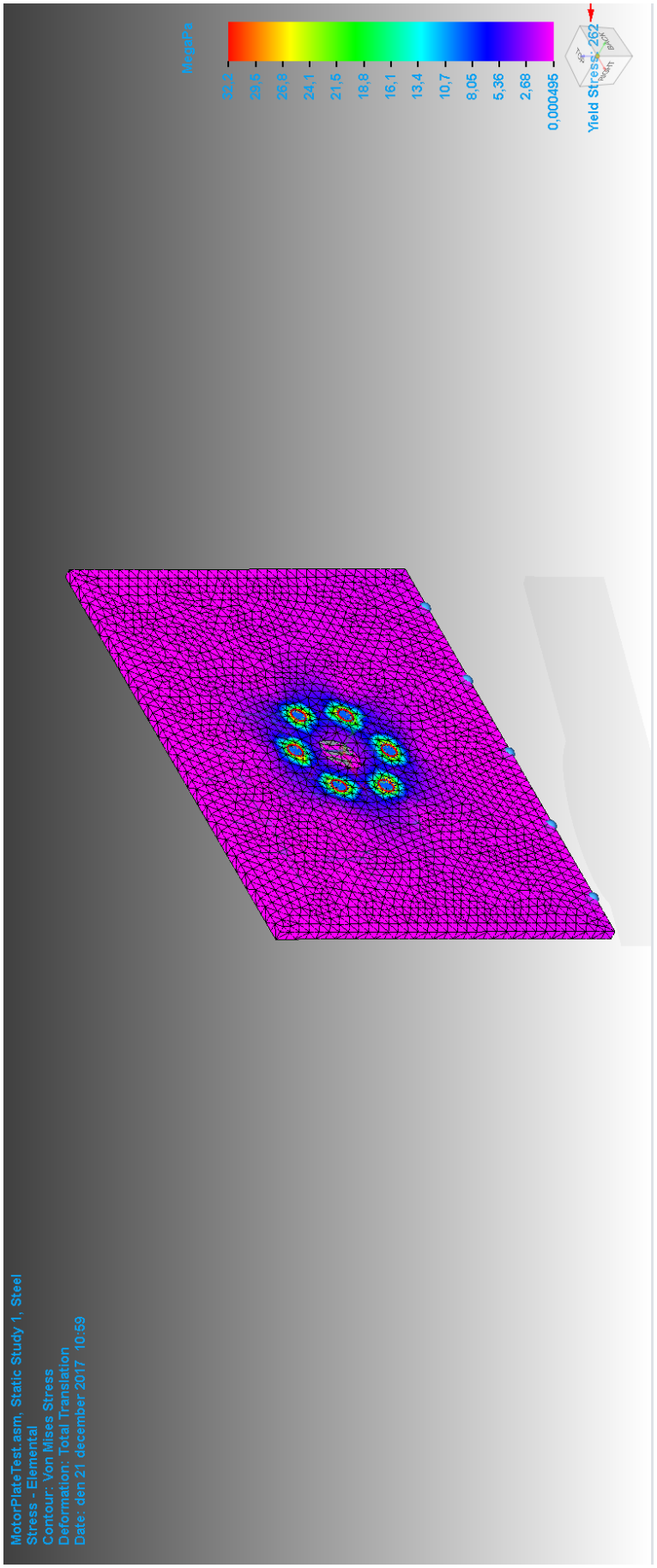

Figure C.16: Static study simulation with 40 Nm load on the bolt circle for the load motor mount.

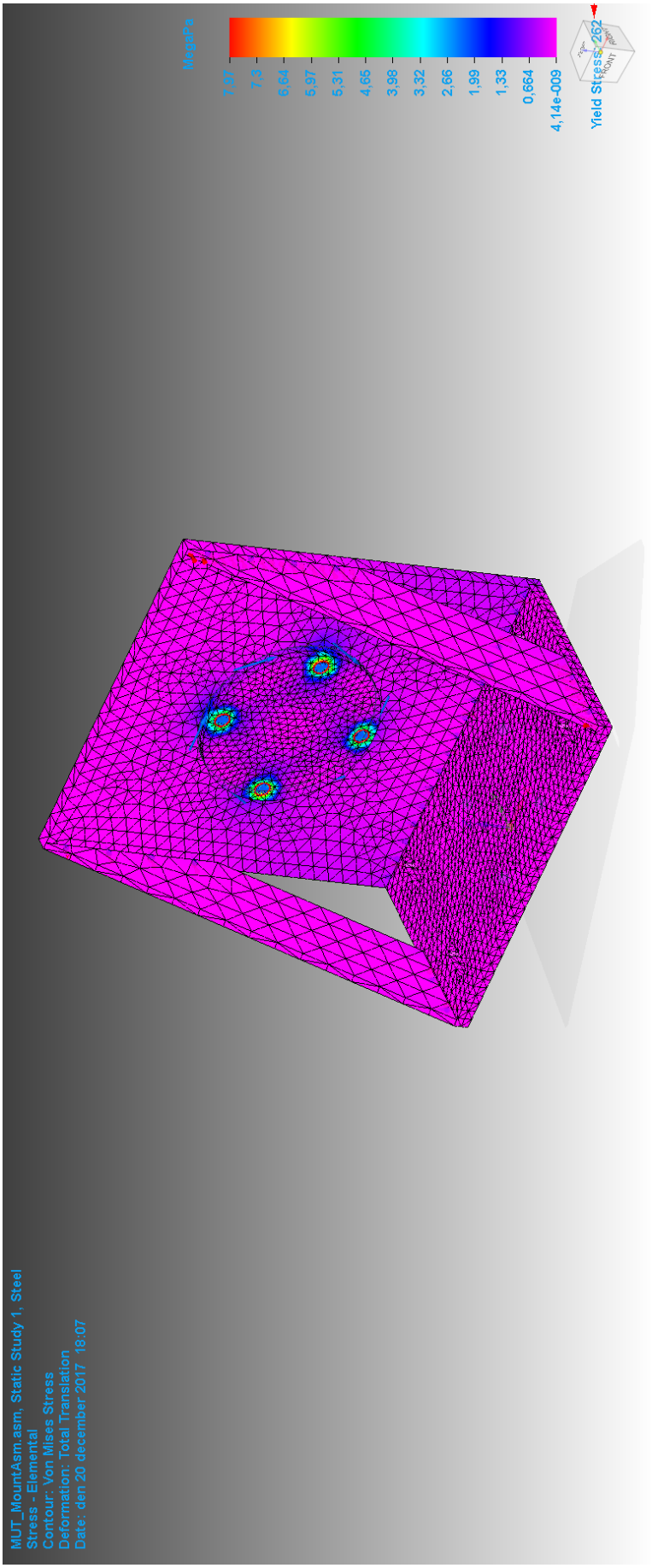

Figure C.17: Static study simulation with 40 Nm load on the bolt circle for the motor under test mount.

D - Connectors and Wiring

This appendix shows a table of the connectors and wires to the enclosure box for future work with the wiring. Figure D.1 shows the names of all connectors to complement Table D.1.

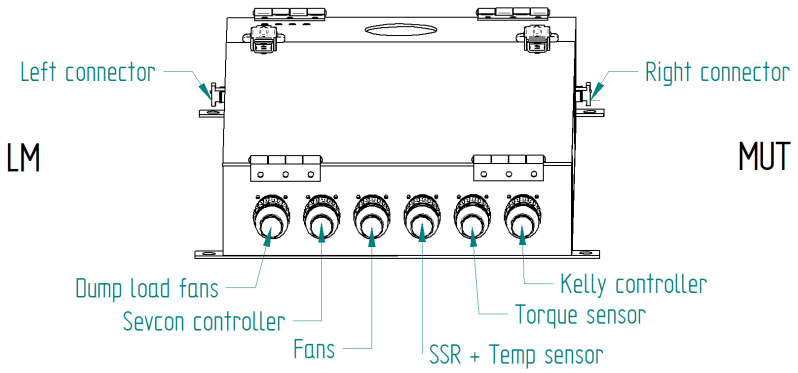

Figure D.1: Specified connector names.

	Connector pin	Wire color	Connection	Origin
Left connector:	1	Red	12V	LM batteries
	2	Red	84V	LM batteries
Right connector:	1	Red	24V	MUT batteries
	2	Blue	Common GND	All batteries

	Connector pin	Wire color	Connection	Datasheet notation
Kelly controller	1	Blue	LED GND	Black, pin 1
	2	Red	LED +	Brown, pin 2
	3	Blue	Trottle GND	Black, pin 8
	4	Red	Power (24V)	Red, pin 9
	5	Blue	Throttle switch GND	Black, pin 10
	6	Red	Throttle switch +	White, pin 5
	7	Purple	Throttle output	Blue, pin 6
	8	Red	Throttle +	Orange, pin 7
	2	Blue	-15V	Blue wire
Torque sensor	3	Purple	Velocity	White wire
	4	Green	GND	Green wire
	5	Red	+15V	Red wire
	6	Green	Torque	Yellow wire

	Connector pin	Wire color	Connection	Fem connector pin
SSR +	1	Black	Arduino SDA	Red
Temp sensor	2	Green	Arduino Digital Pin	Green
	3	Purple	Arduino GND	Brown
	4	Red	Temp sensor GND	White
	5	Blue	Temp +	Yellow
	6	White	Arduino SCL	Black

	Connector pin	Wire color	Connection	Fan wire color
Fans	1	Red	Fan 1 +	Red
	2	Blue	Fan 1 GND	Black
	3	Red	Fan 2 +	Red
	4	Blue	Fan 2 GND	Black
	5	Red	Fan 3 +	Red
	6	Blue	Fan 3 GND	Black
	7	Red	Fan 4 +	Red
	8	Blue	Fan 4 GND	Black

	Connector pin	Wire color	Connection	Datasheet notation
Sevcon controller	1	Purple	CAN High	13
	2	Red	Throttle +	34
	3	Blue	Throttle vary	22
	4	Green	Temperature GND	Motor temp sensor
	5	Red	Footbrake vary	30
	6	Red	Forward switch	18
	7	Red	Foot switch, FS1	19
	8	Purple	7A Fuse	B+ and key switch
	9	Purple	CAN Low	24
	11	Red	Seat switch	31
	12	Red	Key switch	1

	Connector pin	Wire color	Connection	Fan wire color
Fans dump load	2	Red	Fan 1 +	Red
	10	Blue	Fan 1 GND	Black
	9	Red	Fan 2 +	Red
	1	Blue	Fan 2 GND	Black
	8	Red	Fan 3 +	Red
	4	Blue	Fan 3 GND	Black
	12	Red	Fan 4 +	Red
	3	Blue	Fan 4 GND	Black
	6	Red	Fan 5 +	Red
	7	Blue	Fan 5 GND	Black
	11	Red	Fan 6 +	Red
	5	Blue	Fan 6 GND	Black

Table D.1: All connectors, wires and pins to the enclosure.

E - Motor Model

In this section screenshots from the Simulink model are presented. A detailed explanation of the model can be found in chapter 5.1 - Modeling.

