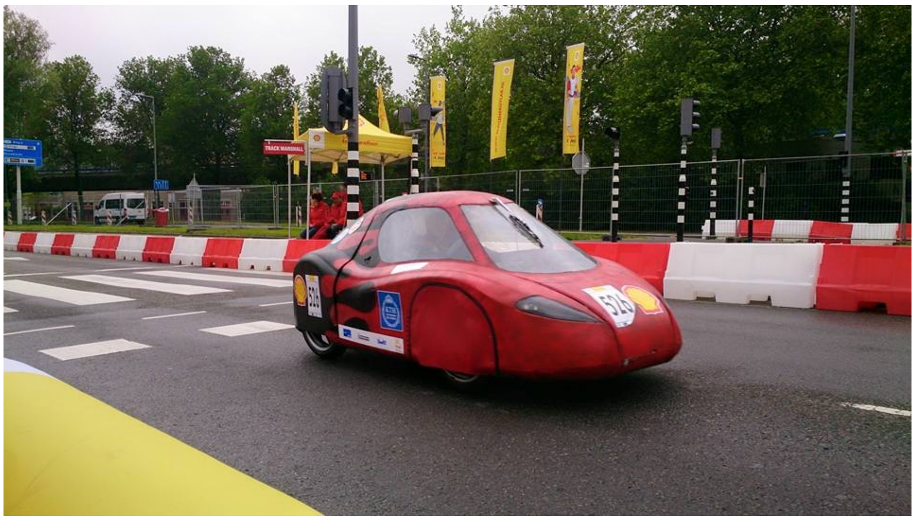
KTH Royal Institute of Technology

The Eco Cars Project - Elba

Architectural and platform design of hybrid electric vehicle powertrains for the KTH Eco Cars project with implementation



Authors:
Milivoj Persson (`milivoj@kth.com`)
Tobias Gustafsson (`togustaf@kth.com`)
Victor Malmgren (`vmal@kth.se`)
Simon Nelson Landén (`simonnl@kth.se`)
Helen Ghattas (`helents@mail.se`)
Ruoyu Zhang (`ruoyu@kth.se`)
Jimmy Karls (`jikarls@kth.se`)

Supervisor:
Mohammad Khodabakhshian Khansari

2016-01-18

Abstract

With an increasing fleet of personal road vehicles the importance of more environmentally friendly solutions with regards to fuel consumption and emissions are graver now more than ever. In addition, more diverse solutions and products are requested for different markets creating complex requirements for manufacturers of these vehicles. This report describes how one solution to this issue can be solved by designing and realizing , both in software and hardware, a modular powertrain for hybrid electric vehicles for Integrated Transport Research Lab at KTH. The powertrain produced is implemented for an urban concept car, designed however for the Shell Eco-marathon and does not accommodate the features of a regular road-legal car. The results of the effort made show a fuel efficiency of 145 km/l gasoline with the concept car Elba with a powertrain based on this platform. Future improvements should make a fuel efficiency of 200 km/l gasoline possible as calculations and tests, real and simulated, points towards this. The modular platform used shows great potential and it is advised that future Eco Cars teams continue the development of the platform, with the created and acquired tools, but also to add new knowledge and finally optimizing it.

Acknowledgement

The Eco Cars team would like to express our sincere thanks to Mikael Hellgren for his continuous support, availability and flexibility, and the mechatronic department's teaching team for enabling and making this project possible by believing in the student team leading the project and allowing a great deal of freedom for creativity. Additional thanks is called for Integrated Transport Research Lab for housing and sponsoring the project. Final thanks goes out to all the numerous sponsors that without doubt made the end result possible.

Nomenclature

Abbreviation	**Description**
CAN	Controller Area Network
CANH	Controller Area Network High
CANL	Controller Area Network Low
DMS	Deadman Switch
DMA	Direct Memory Access
ECU	Electric Control Unit
HEV	Hybrid Electric Vehicle
HW	Hardware
HMI	Human Machine Interface
ICE	Internal Combustion Engine
ITRL	Integrated Transport Research Lab
PCB	Printed Circuit Board
PWM	Pulse Width Modulation
UART	Universal Asynchronous Receiver/Transmitter
UUID	Universally Unique Identifier

Table of Contents

1 Introduction

This is the final report of the project in the course MF2059 Mechatronics Advanced Course at KTH, Royal Institute of Technology, during the authors' second year, third semester, of their masters in mechatronics. The Eco Cars project is a collaboration between the mechatronics department and Integrated Transport Research Lab (ITRL), amongst other departments at KTH, with the focus on developing and realizing smarter and more eco-friendly transportation solutions. In this paper the result of the authors work during the project is presented. The outcome and results are obtained from the realization of the powertrain in the hybrid concept car Elba.

1.1 Background

As the amount of vehicles and goods to be transported by road increases shown in [1], there is an ever increasing need for smarter transportation solutions that creates a smaller environmental impact. One such solution is the hybrid electric vehicle (HEV), which combines both a standard gasoline engine with an efficient electric motor. By combining these two very different technologies the means of transportation will have the potential to house new features and behaviors as well as increased performance. It is therefore desired to develop new tools to effectively design, implement, and these complex systems, but also to devise safe and efficient control strategies.

Apart from having a positive social and environmental value, the project also aims to develop cutting edge technologies and solutions within the field of eco-friendly transportation for ITRL. Because of this, the Eco Cars project and its team members compete in the Shell Eco-marathon in Europe were student teams from all over Europe compete in making the most fuel efficient vehicle to run on the course. For the KTH Eco Cars project, this is a major milestone that exemplifies and measures the performance of the designed and the manufactured powertrain in the vehicle Elba.

Before the competition a number of different student teams were involved in the making of the powertrain. In addition to the mechatronic student team authoring this report there was a mechanical team responsible for the production of the clutch, internal combustinon engine (ICE) team, waste heat recovery team, and one student working with a new design for a tripod driveshaft. After the competition all other teams, except the mechatronics team, pulled out being finished with their contribution.

Since the team consists exclusively of students from the mechatronics track, there is a great need for the team members to evolve as both engineers and also as efficient members of a development team. As a result, some of the effort put into the project is focused on project and team management. This becomes natural as the whole car, not only the powertrain of the vehicle, is developed by a number of different student teams from different fields of study. The mechatronics team has in this context shouldered the role as system architectures and moderators over the full functionality of the vehicle, overseeing the entire project and process.

1.2 Shift of Focus

During the course of the project the focus has shifted as new input was added and experience gained. Starting out in early February 2015 as a project about optimizing the control system of a HEV meant to compete in Shell Eco-marathon, the project did not cover the overall architecture of the powertrain. In fact, it was believed and assumed, before the first encounter with the car itself, that the platform, on which the control system was supposed to be implemented on, already existed. However, this was not the case. Elba did exist, but the powertrain had previously been of pure electric nature and did not support the necessary features and properties imposed by the new vehicle type meant to be implemented (HEV). Since the discovery of this misconception the focus has changed and have been reimagined up until late August 2015.

The first reconsideration with the project was concerning the scope of the task at hand. Shifting from being a pure control engineering task to include system and architecture design the scope increased immensely, but also increased the amount of degrees of freedom.

As time passed and the project progressed it became obvious that an optimized control system of the car was unfeasible as the car was meant to be operational at the time of the competition on the 19th of May, leaving only approximately three months in total for the design and implementation of the complete powertrain. Instead, it became clear that independent of the task and goal of the project, the car had to be able to run in order to fully design and assess the performance of the realized solution. Therefore, a great deal of effort was put into architecture design and making sure that the car would work as a platform for control design. As a logical deduction of this insight the project leaned towards making the architecture, but also the design and work process, more modular to be a platform for different future solutions in order to improve the hand over, as legacy and hand over has been a problem in the past.

With a successfully completed and approved race around the Shell Eco-marathon track in Rotterdam, difficulties emerged and Elba suffered from broken mechanical components and loose connectors. After returning to Stockholm from the competition it was agreed upon within the project group and its supervisors that the focus had to, yet again, be shifted. Previous meetings with students from earlier Eco Cars teams had resulted in a summary that concluded that robustness and reliability are a reoccurring factors that often topples the project at the goal line. This convinced the team that robustness and modularity was to be given an even greater focus for the same reason as stated in the previous paragraph. However, this meant that the control strategies for the current project were unable to be optimized since the car was not fully operational at that time. The reasoning was that even though an optimal control strategy deduced from calculations and assumptions were to be implemented, it would never be verified correctly with a broken vehicle, hence unusable and practically ineffective. It is therefore more beneficial to design and implement a heuristic control algorithm and ensuring that the car is operational in order to verify the control to use this case not only as a proof of concept, but also as a fully functioning prototype, for future Eco Cars projects.

1.3 Purpose

The purpose of this project is to develop a hybrid electric vehicle for the Shell Eco-marathon competition. By doing so, the project aims to realize a sustainable work process and technical platform on which future designs and implementations can be conceived and designed upon. In addition, it is the project and its team members intent that a better model for the car is developed. This model includes the design process itself as well as the simulation capabilities of the components, subsystems, and the system as a whole. This model structure is to be verified and validated with the current design and implementation in order to prove its effectiveness.

1.4 Scope and Delimitations

Both this paper and the project as a whole are defined by a number of limiting factors and choices. Both addresses the design and work process of conceiving, designing, and building, although not exclusively, HEV powertrains for ITRL, KTH, and Shell Eco-marathon. In this process tools used, design decisions made, and project philosophy are introduced and explained.

It is not only the process used for the project that is addressed, but also the implementation itself and the fully functioning prototype made. This prototype includes software, hardware, control design. In the software application, hardware and interface drivers are designed and deployed onto the Simulink platform chosen for the model based design.

However, it is not the intention nor purpose of the project to design an optimal solution in terms of global or local optimization. Instead the goal is to provide a robust, changeable, and modular system that enables for further optimization, globally and locally, being able to incorporate new technologies with little effort. The developed solution is therefore also to be operated in a safe fashion together with the tools that are developed together with the car itself.

The concrete areas that are included are system architecture and the possible modes of operation for a HEV, software tools and support including drivers, safe operation implementation of automatic driving mode control, human-machine interface, logging and monitoring of bus activity with communication interface, development environment for verification and validation.

2 Method

The methods worked with were chosen on the basis that they allowed for semi-independent development for the group for a period of time, with unit testing before synchronizing the progress with the rest of the team. The testing was done independently first then an incrementing number of subsystems were connected before implementing it in the car to do the final testing while driving. On each level of testing, the testing that could be carried out was exhausted before moving on and implementing more systems and introducing error sources, i.e. increasing the complexity.

2.1 Model-Based Design

The whole argument for having a model-based design approach to the project can be found in [2, 3, 4]. The main points being that it:

- simplifies inter-disciplinary development
- speeds up development
- simplifies debugging
- simplifies testing
- allows for code generation

All but one subsystems have the whole system in a Simulink model, the exception being the logging due to a lack of software support. The drivers are the only manually coded parts of the models so changing setup will take a matter minutes once the drivers for the new components have been completed. Together with the plant model different configurations can be tested and simulated before being realized physically.

2.2 V-model

To describe the work flow and the tasks, a V-model can be used, see Figure 1. The top level requirements are the requirements for the complete car and comes from the stakeholders at KTH and rules from the Shell Eco-marathon competition. The system requirements were created together with other groups in the KTH Eco Cars team. The requirements for all parts of the car includes the requirements for control system, the electric motors and the clutch actuation, which are also in the scope of this project. In this part the requirements on systems that other groups work on are also made. The system requirements are fulfilled by the subsystems and their requirements. To fulfill the subsystems requirements detailed design is made and then implemented.

Testing is done on different levels. When the implementation is done for a separate unit, e.g. a driver module, this unit is tested to fulfill the detailed design. This part is iterated and then integrated to a subsystem that is tested towards the subsystem requirements. In this way the work is iterated and parts are integrated until they are tested and working on the top level.

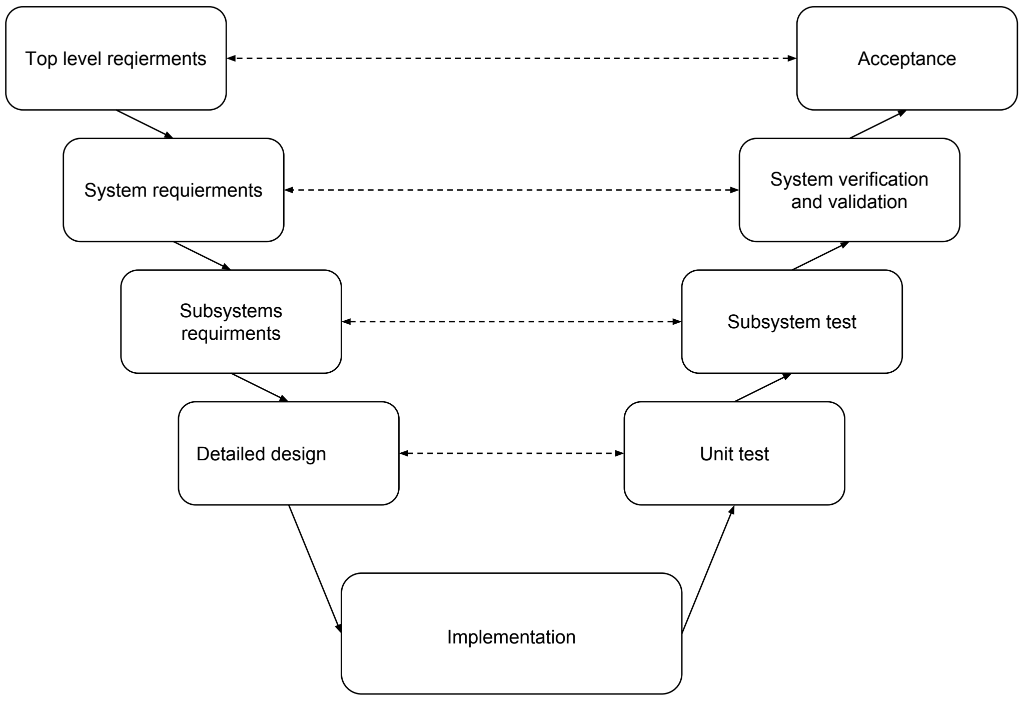


Figure 1: V-model

2.3 Testing Environments

To be able to work efficiently, mistakes and problems needs to be found as early as possible. It is generally easier to correct a problem found in a unit test rather than during an acceptance test. However, this is easy to say, but to find these problems, tools, and models and methods, to test the implementations early, are needed.

At the same time as the design of the car and the control system is done, these tools and methods are being developed to enable better testing, verification and validation. It is important that this work is not thrown away but instead utilized and improved by the next EcoCars team.

The different test environments that are being used are; test on the model in Simulink, test on the HW with Simulink, test on the HW using CANoe, test in the car in the garage and finally test in the car at a test track. They are all important to establish an efficient development process. All the different test environments are explained in more detail in the following sections.

2.3.1 Test On the Model In Simulink

When the implementation is done it is directly tested without using any hardware or microcontroller. This is done using models in Simulink. There exist a plant model of the car that can be used to test the behavior of the car or a subsystem. The plant model will in this case emulate the behavior caused by the new implementation.

2.3.2 Test on the HW with Simulink

To test on the real parts the programs are be deployed to an ECU hardware such as the Arduino Due. Simulink is then used to monitor and tune parameters. Subsystems or single parts, such as motors or electronic circuits, are tested at this stage using breadboard experimental boards or final circuit boards.

2.3.3 Test On the HW Using CANoe

When testing the subsystems with the real hardware using Simulink the CANoe software and hardware can be used to simulate other subsystems in the car. This is done with parts of the Simulink models from the plant model. The Simulink models can run from Simulink synchronized with the CANoe or compiled and executed directly from CANoe. The microcontroller or ECU that is going to be tested is then connected through a CAN bus to the CANoe hardware. All other ECUs are then simulated and the tested ECU or subsystem can be tested as if it is in the car while it was running.

2.3.4 Test In the Car In the Garage

Some things are tested on the car in the garage. This is much easier than actually driving the car. The driving wheel is placed on two rolling bars so the motors and the ICE can be tested, however the inertia of the car, air drag and many other things are not the same as when running the car on the road. When running the ICE it is important to make sure that exhaust gases are well ventilated. One way is to place the car close to a door and make a small opening for the exhaust pipe.

2.3.5 Test In the Car At Test Track

Test the car on the test track is very time consuming. One day of work for at least four people and preparations and planning is needed in advance. The test drives have up till now been done on Barkarby Airfield. Real test drives tests the car and the systems in real situations and are necessary to make sure that the requirements are fulfilled and are also necessary to validate models and subsystems. During real test external factors such as inclination and wind speed can not be controlled.

3 System Design

In this section the design of the system is explained. It contains both software and hardware design and architecture.

3.1 Control System Architecture

The control system architecture is shown in Figure 2. In this system there are four main ECUs; the Front ECU, Back ECU, ICE ECU and the Logging ECU. They communicate with each other using the main controller area network (CAN) bus that they are all connected to.

The front ECU is reading the driver requests from the steering wheel buttons, panel buttons, and tablet. It also displays information to the driver on the tablet and controls the headlights, wiper, horn and front indicators. Information is sent and received through CAN and the requested drive mode and reference speed are sent to the back ECU while current speed and powertrain status are instead received.

The back ECU is controlling the powertrain, all the motors, the clutches, as well as the rear lights and back indicators. Also the automatic drive mode control system is running on this ECU. The small motor is controlled using PWM through a galvanic isolation. The purpose of the galvanic isolation is to isolate the super capacitor from the auxiliary battery. Also the clutch control (digital I/O signals) goes through the galvanic isolation to the clutch control subsystem. A sub-CAN that uses CANOpen protocol is used to send commands and read information from the controller for the big motor. An encoder on the main shaft is used to read the rpm and calculate the velocity of the car. CAN messages are sent to the ICE ECU to control the operation of the ICE.

The ICE ECU reads the requests from the back ECU and controls the injection and enables the ignition. A lambda sensor is used to read and control the lambda value. Another encoder is used to read the position and speed of the ICE.

On the main CAN there is also a logging ECU that log all CAN traffic so in the case were some internal variable needs to be logged it has to be sent out with a CAN message. The logging ECU also has support to send information to an off-board server through GPRS. However this functionality is not implemented in the current revision.

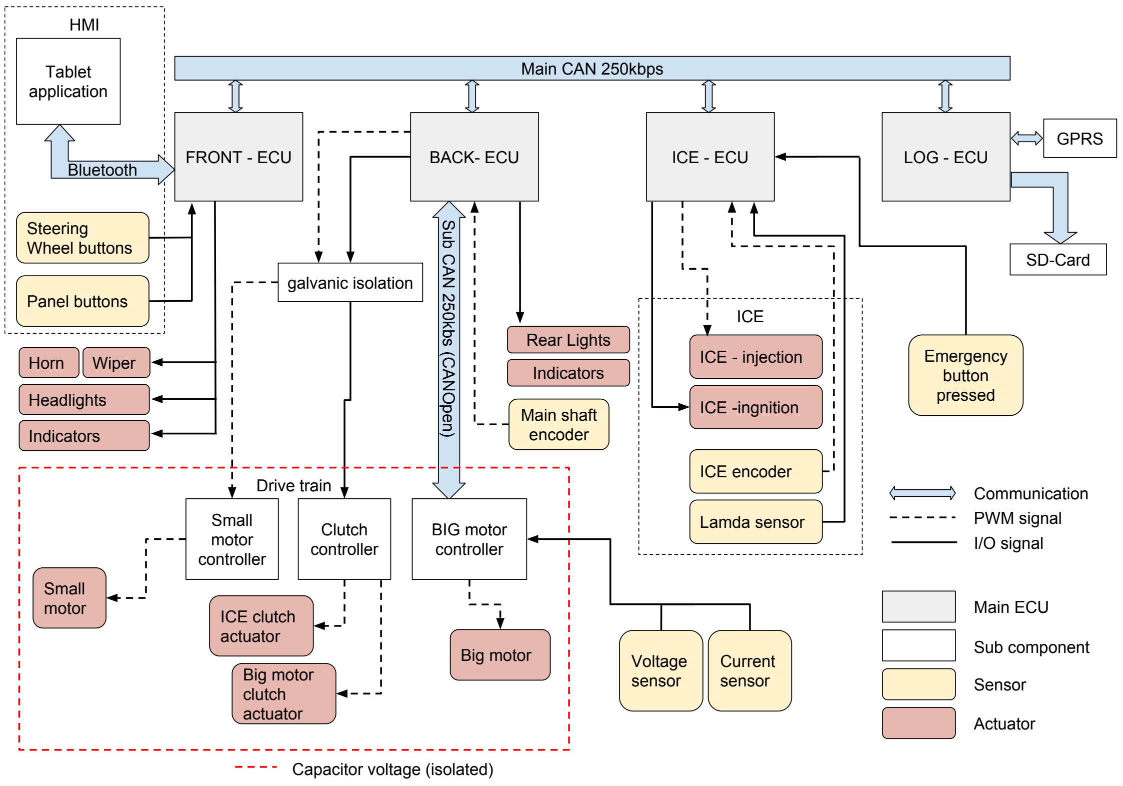


Figure 2: The control system architecture

3.2 Hardware

The following section explains the physical components that makes the car run.

3.2.1 Powertrain Architecture

The powertrain of Elba is similar to a complex hybrid system. The only difference is that the ICE and the electric motors don't have any direct connection between their output shafts, everything is connected through the transmission as seen in Figure 3.

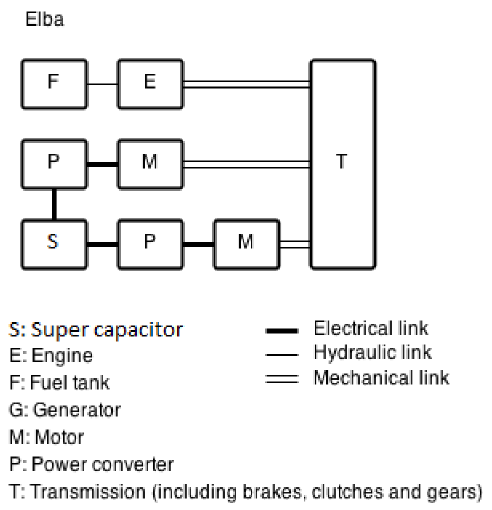


Figure 3: Powertrain of Elba

There are several reasons for choosing a complex configuration for Elba:

- At its maximum efficiency point, the ICE delivers significantly more power than what is needed to keep the vehicle at the desired constant speed.
- The electric motor used for regenerative braking must be able to handle most of the power delivered by the ICE, and does this at its maximum efficiency point.
- The electric motor used to propel the vehicle at the desired constant speed must do this at its maximum efficiency point.
- The electric motor which has its maximum efficiency point at the desired constant speed does not deliver enough power to accelerate the vehicle up to this speed. Therefore the motor used for regenerative braking can be used to accelerate the vehicle and as well as to kickstart the ICE.

The reason for choosing an ICE with higher power is because during early decision-making the group in charge of the ICE came with a proposal to increase the engine size and their reasoning was that it would double the power but still keep the same fuel consumption. From the presented reasons it is clear that two different electric motors, one bigger and one smaller, is necessary since the power needed for braking the ICE and the power needed to keep the vehicle at constant speed is not the same. Moreover, it's not possible to have one motor for both tasks if operation at maximum efficiency point is desired.

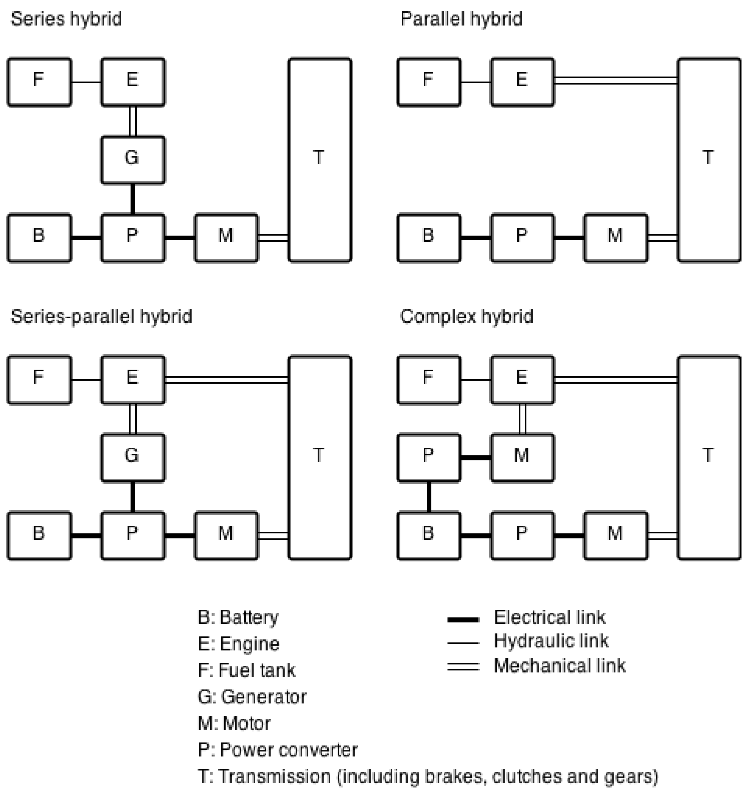


Figure 4: Classifications of hybrid electric vehicles. Figure influenced by [5]

The reasons why none of the common powertrain configurations, seen in Figure 4, are used in Elba is because:

- A series hybrid could work with a more powerful electric motor that could also accelerate the car. But the main reason why a series hybrid was never taken into consideration is because the task given from one of the stakeholders was to design parallel hybrid.
- A parallel hybrid would be inefficient because the big electric motor would be inefficient when propelling the car at constant speed. This was the original assignment but after deciding on the more powerful combustion engine, a new and more efficient powertrain was designed to handle the extra power. An efficiency model of the car with all the motors and the transmission was created. The outcome of the model showed that the current configuration was more efficient than a parallel configuration. However, it is important to clarify that the data from the past years is not reliable as the team mentioned in their report that they were skeptical about the results.
- A series-parallel would not work either because the small electric motor is not pow-

erful enough to accelerate the car. For this sole reason, a more powerful motor is needed. That is why in the current configuration the big electric motor is used to propel the car during acceleration and regenerative braking as well.

