NTNU – Trondheim
Norwegian University of
Science and Technology

Implementation of an Embedded Control System for Electronically Adjustable Suspension in a Formula Student Racing Car

Håkon Devold

Master of Science in Engineering Cybernetics
Submission date: June 2013
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Acknowledgements

"If we taught music the way we try to teach engineering, in an unbroken four year course, we could end up with all theory and no music. When we study music, we start to practice from the beginning, and we practice for the entire time, because there is no other way to become a musician. Neither can we become engineers just by studying a text book, because practical experience is needed to correlate the so called theory with practice."

- Charles F. Kettering

I would like to give a very special thanks to a few people:

- My advisor, Amund Skavhaug, for a lot of great advice when working with the thesis.
- All my teammates in Revolve for enormous inspiration, and great team spirit.
- A particular thanks to Hans Erik Frøyen and Christer Oldeide for designing the controller and mechanics of the system.
- SimPro, for invaluable for help with production and assembly of PCBs.
- Silje, for motivation, help and so very many other things.

Håkon Devold

Trondheim, June 24, 2013

Problem Description

In 2013 Revolve NTNU participates for the second year with a self-built racing car, in Formula Student. Formula Student is a competition for engineering students, where teams design and build a racing car. The cars will be evaluated in a series of tests on performance and design.

To improve the performance of their car, Revolve's R&D group wants to develop a system for adaptive suspension. A control system shall be implemented using embedded technology, and will be integrated with the other electronic systems of the car.

The candidate has already compiled a prestudy on how such a system may be designed. The candidate shall continue this work by:

- Revise solutions from this previous design
- Develop further requirements for the system
- Create hardware designs for the system modules
- Implement prototype devices for these system modules
- Develop the necessary software to achieve baseline system functionality
- As far as time permits it, extend this functionality to increase performance of the system
- Evaluate suitability, quality and performance of the final system

Supervisor: Amund Skavhaug, ITK

Summary

Good road handling is one of the most important characteristics of a racing car. To expand the capabilities of conventional suspension setups, the usage of adaptive suspension has become increasingly popular in high performance cars. Advancements in electronics and embedded technology has allowed such systems to be implemented with little weight gain over conventional suspension. The Formula Student team Revolve NTNU therefore wishes to implement a system for adaptive suspension on their car – the KA Aquilo R. A previous prestudy by the same author has evaluated the feasibility of such a solution, and proposed a top level design for an embedded control system.

This thesis will present the design, implementation and testing of a distributed embedded system to control a continuously controlled electronic suspension(CES). Based on the top level design proposed in the prestudy, a complete hardware design has been prepared for the system. The system was distributed into a Central Controller Unit, and Wheel Controller Units for each damper. A complete set of prototype system units has also been implemented, by the help of electronics manufacturer SimPro. The implemented hardware design has been tested and verified to be working.

Furthermore, a software implementation for the system's units has been implemented. For the Central Controller Unit a execution framework has been implemented, to allow the development of controller algorithm to be continued easily. The wheel controller units has implemented necessary software to control damping parameters of each shock absorber, according to supervisory control signals sent by the Central Controller. A common communication protocol to interconnect the CES system with the other electronic systems of Revolve's car has also been implemented.

Some elementary tests has also been performed to verify the performance of the system. The result is a a prototype system that may be used for evaluation of concept and on-car performance.

Sammendrag

Gode kjøreegenskaper er en av de viktigste egenskapene til en racerbil. For å utvide ytelsen til konvensjonelle understell har bruken av adaptive understell blitt stadig mer populært på høyytelsesbiler. Fremskritt innen elektronikk og embedded teknologi har ført til at slike systemer nå kan implementeres med lav vektøkning, sammenliknet med konvensjonelle understell. Formula Student-laget Revolve NTNU ønsker derfor å implementere et slikt system til bil – KA Aquilo R. En tidligere forstudie av samme forfatter har evaluert mulighetene for en slik løsning, og foreslått et mulig toppnivå design for et embedded styresystem.

Denne oppgaven vil presentere et design, implementasjon og evaluering av et et embedded styresystem for elektronisk justerbar dempning, også kalt CES. Basert på toppnivå designet foreslått i forstudien, har et fullstendig hardwaredesign blitt utviklet for systemet. Systemet er delt mellom en såkalt "Central Controller" enhet og "Wheel Controller" enheter som monteres ved hver demper. Et fullstendig sett av enheter for systemet har også blitt implementert, ved hjelp av elektronikkprodusenten SimPro. Hardwaredesignet også har blitt testet, og funksjonene verifisert.

Videre har en programvare blitt implementert for systemets enheter. For Central Controller enheten har et rammeverk blitt utviklet for kjøring av kontrollalgoritmen, slik at arbeidet med denne enkelt kan videreføres. Wheel Controller enhetene har implementert den nødvendige programvaren for å styre parameterene på de enkelte demperene, basert på kontrollsignaler fra Central Controller enheten. En felles kommunikasjonsprotokoll for sammenkobling av CES systemet med andre enheter på bilen har også blitt utviklet.

Noen elementære tester har til slitt blitt gjennomført for å vurdere systemets ytelse. Resultatet er en systemprototype som egner seg til videre evaluering av konsept og ytelse på bilen.

Contents

List of Figures

List of Tables

Glossary

CAD	Computer-Aided Design
CAN	Controller Area Network
CCU	Central Controller Unit
CES	Continuously Controlled Electronic Suspension
DC	Direct Current
ESD	Electrostatic Discharge
ESD	Floating-point Unit
IC	Integrated Circuit
ITK	Department of Engineering Cybernetics
KA	Kongsberg Automotive
LIN	Local Interconnect Network
MCU	Microcontroller Unit
NTNU	Norwegian University of Science and Technology
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative (Controller)
PLL	Phase Locked Loop
PSC	Power Stage Controller
PWM	Pulse-width modulation
R&D	Research and Development
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
WCU	Wheel Controller Unit

Part I

Introduction

1 Introduction

*"No one remembers who took second place,
and that will never be me."*

- Enzo Ferrari

Sections 1.1, 1.2 and 1.3 of this chapter have been obtained from the authors earlier work in [1].

