NTNU – Trondheim
Norwegian University of
Science and Technology

Driver Interface for Shell Eco Marathon Vehicles

Vebjørn Røed Myklebust

Master of Science in Cybernetics and Robotics
Submission date: June 2015
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Summary

The goal of this thesis was to create a touch-screen system for the driver in the new DNV GL Fuel Fighter Urban Concept vehicle. The screen was to display relevant information for the driver, as well as a video-stream from the rear of the vehicle, acting as a supplement to the traditional rear-view mirrors. The same video stream was also to be shown online, to allow for anyone to follow the race in real time. In addition to this, a screen was to be created for the DNV GL Fuel Fighter Prototype vehicle, as well as systems for handling input from the driver, in both cars. The aim was to complete all these subsystems and install them in the vehicles, making them ready to compete in the Shell Eco Marathon competition in May, 2015. The work was performed in great collaboration with the other members of the team, requiring input and interaction with many other subsystems of the vehicle. It required careful planning and a methodical and scheduled approach. Most of the work was highly practical, and countless hours were put in at the workshop. The systems utilized premade parts, as well as custom designed parts. Required circuit boards were designed in Eagle, and custom programs were written in Python and C. The work in this thesis did not build upon any related previous work, and everything was done during the spring semester of 2015. As well as working on this thesis, the author put in time in helping the mechanical team to complete the car on schedule. The result was highly satisfactory, and the finished product of this thesis was embedded and used in the Fuel Fighter vehicles during competition. The systems were reported as being functional and beneficial, and would definitely be reused in the future.

Sammendrag

Målet med denne avhandlingen var å skape et touch-skjerm system til sjåføren i det nye DNV GL Fuel Fighter Urban Concept kjøretøyet. Skjermen skulle vise relevant informasjon for føre-nen, samt vise en video-strøm fra baksiden av bilen som et supplement til de vanlige kjørespei-lene. Den samme video-strømmen skulle gjøres tilgjengelig på internet, slik at hvem som helst skulle kunne følge racet i sanntid. I tillegg til dette skulle det lages en skjermløsning til DNV GL Fuel Fighter Prototype kjøretøyet, samt systemer for å behandle input fra føreren i begge bi-lene. Målet var å ferdigstille alle de overnevnte systemene og installere de i bilene så de ble klare til konkurransen Shell Eco Marathon i Mai, 2015. Arbeidet ble utført i sterkt samarbeid med de andre lagmedlemmene ettersom det var mye samspill mellom de elektriske og mekaniske systemene. Arbeidet krevde nøye planlegging og en metodisk og tidsbestemt tilnærming. Det meste av oppgaven var praktisk arbeid, og utallige timer ble tilbrakt på verkstedet. Systemet tok bruk av både ferdiglagde produkter og spesiallagde løsninger. Kretskort ble designet i Eagle, og det meste av programvaren ble skrevet i Python og C. Avhandlingen bygger ikke på tidligere arbeid, og alt ble planlagt og utført iløpet av vårsemesteret 2015. I tillegg til arbeidet på denne oppgaven, ble det også investert mye tid i å hjelpe den mekaniske gruppen i å ferdigstille selve bilen. Resultatet ble meget tilfredsstillende, og det ferdige produktet ble brukt i kjøretøyene un-der konkurransen. Systemene ble evaluert til å være både funksjonelle og til stor hjelp, og vil definitivt bli brukt igjen.

Preface

This thesis was carried out during the spring semester of 2015, as part of the DNV GL Fuel Fighter project. The project was conducted under the Department of Engineering Design and Materials, although most of the work in this thesis was conducted under the Department of Engineering Cybernetics, NTNU.

This thesis is intended for readers with an interest in embedded systems, or wish to learn more about the systems embedded in the the Fuel Fighter vehicles. Given the strong connection with the mechanical systems in the cars, this thesis would also be interesting for readers with a background in design or mechanical engineering.

I would like to thank everyone on the Fuel Figher team for an unforgettable semester, as well as supervisor Amund Skavhaug for his assistance and guidance with this thesis.

Trondheim, 2015-07-15

Vebjørn Røed Myklebust

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Figure 1.1: Shell Eco Marathon group photo, Rotterdam, 2014 [15]

1.1 Background

Shell Eco Marathon is an annual competition held simultaneously in America, Europe, and Asia. It is a competition between student teams from different universities that design, build, and race custom built cars that aim to be as efficient as possible. The race consists of driving a given distance (16 km) within a given time (39 minutes), with the goal to use as little energy as possible. There are several methods of propulsion a car may use, including standard gasoline and diesel combustion engines, battery or fuel cell powered electric motors, or alternative fuels like ethanol. The common metric is km/kWh which is used to compare results from the different methods and also to determine the winner. As well as different classes for propulsion, there are two different classes for the type of cars; Urban Concept, and Prototype. The Urban Concept class is for cars that look like traditional road cars in the sense that they have 4 wheels, front and rear lights, indicators, windshield wipers, etc. The Prototype class, however, is less restricted and the cars usually have 3 wheels and a more futuristic appearance.

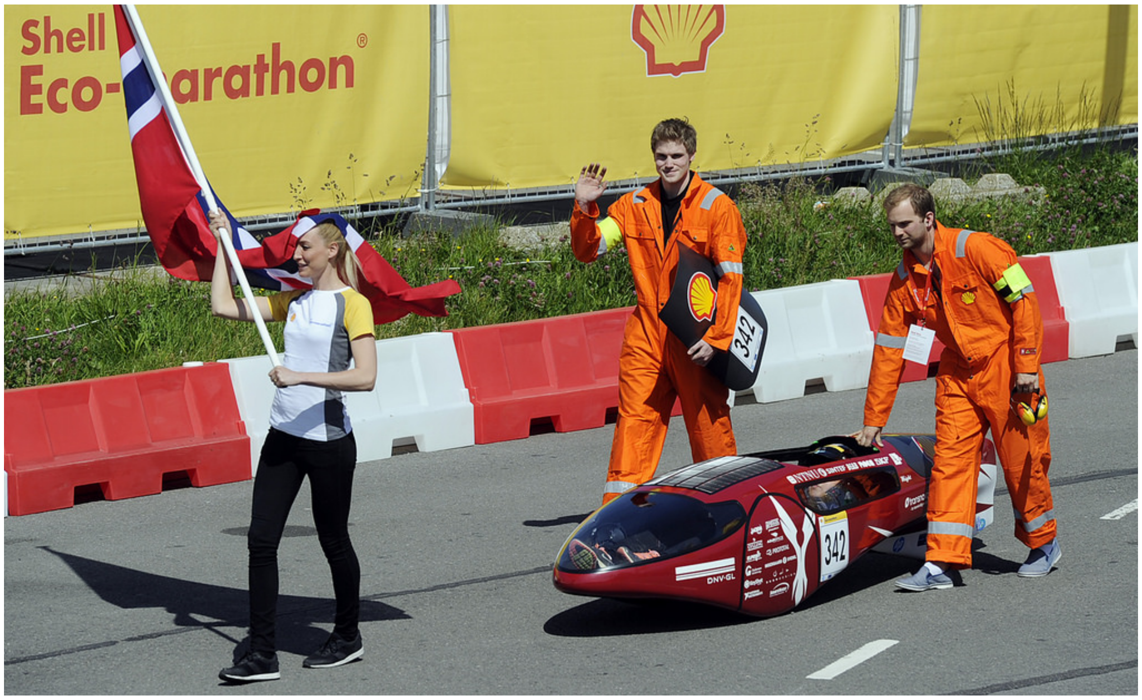


Figure 1.2: Prototype Vehicle [17]

Each year since 2009, NTNU has participated in the competition with a team mainly consisting of last year students. The car has always been in the Urban Concept class except in 2014 when the team built a new Prototype car and participated in both classes. The same prototype car was also used in 2015, but the team decided early to build a new Urban Concept class car for the 2015 competition. The race this year was held in Rotterdam, Netherlands. The winning team from last year managed to drive 1091 km/kWh, while the Fuel Fighter prototype managed 612 km/kWh [28].

1.1.1 The team

The team for the 2015 competition was formed early in the autumn semester of 2014, and gradually expanded throughout the semester. The project required a broad range of expertise and consisted of 6 master thesis students, 2 bachelor students, and 19 volunteer students. The fields of study included mechanical engineering, electrical engineering, material science, computer science, media studies, film production, and interaction design. The team was also joined by three students from HiST as well as two exchange students from Germany and one from France who could unfortunately only work on the project for one semester.

Figure 1.3: DNV GL Fuel Fighter 2015 team [16]

1.1.2 The project

As decided by the current team members during the autumn semester, 2014, the team was going to create an entirely new Urban Concept car. The team would also improve the Prototype vehicle, and both cars were going to be used in the 2015 competition. The improvements of the prototype vehicle included mainly mechanical work to decrease the rolling friction, but also to upgrade some of the electrical systems. The propulsion system in both cars were planned to be identical, using batteries and the same motor-controller and motors. However, due to new Shell Eco Marathon regulations, a team may not compete with two cars using the same propulsion method, which was an initial setback. The team was forced to use a different system in the Urban Concept vehicle. Changing the propulsion system in the prototype would have required too much work, and there was very limited space to work with. The compartment of the Prototype had been designed with a small electric motor and a battery in mind, so installing a combustion engine or fuel-cell stack would not have been possible. It was decided by the team that the Urban Concept should still use an electric motor, but with a hydrogen fuel cell as an energy source. This meant that little change was required regarding the control system, and most of the old components could be reused.

1.1.3 Prototype Electrical system

The existing electrical system of the prototype is outlined in figure 1.4. A similar system of the Urban Concept can be seen in figure 1.7.

Since the prototype vehicle was functioning well with its current system, most of the focus was shifted towards implementing a system in the new Urban Concept vehicle. Both systems were based on the same backbone as nearly all previous cars. This backbone is a modularized CAN-based system with no central brain in the car, but rather a set of equal nodes. Each node performs a simple function, based on input from another node. All nodes share a common data bus, to which nodes can easily be connected or disconnected. Most nodes use a universal module as an interface to the CAN-bus, and then build their functionality on top of this.

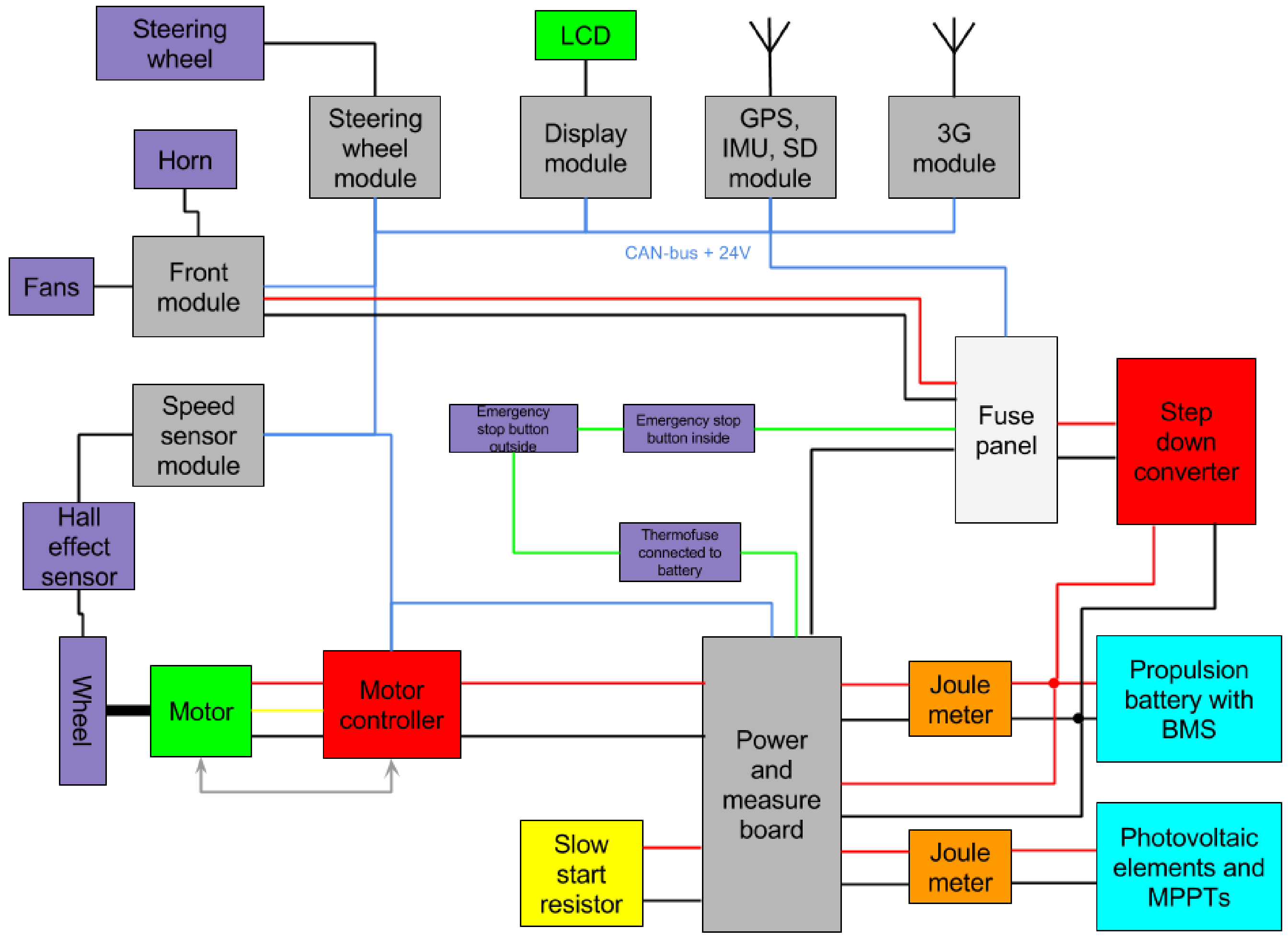


Figure 1.4: Block diagram, Electrical System - Prototype [7]

Universal Module

The Universal module is a small PCB designed by Anders Guldahl in 2009 for use in the Shell Eco Marathon vehicle at the time. It contains an at90can128 microcontroller from Atmel as its processing unit, a 5V buck converter that can take a large range of input voltages, a CAN-transceiver to connect to the CAN-bus, and a variety of pins connected to a GPIO header to which other modules can attach. The physical layer of the CAN-bus consists of standard RJ11 plugs and cables, also called telephone-cables. The plugs are 6P6C although only 4-wire cables are used. Using a 4P4C plug would not have been possible, because two of the necessary signals are wired to pin 1 and pin 6 respectively. The universal module was designed so that two additional signals could be added in the future, but this has not been necessary yet. The 4 signals consist of the standard differential CAN-bus lines CANH and CANL, as well as GND and 24V. The 24V signal is used to power the nodes connected to the CAN-bus. There are two RJ11 ports on each module so that the bus can be terminated in that node, or passed on to another node. The GPIO header contains 20 pins - two ground pins, one 5V pin, one 3.3V pin, and 16 GPIO pins from the at90can128 microcontroller. Among these 16 pins are UART-pins, SPI-pins, and four ADC pins.

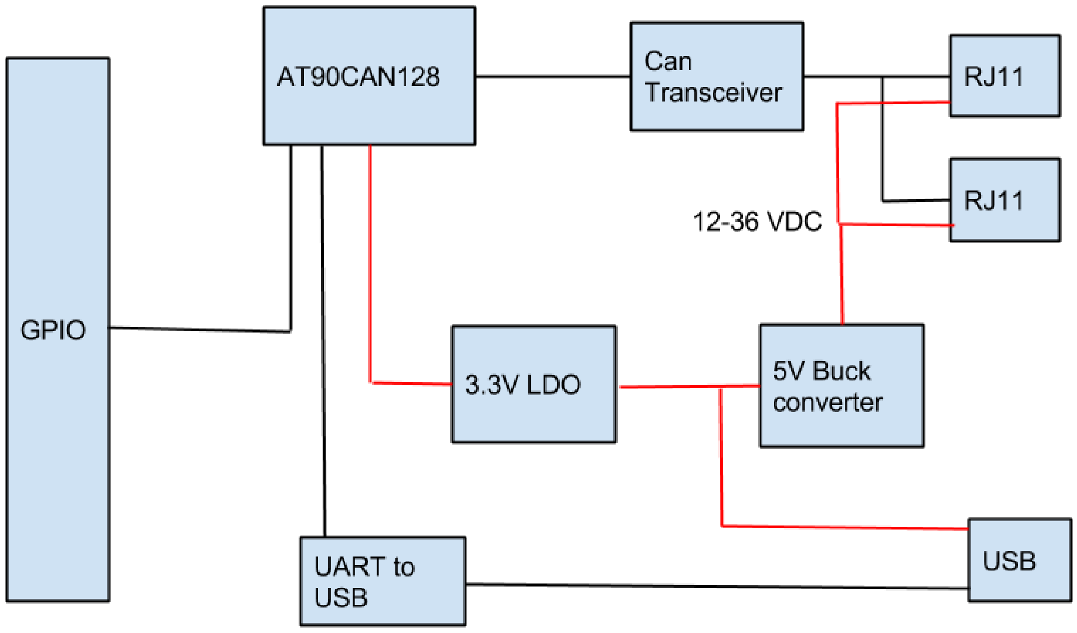


Figure 1.5: Block diagram, Universal Module

Power Supply

For the Prototype vehicle, all the energy in the car came from one battery. This was a change from last year, when one battery was for propulsion, and another battery was allowed to power

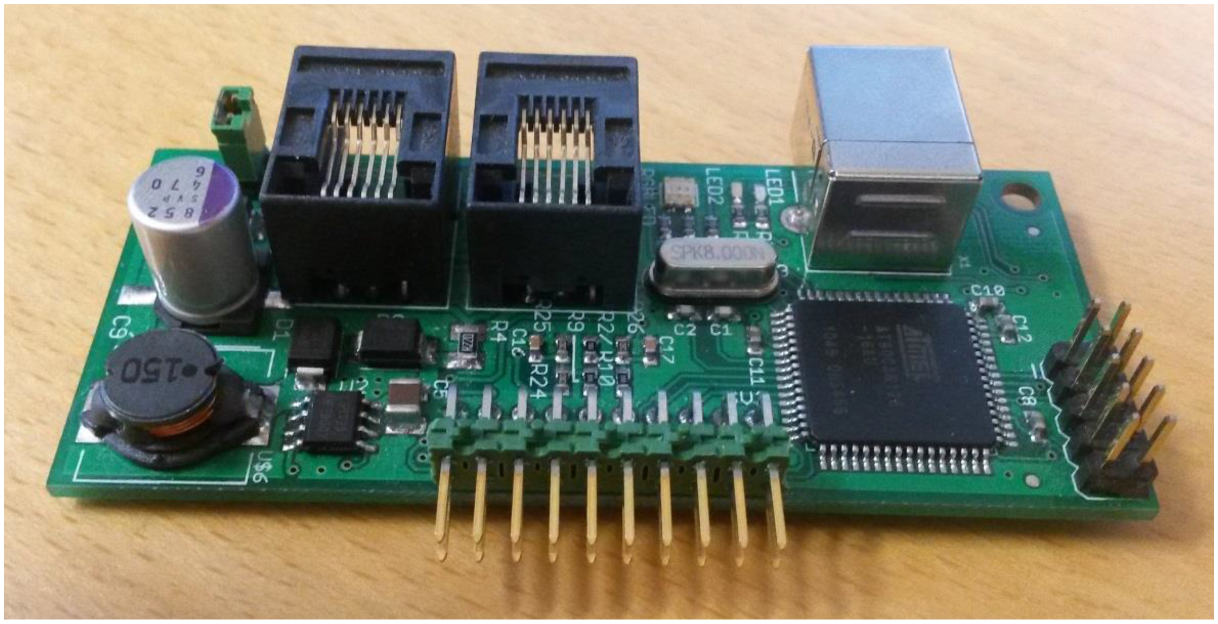


Figure 1.6: Universal Module

all the control-units. The main battery had a nominal voltage of 46.2V, and the solar panels were set to give an output voltage of 50.5V. This was because 50.5V was the preferred voltage to charge the batteries with. In order to supply the system in the car with 24V, a step down converter was used to transform the battery voltage to 24V. This was then passed through a fuse-board and made available on the CAN-bus and to the other modules that needed 24V. The front-module for example had its own 24V line instead of relying just on the CAN-bus for power. This was because the front-module controlled the horn and the fans which required too much power to safely transmit through the thin RJ11 cables. Additionally, if the horn or fans were to somehow blow the fuse in the fuse-board, it would only cause the front-module to stop working, and not the entire system of the car since that would be on a different fuse.

Emergency Circuit

As well as fuses on the accessory power circuit, there were also safety features implemented in the main power lines to the motor. Firstly, the battery itself had a built-in battery management system (BMS) which monitored each cell in the battery both during charging and usage. The BMS would isolate the battery if an unsafe condition occurred within a single cell, for example too high or low voltage, or too much current being drawn. The battery could also be isolated with two different emergency stop buttons, and a thermal fuse if the temperature in the motor-room were to pass 77°C. One stop button was located inside the car available to the driver, and one button on the outside so that anyone could easily turn off the entire system if something were to go wrong.

3G Module

The 3G module was able to transmit information from the vehicle using a mobile network. It required a SIM-card with available data-traffic, and could be used to transmit telemetry data such as speed, location, power usage, etc. Although the hardware had been completed for last years competition, there was no time to implement a working solution, so it was not used during the race.

GPS, IMU, SD-module

This module contained a GPS receiver, a 9-DOF IMU, and an SD-card slot which could be used for logging data. The module used the GPS receiver and transmitted the cars location on the CAN-bus so that any other node could use this information. The IMU was planned to measure the cars acceleration, and could also have been used to implement some INS system, but was unfortunately not used. The SD-slot was also not used, as there was not enough time to perform extensive testing which required logging of large amounts of data.

Display Module

The display module read relevant information from the CAN-bus, such as speed and torque reference for the motor. The information was displayed on a small LCD screen for the driver.

Steering-wheel module

This module read input from the driver, through a set of buttons located on the steering-wheel. A joystick controlled the desired torque, one button controlled the horn, and one button controlled the fans. A few extra buttons were placed also on the steering-wheel, so that extra functionality could easily be added.

Front Module

As previously mentioned, the front module controlled the horn and the fans. A special node was required to perform these tasks, as the fans and the horn required a relatively large amount of power. The board consisted mainly of two relays to turn the horn and fans on and off.

Speed sensor module

This module accurately read the angular speed of the rear-wheel of the car and periodically sent this information on the CAN-bus. This was done by placing 4 strong magnets equally spaced around the wheel and measuring their passing with a hall effect sensor.

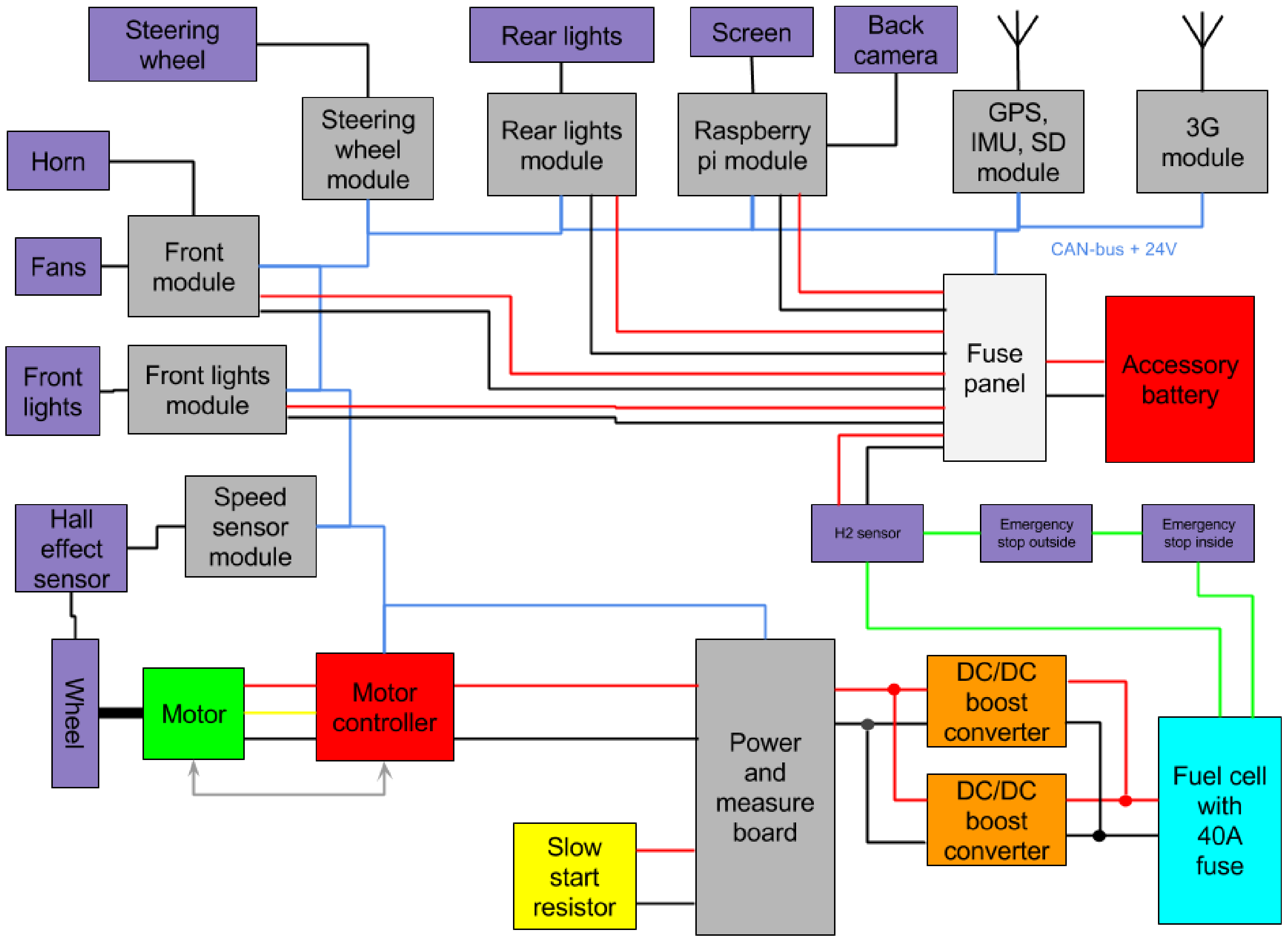


Figure 1.7: Block diagram, Electrical System - Urban Concept [7]

1.1.4 Urban Concept Electrical

A similar block diagram was created for the planned system of the Urban Concept vehicle. The main difference was the main power supply, which in this case was a hydrogen fuel cell. As opposed to the Prototype vehicle, the control system in the Urban Concept car could be powered by an accessory battery. This was in fact required, since the hydrogen sensor connected to the emergency stop system in the car required external power. It would be unwise to have this powered by the fuel-cell, in case it stopped working and leaking hydrogen at the same time. Safety was of course a very important factor in this competition, and since hydrogen is highly explosive, safety features were a high priority.