- A complex hybrid configuration fits the chosen ICE and two electric motors well. The drivetrain team, who are in charge of designing the transmission, presented a design similar to a complex hybrid configuration from the start of the project which suited the configuration after some design choices were improved. The final powertrain in Elba is closer to a complex configuration than a parallel.

The small electric motor will propel the car for the majority of the time, but its power output is too low to accelerate the car or use the regenerative braking to decelerate since it will take too long time and more losses are introduced due to friction. The acceleration and the start of ICE has to be done by the electric motor with higher power output instead. The reason why the ICE can not start by itself is because it does not have a starter motor, the big electric motor have this function instead. To avoid too many losses during acceleration and regenerative braking, the drivetrain of Elba is therefore designed so that each of the two electric motors and the ICE are separately connected to the transmission. The big electric motor and the ICE are attached to the mechanical clutches that are controlled by the linear actuators to engage and disengage. In other words, they can be controlled to connect to the drive shaft whereas the small electric motor is always connected. The main reason for this configuration, in comparison to the complex hybrid is because it is desired to add the freedom of choosing to accelerate using the big electric motor without having it connected to the ICE. Since the ICE does not start until the car have reached a certain speed, this means that the electric motor would have idling losses during early acceleration if the car had a complex hybrid configuration. The ICE experience cogging when it is not running which means that the losses would be huge. Therefore, the solution is to separate the ICE from the big electric motor, but this does not come without a drawback. The power transmission between the ICE and the electrical motor is through the clutch system, which means the losses will be higher and the regenerated energy is smaller than the preliminary calculations made. This leads to changes in the control strategy where the ICE have to be driven under longer periods of time during each drive cycle in order to have enough energy in the super capacitor.

3.2.2 ECU Platform

One of the important priorities of this project is that the control system that are designed and built are easy to modify and extend when the components or parts in the car are changed or if a control system for another car is needed. To make it easy to change the system, all main ECUs are based on a common ECU platform were some solutions for hardware and software already exists. In this way, most of the work made when designing and building the first ECUs can be reused when new ECUs are needed. When ECUs need new functionality the solutions should be added to the ECU platform to extend it for future use. For example, if new software drivers are made they should be added to the library for the ECU platform instead of just being a part of the software for that particular ECU.

For the competition the Arduino Mega board with the ATmega1280 microcontroller was used as a base and it worked satisfactory, but some things could be improved. The Arduino Mega has no CAN controller and no quadrature decoder counter built into the chip and this means that external hardware needs to be added.

After the competition there were more time over to look for alternatives to the Mega board. After some investigation and evaluation the decision was made to change to the Arduino Due board with the Atmel SAM3X8E ARM Cortex-M3 microcontroller. This board looks very similar to the Mega board (e.g. exactly the same size and number of pins), but it is very different in terms of internal components and features. The main benefits are that it has a built in CAN controller with two CAN channels and quadrature decoder counter for encoders. This makes the hardware less complex for the encoder and the CAN. The performance of CAN is also much better since it uses shared memory with the processor that have DMA (direct memory access) so the transfer of CAN data does not need the CPU. Other benefits with the Arduino Due board are; 32 bit processor, 84 MHz clock speed, larger and faster memory, larger flash memory for code. However, the Arduino Due is not without drawbacks, the pins on the Due board only sinks 9-16 mA, outputs maximum 3 mA and at 3.3 V compared to 5 V for the Mega board. This means that more hardware is needed when interfacing the microcontroller to other hardware. The ARM Cortex M3 microcontroller is also much more complex to program and a lot of drivers that are needed are not available and needs to be made from manually. The datasheet for SAM3X8E, [6], is 1450 pages compared to the 426 pages for the ATmega1280, [7]. ARM microcontrollers are also new to most of the mechatronics students at KTH compared to the AVR microcontrollers that are used in the embedded systems courses. All this together means that more time is needed to get things to work on the Arduino Due board compared to the old Arduino Mega board. In the end Arduino Due is the best choice to use as a base for the ECU platform and now when the basic functionality is in place it is a good board to build upon in the future.

After the competition more parts of the ECU has been included in the common ECU platform. Now all ECU are using the same base for the housing of electronics, connectors and circuits as well, see section 3.2.3.

3.2.3 Connectors and Housing

During the competition in Rotterdam, Shell Eco-marathon 2015, the opportunity to improve the result was lost partly because of bad connections in the connectors. The encoder wires were disconnected or glitched and due to the current design at that time there was no feasible way to solve the issue with time and tools at hand. The connectors used was MOLEX kk connectors. The missed opportunity to succeed with a second attempt lead to the result that the team for Elba fell short in comparison to other competing teams. Additionally wiring within the boxes themselves also posed a problem as some had to be resoldered or rerouted. One of the previous iteration of these boxes can be viewed in Figure 5. Given this result a great level of effort has been put into correcting these problems with a future proof solution.

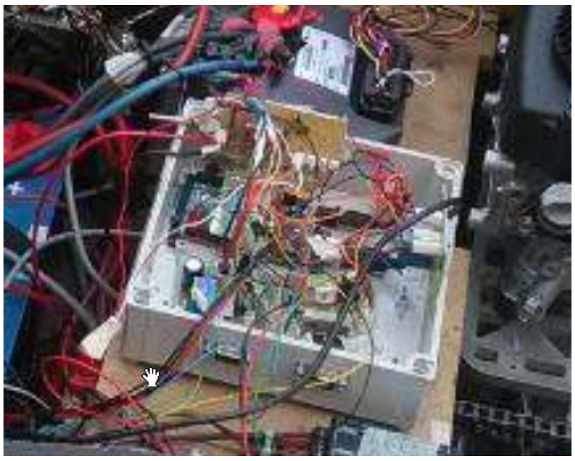


Figure 5: One of the ECU boxes during the competition

Connections between wires and boards are the weakest points in the ECU board design. Vibrations from the car causes these wire connections to glitch and in the end results in other unexpected and undefined results.

The solution chosen for the connector issue is to have all wires from and to the micro-controller go through one connector on the ECU board. In addition, all internal wires for the ECUs where exchanged for one printed board having the previous wire connections go through the printed circuit board (PCB) routs. The connector itself needs also to withstand vibrations, stress, and tension caused by the environment the board is located in. Therefore the connector needs to be soldered directly on the PCB, but also have a physical and mechanical attachment mechanism so that not all the forces inflicted on the connector is handled by the soldering. Since only one connector is used instead of many, as previous design utilized, the PCB design grow in complexity. As a result of this, the PCBs are instead ordered from a PCB manufacturer, Cogra Pro AB, to increase the level of robustness and reliability. One connector pair that fulfills these requirements is the TE Connectivity 776087-4 [illegible] Figure 6 and Figure 7.

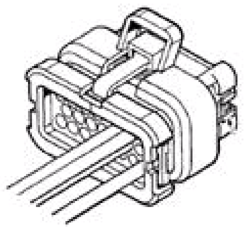


Figure 6: TE Connectivity 770680-4

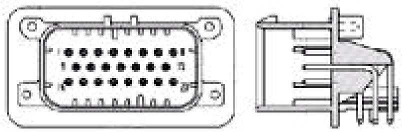


Figure 7: TE Connectivity 776087-4

Improving only on the connector is not enough to ensure safe and reliable operation, the housing of each ECU must also meet the same level of standard as to not fall short in this aspect. For this 3D printed boxes are used which allows for any dimension, shape, or customization as long as it is confined within the 3D printer volume capacity.

The first prototype utilized a slot design for inserting the box top cover. However, despite the ease of mounting the sealing cover, the 3D printer is unable to successfully print such a box due to the fact that the tight slot integrity is corrupted when new material is added on top of it, since the new material is pouring into the slit. This prototype can be studied in Figure 8.

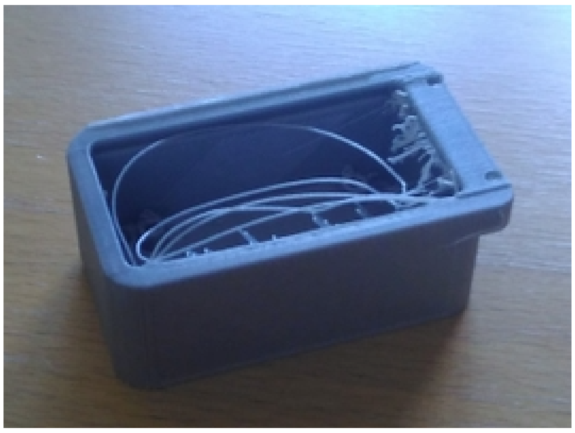


Figure 8: Fisrt prototype for the ECU boxes

The second iteration uses a more open design were the lid is placed from above on the box and then screwed in place by four screws. The PCB is placed inside this box on four cylindrical rests, all having a hole for screwing the PCB in place within the box. The connector is brought outside by an opening on one of the four walls on the box. By placing the PCB in such a way that there is sufficient clearance towards the side were the USB-connector for the Arduino Due is placed it is possible to connect to the Arduino USB connector while it is placed within the box.

During the printing of the boxes the material density does not have to be 100 percent. Instead, 30 percent has proven to be enough to retain integrity while at the same time shorten the printing process and decreased the amount of material used. Another factor is the printing speed. Despite its direct impact on the printing duration, printing speed does not significantly reduce the printing time. This is because faster printing speeds tends to corrupt the result generating an unusable box. Using this technique and design the second iteration of the ECU box is displayed in Figure 9 with a mounted PCB, connector, and the Arduino Due connected with a USB-cable.

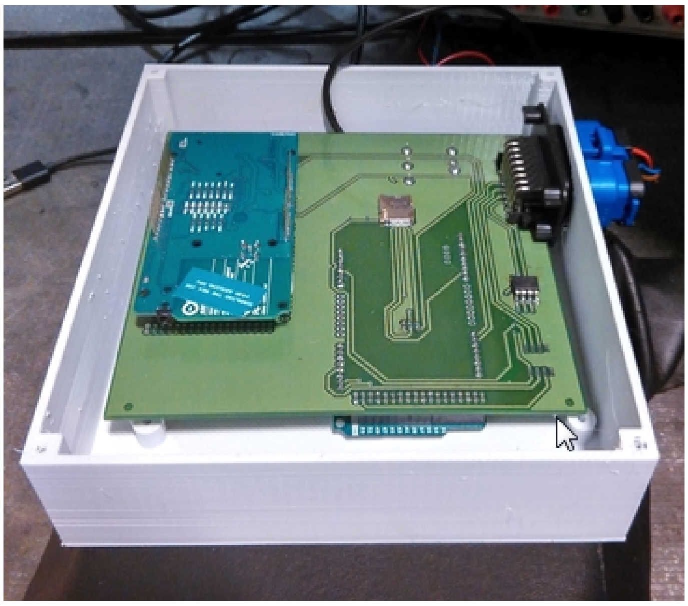


Figure 9: Logging ECU

3.3 Software Architecture

The software architecture for the ECUs is shown in Figure 10. On the top level is the Simulink model. Every ECU has its own Simulink project and a main model used to create and deploy the software to the microcontroller. This model is at the top of the abstraction level in Figure 10. In this way every ECU used in the Eco Cars project and in Elba that are Simulink models and the Eco Cars specific library will have the same architecture.

Most commonly, changes are made on the top level in the Simulink model. This is done through the graphical interface that is provided by Simulink. New blocks from the library can be added, dragged and dropped, and parameters can be changed. It is at this level the actual control of the HEV is performed. Here the developer has the possibility to develop an optimal solution for fuel efficiency, smart functionality, or other desirable implementations.

When a change in the software requires new functionality, that is not already provided by some of the existing blocks, a new driver block needs to be created. There are three type of drivers used: drivers supplied by the Simulink support package for Arduino, drivers made using Simulink s-function builder, and handwritten s-functions. This is shown in the center of the architectural diagram in Figure 10.

Some general functionality such as digital I/O, PWM, ADC, and basic serial communication are provided by the support package. It is however recommended to first check if the latest support package have the required functionality before making custom drivers in order to secure successful integration.

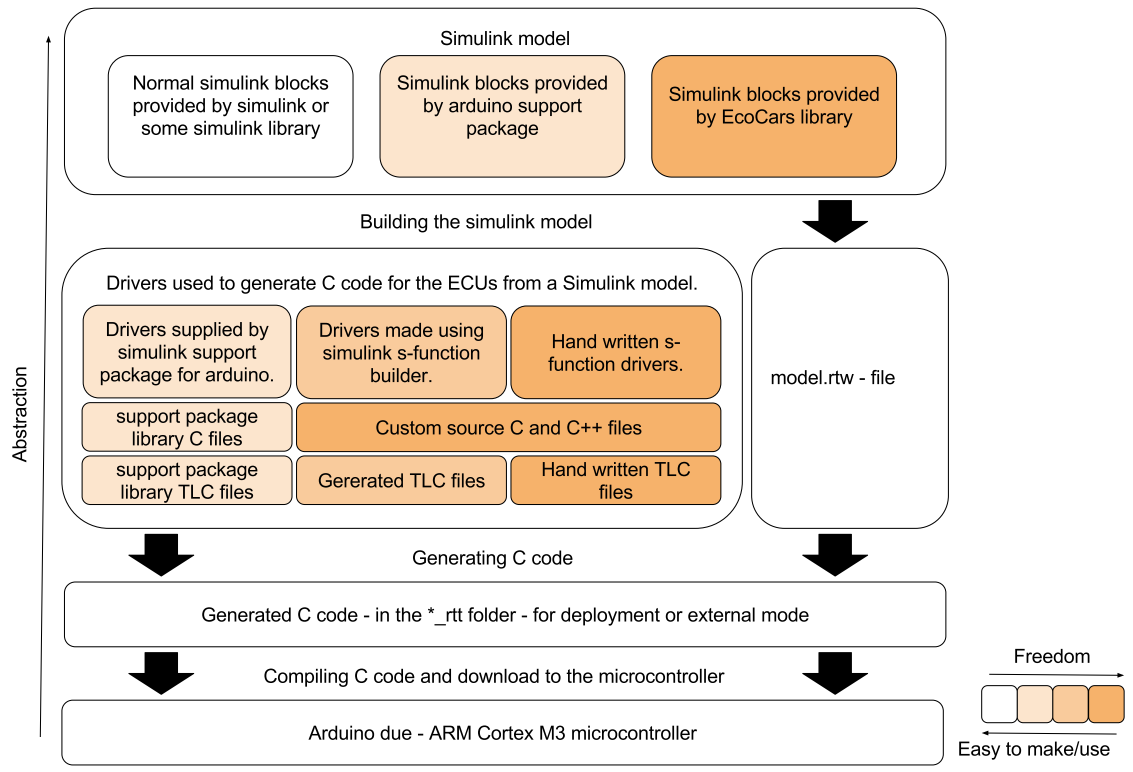

Simulink model
Normal simulink blocks provided by simulink or some simulink library
Simulink blocks provided by arduino support package
Simulink blocks provided by EcoCars library
Building the simulink model
Drivers used to generate C code for the ECUs from a Simulink model.
Drivers supplied by simulink support package for arduino.
Drivers made using simulink s-function builder.
Hand written s-function drivers.
support package library C files
Custom source C and C++ files
support package library TLC files
Gererated TLC files
Hand written TLC files
model.rtw - file
Generating C code
Generated C code - in the *_rtt folder - for deployment or external mode
Compiling C code and download to the microcontroller
Arduino due - ARM Cortex M3 microcontroller
Abstraction
Freedom
Easy to make/use

Figure 10: Simulink architecture and abstraction levels

Simulink has its own s-fuction builder block. Using this block is the best approach to start creating new drivers. When using the already provided s-function builder, .tlc file(s) and s-function (.c) files are generated by Simulink. It is possible to include custom made source code in C and C++ and Arduino libraries. However, caution has to be practiced to ensure that the new driver does not interfere with other drivers, e.g. if the same hardware counter/timer is used in two drivers, there will most surely emerge errors or bugs because of it.

In some cases handwritten s-function drivers are instead needed. These cases may include; use of custom interrupts, total control of code generation, sample time offsets during startup, special use of Simulink parameters, and control over where generated code should be positioned. When a handwritten s-function is made it is recommended to start with a copy of an already existing driver such as the CAN driver, encoder driver, or injection driver. These types of drivers are complex to make, but gives instead more freedom and possibility to customize the code. It is required to write both s-functions (.c) files and .tlc files. The s-function files are used to interface with Simulink and defines the Simulink block and number of input and output parameters etc. The .tlc files are used to control the code generation and governs how the parameters from Simulink should be used and how and where the C-code for the block will be generated. The .tlc file uses the model.rtw file that is created by Simulink during the build of the model. To understand the details MATLAB documentation and especially the Target Language Compiler manual [8] needs to be studied.

4 Theory and Simulation

This section presents what theories and equations that are used in order to create a mathematical representation of the powertrain and the vehicle.

4.1 Mathematical Representation

When designing the powertrain of a hybrid vehicle it is necessary to have a mathematical representation of the system in order to make good design choices. The main design choices for the powertrain of the vehicle includes first of all which type of hybrid vehicle configuration is desired. When this is established it is necessary to decide what motors to use in the system and what gearing is optimal for the application. These design choices are based on the dynamical properties of the whole vehicle, what driving conditions are present, and what driving properties are desired. To make these design choices without analyzing logged data from a running vehicle, it is necessary to run simulations of a system that represents the vehicle mathematically.

By designing a plant model which mathematically represents the dynamics of the vehicle, it is possible to investigate different design choices in order to find the most suitable. However, the process of creating a plant model can be an ample task of high complexity and requires a great amount of time. In this project the plant model has a moderate complexity since many mathematical simplifications have been made. The justification for this lies in that the simplified models have turned out to be of sufficient coverage for this application, whereas more complex mathematical models have turned out to be less useful than their simplified versions.

The plant model is not only usable when making mechanical design choices for the powertrain, but also when designing a control system. In this project the plant model has been of great value when designing the automatic control system for the vehicle. A few reasons for this being rapid testing of behaviors and properties emerged from the control design and highly controlled testing environment with determinable inputs to the overall system.

4.2 Plant Model

In order to design a plant model for the powertrain of the vehicle, it is necessary to use mathematical models of the different components. The order in which the plant model is designed can vary, in this project it follows a natural order that corresponds to the order in which design choices have been made throughout the project.

4.2.1 Vehicle Dynamics

The forces that acts on the vehicle and affects the dynamics during propulsion are the driving force, drag force, rolling resistance force and the gravitational force. The drag resistance is a type of friction that is present due to the air surrounding the vehicle. The mathematical model for air drag is

$$F_D = C_D v^2 \tag{1}$$

where F_D is the drag force, C_D is the coefficient of drag for the vehicle and v is the speed of the vehicle. The rolling resistance comes from the deformation of the tires when it rolls over a surface. The mathematical representation of the rolling resistance is

$$F_R = C_R N \tag{2}$$

where F_R is the rolling resistance force, C_R is the rolling resistance coefficient and N is the normal force on the tires. The force due to rolling resistance is implicitly dependent on the speed of the vehicle in the sense that it is only present once the vehicle starts to move, which needs to be taken into consideration when implementing the model in software.

The gravitational force affects the vehicle dynamics when there is a slope present, the mathematical model for the gravitational force is

$$F_G = mg\sin(\alpha) \tag{3}$$

Where F_G is component of the gravitational force that acts in the direction of propulsion, m is the total mass of the vehicle, g is the gravitational acceleration and α is the slope angle.

The driving force of the vehicle is dependant on the transmission of the powertrain and the torque output from the motors. In this project the driving force is modelled as

$$F_{in} = \frac{1}{r_{wheel}} R_p (T_{motors\_total} - T_{loss}(v)) \tag{4}$$

where F_{in} is the driving force, r_{wheel} is the radius of the tire, R_p is the gear ratio of the planetary gear, T_{motors_total} is the total motor torque after corresponding gears and motors, and $T_{loss}(v)$ is the torque loss present in the transmission, which is modelled as a function of the vehicle speed v. Since the viscous friction of the two motors are very small compared to the losses in the transmission, they can be neglected in the motor models and are instead included as an average viscous friction in $T_{loss}(v)$.

The function $T_{loss}(v)$ is based purely on measurements and is derived from the effect loss model shown in Figure 11

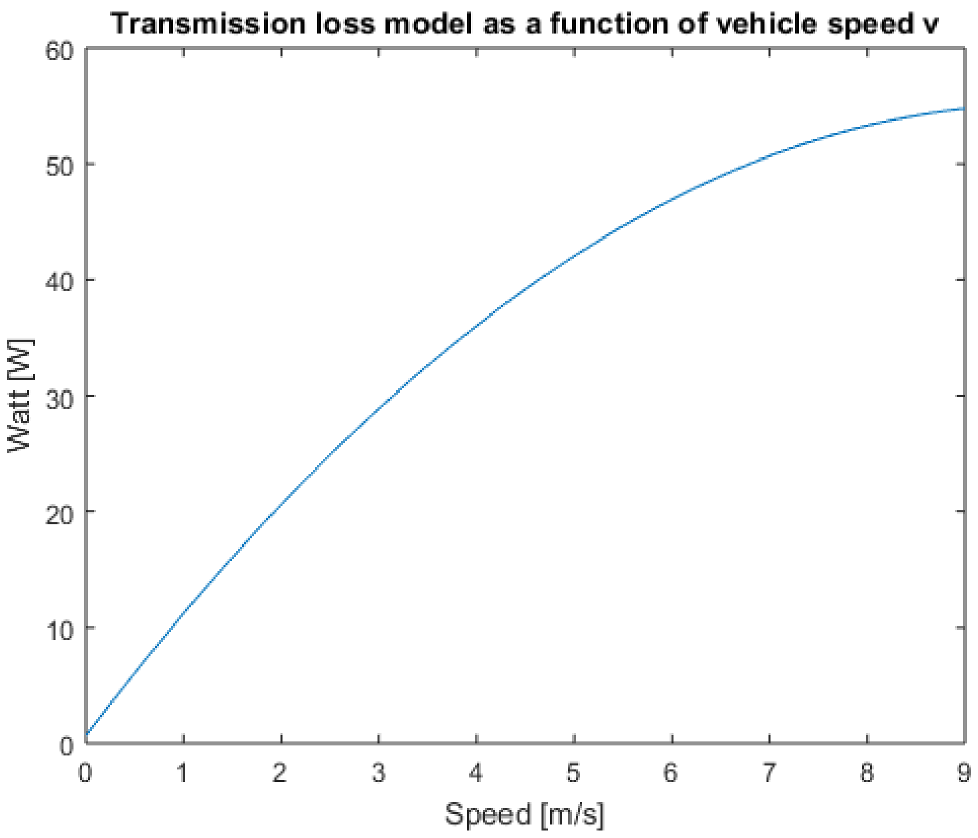


Figure 11: Model for effect losses in transmission

The vehicle dynamics are derived from Newton's second law by using Eq. (1) to Eq. (4). The result is a differential equation of the vehicle's speed v

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1}{m}(F_{in} - F_D - F_R - F_G) \tag{5}$$

where $\frac{\mathrm{d}v}{\mathrm{d}t}$ is the time derivative of the velocity of the vehicle, i.e the acceleration of the vehicle.

There is a simplification in Eq. (5) where the inertia of the wheels and the motor shafts are neglected. The reasoning for this is that they do not affect the dynamics significantly since the inertia due to the mass of the vehicle being much bigger.

The Simulink model for the vehicle dynamics is shown in Figure 12

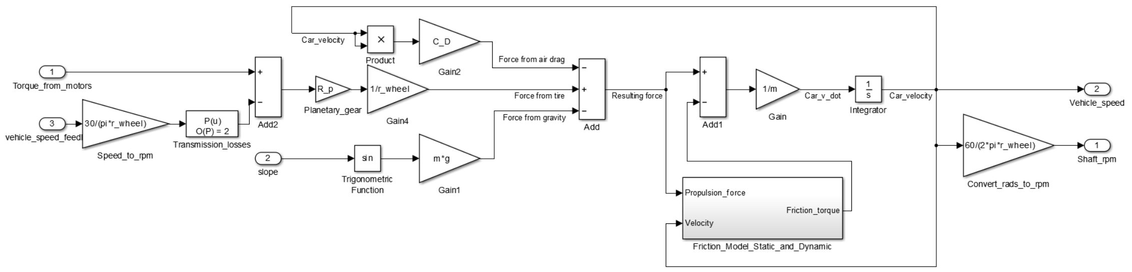


Figure 12: Simulink model for vehicle dynamics

4.2.2 Supercapacitor

In this project, the electrical power in the powertrain is stored in a supercapacitor. The energy stored in a supercapacitor is

$$W = \frac{C_{CAP}}{2}U^2 \tag{6}$$

where W is the energy stored in the supercapacitor, U is the voltage across the supercapacitor and C_{CAP} is the capacitance of the supercapacitor.