1.1 Motivation

The market for embedded computers is exploding. As we speak, billions of devices using embedded technology are operating around us and predictions indicate that within a decade, the numbers of such devices will surpass the numbers of humans on the planet by a factor of 100.

The automotive industry has not been sitting on its hands. Solutions like CAN bus[2] has allowed embedded systems to be interconnected efficiently, and thus be distributed throughout cars. These days, everything from a light cluster to a door handle may have a microcontroller built into it.

In modern passenger cars electronically adjustable suspension has become increasingly popular. Volvo's 4-C Chassis and Audi's Magnetic Ride adaptive damping systems are two of several variants. By employing modern technology such systems may now be made less mechanically complex, and thus lighter. While it is mainly in the last few years that electronically adjustable suspension has come into usage in passenger cars, it has been used in racing cars for several decades. During the 1980's several teams in Formula 1 developed systems for active suspensions, up until a comprehensive ban in 1993.

With modern technology available, it now makes more and more sense to employ such systems also in lightweight racing cars like those used in Formula Student. Systems such as those considered in this thesis may improve handling and cornering abilities significantly, while still accounting for just a small fraction of total suspension weight. A key focus for the system presented in this thesis has therefore been weight and simplicity

1.2 Formula Student

Formula Student is an international engineering competition where the students design, build and compete with a racing car. The competition has been arranged annually for some 10 years, and has its roots in the American Formula SAE competition that uses the same homologation rules. A typical Formula Student car is shown in Figure 1.1.

Combined, these two competitions make up the worlds largest competition for university students by a considerable margin. In the 2012 competition, hosted at the Silverstone Formula 1 circuit, more that 134 teams participated. A similar number of teams compete in the American Formula SAE competition, and there are also derivatives in Germany and Australia attracting a growing number of teams.

Formula Student cars may use combustion powertrains with a maximum displacement of $610cm^3$, however a $20mm$ diameter air intake restrictor must be fitted to limit the achievable power. Alternatively cars may use electric powertrains, limited to a maximum power of $85kW$. As of the 2012 season, combustion and electric cars are competing head to head, motivating students to opt for environmentally friendly solutions. Additionally, there are also several rules concerning the construction of body, suspension, etc. The interested reader may find the complete rules in [3].

Event	Maximum score
Engineering design	150
Cost	100
Skid pad	50
1 km sprint	150
75 m acceleration	75
22 km endurance	300
Fuel efficiency	100

Table 1.1: Score distribution of Formula Student events

The participating teams compete in a number of static and dynamic events, each awarding a given number of points. These are distributed as in Table 1.1. The mixture between static and dynamic events require the team to both perform well on the track, as well as pay careful attention in the design and manufacturing phases of their project. Innovative features and designs are awarded, when the participating teams must justify their solutions in front of the judges, during a design presentation.

1.3 Revolve NTNU

Revolve NTNU is one of the two Norwegian teams competing in the Formula Student 2013. As with all Formula Student teams, Revolve is an ideal organization and relies on voluntary work from students. Financial support is delivered through sponsorships from the industry and businesses. The team is affiliated with the Norwegian University of Science and Technology(*nor.: NTNU*) and consists of 46 students, representing more or less all engineering programmes.

Revolve was first established in 2010, and entered the competition in the 2012 season with their KA Borealis R. At Silverstone, the team managed to earn the title of "Best Newcomer".

This year the team is entering the competition with an entirely new car – the KA Aquilo R, shown in Figure 1.1. The team have set themselves high ambitions, and is aiming for place among the top five teams in the competition.

Figure 1.1: The 2013 KA Aquilo R at its unveiling

1.4 Previous Work

During the fall of 2012, the author wrote a project thesis[1] on the subject of electronically adjustable suspension for use in Formula Student cars. The project thesis compiled a prestudy for how an embedded system may be used to control electronically adjustable suspension in such a context. This included an overview of specifications and requirements that such a system would need to fulfil, as well as an evaluation of possible designs and solutions for a top level system architecture. The prestudy also presented a selection of appropriate hardware components that could be used to implement such a system. The prestudy is available as a digital attachment to this thesis.

1.5 Scope and Outline

This masters thesis will describe the development of an embedded control system for adaptive suspension in a Formula Student racing car.

The results from the prestudy have been used extensively as basis for this masters thesis. Throughout this thesis, references will be made to the prestudy whenever it has been used as basis for choices during implementation. The reader is encouraged to seek further documentation in this document.

This thesis has been divided into four parts:

Part I will give an introduction to to the topic, and review the most important background theory that was presented in the prestudy. This should help the reader understand the motivation for the system, and some of the external requirements that apply. The part will also introduce the top level design that was chosen for the prestudy, and some of the requirements to the overall system. These requirements will form a basis for further design.

Part II will present the hardware design of the system. A set of hardware prototypes for the system modules has been developed, to be used for evaluation and software development. The recommendations for design and selection of hardware components given by the prestudy has been used where it was possible. This part will also give a presentation of how production of the final prototypes has been done. An overview of how the design has developed from an initial prototype, to the current design is also given.

Part III will give a presentation of the software that has been developed for the system. As the system presented uses a distributed architecture the software part will start with a description of the protocol that is used for communication between the units. Based on the top level requirements identified in Part I, system requirements will be presented for each module. A separate chapter has then been assigned to present the software implementation for each module.

Part IV will present a discussion and review of the design and implementation of the system. Results from some elementary performance tests has been included. The discussion will present a review of the solutions chosen for the system design. Finally some concluding remarks will be made to summarize the thesis.

Part V includes appendices showing large size schematics, PCB layouts and extracts of the program code for the developed software.

Complete program code has been included as a digital attachment for the thesis. The prestudy for the system is also available as a digital attachment.