1.1.5 Tasks

As the author of this paper was part of the 2014 team, there was already considerable knowledge about the system and the required work. Ole Bauck and myself were put in charge of developing the complete electrical system of both vehicles. A list was created to highlight the desired changes in the Prototype, and what had to be created for the Urban Concept vehicle. Based on the previous years experiences, it was decided to create a new motor controller since the old one used too much power. The competition is after all in efficiency, and we felt that increasing the efficiency of the motor and motor controller would play a deciding role. In standby mode, the old motor controller used approximately 10 W, which we felt we could improve. Additionally, the steering wheel, the display module, and the front module could all be improved. We quickly realized that to perform all the changes we wished, we would need additional help. More team members were recruited, and set to perform specific tasks. The motor and fuel cell were also set to be handled by dedicated team members since these tasks would require a lot of attention. Exchange student Marius Hofman was in charge of the fuel cell, while exchange student Simon Fuchs was in charge of the motor and motor controller. Two additional members were added to the electrical team to help with improving the old system in the Prototype as well as developing new solutions for the Urban Concept - Simen Hexeberg, and Bjarne Kvæstad. Three main problems were identified with the old Urban Concept system. Primarily, there was no built in communication between the car and the crew. This made it difficult during the race when something unexpected happened to the vehicle. Ideally there should have been a system through which the driver and the crew could talk with each other, as well as the crew to be able to monitor the cars condition and performance. Another problem was that the driver didn't have information about the car readily available while driving. A small LCD-screen with a resolution of 4 x 20 characters was used in both vehicles, on which it was difficult to present all the required information to the driver. A third problem was that the mirrors for the driver were small and mounted on the inside of the vehicle to reduce air-resistance. This made it difficult for the driver to see behind the vehicle. During a race with numerous takeovers by other cars, this could potentially be hazardous. The first problem was handled by Ole Bauck, while the two latter problems were handled by myself. Unfortunately, the mirrors could not be replaced, but it was desired to supplement the mirrors with a large screen for the driver connected to a rear-view camera. The same screen could also be used to display relevant information for the driver. Additionally, it was thought of as a nice feature to be able to stream the video online so that anyone could follow the race in real time. I was also put in charge of upgrading the steering wheel module in both cars, as this was considered a part of the driver interface. The remaining issues of upgrading the front module was handled by Simen Hexeberg, and the speed module and lighting module for the Urban Concept vehicle was handled by Bjarne Kvæstad.

1.2 Previous Work

It was investigated to see whether a high resolution screen and a rear view camera had been used before in the Eco Marathon competition, or if other applicable solutions existed. All previous reports related to the Fuel Fighter project had been compiled throughout the years and was available at the Fuel Fighter office, as well as in an electronic database. It appeared that most previous vehicles had used small low-resolution LCD or OLED displays. In 2009, an LCD screen with 2 lines of 16 characters was used [6], while in 2010 a slightly larger screen with 4 lines of 20 characters was used [20]. This screen was also reused in 2011. In 2012, a hand-held Garmin GPS was embedded in the steering-wheel and used for giving the driver feedback regarding time and speed [3]. In 2013, a more advanced user interface was developed using a small touch-screen connected to a Raspberry Pi - a small single-board computer. This solution provided the driver with all the information he needed and appeared to be a good solution [4].

Regarding the rear-view camera, this had not previously been done by the DNV GL Fuel Fighter team. However, it was observed during the competition in 2014 that one team had developed an app for an android device connected to a camera so the driver could see behind the vehicle. In fact, using a smart device could potentially solve all the previously mentioned problems, as it can be used for communication, GPS measurements, and as a user interface for the driver. However, since most of these modules were already created, like the communication and GPS device, it would have been a waste to disregard a full semesters work and spend a lot of time to incorporate all of these solutions into one device. Since one of the main objectives with the Eco Marathon Competition is to drive innovation, it was felt creating the systems from scratch was more rewarding and instructive. Several consumer electronics products also existed that could be used a rear-view camera. There were several solutions that connected a camera to a screen both wirelessly and wired. However, these solutions are mostly closed-source, which meant it would be difficult to show anything else on the same screen without reverse-engineering the device.

1.3 Problem description

In light of the previous findings, it was decided by the author that a small touch screen with a user interface was to be developed. This screen would provide the driver with useful information, as well as take commands and perform various tasks. The screen would also show live video from a rear-view camera, giving the driver a clear view of cars passing from behind. The video feed from the camera was also to be streamed to a remote server, allowing anyone to see the race live. In addition to the touch screen in the Urban Concept, the screen system in the Prototype should be upgraded. Additionally, a system for handling driver input on the steering wheel in both cars were to be developed.

1.3.1 Objectives

The main objectives of this master thesis are

1. Develop hardware and software to create user-interface for driver

2. Enable the driver to see behind the vehicle with rear-view camera
3. Stream video from car to remote server
4. Upgrade screen and steering wheel in Prototype vehicle
5. Create steering wheel system in Urban Concept vehicle

1.4 Limitations

The aim in this thesis is not to create revolutionary new technologies to achieve its objectives, but rather try to utilize existing solutions. Various elements needed to complete the system - for example a high-resolution screen, camera, and telecommunication device to stream the video - will be bought to fit the needs. However, the process will strive to keep the system as tailored and customized as possible to truly fit the description of an embedded system.

1.5 Approach

The work in this project thesis will be mostly practical and little theoretical. The goal is to produce a working system that will be embedded in the DNV GL Fuel Fighter vehicles and used in the Eco Marathon competition in May 2015. The system will be tightly integrated with the rest of the vehicles functions, with many other components depending on its correct operation. There will therefore be strong collaboration with the mechanical team and the other members of the electrical team. It was established among the team that there would be weekly meetings between the members where relevant questions and information could be shared. In addition, all proposed systems, subsystems, and their components were reported to the teams systems engineer who ensured that everything went as planned. The systems engineer also performed a risk analysis on all components.

1.6 Structure of the Report

This report is structured into six chapters, where the last two are discussion and conclusion. Chapter one contained an introduction, giving the reader sufficient background information on the project that this thesis is a part of. The introduction also explained the existing systems, and the reason for choosing the task handled in this thesis. The universal module was thoroughly accounted for, as this is a vital building block of the components created in this thesis. Chapter two and three will discuss the screen solutions for the Urban Concept and Prototype vehicles, respectively. The largest focus in this thesis will be on the screen solution for the Urban Concept, as this was the primary goal of the thesis, and it also required the most work. Chapter four will discuss the steering-wheel systems for both cars. Each chapter will contain its own discussion, and all choices and results will be explained throughout the report for a better readability and understanding for the reader. Chapter five will contain a summarized discussion for all chapters, while the last chapter will be an overall conclusion for the report.

Chapter 2

Driver Interface - Urban Concept

In order to create a screen interface for the driver, with capabilities to display a video stream, there were many things to consider. Some of the important design parameters that had to be established are listed below.

Interface parameters

1. How large the screen should be
2. What should drive the screen/create content
3. How to connect the system to the CAN-bus
4. If it should be used as an accessory to or replacement of buttons on the steering-wheel
5. What type of camera to use, and how to display video on the screen
6. How to stream the video to a remote server

Inspired by some of the previously mentioned solutions, it was considered to embed the screen in the steering-wheel. However, this required some modification of the already existing steering-wheel, and it was also decided that having a screen that rotated with the steering wheel could be confusing and impractical. The screen would therefore be placed on the dashboard which allowed for quite a large screen. With this in mind, the focus was shifted towards finding a suited platform to process the required information and to create the content for the screen.

Initially it was desired to create a standalone low-level platform that could drive the screen, process input from a camera, and communicate with the CAN-network in the vehicle. However, the required hardware to process input from a camera as well as driving an LCD screen is complex and beyond the scope of this thesis. Creating such a system from scratch would have been unnecessary and other solutions were investigated instead. A few choices were looked into at first, and two solutions were considered.

2.1 Choosing an embedded platform

2.1.1 BeagleBone Black

A popular single-board computer for hobbyists is the BeagleBone Black. It is 86.4mm x 53.3mm and weighs 40 grams. It has a 1 GHz CPU, 512MB RAM, a micro-HDMI port, camera port, two CAN-bus ports, and one LCD-port. The BeagleBone seemed very attractive at first, as it had all the interfaces required, and it also supported numerous operating systems, like Android, Ubuntu, Debian, and Windows CE to mention a few. However, the CAN-port was only a controller which meant a CAN-transceiver was also needed to convert the signal from a single-ended one to the differential signal otherwise used in the vehicle. In addition, the existing CAN-bus in the vehicle used RJ11 connectors between nodes, meaning that special cables would have had to be made, and the BeagleBone could not have been placed arbitrarily on the CAN-bus. In other words, some additional hardware would have been required for the BeagleBone to work. The camera-port also turned out to be less useful than it initially seemed as there exist very few compatible cameras, which all require a *cape*, which is a PCB mounted on top of the BeagleBone.

2.1.2 Raspberry Pi 2

Another even more popular single-board computer is the Raspberry Pi - referred to as simply the Pi. The new edition, Raspberry Pi generation 2, was released February 2 2015, just in time for this thesis. The specifications on the new Pi were severely increased from the previous version, with for example the CPU being upgraded from a 700 MHz single core processor, to a 900 MHz quad-core processor. RAM was doubled to 1GB, it now had 4 USB ports instead of 2, and the

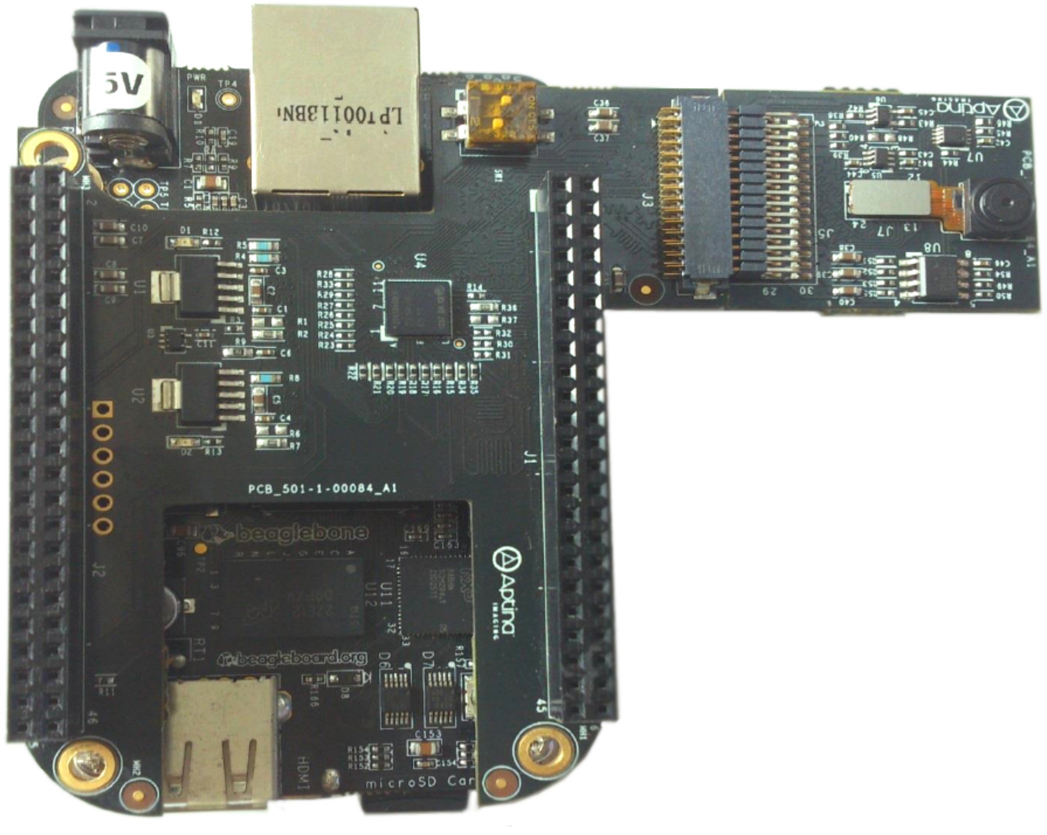


Figure 2.1: BeagleBone Black with camera cape [8]

number of available GPIO pins increased from 8 to 17. The physical size was 86.6 mm x 56.5 mm, the same as its predecessor, and roughly the same size as the BeagleBone. It came with serial display and camera interfaces, although the display interface couldn't be used at the time, since Broadcom who makes the processor had not made the specifications available. It was supposed to follow the MIPI standard, which has tried to make a standard interface between cameras, displays, and processors mainly in cellular phones. To this date, there was no official driver supplied by the Raspberry Pi foundation, and to make a generic LCD screen work with the Pi would have been a time consuming task. However, the Pi did have an impressively powerful integrated GPU able to decode 1080p video streams. It was also able to decode input from the camera connector and pass it on, bypassing the CPU altogether. The BeagleBone also has this capability, but its GPU was not nearly as powerful, and not able to handle 1080p.

Another positive thing about the Pi was that it seemed to have a large user community. The Pi had sold over 2 million units in late 2013, whereas the BeagleBone had barely passed 100,000

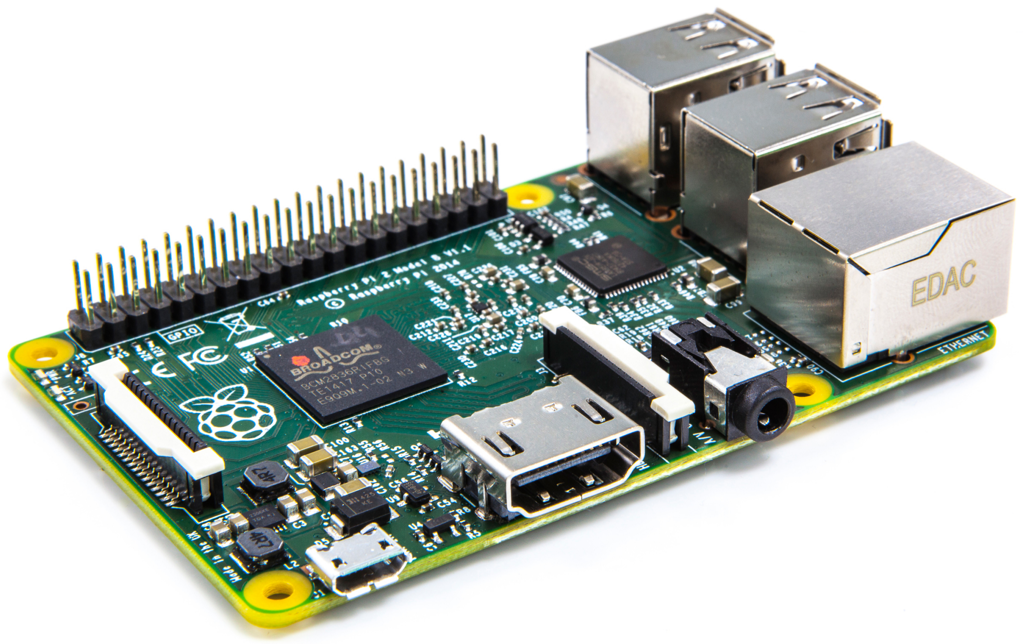


Figure 2.2: Raspberry Pi 2 [10]

units **(author?)** [9]. With a larger user base, there were much more resources and documentation available, in form of guides and tutorials on nearly every feature of the Raspberry Pi. The Pi also had its own tailored operating system - Raspbian - which was based heavily on Debian. Most packages and programs that worked with Debian therefore also worked with Raspbian. All references in this thesis to either the Pi or Raspberry Pi are in actuality references to the Raspberry Pi generation 2.

2.1.3 Comparison

The BeagleBone and Raspberry Pi 2 were quite similar in size and capabilities, and the choice was difficult. Although the BeagleBone had a few more features, like built-in CAN-controller and LCD-driver, these features didn't matter as much in this case. Additional hardware would have been required no matter what, and the choice therefore fell on the Pi, which seemed better suited to handle video processing, as well as having a more powerful central processor. Another important factor was the community around the two devices, with the Pi having more users. This had led to more examples and more peripheral devices being ported and made compatible with the Raspberry Pi.

2.2 Screen

Since the Raspberry Pi 2 was chosen as the platform, it somewhat limited the possible screens to be connected. Although it had a DSI connector for raw LCD displays, there were not yet any drivers available to drive an LCD screen. It had been reported that the Raspberry Pi foundation was working on a firmware upgrade that would enable LCD touch-screens, but at the time of writing this, this upgrade was not available. Apart from the DSI connector, the Pi had an HDMI port, and an analog port for composite video. A third possible interface was SPI, as there were several low-resolution LCD displays with this interface. However, these displays were not suited for video-rendering as their frame refresh rates were quite low. After searching for suitable screens online, the most prevalent interface seemed to be HDMI as opposed to composite. Most screens also seemed to come with large controller boards to support multiple inputs. These boards that usually supported VGA, HDMI/DVI and composite/component inputs were unnecessarily large and chunky, since only one input was sufficient. After some more searching, an ideal display was found on adafruit.com [1]. It was a 5-inch display with a resolution of 800x480, custom-made by Adafruit to be used with the Raspberry Pi in embedded projects. It had an HDMI port, a built in touch-controller with a USB connector. The HDMI-decoder was not able to scale the input, but this was no issue as the exact output resolution on the raspberry pi could be set. The display also had four mounting holes in each corner which made it easy to attach to the dashboard of the vehicle.

2.3 Camera

Since the Raspberry pi had a CSI port, the first choice for a camera was one that could be attached to this port. There were several different camera modules that could be used. Since the camera was to be used to provide the driver with a view of the rear of the car to reduce blind spots and get a better view, a wide field of vision as possible was desired. The standard camera module designed by the Raspberry foundation has a field of vision of about 56 degrees. Instead,

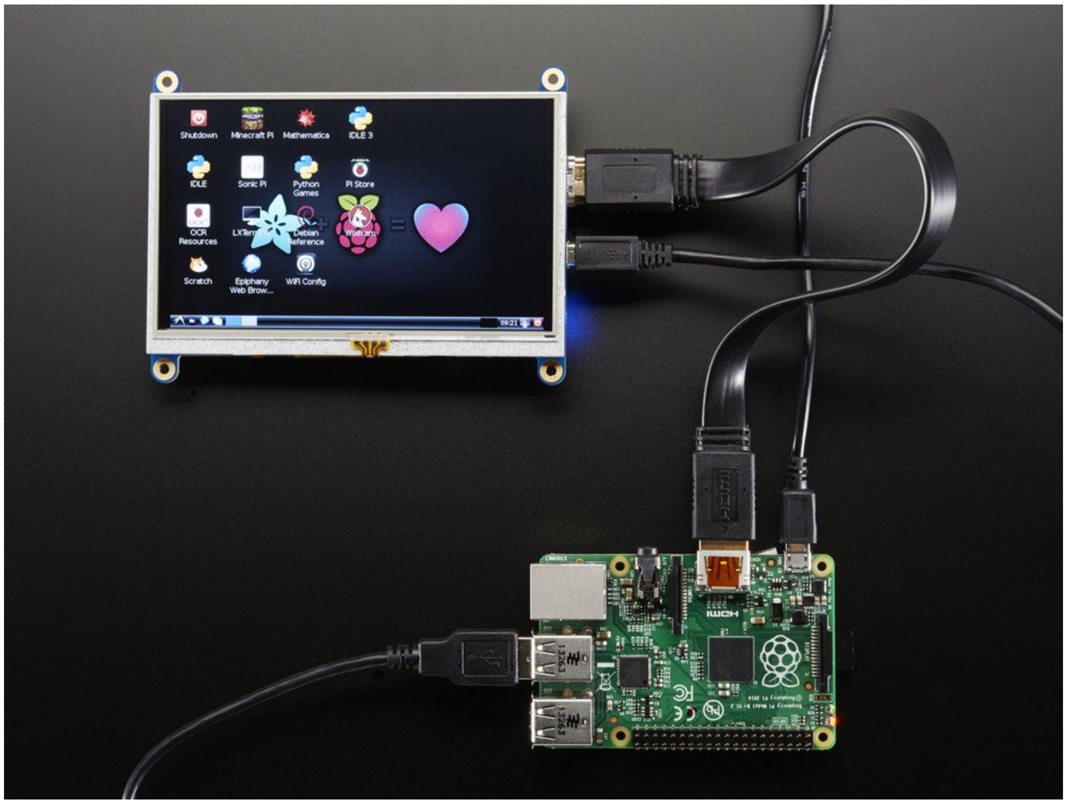


Figure 2.3: Display connected to Raspberry Pi [1]

an alternative module was purchased from Amazon, with a wide angle fish-eye lens. The field of vision on this module was not supplied, but it was assumed to be better than the standard module. The camera had 5 megapixels and the ability to record video at 1080p at 30 fps. [23]

2.4 3G Modem

In order to stream live video during the race a reliable high-speed connection had to be made between the Raspberry Pi in the car, and to the internet. This could be achieved with either a wifi connection or using a cellular modem connected to the Pi. From experience from last year, it was known that an open wireless network covered most of the paddocks where the crew would be working, as well as some of the race-track. However, the coverage was unlikely to extend to the entire track, and the internet connection was also terribly slow due to an extremely large amount of users. The wifi around the track was therefore not an option, and setting up our own network to cover the entire area seemed difficult. Instead a 3G USB modem was purchased for

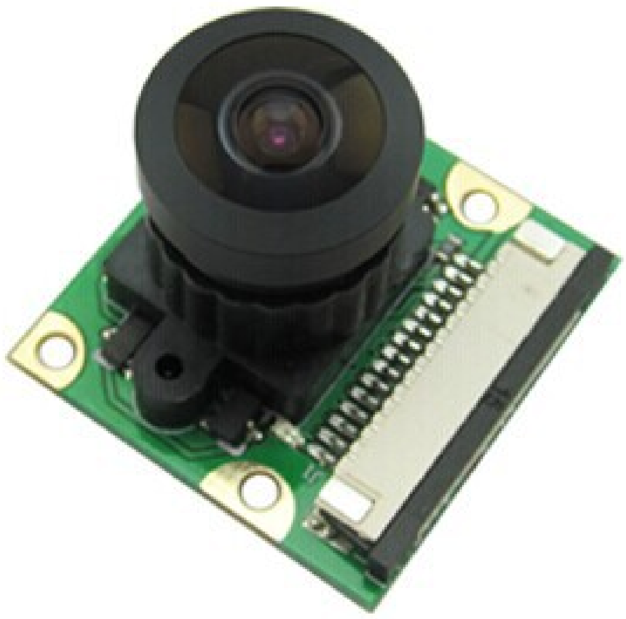


Figure 2.4: Fish eye camera module for Raspberry Pi [13]

the purpose of connecting the Pi to the internet. All it required was a SIM-card with a subscription for sufficient data traffic. It was also considered that the same 3G modem could be used for Ole Baucks telemetry unit, but it was decided that keeping those two systems separate was a better idea. The 3G network uses time division multiple access schemes, so the two cellular systems would in any case not interfere with each other A SIM-card adapter was also purchased to allow for micro-sim to be used in the modem.

Figure 2.5: 3G USB Modem [14]

2.5 HAT Hardware

With most of the hardware in place, an interface between the Raspberry Pi and the CAN-bus in the vehicle had to be created. The solution mentioned in chapter 1.2 had also created an interface between the Pi and the can-bus, but the solution was not a good one, so a new robust system was required. The Pi was luckily designed with the ability to attach shields in mind. The Pi foundation had even created a specification for a standard shield, also called a HAT (Hardware Attached on Top). This specification served two main purposes. Mainly, it was a mechanical spec, which outlined where holes for the DSI and CSI ports were, which would otherwise be blocked by the HAT. However, it also stated that to conform to the standard, the HAT must contain EEPROM with stored information about the HAT. This information was read by the Pi during boot, and since it was able to identify the hardware on top, it could also load the appropriate drivers and set the GPIO pins accordingly. However, since the HAT in this project was most likely not to be used commercially, and only on a single Raspberry Pi in the DNV GL FuelFighter Urban Concept vehicle, this identification scheme was strictly not necessary. It was still decided to conform to the standard when developing the HAT as far as possible.