Electrical power is defined as

$$P = UI \tag{7}$$

where P is electric power and I is the current.

Integrating electical power results in electrical energy

$$W = \int UIdt \tag{8}$$

Combining equation Eq. (6), Eq. (7) and Eq. (8) yields the implicit differential equation

$$U = \sqrt{(\frac{2}{C_{CAP}}\int UIdt)} \tag{9}$$

The Simulink model for the supercapacitor is shown in Figure 13

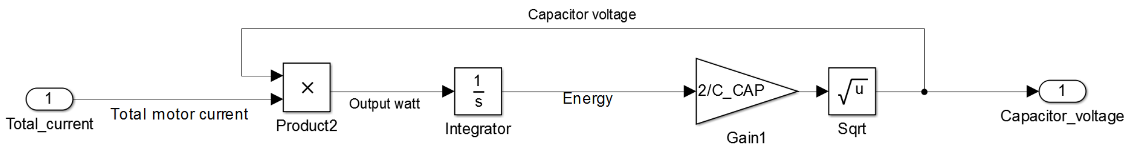


Figure 13: Simulink model for the supercapacitor

4.2.3 Internal Combustion Engine

Modelling of an internal combustion engine can be very difficult and complex. In this project the engine model is simple and is derived from the effect curve provided by the manufacturer of the engine.

The torque curve for the motor is shown in Figure 14

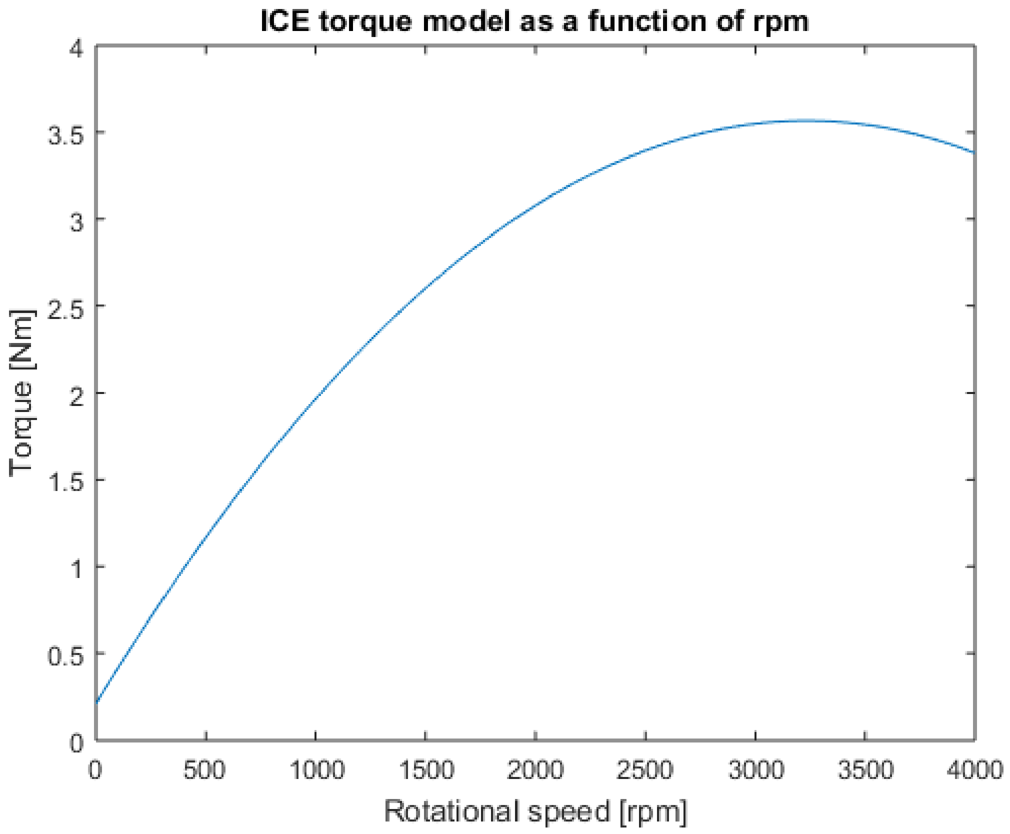


Figure 14: Torque model for the internal combustion engine

In the application of this project, the engine operates at an almost constant speed at all times, where the lambda sensor does not vary the injection amount significantly. Therefor a simplification has been made in the model where the amount of fuel injected, i.e. the fuel consumption, per revolution is constant. A correction factor has been added to the output torque, where the value of the factor is based on measurements. The Simulink model of the engine is shown in Figure 15.

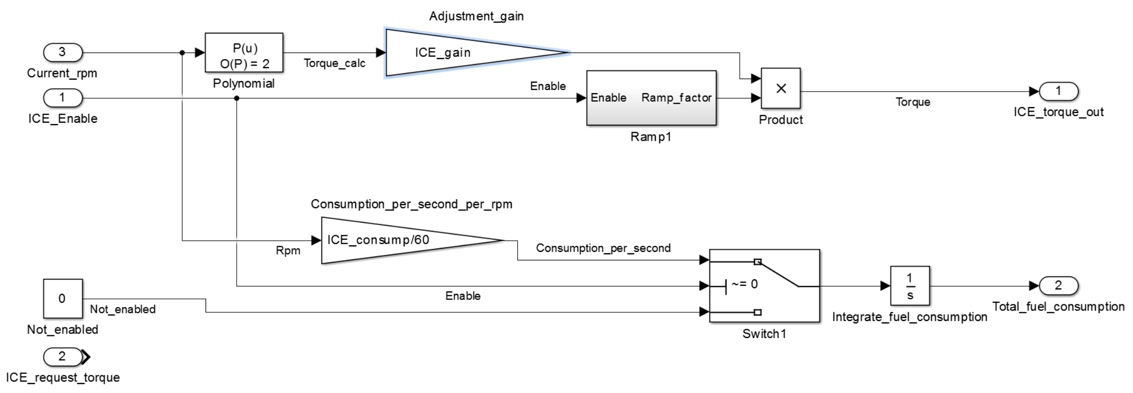


Figure 15: Simulink model for the internal combustion engine

4.2.4 DC Motor

A DC motor model is presented in Figure 16

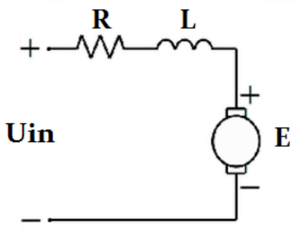


Figure 16: Circuit model for DC motor

The mathematical model of a DC motor is derived by applying Kirchoff's voltage law to the circuit in Figure 16. The equation becomes

$$U_{in} = Ri + L\frac{\mathrm{d}i}{\mathrm{d}t} + E \tag{10}$$

where U_{in} is the terminal voltage of the motor, R is the winding resistance of the motor, L is the winding inductance of the motor, i is the motor current. E is the induced back-emf due to the motor winding and is modelled as

$$E = K_{emf}\omega \tag{11}$$

where K_{emf} is the back-emf constant of the motor and ω is the angular speed of the motor.

The output torque of the motor is

$$T_m = K_{emf} i \tag{12}$$

where T_m is the motor torque.

An overview of the Simulink model for the small DC motor is shown in Figure 17.

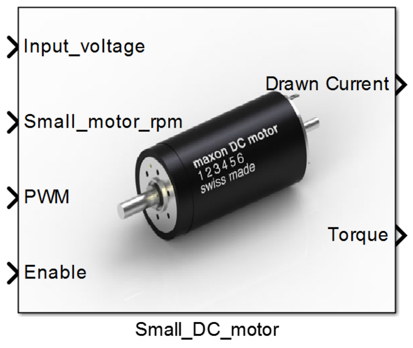


Figure 17: Overview of Simulink model for the small DC motor

4.2.5 DC motor driver

The small DC motor is driven by a motor driver from Maxon. Modelling a motor driver can be very complex if done thoroughly. In this project the motor driver does not act much more than as a PID controller combined with a current limiter for the small DC motor. The current limiting model is based on the motor model and is shown in Figure 18.

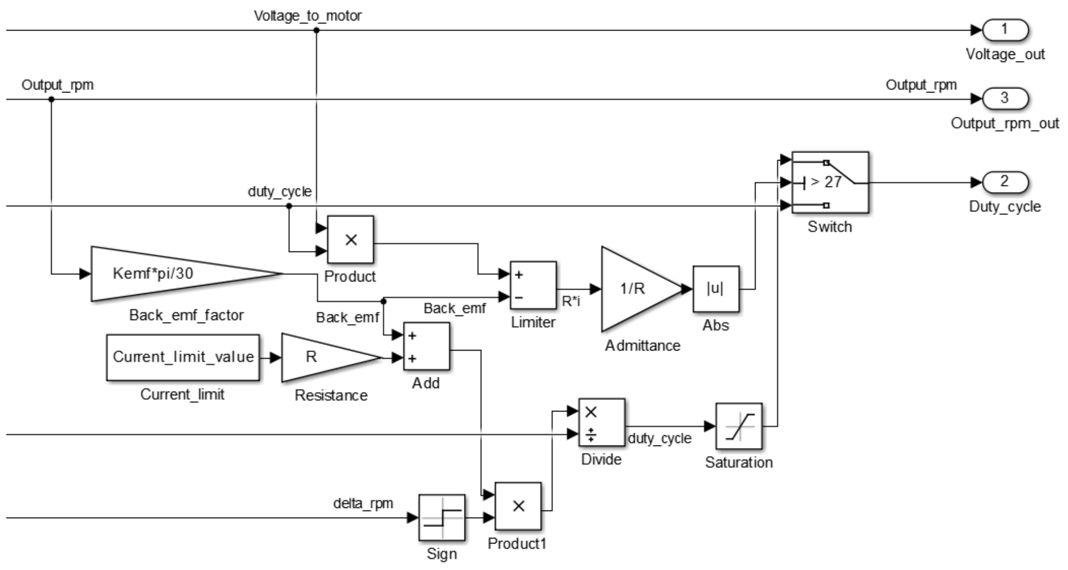


Figure 18: Model of current limiter in Simulink

However, there are some losses present in the motor driver which is corrected through an efficiency factor which is obtained from the data sheet of the driver.

The PID controller is represented by the PID component block in Simulink. The parameters of the PID controller are tuned to give satisfactory control behavior.

The Simulink model for the small DC motor driver is shown in Figure 19.

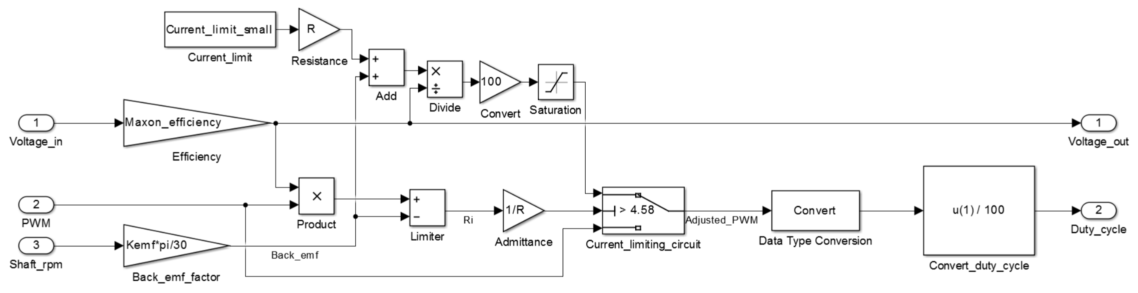


Figure 19: Simulink model for the motor driver for the small DC motor

4.2.6 BLDC motor

The big electric motor used in this project is a BLDC motor, which means that it is driven differently from a regular DC motor. However, it is possible to make a constant current approximation of a BLDC motor, which means that it is basically approximated as a regular DC motor, however it needs to be induction compensated [9]. Figure 20 shows the simplified motor model

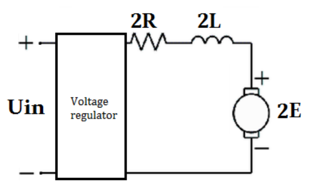


Figure 20: Circuit model for approximated BLDC motor

Using Kirchoff's voltage law on the system above results in

$$U_{in} = U_{regulator} + 2Ri + 2L\frac{\mathrm{d}i}{\mathrm{d}t} + 2E \tag{13}$$

where $U_{regulator}$ is the voltage drop over the motor controller and E is the induced back-emf due to the motor winding and is presented in Eq. (11). The motor torque T_m is still described by Eq. (12).

An advanced BLDC motor model have been made in this project as well, it is however not being used in the plant model due to difficulties in identifying all the unknown parameters. The theory behind the advanced BLDC model is not presented in this report.

An overview of the Simulink model for the BLDC motor is shown in Figure 21.

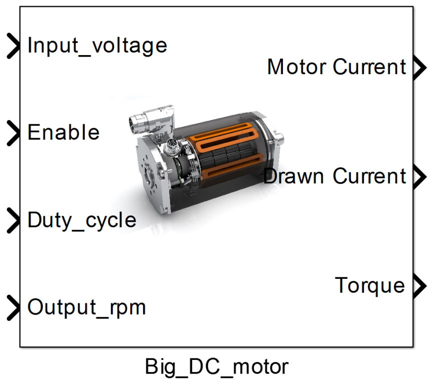


Figure 21: Overview of Simulink model for the BLDC motor

4.2.7 BLDC motor driver

The BLDC motor is driven by a motor driver from Inmotion. The mathematical model for this component can be made very complex. Since the interest in this plant model does not lie in what happens within the driver but how the dynamics of the entire system behaves, the model can be simplified.

The model of the driver is basically a cascaded PI controller. The inner loop consists of a PI controller for motor current and the outer one consists of a PI controller for speed. The controller has two control options, speed control and torque control. Controlling the speed is done by the cascaded PI controller. When torque control is desired, the cascaded PI controller becomes only a PI controller for current, i.e. torque. The driver also includes a current limiter which clamps the current to the maximum allowed current of the motor. The model for the current limiter is the same one as for the small motor driver and can be seen in Figure 18.

Similar to the DC motor driver, the model of the BLDC motor driver uses an efficiency factor to correct for the losses that are due to the power electronics within the controller.

An overview of the Simulink model for the BLDC motor driver is shown in Figure 22.

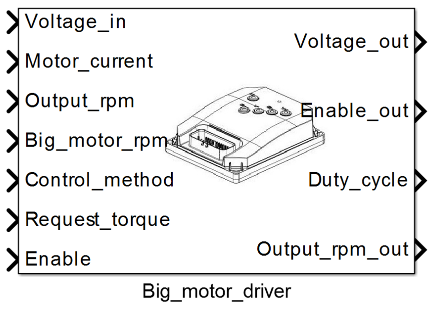


Figure 22: Overview of Simulink model for the BLDC motor driver

4.2.8 The Clutch

The clutch system is driven by two linear actuators which are controlled by two H-bridge circuits. Since the clutch is moving only a fraction of the total time, the model for the clutch used in the plant model is very simple. Another reasoning for the simplification, is that the mechanical parts of the clutch are too weak for the linear actuator which makes the system behave non-linear.

The model for the clutch system is included in previous models in two ways. Firstly, the attaching and detaching of the clutch is included in the motor model as a type of enable input, the necessity of this lies in that the clutch is moving rather slowly which means that it affects the movement of the vehicle due to timing constraints. Secondly the losses due to the clutch are included in the vehicle dynamics model, where it is part of the transmission loss model.

5 Implementation

Since most of the component and equipment hardware used in this kind of projects are not very common to be used by a lot of people, there is no built-in software support for them. This means a number of drivers had to be developed for the systems to manipulate the hardware as intended, but also to extend the functionality of the car. All drivers, except for the logging, are created as Simulink project and thus follows the model-based design approach. To make the final product of the project more future proof, standard solutions like CAN was chosen to be implemented for all subsystems. This allows for better control, easier manipulation, and more design freedom of the system. For instance, the individual subsystem in the ECUs can be made aware of the overall status of the car and fewer customized solutions are needed. Furthermore, explanations will be provided regarding how the different ECUs work individually and which system they collaborates with.

5.1 Powertrain Control Unit (Back ECU)

The Powertrain Control Unit, also called the Back ECU, is responsible for controlling the powertrain, all the motors, the clutch system and the Internal Combustion Engine (ICE). The Back ECU is receiving the driver requests from the Front ECU through the CAN bus on channel 0. The control system in the Back ECU constantly reads all the sensors in the powertrain in order to control the powertrain to achieve the driver requests. The ECU is built upon the common ECU platform, which means hardware-wise the usage of an Arduino Due board, a 3D printed box, and an AMPSEAL connector, but also includes a Simulink model with code generation functionality. All of these parts are described in section 3.2.3 Connectors and Housing and this chapter will only bring up the specific things for the Back ECU.

5.1.1 Software

The software is made in Simulink and uses the CAN driver, Encoder driver and other parts from the common library. The model is divided in one control part and one interface part, see Figure 23. In the control system part are everything that takes the input signals and calculate the control signals, e.g. PID controllers, driving mode control, and the automatic mode selection algorithm. The interface part is the interface to the real signals, responsible for reading the CAN messages, sending requests and actuation signals and reading and filtering sensor signals. Because the control system part is clearly decoupled from the interface to the real hardware, it is easy to exchange the interface to use the plant model instead. The control system consists of three main things; one PID controller for the small motor, a stateflow chart for the automatic driving mode selection, and a stateflow chart for control of the driving modes (how to change between modes). More information about the automatic driving mode and driving mode control could be found in section 5.6 Automatic Driving Mode.

Most of the parts in the HW interface part of the Back ECU model, see Figure 24, is easiest to understand by the model itself. Some blocks are needed to run background tasks and services, these are; sending the "BackECU alive message" on the CAN bus informing other ECUs and the diagnostics tools (CANoe) that the ECU is online and working, setting up CAN channel 0-1 and send commands and read sensors on the InMotion controller through the CANopen protocol on CAN channel 1. The most complex part is the CANopen communication and this uses the stateflow chart seen in Appendix F, more information about CANopen can be seen in [10]. All actuation signals and CAN messages that are sent out from the Back ECU are gathered together on the left in Figure 23. They include controlling the clutches through output pins, sending ICE control signals through CAN, enable and control the current in the small motor through one output pin and a PWM signal, control rpm and enable the Big motor through CANopen on channel 1, and sending back information to the Front ECU through the main CAN on channel 0. The signals received by the Back ECU are; signals from the front ECU through a CAN message, rotation of the main shaft through a quadrature encoder, and signals from the Big motor through CANopen. To get more information about the Back ECU Simulink project, see the video tutorials.

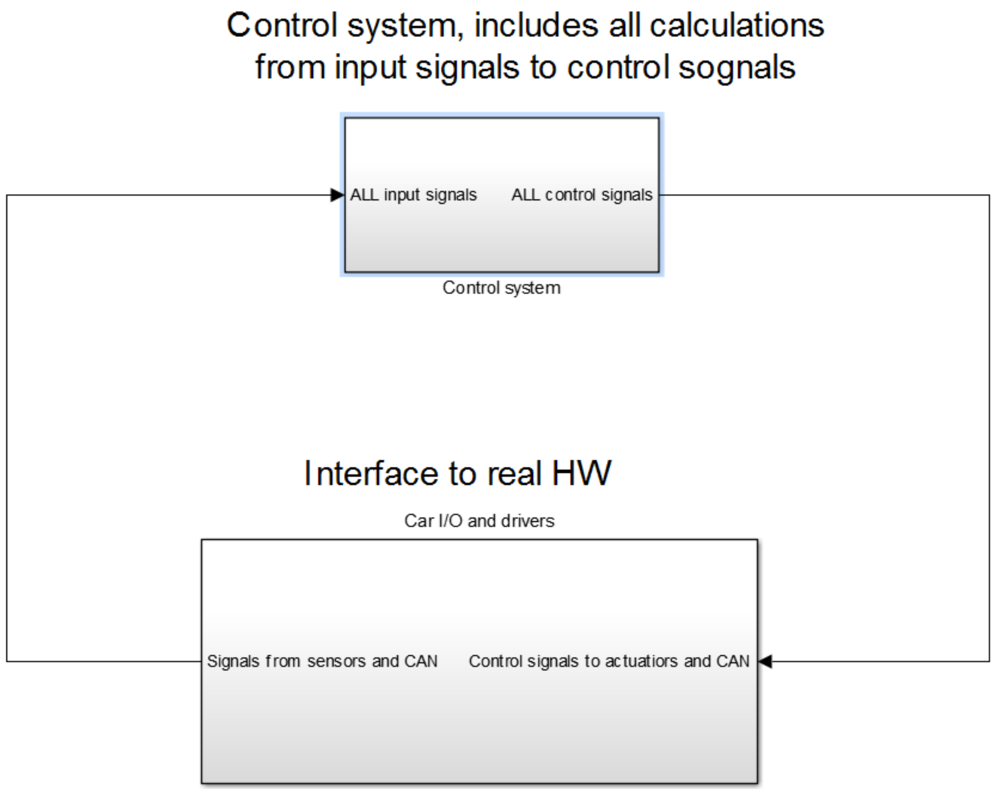

Control system, includes all calculations from input signals to control sognals
ALL input signals
ALL control signals
Control system
Interface to real HW
Car I/O and drivers
Signals from sensors and CAN
Control signals to actuatiors and CAN

Figure 23: Back ECU Simulink project

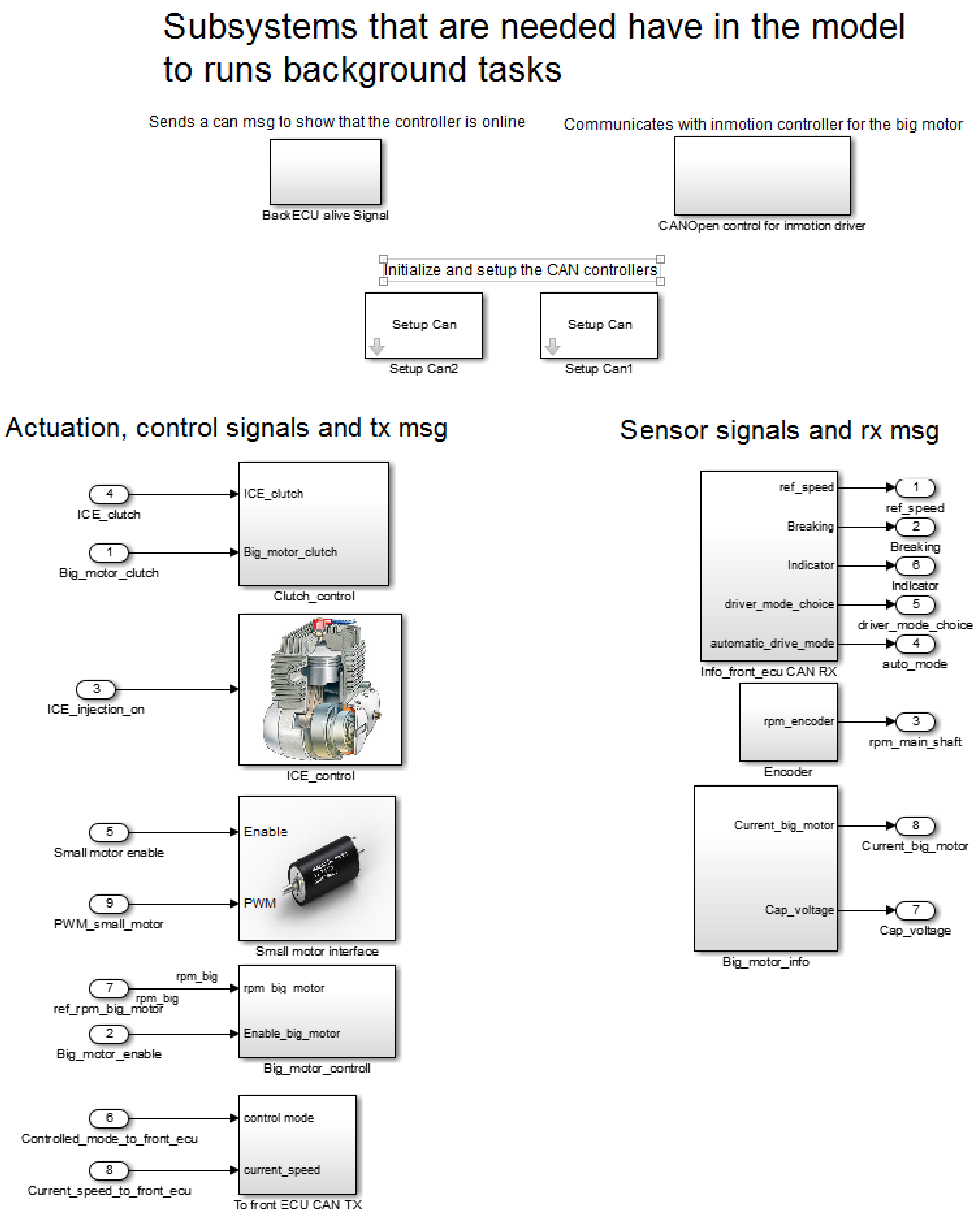


Figure 24: Back ECU Simulink project, real HW interface section

5.1.2 Hardware

As all ECUs using the common ECU platform, the Back ECU has a 3D printed box, a ampseal connector and a printed circuit board. The circuit board can be seen in Figure 25, the components used are the same as the one used in other ECUs e. g. the ICE ECU. In the middle is a place to put the Arduino Due board and it is connected to nine transistors, one logic level converter and two CAN transceivers. Details about these components can be found in section 5.2 Internal Combustion Engine Control Unit about the ICE ECU. Eight of the transistors are connected to create open drain outputs to the ampseal connector. This means that when the corresponding pins on the Arduino Due (5, 6, 7, 8, 10, 22, 24, 25) is high, the open drain output will be connected to ground. These outputs can be used for driving high current and support PWM. One transistor (pin 30) is prepared to connect a LED or similar. This can be used for example to indicate errors. However since the diagnose through CAN worked fine this output is not used at the moment. The logic level converter is used for the encoder and the two CAN transceivers are needed for CAN 0 and CAN 1. To see a complete pin map of the connector for the Back ECU see Appendix D.