2 Background Theory

"When I do retire, I know for a fact that I'll never be able to replace the incredible feeling I get when I'm driving a F1 car."

- Jenson Button, Formula One driver

As mentioned in Section 1.5 this thesis is based on the authors own prestudy on the same topic. This prestudy included an presentation of the relevant background material such as vehicle dynamics, suspension modelling, as well as the principles of active and semi-active suspension. This chapter will not present these topics in full, but rather summarize the most important parts of the contents that is directly relevant to the work presented in this thesis. The interested reader is referred to the prestudy itself[1] for further information. While these topics are not directly connected to the rest of the thesis, basic knowledge is necessary to identify the motivation for the CES system, as well as external requirements with respect to reliability , real-time properties and system functionality.

2.1 Suspension

Chapter 2 of the prestudy gave an introduction to the basic principles for automotive suspension, suspension modelling and evaluated possible control strategies for a CES system. This section will give a summary of the content in that chapter to familiarize the reader with basic theory of automotive suspension.

2.1.1 Suspension Modelling

Mathematical models are widely used to understand and tune the behaviour automotive suspensions. The simplest form such these popular models is called a *quarter car model.* A conceptual illustration of such a model is shown in Figure 2.1a.

This model isolates the suspension of one single wheel, and is limited to modelling vertical behaviour, i.e. along the z-axis. As can be seen from the figure, the mass of the vehicle is divided into the following two parts

- The wheel of the car(m_u), which is suspended only by the tire
- The car body(m_s), which is suspended on top of the wheel by a spring and a damper.

Under the assumption that there are no external forces on the system, a mass suspended by a spring and damper can usually be modelled as a simple second order differential system

$$m_s\ddot{z_s} + c_s\dot{z_s} + k_s z_s = 0$$

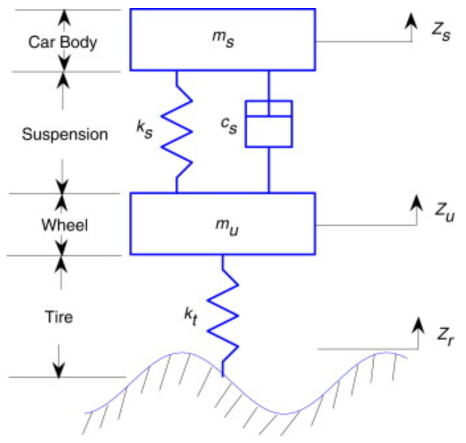
In the above equation m_s is the suspended and x_s is its position. c_s and k_s are respectively the damper and spring constants.

In the real world, however, important to notice that the deflection in the tyre wall introduces a significant amount of damping into a suspension system. For sufficient accuracy, this effect must be included into the model. This gives the following system

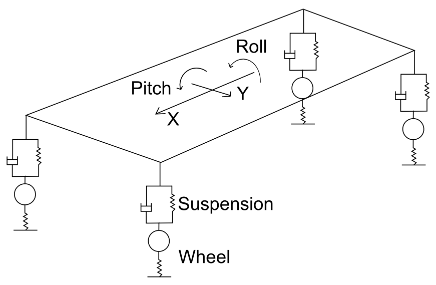
$$\begin{aligned} m_s\ddot{z_s} + c_s\dot{z_s} + k_s z_s &= c_s\dot{z_u} + k_s z_u \\ m_u\ddot{z_u} + c_s\dot{z_u} + (k_s + k_t)z_u &= c_s\dot{z_s} + k_s z_s + k_t z_t \end{aligned}$$

This is no longer a simple second order system. If combined, these equations will yield a non-linear equation of motion. Consequently, tuning of suspension is a task that require much effort.

The system implemented in this thesis will allow adjustment of the damper c_s, suspending the sprung mass m_s. As we can see from the equations, the behaviour of the system may be tuned be adjusting this parameter.



(a) Quarter car suspension model[4]



(b) Full car suspension model

Figure 2.1: Suspension models

To study the kinematic effects on the vehicle as a whole, a common procedure is to expand the quarter car model into a *full car model* . This model combines four quarter car models with a rigid body with known moments of inertia around the $\vec{x}$ and $\vec{y}$ axes. The rotation around $\vec{x}$ and $\vec{y}$ axes, called correspondingly *roll* and *pitch*, may then be simulated.

This extension will allow modelling of behaviour when the body is subjected to external force during e.g. acceleration, braking or cornering. As vehicle dynamics is not a primary part of this thesis, the complete set of equations will not be presented here. However, a conceptual illustration of the full car model is shown in Figure 2.1b.

2.1.2 Active and Semi-Active Suspension

This section is a summary of the authors previous work presented in the prestudy[1, Ch. 2.3].

Popular jargon tends not to differentiate between active and semi-active suspension. The terms Active- and Adaptive- suspension is often used interchangeably. In the perspective of suspension theory, however, the differences between these techniques are crucial.

The principle of ***semi-active suspensions*** is similar to that of conventional i.e. passive suspensions. Section 2.1.1 describes the principle of the quarter car model. In this context, a semi-active suspension will allow suspension parameters such as

k_s Spring stiffness

c_s Damper coefficient

z_{s_0} Ride height

to be adjusted dynamically, based on the state of the vehicle. Semi active suspensions are therefore, strictly speaking, also passive systems as they cannot add energy to the suspension. Different control strategies for this type of suspension is described in Section 2.1.3.

Active suspensions may on the other hand add energy to the suspension. This may be done by the means of an actuator driven by i.e. hydraulics or electricity. This often allow elimination of traditional suspension components, such as springs and dampers altogether, as suspension force may be synthesised by the actuator. This type of suspension has not been considered in detail in this thesis. However, if the suspension is to synthesize damper and spring forces it is imperative that a high bandwidth controller is necessary.

2.1.3 Control Objectives

Chapter 2.5 of the prestudy presented several possible control strategies for an adaptive suspension system. While the design of a control strategy for adaptive suspension is not part of this thesis, this section will repeat the most important control strategies that are relevant for this system. Section 2.1.2 introduced the concepts of active and semi-active suspension. This section will, however, focus on control objectives for semi-active suspension, such as those considered for the system in this thesis.