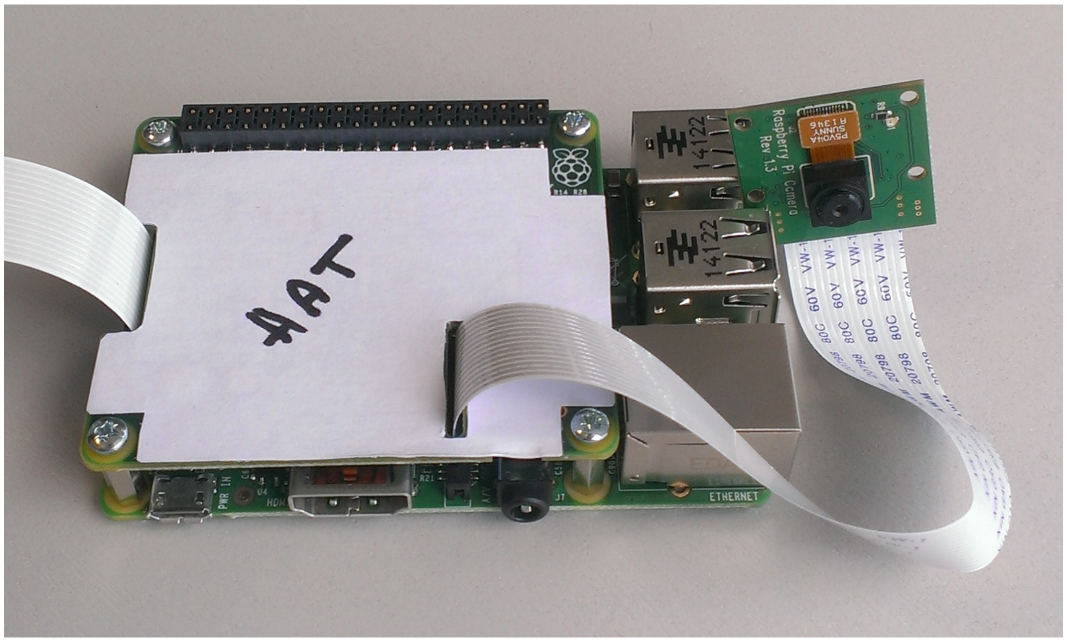


Figure 2.6: Raspberry Pi 2 with HAT [2]

2.5.1 Schematic

The purpose of the HAT was to connect the Raspberry Pi with the CAN-bus in the vehicle. By designing the HAT similar to the universal module, with two RJ11-ports, it would mean that the device could be placed anywhere on the bus and the system would remain flexible. It would appear as if just another universal module was connected. Most of the schematic for the universal module was therefore used as inspiration for the HAT.

CAN Controller

Since the HAT's primary function was to act as a gateway between the Raspberry Pi and the CAN bus, it needed a CAN-transceiver and CAN-controller. An apparent choice for the transceiver was the popular MCP2551, the same one used in the universal modules that the HAT would be communicating with. It implemented the ISO 11898 standard, which defines the physical layer and the link layer of CAN for use in real-time applications in road vehicles. As for the CAN-controller, any standalone controller could be used with for example an SPI interface that the Raspberry could control. However, it would be a good idea to use a controller with more computational power that could take some load off the Raspberry, as well as perform its own tasks. Additionally, considering the amount of different CAN-messages being transmitted, at high frequencies, it would be wise to use a controller that could handle all the different messages. The MCP2515 CAN controller for example only has two receive buffers, and only 3 different filters can be applied to each buffer. An interrupt would be generated upon reception of a CAN message, but to evaluate the contents would require an SPI transfer by the Pi. The at90can128 as was used on the universal modules seemed like a better solution, as a more flexible message management could be set up. The at90can128 had 15 receive and transmit buffers that each had their own mask and filter on incoming data. The mcu could then evaluate the data and determine if it was necessary to pass it on to the Pi. For example, the speed-sensor in the vehicle transmitted the current speed of the vehicle on the CAN-bus at a rate of 20 Hz, which was required for the motor-controller to work best as possible. However, the driver did not need to be updated 20 times per second about how fast he or she was driving. The MCU could instead filter out most of the speed messages, and then once a second transmit an average or the latest received speed-measurement to the Pi. As well as acting as a pure CAN-controller, the HAT itself could also be programmed to set LEDs to indicate its current status.

Powering

Considering that the universal module could be powered through the CAN-bus, it was investigated whether this could also power the Pi. Normally, the Pi is powered through its USB connector, but the GPIO header suggested that it could also be powered through the +5v pins. Unfortunately, despite the Raspberry Pi foundation being a charity organization aimed to promote computer science, they had not published the full schematics of the Raspberry Pi. However, the HAT specification made it clear that the Pi could indeed be powered through the +5V pin on the GPIO header. A crude backpowering scheme was published in the specifications, as shown in figure 2.7.

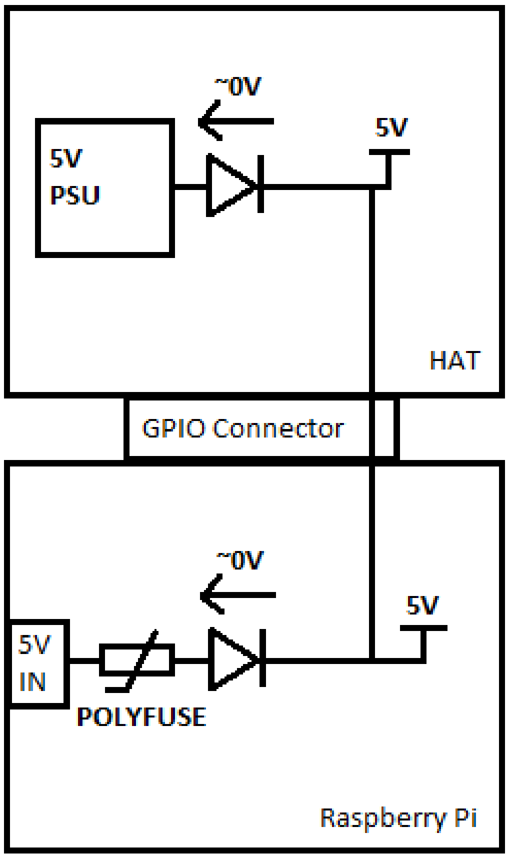


Figure 2.7: Raspberry Pi 2 backpowering [21]

The +5V pin connected straight to the 5V rail on the Pi, bypassing the onboard fuse. It was suggested to place a protection diode on the HAT so no current could flow from the Pi and back to the Hat if the Pi was being powered by USB at the same time. Since the universal module was designed to provide a 5V signal, this could be applied to power the Raspberry Pi. One question that remained was whether the entire module could be powered through the CAN-bus. The RJ11 wires weren't designed to carry large currents, so having a separate power supply for the Pi module was desired. Additionally, since the Pi took quite a long time to boot, having it powered by the CAN-bus was undesirable because when other modules were removed from the bus, the power-line was also raptured, causing it to shut down. Having the Pi on a separate line also meant that it would have its own fuse on the fuse board, so if some of the other modules would short circuit or draw too much power, the Pi would still be running. The Pi also has a 3v3 output pin on the GPIO header, and it was investigated whether this could power the microcontroller on the HAT, but Pi specifications said that max 50mA should be drawn from this pin. This was deemed insufficient to drive the microcontroller, the can-transceiver, and the two LEDs on the HAT. The can-transceiver alone had a peak current consumption of 75 mA during transmission [25]. This only strengthened the idea of using the powering circuit of the universal module on the HAT, which could deliver 200mA at 3.3V.

ID EEPROM

As stated in the HAT specification, the HAT had to contain EEPROM, connected to two dedicated I2C pins on the Raspberry Pi. The type of EEPROM was strictly specified, and there was also a recommended circuit diagram to follow. The requirements are reproduced below [27].

1. 24Cxx type 3.3V I2C EEPROM must be used
2. The EEPROM must be of the 16-bit addressable type
3. Do not use 'paged' type EEPROMs where the I2C lower address bit(s) select the EEPROM page.
4. Only required to support 100kHz I2C mode.
5. Devices that perform I2C clock stretching are not supported.

6. Write protect pin must be supported and protect the entire device memory.

The EEPROM was supposed to hold information about the manufacturer of the HAT, GPIO setup, and a device tree. The device tree is detailed description of the HAT hardware so that the Pi can set itself up correctly during boot by loading necessary drivers and setting pins, etc.

GPIO

The commonly used GPIO library for the Raspberry Pi used specific pins for specific protocols. UART, SPI, and I2C was available on the GPIO header, but it was decided that it was sufficient to route only UART and SPI to the microcontroller on the HAT. Additionally, two of the pins had to be connected to the EEPROM in order to follow the HAT specification. An RG-LED was also connected to two of the Raspberry Pi pins so it could directly control these to indicate status.

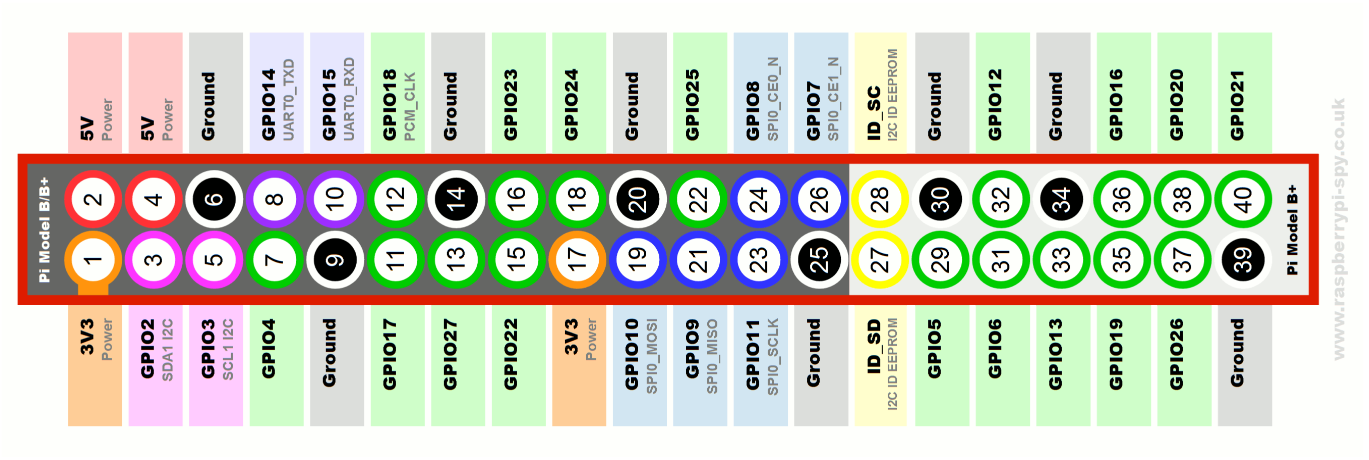


Figure 2.8: Raspberry Pi 2 GPIO header [24]

Complete Schematic

The finished schematic can be seen in the Appendix, figure B.1. The EAGLE-project for the HAT is appended in the attached zip-file. The CAM-job for creating the gerber files is also included in the attachment.

2.5.2 Layout

To comply with the HAT specification, the mechanical drawing released by the Pi foundation was used to create the layout of the HAT. The drawing can be seen in figure 2.9. The minimum requirements to comply to the standard are listed in table 2.5.2.

1. Board must be 65x56.5mm
2. Board must have 3mm radius corners
3. Board must have 4 mounting holes as per mechanical drawing
4. Board mounting holes must follow mounting hole specification
5. Board must have full 40-pin GPIO connector

Further recommendations were to add slots for the camera and display cables as per the drawing, but this was not required to comply to the standard. Since the board was a fixed size, with relatively few components, there was no point in going to great lengths to optimize the layout. Additionally, the distance between the Pi and the HAT when mounted was only 11mm so all components were kept on the top layer to avoid components crashing with the Pi. This also made the process of soldering the board less complex as all components could be placed on the same side and placed in a reflow oven. The finished layout can be seen in the Appendix, figure B.2.

2.5.3 Production and Mounting

The boards were ordered online from a PCB manufacturer in China. The production specifications of the manufacturer were considered when making the layout, such as minimum track width and hole diameter. The quality of the finished boards was excellent. The components were mounted using a reflow oven for all surface mount components, and all through-hole components were hand soldered. The author had never used a reflow oven before, but by following a tutorial for the LPKF ProtoFlow found in the D-block basement on campus it turned out to be quite easy. A standard reflow profile was followed, except the reflow time was extended. The reflow oven had a function to extend the time of the current phase, and it could be seen during

Table 2.1: Reflow parameters

Phase	Temperature (°C)	Time (seconds)
Pre-heat	150	150
Soak	150	90
Reflow	260	80
Cool-down	-	120

the last phase that not all the pads had properly reflowed. The final parameters that were used can be seen in table 2.1.

2.5.4 Result

The finished board was mounted on the Raspberry Pi by simply connecting the two pin-headers. To make the system even more rigid, the HAT was attached to the Pi using the four mounting holes on both PCBs. The mounting holes on the Pi were an untraditional size - 2.5mm - so special bolts, spacers, and nuts had to be purchased. The distance between the two boards was 11mm, and the only spacers available were 2.5x10m and 2.5x12mm. A set of 12mm spacers were bought, and then filed down to be 11mm. Both the spacers and nuts were nylon, but 2.5mm bolts in nylon was not found, so normal steel-bolts were used. Because steel is conductive, it was double checked that the bolts didn't short circuit anything they weren't supposed to. The result can be seen in figure 2.10. Although it looked very good, there were a few issues that had to be looked into, as explained below.

Camera Cable Slot

Firstly, the camera-cable slot seemed to be off by about 1mm, causing the cable to get stuck in the slot, damaging it slightly. This was fixed by filing off the edge of the slot with a small rounded needle file. The cause of this error could not be determined. The PCB was measured and found to match the mechanical specifications exactly. The Pi foundation was contacted to see if the specification was wrong, but they insisted it was correct. No more effort was put into this matter, as it was simple to correct.

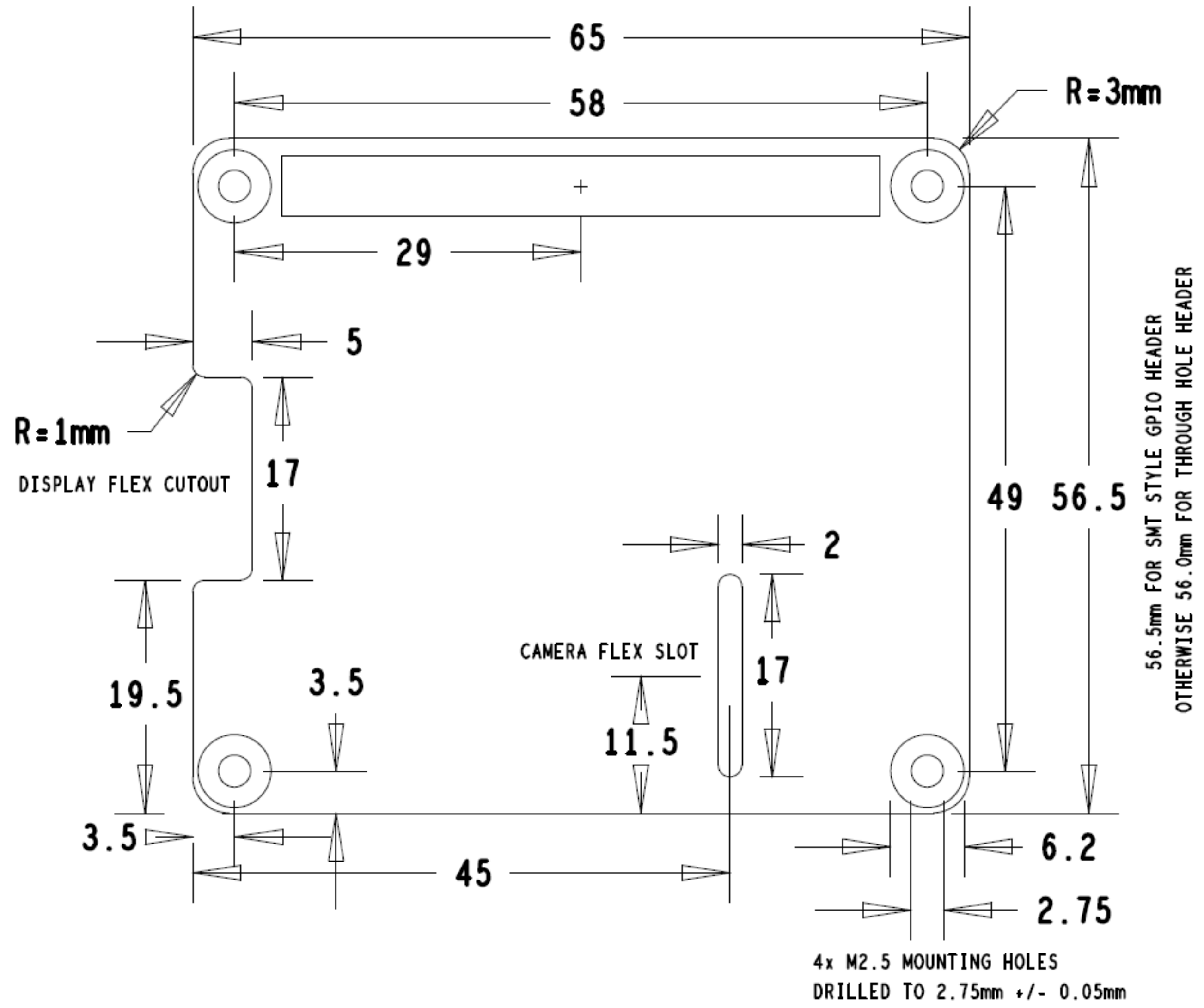


Figure 2.9: Raspberry Pi HAT mechanical drawing [22]

5V Regulator

A more important issue was with the 5V regulator. The schematic for the regulator can be seen in 2.12. The voltage regulator will regulate the OUT-pin so that a 1.23V potential is maintained at FB. Using Ohm's law it is easy to calculate the expected output voltage given the values of the two resistors R1 and R2.

$$V_{OUT} = (R_1 + R_2) * \frac{V_{FB}}{R_2} \tag{2.1}$$

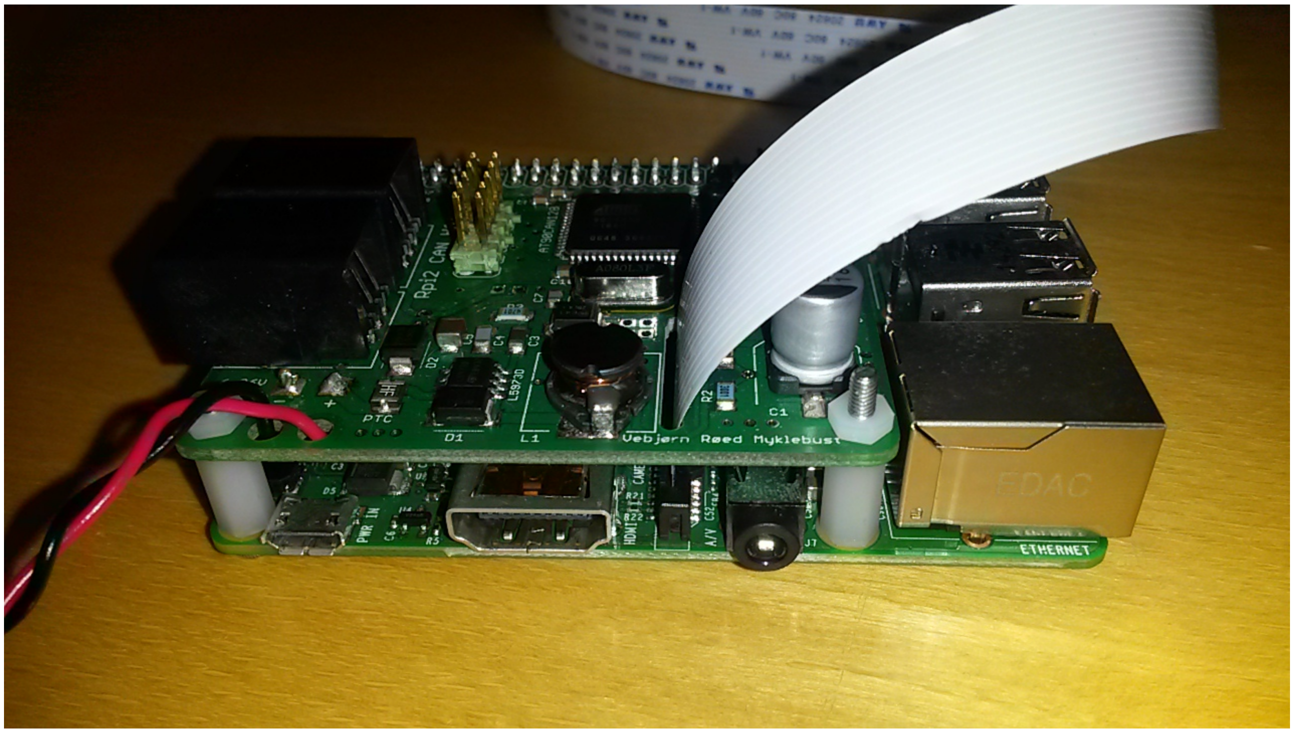


Figure 2.10: Raspberry Pi 2 with HAT attached

With $R_1 = 6800\Omega$, $R_2 = 2200\Omega$, and $V_{FB} = 1.23V$, we get $V_{OUT} \approx 5.03V$. This was good, except the diode, D5, had a voltage drop of about 0.35V with the Raspberry Pi connected. This meant that the supply voltage for the Pi was about 4.68V which was outside its recommended operating voltage of 4.85 - 5.25V. This could also be noted by observing the flickering power-LED on the Pi, which indicates whether the Pi has sufficient power or not. In fact, this voltage drop was an issue with all the universal modules as well, but had never caused a problem before. One solution would have been to remove the diode, and make sure never to power the Pi with a USB cable and through the HAT at the same time. This didn't seem like a good solution however, as removing safety features could quickly lead to problems. Instead, the voltage divider was modified so V_{OUT} was set slightly higher than 5V. Setting $R_1 = 10k\Omega$, $R_2 = 3k\Omega$, V_{OUT} was now instead 5.33V. Due to the voltage drop over the diode, the supply-voltage for the Pi was now 5.02V instead of 4.68. This still wasn't an ideal solution since the voltage heavily relied on the current drawn through the diode, but the Pi typically consumed about 700 mA [18], so the solution was deemed sufficient.

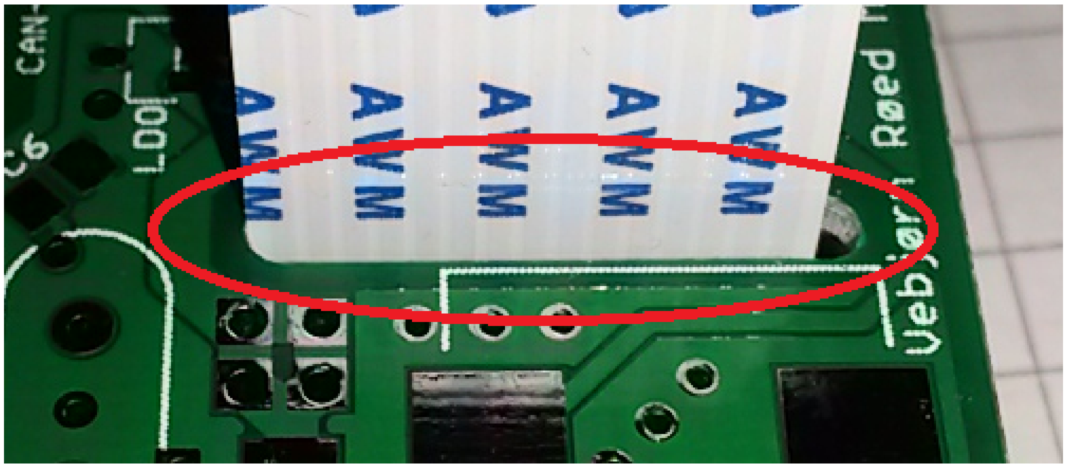


Figure 2.11: HAT Cable Slot

Reset button

It is often convenient to be able to reset microcontrollers, an issue that had been overlooked when designing the HAT. During testing it was quickly discovered that the need for a reset button was present. Performing a hard reset by cutting off the power was not a good solution, since this would cut power to the raspberry pi as well. Luckily, performing a soft reset wasn't too difficult, as the reset signal was found on the JTAG-header. Connecting a button between the reset-pin and ground became a quick and easy solution to the problem. During testing, the board could also be reset through Atmel Studio, while the HAT was connected to the JTAG.

2.5.5 Casing

Since the inside of the vehicle where the Pi was to be mounted was made of carbon fiber, it was a good idea to electrically shield the system. The bottom-side of the Raspberry Pi had a lot of exposed pads, and short circuiting all these would be a bad idea. A standard transparent case was purchased online, but it didn't quite fit the Pi with the custom HAT. Some of the components were sticking outside the case, such as the RJ11-ports, a capacitor, the CAN termination header, and the JTAG header. The case was simply cut out so the HAT fit, as can be seen in figure 2.13. The white arrows indicate the components that had to be be mode room for. Figure 2.14 offers another view of the finished case.