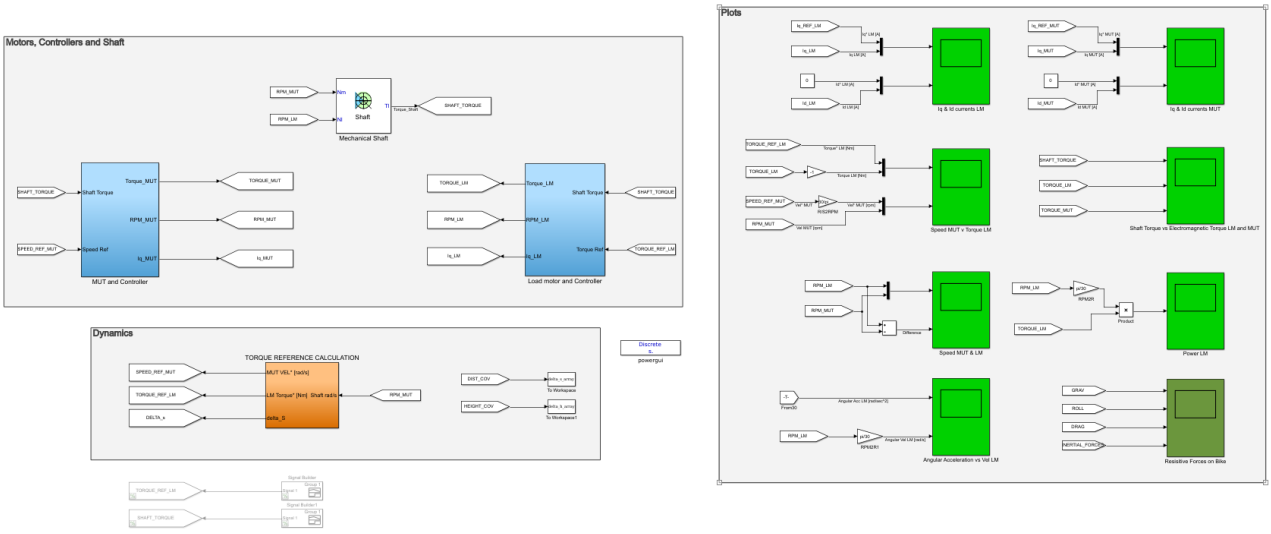

Figure E.1: Overview of the test rig in Simulink

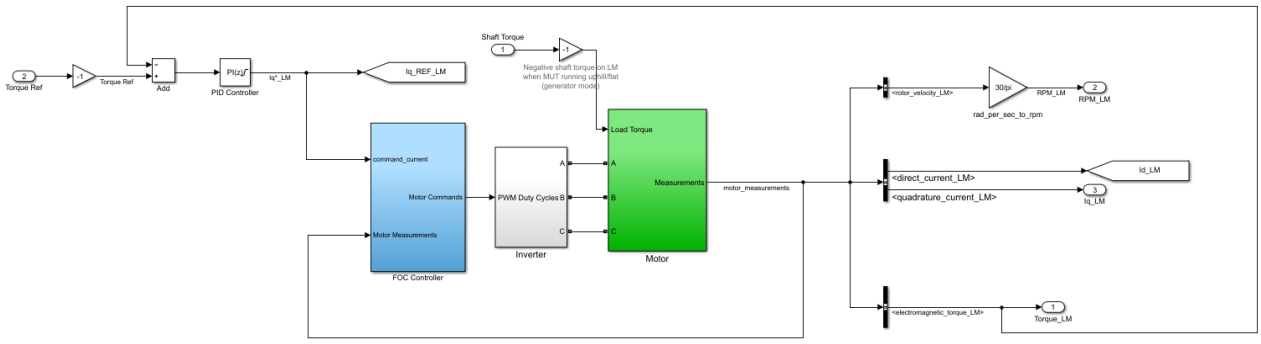

Figure E.2: Torque loop of LM in Simulink

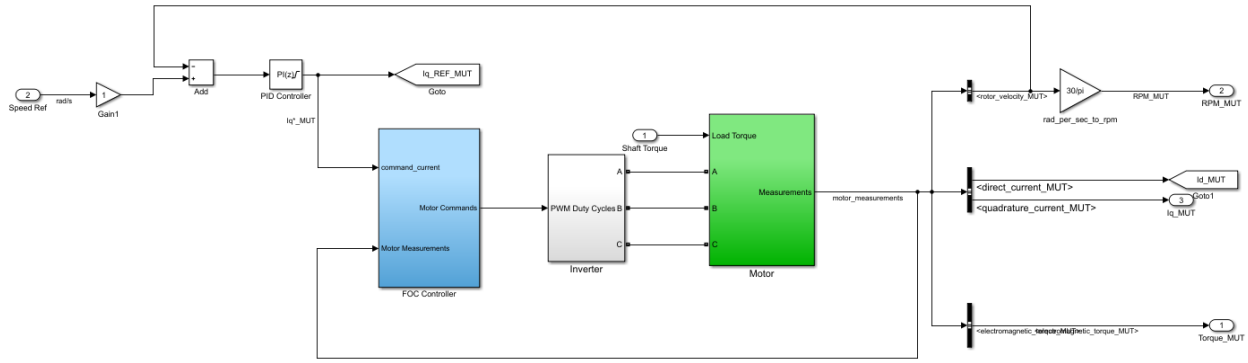

Figure E.3: Speed loop of MUT in Simulink

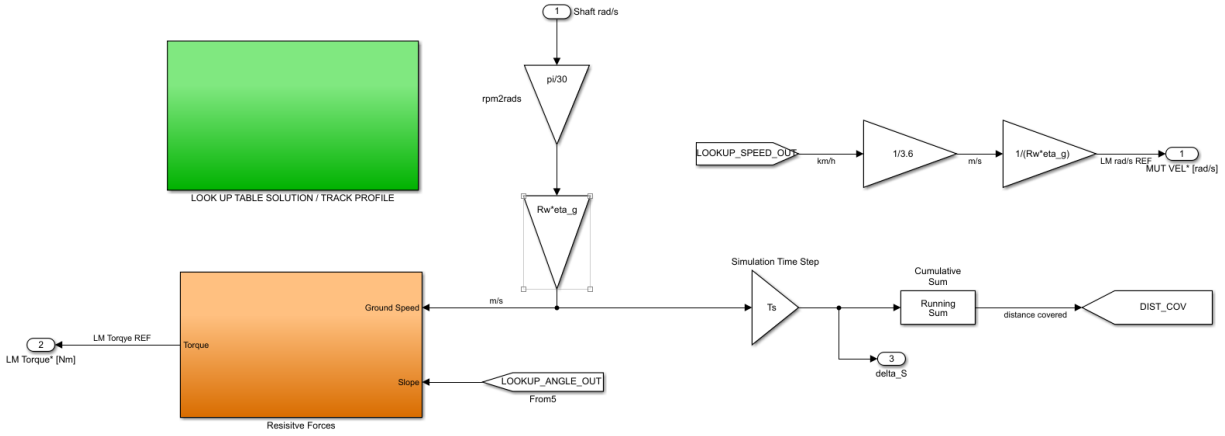

Figure E.4: Dynamics model in Simulink

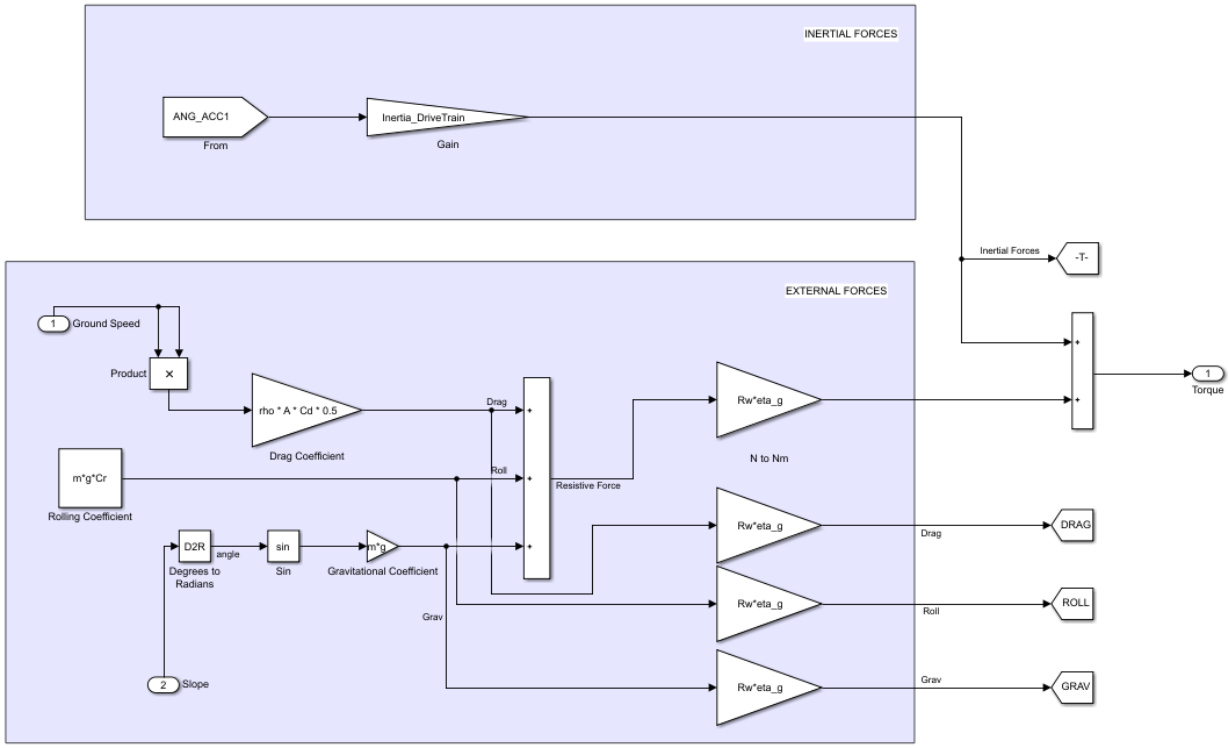

Figure E.5: Resisitve forces on the bike in Simulink

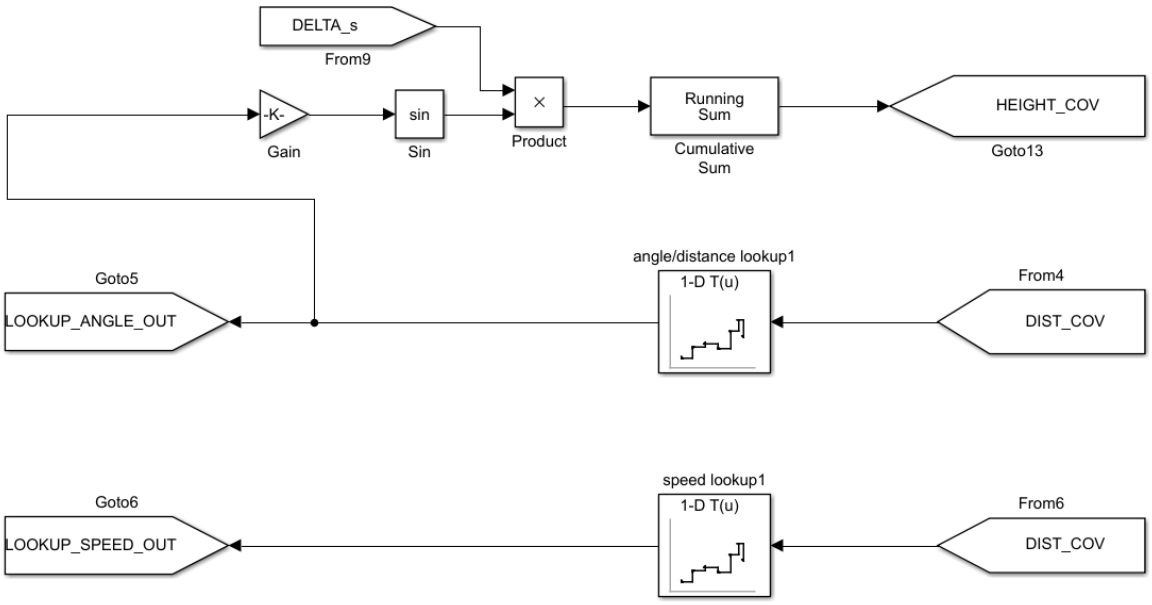

Figure E.6: Look up table to get the bikes position on the simulated track

F - DVT Software

The DVT software was installed on laptop to allow us to monitor and configure the parameters of any Sevcon CANopen node such as the Gen4 controller.This section presents complete setup and overview of DVT software. When we powered up the controller and started the proprietary software. The screen as shown in figure F.1 appeared.This shows us the selected baud rates.