The most basic form of adaptive suspension is one that will allow automated ***manual adjustment*** of suspension parameters. This may be realized through binary settings, or an adjustable range for parameters. The suspension may be adjusted at the touch of a button, rather than performing mechanical adjustment to the suspension itself. Such system are common in many upmarket passenger vehicles. As no dynamic control is implemented, the system may be implemented with few requirements to hardware and software performance.

If a control system is implemented for the adaptive suspension system, the suspension may change its parameters continuously according to the state of the vehicle. Different types of vehicles may define different targets for such control. For instance, a passenger car may employ such system to increase comfort.

In motor racing, however, such systems are better suited for improving the handling of a vehicle. When a car accelerate, brake or corner its inertia shifts the weight of the vehicle away from its static centre of gravity. This will cause the vehicle to *roll* or *pitch* so that some tyres are subjected to more load than others. Thus, the suspension of the vehicle will have to be tuned to a compromised setting that allow the suspension to handle all these situations. In reality however, different setting for each state would be optimal.

Different types of controller strategies may be implemented depending on the bandwidth available from the control system. Simulations by other team members have indicated that a bandwith of 10-50 Hz is sufficient to correct slow movement in the vehicle body, i.e. ***pitch and roll***. High bandwidth control may be used to to eliminate the effect of road surface anomalies. A widely employed control strategy for such corrections is called ***ground-hook control***. The interested reader is advised to seek further information about suspension control strategies in Chapter 2 of the prestudy. A more comprehensive collection of information may be found in [5]. Advanced controllers like ground-hook will require bandwidths that are several orders of magnitude higher, i.e. 500 Hz and upwards.

3 Top Level Design

"I remember my first test in F1. After five laps, I came back to the pits and tried to play it cool - 'Oh yeah, I'm fine, I'm on top of this' - but I was completely lost."

- Sebastian Vettel, Formula One driver

This chapter will present the top level design that has been used for implementation of the CES system. The top level design has to a large extent been based on recommendations from the authors own work presented in the prestudy[1]. However, as more detailed specifications is now available for the rest of the system, the top level design has been reviewed to comply with this information.

3.1 System Requirements

3.1.1 Damper Actuation

The prestudy performed an evaluation of the most popular physical principles for use in a semi-active suspension system for a formula student car. The alternatives that were evaluated are listed below.

- Magneto-rheological dampers
- Pneumatic and hydraulic dampers

- Valve actuated dampers

Of the above alternatives, a system using valve actuated dampers was deemed to be the most reasonable choice. Such a system would require little modification to the off-the-shelf dampers that were already used in Revolve's Formula Student car. As the adaptive suspension system is a development project, it is important to have a working backup solution if it should not produce the desired results.

The system considered in this thesis is therefore designed so that it may be mounted onto a stock Öhlins TTX25 FSAE MKII damper with no modifications. In this way, the system may be installed or removed from the car in a matter of minutes.

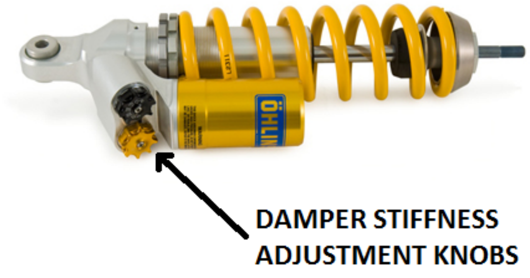


Figure 3.1: Öhins TTX25 FSAE damper

As can be seen from the Figure 3.1, these dampers are equipped with two externally adjustable valves. These valves adjust the damping coefficient of the damper, when being either compressed or expanded(also referred to as rebound). Two actuators are therefore needed to adjust each parameter freely. Each knob may be rotated a total of 4.5 turns($\approx 1680 deg.$), to change from a fully open to a fully closed valve.

For this purpose, two brushed DC motors have been chosen. The motors that will be used in this system are of the type Maxon RE-max. A comprehensive datasheet for the motor may be found in [6, p. 121]. The DC motors will be combined with a planetary gear-head, with a ratio of 5.4 : 1. The datasheet of the gear-head may be found in [6, p. 217]. A summary of specifications for the motor and gear is given in Table 3.1.

Motor article nr.	216000
Gear article nr.	118184
Nominal voltage	12V
Nominal speed	8220 rpm
Stall torque	14.4 mNm
Stall current	1.45 A
Max continous current	0.415 A
Torque constant	9.93 mNm/A
Gear ratio	5.4:1
Stall torque w/gear	77.8 mNm
Nominal speed w/gear	1522 rpm

Table 3.1: Specifications of Maxon RE-max 17mm motor and gearhead

Under the assumption that acceleration time of the motors are negligible, the geared nominal speed of $1522rpm$ gives us a maximal delay time of $4.5 \cdot \frac{60000}{1522} = 177ms$, from fully open to fully closed valve.

Simulations by the developers for the supervisory control algorithm concluded that the controller frequency for a PID motor controllers should exceed $500Hz$.

3.1.2 CAN Interface

An important feature of the CES system is its ability to efficiently read data from sensors placed on the car. The KA Aquilo R features a data acquisition system, that is used for monitoring and gathering data while the car is being driven. The data aquisition system of this years car is based on CAN bus. A special CAN Sensor Unit has been developed for the data acquisition system, that allows the connection of either a digital or analogue sensor. Sensor values will be converted to a 16-bit value and transmitted as a CAN message.

CAN messages are broadcast on the CAN bus, and may be read by any unit connected to the bus. By utilizing this already installed CAN bus, we may easily implement data acquisition for the CES system with little need for extra hardware. Means to connect the CES system to the data acquisition CAN bus should therefore be implemented.

As will be described further in Section 3.2 the system will be implemented using a distributed architecture. As a result, there will also be a need for the units to communicate. As a CAN bus is already available throughout the car, it should be used for system interconnection in further development.