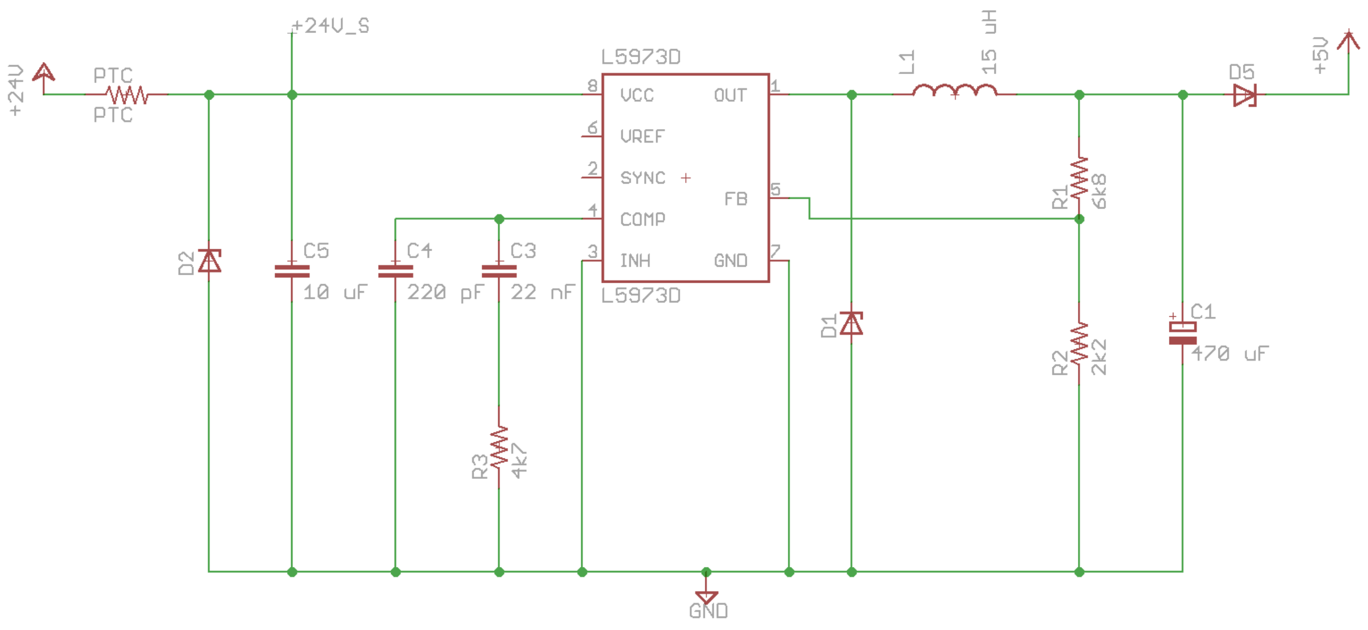


Figure 2.12: 5V regulator schematic

2.6 Hat Software

2.6.1 CAN message format

With the HAT functioning, it was time to start writing code for both the Raspberry and the HAT. Before this could happen, it was necessary to establish among the electrical team which IDs were going to be used for what messages in the CAN network. The ID in a CAN packet is used for multiple access purposes, since all the nodes share a common data bus. It is important that no nodes send packets with the same ID, and the packet with the lowest ID has a higher priority. After a quick discussion the IDs were assigned as shown in table 2.3. Since the input from the driver was deemed the most important information on the CAN bus, this was given a priority of 1. This was the highest possible priority meaning it would always be able to transmit. Other important messages included the brake sensor, and the speed sensor. IDs were assigned with ample space between them, in case other modules were to be added with a priority between two existing modules.

Since the data field of a CAN message was only 8 bytes, it was up to each member of the electrical team how the format of the messages they sent would be designed. The brake signal was only a single byte, with the LSB either 0 or 1. The speed however, was sent as $\frac{cm}{s}$ represented

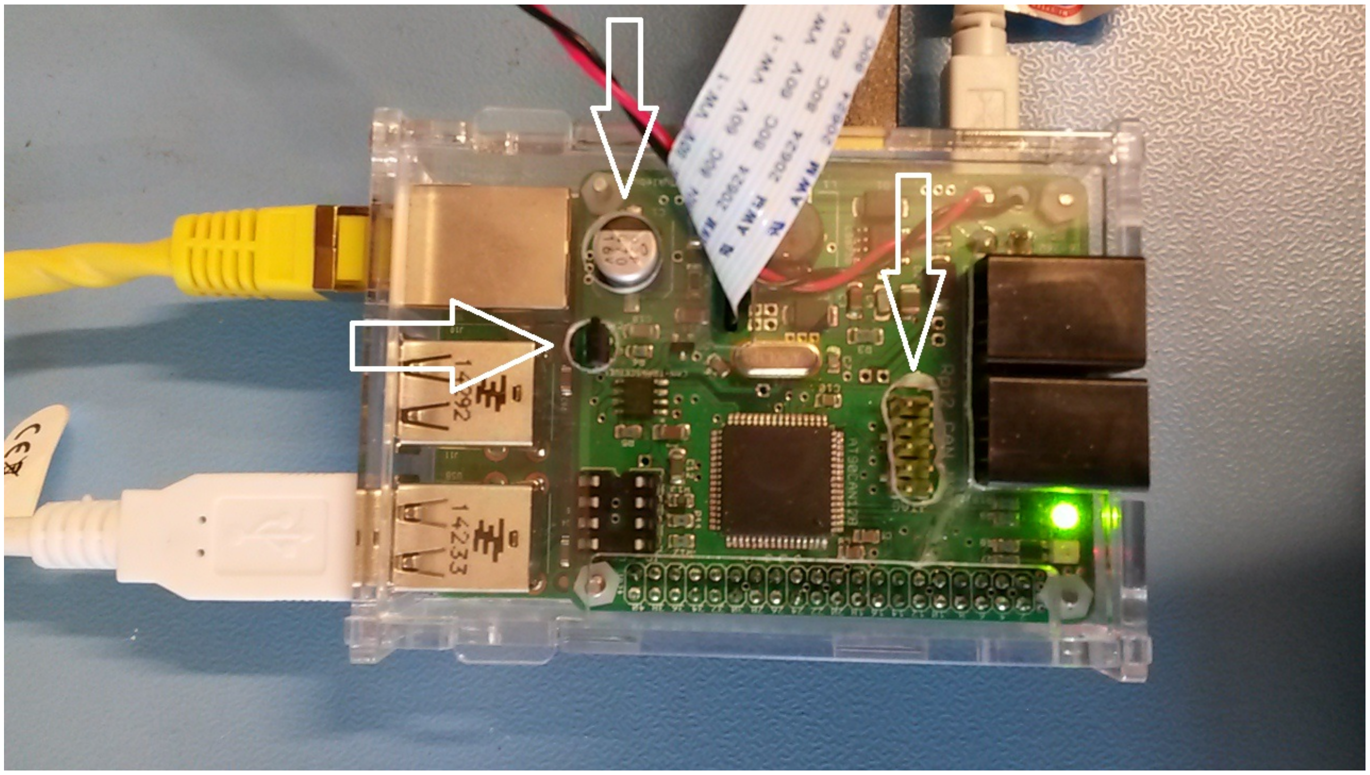


Figure 2.13: Raspberry Pi 2 Case, Top View

Message	ID
Steering-wheel data	0x01
Brake-signal	0x05
Speed	0x0A
GPS-data	0x14
Power-measurments	0x15

Table 2.3: CAN message IDs

by an unsigned 16-bit integer, with the most significant byte sent first.

2.6.2 Code

It was initially planned to communicate between the Pi and the at90can128 using SPI, but SPI uses a master-slave architecture. Among the Pi and the HAT there was no clear master role. If the Pi wanted to transmit a message on CAN it should tell the HAT to immediately do so, but if the HAT received a CAN message it should immediately forward it to the Pi without waiting for the Pi telling it to do so. It was instead decided to simply use the UART-protocol between the two devices. This meant that each device could send asynchronously to the other, at full duplex [5].

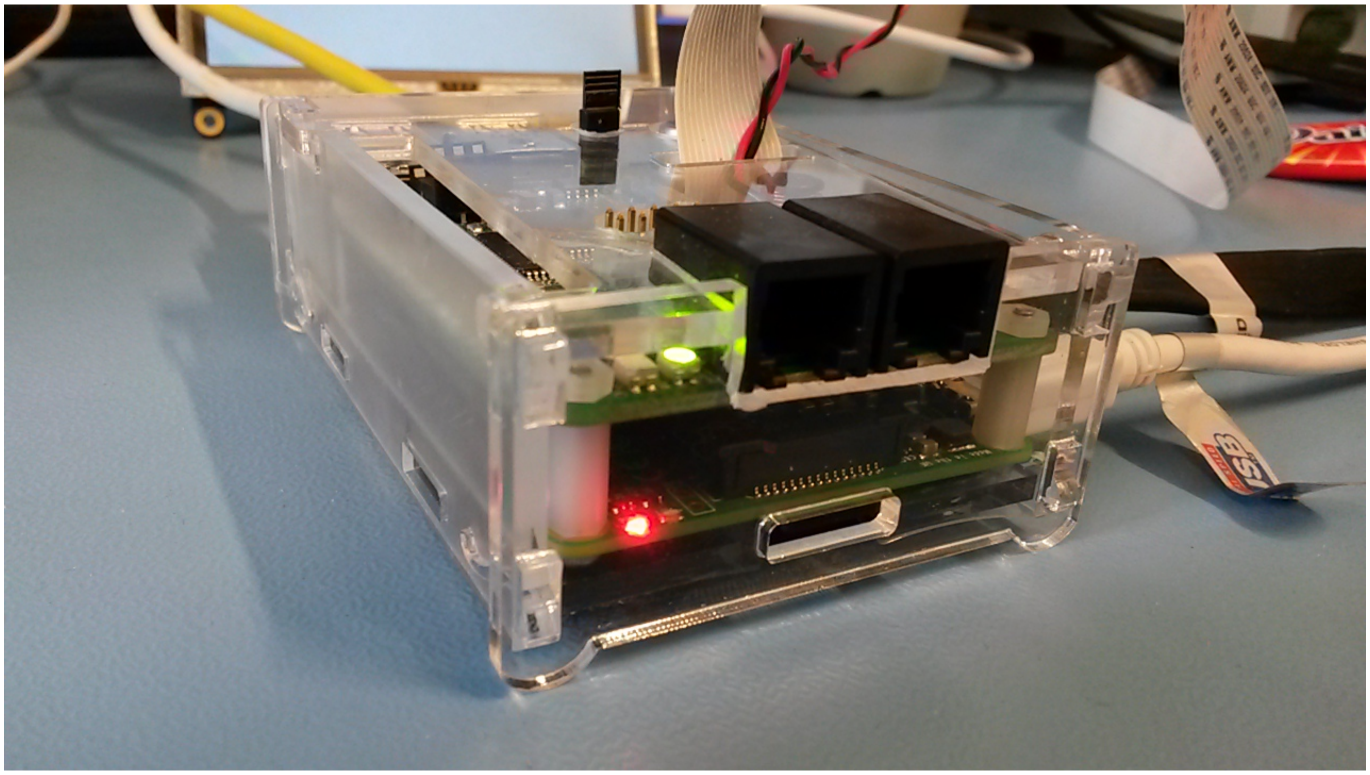


Figure 2.14: Raspberry Pi 2 Case, Front View

A certain protocol had to be devised on top of UART in order to transfer CAN packets between the MCU and the Pi. When the MCU received a CAN-packet it was set to transfer the packet to the Pi in the format seen in table 2.4. First, a character indicating start of message, 0xFE, was sent, then the ID as one byte, then another 0xFE character followed by the contents of the CAN-message. Lastly, the character 0xFF as well as a newline-character was sent to indicate end of transmission. The same protocol worked the other way, by the Pi sending a UART message to the HAT, and it would send the message on CAN. The Atmel Studio project for the Raspberry Pi HAT is included in the attached zip-file.

2.6.3 EEPROM

When booting the Pi with the HAT attached it became clear that some of the GPIO pins were floating, since the two LEDs connected to the Pi were illuminated, often at different and varying intensity. One of the purposes with the EEPROM was to avoid exactly this. However, due to lack of time, the setup of the EEPROM was not prioritized, and in the end it was unfortunately never done. Instead, a simple script was written on the Raspberry Pi that set all pins to low after the Pi had booted. How this was done is explained in section 2.7.2.

0xFE	ID	*0xFE*	DATA[0..7]	*0xFF*

Table 2.4: CAN packet format

2.7 Raspberry Pi

The Raspberry Pi was the central unit of the system and the majority of the time was spent on developing software for the Pi. The Pi was supposed to generate the content displayed on the screen for the driver, such as the video stream from the rear view camera and display diagnostics about the car. This meant developing a simple GUI, and unfortunately the author had no experience in doing this.

2.7.1 Operating system

As previously mentioned, the Raspberry Pi didn't come with an operating system, nor did the Pi foundation make one. However, there were several independent volunteer groups that were developing tailored operating systems for the Pi. Among these was Raspbian the most popular. Raspbian was essentially Debian customized for the Pi. The latest version at the time was downloaded and loaded onto the Pi. Using a Windows-machine, the steps to do so are summarized below.

1. Download latest release from https://www.raspberrypi.org/downloads/
2. Download and install *Win32 Disk Imager*, or similar software
3. Format SD-card to FAT32
4. Flash Raspbian image file onto SD-card, using Win32 Disk Imager
5. Insert SD-card in Pi and boot

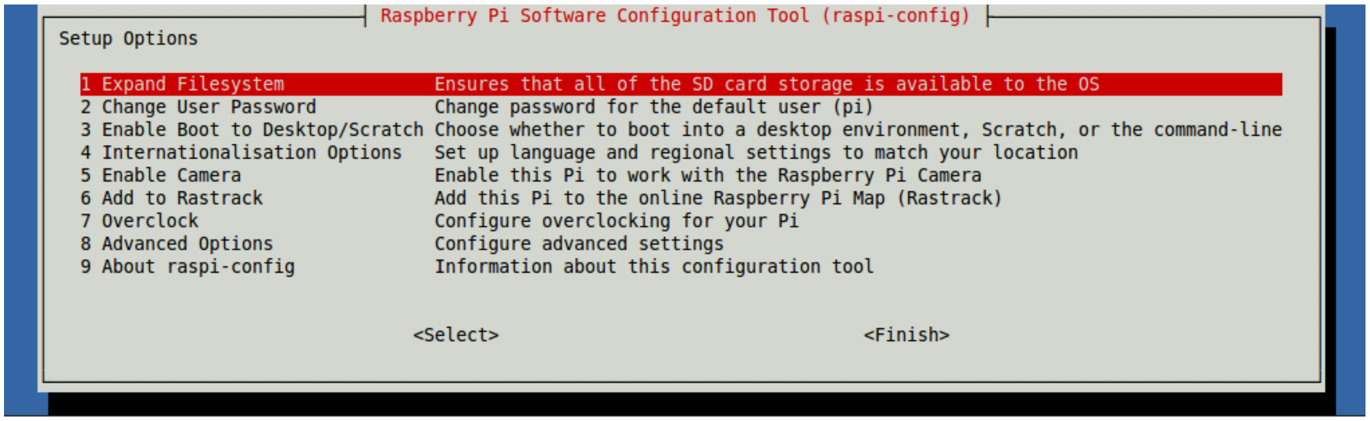
A 16GB SD-card was used, to allow for extra storage to record video or log data for example. Having a 16GB partition meant that a backup of the entire system could easily be done. In case of a corruption of the SD-card or something going wrong with the data, the SD-card could simply be swapped with a copy, and the system could be up and running within seconds. During development, a copy of the SD-card was made approximately every two weeks to have a recent fallback if something were to go wrong. Because a lot of different software had to be installed before the program could function, it was easier to simply load a pre-made image file than installing a fresh copy of Raspbian and then all the required packages. During the competition, a second SD card was kept handy with a duplicate of the current system, which could be swapped in an instant if something were to happen to SD-card in the Pi. Sadly, the image file was too large to append to the attached zip file, but it can be found on the Fuel Fighter network area - *\\webedit.ntnu.no\groups\ecomarathon2015\03 - Systems\Cybernetics\RaspberryPi*. To gain access to this area, contact vebjornr@stud.ntnu.no, or indeed anyone else on the Fuel Fighter 2015 team.

OS setup

Considering that the system would be running in an embedded environment, certain settings had be configured to ensure proper operation. First of all the output resolution had to be fixed so that it fit the screen. This was done by adding the following lines to the file */boot/config.txt*.

```
hdmi_group=2
hdmi_mode=1
hdmi_mode=87
hdmi_cvt 800 480 30 6 0 0 0
```

This set the output resolution to 800x480 pixels, with a refresh rate of 30 Hz. Secondly, when the vehicle itself was powered, it was desired that the interface would start automatically. The Raspberry Pi would boot on power, so an on-button or something similar was not necessary. How the application was set to run will be explained in section 2.7.3. In Raspbian there was a configuration tool where most important settings could be changed. This tool can be seen in figure 2.15. Here, the filesystem was expanded to include the entire SD-card for extra storage space. Additionally, the Pi was set to boot into a desktop environment, and the camera-port was enabled. Additionally, since the pi would be handling video, the memory split of the RAM was changed in favour of the GPU. The GPU and CPU shared the 1GB of RAM, and with the camera enabled, the GPU got 128 MB by default. This was instead increased to 512. The Pi was also configured to never disable the screen output or go into sleep-mode. Lastly, the SSH server on the pi was enabled so that remote connections were possible to allow for development.

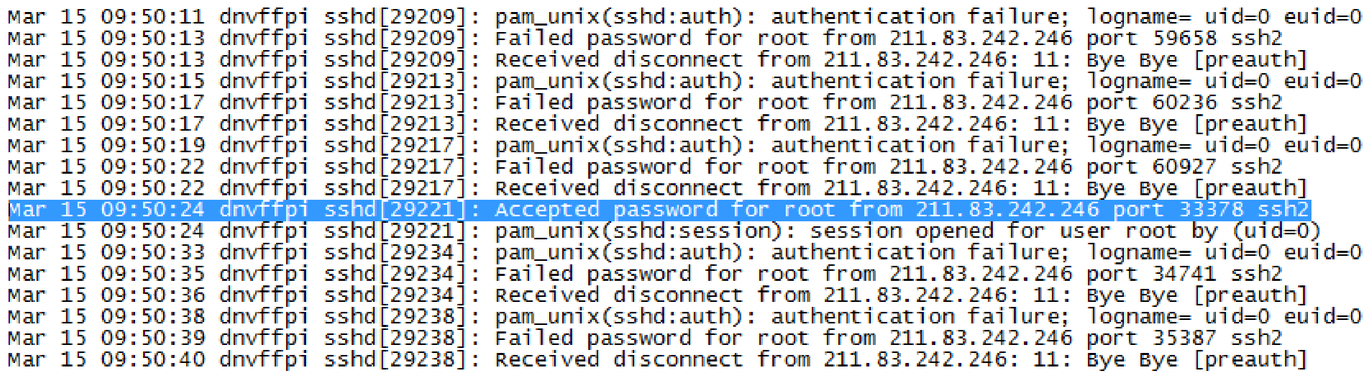


```
Raspberry Pi Software Configuration Tool (raspi-config)
Setup Options

  1 Expand Filesystem              Ensures that all of the SD card storage is available to the OS
  2 Change User Password           Change password for the default user (pi)
  3 Enable Boot to Desktop/Scratch Choose whether to boot into a desktop environment, Scratch, or the command-line
  4 Internationalisation Options   Set up language and regional settings to match your location
  5 Enable Camera                  Enable this Pi to work with the Raspberry Pi Camera
  6 Add to Rastrack                Add this Pi to the online Raspberry Pi Map (Rastrack)
  7 Overclock                      Configure overclocking for your Pi
  8 Advanced Options               Configure advanced settings
  9 About raspi-config             Information about this configuration tool

                      <Select>                                   <Finish>
```

Figure 2.15: Raspbian configuration screen

Security

During development, the Raspberry Pi was connected to the NTNU network through the ethernet port. This was done in order to transfer files to the filesystem, since all development was done on a desktop and then transferred by SFTP. SSH connections were also established to execute programs on the Pi. The problem with this setup was quickly discovered by the author, as there were suddenly an abundance of files in the Raspberry Pi filesystem that had not been put there by me. The author had naively assumed that the NTNU network was behind a firewall, and that none of my fellow students would be attempting to hack or perform malicious tasks on the Raspberry Pi. For this reason, the default password had not been changed. However, the Pi was open to the world wide web, with the default password "raspberry" for the default user "pi". The root password had also been set to something easy to remember, and not very difficult to crack. The result can be seen in figure 2.16. Raspbian kept a log of all system activity, and also remote login activity, found in log.auth in /var/logs/. This file was inspected, and it turned out that nearly every second a remote client tried to login to either the root user or the pi user. Occasionally, they succeeded in the brute-force attack, which is highlighted in figure 2.16.



```
Mar 15 09:50:11 dnvffpi sshd[29209]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
Mar 15 09:50:13 dnvffpi sshd[29209]: Failed password for root from 211.83.242.246 port 59658 ssh2
Mar 15 09:50:13 dnvffpi sshd[29209]: Received disconnect from 211.83.242.246: 11: Bye Bye [preauth]
Mar 15 09:50:15 dnvffpi sshd[29213]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
Mar 15 09:50:17 dnvffpi sshd[29213]: Failed password for root from 211.83.242.246 port 60236 ssh2
Mar 15 09:50:17 dnvffpi sshd[29213]: Received disconnect from 211.83.242.246: 11: Bye Bye [preauth]
Mar 15 09:50:19 dnvffpi sshd[29217]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
Mar 15 09:50:22 dnvffpi sshd[29217]: Failed password for root from 211.83.242.246 port 60927 ssh2
Mar 15 09:50:22 dnvffpi sshd[29217]: Received disconnect from 211.83.242.246: 11: Bye Bye [preauth]
Mar 15 09:50:24 dnvffpi sshd[29221]: Accepted password for root from 211.83.242.246 port 33378 ssh2
Mar 15 09:50:24 dnvffpi sshd[29221]: pam_unix(sshd:session): session opened for user root by (uid=0)
Mar 15 09:50:33 dnvffpi sshd[29234]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
Mar 15 09:50:35 dnvffpi sshd[29234]: Failed password for root from 211.83.242.246 port 34741 ssh2
Mar 15 09:50:36 dnvffpi sshd[29234]: Received disconnect from 211.83.242.246: 11: Bye Bye [preauth]
Mar 15 09:50:38 dnvffpi sshd[29238]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
Mar 15 09:50:39 dnvffpi sshd[29238]: Failed password for root from 211.83.242.246 port 35387 ssh2
Mar 15 09:50:40 dnvffpi sshd[29238]: Received disconnect from 211.83.242.246: 11: Bye Bye [preauth]
```

Figure 2.16: Excerpt from auth.log

A vast number of different IP-addresses had been systematically trying to login to the Pi for several weeks, and many had succeeded. All the IP-addresses originated for either China or Hong-Kong. After these findings, the SD-card was formatted and everything was reinstalled. Before connecting the Pi to the internet again, the user passwords were changed to something drastically harder to crack, and the default ports for SSH and SFTP were changed. Additionally, a software named fail2ban was installed, which monitored the log-files and banned all IP-addresses that showed malicious behaviour. After these steps, the system was safe, and no other breaches occurred

2.7.2 Interface application

As to the program that would run on the Pi, it had to be established how it was going to be made. A few options were investigated, and it appeared GTK+ and QT were among the most popular frameworks for creating GUI applications. Both were cross-platform compatible which meant that development and compilation could be done on a more powerful computer than the Pi itself. However, some features of the program such as the camera-interface would be specific only to the Pi, which meant setting up binaries for cross compilation would be difficult. An attempt was made to set up a cross compilation environment, but it quickly became apparent that it would be more work setting it up than it would be worth. After some research, it seemed that GTK+ was a light and well suited framework to achieve the desired functionality. GTK+ was available in both Python and C++ libraries. A lot of research also went into finding suited packages to achieve streaming of the video camera both to the screen and to a remote server.

Gstreamer had both of these functionalities, and was also available both for Python and C++. A few tutorials were found for Python, both for streaming and for making simple GUIs, which were of great help. The major elements of the application will be discussed and explained below.

GTK+

GTK+, short for GIMP Toolkit+, is a multi-platform toolkit used to create anything from simple GUIs to complete desktop environments and applications. It is free to use, and is subject to the GNU Lesser General Public License, which means anyone may incorporate the API into their own proprietary software for distribution.

Gstreamer

Gstreamer is an extensive multimedia framework written in C, but has APIs available for most programming languages and platforms. Gstreamer uses so called pipelines, where media is transported from a source to a sink. A pipeline is constructed by elements with input and output pads, and the connection between these pads create the pipeline. A source can be a file on disk, a tcp-connection, or a usb-camera. A typical sink can also be a file, another tcp-connection, or a monitor. An example is given in figure 2.17, where we see a source-file is sent to a demux element which splits the file into two parallell pipelines. One pipe is passed through an audio decoder and then to an audio-sink, and the other pipe goes through a video decoder and finally to a video sink. Elements can have different capabilities, or settings, which affect the outcome of the pipeline. Gstreamer uses the same GNU license as GTK+, and was encouraged to be used together with GTK+ applications, by the GNOME project. The latest binaries available in the Raspbian repository was gstreamer version 1.2, which was installed. After it was installed, the correct paths to the binaries had to be set by configuring three environment variables in Raspbian. This had to be done on each system boot, so the following three lines were added to ~/.profile for the pi user.

```
export GST_PLUGIN_PATH=/usr/local/lib/gstreamer-1.0/
export GST_OMX_CONFIG_DIR=/usr/local/etc/xdg/
export LD_LIBRARY_PATH=/usr/local/lib/
```

This is a file that is executed on each boot. Although the script was executed on boot, the variables were not present in the desktop environment after boot, so an additional fix had to be applied. The file /.xsession was modified to look like the following lines.

```
[ -r /etc/profile ] && source /etc/profile
[ -r ~/.profile ] && source ~/.profile
exec startlxde
```

When the xserver was started, this script would be loaded, which in turn loaded the required environment variables and finally started the desktop environment. The second line made sure the contents of ~/.profile was executed in the current shell, causing the variables to remain there.