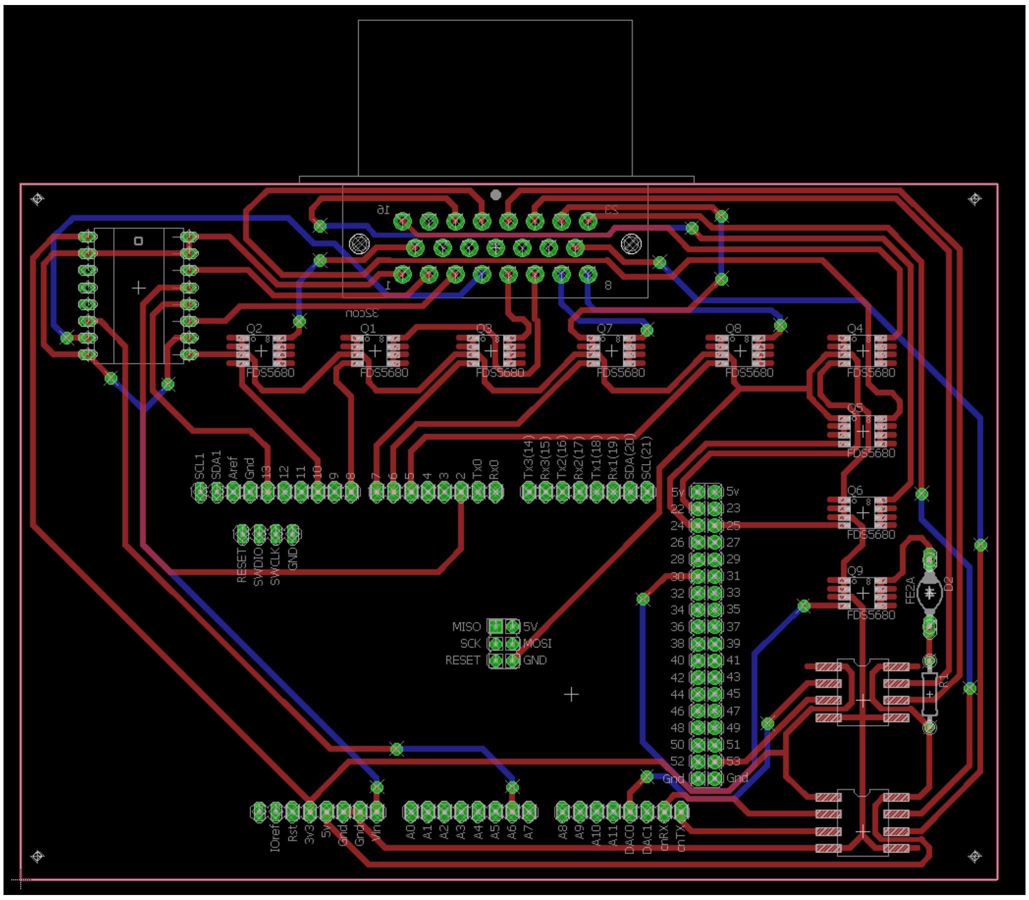


Figure 25: The circuit board for the Bask ECU.

5.2 Internal Combustion Engine Control Unit

The ICE ECU developed by the internal combustion team for the competition in Rotterdam was not very robust and had some software flaws that could be improved. A new ICE ECU was developed in order to eliminate these issues and make it more convenient to use and further improve in the future. The microcontroller used in the improved ICE ECU is an Arduino Due which increases the computational power of the ECU compared to the old ECU. The new controller operates at 3.3 V and since the transmission voltage in the car is 12 V, a voltage shifter is implemented to shift the voltage in between. The voltage shifter is of type “TEXAS INSTRUMENTS CD40109BPW” and can convert in both directions. N-channel transistors of type “STMICROELECTRONICS STS6NF20V MOSFET transistor” were used on the PCB for signals. A CAN bus transceiver “TEXAS INSTRUMENTS ISO1050DUBR CAN Bus” is used to convert normal Can bus signal to can bus signal that is suitable for Arduino Due. A diode is used to indicate the calibration of the lambda sensor. When the ECU is turned on, the diode starts blinking and the calibration is ready when the diode stops blinking. The normal shutdown and emergency shutdown are series connected. An NC relay is utilized. The engine will shut down when emergency switch is pressed or shutdown signal is received.

The ECU is directly compiled from a MATLAB Simulink project. The Simulink project is seen below in Figure 26

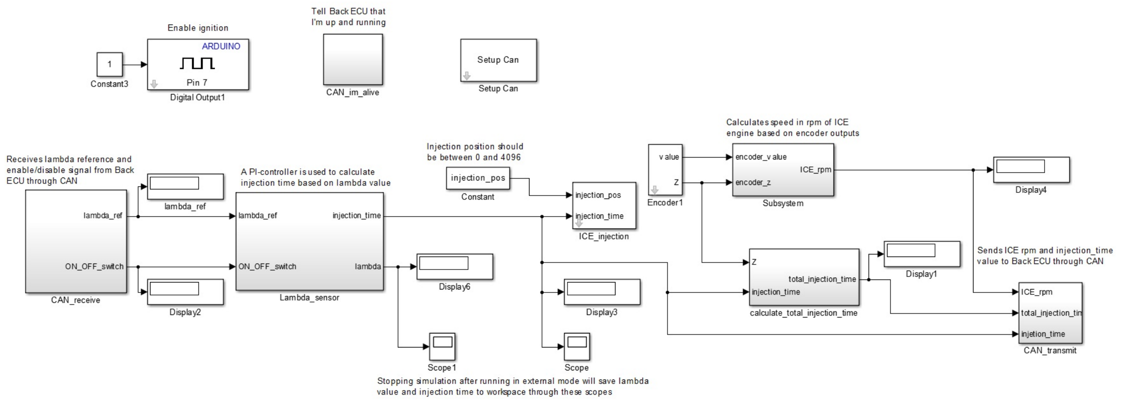

Enable ignition
ARDUINO
Pin 7
1
Constant3
Digital Output1
Tell Back ECU that I'm up and running
CAN_im_alive
Setup Can
Setup Can
Receives lambda reference and enable/disable signal from Back ECU through CAN
lambda_ref
ON_OFF_switch
CAN_receive
lambda_ref
Display2
A PI-controller is used to calculate injection time based on lambda value
lambda_ref
ON_OFF_switch
injection_time
lambda
Lambda_sensor
Display6
Scope1
Injection position should be between 0 and 4096
injection_pos
Constant
injection_pos
injection_time
ICE_injection
Display3
Scope
Stopping simulation after running in external mode will save lambda value and injection time to workspace through these scopes
value
Z
Encoder1
Calculates speed in rpm of ICE engine based on encoder outputs
encoder_value
encoder_z
ICE_rpm
Subsystem
Display4
Z
injection_time
total_injection_time
calculate_total_injection_time
Display1
Sends ICE rpm and injection_time value to Back ECU through CAN
ICE_rpm
total_injection_tim
injetion_time
CAN_transmit

Figure 26: ICE ECU Simulink project

The main function of the Simulink project is the control algorithm that controls the ICE. A lambda sensor attached to the ICE is read by the program and compared to a lambda reference value sent from the back ECU through CAN. A discrete PI controller is then used to calculate how long the injection time should be in milliseconds. The PI controller has a lower saturation limit of 0 ms and an upper limit of 10 ms, the upper limit was decided through a simple calculation; if the car runs at 27 km/h, the ICE rotates at 3370 rpm which means one revolution is 17.7 ms. Therefore 10 ms would mean the injection is about 56 % of a revolution. The calculated injection time is then sent to the ICE_injection S-function block described in section 5.3.3 Injection. The control of the ICE can be seen in Figure 27.

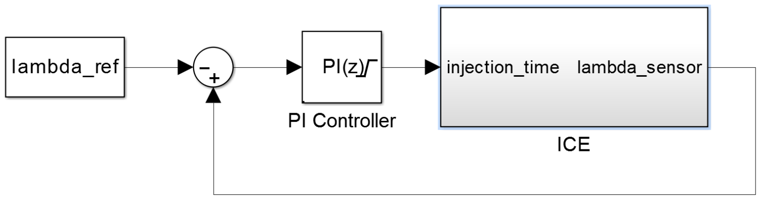


Figure 27: ICE control

The P and I value for the PI controller is 0.5 and 15 respectively. The PI controller was tuned while running the ICE ECU in external mode, different P values were tried to see how the lambda sensor and injection time changed. The ICE started running on P value 1 and then successively increased up to 4 before trying a last value of 0.5. The test was also saved to MATLAB workspace and plotted.

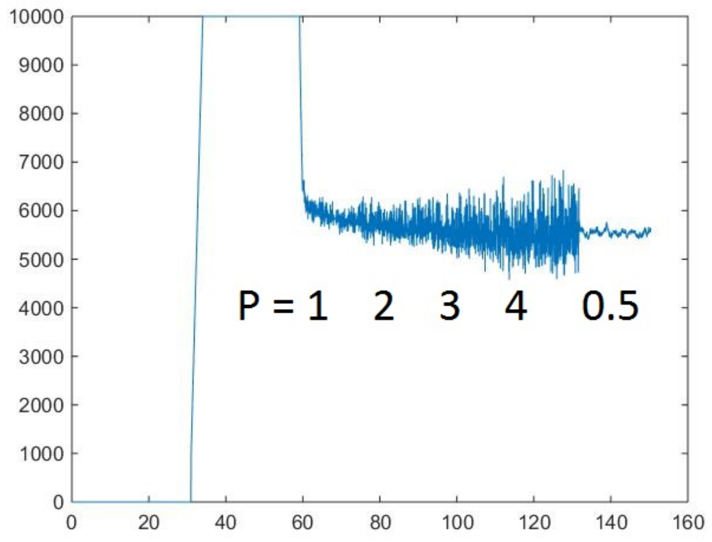


Figure 28: Recorded injection time

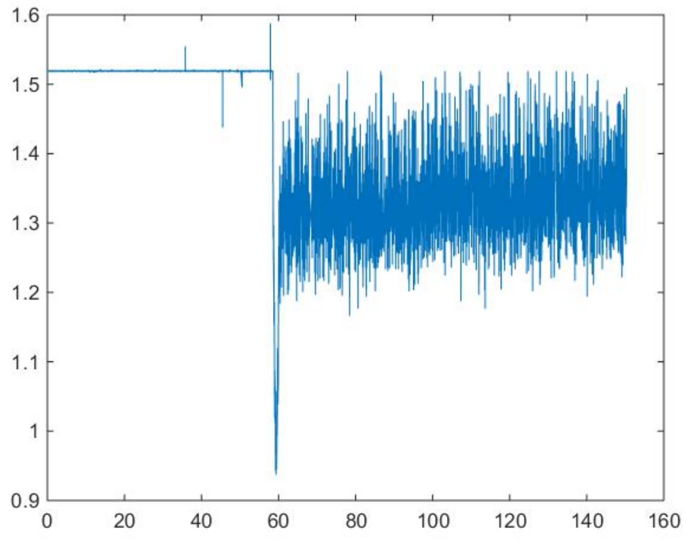


Figure 29: Recorded lambda value from lambda sensor

As seen in Figure 28 and Figure 29, the value from the lambda sensor is between 1.2 and 1.5 no matter what the injection time to the ICE is and the lowest P value was chosen for a stable injection. Figure 28 shows the injection in microseconds.

The ICE ECU utilizes the encoder Z S-function described in section 5.3.2 Encoder for determining the injection position. The position for the injection is the same as the one used in the previous version of the ICE ECU. Other functions of the ICE ECU include sending the ICE rpm, injection time, total injection time and a signal that says it is online to the back ECU. The ECU also receives an ON/OFF signal from the back ECU, the control system of the ICE is temporarily shut off when the OFF signal is received to ensure that no windup occurs.

5.3 Software Drivers

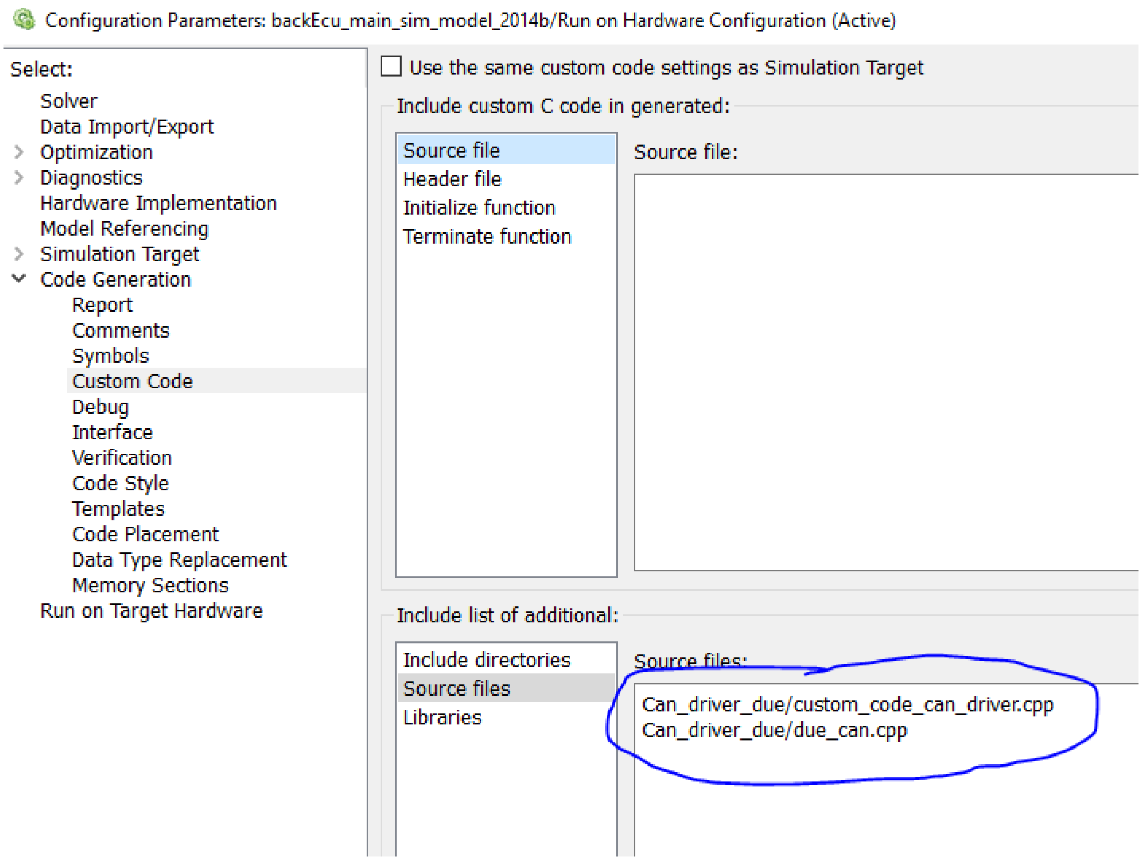
This section describes the software drivers developed in the project. Most of the software in the project is in the form of Simulink models that are directly compiled to Arduinos. This insured the software was easy to debug and change. The Simulink support package for Arduino included some simple digital and analog read and write, but not advanced functions such as CAN and interrupts. These functions were first written in C/C++ code and then implemented in Simulink S-functions that was easy to drag and drop into Simulink projects.

5.3.1 CAN

To enable all ECUs to communicate on the CAN network a software driver for the Arduino Due and the cortex M3 processor is needed. No supported driver from Arduino Due exist but a home made driver from github [11] is used as a starting point. Then parts of this code is used to create a driver for Simulink. Custom C++ code is used and therefore two files from the CAN driver folder needs to be added to the Simulink models custom code, "due_can.cpp" and "custom_code_can_driver.cpp", see Figure 30.

The logic for the CAN driver is shown in Figure 31. For every CAN RX block that is added to the Simulink model the corresponding CAN id is placed in a buffer. This is done when the CAN driver are initialized during power up of the ECU. When a CAN message is received the "CAN msg received interrupt" routine is executed and the message id is compared with the global buffer. If a buffer position has the corresponding CAN id the data of the message is written to this position. If the CAN id is not in the buffer the interrupt routine will loop through the complete buffer. With a small number of messages and a small buffer this is not a problem but if the buffer gets too big the interrupt can take too long time to execute. This can be solved using different mailboxes and message masks to make different callbacks for different groups of CAN messages. However this far the number of messages is small enough (10-30) for the general callback solution to work.

When a message is sampled and read by the Simulink model the CAN driver gets the latest data from the global buffer. Note that sampling the data faster does not help if the CAN message interval is long, the CAN driver will then return the same data until a new CAN message arrives.


Configuration Parameters: backEcu_main_sim_model_2014b/Run on Hardware Configuration (Active)

Select: Code Generation > Custom Code

- [ ] Use the same custom code settings as Simulation Target

Include list of additional: Source files

Source files:

```
Can_driver_due/custom_code_can_driver.cpp
Can_driver_due/due_can.cpp
```

Figure 30: CAN driver, add custom code

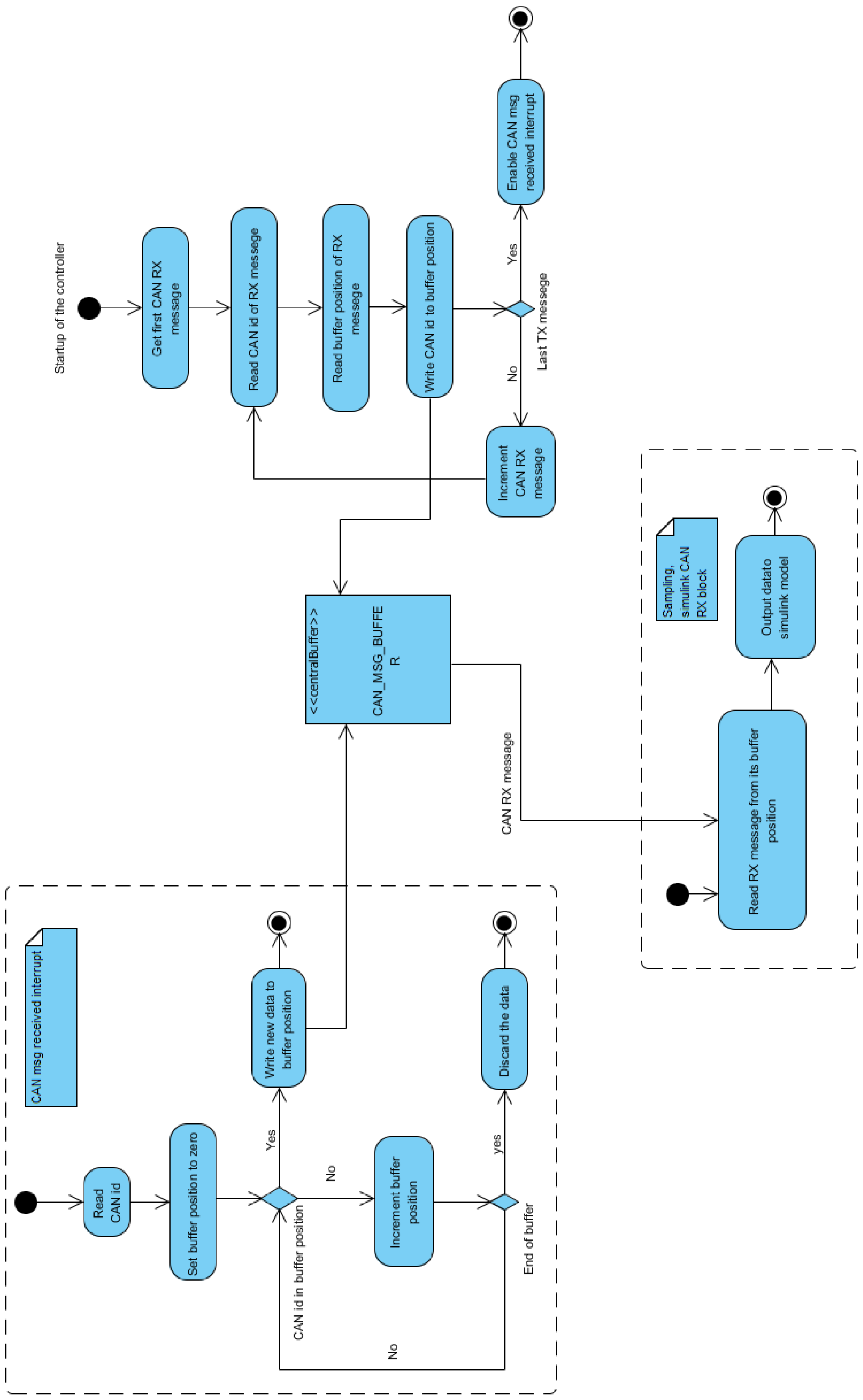
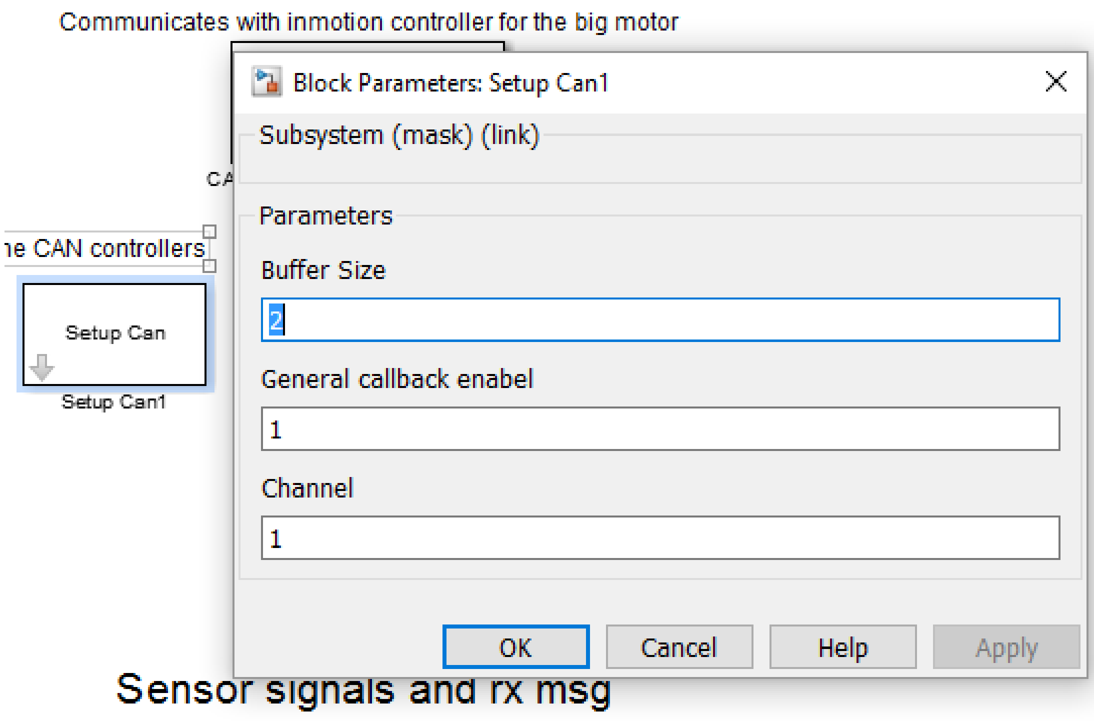

Startup of the controller
Get first CAN RX message
Read CAN id of RX messege
Read buffer position of RX messege
Write CAN id to buffer position
Yes
No
Last TX messege
Enable CAN msg received interrupt
Increment CAN RX message
<<centralBuffer>>
CAN_MSG_BUFFER
CAN RX message
Sampling, simulink CAN RX block
Read RX message from its buffer position
Output datato simulink model
CAN msg received interrupt
Read CAN id
Set buffer position to zero
CAN id in buffer position
Yes
No
Write new data to buffer position
Increment buffer position
End of buffer
yes
No
Discard the data

Figure 31: CAN driver logic

The simulink CAN driver contains three Simulink blocks, Setup Can, CanRx and CanTx.

The Setup Can block initializes the CAN interface and setups the CAN message interrupt, for each CAN channel used one “Setup Can” block is needed. Figure 32 shows the parameters dialog for the “Setup Can” block. The “buffer size” parameter sets the size of the global buffer used for this CAN channel, easiest is to set the same number as numbers of CANids that will be received by the CAN channel. “General callback enable” needs to be 1 if the interrupt routine is going to be called every time a CAN message arrives. To set the CAN channel use the “Channel parameter”, CAN 0 and CAN 1 is available.