3.1.3 Ingress and ESD Protection

A system that is mounted on a car is exposed to a harsh environment. Rain, dust and high temperatures are some of the factors that the system must be designed to handle. For the CES system, the two most important factors are protection against water and dust. To check the encapsulation electronics, cars are spray tested at the competition. Figure 3.2 shows a Formula Student car being sprayed with water at the skid pad test. All system units much therefore be mounted with sufficient ingress protection.

Another important factor for a system to be mounted in a car is protection against electrostatic discharge. Electronics mounted inside a non-conductive enclosure will generally be well protected against ESD. It is, however, important that the external interfaces of all units also are protected against discharges and transients.

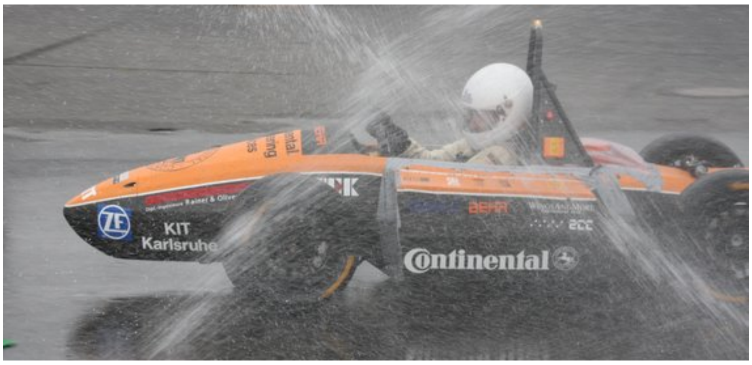


Figure 3.2: Formula Student car during water spray test

3.1.4 Fail-Safe Mode

In motor racing, cars are pushed to their limits to find extra seconds out on the track. The margin for error is consequently much smaller than in a passenger car, as drivers has to rely heavily on their prediction of how their car will behave. The system described in this thesis will have the direct ability to change the handling parameters of the car. Any abrupt changes in the cars handling may therefore have huge consequences. A driver loosing control of the car may lead to damages to the car, as well as injuries to both the driver and spectators.

To avoid such consequences it is important that any failures of the system should not cause unpredictable behaviour. Any failures that may lead to decreased system performance should also be identified, so that preventive measures may be taken. It is also vital that the driver is made aware of the error as quickly as possible.

To meet these requirements, the system needs to implement mechanisms for error detection. The prestudy proposed to exploit the fact that the system uses a distributed architecture to implement error detection. This requirement has been somewhat extended during the for safe implementation of the system, giving error detection on two levels:

Internal error detection in each module: A watchdog timer should monitor program execution internally in each module(both CCU and WCUs). If the program execution stops, the watchdog timer should reset the processor of the module, so that the program may be restarted.

External monitoring between the units: The different units of the system should mutually monitor each others states. Messages should be sent periodically to indicate that there are no errors. All messages should include error bits, that may explicitly signal the type of error that has occurred.

3.1.5 Control Panel & Configuration

It is necessary to implement a way for the driver to interact with the system, when it is mounted in the car. Driver interaction may range from elementary features, such as activation and disabling of the system and changing operation mode, to more complex operations such as changing system control parameters. The electronics group has developed a driver interface in the steering wheel and dashboard so that the driver may interact with the various electronic systems of the car. The driver interface is connected with the electronic modules of the car using CAN bus.

A 3-way switch in the driver interface has been allocated to operate the CES system. This switch should be used for elementary operation of the system, such as the selection between two system modes and disabling the system. As described in Section 3.1.4 it is also necessary to implement a way to alert the driver in case of failure of the system. A warning led has therefore also been allocated to the CES system.

For more complex configuration of the system, a OLED screen has been implemented in the steering wheel of the car. Menus may be implemented for this screen to alter system parameters. The dashboard and steering wheel of the car is shown in Figure 3.3. The car is also equipped with a bidirectional wireless data link, that may be operated using a computer at the track-side. This telemetry system is also connected to the CAN bus, and may also be used for configuration of the system.

3.1.6 Bootloading

Given a distributed architecture, as will be presented in 3.2, the wheel actuating units will be mounted inside an enclosure on each damper. Access to these units may therefore be restricted. To easily be able to reprogram the units of the system, a way to load a new program over the CAN bus should be implemented.

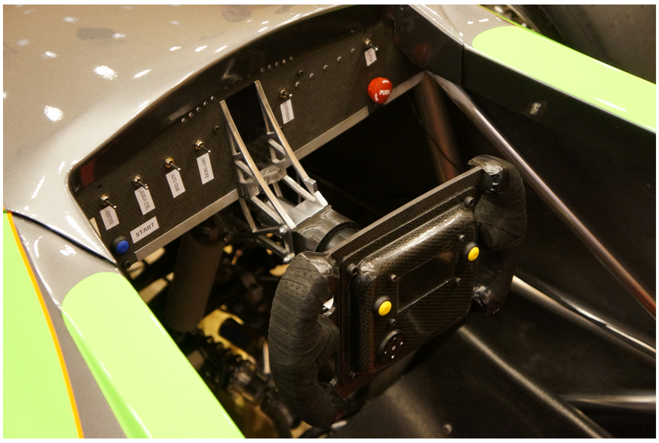


Figure 3.3: Dashboard and steering wheel of KA Aquilo R

3.2 System Architecture

Chapter 3 of the prestudy evaluated several possible system architectures for the CES system. A distributed system architecture was recommended for further development of the CES system. Such an architecture is suitable for this system, as actuators will have to be mounted at the dampers. The dampers are normally mounted at each corner of the car, and using a distributed architecture would therefore reduce the need for cabling, as all units may be connected to a common data-bus.

The distributed architecture suggested in the prestudy was thus adopted for further system development. The chosen system architecture is shown Figure 3.4. A short description of the two types of modules used in the system is given below.