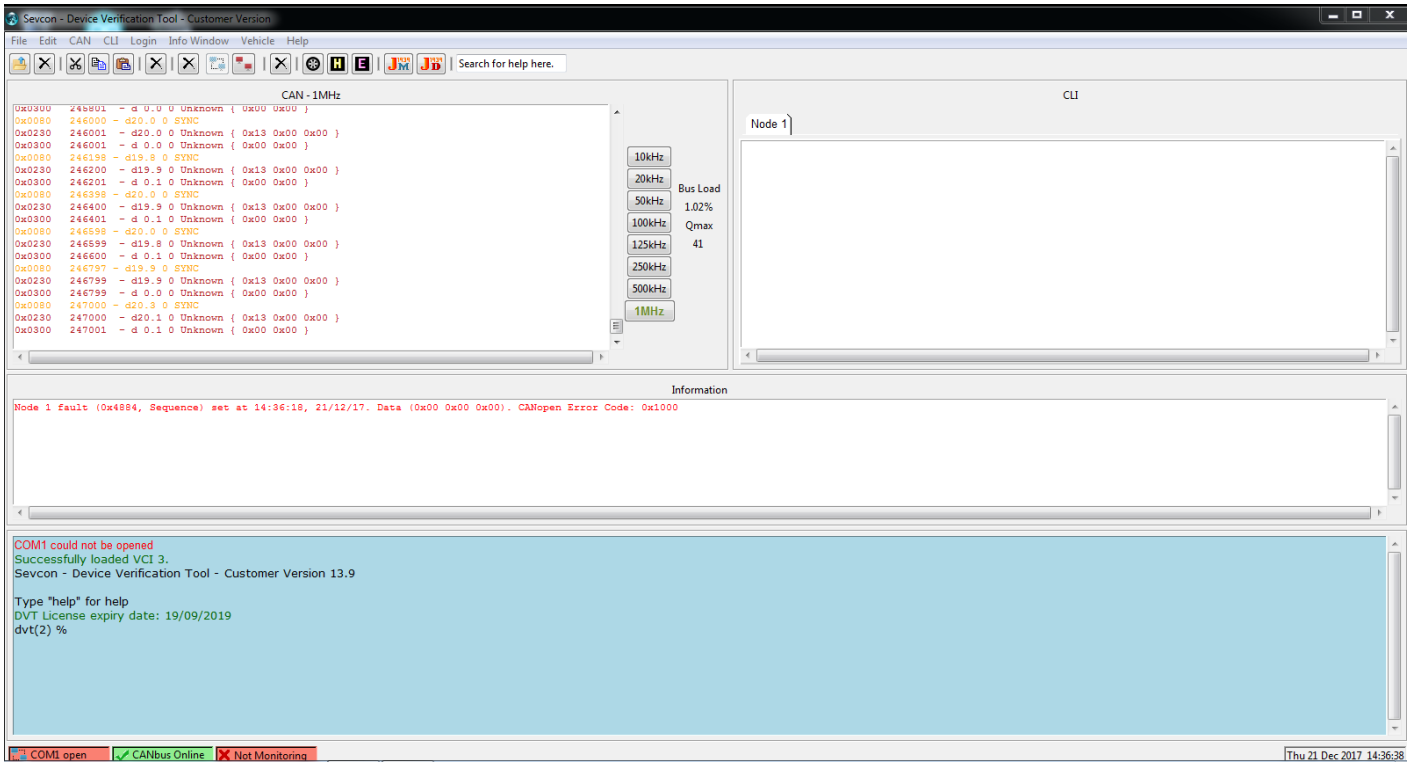

Figure F.1: Starting the DVT software

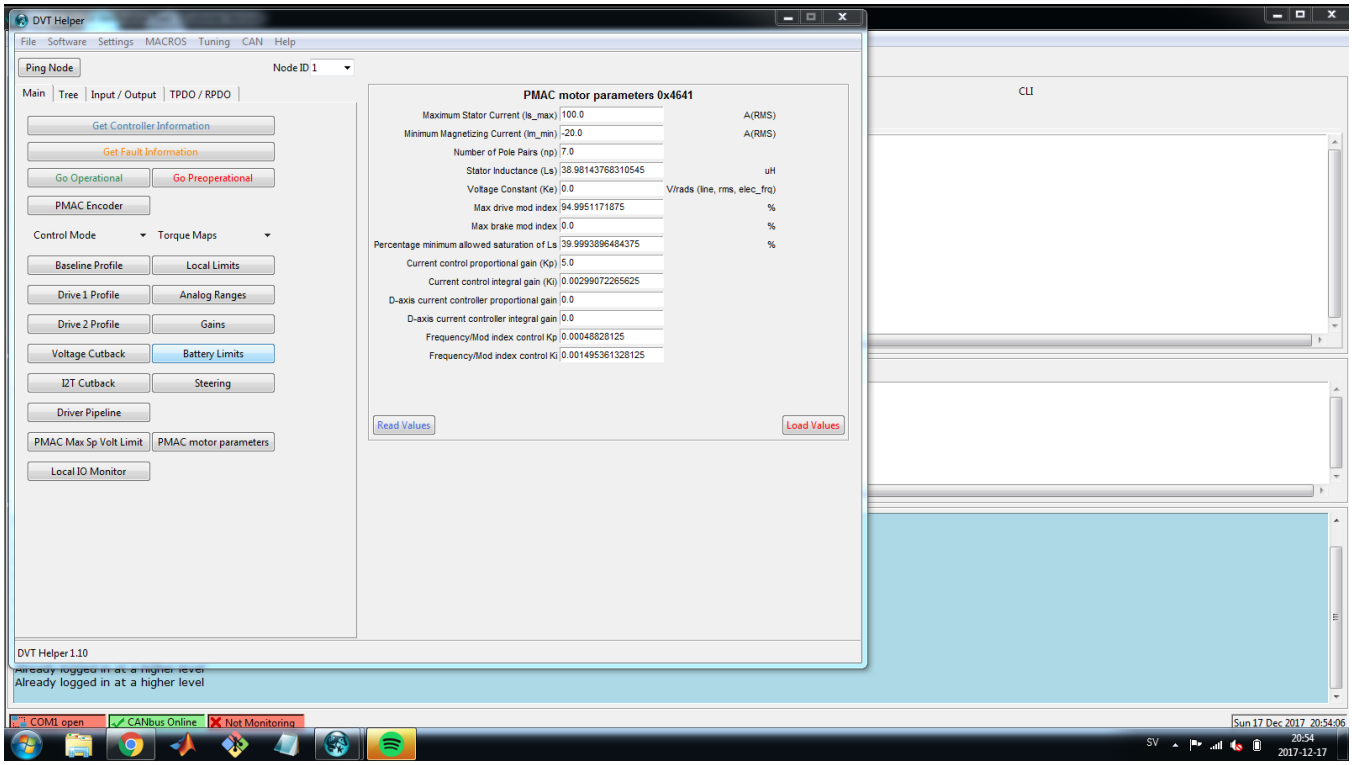

Figure F.2: PMAC motor parameters

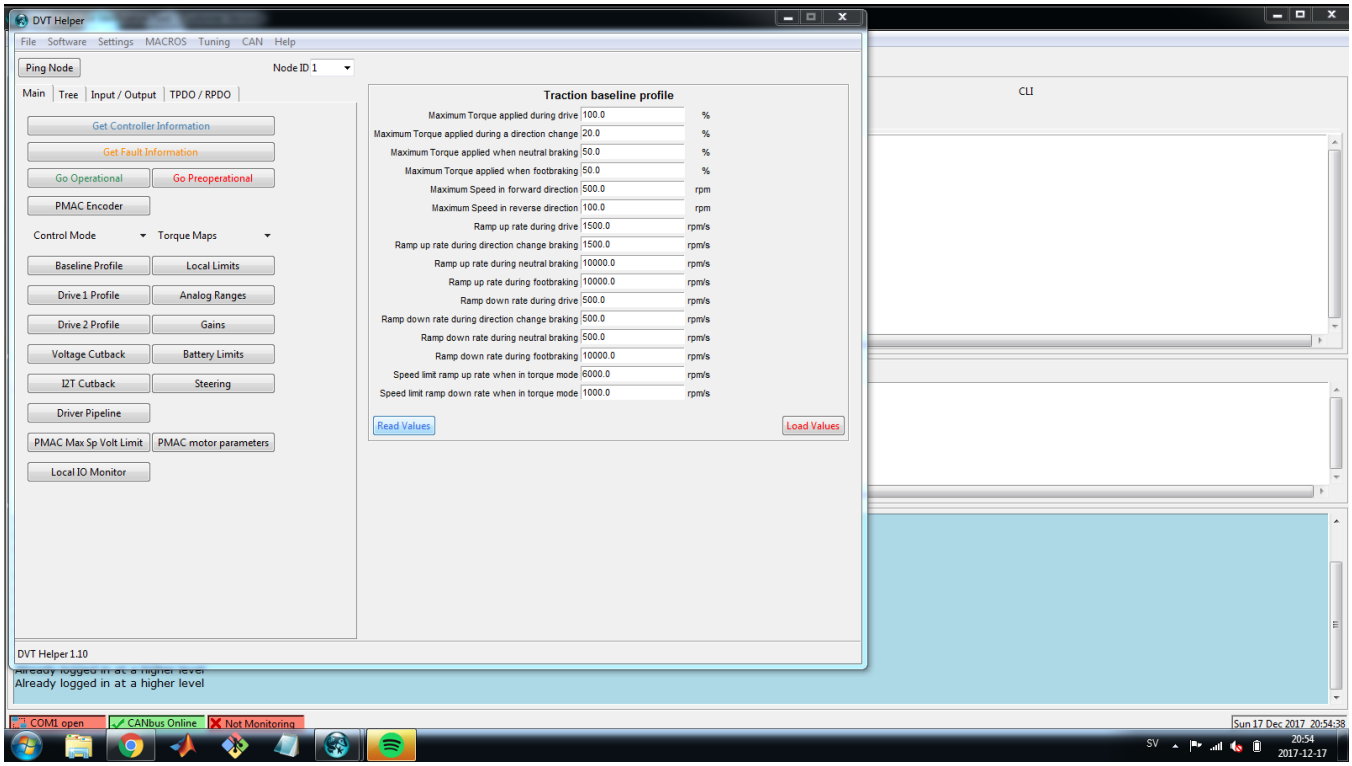

Figure F.3: Traction Baseline profile

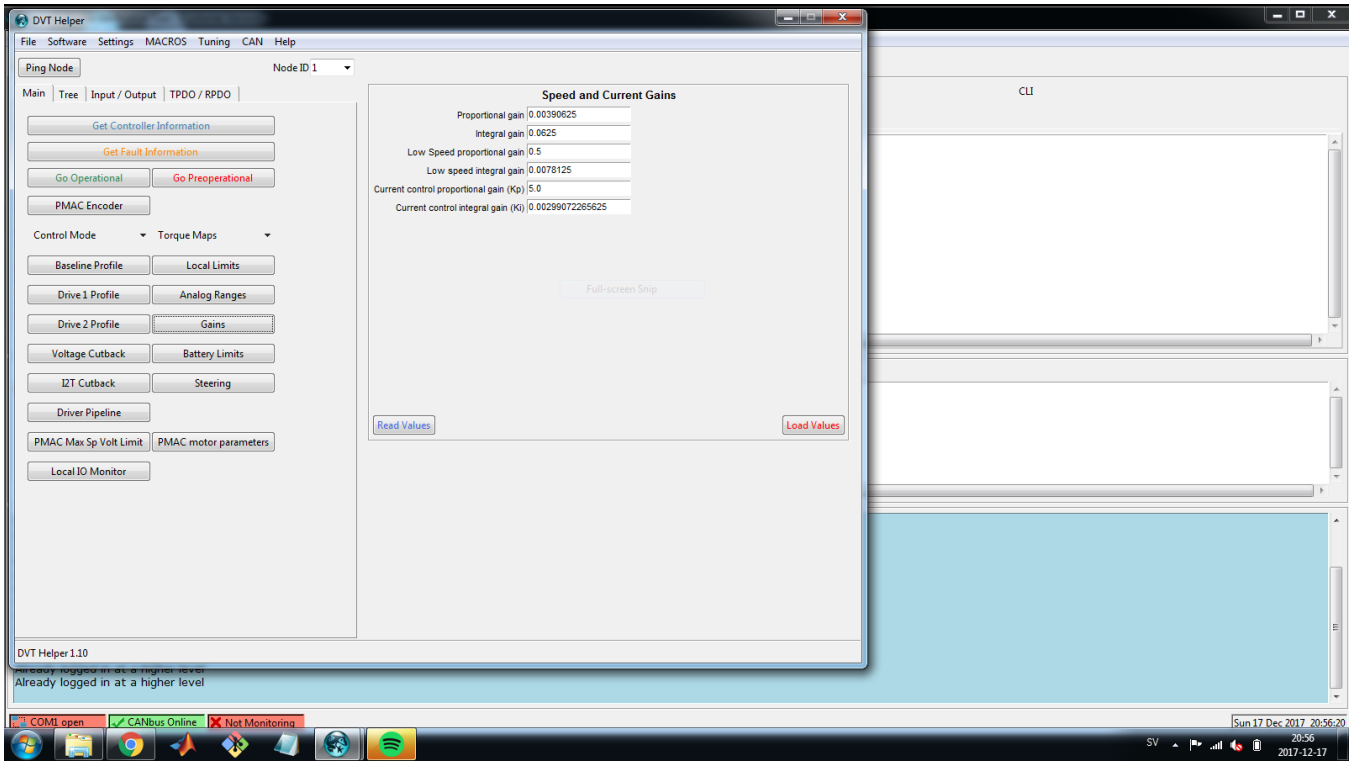

Figure F.4: Control parameters

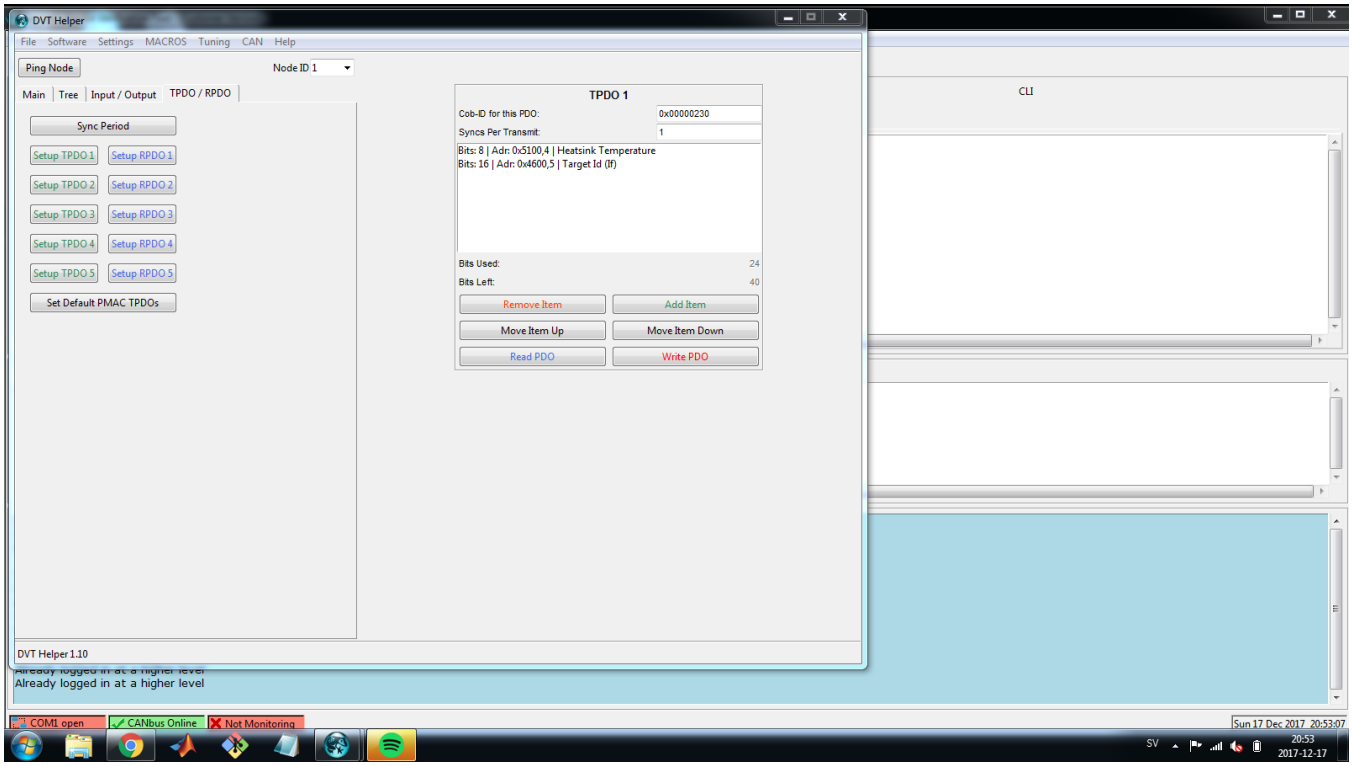

Figure F.5: PDO with id 0x300 sent out from Sevcon

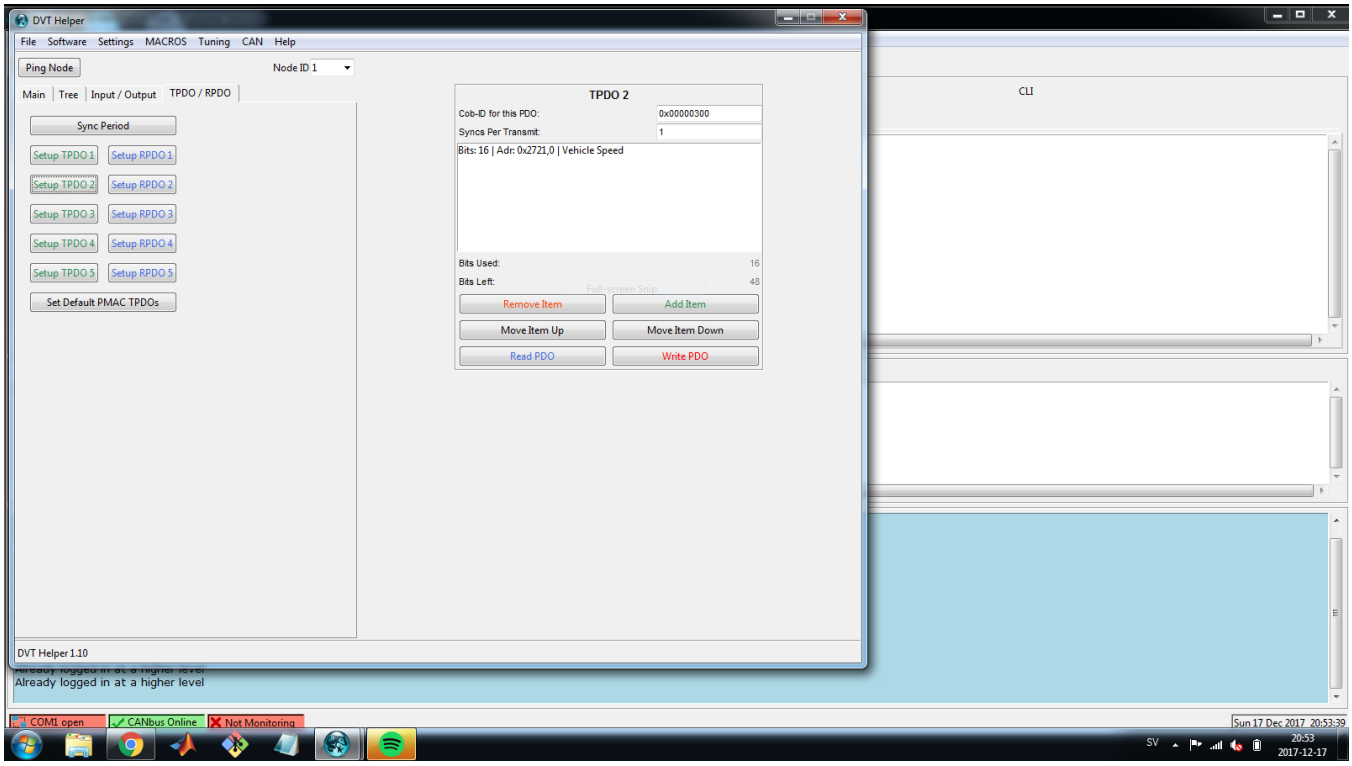

Figure F.6: PDO with id 0x301 sent from Sevcon

G - State flow

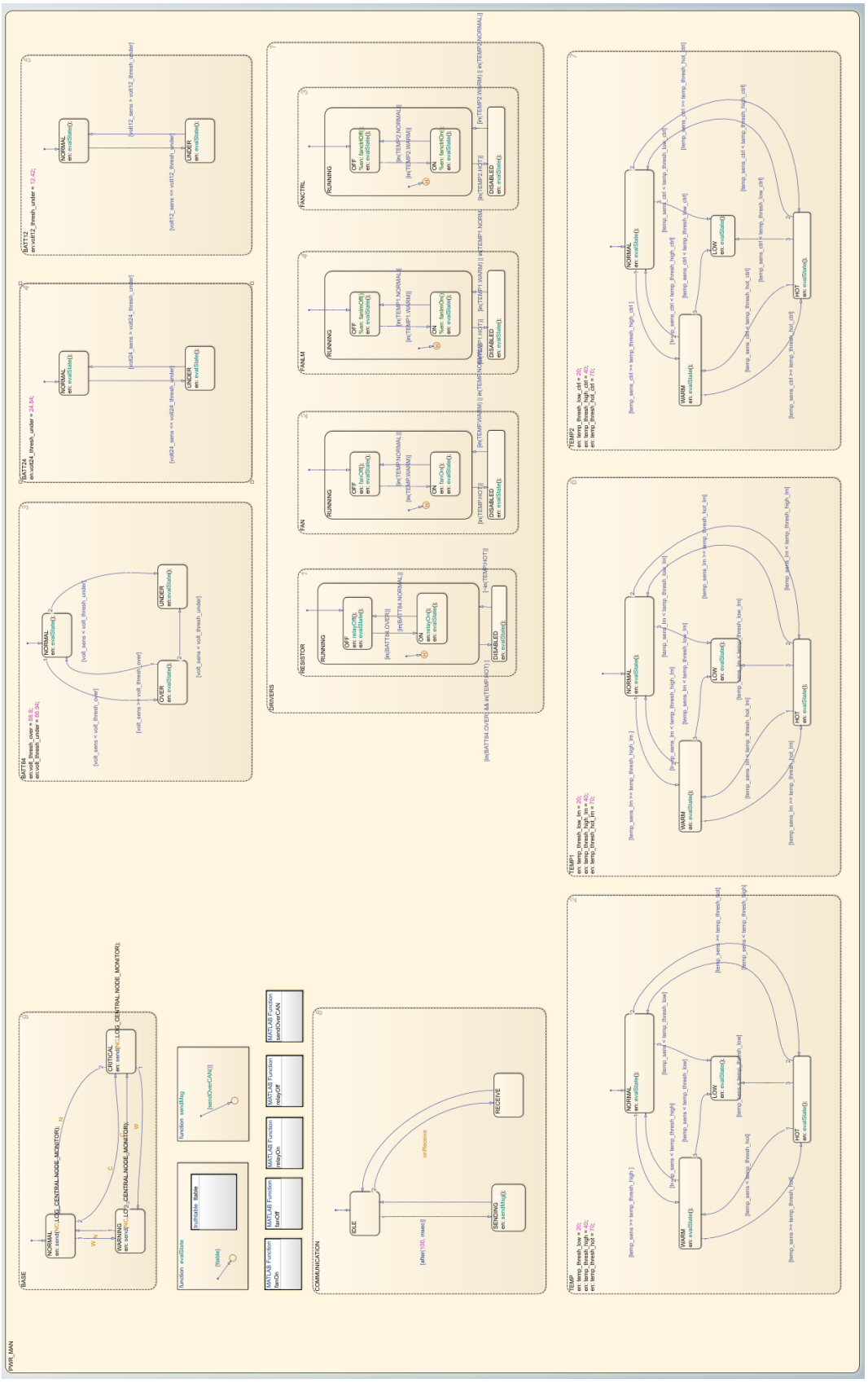

Figure G.1: Power management node state flow

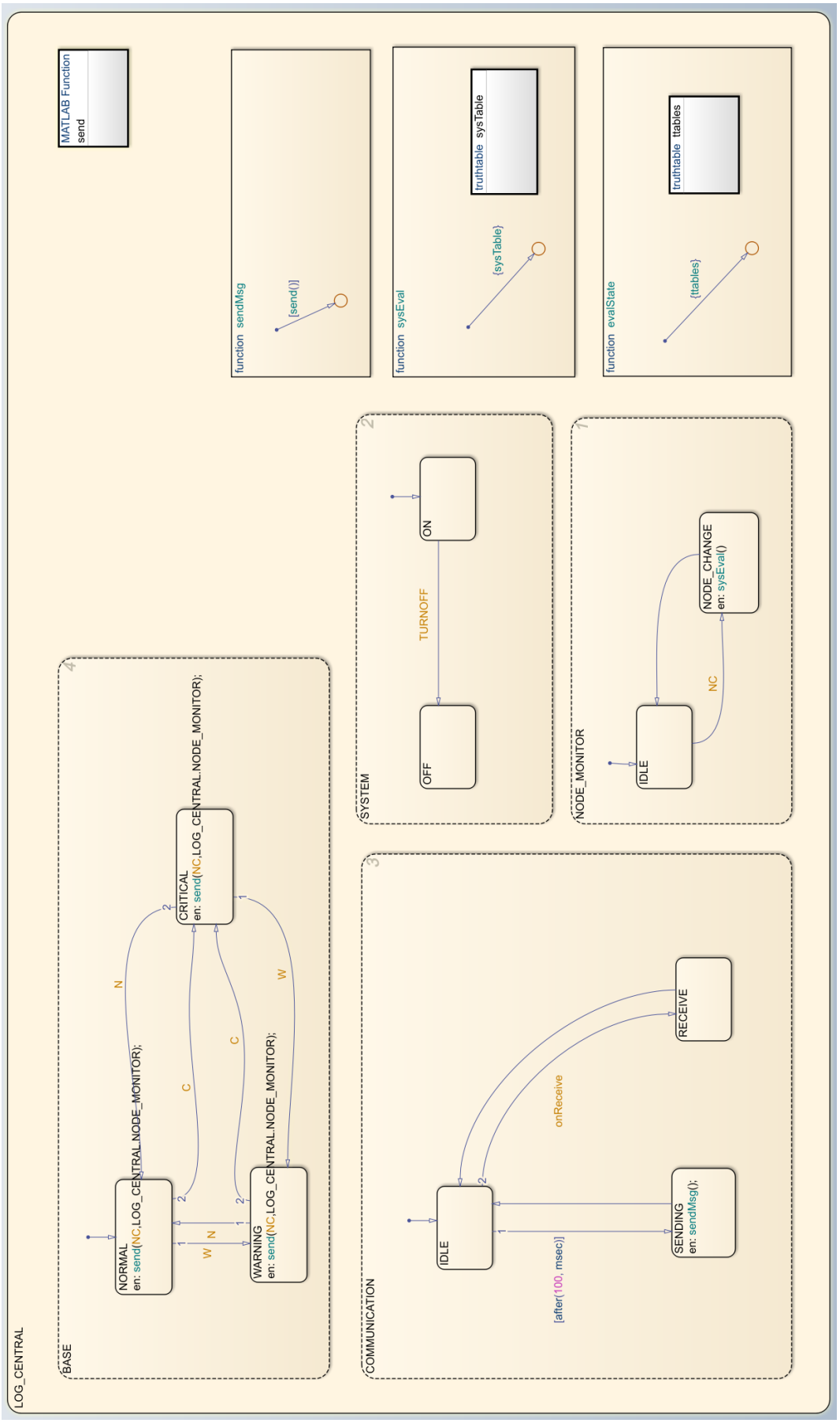