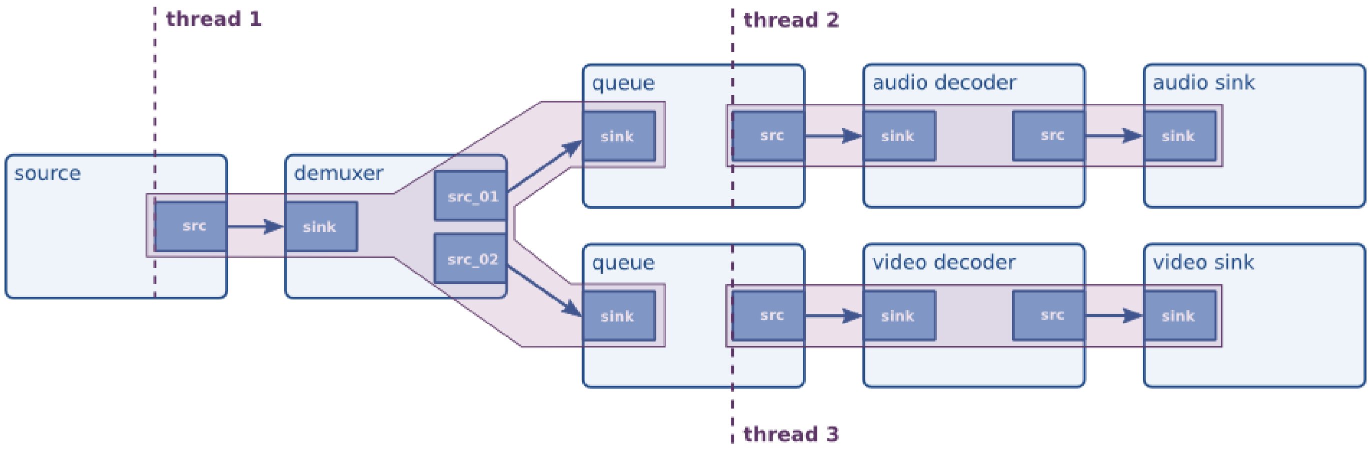


Figure 2.17: Gstreamer pipeline example [26]

RPIcamsrc

In order to use the camera connected to the CSI port on the Raspberry pi as a source for gstreamer, a plugin for this had to be installed. Luckily, someone had actually created this plugin - called rpicamsrc - and made it available on github [30]. The installation instructions in the readme were followed, and gstreamer could now use the raspberry pi camera as a source. The plugin had capabilities to flip the video horizontally and vertically, as well as adding effects and much more.

3G modem

In order to use the 3G modem, some additional software had to be installed on the Raspberry Pi. A guide was found online that explained the required steps [31]. First of all, the USB modem established a serial connection between the modem and the Raspberry Pi, over which the Internet Protocol would run. To accomplish this it was necessary to install a program called ppp (Point to Point Protocol). A program called Sakis3G was then installed. This program simplified the process of recognizing and configuring the SIM card in the USB modem, and then finally establishing an internet connection. All that was required was to confirm the Network Provider that the program guessed, enter the pin-number of the SIM card, and Sakis3G handled the rest. Sakis3G had to be run as root, otherwise there were some complications, but the program managed to set up a connection using my own SIM-card from Netcom. The connection was tested, and worked well. The following command was executed on the Raspberry Pi to connect to the internet.

```
sudo ./sakis3g connect
```

Serial Port

In order to use the serial port on the Raspberry Pi, there was a trick to perform in order to gain complete control over the port. The Raspbian Kernel uses the serial port by default during boot to output information, and it also allows a user to log in using the serial port. Additionally, the kernel uses the port to output debug messages. To disable this, the boot configurations had to be changed by altering the file */boot/cmdline.txt*. The serial port was found on /dev/ttyAMA0, and removing all references to *ttyAMA0* in this file released the control of the port from the kernel. The Raspberry Pi could now freely communicate with the at90can128 microcontroller over UART, without being interrupted by the kernel.

Enabling GPIO

In order to easily control the GPIO pins on the Broadcom CPU, some additional software was installed. A popular library for gaining control over the GPIO pins was the wiringpi library [12]. It is written in C++, but a wrapper for python and python3 was created by the online community Gadgetoid. The installation instructions were followed on their github site [19]. Using this library, it was now possible to turn on and off the LEDs on the HAT that were connected to the Raspberry Pi header. As previously explained, the EEPROM was unfortunately not set up to configure the GPIO, but the wiringpi library was used instead. The file ~/.profile was edited to contain the following lines:

```
gpio mode 8 out
gpio mode 9 out
gpio write 8 0
gpio write 9 0
```

These lines used the command line utility *gpio* that was installed with the wiringpi library. The parameters *mode 8 out* set the mode of pin 8 to be an output pin, while the parameters *write 8 0* wrote a logical 0 to pin 8. In other words, the LEDs connected to pin 8 and 9 were turned off.

2.7.3 Development and Test of program

It was decided to use Python to develop the application, because there were many user-friendly tutorials on each of the major libraries used in the program. Raspbian also came with python3 as well as some of the libraries required for GTK+.

After some testing it became apparent that there were issues with displaying the video from the camera on screen using GTK+. Ximagesink and xvideosink - two popular sinks for displaying video on screen using the linux X windows system - had known issues with gstreamer version 1.2. It was found that version 1.4 was indeed available, but would have to compiled from source. A script was found online where someone had installed gstreamer 1.2 from source on the raspberry pi [11]. The script was outdated, but by changing the urls to the newest source files it worked without too much hassle. It took approximately 5 hours to compile and install the latest version on the Raspberry. With the latest version installed everything worked as expected. The modified script has been included as an attachment.

The program was developed on a Windows desktop computer and the code was uploaded to the pi using an SFTP client. The program was launched by starting the process in an SSH session. In order to do this, the environment variable DISPLAY had to set in the SSH session so that the python script would direct its output to the correct display. This was simply done with the following command

```
export DISPLAY=:0
```

In order to automatically launch the application on boot, the following line was added to */etc/xdg/lxsession/LXDE/autostart*:

```
@/usr/bin/python3 /home/pi/car.py
```

The code was stored in a Dropbox-folder which automatically kept the code backed up and also served as version control. An adjustable power supply was used to power the Raspberry Pi, as to simulate the power supply in the car. The voltage was set to 24V. An anti-static mat was used in the workplace to make sure no components were damaged due to static electricity. The Mat was grounded to the main sockets, and a wristband connected to the mat was worn whenever possible.

CAN communication

One problem with using the serial port for communicating between the Pi and the hat was that only one application could open the port at a time. For this simple application it wouldn't be a problem, but it can be imagined that in the future there will be multiple applications running on the Pi that wish to communicate with the CAN-bus. To overcome this, a server was created that gained control over the serial port and that any other application could connect to. The server was set up to use Unix Domain Sockets, a type of socket meant for inter-process communication. They behave just as TCP or UDP sockets, except they use the name-space of the filesystem to address eachother instead of IP-addresses. All communication essentially goes through a file on the file-system. Another advantage with this approach was that together with the 3G modem,

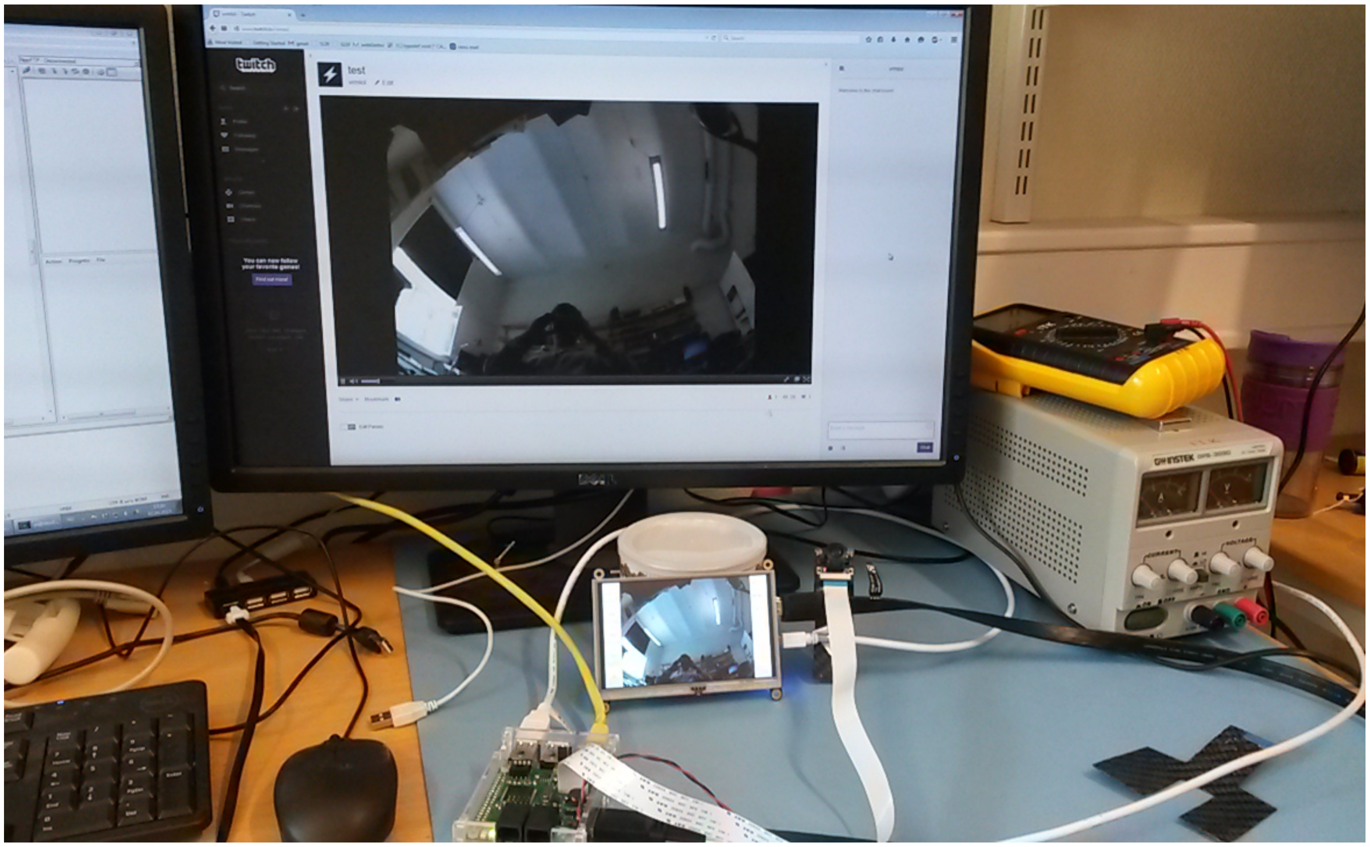


Figure 2.18: Development setup

the CAN-bus could easily be forwarded to the world wide web, essentially allowing for a remote wireless CAN-node. This program can be found in the attached zip file, as *car_server.py*.

GUI

The GUI was planned to allow the driver to select between a few simple dialog-windows, as well as perform simple functions with the car such as controlling the lights, fans, and handsfree system. A plan was also to create a map of the race-track on screen where the driver could see his location on the map in realtime, with the help of GPS data. A window with system diagnostics was also planned to show all available data of the car, such as fan, wiper, and light-status, as well as detailed information about the fuel cell stack. However, as time didn't permit the development of all these features, the screen was programmed to show the rear-view camera, as well as speed, torque reference, and time remaining of the race. The buttons were made as large as practically possible so it would be easy for the driver to press them, although in the final product they sadly had no function.

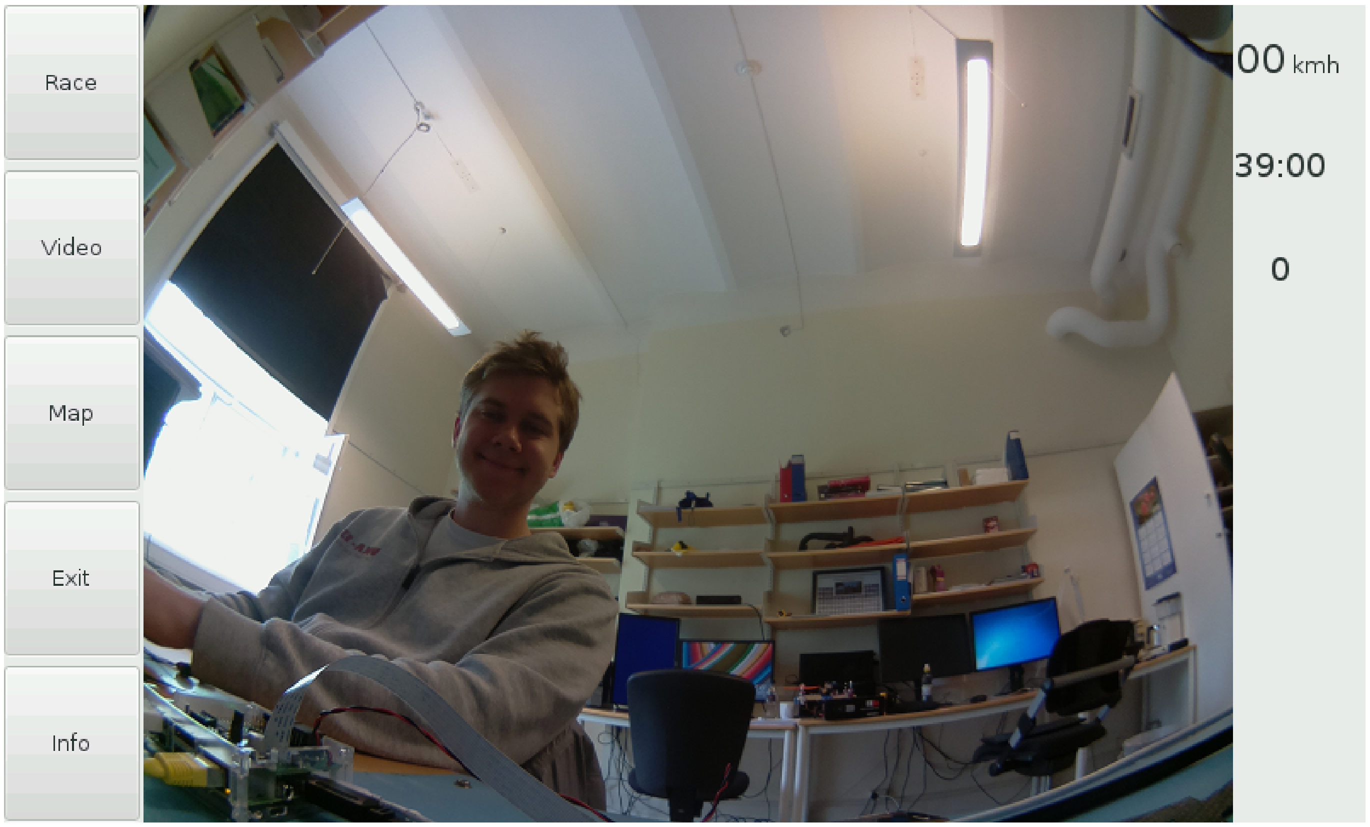


Figure 2.19: Screenshot of driver interface in development environment

Creating the user interface turned out to be a little more difficult than anticipated. Although there were a few examples on creating simple programs with GTK+ in python, the actual documentation was lacking. To create elements that were not in the examples, the C++ documentation had to be consulted and educated guesses had to be made in order to find the python equivalent. A challenge with the GUI application in GTK+ was that everything was event-driven through interactions with the screen elements, such as buttons, etc. The main control flow was *Initialize* -> *Start GTK+ main function* -> *wait for interrupts*. The initialization phase consisted of creating the window and all the widgets in it, as well as linking callback functions to events on each widget. When the main function *Gtk.main()* started, the program was running and the callbacks were active. The main function never returned, and the program could only be stopped by calling *Gtk.main_quit()*. The program also behaved like a client and connected to the serial port server to establish communication with the CAN-bus. However, in order to keep the client active in the background as well keeping the GUI alive, a second thread had to be spawned for the client process. This process then received information such as speed and torque reference from the driver, and updated this information on screen. However, it was quickly discovered that multithreading with GTK+ was very unstable. After some digging in the C++ documentation it was found that only the thread that *Gtk.main()* was running in could safely modify GUI elements. Any other thread would have to call a special GUI function - *GLib.idle_add(function)* - to queue other functions that could modify GUI elements. The desired function would then be safely interleaved with the main function in the main thread. During testing, the connection between the server and client suddenly started dropping, and the cause could not be found for this. To be safe, the main application was set up to open the serial port instead of relying on the server. The main file is found in the attachment as *car.py*.

Video Streaming

A few different solutions were looked into regarding streaming video from the Raspberry Pi. The first idea was using a udpsink in gstreamer - an element that allows streaming to a UDP socket - and simply stream the video to a remote server also running gstreamer. One large drawback with this method was that the receiving end had to also run gstreamer. Additionally, enabling anyone to see the stream would be difficult as there was no easy way to display the received stream. Instead, existing streaming websites like youtube.com and twitch.tv were considered. Youtube had a feature called Youtube Live, where live events could be shown in your personal channel. However, streaming video to Youtube Live using gstreamer proved quite difficult. Instead, the immensely popular streaming site twitch.tv was used. Twitch was primarily used to stream live gameplay of videogames, but they had for this reason made it very easy for anyone to display their personal video streams. Twitch was set up to accept incoming RTMP data, and gstreamer luckily had an rtmpsink element for video. All that was required was to direct the video pipeline into the rtmpsink, and configure it to point to the twitch.tv url, containing a user-unique identifier so that video would be displayed in the correct personal channel. A downside of using Twitch was that they had a built-in delay of about 20 seconds on all their streams. This wasn't a big problem though, as it wouldnt matter to users around the world watching the car going around the track if it was lagging by 20 seconds. Another problem with the video streaming was to construct the pipeline itself. Gstreamer was not a straight-forward tool to use, and especially not since it was poorly documented for python. The video stream had to be split in two parallell branches, where one was displayed on screen for the driver, and the other was piped into the rtmpsink. It also turned out that some of the pipeline elements had dynamic output pads, or in other words the output pad was only created when it had valid input. Linking the complete pipeline therefore had to be done during runtime and not during initialization because the stream had to be started before the pipeline could be linked. The application showing the stream on both the display for the driver, and on twitch.tv can be seen in figure 2.18. The video was sent in h264 format, with a resolution of 640x480, and a bitrate of 1Mbit/s.

2.7.4 Mounting of system

The Pi was mounted in the rear-compartment of the car in order to keep the camera cable as short as possible. The case was attached to the wall of the compartment with velcro so it could easily be taken in and out. The camera was screwed to a small board of carbonfiber which was taped onto the wall so that the camera lens fit exactly into the camera hole in the rear of the vehicle. The diameter of the hole was drilled so that the camera itself could be wedged stuck inside the hole. A 5 meter long HDMI cable and a micro-usb cable were connected between the pi and the screen in the front of the car. Special 90-degree angled cables were purchased so they would fit inside the 3D-printed casing for the screen. The screen case was designed and made by mechanical team member Terje Mork. The camera can be seen in the rear of the car, as indicated by the arror in figure 2.22. Figure 2.20 shows the Raspberry Pi inside the rear compartment and can be seen in lower right corner. The Camera is seen in top middle of the picture. Figure 2.21 shows the camera mount in greater detail. The display itself can be seen in figure 2.23 which shows the inside of the drivers compartment.

2.8 Fuel Cell Monitor

The fuel cell to be used in the Urban Concept was purchased by Horizon, and it turned out to contain a lot of accessory functions. Firstly, the fuel cell controller board contained a small LCD screen where real time data from the fuel cell could be monitored. This included voltage, current, internal and external temperature, etc. Instead of simply putting the extra LCD screen

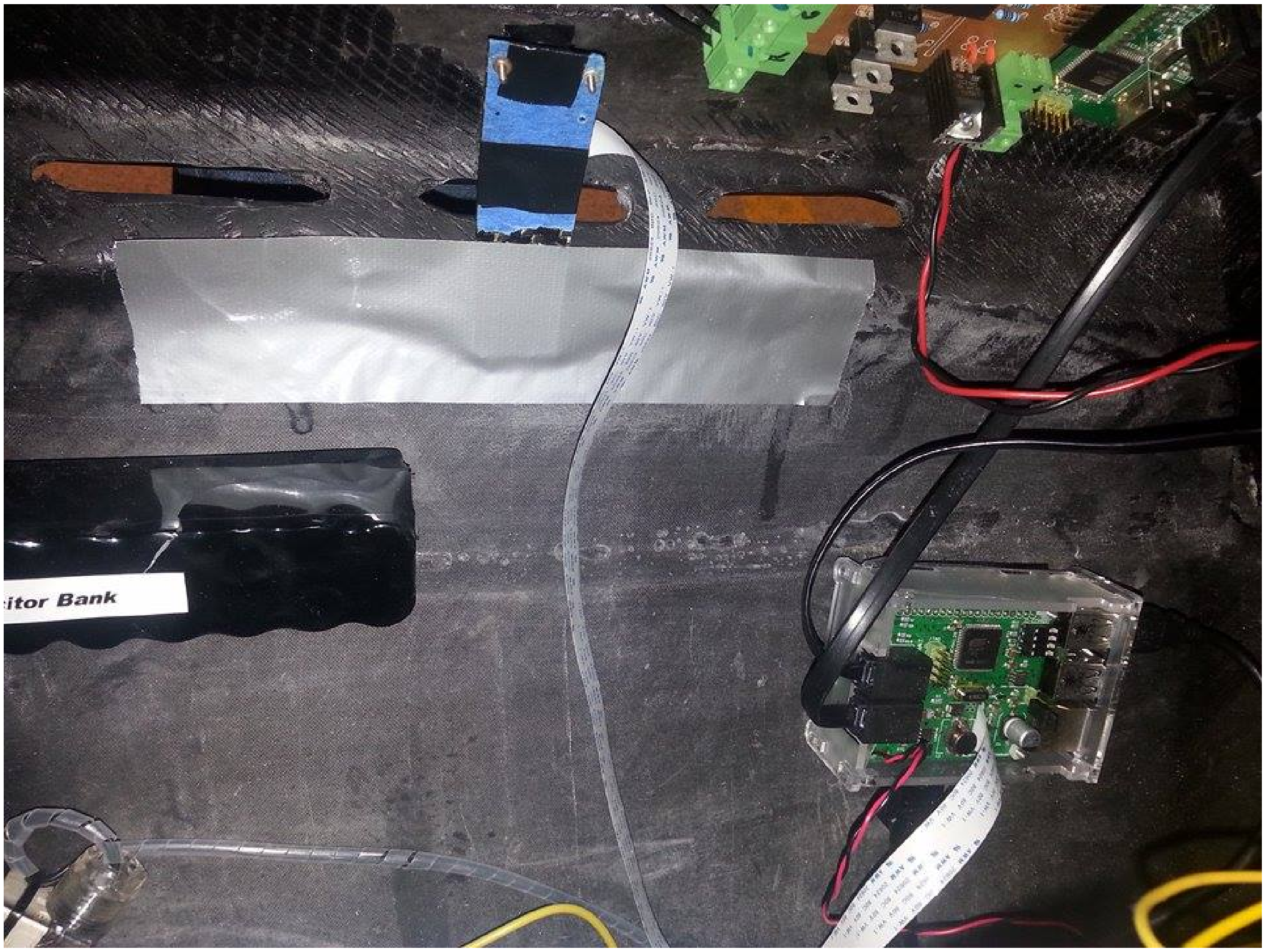


Figure 2.20: Raspberry Pi mounted inside rear-compartment

inside the drivers compartment, it was desired to make the data available on the CAN-bus and incorporate it into the existing screen, as well sending it through the telemetry system Ole Bauck had created. As well as the LCD screen connector on the fuel cell controller there was also an RS232 port that output the same real time data. According to the documentation of the fuel cell, the rs232 cable could be connected to a computer, and software from Horizon could be used to monitor the operation of the fuel cell. However, the software was expensive, without documentation, and only for for the Windows platform. In any case, having a computer running Windows inside the vehicle for the purpose of monitoring the fuel cell was out of the question. Instead, a small system to read the data from the serial port and make the information available on the CAN bus was developed.