The Central Controller Unit, or CCU, should act as the main controller unit for the system. This unit gathers sensor data, and generates supervisory control signals for the units that are mounted locally at each damper. The CCU will also communicate with control panels and other electronic equipment of the car. It should also monitor the state of the Wheel Controller Units.

The Wheel Controller Unit, or WCUs, will be mounted locally at each damper, and control the actuation of that particular damper. The WCUs will implement two PID controllers for actuator control, and receive new setpoints for these controllers periodically from the CCU. The WCUs should also be able to operate independently in case of failure in the CCU.

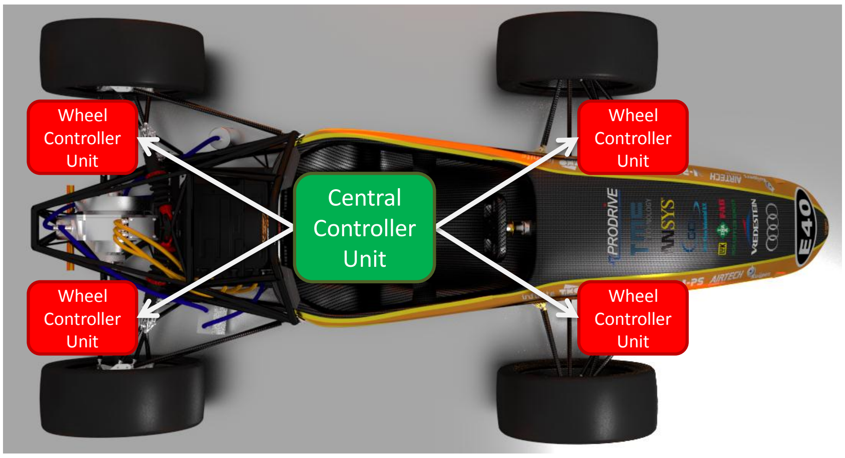


Figure 3.4: CES system distribution in a Formula Student car

Part II

Hardware

4 Introduction to Hardware Part

"Hardware: The part of a system that you can kick."

- Anonymous

Development of hardware for the CES system has played a major role in the work with this thesis. During early development stages, development boards and breadboards were use to test compatibility and performance of single components. However, to create a system that is efficient with respect to size, cost, etc. it is necessary to develop custom designed hardware that will fulfil the requirements for each module.

This part will give a description of the hardware that has been developed for the CES system. Different chapters has been assigned to the description the Central Control Unit and the Wheel Control Unit introduced in Section 3.2. The design of each module has been broken down into four stages. First, a set of specific requirements will be presented for each module. Once these requirements was identified, a complete set of schematics has been developed for all units. The prestudy[1, Ch. 5] presented a selection of components that could be used for further hardware development of system units. To a large extent, the components chosen for the final design has been selected on the basis of these recommendations. The work presented in this part has therefore primarily focused on how each component should be integrated into the design.

Based on the developed schematics, complete PCBs has been designed and implemented for all modules. The finished modules has also undergone elementary tests to verify the hardware design. All schematics and PCBs has been developed using Altium Designer. This is a state-of-the-art software package that may be used for simple prototyping, up to full scale design of advanced electronic systems. For reference the exact version used has been `ver. 27009`.

Furthermore, a chapter has been added for the description of the vehicle network that will be used for communication between system units. The part will also give a presentation of how the final prototypes has been produced, as well as a description of how the system has been installed in the car for testing. After the hardware design has been presented, Chapter 9 will discuss the design of the system. Complete schematics and PCB layouts for the system is available in appendices A and B.

5 Vehicle Network

"Obstacles are things you see when you take your eyes of your goal."

- Henry Ford

5.1 Requirements

Given the system architecture presented in Section 3.2, it is clear that the system need an efficient way to interconnect the Central Controller Unit and the Wheel Controller Units. The top level design requirements specified that CAN bus should be used to interconnect the CES systems modules. This decision was made as a joint decision within the 2013 Revolve team, on the basis that CAN is a widely available and lightweight standard for embedded networks. The interested reader should refer to Chapter 4 of the prestudy for more information about this decision, and details about the CAN bus standard.

Each unit using CAN communication need not be able to access the internal communication of other systems of the car. A common decision was also made within Revolve to separate the network into several buses. For the following chapters the term *CAN bus* will be used to describe a single bus, while the term *CAN network* will be used to describe all can buses that are used together in a system.

The data acquisition system, with more than 20 sensor modules, is by far the most comprehensive subsystem of the car. To not occupy a large amount of capacity of the CAN bus with this communication, it was decided that a separate bus should be made available for the data acquisition system. As presented in Section 3.1.2 an important feature of the CES system is to receive appropriate sensor data. It would therefore seem logical to connect the CCU directly to the data acquisition CAN bus. Chapter 4 of the prestudy also evaluated the bandwidth and real time properties of CAN. It concluded that the implementation of a CES system with control frequency of 100 Hz, combined with a high frequency data acquisition system, still only used a fraction of the total bus capacity.

The following two buses have been deployed on the car:

The Main CAN bus, also referred to as ***CAN0***, will act as the backbone of the electronic system of the car. All electronic systems such as Engine Control Unit, Gear Control Unit, Datalogger and dashboard will be connected to this bus. The activity on this bus will mostly be driven by events such as e.g. status change from dashboard, or the passing of commands between electronic modules. However, units will also broadcast status messages on this CAN bus periodically.

The Sensor CAN bus, also referred to as ***CAN1***, is primarily used for the data acquisition system of the car. As described in Section 3.1.2, all sensors will be connected to the data acquisition system using a CAN Sensor Unit. Activity on this bus will to a large extent be periodical, as each sensor module will broadcast its value with a predefined interval. As sensors are mounted throughout the car, this bus will also have to span through the entire length of the car, to allow connection of all sensors. This bus will also be used for interconnecion of the CES system.