Figure G.2: Logging node state flow

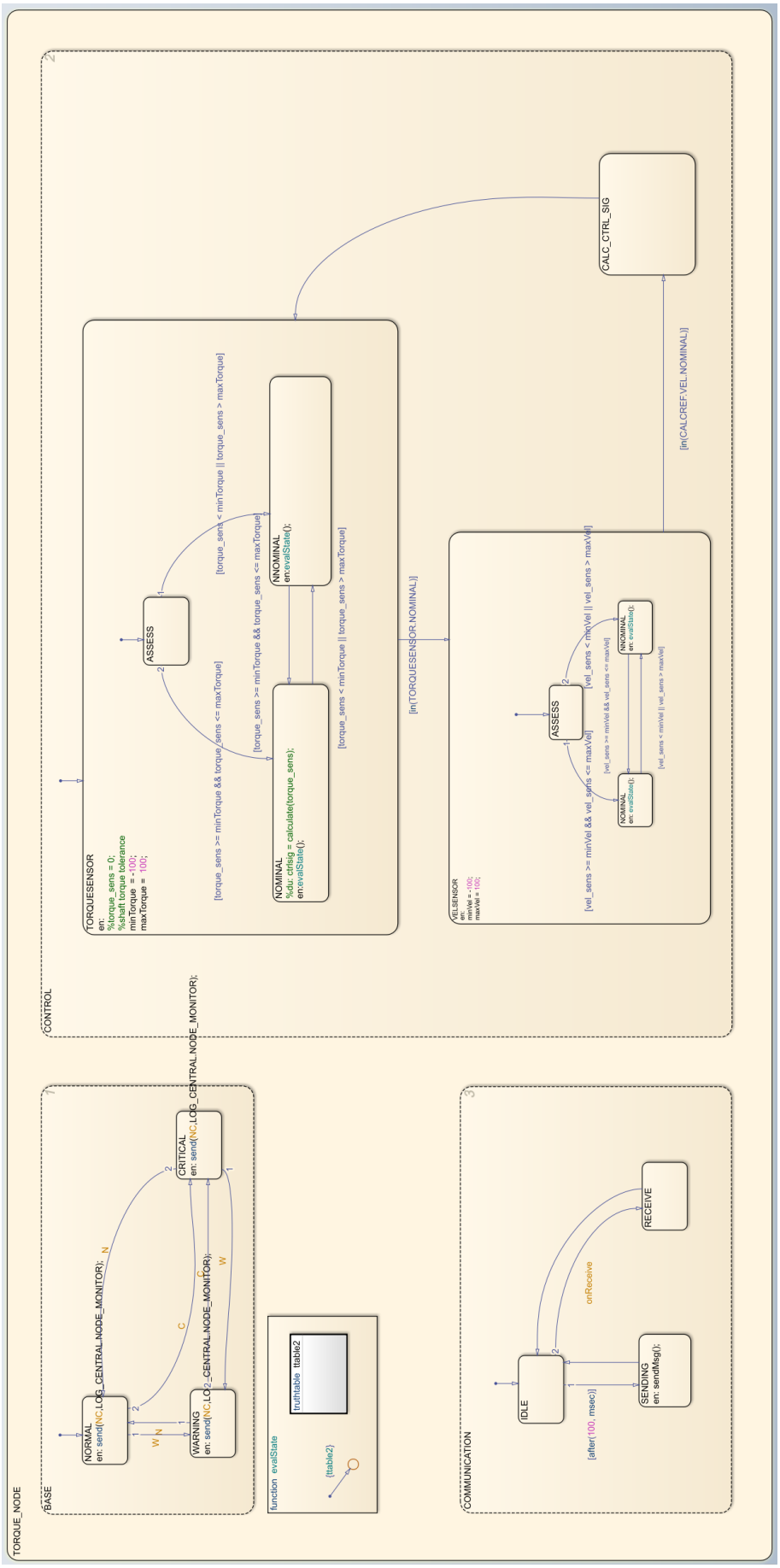

Figure G.3: Torque control node state flow

H - UI

Matlab script to upload track profile on Arduino:

```
delete(instrfindall);
arduino=serial('COM8','BaudRate',115200);

len = 1024;
chunk = 64;
height_diff = 60;
track_length = 5000;
x = linspace(0,track_length,len-1);
y = @(x) height_diff*(sin(x*8*pi/track_length)+1);
uphill = uint8(floor(y(x)));
plot(x,y(x));
z = uint8(1);
data = [uphill,z];

fopen(arduino);
pause(1);

for i = linspace(0,len-chunk,len/chunk)
      disp('sending:');
      disp(data(i+1:i+chunk));