Communicates with inmotion controller for the big motor
ıe CAN controllers
Setup Can
Setup Can1
Sensor signals and rx msg
Block Parameters: Setup Can1
Subsystem (mask) (link)
Parameters
OK
Cancel
Help
Apply

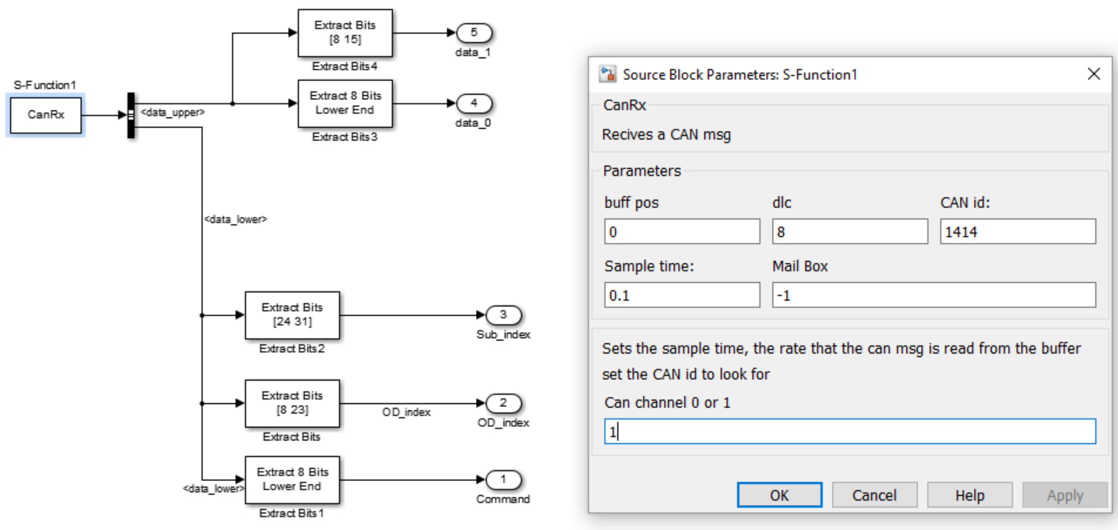
Buffer Size: 2

General callback enabel: 1

Channel: 1

Figure 32: CAN driver, CAN setup block

Figure 33 shows the “CanRx” block and the parameters dialog. The “buff pos” parameter sets where in the global buffer the message will be stored, “dlc” is the the data length of the message (8 bytes is full message) and the “Mail Box” parameter can be used to set a separate mail box and callback. If “Mail Box” set to -1 the general callback will be used, this is the recommended value. “CAN id” is the CAN id to read, sample time is the sample time in which the block will read the buffer (-1 can be used for inherited sample time) and “Can channel” set which CAN channel the message is read from. The output from the “CanRx” block is a “can msg bus” (Simulink user defined bus) defined as two 32 bit integer, data lower and data upper as well as a integer for the CAN message id. Using the Simulink blocks “Bus Selector” and “Extract Bits”, the values of the signals inside the CAN message can be extracted from the lower and upper data integers.



CanRx

Recives a CAN msg

Parameters

buff pos	dlc	CAN id:
0	8	1414

Sample time:	Mail Box
0.1	-1

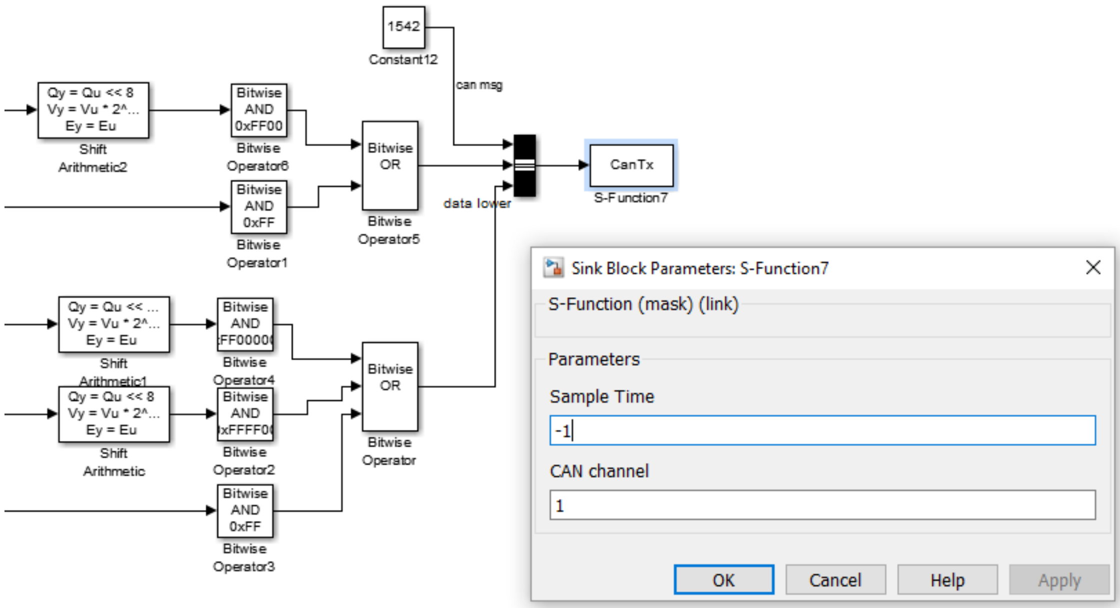
Sets the sample time, the rate that the can msg is read from the buffer
set the CAN id to look for

Can channel 0 or 1

1

Figure 33: CAN driver, CanRx block

The “CanTx” block shown in Figure 34 sends messages out to the CAN bus. There are only two parameters, the “Sample Time” is the rate at which the messages will be sent (-1 can be used for inherited sample time) and the “CAN channel” is simply the channel the messages will be sent on. One input signal is going to the “CanTx” block, that is a the same type, “can msg bus”, as the output of the CanRx block. To send a proper message “shift”,”and” and “or” simulink blocks can be used to put the signals in the right place of the data in the message. To get more information about the CAN drivers, see the video tutorials.



Sink Block Parameters: S-Function7

S-Function (mask) (link)

Parameters

Sample Time

-1

CAN channel

1

Figure 34: CAN driver, CanTx block

5.3.2 Encoder

There are two S-functions created in Simulink for reading encoders, one with continuous counting and one that interrupts at Z and resets the encoder value. In both S-functions it is possible to set the sampling time of the output signal by clicking on the Simulink block. The S-function outputs an encoder value four times the resolution of the encoder used (the encoder used has a resolution of 1024, which means that every revolution, the encoder value will increase by $1024 \cdot 4$).

The encoder with continuous counting, outputs an unsigned 32 bit value and will therefore overflow when it reaches $2^{32} - 1$. The value of the encoder when the program starts is 0 and if it's turned backwards it will also reach $2^{32} - 1$ and count down instead. Given that the car will run at a continuous speed of 27 km/h and that the encoder is attached on the rear axis that rotates at an average of 2750 rpm, the total time the car can run before the encoder overflow is about 6.4 hours. The encoder Z Simulink block interrupts at Z and clears the encoder value.

5.3.3 Injection

To improve the injection system for the ICE ECU Simulink model, an interrupt based model was needed. An S-function was designed to perform the task. The ICE_injection S-function only works together with the encoder Z described in section 5.3.2. The ICE-_injection S-function can be seen in Figure 35.

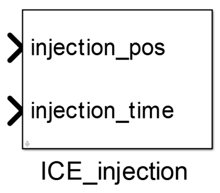


Figure 35: Injection S-function

The two inputs are injection position and injection time. The injection position is dependent on the encoder value. The encoder Z in the car reset every time it passes Z, which in the car means the encoder value will reset every time it reaches 4096. The injection position is therefore a value between 0 and 4096. An interrupt occurs when the injection position is reached, the process is described in Figure 36.

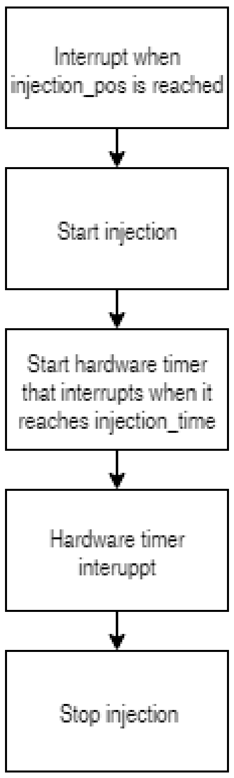

Interrupt when injection_pos is reached
Start injection
Start hardware timer that interrupts when it reaches injection_time
Hardware timer interuppt
Stop injection

Figure 36: How the injection S-function works

There is an option to choose sample time by clicking on the ICE_injection S-function block. This sample time is only used for choosing the injection position and injection time used for the interrupts, and does not improve the performance of the injection. The injection time parameter for the S-function should be in microseconds.

5.4 Logging

The key requirements put on the logging was that it should be fast and have enough storing capacity to be able to store a day worth of data when testing and driving.

The hardware choice fell on using the same microcontroller hardware as the rest of the car for several reasons. Arduino boards have support for peripheral equipment like SD card and GSM communication which allows for design freedom, but it also have built-in support for two CAN buses. By using the same microcontroller as the rest of the car allows for the team to be familiar with fewer systems, and hence different people can work on different parts of the system more easily. This decreases the need for having areas of expertise in the group, a greater number of people can manage a greater number of tasks. Another side benefit is that when travelling it is convenient to have the same microcontroller for the whole car in case something breaks and needs to be replaced.

Even though Arduinos have support for saving data to SD cards, the drawback of using a low level system like Arduino is saving the data and organizing it at the same time is harder because it has to be done in C and simpler file formats. The rate at which data can be logged depends on the speed of the SD-card and the frequency of the processor.

Using SD-cards for logging gives a few benefits compared to storing it on internal flash memory; if a memory card is filled, it is easy to replace it and with another within a minute without the need of connecting a computer, it is also a cheap solution.

An initial hope was that Simulink and a graphical model-based design approach could be used for the logging as for the rest of the system. The combination of hardware proved to be unsupported as something locked up in Simulink when saving data to the SD-card was tested. But as the code is basic, the code code was written by hand in the Arduino IDE.

One drawback of using Arduinos for logging is that there are not a lot of support for making the saved data easy to sort and save, and with a constant stream of incoming data saving the data has to be done with a good margin before the next message comes to make sure that no data is missed.

The GSM module is in place and can be easily implemented if the right software libraries are released for the Arduino Due.

There is a server at this point in time there was not ready support for the online logging, but as it was not a priority a decision was made not to put any more time on it as other matters were more pressing.

A Matlab script was created for managing the data. The script is dynamic in the sense that when the CAN signals database are changed in the Vector program, the script is automatically updated by parsing the database file created by the program, where all the CAN signals are identified and organized. The focus of this implementation was on enabling the user to sort the data conveniently and able to handle changes in the CAN signal structure without the need of changing any code. When the data has been sorted all the user has to do to view the data is to call a plotter function with the desired CAN-signal and database as argument.

5.5 Human Machine Interface

This chapter will include all the parts that build up the Human Machine Interface (HMI). The parts that constitute the HMI are the Front ECU (The main controller), the android application (driver interface), sensors, actuators and additional circuit boards for the actuators.

5.5.1 The Front ECU

This section will describe the Front ECU concerning the hardware and the software. Starting with the software, all the software of the Front ECU was done in Simulink and is therefore, model-based. The reason for this is that model-based iterations are quicker and the software allows for changes in an easier way as mentioned in earlier chapters. This was critical for the Front ECU design as changes happened constantly under the development and test periods. The Simulink project, as can be seen in Figure 37 below, is made up of three parts:

The first part, highlighted in blue in the figure, consists of reading all the input signals that come from three sources: The CAN bus, the steering wheel and panel, and the android application. These messages are either read internally through the serial port in the Arduino Due, or through the CAN driver or through the digital input pins of the Arduino Due, where the steering wheel buttons, brake and Dead Man switches and the panel buttons are connected to.

The serial port of the Arduino Due deals with the messages that are sent both ways from and to the android application. The signals that the android application sends to the Front ECU are listed in a table in Appendix A. These messages are 1 byte long and has a unique number. Every unique signifies a specific action to be done. For example, receiving the following message ”01100000” means that the driver of the car requires to change the driving mode from manual mode to automatic mode, meaning that the back

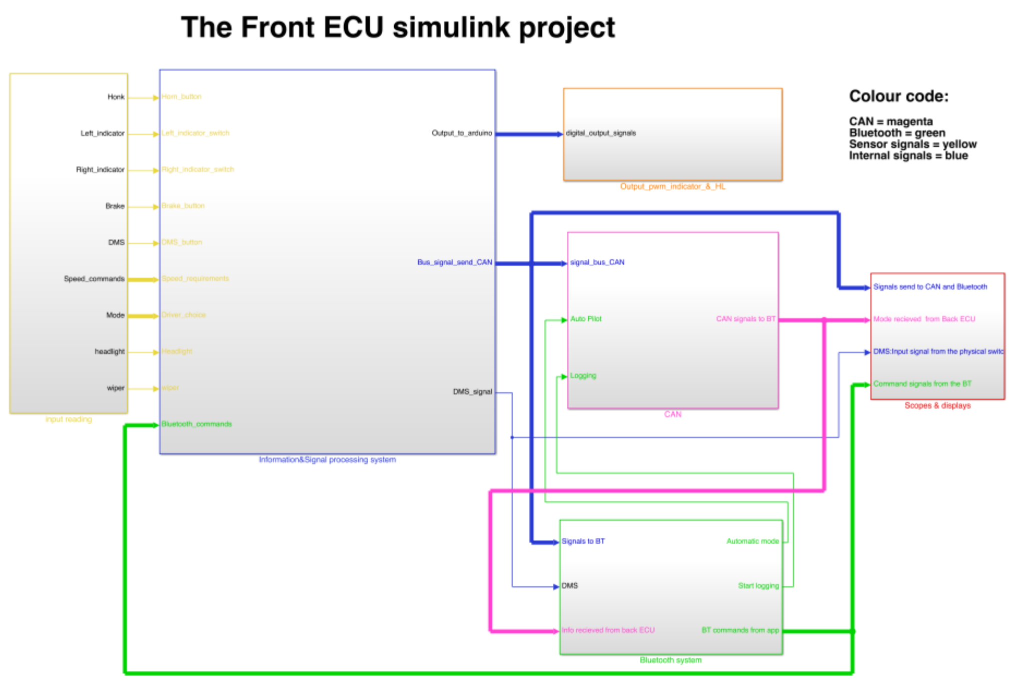


Figure 37: Simulink architecture for Front ECU

ECU is in charge of the changing between driving stages depending on the voltage level and so on. This is mainly to reduce the load of work that the driver has to do while driving. Moreover the Front ECU sends a great deal of messages, which are also listed in a table in appendix B. Notice, however the structure of the code in the table. These messages are mostly feedback messages so that the driver can be reassured that the commands are being received and activated. These messages are printed on the screen of the android application.

Furthermore, the signal interpretation is done in the Simulink project. The Simulink model reads the information sent through the Bluetooth and analyses it. Accordingly to the information that is sent to the Front ECU, it does some operations or sends the data to the back ECU to get processed there. Instruction for using the project and how the front ECU processes the signals will be covered by the video tutorials and only a quick overview of the system will be presented here besides detailed information about the signals.

The second part of the project is the CAN drivers, which are the same as the ones mentioned in previous chapters. The CAN driver, as can be seen in Figure 38, extracts three messages from the CAN bus and sends also 7 messages to the CAN bus.

The third part is the Bluetooth; where a Bluetooth library was built for this project in order to send and receive data. The Bluetooth library is found under the main Simulink project for the Front ECU and has to be dragged into the simulink model to use it. The input and output signals are easily changed in the library. Furthermore, the Bluetooth

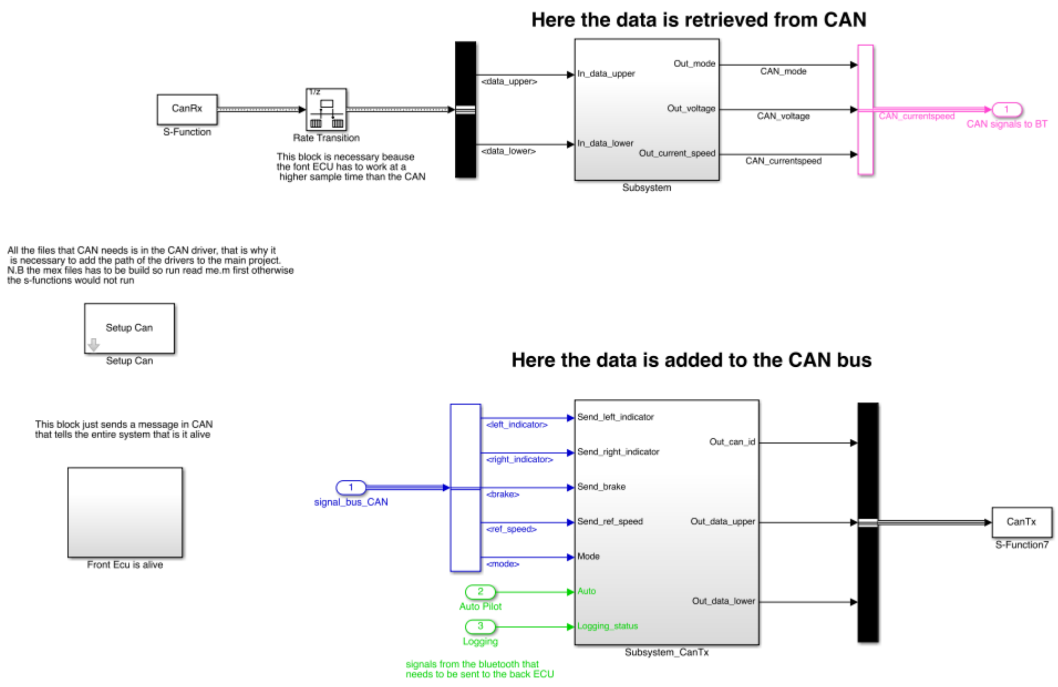

Here the data is retrieved from CAN
CanRx
S-Function
1/z
Rate Transition
This block is necessary beause the font ECU has to work at a higher sample time than the CAN
<data_upper>
<data_lower>
In_data_upper
In_data_lower
Out_mode
Out_voltage
Out_current_speed
Subsystem
CAN_mode
CAN_voltage
CAN_currentspeed
CAN_currentspeed
1
CAN signals to BT
All the files that CAN needs is in the CAN driver, that is why it is necessary to add the path of the drivers to the main project. N.B the mex files has to be build so run read me.m first otherwise the s-functions would not run
Setup Can
Setup Can
Here the data is added to the CAN bus
This block just sends a message in CAN that tells the entire system that is it alive
Front Ecu is alive
1
signal_bus_CAN
<left_indicator>
<right_indicator>
<brake>
<ref_speed>
<mode>
2
Auto Pilot
3
Logging
signals from the bluetooth that needs to be sent to the back ECU
Send_left_indicator
Send_right_indicator
Send_brake
Send_ref_speed
Mode
Auto
Logging_status
Out_can_id
Out_data_upper
Out_data_lower
Subsystem_CanTx
CanTx
S-Function7

Figure 38: CAN drivers in the Front ECU architecture

library contains a custom code called ”my_custom.cpp” that reads the data retrieved from the serial port and processes it as well as sends the messages and the feedback messages to the android application. The custom code has to be included in the main project otherwise it would not run. Look at Figure 30 in 5.3.1 for more information, the same principle used in the back ECU is applicable for the Front ECU. The Bluetooth sends and receives messages with a sample time of 0.01 s, which is as fast as the sample time of the entire project. The Bluetooth module receives the messages from the application one at a time, which means that the messages are kept in a queue and there is no priority function to decide which messages to send depending on the hazardous level. This functionality should be fixed for next year as the brake messages is more important than the light indicator message for instance. The custom code of the Bluetooth library handles the messages and reads them, so if changes are to be made it will be there. The language used is C++ and is part of the Simulink driver in the Bluetooth folder.

Now to the hardware part, the Front ECU consists of an Arduino Due as the main micro-controller that is put inside a 3D-printed box. The AMPSEAL connectors gathers all the wires in one place as can be seen in Figure 39 and that makes the circuit board more robust. The Bluetooth module is also beside the front ECU circuit board inside the Front ECU box pointing up to get good reception.

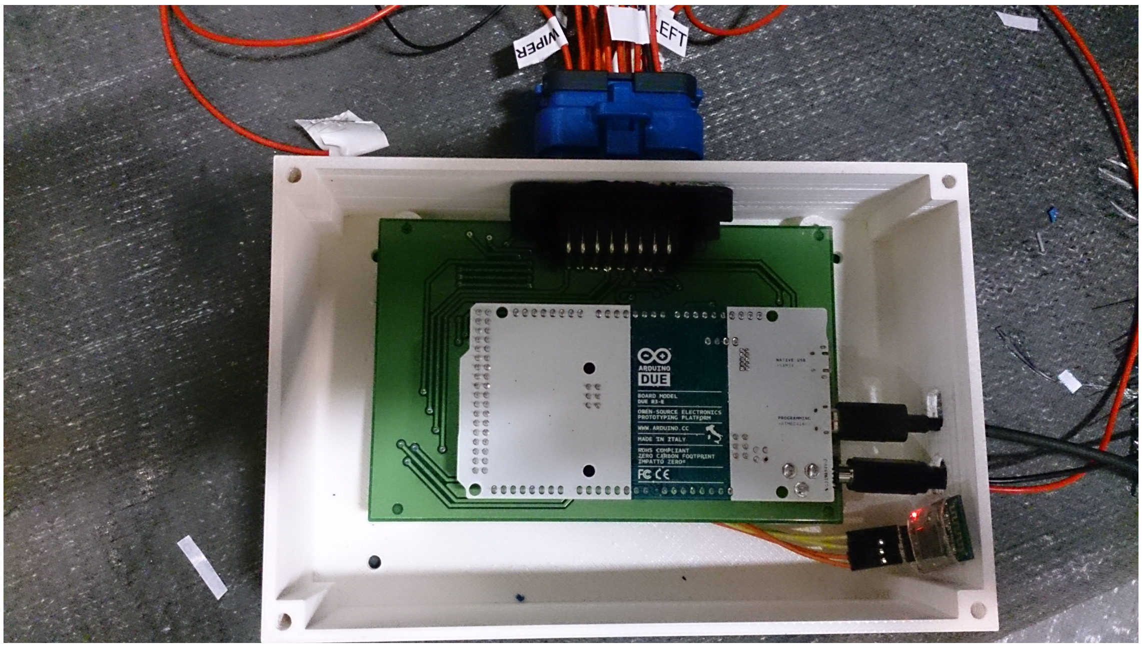


Figure 39: Front ECU circuit board

The schematics in Figure 40 show the layout of the entire Front ECU. The signals come from the AMPSEAL connector and are connected to different digital input signals on the Arduino Due. The CAN driver receives CANH and CANL from CAN through the AMPSEAL connector as well. The Bluetooth module is connected to the Arduino Due through wires from the circuit board as can be seen in Figure 40. For more detailed information about the pin connections on the Arduino Due, see appendix C.

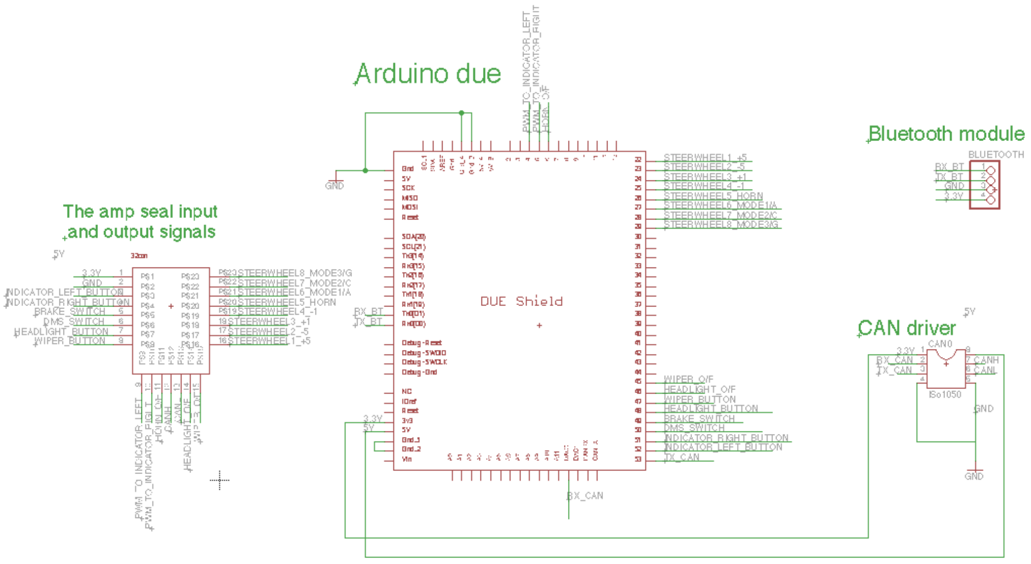


Figure 40: Front ECU Schematics

5.5.2 Bluetooth Module

The Bluetooth is a cheap and reliable wireless communication technology for short distances and has proved to be suitable for the car environment. The Bluetooth connection has been working well in the car as the android tablet and the front ECU are within the range of $0,5$ m and does not have any external interference.

Moreover, the Arduino Due supports serial communication and has three serial ports. The Bluetooth communication is done through the serial port2 as can be noticed in pin layout in the appendix, and in that way the messages that are sent over the serial port can be seen using the Arduino serial monitor as well as when testing the Bluetooth communication the external mode on Simulink.