5.2 Implementation

While the CAN 2.0 specification does not specify the properties of the physical layer for CAN bus, the two wire connection defined by ISO11898 standard has become the de-facto standard for CAN physical layer. To allow as much bandwidth as possible, a data rate of $1Mb/s$ was defined for the CAN network. This is the maximum data rate refined by ISO11898, and also the maximum data rate supported by most CAN controllers and transceivers. The real-time properties is, as for all shared mediums, tightly connected to grade of utilization of a CAN bus. A higher data rate giving a lower utilization, will thus contribute to stable real-time properties.

ISO11989 does not specify the physical properties for connectors and wires used for a CAN bus, however it is common practice to use a twisted copper pair cable. For applications where the data-rate approaches $1Mb/s$ it may also be necessary to use a shielded cable to improve resistance to electromagnetic noise.

In a car shielding of cables is particularly important, as there are typically several sources of electromagnetic radiation, such as ignition coils, that may disrupt communication of caution is not taken. The use of shielded twisted pair cable was therefore chosen for all CAN buses in the wiring harness. Further details on implementation of the CAN physical layer may be found in [7].

The exact cable chosen for this application was of model 2401C by Alpha Wire, as shown in 5.1. The 2401C has stranded wires as which allows a cable with relatively low bending radius, which is necessary for installation in a car. The cable is also equipped with foil shielding, which gives a low outer diameter and high flexibility. The complete datasheet for the cable is available in [8, p. 314]

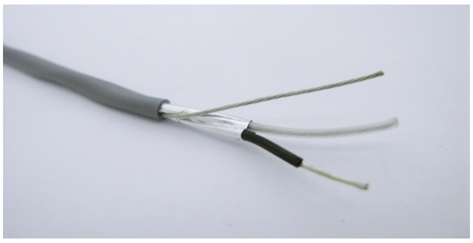


Figure 5.1: Alpha Wire 2401C twisted pair cable

6
Central Controller Unit

"There is nothing in machinery, there is nothing in embankments and railways and iron bridges and engineering devices to oblige them to be ugly. Ugliness is the measure of imperfection."

- H.J. Wells

A hardware design for the Central Controller Unit has been designed and implemented during the work with this assignment. This chapter will present the hardware that has been developed for this unit. The chapter will start by defining the requirements that apply to the hardware design for this unit.

6.1 Requirements

6.1.1 Power Supply

The need for a stable power supply is imperative for the implementation of a critical system like the CES system. In this case, the system will be mounted on a car. When installed in the car, the system will be powered through a common wiring harness delivering both CAN bus and 12 Volt supply. The wiring harness will be fed by a battery and generator. A power supply must be implemented to allow noiseless and stable voltage to the system.

As will be discussed in Section 6.2.6 the Central Control Unit has implemented a SD card interface to save system log files, store system settings, etc. While all other components of the CCU have been selected to use a 5V power supply, SD cards are standardized to use a supply voltage of 3.3V[9]. A second power supply is thus needed for this purpose.

6.1.2 Microcontroller

As described in Section 3.2, the Central Controller Unit will execute the main control algorithms for the adaptive suspension system. With respect to further development of the system, it is important that this unit is capable of handling computationally complex algorithms with relatively high bandwidth. An example might be the ground-hook controller described in Section 2.1.3.

It should also be made possible for other students specializing in other fields of study to continue development without needing expertise within embedded systems. Most microcontrollers do not have hardware support for floating-point arithmetics. When implementing advanced mathematic algorithms, this may be considered a significant drawback. The developer must choose between significantly reduced performance, or investing extra time in converting controller algorithms to integer numbers. A paper evaluating the performance of the AVR32 architecture for control applications[10] suggests that the floating-point performance of an AVR32 microcontroller without an FPU was only 25% of the measured integer performance for the same unit. This issue should therefore be addressed when a microcontroller is selected

6.1.3 Communication

As described in Chapter 5, the CCU shall be connected to two CAN buses: The *Main CAN bus* and the *Sensor CAN bus*. Hardware to connect to these two CAN buses must therefore be implemented. Specifications of this hardware must be according to those presented in Section 5.2.

When developing an embedded system like the one in this thesis, an easy way to debug the system is essential. As described in Section 6.2.1, the microcontroller is paired with a JTAG debugging port. However, this type of debugging is cumbersome to use when the system is deployed on the car. Debugging with JTAG also requires normal program flow to be halted, which will disrupt the normal operation of the system. Because of this, the prestudy suggested to implement an RS-232 interface that may be used for debugging and execution monitoring of the system.

6.1.4 Configuration

An SD card connector should be provided in the hardware design of the CCU. As described in 3.1.5 a SD card may be used for storing configurations and saving operation log-files for the CES system.

6.2 Implementation

6.2.1 Microcontroller

MCU

The prestudy concluded that an Atmel AVR32 UC3C 32-bit microcontroller would be well suited for use in the Central Controller Unit. This series of microcontrollers feature a built in floating-point unit with 32-bit precision. This will simplify the implementation significantly, as algorithms can be implemented in application code without significant alterations. Chapter 7 of the prestudy also evaluated the possibility of using Matlab Embedded Coder for auto-generation of program code. Auto-generated code, like this may benefit from hardware with floating-point support.

The AVR32 UC3C is available in several different packages, with combinations of QFN or QFP design, and 64, 100 or 144-pin packages. For this project, prototyping had to be done locally at the university. The types of packages are shown in figure 6.2. As the QFP package has exposed legs, it may be soldered on to the PCB using a regular soldering iron. The