      fwrite(arduino,data(i+1:i+chunk));
end

fclose(arduino);
```

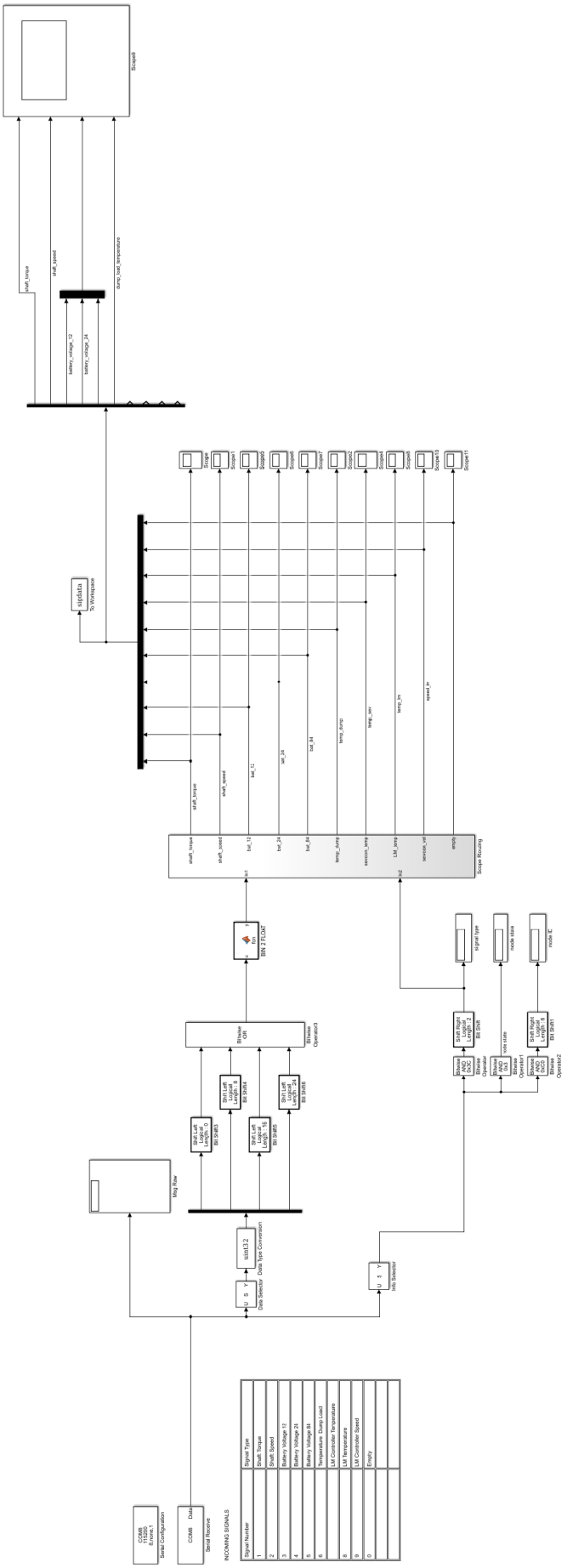

Figure H.1: Simulink UI

I - Code

This appendix contains the code written for the test rig. Sections I.1, I.2 and I.3 contains the main code for the three embedded nodes: the Data Logging Node, the Torque Control Node and the Power Management node. Section I.4 contains the code for the Node library which contains source code for functionality that is common between nodes, such as IO functions for sending and receiving data on the CAN bus, definitions of how a message sent between nodes is formatted and a module for handling states.