The Bluetooth module that is used in this project is the HC-06, which can be found in many retail sellers in Stockholm. The device is applicable for handsfree service in cars and for Bluetooth GPS service, keeping a high-performance standard for wireless transceiver system. Furthermore, the device has a low error rate and is very light, cheap and small, see Figure 41. The device has low-power consumption, which is grand as the rules for the competition might change and require measuring the power consumption from the auxiliary battery. Also, the device is modular in the sense that the device name, pin code and the baud rate can be changed. In order to manipulate the device settings the modem commands (AT commands) are used. For example, sending AT+BAUD8 over the serial port will lead to a change of baud rate to 115200. For more details, see the Arduino code for serial communication on SVN, where the Arduino SoftwareSerial library was used.

5.5.3 Android application

This section will describe the android application that was programmed for the Samsung tablet that has been previously used in the Eco Cars project. The Android Studio is an open-source integrated development environment (IDE) for developing android platforms; the IDE is based on the open-source community IntelliJ IDEA. The IDE is developed to be easy to learn and provide a lot of support Libraries. The IDE provides a integrated capabilities such as adding missing libraries, providing advanced code navigation and establishing a layout editor with support for drag and drop theme editing etc.

The android application was built up from scratch in order to spread the concept of modularity to the java programming as well. The GUI was designed using the graphical interface that the Android Studios provides when using the xml files. After which, the buttons on the screen were given modular names-meaning that the function of the buttons are

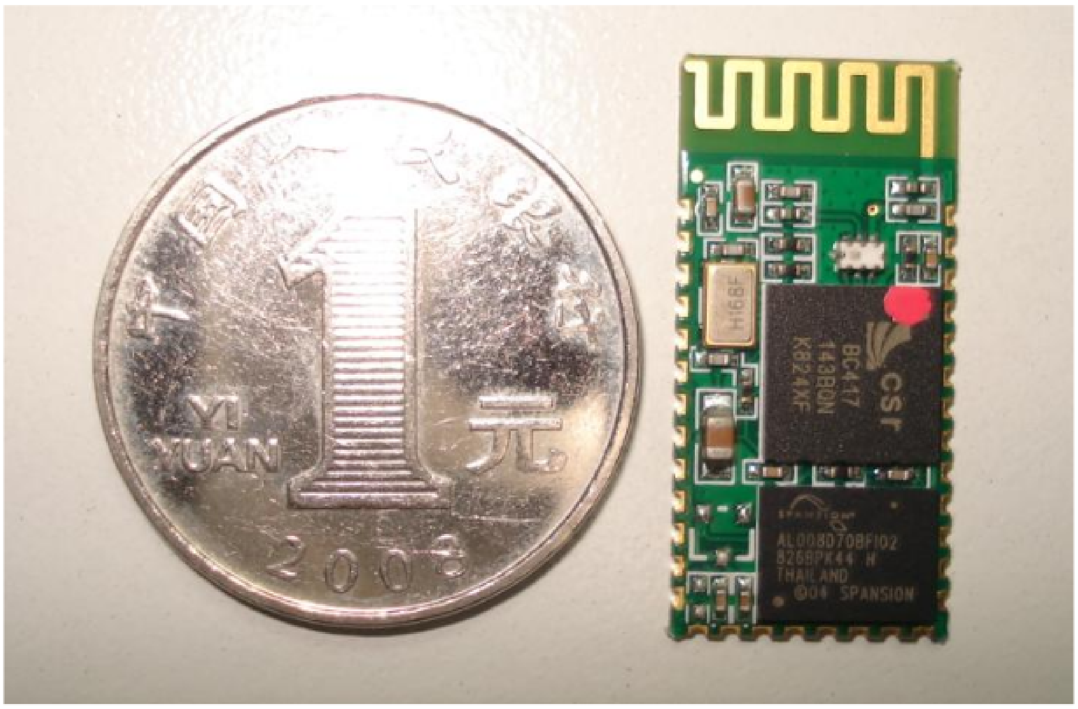


Figure 41: Bluetooth physical dimensions

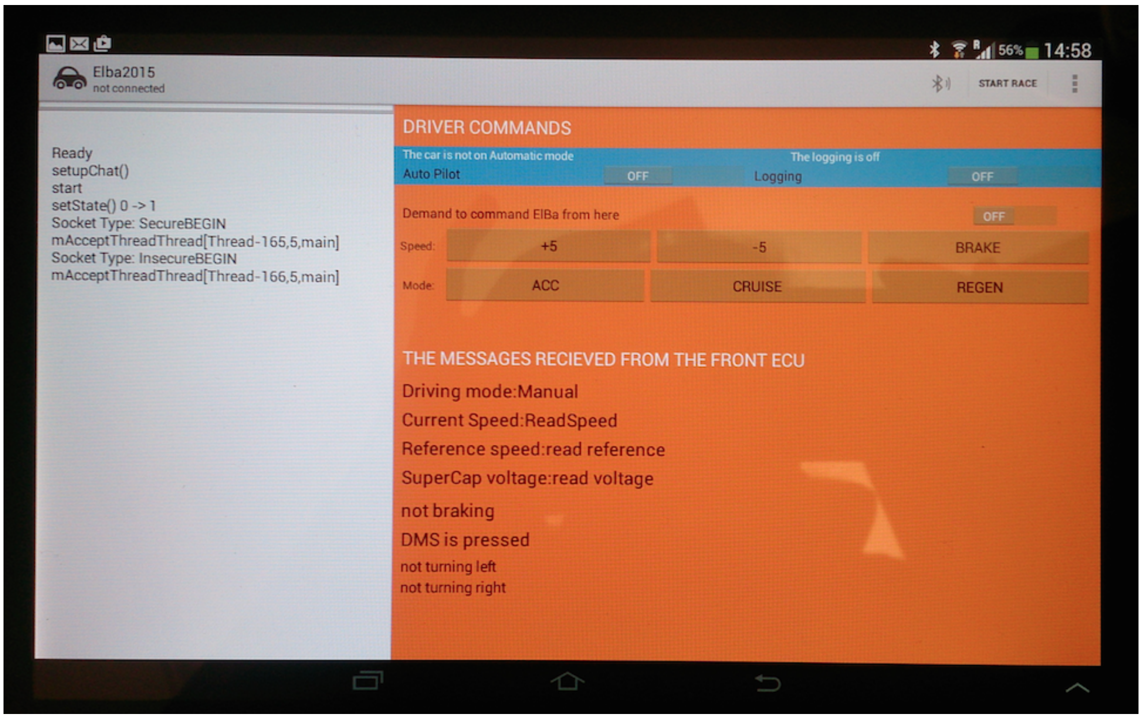
easily changed because they are given names as button$_A, button_B etc. - sothatifthefunctionalityofthebuttons$

The main focus of the application was to establish the wireless connection to the Bluetooth. The application looks like shown in Figure 42, where the toolbar contains the buttons that requests to connect the Bluetooth module and starts to receive and send messages when a connection is established. The feedback messages from the Front ECU are shown on the application layout as text views. However, the application can only connect to Bluetooth modules because it creates a secure outgoing connection to a remote device using the SDP lookup of the given universally unique identifier (UUID) and therefore the application cannot be connected to a mobile phone as the UUID is different.

5.5.4 The different circuit boards

This section will describe the circuit boards that are used in the car concerning the Horn circuit, the indicator driver board and the driver board for the wiper. Both status and recommendation will be discussed shortly.

Firstly, the driver board for the wiper was not used in the last iteration because the servo-motor that is used for the wiper needs an input signal ranging from $6 - 7$ V and the Arduino Due sends out a maximal output voltage of 5 V. The servo-motor is a strong one and is fast, therefore perhaps a smaller motor could be sufficent if the friction on the windscreen could be reduced. A solution for this is to use a Traco for a dc/dc



```
Ready
setupChat()
start
setState() 0 -> 1
Socket Type: SecureBEGIN
mAcceptThreadThread[Thread-165,5,main]
Socket Type: InsecureBEGIN
mAcceptThreadThread[Thread-166,5,main]
```

The car is not on Automatic mode
Auto Pilot OFF
The logging is off
Logging OFF
Demand to command ElBa from here OFF

Driving mode:Manual
Current Speed:ReadSpeed
Reference speed:read reference
SuperCap voltage:read voltage
not braking
DMS is pressed
not turning left
not turning right

Figure 42: The android application on the Samsung tablet

voltage regulator in order to get a higher voltage level. In order to fix this problem a quick solution would have been to include a Traco in a simple designed board, and then connected by wires to the circuit driver that contains a Atmel Atmega 16 microcontroller, see Figure 43. However, that would have not been a robust solution for a long-term use as it is an experimental board. Furthermore, as that board was not included in the scope, the circuit was left as it is and a more robust and well designed board should be included in the future work of the Front ECU.

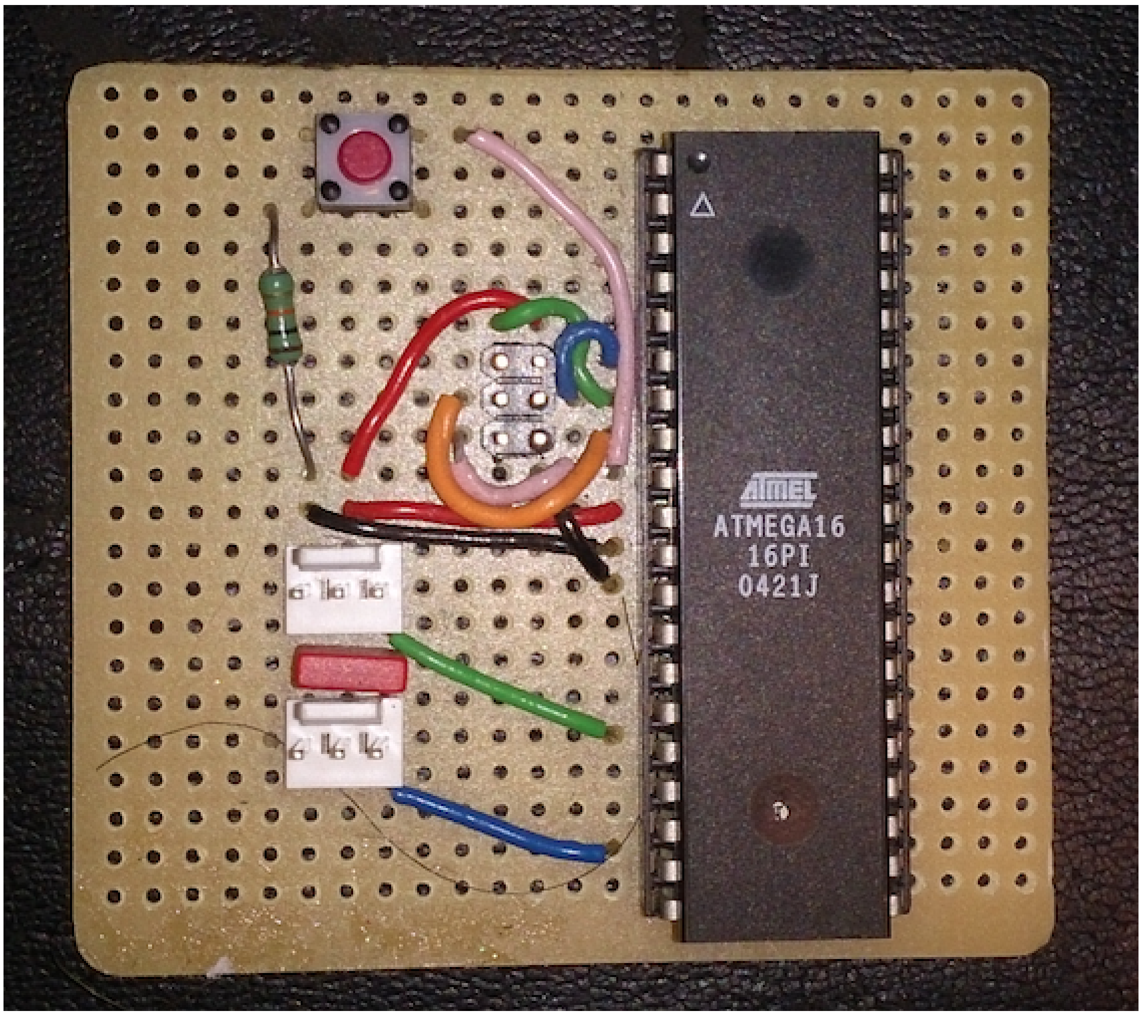


Figure 43: The Wiper experimental board from 2014

Secondly,the light indicator driver was included in the scope due to the fact that these signals were controlled from the old Front ECU and when the new design was done, it was concluded that it was more robust and lighter to have in total 5 wires (12 V, GND, 3 I/O for headlight, left indicator and right indicator) instead of connecting each light-bulb a 12 V and a GND. Moreover, with a driver the brightness of the indicators and headlight can be controlled. This feature is necessary in future cases where the consumption of the battery is included in the competition. This way the team could regulate the power consumption through the PWM that is send from the Front ECU, see Figure 44.

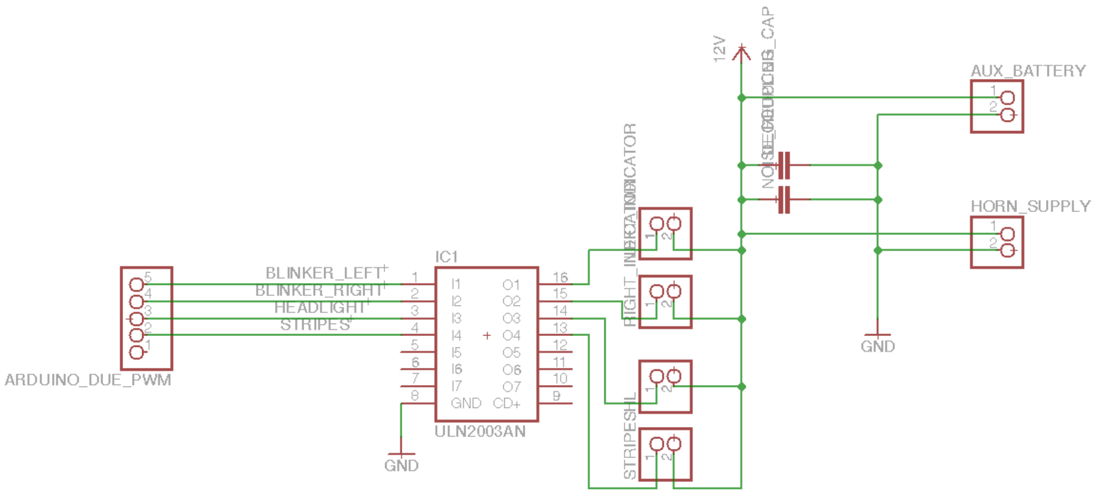


Figure 44: The Schematics of the light-driver

Thirdly, the voltage supply in the light driver circuit board comes from the auxiliary battery, and was designed so that the voltage level and the capacitors could be changed if the level of noise were high. The Horn supply pins are designed to provide the horn circuit with 12 V supply voltage but due to high level of noise on the input signals, the 12 V wire were separated, see Figure 45.

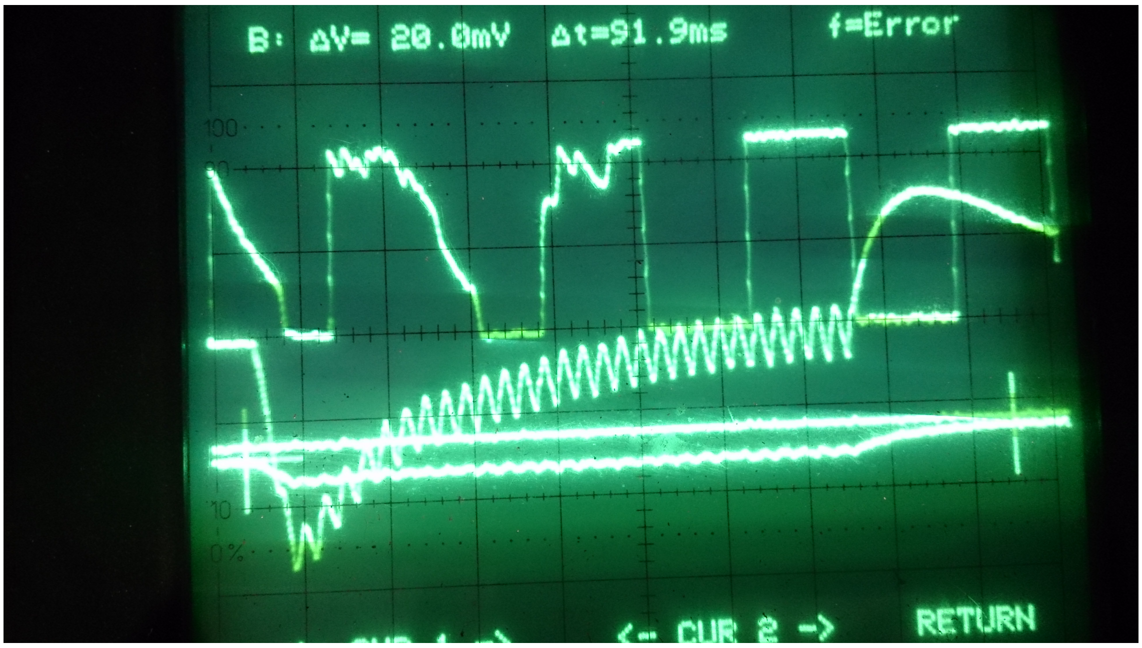


Figure 45: The disturbances caused by the horn sound wave caught on the input and the supply line

The above figure shows how the difference between the signals look like when the horn was pressed and when it was not. The PWM-signals in the middle of the screen were measured from the input signal on the horn circuit (the input signals from the Front ECU). Moreover, the second signal was measured on the 12 V line that was feeding the horn-circuit; the line was also affected by the horn. Due to the high current that went in the split-second when the horn was honking, the voltage level sank and then became high again most probably because of the capacitor on the board. This design is from 2014 and should be re-designed for the competition in 2016. The solutions available are to either change the horn or to analyze the cause and effects of the horn and design a board that can handle the interference on the 12 V line. For more details about the horn-circuit look at the report written in 2014 for the HK-course that cane be found on SVN.

Lastly, there are some recommendations for how to improve these circuit boards. The horn circuit that was build by the previous team used transistors that worked in the logical level of 5V and the Arduino Due sends a 3.3V supply volt via the on-board regulator and therefore the transistors had to be replaced. This is however a provisional solution. Also, the max current that is drawn is 50 mA, which contributes to lower consumption but it is advised to be careful in order not to draw too much current from the Arduinos because otherwise they will be damaged. Moreover, a new design for the horn circuit is in order and should be carefully done due to the fact that the horn disturbs the input signals as well as the 12V wire, look at previous reports from the Eco cars projects where a more detailed description of the board is available. The Front ECU uses serial port 2 (UART), and that is due to that fact that the Arduino Due cannot run on external mode if the serial port 1 is used. However, when designing the board a mistake was done and serial port 1 was used and to solve the problem wires were soldered on the boards that were manufactured at Cogra Pro AB. This can be easily fixed in the schematics.

5.6 Automatic Driving Mode

To further decrease the fuel consumption, an automatic driving mode is designed for the car. The main purpose of this controller is not to globally optimize the fuel consumption, but instead ease the interaction for the driver whilst driving. In a sense, this controller can be seen as a automatic transmission selecting the drive mode in accordance to a number of parameters. Two of the current parameters are the reference velocity set by the driver and the actuation of the break, which constitutes all the current interaction to the automatic driving mode made by the driver.

Not only does this improve the drivers possibility to drive more fuel efficient by not having to ensure safe operation manually, the automatic driving mode also provides safe operation for drivers without in-depth knowledge about subsystem thresholds and critical limitations for safe operation. This mode can therefore be seen as a baseline as to what the current configuration for Elba demands, for example a capacitor that reaches a voltage level no higher than 48V during its operation.

The automatic drive mode is designed using MATLAB Stateflow. Stateflow is an environment were it is possible to model state machines and in the case of the automatic drive mode, the state machine is the different driving states [12]. As seen in the automatic drive mode displayed in Figure 46, the output variable is only what mode the transmission should switch to depending on the reference velocity, current velocity, voltage in the car, and if the driver is breaking.

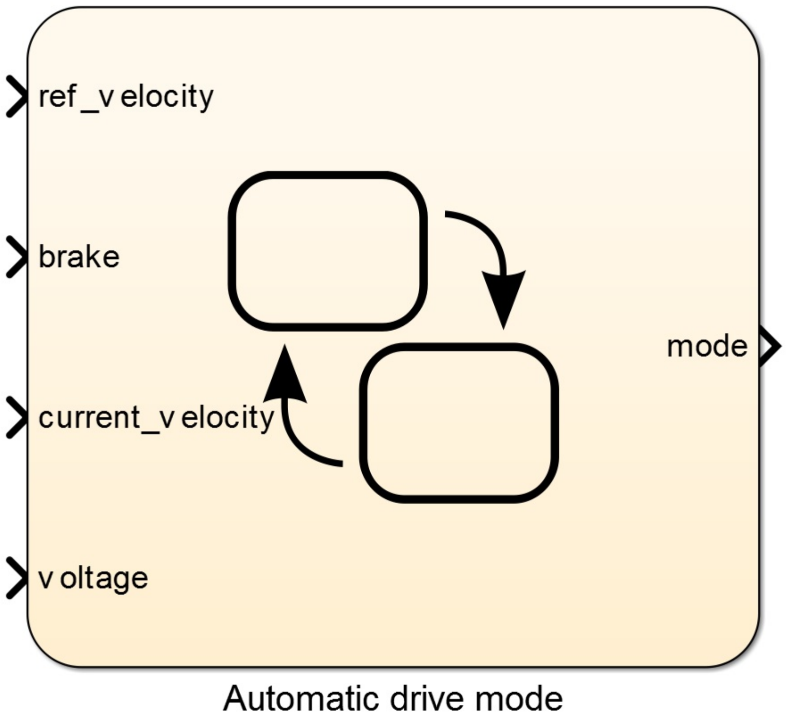


Figure 46: Automatic drive stateflow model

The signals inside the Stateflow block in Simulink are connected through a series of internal and external states with guards and actions for each transition between them. The controller utilizing a heuristic control strategy aims to achieve mainly two objective; safe driving condition and situation for the driver of the car and those around it, and safe and non-detrimental operation of the mechanical components withing the powertrain's clutch system.

Being the most crucial and essential objective, the safe operation of the vehicle is displayed in Figure 47 as the "BREAK" state. In this way the vehicles state has been broken down into either "braking" or "not braking/CAR RUNNING". As these two states are at the highest level in the state hierarchy they will always be checked no matter in what state the car should be in otherwise. A desired result of this is that when pressing down the break pedal, the car should disengage the electric motors, shut off the ICE, and disengage all the power generators from the drive shaft as to not force the driver to break the cars motors or engine at the same time as he or she tries to break the kinetic energy of the car.

Objectives of lesser importance such as a non-damaging operation or state of the vehicle may concern e.g. voltage level ratings of the super capacitor. This is maintained through transition guards evaluating that the voltage level never falls beneath 41 V without entering or currently residing inside the "CHARGING" state and that the the drive mode is never inside the "CHARGING" state if the voltage level reaches 48 V or higher.

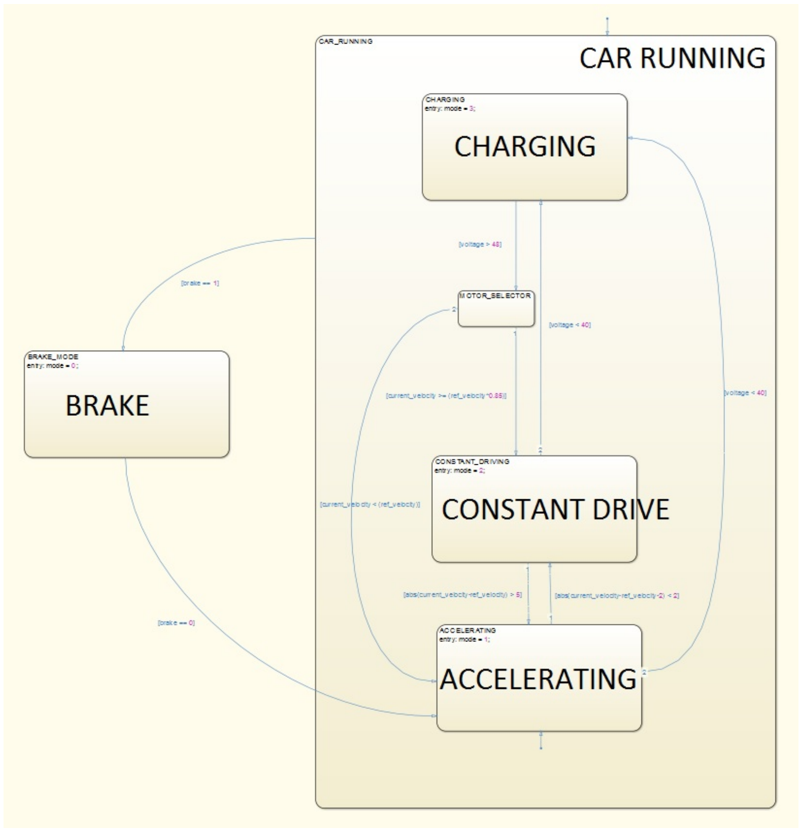


Figure 47: Automatic drive stateflow model

Other objectives that do not have the same level of importance also exist in this stateflow diagram. Two of these objectives are when to switch between utilizing the bigger electric machine for acceleration and when to cruise at a constant speed with the small electric machine. Since the small motor according to calculations and driving tests conducted during the competition in Rotterdam cannot maintain a constant velocity due to its lack of power output it is expected that the driving mode is to oscillate. A simple chain of events would be that the driver requests a higher velocity at which the bigger motor is engaged. After reaching the reference speed the smaller motor takes its place, but instantly causes the vehicle to lose velocity. This forces the bigger motor to once again engage and thus enclosing the circle of events leading to an oscillatory behavior. A solution to this problem is to include a more complex transition guard for transitions between "CONSTANT DRIVE" and "ACCELERATING".