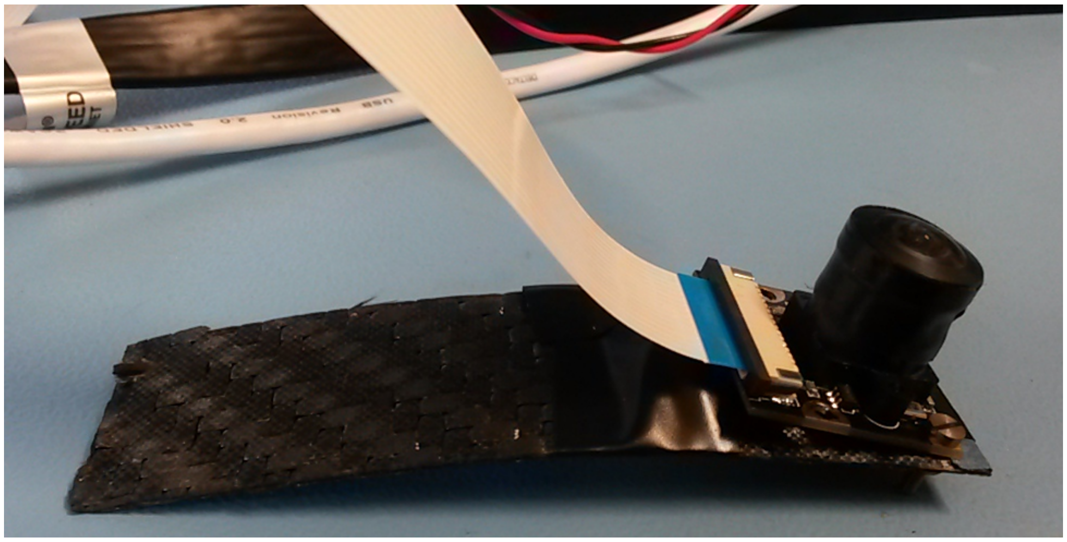


Figure 2.21: Raspberry Pi Camera Mount

2.8.1 Hardware

As always, the universal module was used as a basis for the small project. The universal module had UART available on the GPIO header, so reading data from an RS232 port was easy. The two protocols were identical in the way they transmit data, the only difference was in the voltage levels. A converter board between the two signal lines had to be made. A popular IC for doing exactly this was the MAX3232, which converted between RS232 and 3V TTL. This chip was essentially all that was required, but a few RG-LEDs were also added to the board to indicate the status of the important variables in the system. Although the LCD screen could supply the same information, a board that simply flashed red or green would have been easier to pay attention to, instead of monitoring four different numbers and figuring out if they were above or below the desired threshold. The small board was designed, ordered, and mounted in the same manner as previously explained. The board was tested by connecting it to the a computer with an RS232 cable, and passing information between the computer and the universal module. Everything worked as expected. The finished board, connected to a universal module can be seen in figure 2.24.

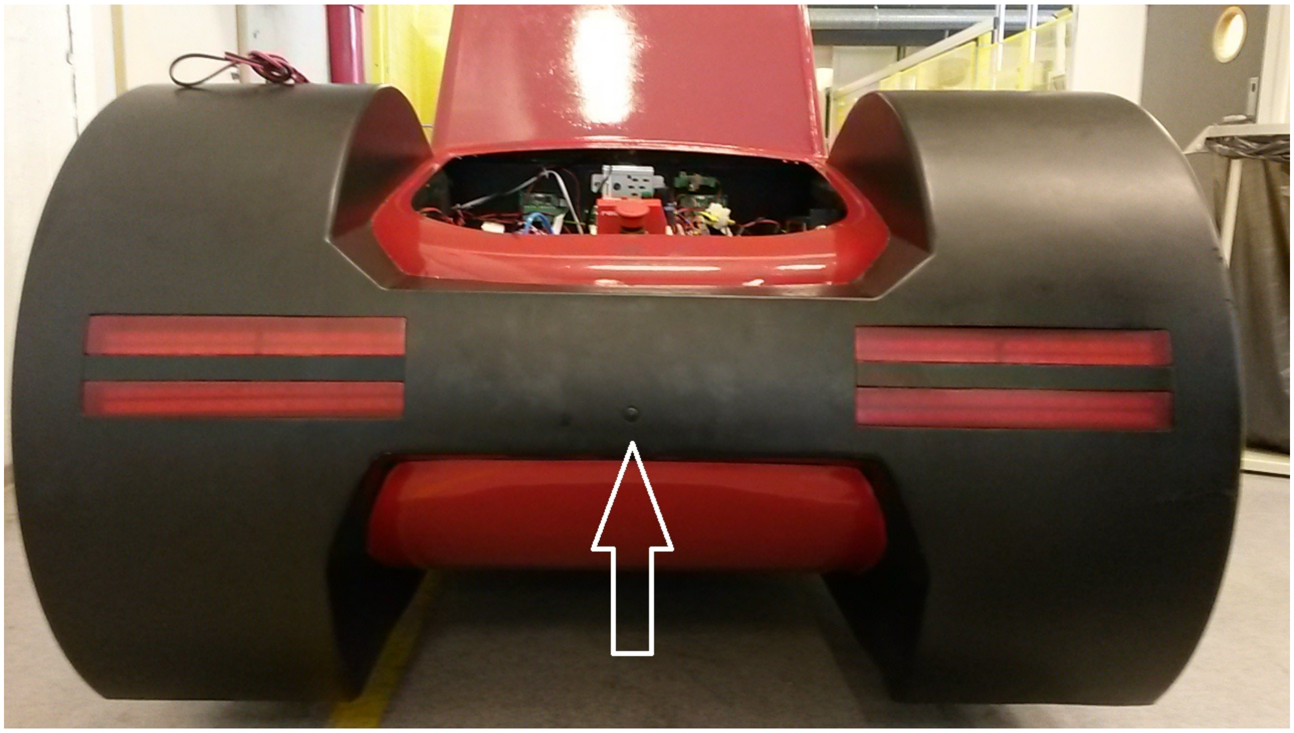


Figure 2.22: Camera seen from behind

2.8.2 Implementation

Unfortunately, since the protocol between the fuel cell controller and the software by Horizon was not given, it was not certain that it would be possible to extract data with the newly created board. The only information given was that the controller communicated on a baudrate of 9600 bps, with 1 stop bit and no parity checks. However, the information itself could be encrypted or difficult to interpret. It would in any case have to be reverse engineered before a useful program could be made to relay the information onto the CAN bus. Unfortunately, to be able to reverse engineer the data stream, the fuel cell had to be running, and the fuel cell itself was not mounted until very late in the semester. Additionally, the team had difficulties in acquiring Hydrogen to feed the fuel cell. Because of all these factors, there was unfortunately no time to try to interpret the data coming from the RS232 port, and the whole project was abandoned in favour of more important tasks. However, during the race the fuel cell started acting strange, and it would occasionally shut down without anyone knowing why, and gathering information from the fuel cell became a priority. The LCD screen connected to the fuel cell controller was attached in the motor compartment and could only be seen during testing while the car stood still and was at

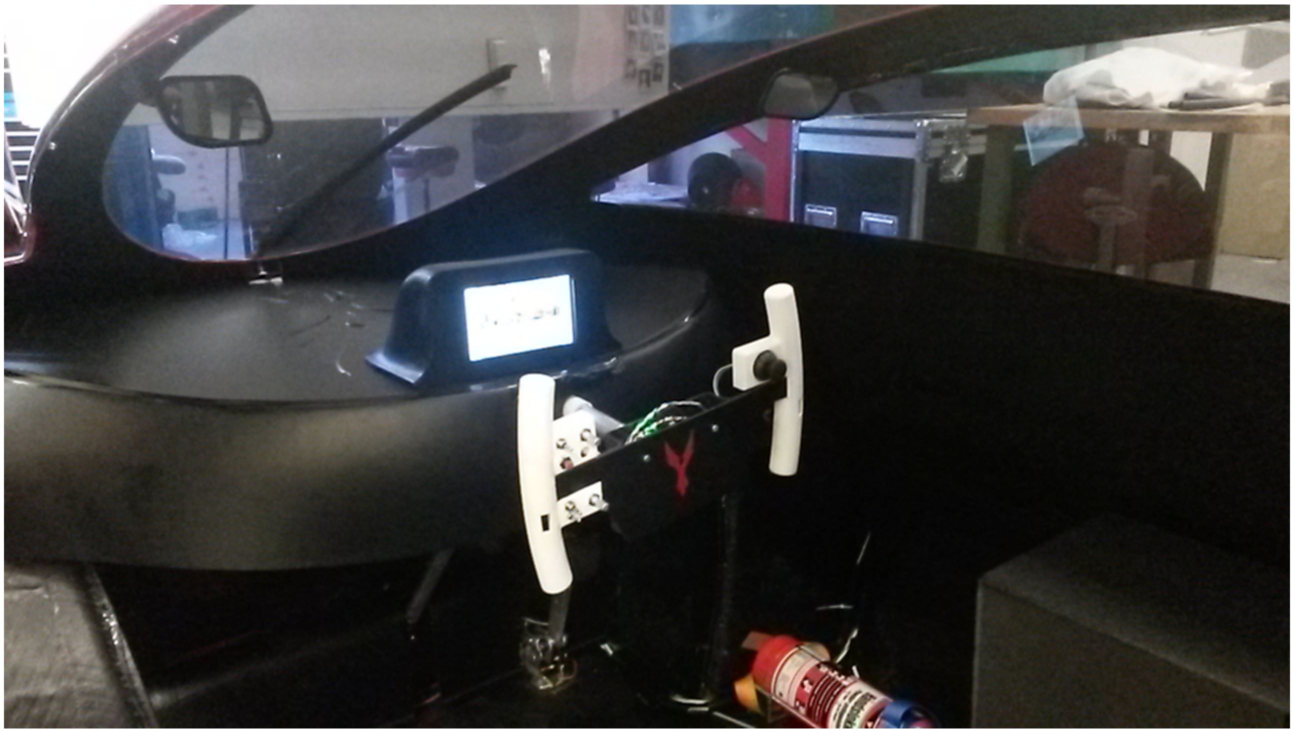


Figure 2.23: View of the driver compartment

first useless. To fix this, the cable between the LCD screen and the controller was extended, and the screen was put inside the drivers compartment so the driver could catch any error messages that appeared, if something went wrong with the fuel cell.

2.9 Evaluation

The finished result was satisfactory with respect to the goals initially set out, although with a some minor shortcomings. As stated, there was extra functionality that had been planned for the interface but time had not permitted the implementation of all the desired features. Additionally, there were a lot of accidents with other more vital components of the system that required attention during the race, so the live-streaming was unfortunately abandoned. The driver could still see the videostream on screen, which ironically nearly caused an accident in the first trial round. Something that neither the designer nor the driver had considered was that the video he saw on screen was mirrored horizontally. During a trial run around the track the driver saw a car approaching on the right side of the screen, and naturally turned left to give way.

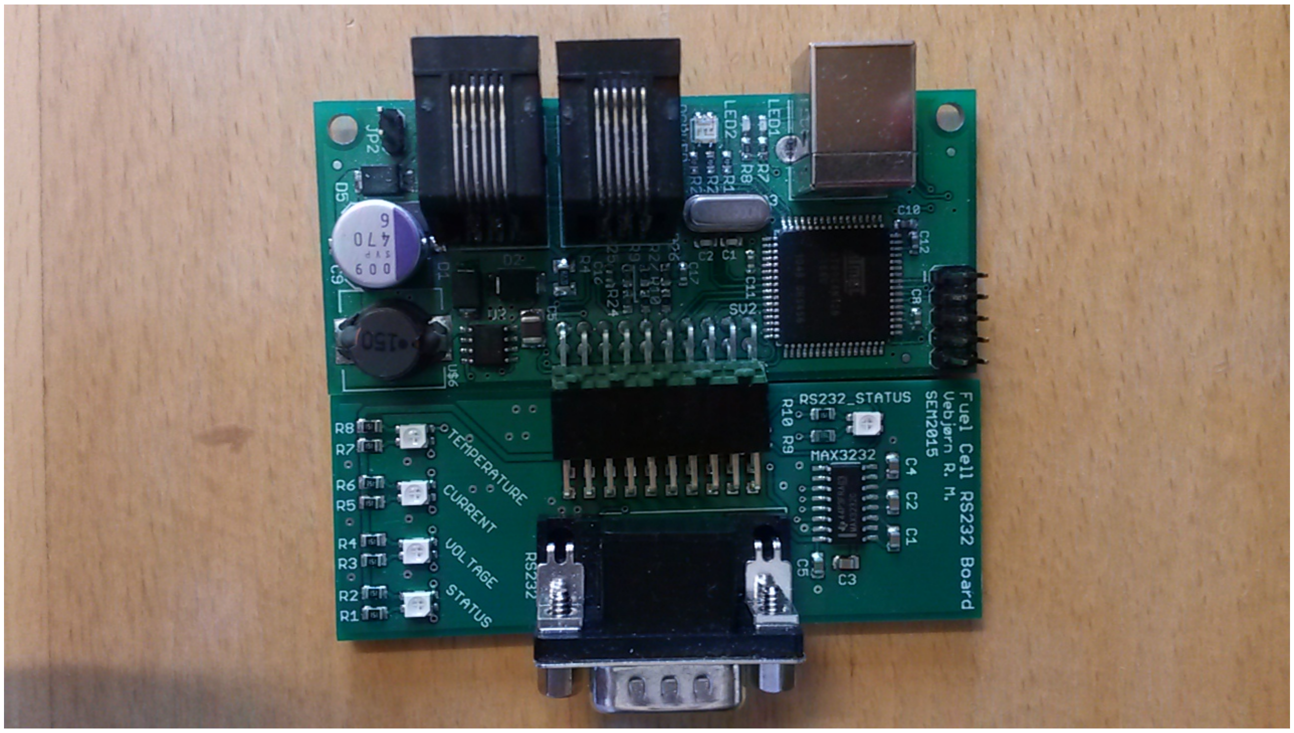


Figure 2.24: RS-232 Monitor and universal module

However, the car was actually approaching on the left side, and they nearly crashed had not the driver also seen the car in the mirrors before it was too late. This was probably one of the reasons for why cameras were not allowed as a complete substitute for mirrors, but could be used as a complement. The error was easily fixed by flipping the video horizontally in the rpicamscr element in gstreamer. Apart from this the video was smooth and without any delay whatsoever. The driver reported it as being of excellent help. Another problem was that the voltage regulator on the HAT got very hot during operation. Since the regulator supplied both the Raspberry Pi and the screen, it probably got very close to its limit of 2.5A. However, the module was measured to only use about 6 W, and it functioned without problems for the duration of the event.

Chapter 3

Driver Interface - Prototype

The screen for the driver in the Prototype also had to be created, but it was not meant to be as complex as the screen in the Urban. It would of course be an advantage for the driver to be able to see clearly what was happening behind the car, but there simply was not enough space for a large screen in the Prototype. Additionally, the Prototype vehicle was considered the more competitive of the two cars regarding efficiency, while the Urban Concept was designed mostly for appearances. For this reason, we didn't allow any components in the Prototype that unnecessarily added weight to the vehicle. The screen would therefore be as simplistic and minimalistic as possible.

3.1 Design

The existing screen in the vehicle was functioning well, but from a designers perspective it looked quite bad. It was something that had been hurriedly put together last year, and it was holding back our chances of winning the design award with the Prototype vehicle. Since the mechanical team who usually create physical parts were quite busy with more important tasks, the author of this paper took it upon himself to create the entire screen system. The actual LCD-display from the year before was sufficient and could be re-used, but it had to be integrated into the vehicle in a better fashion. The task therefore came down to creating a casing that was both functional and looked decent. Given the small size of the required case it was decided that it could be 3D printed. The author had little experience in 3D printing, but was advised that learning the required tools was relatively easy. Autodesk Inventor was chosen as a CAD-tool because the professional version of this software was free to use for students. A day was spent following the in-program tutorials and the part was created in a matter of hours using only the extrude and hole tool. The part was designed so that the LCD-screen would fit exactly inside, as well as a universal module. The part was exported to STL-format and printed on a uPrint SE Plus printer, located at Gløshaugen NTNU. The printing quality of the printer was excellent, and the result was very satisfactory. The first print didn't fit because of a slight misunderstanding of measurement constraints in Autodesk, but the second attempt was perfect. The screen was mounted to the case with four 3mm bolts and nuts. The universal module was fixed with only two bolts because the module only had two mounting holes. If the reader is seeing this pdf in a relatively new version of Adobe Acrobat Reader, an interactive 3D model of the LCD case can be seen and played with in figure 3.1. The finished result can be seen figure 3.2 and 3.3, while the previous system can be seen in figure 3.5. The model can be found in the attached zip-file. Although the car didn't win, it came as third runner-up for the Vehicle Design Award [29].

3.2 Code

The LCD screen module from Sparkfun had an onboard driver and the only input was UART RX. The received serial data was immediately displayed on the screen. However, there were also a set of control codes that could be sent to the screen to change certain settings, or set where the text was to appear etc. There was also a splash-screen that appeared for two seconds when the module was powered. This text was set to display *DNV GL Fuel Fighter Prototype*. Occasionally, the screen would randomly receive control-codes that somehow changed the backlight-settings and other configurations. This was fixed by sending a reset code to the screen on powerup. This made sure the screen was always set correctly after a reboot. The cause of these occasional errors was never determined, but it was probably due to noise on the UART line being interpreted as control characters. In order to easily display information on the screen, a simple library was made in C for displaying the desired values for speed, torque, fan-intensity, motor-status, and time remaining. The Atmel Studio project for the screen can be found in the attached zip-file.

(lcdcase.u3d)

Figure 3.1: LCD Case - Interactive 3D model

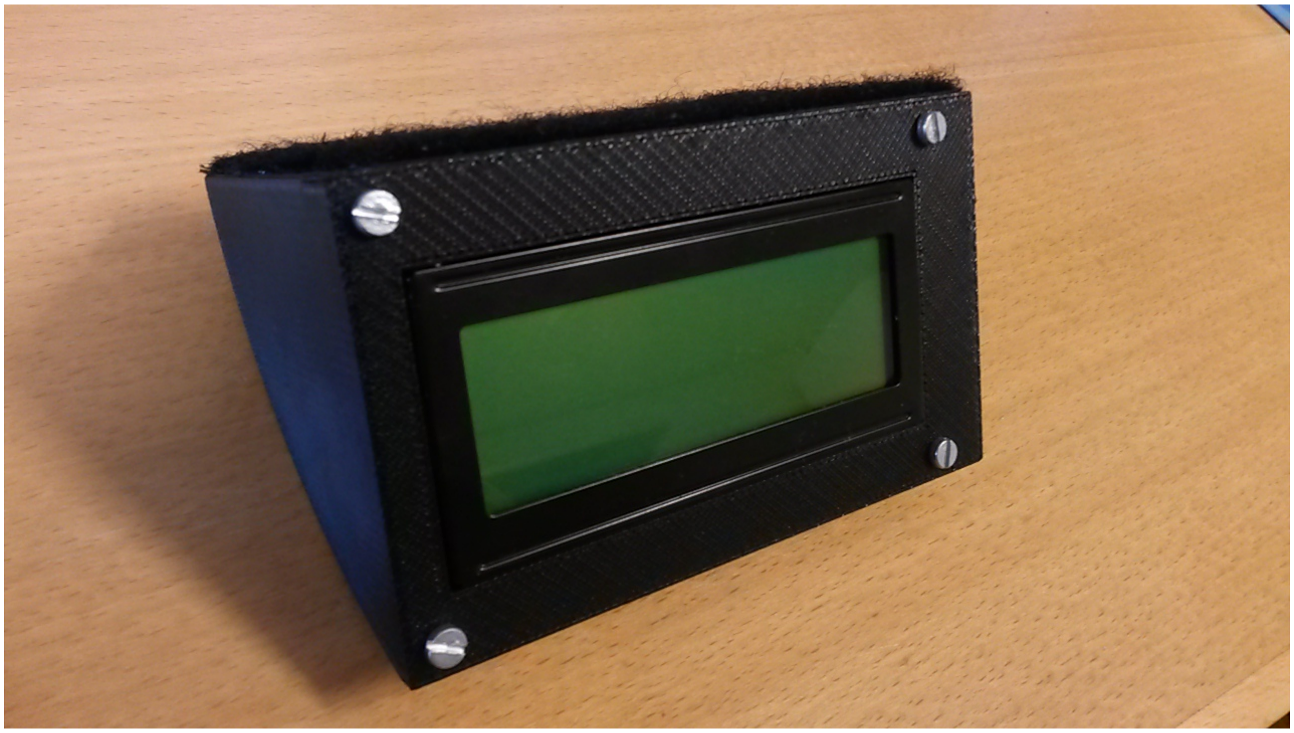


Figure 3.2: Prototype Screen, Front

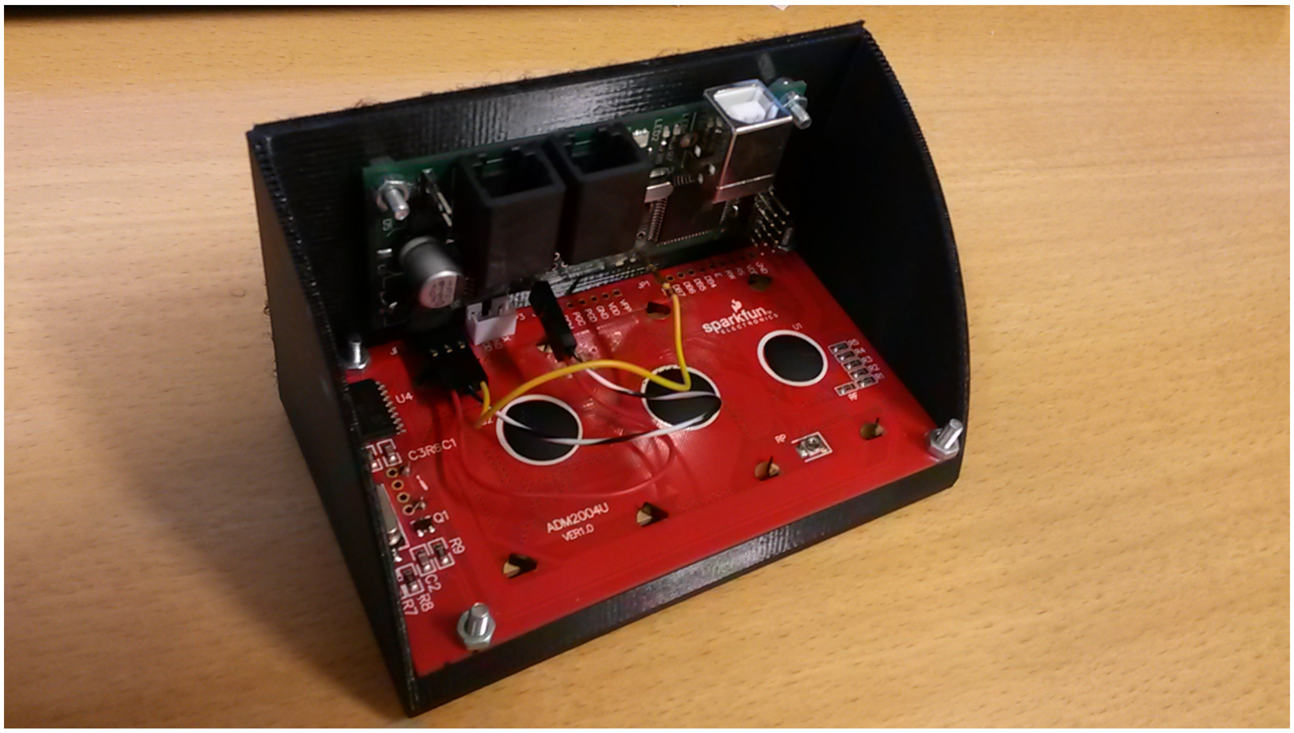


Figure 3.3: Prototype Screen, Rear

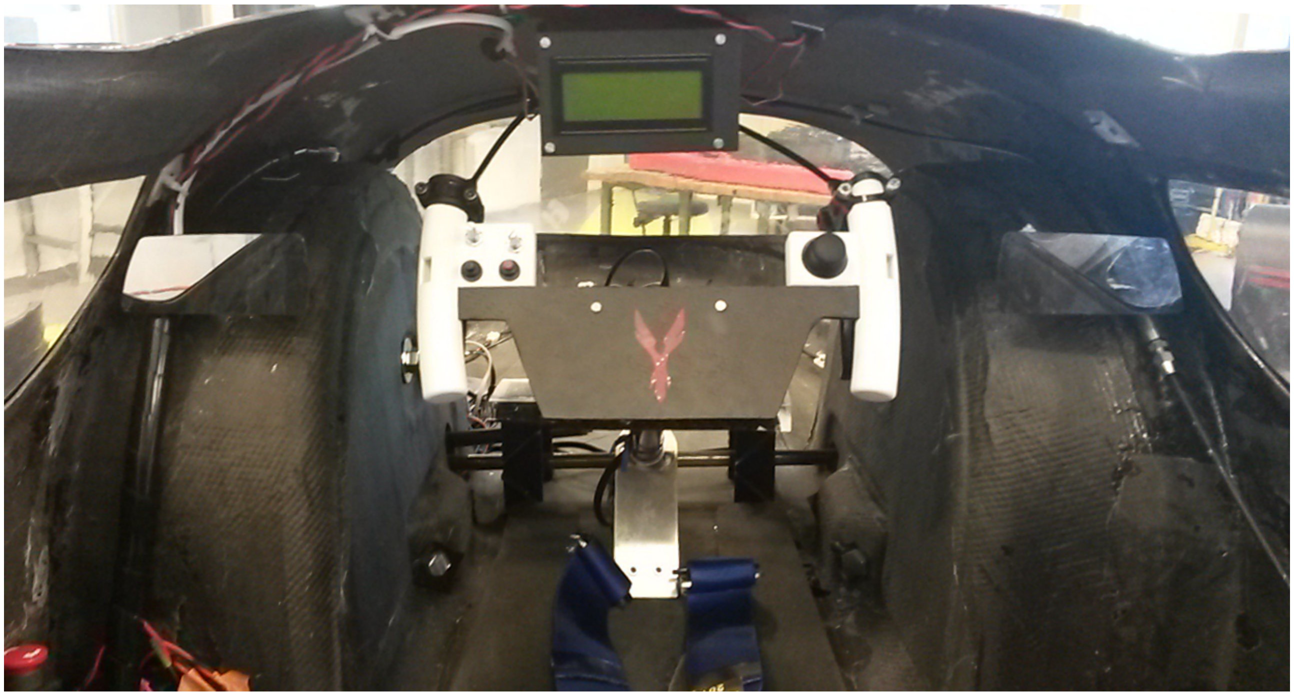


Figure 3.4: Prototype Screen Mounted in Vehicle

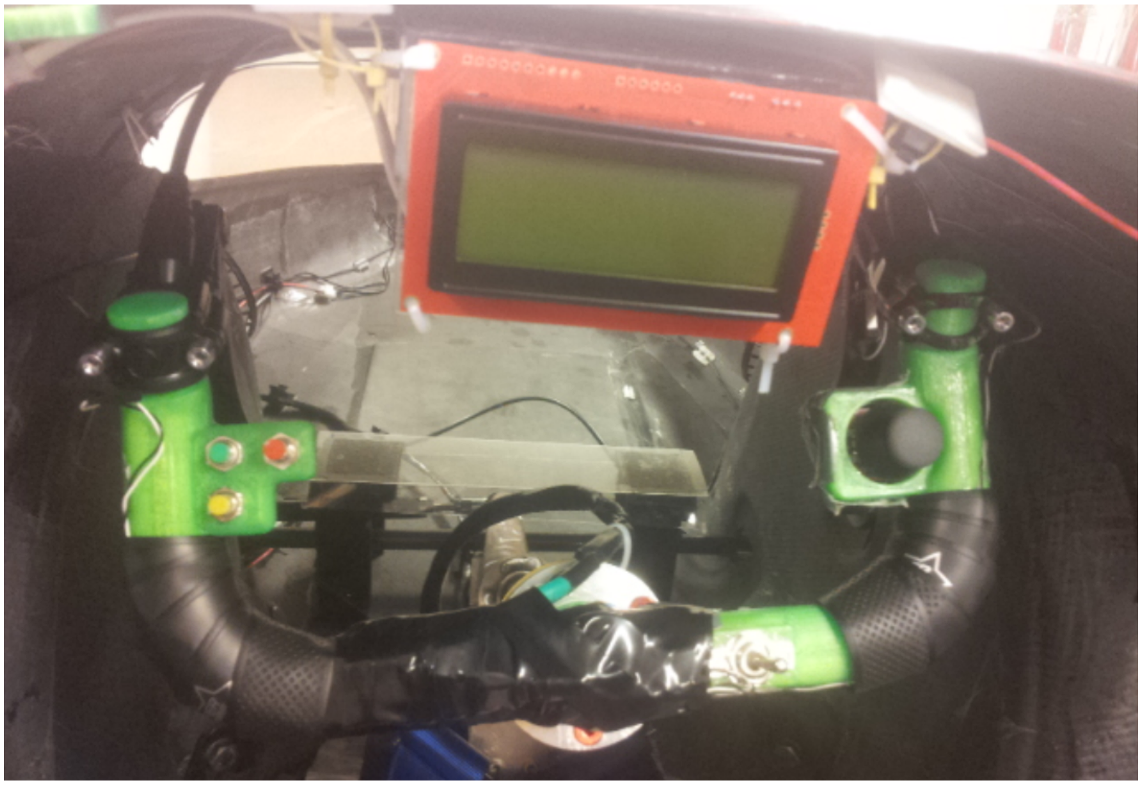


Figure 3.5: Old Prototype screen and steering wheel

Chapter 4

Steering-wheel

The last task to do was to create an interface on the steering wheels so the driver could control all the vital mechanisms of the vehicles. The steering wheels in both cars were nearly identical so the task was made easy by creating one set and simply duplicating it. The only difference was that the Prototype required fewer functions, as it had less components to control such as lighting and window-wipers.