I.1 Data Logging Node

```
1  #define DEBUG                                    0
2
3  #define _TASK_TIMECRITICAL
4  #define _TASK_WDT_IDS
5  #define _TASK_PRIORITY
6
7  #include "TaskScheduler.h"
8  #include "Node.h"
9
10 #define GET_TRACK_EN                             1
11
12 #define READ_MS                                  10
13 #define TRACK_ARR_LEN                            1024
14
15 // THE CS pin of the version after v1.1 is default to D9
16 // v0.9b and v1.0 is default D10
17 #define CAN_SPI_CS                               9
18 #define STATUS_PIN                               6
19
20 struct node_data
21 {
22         float torque;
23         float vel;
24         float t_ref;
25
26         bool new_torque;
27         bool new_vel;
28         bool new_t_ref;
```

```
29 };
30
31 int receive_track(uint8_t arr[]);
32 void send_track(uint8_t arr[]);
33 int send_msg(int sig, uint8_t * data, int serial);
34 int enter_base_state(State s);
35
36 /* callbacks */
37 void read_can_cb();
38
39 /* global stuff */
40 Scheduler r;
41 IO * io;
42 StateHandler state;
43
44 volatile struct node_data data;
45 uint8_t track_profile[TRACK_ARR_LEN] = {0};
46
47 Task task_read_can(READ_MS, TASK_FOREVER, &read_can_cb, &r);
48
49 void
50 setup()
51 {
52         Serial.begin(115200);
53         pinMode(STATUS_PIN, OUTPUT);
54
55         digitalWrite(STATUS_PIN, LOW);
56
57         io = new IO(IO_CAN_NODE_DATA);
58
59 #if GET_TRACK_EN
60         receive_track(track_profile);
61         send_track(track_profile);
62 #endif
63
64         enter_base_state(NORMAL);
65 #if DEBUG
66         Serial.println("[info] setup done");
67 #endif
68 }
69
70 void
71 loop()
72 {
```

```
73          r.execute();
74  }
75
76  int
77  receive_track(uint8_t arr[])
78  {
79          byte len = 0;
80          uint32_t buf_read_len = 64;
81          uint32_t idx = 0;
82
83          for (;;) {
84                  if (Serial.available() > 0) {
85                          if (idx > TRACK_ARR_LEN - 65) {
86                                  buf_read_len = TRACK_ARR_LEN - idx;
87                          }
88
89                          len = Serial.readBytes(&(arr[idx]),
                                buf_read_len);
90                          idx += (uint32_t)len;
91                  }
92
93
94                  if (idx > TRACK_ARR_LEN - 1) {
95                          break;
96                  }
97          }
98
99          return 0;
100 }
101
102 void
103 send_track(uint8_t arr[])
104 {
105         byte err = 0;
106         uint8_t data[8] = {0};
107         uint32_t len = CAN_MAX_CHAR_IN_MESSAGE;
108
109         for (int i = 0; i < TRACK_ARR_LEN - 1; i += len) {
110                 if (i > TRACK_ARR_LEN - 8) {
111                         len = TRACK_ARR_LEN - i;
112                 }
113
114                 memcpy(data, (uint8_t *)&(arr[i]), len);
115                 do {
```

```
116                         err = io->send_msg(data, len);
117                 } while (err != CAN_OK);
118                 delay(10);
119         }
120 }
121
122 int
123 send_msg(int sig, uint8_t * data, int serial)
124 {
125         struct msg m;
126         m.hdr.val = 0;
127         for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
128                 m.data[i] = 0;
129         }
130
131         m.hdr.bits.sig_t = sig;
132         m.hdr.bits.node_state = state.get_state(BASE);
133         m.hdr.bits.node_id = IO_CAN_NODE_DATA;
134
135         m.data[0] = m.hdr.val;
136
137         switch (sig) {
138         case MSG_SIG_TORQUE:
139                 memcpy((void *)&(m.data[1]), data, sizeof(float));
140                 break;
141
142         case MSG_SIG_VEL:
143                 memcpy((void *)&(m.data[1]), data, sizeof(float));
144                 break;
145
146         case MSG_SIG_EMPTY:
147                 memcpy((void *)&(m.data[1]), 0, 7);
148                 break;
149         }
150
151         if (serial) {
152                 Serial.write((uint8_t *)m.data,
                    CAN_MAX_CHAR_IN_MESSAGE);
153
154         } else {
155
156                 if (io->send_msg((uint8_t *)m.data,
                    CAN_MAX_CHAR_IN_MESSAGE) != CAN_OK) {
157                         return -1;
```

```
158                     }
159             }
160
161             return 0;
162     }
163
164     int
165     enter_base_state(State s)
166     {
167             switch (s) {
168             case NORMAL:
169                     r.disableAll();
170                     task_read_can.enable();
171                     state.set_state(BASE, NORMAL);
172                     digitalWrite(STATUS_PIN, HIGH);
173                     break;
174
175             default:
176                     return -1;
177             }
178     }
179
180     void
181     read_can_cb()
182     {
183             struct msg in;
184             uint8_t data[8] = {0};
185             uint8_t len = 0;
186             uint32_t can_id = 0;
187
188             for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
189                     in.data[i] = 0;
190             }
191
192             while (io->get_msg(in.data, &len, &can_id) == CAN_OK) {
193     #if DEBUG
194                     Serial.print("got msg:\tid:");
195                     Serial.print(can_id, DEC);
196                     Serial.print("\tdata:");
197                     for (int i = 0; i < len; ++i) {
198                             Serial.print(in.data[i], HEX);
199                             Serial.print(" ");
200                     }
201
```

```
202                 Serial.println();
203 #else
204
205                 /* format sevcon message if message received is not
                        from torque or power nodes */
206                 if (can_id != IO_CAN_NODE_TORQUE && can_id !=
                        IO_CAN_NODE_POWER) {
207 #if DEBUG
208                 Serial.println("got sevcon msg");
209 #endif
210                         /* XXX can be done without copying buffer
                                */
211                         memcpy(data, in.data, len);
212                         in.hdr.val = 0;
213
214                         switch (can_id) {
215                         case IO_CAN_SEVCON_VEL:
216                                 in.hdr.bits.node_id =
                                        IO_CAN_NODE_SEVCON;
217                                 in.hdr.bits.sig_t =
                                        MSG_SIG_VEL_CTRL;
218                                 break;
219
220                         case IO_CAN_SEVCON_TMP_CTRL:
221                                 in.hdr.bits.node_id =
                                        IO_CAN_NODE_SEVCON;
222                                 in.hdr.bits.sig_t =
                                        MSG_SIG_TMP_CTRL;
223                                 break;
224
225                         case IO_CAN_SEVCON_TMP_LM:
226                                 in.hdr.bits.node_id =
                                        IO_CAN_NODE_SEVCON;
227                                 in.hdr.bits.sig_t = MSG_SIG_TMP_LM;
228                                 break;
229
230                         default:
231 #if DEBUG
232                                 Serial.println("[err] unknown msg
                                        received");
233 #endif
234                                 return;
235                         }
236
```

```
237                         in.data[0] = in.hdr.val;
238                         memcpy(&(in.data[1]), data, len);
239                     }
240
241                     Serial.write(in.data, CAN_MAX_CHAR_IN_MESSAGE);
242
243 #endif /* DEBUG */
244
245         }
246 }
```

I.2 Torque Control Node

```
1  #define _TASK_WDT_IDS
2  #define _TASK_TIMECRITICAL
3  #define _TASK_PRIORITY
4
5  #include "TaskScheduler.h"
6  #include "Node.h"
7  #include <math.h>
8
9  #define DEBUG                                   1
10
11 #define CAN_EN                                  1
12 #define TORQUE_EN                               1
13 #define VEL_EN                                  1
14 #define GET_TRACK_EN                            0
15
16 #define TORQUE_SAMPLE_MS                        50
17 #define VEL_SAMPLE_MS                           100
18 #define SEND_LOG_DATA_MS                        300
19
20 /* number of data points in track profile */
21 #define TRACK_ARR_LEN                           1024
22
23 /* torque sensor calibration values for conversions */
24 #define TORQUE_FREQ_SHIFT                       10044
25 #define TORQUE_VEL_Z                            60
26
27 #define TORQUE_MAX                              30
28
29 /* pin definitions */
30 #define TORQUE_IRQ_PIN_TORQUE                   3
```

```
31 #define TORQUE_IRQ_PIN_VEL                    18
32
33 #define THROTTLE_PIN                          5 // Timer 3
34 #define BRAKE_PIN                             7 // Timer 4
35
36 // THE CS pin of the version after v1.1 is default to D9
37 // v0.9b and v1.0 is default D10
38 #define CAN_SPI_CS                            9
39
40 struct node_data
41 {
42         float torque;
43         float vel;
44         float t_ref;
45
46         float t_sum;
47         float v_sum;
48
49         uint32_t v_count;
50         uint32_t t_count;
51
52         bool new_data;
53 };
54
55 /* dynamics parameters */
56 uint8_t track_p[TRACK_ARR_LEN] = {0};
57
58 /* used to get track from can bus */
59 int can_get_track(uint8_t * track);
60 void check_delay();
61
62 float track_res = 0;
63 const float wheel_r = 0.3;
64 const float transmission = 0.1415;
65 const float roll_coeff = 6.874;
66 const float mg = 1375;
67 const float j_bike = 0.0342;
68 const float drag = 0.3216;
69 const float p = 1;
70
71 volatile float vel_old = 0;
72 volatile float pos = 0;
73 volatile float pos_x = 0;
74 volatile float pos_x_since_last = 0;
```

```
75 volatile float cos_theta = 0;
76 volatile float sin_theta = 0;
77
78 /* forward declarations */
79 void update_pos(float vel);
80 void update_track_angles();
81
82 float get_acc(float vel, float old);
83 float get_torque_ref(float acc, float vel_ms_MUT);
84 float get_torque_cmd(float torque, float torque_ref);
85
86 int set_throttle(float t_ref);
87
88 /* XXX shall only be used for measuring torque */
89 volatile uint32_t t_duration_us;
90 volatile uint32_t t_pulse_count;
91 volatile uint32_t t_prev_us;
92
93 /* XXX shall only be used for measuring velocity */
94 volatile uint32_t v_duration_us;
95 volatile uint32_t v_pulse_count;
96 volatile uint32_t v_prev_us;
97
98 /* callbacks */
99 void t_pulse_count_cb();
100 void v_pulse_count_cb();
101
102 void sample_data_cb();
103 void print_cb();
104
105 void send_log_data_cb();
106
107 Scheduler hpr; // Highest priority scheduler
108 Scheduler r;
109
110 StateHandler state;
111
112 #if CAN_EN
113 IO * io;
114 #endif
115
116 /* struct that holds node data */
117 volatile struct node_data data;
118
```

```
119 /* Tasks */
120 Task tsk_sample(TORQUE_SAMPLE_MS, TASK_FOREVER, &sample_data_cb, &
      hpr);
121 Task tsk_can_send(SEND_LOG_DATA_MS, TASK_FOREVER, &send_log_data_cb
      , &r);
122
123 #if DEBUG
124 Task tsk_print(10, TASK_FOREVER, &print_cb, &r);
125 #endif
126
127 void
128 setup()
129 {
130         Serial.begin(115200);
131
132
133 #if CAN_EN
134         io = new IO(IO_CAN_NODE_TORQUE);
135
136 #if DEBUG
137         Serial.println("[dbg] waiting for track");
138 #endif
139
140 #if GET_TRACK_EN
141         /* wait to receive can track on bus */
142         if (can_get_track(track_p) < 0) {
143                 Serial.println("[err] failed to get track");
144         }
145
146         track_res = float(track_p[TRACK_ARR_LEN - 1]);
147         update_track_angles();
148 #else
149         track_res = 1;
150 #endif
151
152 #if DEBUG
153         Serial.print("[dbg] track received, track res:");
154         Serial.println(track_res, 2);
155 #endif
156
157 #endif /* CAN_EN */
158
159 #if TORQUE_EN
160         attachInterrupt(digitalPinToInterrupt(TORQUE_IRQ_PIN_TORQUE
```

```
            ),
161                         t_pulse_count_cb, RISING);
162 #endif /* TORQUE_EN */
163
164 #if VEL_EN
165         attachInterrupt(digitalPinToInterrupt(TORQUE_IRQ_PIN_VEL),
166                         v_pulse_count_cb, RISING);
167 #endif /* VEL_EN */
168
169         r.setHighPriorityScheduler(&hpr); // set priority of
              schedulers
170
171         /* enable tasks */
172         tsk_sample.setId(0);
173         tsk_sample.enable();
174
175         tsk_can_send.setId(1);
176         tsk_can_send.enable();
177
178 #if DEBUG
179         tsk_print.setId(2);
180         tsk_print.enable();
181 #endif
182
183         state.set_state(BASE, NORMAL);
184 }
185
186 void
187 loop()
188 {
189         r.execute();
190 }
191
192 int
193 can_get_track(uint8_t * track)
194 {
195         uint8_t len = 0;
196         uint8_t buf[CAN_MAX_CHAR_IN_MESSAGE] = {0};
197
198         uint32_t idx = 0;
199
200         do {
201                 if (io->get_msg(buf, &len) != CAN_NOMSG) {
202                         if (idx > TRACK_ARR_LEN - len) {
```

```
203                                     len = TRACK_ARR_LEN - idx;
204                             }
205
206                             memcpy((void *)&(track[idx]), buf, len);
207                             idx += len;
208 #if DEBUG
209                             Serial.print("[dbg] idx:");
210                             Serial.print(idx, DEC);
211                             Serial.print("\t");
212
213                             for (int i = 0; i < 8; ++i) {
214                                     Serial.print(buf[i], DEC);
215                                     Serial.print(" ");
216                             }
217                             Serial.println();
218 #endif
219                     }
220
221         } while (idx < TRACK_ARR_LEN);
222
223 #if DEBUG
224         Serial.println("[dbg] got track:");
225         int j = 0;
226
227         for (int i = 0; i < TRACK_ARR_LEN; ++i) {
228                 Serial.print(track[i]);
229                 if (j++ < 17) {
230                         Serial.print("\t");
231
232                 } else {
233                         Serial.println();
234                         j = 0;
235                 }
236         }
237 #endif
238 }
239
240 void
241 check_delay()
242 {
243         long delay = 0;
244         Scheduler &s = Scheduler::currentScheduler();
245         Task &task = s.currentTask();
246
```

```
247         if ((delay = task.getStartDelay()) > 0) {
248                 Serial.print("[err] tsk:");
249                 Serial.print(task.getId());
250                 Serial.print(" delay:");
251                 Serial.println(delay);
252         }
253 }
254
255 void
256 update_pos(float vel)
257 {
258         /* speed in rad/s on torque sensor shaft */
259         float vel_rads = vel * 3.14 / 30;
260         /* speed in m/s MUT wheel */
261         float vel_ms_MUT = transmission * wheel_r * vel_rads;
262
263         /* accumulated sum of distance covered */
264         pos += vel_ms_MUT * TORQUE_SAMPLE_MS * 0.001;
265
266         pos_x_since_last += vel_ms_MUT * TORQUE_SAMPLE_MS * 0.001 *
267                 cos_theta;
268
269         if (pos_x_since_last > track_res){
270                 update_track_angles();
271         }
272 }
273
274 float
275 get_acc(float vel, float old)
276 {
277   return (vel - old) / TORQUE_SAMPLE_MS;
278 }
279
280 void
281 update_track_angles()
282 {
283         float diff_h = 0.0; // height diff
284         float hyp = 0.0; // hypothenuse
285         pos_x += pos_x_since_last;
286         pos_x_since_last = 0;
287
288         diff_h = track_p[(int)floor(pos_x / track_res) + 1] -
289                 track_p[(int)floor(pos_x / track_res)];
290
```

```
291         hyp = sqrt(pow(diff_h, 2) + pow(track_res, 2));
292         cos_theta = track_res / hyp;
293         sin_theta = diff_h / hyp;
294 }
295
296 float
297 get_torque_ref(float acc, float vel_ms_MUT)
298 {
299         float t_internal = acc * j_bike; // torque internal
300
301         float air_drag = pow(vel_ms_MUT, 2) * drag; // force from
              air_drag
302         float g_force = mg * sin_theta; // force from gravity
303
304         float t_ext = (air_drag + g_force + roll_coeff) *
              transmission *
305                 wheel_r; // torque external
306
307         float torque_ref = t_internal + t_ext;
308         return torque_ref;
309 }
310
311 float
312 get_torque_cmd(float torque, float torque_ref)
313 {
314   float torque_cmd = p * (torque_ref - torque);
315   return torque_cmd;
316 }
317
318 int
319 set_throttle(float t_ref)
320 {
321         int duty_cycle = int((1 - (t_ref / TORQUE_MAX)) * 255);
322         analogWrite(THROTTLE_PIN, duty_cycle);
323         return 0;
324 }
325
326
327 /* callbacks */
328 void
329 t_pulse_count_cb()
330 {
331         uint32_t current_us = micros();
332         t_duration_us += current_us - t_prev_us;
```

```
333         t_prev_us = current_us;
334         t_pulse_count++;
335 }
336
337 void
338 v_pulse_count_cb()
339 {
340         uint32_t current_us = micros();
341         v_duration_us += current_us - v_prev_us;
342         v_prev_us = current_us;
343         v_pulse_count++;
344 }
345
346 void
347 sample_data_cb()
348 {
349         check_delay();
350
351         float v_f = 0.0; // vel freq
352         float t_f = 0.0; // torque freq
353         float t = 0.0; // torque
354         float v = 0.0; // vel
355
356         float pos = 0.0;
357         float acc = 0.0;
358         float vel_mut = 0.0;
359
360         float t_ref = 0.0;
361         float t_err = 0.0;
362
363         /* snapshot of torque and vel */
364         uint32_t t_dur = t_duration_us;
365         uint32_t t_p_count = t_pulse_count;
366
367         uint32_t v_dur = v_duration_us;
368         uint32_t v_p_count = v_pulse_count;
369
370         /* reset counters */
371         t_duration_us = 0;
372         t_pulse_count = 0;
373         v_duration_us = 0;
374         v_pulse_count = 0;
375
376         /* calculate frequencies */
```

```
377         v_f = 1e6 / float(v_dur);
378         v_f *= v_p_count;
379
380         t_f = 1e6 / float(t_dur);
381         t_f *= t_p_count;
382
383         /* calculate torque */
384         t = t_f - TORQUE_FREQ_SHIFT;
385         t /= 50;
386
387         /* calculate vel */
388         if (isnan(v_f)) {
389                 v = 0;
390
391         } else {
392                 v = v_f * 60;
393                 v /= TORQUE_VEL_Z;
394                 v = (t > 0) ? v : -v;
395         }
396
397         /* updates global variables for position */
398         update_pos(v);
399         acc = get_acc(v, vel_old);
400
401         vel_mut = (wheel_r * transmission*  v * 3.14) / 30;
402
403         t_ref = get_torque_ref(acc, vel_mut);
404
405         /* TODO compute control signal and set throttle output
                using set_throttle(float t_ref)
406                 which sets the duty cycle of the pwm throttle pin
                    */
407
408         /* update global variable vel_old */
409         vel_old = v;
410
411         /* store torque and vel */
412         data.torque = t;
413         data.t_sum += t;
414         ++data.t_count;
415
416         data.vel = v;
417         data.v_sum += v;
418         ++data.v_count;
```

```
419
420         data.t_ref = t_ref;
421         data.new_data = true;
422 }
423
424 void
425 send_log_data_cb()
426 {
427         struct msg msg_t; // torque message
428         struct msg msg_v; // velocity message
429         int s = (int)(state.get_state(BASE));
430
431         float t_avg = data.t_sum / data.t_count;
432         data.t_sum = 0;
433         data.t_count = 0;
434
435         float v_avg = data.v_sum / data.v_count;
436         data.v_sum = 0;
437         data.v_count = 0;
438
439         // initialize messages
440         msg_t.hdr.val = 0;
441         msg_v.hdr.val = 0;
442
443         for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
444                 msg_t.data[i] = 0;
445                 msg_v.data[i] = 0;
446         }
447
448         /* configure torque message */
449         msg_t.hdr.bits.node_state = s;
450         msg_t.hdr.bits.sig_t = MSG_SIG_TORQUE;
451         msg_t.hdr.bits.node_id = IO_CAN_NODE_TORQUE;
452         msg_t.data[0] = msg_t.hdr.val;
453
454         memcpy(&(msg_t.data[1]), (uint8_t *)&t_avg, sizeof(t_avg));
455
456         /* configure vel message */
457         msg_v.hdr.bits.node_state = s;
458         msg_v.hdr.bits.sig_t = MSG_SIG_VEL;
459         msg_v.hdr.bits.node_id = IO_CAN_NODE_TORQUE;
460         msg_v.data[0] = msg_v.hdr.val;
461
462         memcpy(&(msg_v.data[1]), (uint8_t *)&v_avg, sizeof(v_avg));
```

```
463
464         io->send_msg(msg_v.data, CAN_MAX_CHAR_IN_MESSAGE);
465 #if 0
466         Serial.print("[dbg] vel data sent ");
467         for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
468                 Serial.print(msg_v.data[i], DEC);
469                 Serial.print(" ");
470         }
471         Serial.println();
472 #endif
473         delay(1);
474
475         io->send_msg(msg_t.data, CAN_MAX_CHAR_IN_MESSAGE);
476 #if 0
477         Serial.print("[dbg] torque data sent:");
478         Serial.print(t_avg);
479         Serial.print(" len:");
480         Serial.print(sizeof(t_avg), DEC);
481         Serial.print(" ");
482         for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
483                 Serial.print(msg_t.data[i], HEX);
484                 Serial.print(" ");
485         }
486         Serial.println();
487 #endif
488 }
489
490
491 #if DEBUG
492 void
493 print_cb()
494 {
495         if (data.new_data) {
496                 Serial.print("t:");
497                 Serial.print(data.torque, 2);
498
499                 Serial.print(" v:");
500                 Serial.print(data.vel, 2);
501
502                 Serial.print(" t_ref:");
503                 Serial.print(data.t_ref, 2);
504
505                 Serial.print(" pos:");
506                 Serial.print(pos, 2);
```

```
507
508                 Serial.print(" pos_x:");
509                 Serial.print(pos_x, 2);
510
511                 Serial.print(" pos_x_since_last:");
512                 Serial.print(pos_x_since_last, 2);
513
514                 Serial.print(" cos theta:");
515                 Serial.println(cos_theta, 2);
516
517                 data.new_data = false;
518         }
519 }
520 #endif
```