In the case of a greater desired acceleration the driver will most surely demand a change of reference speed in such a way that the smaller motor cannot supply the necessary power outputs. This would in this case correspond to a change of velocity by ±5 km/h. This behavior is captured in

$$|v_{current} - v_{ref}| > 5 \tag{14}$$

were the $v_{current}$ is the measured current velocity of the vehicle and v_{ref} is the reference velocity set by the driver. This works for both accelerating and decelerating due to the absolute value operand and effectively causes the bigger motor to absorb the power of a harder generative break, which is desirable.

The opposite transition needs to take into account that the small motor cannot maintain the desired reference velocity and should therefore strive to overshoot[1] the reference velocity by a small amount, during certain conditions. The overshoot should occur when driving at higher velocities than reference velocity so to not break away too much kinetic energy in the process. It is therefore considered beneficial if the transition occurs before the measured velocity meets the reference velocity since the small motor can cruise towards the target velocity without taking into account clutch engagement losses near the target value. In the case were the measured velocity is lower than the reference velocity there should be little or no overshoot to decrease the amount of excessive power output from the powertrain. This type of functionality is generated by using

$$|v_{current} - v_{ref} - 2| \leq 2 \tag{15}$$

as a transition guard from "ACCELERATING" to "CONSTANT DRIVE". Here the automatic driving mode will switch to the "CONSTANT DRIVE" state if the current speed overshoots the reference velocity. For acceleration this overshoot will be 1 km/h

[1]Overshoot meaning that the velocity is higher than the reference velocity

and for deceleration this will be 3 km/h, with overshoot meaning a velocity, $v_{current}$, *higher* than the requested reference velocity, v_{ref}.

The chosen mode is then sent to another Stateflow model that controls the clutch system and enables the different motors. This drive mode control can be seen in Figure 48.

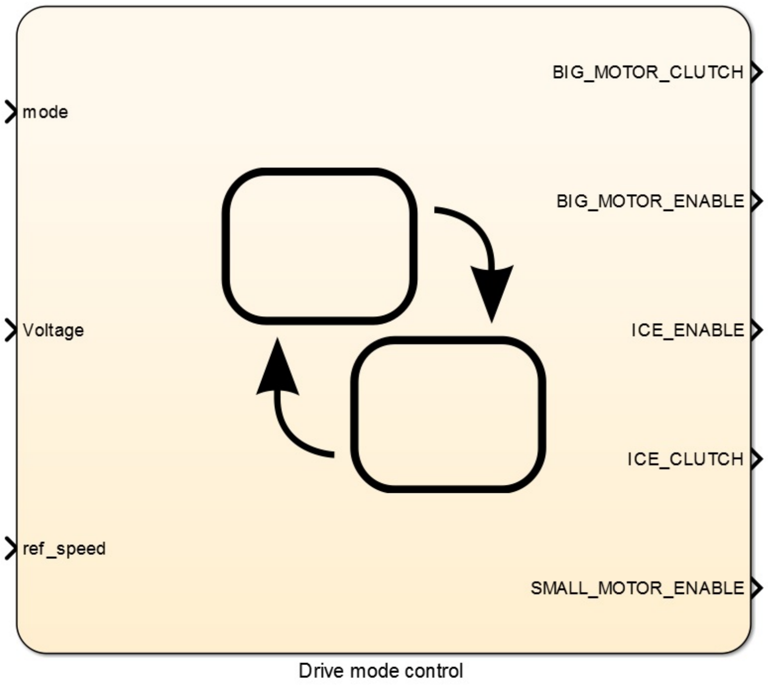


Figure 48: Drive mode control stateflow model

There are a few reasons for separating what mode to choose from and the actual clutch and motor enables and actuation. The main reason is that the drive mode control is dependant on the current clutch system implementation. It is now possible to change the clutches without having to completely change the mode-select logic. Hence, the heuristic control logic or whichever logic implemented in the automatic drive mode stateflow diagram is loosely connected to the hardware implementation and therefore becomes modular and adaptable depending on the desired behavior.

Another benefit of such a decoupling strategy for the mode selection is that the latter control stateflow, the drive mode control, ensures safe transitions from one state to the other by implementing logical boundaries and demands on its internal states retrieved from the hardware of the clutch system itself. This means that even though changes to the drive mode are requested at a fast rate the operation of the mechanical clutch will not directly respond to these demands. Instead the drive mode control will take into account that changes in the mechanical design are not instantaneous, as they are in the automatic driving mode, and will therefore not allow rapid changes between driving modes that can lead to a dangerous driving condition for the driver, but also a harmful engagement for the intrinsic mechanical components.

6 Results

This section presents the results from simulations and logged data from running the real system.

6.1 Automatic drive mode

The automatic driving mode has been tested thoroughly by both simulations and on the real system. The automatic driving mode works as intended and assists the driver by making choices in terms of what motor should be run at a certain time.

Figure 49 shows the capacitor voltage as a function of time during the automatic drive mode.

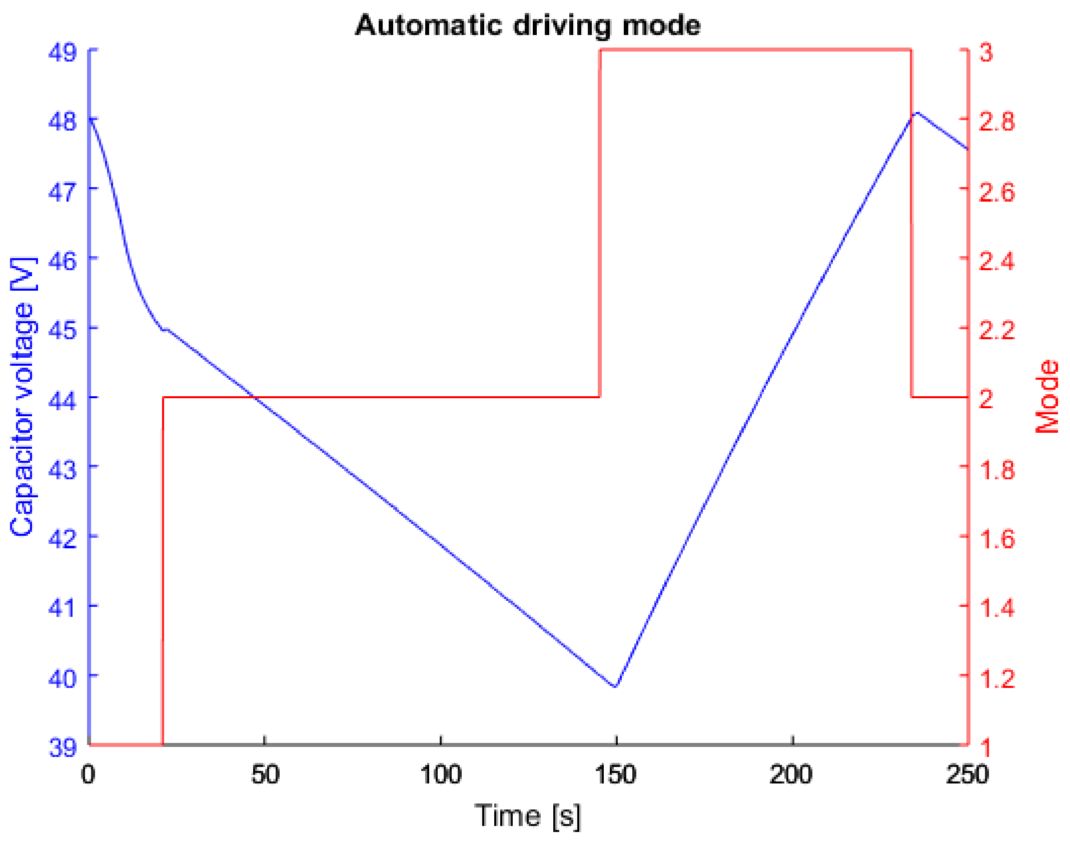


Figure 49: Capacitor voltage and mode value shown as a function of time during automatic drive mode.

6.2 Plant model

Simulations have been done in order to evaluate the correctness of the plant model. The simulations have been carried out during conditions that are similar to the driving conditions which were present during logged data. The data used for comparision and evaluations was logged at the Shell Eco-marathon race, during test run at Barkaby airport and during freewheel tests in lab.

Figure 50 shows the capacitor voltage and Figure 51 shows the output shaft rpm during test run at Barkaby airport.

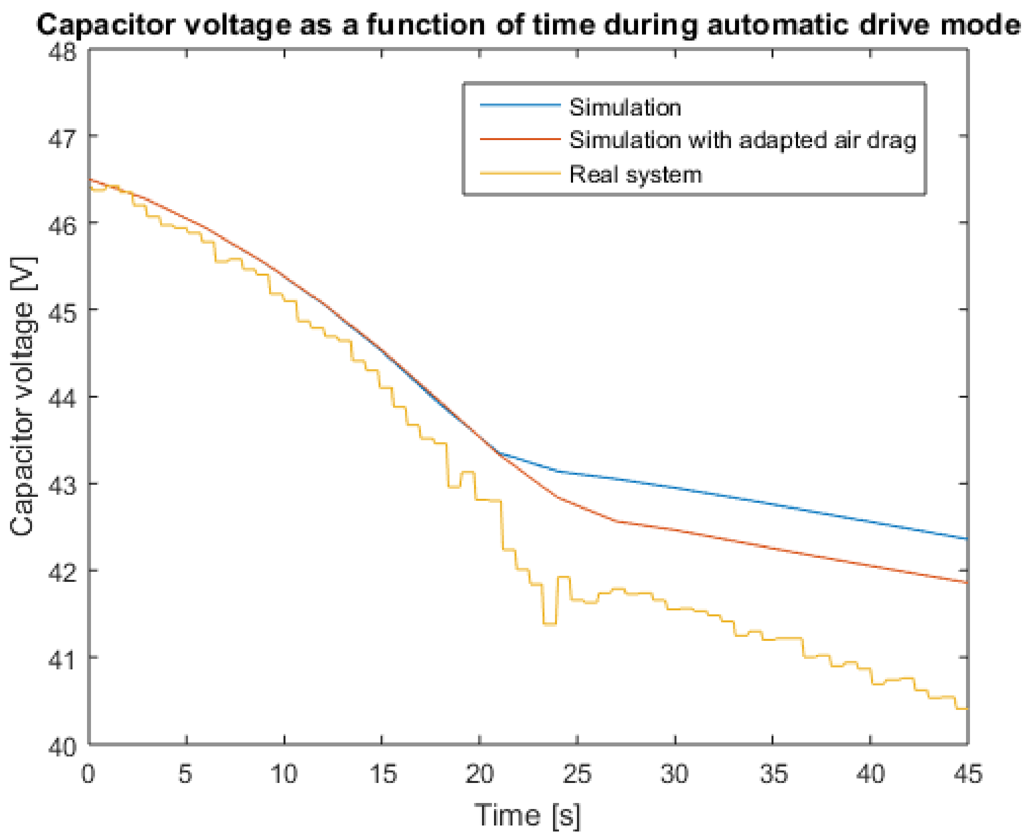


Figure 50: Capacitor voltage during automatic drive mode

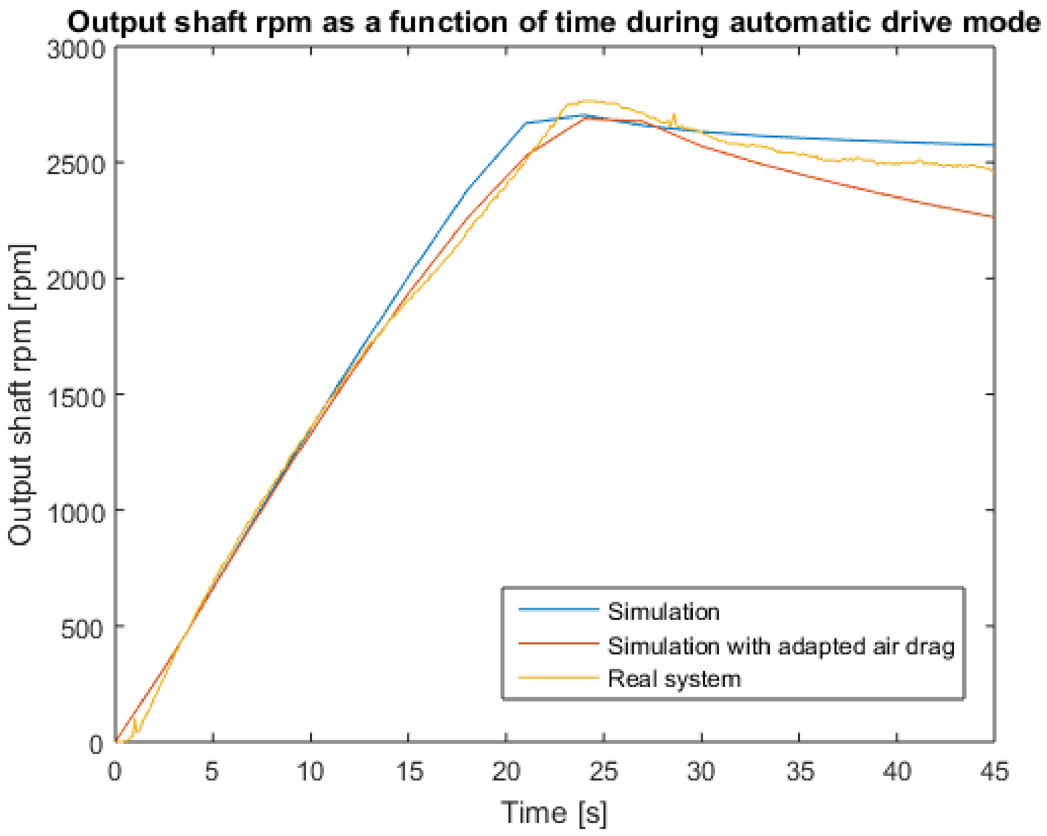


Figure 51: Output shaft rpm during automatic drive mode

Note that by output shaft, it refers to the output of the transmission just before the planetary gear, hence a factor 10 higher than what is on the wheel. The simulation differs from the logged data by a bit. One of the reasons for this is due to the heavy wind that was present during the test run. In order to show how it affects the simulation result was done with an adapted air drag which was used to simulate the presence of wind. It's possible to see that the simulation with adapted air drag is closer to the real system. However, there is still a difference between the curves in terms of slope. One of the reasons for this difference is that the real system has a bit of an overshoot compared to the simulated system, which results in that it consumes more energy to reach it's final speed. Another reason for the difference is that the real system is a bit less energy efficient than the simulated one at this power output. A way to make the difference smaller is by modifying the efficiency curves of the two motors slightly. The automatic drive mode attaches the small DC motor when it reaches 2750 rpm, which can be seen in the graphs. It is clear that the model of the small DC motor differs from the real system. One of the reasons lies in that the driving condition is not easily reproduced in simulations. Another reason for the slope difference is that the small motor is not strong enough to keep the speed which is why the current limiter within the motor controller affects the system. The current limiter of the real system and the plant model are not identical in operation, which results in a slight difference at which current the system is clamped, hence the slope difference.

Figure 52 shows regenerative breaking of the system at a speed of 2750 rpm when running the wheel freely in the air.

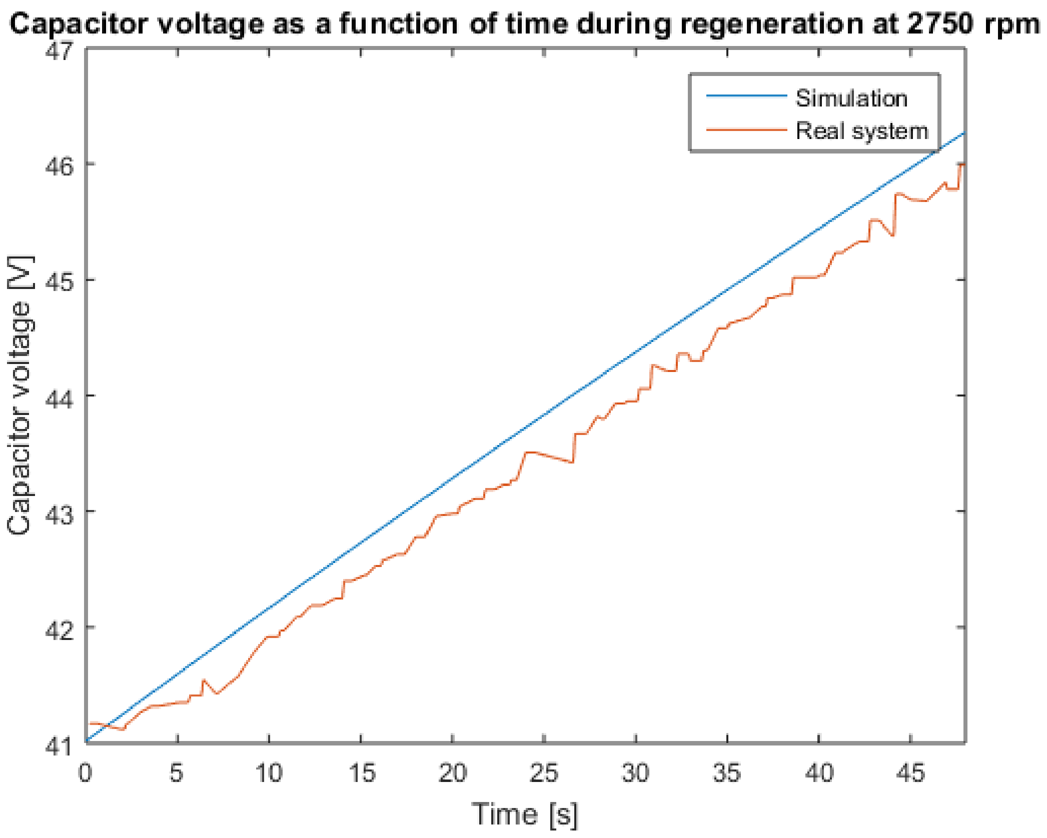


Figure 52: Capacitor voltage during regenerative breaking during freewheel drive at 2750 rpm

The plant model was modified for this test case by removing the influence of air drag and rolling resistance in order to make it as close to the driving conditions of the real system as possible. The slope is almost identical between the simulation and the real system. The reason that the curves do not lie completely on top of each other is due to the real system having a small voltage drop at the start of the regenerative breaking shown in the figure.

Figure 53 shows regenerative breaking of the system at a speed of 2750 rpm during the Shell Eco-marathon.

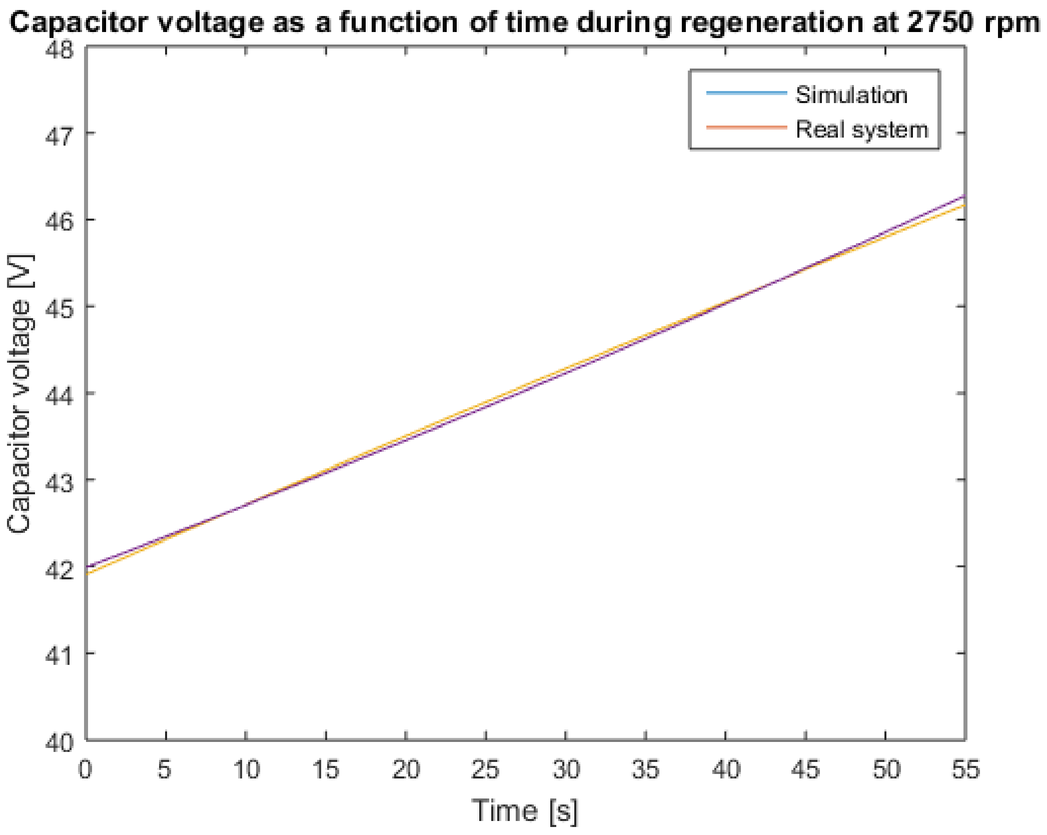


Figure 53: Capacitor voltage during regenerative breaking when driving on road

Interpolation has been done on the logged data from Shell Eco-marathon, due to the low sample rate which was used. The simulation and real system are almost identical, it's possible to see however that the real system is not completely linear during this test, like the mathematical model suggests. The reason for this is that the real system will never be completely linear since there are external factors that are nonlinear, such as the wind changing and the road surface not being ideal.

The result from the plant model in terms of fuel consumption shows that the Shell Eco-marathon race takes 38.3 minutes with the current setup and the average fuel efficiency is 200 km per litre. Compared to expected results, this is quite good. From data analysis of the unsuccesful race from the competition, where the small motor was working properly, the result should be between 150-200 km per litre.

In conclusion, the results show that the plant model simulates the real system correctly in terms of behaviour. There are differences due to simplifications in the plant model, parameters not being completely correct and external factors, such as wind and uneven road surface.

7 Discussion and Conclusion

As a general and summarizing conclusion the project and its results meet the expected outcomes set out for the project in its earliest stages after the Shell Eco-marathon competition. Therefore it is concluded that the working process of this project and the methods and tools used should be considered successful in the context of HEV powertrain development at ITRL.

Building on this deduction it is also very important that future student teams working on the Eco Cars continues in the same direction as set out by this project. As the main idea for making a robust and modular design is to enable continuous improvements over time and across multiple project teams, all work will be for naught if an entire new platform is conceived and implemented again. Even though the results from the competition or the test runs after the competition point towards greater capacity when it comes to fuel efficiency, there is still many areas were improvement would increase the car's potential even further. Even though there exists some inconsistencies within the simulation of the vehicle and the reality, it is to be considered profitable in terms of time and effort to build upon the current platform in order to improve incrementally instead of redesigning and redoing the whole platform from the start. Tools that enable this kind of continuous work are the code-generating capability of Simulink and the custom drivers made for it, the verification and validation tools from external mode in Simulink and CAN-monitoring and simulation from Vector hardware and CANoe software, and the easy-to-come-by Arduino microcontrollers.

Other decisions that have made a positive impact on the results from the project and performance of the team is the choice of powertrain architecture. Even though the architecture's complexity creates many situations were the system, subsystems, or components might break, the design itself allows for many different driving modes. During the competition in Rotterdam this resulted in a completed first attempt since the drivetrain was fixed to minimize the risk of faults. In the second attempt the vehicle was instead configured is such a way that the driving modes could be changed during the race, improving the result by approximately 50 %. At this point the driving modes were still selected by the driver. Therefore, it is obvious that even further improvements are possible with the same architecture and slight changes and improvements may in fact change the ambition of the KTH Eco Cars team to reach for more prestigious race results.

Although there is a possibility to decrease the fuel consumption, reaching amongst the top three in the competition will not be a likely scenario. This is because the most fuel efficient teams only run with a single ICE as traction engine, thus minimizing the losses in the drivetrain, decreasing weight, etc. Despite the advantage of a simple design in the competition, this choice of powertrain architecture does not pose a realistic design for a vehicle to be used in any other context than the Shell Eco-marathon competition. Furthermore, this solution does not involve complex and interesting control algorithms and strategies which will diminish the academic content of the project.

As the decision to use a larger ICE was made early due to its efficiency benefits compared to smaller engines the necessity for a wider efficiency range for the electric motors was apparent. By exploiting the different efficiency ratings from two different motors, the car can more easily target a fuel efficient operating state. In addition, this type of powertrain allows for more interesting driving opportunities considering more driving conditions and at the same time allow for fuel efficiency optimization. This type of solution is highly appreciated since it does not only allow for experimentation and increases the educational value of the powertrain architecture and drive modes, but also a theoretical more fuel efficient HEV.