4.1 Requirements

In order to create a system to control the vehicle, it had to be established exactly what elements there were, and how they were going to be controlled. The system was first designed for the Urban Concept, and then later checked to see if the same system could be applied to the Prototype, only with reduced functionality. The other members of the electrical team were consulted, and an initial list was compiled, as shown in table 4.1.

The angel eyes are the circles of light on the outside of the main head lights, as indicated by the green arrow in figure 4.1. The Eyebrows are indicated by the blue arrow. The eyebrows also serve as the indicators by toggling the color between orange and white. The rear lights were programmed to simply follow the head lights.

It was determined that the number of buttons initially required was slightly too large to fit on the small steering wheel. Instead, the functionality of controlling the angel eyes and the eyebrows were transferred to the Joystick. The joystick had two axis and only one was used to

Table 4.1: Steering wheel requirements

Function	States	Input type
Horn	2 - On/Off	Momentary push button
Window Wiper	2 - On/Off	SPDT Toggle Switch
Throttle	Continuous	Potentiometer - Joystick
Front lights	3 - Off/Dimmed/Bright	SP3T toggle switch
Angel Eyes	2 - On/Off	SPDT Toggle Switch
Eyebrows	2 - On/Off	SPDT Toggle Switch
Indicators	3 - Off/Right/Left	SP3T Toggle Switch
Fans	Continuous	Rotary potentiometer
Handsfree Volume	Continuous	Rotary Potentiometer
Handsfree Call/Hang up	2 - On/Off	Momentary push button

control the throttle. By moving the joystick horizontally it could toggle the eyebrows by moving right, and toggling the angel eyes by moving left. The rules also stated that the vehicle must either be equipped with a dead-mans switch or a spring-loaded accelerator. Since the throttle was controlled by the joystick, which sprung back to neutral position when let go, a dead-mans switch was not required. Another thing that was required by the rules was that the rear brake lights had to be activated when the brake pedal was pushed. This will be further explained in section 4.4.

4.2 Hardware

In order to read the input from the steering-wheel and convey this information on the CAN-bus it was decided to embed a universal module into the steering-wheel. Luckily there were enough GPIO pins on the universal module to attach all the buttons to it without using a GPIO expander. However, since each button also required a ground signal, the buttons couldn't simply be connected to the module because there weren't enough ground pins available. To overcome this, a simple breakout board was designed to allow all the buttons to easily be attached. The board contained nothing but headers, and looked like in figure 4.2. The names of the buttons were added to the silkscreen to make it easy to disassemble and reassemble the whole board.

The buttons were chosen to be as small as possible as to not take too much space on the steering wheel. The physical dimensions were passed along to Sigbjørn Kjensmo who was in

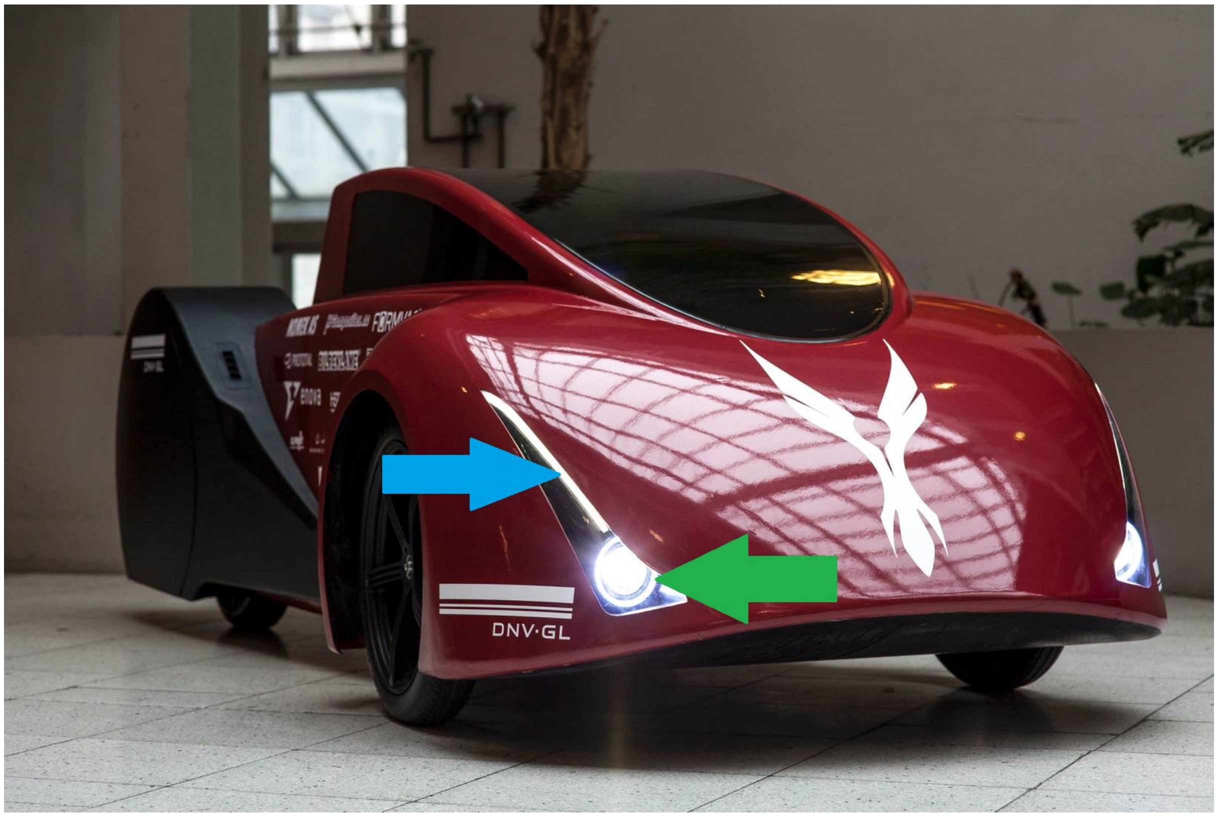


Figure 4.1: Urban Concept Front Lights

charge of constructing the actual steering wheel, partly in carbon fiber, partly 3D-printed. The wheel was then designed to exactly fit all the buttons. The wires between the buttons and the board were cut to exact length and soldered on. Shrinking tubes were added to all terminals to avoid short-circuits. Normal 2.54mm dupont connectors were used to connect the wires to the board.

4.3 Assembly

The breakout was attached to a universal module and put inside the steering-wheel. Two holes were drilled in the front plate of the wheel so that the universal module could be fastened with screws. A JTAG cable was attached to the universal module and kept inside the steering wheel so that the module could be reprogrammed without removing the entire board. Removing the board was a bit troublesome because the two screws were difficult to remove, and all the buttons had to be unplugged before the board could be completely removed. All the buttons were panel

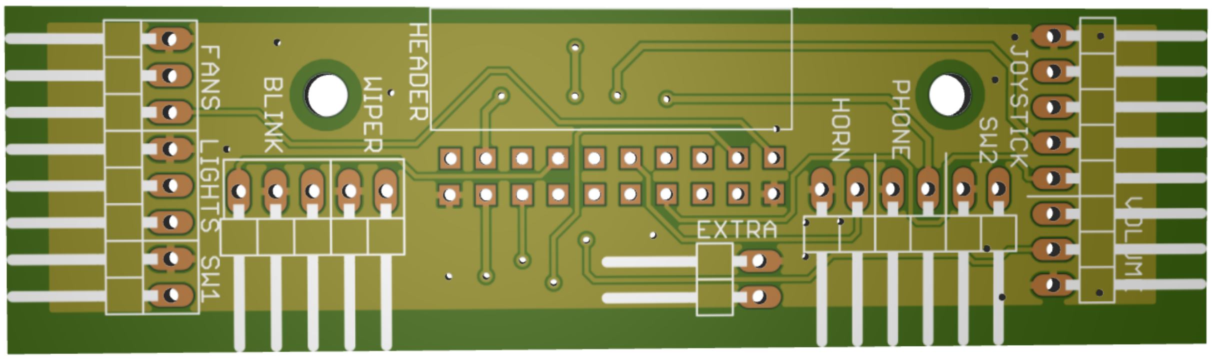


Figure 4.2: Steering wheel breakout board

mounted, meaning they were inserted through a hole on the steering wheel from behind, and screwed tight with a nut on the front of the wheel.

4.4 Brake Switch

Urban Concept

A brake sensor had to be fitted in the Urban Concept, and luckily the one from the previous Urban Concept car could be reused because the very same brake pedal was also reused. New cables were soldered to the sensor because the old ones had to be cut off. The sensor consisted of a normally open lever switch mounted on the base of the brake-pedal. When the brake was not pushed, the pedal itself was holding in the button on the switch, thereby closing the circuit and indicating that the brake was not pushed. By pushing the pedal forwards, the button was released, and the circuit was opened. This automatically turned on the brake-lights, and also sent a message to the motor controller, telling it to stop the motor.

Prototype

According to the rules, a brake sensor was required in the Urban Concept, but not in the prototype. However, it was decided to still implement one. In the prototype, the only function of the brake sensor would be to disengage the motor. This would make sure that driver wouldn't be able to accelerate and brake at the same time. Instead of a brake pedal, the prototype used normal bicycle brake levers mounted on the steering-wheel. Small switches were glued inside

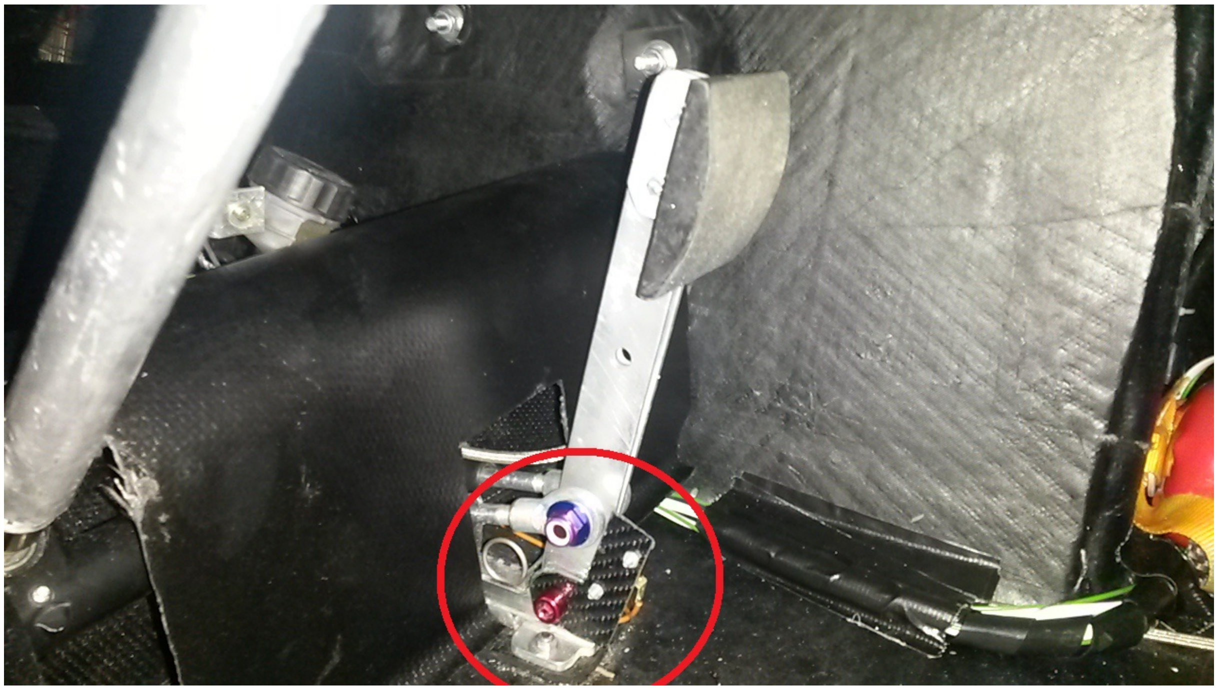


Figure 4.3: Brake pedal with brake sensor

the handles and connected to the steering-wheel module. When the levers were squeezed, the system would simply set the torque reference to 0, overriding whatever value was read from the joystick.

Figure 4.7 shows the Prototype steering wheel from the front. A different view is presented in figure 4.8 where the USB and RJ11 ports are visible, and the lose grey JTAG cable for easy reprogramming is also visible. One problem with the steering wheel was that the central part in which the circuit board resided was made of carbon fiber. Since carbon fiber is conductive, the circuit board had to be isolated. This was fixed by applying non-conductive tape on the inside of the steering wheel, as seen in figure 4.9.