I.3 Power Management Node

```
1  #include "Node.h"
2  #include <High_Temp.h> //Library for temperature sensor
3
4  /*
5          0 - ~12.7volt voltmeter
6          works with 5volt Arduinos
7          uses 5volt reference
8          3.3k resistor from A0 to ground, and 5.6k resistor from A0
              to +batt
9          100n capacitor from A0 to ground for stable readings
10 */
11
12 /*
13         0 - ~25.4volt voltmeter
14         works with 5volt Arduinos
15         uses 5volt reference
16         1.3k resistor from A0 to ground, and 5.6k resistor from A0
              to +batt
17         100n capacitor from A0 to ground for stable readings
18 */
19
20 /*
21         0 - ~89 volt voltmeter
22         works with 5volt Arduinos
23         uses 5volt reference
24         5.6k resistor from A0 to ground, and 100k resistor from A0
```

```
          to +batt
25        100n capacitor from A0 to ground for stable readings
26 */
27
28 /*
29  * Temperature Sensor
30  *
31  * Uses library "Grove_HighTemp_Sensor-master"
32  *
33  * Hardware connections:
34  *
35  * Grove board          ARDUINO
36  * GND (red)            GND
37  * Vcc (black)          5V
38  * A0 (yellow)          SCL or A5
39  * A1 (white)           SDA or A4
40  *
41  * Thermocouple         Grove board
42  * Red                  +
43  * White                -
44  *
45  */
46
47 HighTemp ht(A4, A5); //Assign "room temperature" and "thermocouple"
       pins to A4 and A5
48
49
50 #define Temp_thresh_low 20.0    //Set temperature to start cooling
51 #define Temp_thresh_high 40.0  //Set temperature to start cooling
52 #define Temp_thresh_hot 70.0    //Set temperature to start cooling
53 #define FANS_pin 10 //Define pin used to open/close MOSFET for fans
54 #define SSR_pin 8 //Define pin used to open/close MOSFET for SSR
55
56 #define CAN_EN                                  1
57
58 #define NODE_STATE_NORMAL                       0
59 #define NODE_STATE_WARNING                      1
60 #define NODE_STATE_CRITICAL                     2
61
62 unsigned int total_12 = 0;
63 unsigned int total_24 = 0;
64 unsigned int total_84 = 0; // holds readings
65
66 float voltage_12 = 0;
```

```
67 float voltage_24 = 0;
68 float voltage_84 = 0; // converted to volt
69
70 float Temp = 0;; // store temperature value
71
72 String Batt12, Batt24, Batt84; // battery modes
73 String tempState; // temperature mode
74 String fansState, ssrState; // fans and SSR state
75 String nodeState; // current state of the power management mode
76 unsigned long time;
77
78 #if CAN_EN
79 IO * io;
80 void send_log_data();
81 int configure_msg(struct msg * m, uint8_t node_state, uint8_t sig_t
     ,
82                   uint8_t * data, size_t len);
83 #endif
84
85 void setup() {
86         Serial.begin(115200); // set serial monitor to this value
87         pinMode(FANS_pin, OUTPUT); // output for fans
88         pinMode(SSR_pin, OUTPUT); // output for SSR
89         ht.begin(); // start temperature sensor routine
90         analogReference(DEFAULT); // use default 5V reference for
             voltage calculations
91
92         Serial.println("[info] initiating can");
93 #if CAN_EN
94         io = new IO(IO_CAN_NODE_POWER);
95 #endif /* CAN_EN */
96
97         Serial.println("[info] setup done");
98 }
99
100 float readAnalogVoltage(unsigned char pin) { // Returns voltage
      value (analog ranging from 0 to 1023)
101         float analogInput = 0;
102         analogRead(pin); // one unused reading to clear any ghost
              charge
103         for (int x = 0; x < 64; x++) { // 64 analogue readings for
              averaging
104                 analogInput = analogInput + analogRead(pin); // add
                      each value
```

```
105         }
106         return analogInput;
107 }
108
109 String powersupplyMode(float psVoltage) { // Returns state of the
    power supply battery
110         if (psVoltage < 12.40) {
111                 return "under";
112         }
113         else {
114                 return "normal";
115         }
116 }
117
118 String mutMode(float mutVoltage) { // Returns state of the MUT
    battery bank
119         if (mutVoltage < 24.80) {
120                 return "under";
121         }
122         else {
123                 return "normal";
124         }
125 }
126
127 String loadMode(float loadVoltage) { // Returns state of the laod
    motor battery bank
128         if (loadVoltage >= 89.0) {
129                 return "over";
130         }
131         if (loadVoltage < 89.0 && loadVoltage >= 86.8) {
132                 return "normal";
133         }
134         if (loadVoltage < 86.8) {
135                 return "under";
136         }
137         else{
138         return "error: case not considered"; //If case not covered,
              return an error message
139         }
140 }
141
142 String tempMode(float temperature) { // Returns state of the dump
    load temperature
143         if (temperature >= Temp_thresh_hot) {
```

```
144                 return "hot";
145         }
146         if (temperature >= Temp_thresh_high && Temp <
               Temp_thresh_hot) {
147                 return "warm";
148         }
149         if (temperature >= Temp_thresh_low && Temp <
               Temp_thresh_high) {
150                 return "normal";
151         }
152         if (temperature < Temp_thresh_low) {
153                 return "low";
154         }
155         else{
156         return "error: case not considered"; //If case not covered,
               return an error message
157         }
158 }
159
160 String fansMode(String temperatureMode) { // Return state of the
    dump load fans
161         if (temperatureMode == "warm") {
162                 return "on";
163         }
164         if (temperatureMode == "normal") {
165                 return "off";
166         }
167         if (temperatureMode == "low") {
168                 return "off";
169         }
170         if (temperatureMode == "hot") {
171                 return "disabled";
172         }
173         else{
174         return "error: case not considered"; //If case not covered,
               return an error message
175         }
176 }
177
178 String ssrMode(String batteryMode, String temMode) { //Returns
    state of the solid state relay
179         if (batteryMode == "over") {
180                 return "on";
181         }
```

```
182         if (batteryMode == "normal") {
183                 return "off";
184         }
185         if (batteryMode == "under") {
186                 return "off";
187         }
188         if (batteryMode == "over" && temMode == "hot") {
189                 return "disabled";
190         }
191         else{
192         return "error: case not considered"; //If case not covered,
                 return an error message
193         }
194 }
195
196 String evaluateMode(String tempStatus, String powersuppStatus,
    String mutStatus, String loadStatus, String ssrStatus, String
    fanStatus){
197         if (loadStatus == "under" || mutStatus == "under" ||
            powersuppStatus == "under" || tempStatus == "low" || (
            loadStatus == "over" && ssrStatus == "disabled")){
198                 return "CRITICAL";
199         }
200         else if ((loadStatus != "under" && loadStatus != "over" &&
            ssrStatus != "disabled" && fanStatus == "disabled") || (
            loadStatus != "under" && loadStatus != "over" &&
            ssrStatus == "disabled")){
201                 return "WARNING";
202         }
203         else {
204                 return "NORMAL";
205         }
206 }
207
208 void loop() {
209
210         Temp = ht.getThmc(); //Get temperature value
211
212         total_12 = readAnalogVoltage(A0); // Get analog voltage
            value for the power supply battery (pin A0)
213         total_24 = readAnalogVoltage(A1); // Get analog voltage
            value for the power supply battery (pin A1)
214         total_84 = readAnalogVoltage(A2); // Get analog voltage
            value for the power supply battery (pin A2)
```

```
215
216         // Convert readings to volt. Multiply by 5/1023 to go from
               analog value to voltage. Multiply by (R1/R2)/R2 to go
               from arduino voltage to real voltage. Multiply by 1/64
               for averaging.
217         voltage_12 = total_12 * 0.00020696; // Calibrate by
               adjusting the last digits
218         voltage_24 = total_24 * 0.00040824; // Calibrate by
               adjusting the last digits
219         voltage_84 = total_84 * 0.001621; // Calibrate by adjusting
                the last digits
220
221         Batt12 = powersupplyMode(voltage_12); // get state of the
               power supply battery
222         Batt24 = mutMode(voltage_24); // get state of MUT battery
               bank
223         Batt84 = loadMode(voltage_84); // get state of the load
               battery bank
224         tempState = tempMode(Temp); // get state of the dump load
               temperature
225
226         fansState = fansMode(tempState); // get state of the fans
227         ssrState = ssrMode(Batt84, tempState); // get state of the
               SSR
228
229         nodeState = evaluateMode(tempState, Batt12, Batt24, Batt84,
                ssrState, fansState); // get state of the node
230
231 #if CAN_EN
232         send_log_data();
233 #endif
234
235         fansState = "on";
236
237         // Fans control
238         if (fansState == "on") {
239                 digitalWrite(FANS_pin, HIGH); // FANS on
240         }
241         else {
242                 digitalWrite(FANS_pin, LOW); // FANS off
243         }
244
245         // SSR control
246         if (ssrState == "on") {
```

```
247                 digitalWrite(SSR_pin, HIGH); // SSR on
248         }
249         else {
250                 digitalWrite(SSR_pin, LOW); // SSR off
251         }
252
253         //Serial monitor debug
254         //Serial.print("Power Supply Battery voltage (V): ");
255         //Serial.println(voltage_12,2);
256         //Serial.print("Power Supply Battery mode: ");
257         //Serial.println(powersupplyMode(voltage_12));
258         //Serial.print("MUT Battery voltage (V): ");
259         //Serial.println(voltage_24,2);
260         //Serial.print("MUT Battery mode: ");
261         //Serial.println(mutMode(voltage_24));
262         Serial.print("Revolt Battery voltage (V): ");
263         Serial.println(voltage_84,2);
264         Serial.print("Revolt Battery mode: ");
265         Serial.println(loadMode(voltage_84));
266         //delay(1000); // readout delay
267         //Serial.print("Time: ");
268         //time = millis();
269         //Serial.println(time);          //prints time since
              program started
270         delay(10);
271 }
272
273 int
274 configure_msg(struct msg * m, uint8_t node_state, uint8_t sig_t,
275                 uint8_t * data, size_t len)
276 {
277         if (len > 7) {
278                 return -1;
279         }
280
281         /* configure temperature message */
282         m->hdr.bits.node_state = node_state;
283         m->hdr.bits.sig_t = sig_t;
284         m->hdr.bits.node_id = IO_CAN_NODE_POWER;
285         m->data[0] = m->hdr.val;
286         memcpy(&(m->data[1]), data, len);
287
288         return 0;
289
```

```
290 }
291
292 #if CAN_EN
293 void
294 send_log_data()
295 {
296         struct msg msg_tmp; // temperature message
297         struct msg msg_v_12; // velocity message
298         struct msg msg_v_24; // velocity message
299         struct msg msg_v_84; // velocity message
300
301         int s;
302
303         if (nodeState == "NORMAL") {
304                 s = NODE_STATE_NORMAL;
305
306         } else if (nodeState == "WARNING") {
307                 s = NODE_STATE_WARNING;
308
309         } else if (nodeState == "CRITICAL") {
310                 s = NODE_STATE_CRITICAL;
311
312         } else {
313                 Serial.println("[err] node state invalid");
314                 return;
315
316         }
317
318         // initialize messages
319         msg_tmp.hdr.val = 0;
320         msg_v_12.hdr.val = 0;
321         msg_v_24.hdr.val = 0;
322         msg_v_84.hdr.val = 0;
323
324         for (int i = 0; i < CAN_MAX_CHAR_IN_MESSAGE; ++i) {
325                 msg_tmp.data[i] = 0;
326                 msg_v_12.data[i] = 0;
327                 msg_v_24.data[i] = 0;
328                 msg_v_84.data[i] = 0;
329         }
330
331         /* configure messages */
332         configure_msg(&msg_tmp, s, MSG_SIG_TMP_DUMP, (uint8_t *)&
            Temp, sizeof(Temp));
```

```
333         configure_msg(&msg_v_12, s, MSG_SIG_VOLT_12, (uint8_t *)&
               voltage_12, sizeof(voltage_12));
334         configure_msg(&msg_v_24, s, MSG_SIG_VOLT_24, (uint8_t *)&
               voltage_24, sizeof(voltage_24));
335         configure_msg(&msg_v_84, s, MSG_SIG_VOLT_84, (uint8_t *)&
               voltage_84, sizeof(voltage_84));
336
337         /* send messages */
338         io->send_msg(msg_tmp.data, CAN_MAX_CHAR_IN_MESSAGE);
339         delay(10);
340         io->send_msg(msg_v_12.data, CAN_MAX_CHAR_IN_MESSAGE);
341         delay(10);
342         io->send_msg(msg_v_24.data, CAN_MAX_CHAR_IN_MESSAGE);
343         delay(10);
344         io->send_msg(msg_v_84.data, CAN_MAX_CHAR_IN_MESSAGE);
345 }
346 #endif
```