Even though simulation of the vehicle and its subsystems are performed and the plant model is verified to some extent it is desirable to further test and benchmark the different power generators, motors end the engine, more accurately. If future implementations of optimization of the powertrain is to be done for Elba, it is essential that there exists efficiency mappings for each torque and rotational velocity. This will surely lead to better global fuel efficiency if the controller aims to sustain each motor or engine at this power point. This will be a feasible objective as the torque demand could be split across the powertrain to maximize the global efficiency. Although the direct optimization problem is clear, some changes towards the basic vehicle control needs to be made to accommodate for this new control strategy. This is because the driver input to the car is made in form of a reference velocity, much like a cruise control in a production vehicle. This does not create an as obvious optimization problem when the torque is cascaded into the motor driver in one case, the big electric motor controller, and in another case cascaded with a PID controller inside Simulink, the small electric motor controller, and through an unknown correlation between PID controller governing a lambda value for the ICE and the output torque in a third case.

As this project was the first to run the car Elba on gasoline there was no precedence concerning the vehicles fuel efficiency and hence no demands or goals concerning affected results. With the achieved results at hand it is now possible to set out a goal for future designs and implementations. As the unfinished race, where a mechnical sprint in the angled gear broke, gave a calculated fuel efficiency of 145 km/l this should be the minimum expected result for future iterations. Especially since the gear ratio for the small electric motor is not optimal for its operating point and the performance of the gasoline engine did not meet the initial estimation with regard to its fuel consumption. An estimate of the achievable fuel efficiency should be close to or more than 200 km/l.

8 Recommendation and Future Work

The super capacitor that is used now in the car weighs 13 kg and has a rated capacitance of 165 F, which could be reduced. However, this design choice is dependent of the driving cycle. A suggestion could be that the chemist students can get involved to provide a source of power that weighs lighter and is more innovative.

Regarding the HMI, the wiper board needs a new circuit design or change the servo-motor, which could be easier as it only requires one that works on 5V. For the horn circuit, however, a new design is required and a more robust board needs to include all the parts, which is not the case right now: The light indicator, the horn and the single diode board (flyback diode) should be on the same board to ensure the level of robustness required and in order to reduce the number of wires as well. The Dead man Switch(DMS) that should be in the car, according to the rules, is only acting as a manual switch and that could be integrated in the software so that the driver can judge that the reason for a potential failure was or was not due to a faulty switch. The android switch and the latest iteration of the Simulink model have been designed to be able to read the DMS and show it in the application as a message. However, the remaining part is to change the hardware architecture so that the switch is not a serial-switch with the main car switch but a stand-alone switch.

Furthermore, the android application was build from the start because the old one was not modular and did not permit the user to make major changes. The android application is a first prototype and does not include a good and easily comprehensible GUI due to the fact that the handover should be easy and open for new design. For example, adding time and GPS to the application helps the driver to know exactly where in the lap he/she is and also calculates the time to finish and show the number of laps. These design choices were left to the new team because the location of the competition has been changed and the new team can customize the application accordingly to the new changes.

Regarding the clutch system, there are a lot of things to do and improve. Firstly, the clutch circuit board should be redesigned in order to add robustness to it. The board was done in May 2015 and as it is working well as it was left as it is. However, the same concept that was used for the ECU could be implemented on the clutch circuit board, the board can be manufactured by professionals in Cogra Pro and a 3D-printed box could be printed using the same design used in the three ECU and replace all the connector with a amp-seal connector. A major future work will be to change the entire hardware for the clutch, meaning the mechanical parts, actuators, and the circuit board. The actuators are over dimensioned and should be changed; the current sensor should be replaced with one that is more sensitive to really small currents. The H-bridge should also be changed, or a new one could be built so that the selection of transistor is optimal with consideration to the voltage and the current that the actuators need.

The software for the clutch-system is final and does not need improvements but as the hardware changes in the future, the software has to be adjusted. Future work on the clutch-system is to include CAN as the main communication between the back ECU and the clutch circuit board. Moreover, an Arduino Due should be used to get consistency through the four important boards in the car.

There are a lot of possibilities for improvement with the plant model. The main priorities should be to verify the small electric motor better. Due to lack of tests on real system it is not verified enough. A second part of interest for future improvments is to implement the advanced BLDC motor model in the plant model. It is provided in the delivery of the project, but it is not currently used within the plant model due to difficulties with identifying the parameters. By spending some time on this, it would be possible to improve the accuracy of the overall plant model, which might be beneficial for more accurate investigations regarding design choices.

Furthermore, there are some issues in the car that can be optimized concerning the mechanical parts. For example, the gear that is used between the small motor and the driving shaft is not optimal and has a gear ratio that was decided as a result of miscommunication. Moreover, the shelf were the circuit boards are placed is over the drivetrain but it is not smooth to place all the parts there, especially the encoder for the small motor and that should be redesigned.

Lastly, the small motor is enabled during the automatic mode only if the mode is set on "cruise" so the small motor can not be enabled during accelerating or regenerating. So a future improvement could be to enable the small motor in desired situation other than just when the speed is constant.

9 References

[1] K. He, H. Huo, Q. Zhang, D. He, F. An, M. Wang, and M. P. Walsh, "Oil consumption and {CO2} emissions in china's road transport: current status, future trends, and policy implications," *Energy Policy*, vol. 33, no. 12, pp. 1499 – 1507, 2005.

[2] S. Kirstan and J. Zimmermann, "Evaluating costs and benefits of model-based development of embedded software systems in the car industry–results of a qualitative case study," in *Proceedings Workshop C2M: EEMDD "From code centric to model centric: Evaluating the effectiveness of MDD" ECMFA*, 2010.

[3] P. F. Smith, S. M. Prabhu, and J. Friedman, "Best practices for establishing a model-based design culture," 2007. SAE Technical Paper.

[4] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann, "What is the benefit of a model-based design of embedded software systems in the car industry?," *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, p. 310, 2013.

[5] A. T. Kamil Çagatay Bayindir, Mehmet Ali Gözüküçük, *A comprehensive overview of hybrid electric vehicle: Powertrain configurations, powertrain control techniques and electronic control units.* Çukurova University, Department of Electrical and Electronics Engineering, Balcali, Saricam, Adana, Turkey.

[6] A. Corporation, *Atmel SAM3X8E SAM3X8C SAM3X4E SAM3X4C SAM3A8C SAM3A8C Datasheet.* Atmel Corporation, 1600 Technology Drive, San Jose, CA 95110 USA, 2015.

[7] A. Corporation, *Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.* Atmel Corporation, 1600 Technology Drive, San Jose, CA 95110 USA, 2014.

[8] MathWorks®, *Developing S-Functions.* The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, 2015-09.

[9] K. Krykowski, Janusz Hetmańczyk, "Constant current models of brushless dc motor," *Electrical, Control and Communication Engineering*, vol. 3, pp. 19–23, 2013.

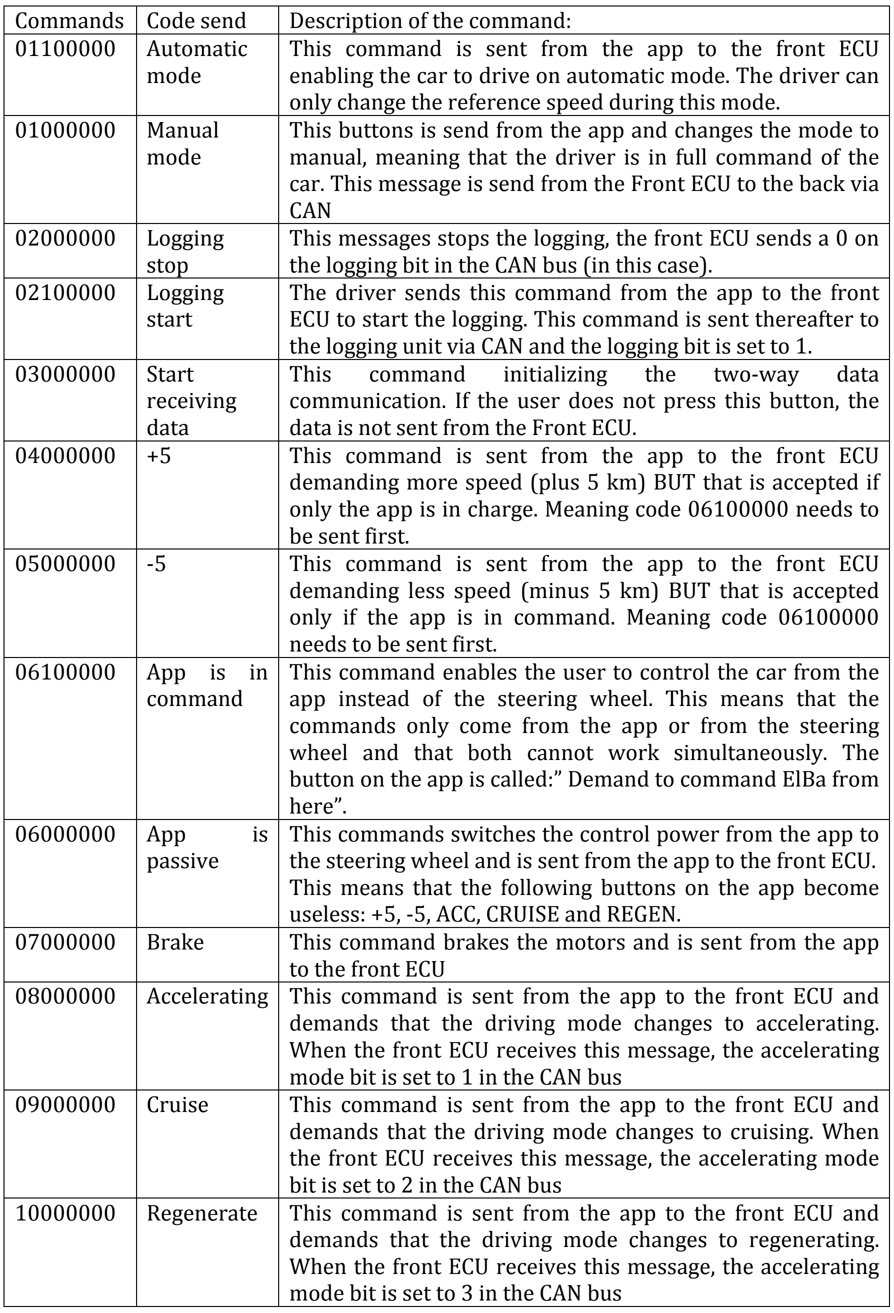
[10] C. Solutions, "*CANopen Solutions.*" http://www.canopensolutions.com/. Accessed: 2016-01-14, Last updated: 2016-01-14.

[11] C. Kidder, "*due_can.*" https://github.com/collin80/due_can. Accessed: 2016-01-14, Last updated: 2015-12-04.

[12] MathWorks®, "*Stateflow.*" http://se.mathworks.com/help/stateflow/index.html. Accessed: 2016-01-13, Last updated: 2016-01-13.

10 Appendix

Appendix A



Commands	Code send	Description of the command:
01100000	Automatic mode	This command is sent from the app to the front ECU enabling the car to drive on automatic mode. The driver can only change the reference speed during this mode.
01000000	Manual mode	This buttons is send from the app and changes the mode to manual, meaning that the driver is in full command of the car. This message is send from the Front ECU to the back via CAN
02000000	Logging stop	This messages stops the logging, the front ECU sends a 0 on the logging bit in the CAN bus (in this case).
02100000	Logging start	The driver sends this command from the app to the front ECU to start the logging. This command is sent thereafter to the logging unit via CAN and the logging bit is set to 1.
03000000	Start receiving data	This command initializing the two-way data communication. If the user does not press this button, the data is not sent from the Front ECU.
04000000	+5	This command is sent from the app to the front ECU demanding more speed (plus 5 km) BUT that is accepted if only the app is in charge. Meaning code 06100000 needs to be sent first.
05000000	-5	This command is sent from the app to the front ECU demanding less speed (minus 5 km) BUT that is accepted only if the app is in command. Meaning code 06100000 needs to be sent first.
06100000	App is in command	This command enables the user to control the car from the app instead of the steering wheel. This means that the commands only come from the app or from the steering wheel and that both cannot work simultaneously. The button on the app is called:" Demand to command ElBa from here".
06000000	App is passive	This commands switches the control power from the app to the steering wheel and is sent from the app to the front ECU. This means that the following buttons on the app become useless: +5, -5, ACC, CRUISE and REGEN.
07000000	Brake	This command brakes the motors and is sent from the app to the front ECU
08000000	Accelerating	This command is sent from the app to the front ECU and demands that the driving mode changes to accelerating. When the front ECU receives this message, the accelerating mode bit is set to 1 in the CAN bus
09000000	Cruise	This command is sent from the app to the front ECU and demands that the driving mode changes to cruising. When the front ECU receives this message, the accelerating mode bit is set to 2 in the CAN bus
10000000	Regenerate	This command is sent from the app to the front ECU and demands that the driving mode changes to regenerating. When the front ECU receives this message, the accelerating mode bit is set to 3 in the CAN bus

Figure 54: The commands send from the application to the Front ECU

Appendix B

Messages	Code	Description of the messages:
Current speed	“Speed=20;”	This messages means that the actual speed is 20 km/h
Reference speed	“Ref=19;”	This messages means that the reference speed is 19 km/h
Super cap voltage	“Voltage=48;”	This means that the voltage is 48 V
Driving mode	“Mode=1;”	This means that it is on ACC mode. Mode=2; means CRUISE and mode=3; means REGEN
Braking status	“Brake=0;”	This means that it is not braking. If message “break=1;” is received then the car is braking.
Left indicator	“Left=1;”	This message means that the left indicator button has been pressed and that the driver is or intends to turn left
Right indicator	“Right=0;”	This means that the car is not turning right, if “right=1;” is received then the right indicator is pressed
Logging status	“Logging=0;”	This is a feedback message. This means that the logging is off and to turn the logging on a “logging=1;” has to be sent from the app, see command code 02100000. If code 02000000 is send from the app, the front ECU responds by sending “logging=0;”
Automatic mode status	“Auto=1;”	This is a feedback message, meaning that if the front ECU receives a code 01100000 then the automatic mode is requested by the user and the front ECU echoes back that it has received the command by sending “auto=1;” otherwise it sends “auto=0;” when code 01000000 is demanded.

Figure 55: The commands send from the Front ECU to the android application

Appendix C

Position on the amp-seal connector /(pin number on arduino due)	The name of the signal	Function of signal
1 /(Vin)	Supply voltage 3.3V	This supply voltage could be used if needed in any circuit board in the future
2/(GND)	GND	This wire is sent to all the boards in the front of the car such as the light-driver, the horn circuit, fans. The GND from the arduino is connected to the battery's GND.
3/(52)	Turn left indicator button	This button is located on the panel on the car and is used to tell that the car is turning left
4/(51)	Turn right indicator button	This button is located on the panel on the car and is used to tell that the car is turning right
5/(49)	Brake switch	The switch is located at the mechanical brake and turn "1" when the brake is pressed
6/(50)	Dead man switch (DMS)	The switch is located at the front of the car and turn "1" when the DMS is released.
7/(48)	Headlight button (HL)	This button is located on the panel on the car and is used to tell that headlight are to be turned on
8/(47)	Wiper button	This button is located on the panel on the car and is used to tell that the wiper is to be set ON
9/(4)	Turn left PWM	If the left indicator is pressed then a PWM is send out on pin 4
10/(5)	Turn right PWM	If the right indicator is pressed then a PWM is send out on pin 5
11/(6)	Horn I/O	If the Horn button on the steering wheel is pressed then a logical 1 is send out.
12/(CAN driver)	CANH	This message is received by the CAN driver
13/(CAN driver)	CANL	This message is received by the CAN driver
14/(46)	Headlight I/O	If the HL button is on then a logical 1 will be set on and send to the light-driver board
15/(45)	Wiper I/O	If the wiper button is on then a logical 1 will be set on and send to the board
16/(22)	Plus five button	The button is on the steering wheel. This indicates that more speed is required
17/(23)	Minus five button	The button is on the steering wheel. This indicates that less speed is required
18/(24)	Plus one button	The button is on the steering wheel. This indicates that more speed is required
19/(25)	Minus one button	The button is on the steering wheel. This indicates that less speed is required
20/(26)	Horn button	The button is on the steering wheel. This indicates that the horn should honk. Turns to logical 1.
21/(27)	Mode 1/ACC button	The button is on the steering wheel. This indicates that the mode is ACC, and the message is send to the CAN bus
22/(28)	Mode 2/ CRUISE button	The button is on the steering wheel. This indicates that the mode is CRUISE, and the message is send to the CAN bus
23/(29)	Mode 3/REGEN button	The button is on the steering wheel. This indicates that the mode is REGEN, and the message is send to the CAN bus
-/(DAC0)	RX	The CAN driver sends the signal to the arduino due
-/(53)	TX	The CAN driver sends the signal to the arduino due
-/(17)	RX2	This pin is connected to the TX pin on the Bluetooth module through the serial port 2
-/(18)	TX2	This pin is connected to the RX pin on the Bluetooth module through the serial port 2

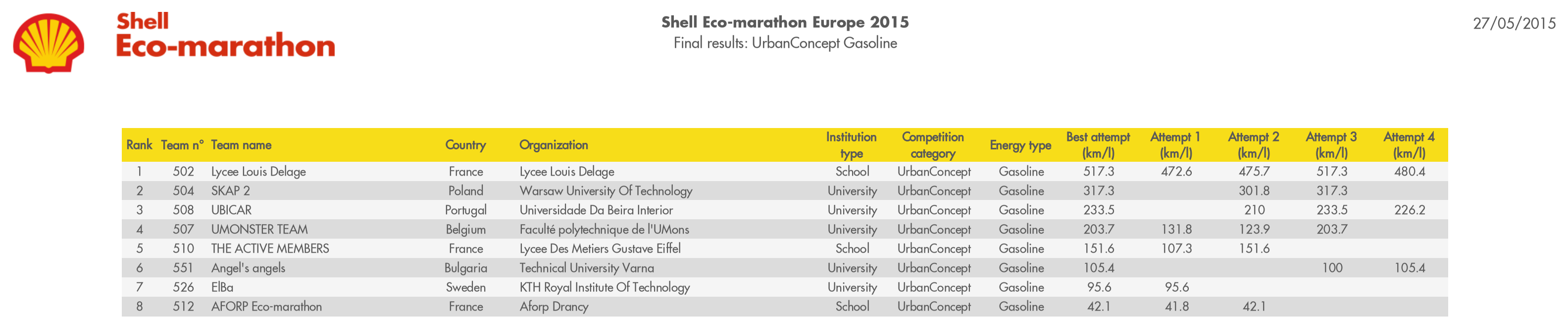
Figure 56: The pin layout on the Front ECU and the Arduino due

Appendix D

Table 1: Pin map of Back ECU AMPSEAL connector

AMPSEAL PIN	Signal name	Function
1	12V	+12V from Aux battery
2	ENCODERA12V	The phase A from the quadrature encodrer, at 12V logic
3	ENCODERB12V	The phase B from the quadrature encodrer, at 12V logic
4	ENCODERINDEX12V	The index, phase Z from the quadrature encoder, at 12V logic
5	INDICATOR_LEFT	Open drain output, PIN 7 on Arduino Due
6	INDICATOR_RIGHT	Open drain output, PIN 8 on Arduino Due
7	BRAKE_LIGHT	Open drain output, PIN 6 on Arduino Due
8	BACK_LIGHT	Open drain output, PIN 5 on Arduino Due
9	SMALL_MOTOR_PWM	Open drain output, PIN 10 on Arduino Due
15	CANL_CH0	CAN low signal of CAN bus for channel 0
16	SMALL_MOTOR_ENABLE	Open drain output, PIN 10 on Arduino Due
16	SMALL_MOTOR_ENABLE	Open drain output, PIN 22 on Arduino Due
18	CLUTCH_BIG_MOTOR	Open drain output, PIN 10 on Arduino Due
19	CLUTCH_ICE	Open drain output, PIN 10 on Arduino Due
20	CANL_CH1	CAN low signal of CAN bus for channel 1
21	CANH_CH1	CAN high signal of CAN bus for channel 1
22	GND	Ground, goes to the minus pole of the Aux battery.
23	CANH_CH0	CAN high signal of CAN bus for channel 0

Appendix E



Shell Eco-marathon

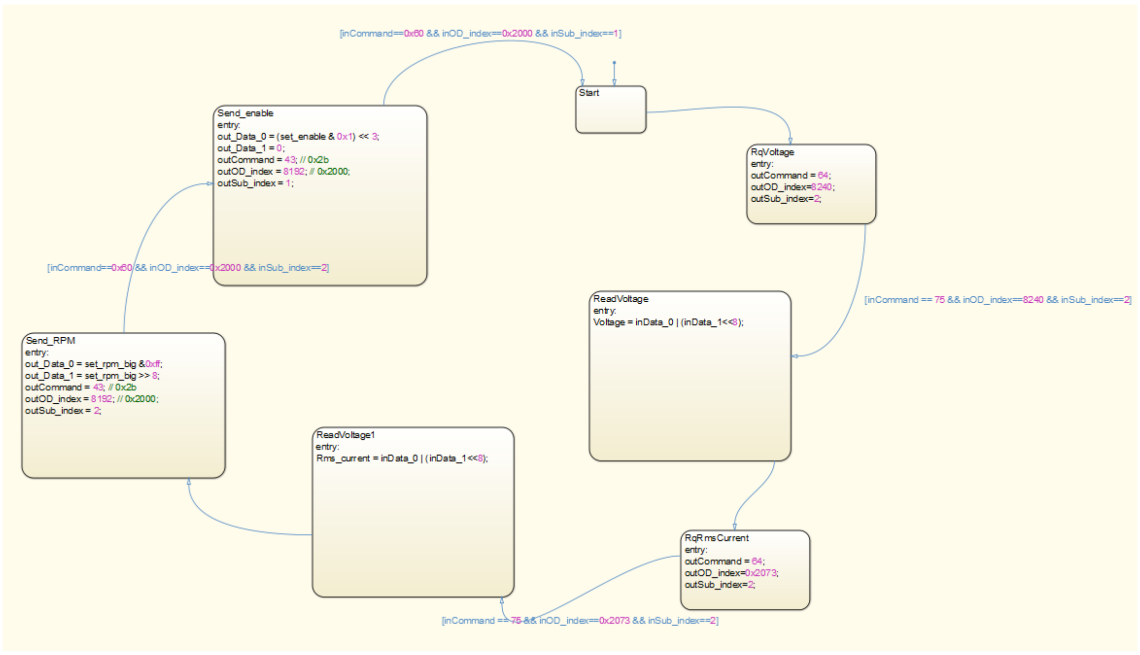
Shell Eco-marathon Europe 2015
Final results: UrbanConcept Gasoline

27/05/2015

Rank	Team n°	Team name	Country	Organization	Institution type	Competition category	Energy type	Best attempt (km/l)	Attempt 1 (km/l)	Attempt 2 (km/l)	Attempt 3 (km/l)	Attempt 4 (km/l)
1	502	Lycee Louis Delage	France	Lycee Louis Delage	School	UrbanConcept	Gasoline	517.3	472.6	475.7	517.3	480.4
2	504	SKAP 2	Poland	Warsaw University Of Technology	University	UrbanConcept	Gasoline	317.3		301.8	317.3	
3	508	UBICAR	Portugal	Universidade Da Beira Interior	University	UrbanConcept	Gasoline	233.5		210	233.5	226.2
4	507	UMONSTER TEAM	Belgium	Faculté polytechnique de l'UMons	University	UrbanConcept	Gasoline	203.7	131.8	123.9	203.7	
5	510	THE ACTIVE MEMBERS	France	Lycee Des Metiers Gustave Eiffel	School	UrbanConcept	Gasoline	151.6	107.3	151.6		
6	551	Angel's angels	Bulgaria	Technical University Varna	University	UrbanConcept	Gasoline	105.4			100	105.4
7	526	ElBa	Sweden	KTH Royal Institute Of Technology	University	UrbanConcept	Gasoline	95.6	95.6			
8	512	AFORP Eco-marathon	France	Aforp Drancy	School	UrbanConcept	Gasoline	42.1	41.8	42.1		

Figure 57: Results from Shell Eco-marathon in Rotterdam 2015

Appendix F



```
Start
```

```
RqVoltage
entry:
outCommand = 64;
outOD_index=8240;
outSub_index=2;
```

```
ReadVoltage
entry:
Voltage = inData_0 | (inData_1<<8);
```

```
RqRmsCurrent
entry:
outCommand = 64;
outOD_index=0x2073;
outSub_index=2;
```

```
ReadVoltage1
entry:
Rms_current = inData_0 | (inData_1<<8);
```

```
Send_RPM
entry:
out_Data_0 = set_rpm_big &0xff;
out_Data_1 = set_rpm_big >> 8;
outCommand = 43; // 0x2b
outOD_index = 8192; // 0x2000;
outSub_index = 2;
```

```
Send_enable
entry:
out_Data_0 = (set_enable & 0x1) << 3;
out_Data_1 = 0;
outCommand = 43; // 0x2b
outOD_index = 8192; // 0x2000;
outSub_index = 1;
```

```
[inCommand==0x60 && inOD_index==0x2000 && inSub_index==1]
[inCommand == 75 && inOD_index==8240 && inSub_index==2]
[inCommand == 75 && inOD_index==0x2073 && inSub_index==2]
[inCommand==0x60 && inOD_index==0x2000 && inSub_index==2]
```

Figure 58: Stateflow chart of CANopen communication with Inmotion controller.