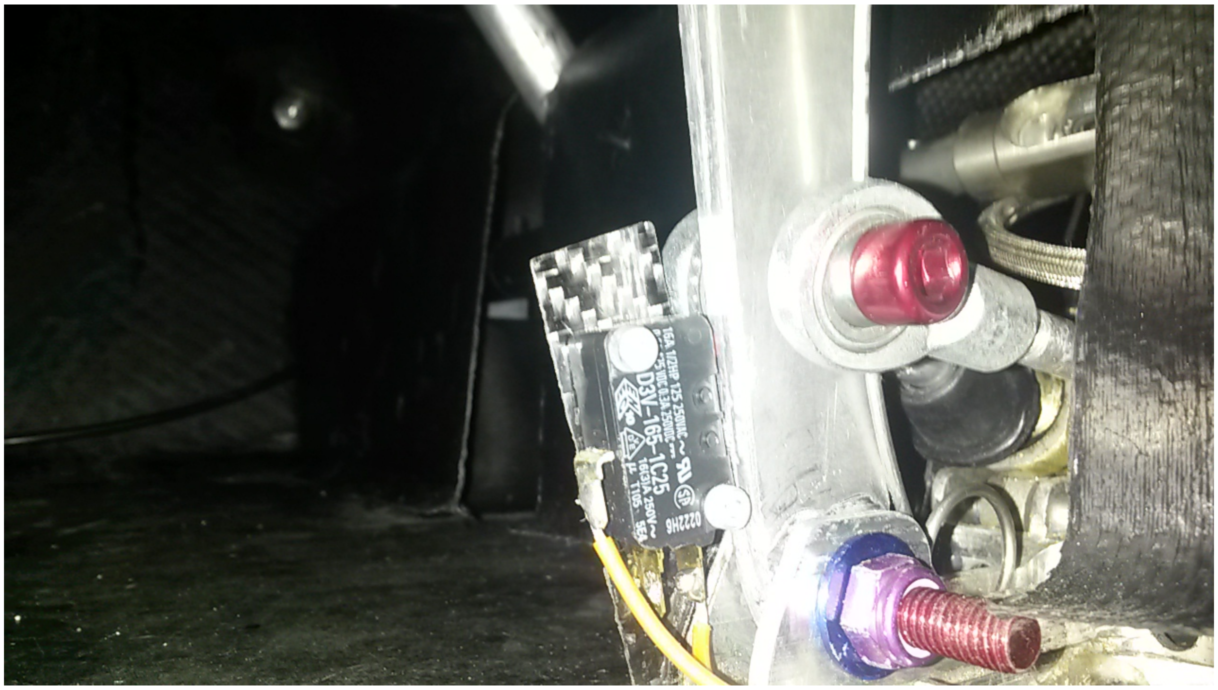


Figure 4.4: Brake sensor

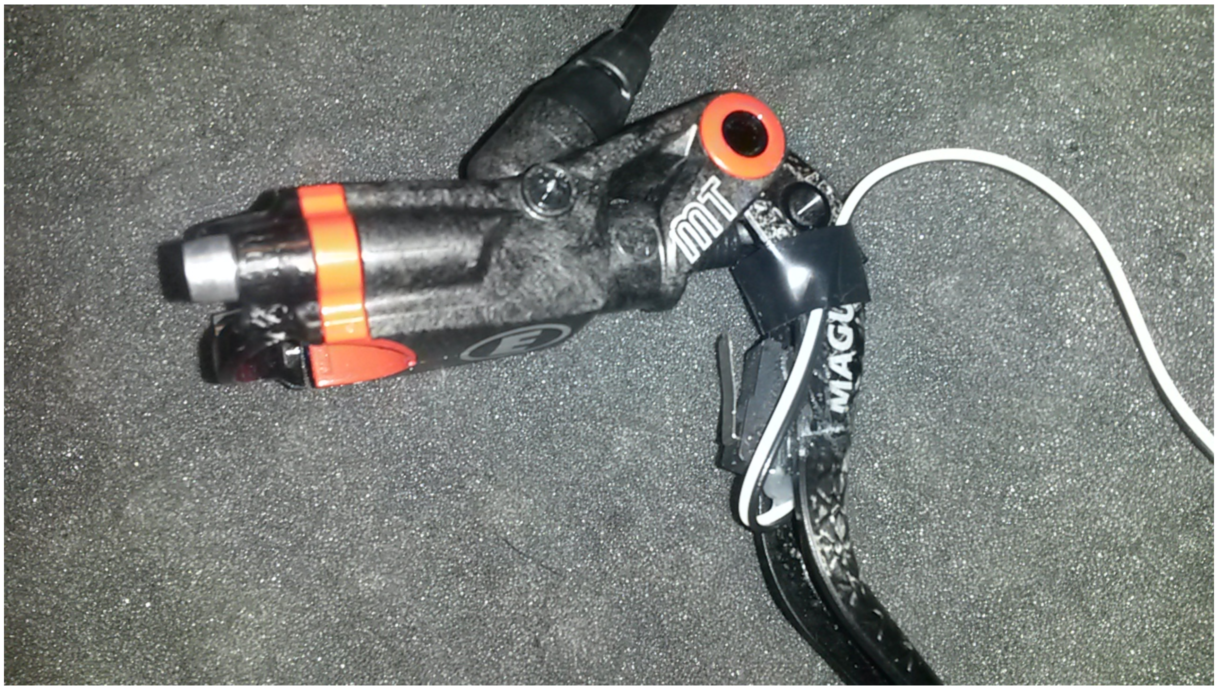


Figure 4.5: Brake handle in Prototype with mounted switch

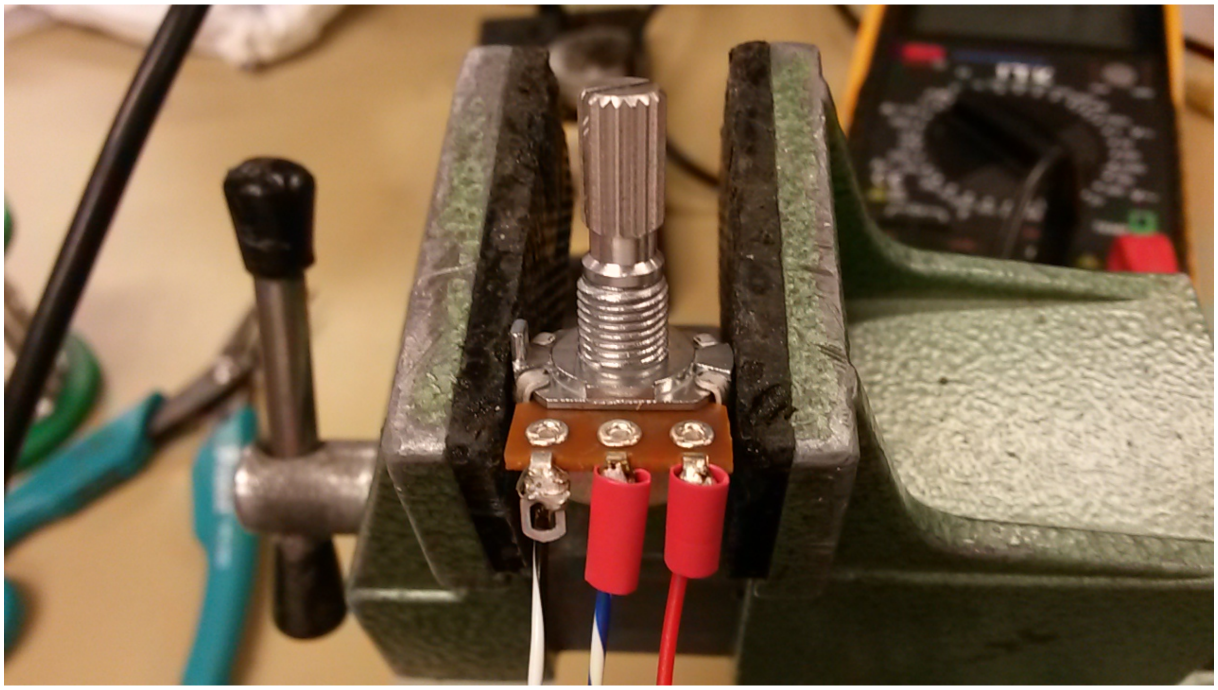


Figure 4.6: Rotary potentiometer being soldered

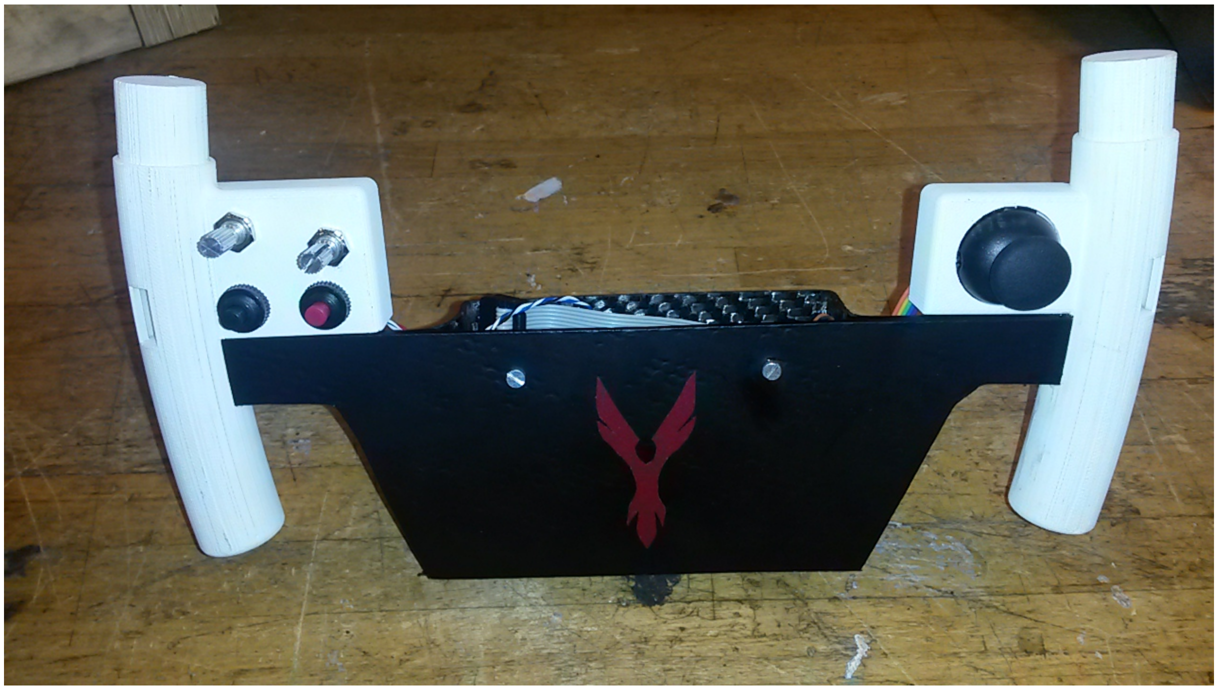


Figure 4.7: Prototype steering wheel, front

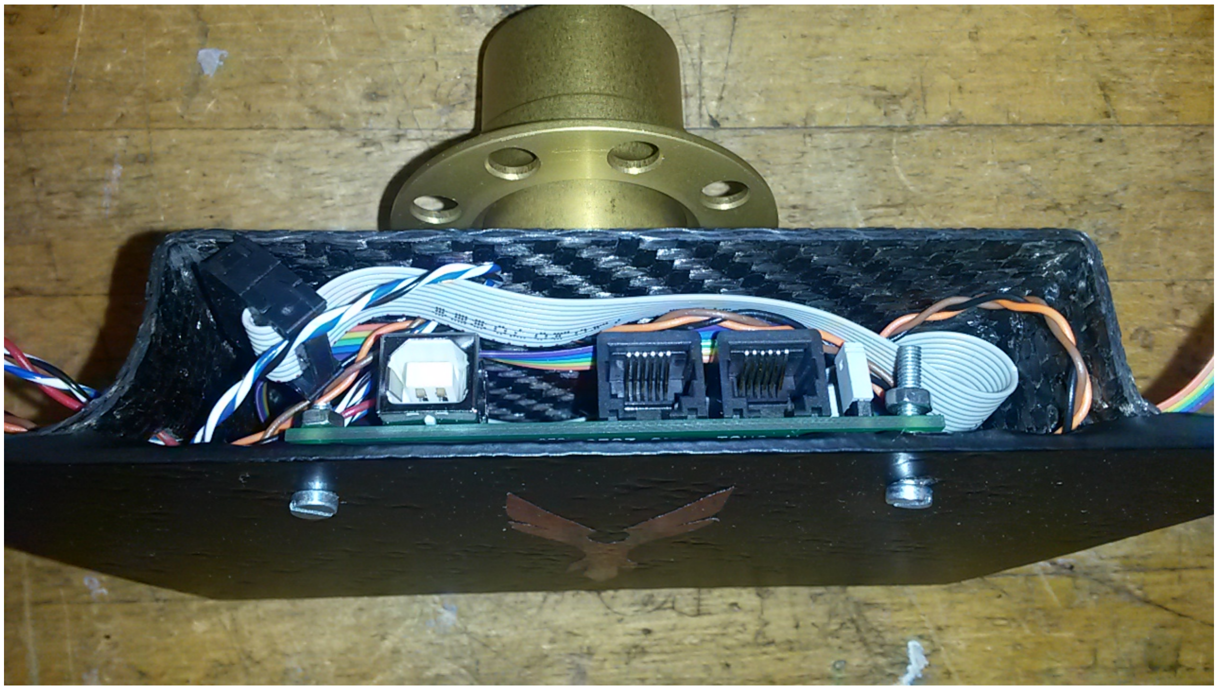


Figure 4.8: Prototype steering wheel, top

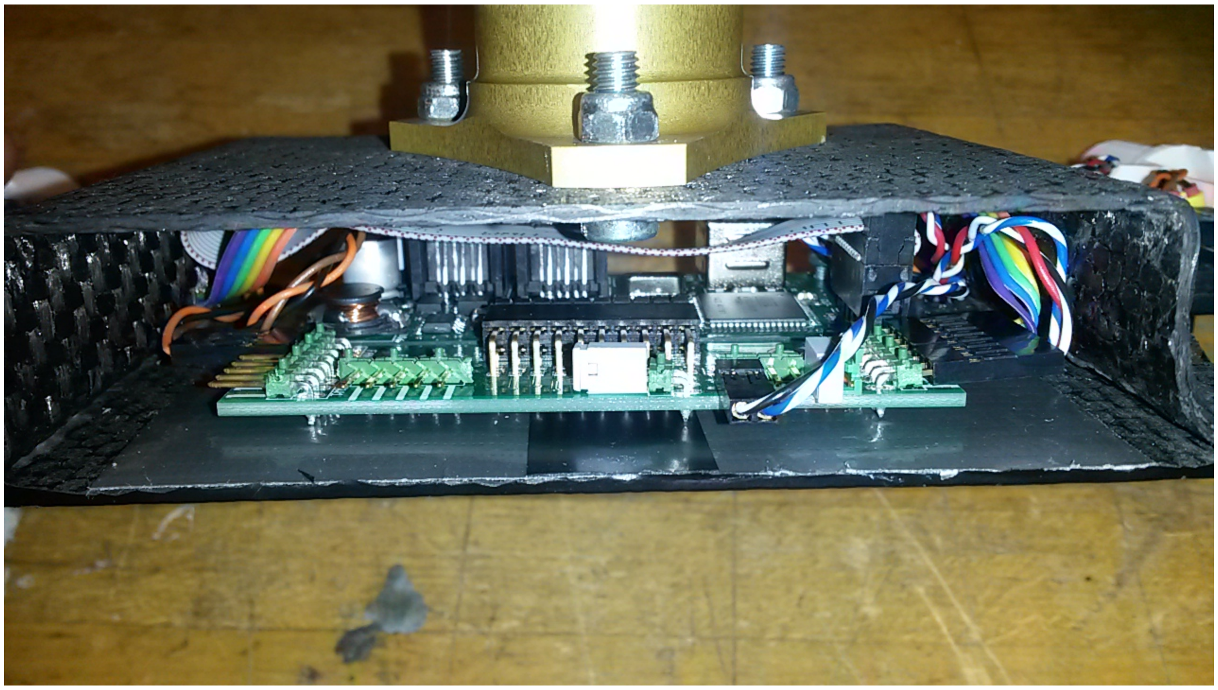


Figure 4.9: Prototype steering wheel, bottom

Chapter 5

Discussion

5.1 Evaluation and further work

5.1.1 Urban Concept Screen

The goal in this thesis was primarily to create a screen for the driver in the new DNV GL Fuel Fighter Urban Concept vehicle. The contents on the screen would include relevant information while driving, such as speed, time, and a video stream from behind the vehicle. In the authors eyes, this goal was met. However, a lot of additional features had been planned that unfortunately were not implemented due to lack of time. One of the primary goals, namely the online video streaming, was sadly not performed during racing. However, with some preparation, it should be perfectly simple to manage to stream the video online for next years team. Some of the work in this thesis has been a dependency in a larger system, and the system had to be functional by a given date, May 15th. Because of this, the most vital components were always prioritized, and any extra features were always postponed. However, a good foundation has been created for further development of the touch-screen system. In hindsight, a better schedule should have been created, and a more realistic workload should have been planned. The project of creating the fuel cell monitor was not planned from the beginning, and should never have been started. It would have been an excellent feature to have, but ended up taking time away from the actual planned work, and wasn't even made to function.

5.1.2 Prototype Screen

A secondary goal was to create a new screen-system in the Prototype vehicle. This was also accomplished. The screen module itself appeared to have a few hiccups, though. Occasionally, but very rarely it would change the backlight setting so that the text would be barely visible. It would also very rarely change the position of where the text would appear. The cause of these problems could never be found, but it was theorized that it was due to noise on the uart line which was interpreted by the module as control codes which affected the settings. Fixing these problems could be done by simply rebooting the screen system. As a cybernetics engineer it was also fun to design a physical component for a change, as opposed to making circuit boards and code. The result also helped improve the looks of the vehicle, and along with the improvements by the mechanical team it resulted in a third place for the Design Award.

5.1.3 Steering wheels

The steering wheel modules were also a success, an no problems occurred during testing or racing. However, assembling the wheel was a bit difficult, and if something had gone wrong it would have taken some time to be able to fix it. Utilizing the small space was difficult, and a better system can probably be made so that removing the circuit board is less troublesome. Additionally, having an open circuit board surrounded by carbon fiber was also a bad idea. The circuit board should probably be better isolated in the future, since a short circuit in this system could potentially render the driver unable to control the vehicle.

5.1.4 Fuel Cell Monitor

The fuel cell monitor is unknown if would work as the communication between the fuel cell and the program will have to reverse engineered. If the next years team have enough time, I suggest making an effort in trying to see if it's possible. To be able to monitor and log all data from the fuel cell during testing and driving could be extremely useful. It could be used to help optimizing the driving strategy, as well as catching error messages if something were to go wrong.

Chapter 6

Conclusion

In this report, my work on the Fuel fighter project has been documented and evaluated. The goal was to create a touch-screen system for the Urban Concept vehicle, competing in Shell Eco Marathon in May, 2015. The screen was to display relevant information for the driver, as well as show a rear-view video stream for the driver. The same video stream was also to be streamed online. Additionally, a screen solution for the Prototype vehicle, as well as systems to take input from the driver were to be created. All the goals were met, although some with reduced functionality.

By utilizing existing solutions as well as creating new hardware and software, a touch screen interface was created for the Urban Concept vehicle. The screen was connected to the peripheral systems of the car, and was able to display relevant information for the driver, as well as a video stream from a rear view camera. The video stream was sadly not streamed online during the race, but the foundation to do so was created. Online streaming was performed successfully in a testbench environment.

The existing screen in the Prototype vehicle was upgraded to a more visually appealing and functional solution. Although some minor bugs were present in the screen it worked well and without problems during the race.

The systems for handling input from the driver in both vehicles were created and worked very well. The same solution was implemented in both vehicles, with minor differences in input capabilities. Some improvements can be made with respect to mounting and instalment of the circuit board in the steering wheel.

Based on the results from this work, it is recommended to reuse the systems in the next years competition. The camera solution in the Urban Concept gave the driver a heightened sense of awareness of his surroundings while driving, and was greatly appreciated. The screen in the Prototype also worked well, and the driver found it helpful. The steering wheel modules were strictly necessary to pilot the vehicles, and based on their performance and robustness they are highly suggested to be reused.

Appendix A

Acronyms

DOF	Degrees Of Freedom
CAN	Controller Area Network
GPIO	General Purpose Input Output
BMS	Battery Management System
BOM	Bill of Materials
SIM	Subscriber Identity Module
SD	Secure Digital
IMU	Inertial Measurement Unit
GPS	Global Positioning System
HAT	Hardware Attached on Top
USB	Universal Serial Bus
FPS	Frames Per Second
RAM	Random Access Memory
CSI	Camera Serial Interface

DSI Display Serial Interface

MIPI Mobile Industry Processor Interface

EEPROM Electrically Erasable Programmable Read-Only Memory

MCU Microcontroller

I2C Inter-Integrated Circuit

SPI Serial Peripheral Interface

TTL Transistor-Transistor Logic

IC Integrated Circuit

RG-LED Red-Green Light Emitting Diode

PCB Printed Circuit Board

CAM Computer-Aided Manufacturing

UART Universal Asynchronous Receiver/Transmitter

SPDT Single Pole Double Throw

SP3T Single Pole Triple Throw

6P6C 6 Positions 6 Contacts

6P4C 6 Positions 4 Contacts

RTMP Real Time Messaging Protocol

UDS Unix Domain Socket

UDP User Datagram Protocol

TCP Transmission Control Protocol

IP Internet Protocol

RJ11 Registered Jack 11

Appendix B

Schematics, Layouts, and Code

B.1 Attached Files

Various files can be found in the attached file, *attachments.zip*. All PCB schematics, layouts, BOMs, and productions files can be found under the folder *PCB*. The project files are for EAGLE version 7.2. The Raspberry Pi HAT schematic, layout, and BOM has been reproduced below. The code for the at90can128 microcontroller on the HAT can be found under the folder *AVR/rpi2_hat*. The code for the steering wheels can be found under *AVR/steering_wheel_module*. The code for the Prototype and Urban Concept screen interfaces can be found under *AVR/ display_module_Prototype* and *Python/*, respectively. All projects except for the Urban Concept screen interface were made in Atmel Studio 6.2, and compiled with avr-gcc 3.4. The Urban screen program was run with python 3.4.3. A script that automates the process of installing gstreamer version 1.4 has also been included in the attached files. It is found under *gstreamer/* and is executed on the target machine as root.

B.2 Raspberry Pi Hat

Table B.1: Bill of Materials, Raspberry Pi HAT

Name	Value	Size
C1	470 uF	SANYO_SMD
C2	1 uF	C1206
C3	22 nF	C1206
C4	220 pF	C1206
C5	10 uF	C1206
C6	22pf	C0805
C7	22pf	C0805
C8	100nf	C0805
C9	100nf	C0805
C10	100nf	C0805
C11	1 uF	C1206
C12	1 uF	C1206
C18	1 uF	C1206
CAN-JP		1X02
CAN-TRANSCEIVER	MCP2551	SOIC127P600X175-8N
D1		SMB
D2		SMB
D5		SMB
HAT-LED	RG_LED	PLCC4
ID-EEPROM		SOCKET-08
J1	RJ11	RJ11-6
J2	RJ11	RJ11-6
JP2		2X20
JTAG		AVR_ICSP
L1	15 uH	CDRH125
L5973D	L5973D	HSOP8
LDO	LDO_3V3	LDO_3V3_SOT23
PI-LED	RG_LED	PLCC4
PTC	PTC	R1210
R1	10k	R1206
R2	3k	R1206
R3	4k7	R1206
R5	22k	R0805
R6	120R	R0805
R7	150R	R0805
R8	150R	R0805
R9	150R	R0805
R10	150R	R0805
R11	3.9k	R0805
R12	3.9k	R0805
R13	1k	R0805
Y1	8MHz	HC49US

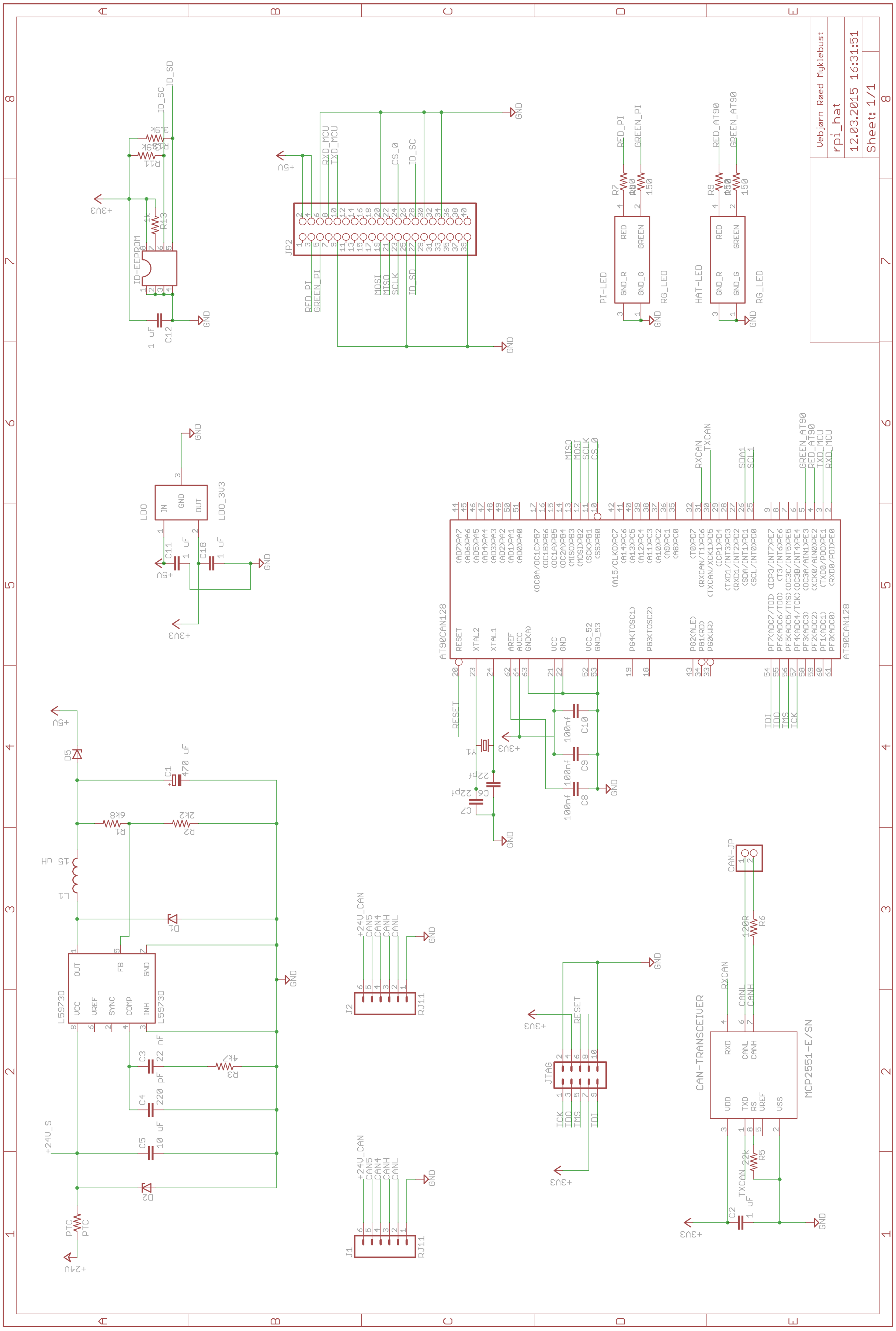


Figure B.1: HAT Schematic

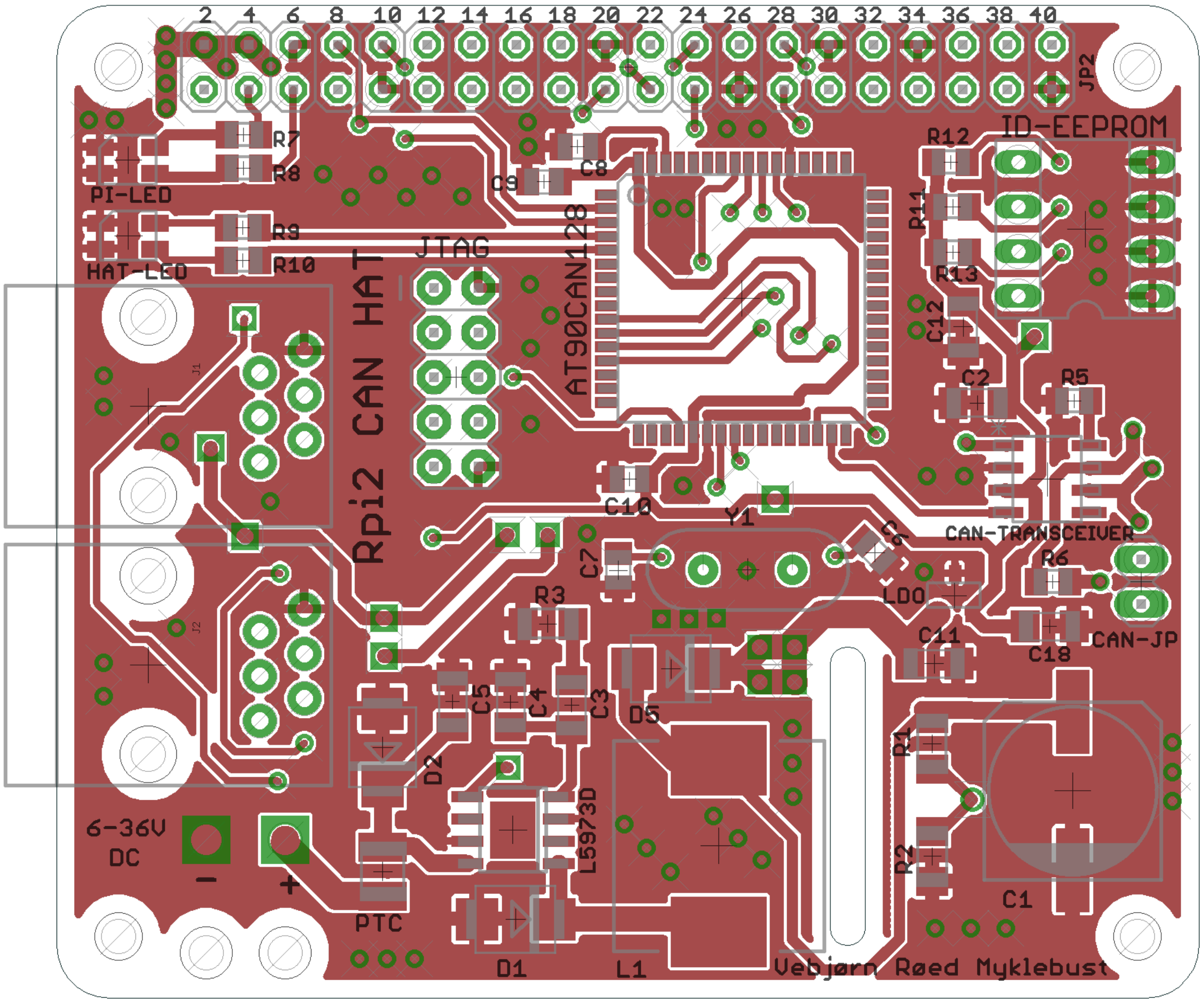


Figure B.2: HAT Layout

Bibliography

[1] Adafruit. http://www.adafruit.com/product/2260, 2015.

[2] James Adams. https://www.raspberrypi.org/introducing-raspberry-pi-hats/, 2014.

[3] et al Aksel Qviller. Eco marathon project report. Master's thesis, NTNU, 2012.

[4] et al Astrid Rasten. Dnv fuel fighter interior design. Master's thesis, NTNU, 2013.

[5] Atmel. http://www.atmel.com/images/doc7679.pdf, 2008.

[6] Jon Martin Harstad Bakken. Styresystem for fremdrift av shell-eco- marathon-kjøretøy. Master's thesis, NTNU, 2009.

[7] Ole Bauck. Hardware and software design for the dnv gl fuel fighter vehicles. Master's thesis, NTNU, 2015.

[8] Eric Brown. http://linuxgizmos.com/beaglebone-black-gains-720p-camera-cape/, 2013.

[9] Eric Brown. http://linuxgizmos.com/beaglebone-black-sbc-surpasses-100000-units/, 2013.

[10] Brad Chacos. http://www.pcworld.com/article/2886260/raspberry-pi-2-review-the-revolutionary-35-micro-pc-supercharged.html, 2015.

[11] Cliff. http://www.onepitwopi.com/raspberry-pi/gstreamer-1-2-on-the-raspberry-pi/, 2014.

[12] Drogon. http://wiringpi.com/, 2015.

[13] eBay. http://www.ebay.com/itm/New-Camera-Module-Board-5MP-Fish-Eye-Lenses-Wide-Angle-
151438284400, 2015.

[14] eBay. http://www.ebay.com/itm/Unlocked-HUAWEI-E220-HSDPA-UTMS-3G-USB-MODEM-Dongle-NEW
250818953808, 2015.

[15] enise shellproject.com. http://enise-shellproject.com/shell-eco-marathon/historique/, 2015.

[16] DNV GL Fuel Fighter. http://www.fuel-fighter.com/, 2015.

[17] Patrick Post/AP Images for Shell. https://www.flickr.com/photos/shell_eco-marathon/14196367312, 2014.

[18] Raspberry Pi foundation. https://www.raspberrypi.org/help/faqs/, 2015.

[19] Gadgetoid. https://github.com/Gadgetoid/WiringPi2-Python, 2015.

[20] Anders Lier Guldahl. Styre- og overvåkningssystem for shell eco-marathon kjøretøy. Master's thesis, NTNU, 2010.

[21] jimbojr. https://github.com/raspberrypi/hats/blob/master/backpowering-diagram.png, 2014.

[22] jimbojr. https://github.com/raspberrypi/hats/blob/master/hat-board-mechanical.pdf, 2014.

[23] Richard J Kinch. http://www.truetex.com/raspberrypi, 2015.

[24] Matt. http://www.raspberrypi-spy.co.uk/2012/06/simple-guide-to-the-rpi-gpio-header-an
2012.

[25] MicroChip. http://users.ece.utexas.edu/~valvano/Datasheets/MCP2551.pdf, 2003.

[26] NicolasFerre. http://www.at91.com/linux4sam/bin/view/Linux4SAM/SAM9M10Gstreamer, 2010.

[27] pelwell. https://github.com/raspberrypi/hats/blob/master/designguide.md, 2015.

[28] Shell. http://s00.static-shell.com/content/dam/shell-new/local/corporate/ecomarathon/downloads/pdf/europe/2014-results/sem-europe-2014-results-prototype-battery-electric-220514.pdf, 2014.

[29] Shell. http://www.shell.com/global/environment-society/ecomarathon/events/europe/2015-highlights/off-track-award-winners.html, 2015.

[30] thaytan. https://github.com/thaytan/gst-rpicamsrc, 2015.

[31] Jacek Tokar. http://raspberry-at-home.com/installing-3g-modem/, 2013.