I.4 Node library

I.4.1 Msg

```
1  #ifndef MSG_H
2  #define MSG_H
3
4  /* signal type defs */
5  #define MSG_SIG_EMPTY                    0
6  #define MSG_SIG_TORQUE                   1
7  #define MSG_SIG_VEL                      2
8  #define MSG_SIG_VOLT_12                  3
9  #define MSG_SIG_VOLT_24                  4
10 #define MSG_SIG_VOLT_84                  5
11 #define MSG_SIG_TMP_DUMP                 6
12
13 #define MSG_SIG_TMP_CTRL                 7
14 #define MSG_SIG_TMP_LM                   8
15 #define MSG_SIG_VEL_CTRL                 9
16
17 struct msg {
18         union {
19                 struct {
20                         uint8_t node_state : 2;
21                         uint8_t sig_t : 4;
```

```
22                          uint8_t node_id : 2;
23                  } bits;
24
25                  uint8_t val;
26          } hdr;
27
28          uint8_t data[CAN_MAX_CHAR_IN_MESSAGE];
29  };
30
31  #endif /* MSG_H */
```

I.4.2 IO

Listing I.1: IO.h

```
1   #ifndef IO_H
2   #define IO_H
3
4   #include "Arduino.h"
5   #include <stdint.h>
6   #include "mcp_can.h"
7   #include <SPI.h>
8
9   #define IO_CAN_SPI_CS                                   9
10  #define IO_CAN_SPEED                                    CAN_500KBPS
11
12  /* CAN ids of different nodes */
13  #define IO_CAN_NODE_DATA                                0
14  #define IO_CAN_NODE_TORQUE                              1
15  #define IO_CAN_NODE_POWER                               2
16  #define IO_CAN_NODE_SEVCON                              3
17
18  /* TODO verify these */
19  /* defines of can id's for different sevcon messages. Must
20   * be consistent with the configuration of the sevcon node */
21  #define IO_CAN_SEVCON_VEL                               0x300
22  #define IO_CAN_SEVCON_TMP_CTRL                          0x301
23  #define IO_CAN_SEVCON_TMP_LM                            0x302
24
25  class
26  IO
27  {
28  public:
29          IO(uint32_t id);
30
```

```
31          /* function to send a message on the can bus */
32          int send_msg(const uint8_t * data, uint8_t len);
33
34          /* function to read a message from the incomming buffer */
35          int get_msg(uint8_t * data, uint8_t * len);
36
37          /* get_msg fcns with can id of sender */
38          int get_msg(uint8_t * data, uint8_t * len, uint32_t * id);
39
40  private:
41          uint32_t id;
42          MCP_CAN * bus;
43
44  };
45
46  #endif /* IO_H */
```

Listing I.2: IO.cpp

```
1   #include "IO.h"
2
3   IO::IO(uint32_t id) :
4           id(id)
5   {
6           bus = new MCP_CAN(IO_CAN_SPI_CS);
7
8           while (CAN_OK != bus->begin(IO_CAN_SPEED)) {
9                   Serial.println("[err][IO] bus init failed, retrying
                        ");
10                  delay(100);
11          }
12  }
13
14  int
15  IO::send_msg(const uint8_t * data, uint8_t len)
16  {
17          return bus->sendMsgBuf(this->id, 0, len, data);
18  }
19
20  int
21  IO::get_msg(uint8_t * data, uint8_t * len)
22  {
23          if (bus->checkReceive() == CAN_MSGAVAIL) {
24                  return bus->readMsgBuf(len, data);
25          } else {
```

```
26                  return CAN_NOMSG;
27
28          }
29  }
30
31  int
32  IO::get_msg(uint8_t * data, uint8_t * len, uint32_t * id)
33  {
34          int err = 0;
35
36          if (bus->checkReceive() == CAN_MSGAVAIL) {
37                  err = bus->readMsgBuf(len, data);
38                  *id = bus->getCanId();
39                  return err;
40
41          } else {
42                  return CAN_NOMSG;
43
44          }
45  }
```

I.4.3 StateHandler

Listing I.3: StateHandler.h

```
1  #ifndef STATEHANDLER_H
2  #define STATEHANDLER_H
3
4  #include "Arduino.h"
5  #include <stdint.h>
6
7
8  enum
9  Module
10 {
11         BASE
12         /* applies to all nodes */
13
14 };
15
16 /* definitions of node states */
17 enum
18 State
19 {
20         NORMAL = 0,
```

```
21          WARNING = 1,
22          CRITICAL = 2,
23  };
24
25  class
26  StateHandler
27  {
28  public:
29          StateHandler();
30
31          /**
32           * \brief Returns the current state of the node
33           *
34           * \return 0 on success, < 0 otherwise.
35           *
36           */
37          State get_state(Module m);
38
39          /**
40           * \brief Used to set the state of the node
41           *
42           * \param s The new state for the node.
43           *
44           * \return 0 on success, < 0 otherwise.
45           *
46           */
47          int set_state(Module m, State s);
48
49  private:
50          State base;
51  };
52
53  #endif /* STATEHANDLER_H */
```

Listing I.4: StateHandler.cpp

```
1   #include "StateHandler.h"
2
3   StateHandler::StateHandler() :
4           base(NORMAL)
5   {
6   }
7
8   int
9   StateHandler::set_state(Module m, State s)
```

```
10 {
11         switch (m) {
12         case BASE:
13                 this->base = s;
14                 return 0;
15                 break;
16
17         default:
18                 return -1;
19         }
20 }
21
22 State
23 StateHandler::get_state(Module m)
24 {
25         switch (m) {
26         case BASE:
27                 return this->base;
28
29         default:
30                 Serial.println("[err][get_state] invalid module");
31                 for(;;);
32         }
33 }
```