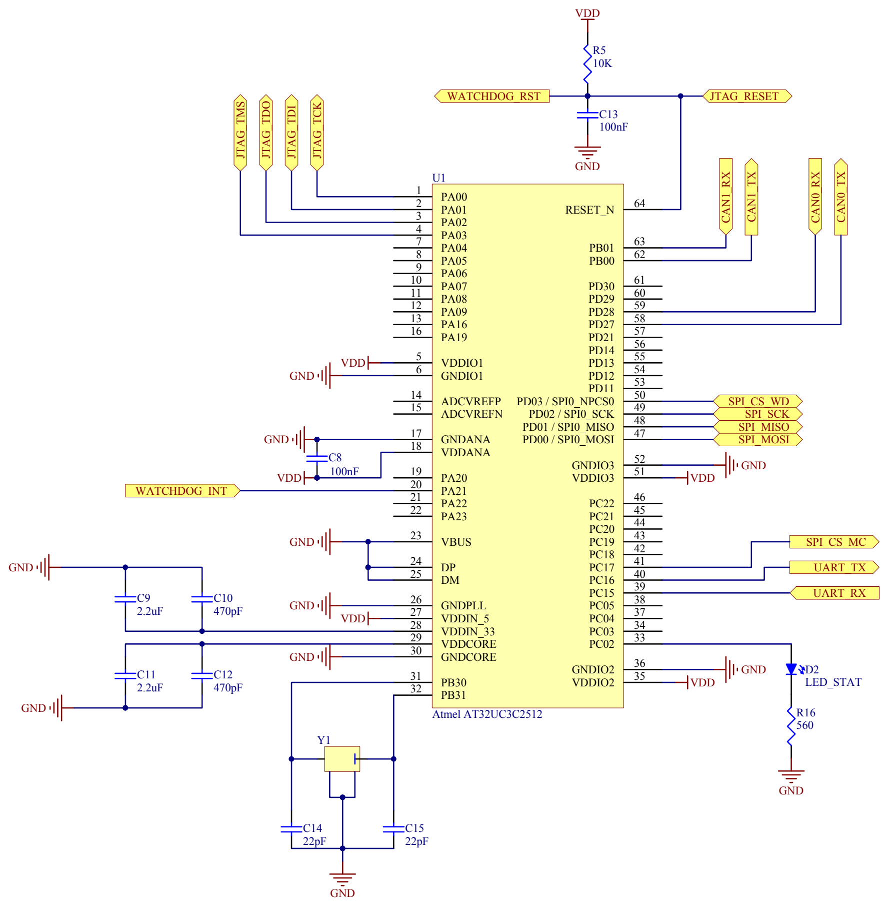
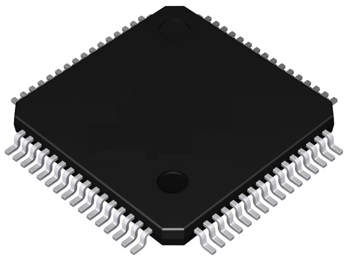


Figure 6.1: Microcontroller schematic

QFN package, on the other hand, requires hot air soldering equipment for assembly. On this basis, a QFP package was chosen for the circuit design.

A list of the peripheral units connected to the microcontroller is found in Table 6.1. Additionally, a surface mounted miniature LED, designated `LED_STAT`, has also been connected to the MCU for debugging purposes.

As can be seen from the table, all peripheral units are connected to the microcontroller using serial data-buses. Few parallel I/O are used. A


(a) QFP Package

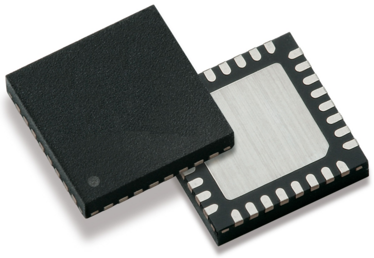

(b) QFN Package

Figure 6.2: Comparison of microcontroller packages (Not to scale)

Unit	**Bus**
SD Memory Card	SPI
System basis chip	SPI + $\overline{\text{INT}}$ + $\overline{\text{RST}}$
CAN0 Transceiver	CAN RX/TX
CAN1 Transceiver	CAN RX/TX
RS-232 Transceiver	UART RX/TX

Table 6.1: Peripheral units connected to CCU microcontroller

64-pin package was therefore deemed to be sufficient for this application. As can be seen from Figure 6.1, there are still several spare pins. To not limit space for future extensions, the model with most program memory, of 512kB, was chosen for the application. The exact model number for the microcontroller used is Atmel AVR32 UC3C2515.

Oscillator

To drive the clock of the microcontroller accurately an external oscillator is needed. [11, p. 83] specifies a maximal frequency of 20 MHz for external oscillators connected to ports `OSC1` and `OSC1`. The AVR32 UC3 series feature internal two internal PLL generators. These may be used to achieve the maximal specified clock frequency for the UC3C of 66Mhz. To achieve maximum performance, a clock as close as possible to 66 MHz is desired.

A crystal oscillator of 16Mhz was chosen to drive the microcontroller. This is a widely available value for crystal oscillators. A PLL multiplier of 4× will be used in software to generate a system clock of 64Mhz – close to the rated clock frequency of the UC3C.

Programming Port

The UC3C has implemented two programming interfaces; the open JTAG standard, and the proprietary Atmel aWire. Both interfaces also allow debugging of the microcontroller. While the aWire interface can be implemented with only 3 wires, the JTAG interface requires a 10-pin header for connection to a programming adapter. The pins used for aWire are, however, a subset of the pins used for JTAG debugging.

As board space is not critical for the CCU, it was decided to implement a full JTAG interface. The connection of the JTAG interface is shown in Figure 6.3, while the corresponding connection to the microcontroller is shown in Figure 6.1.

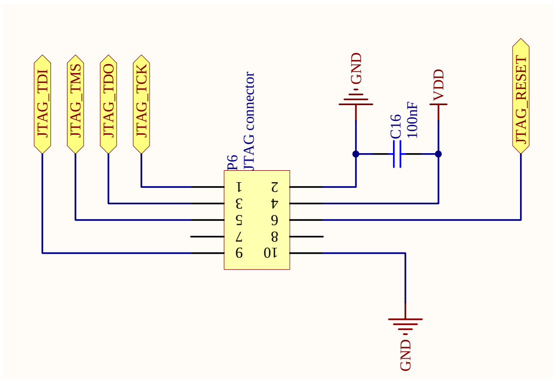


Figure 6.3: JTAG programming port

6.2.2 System Basis Chip

The prestudy suggested to use a so-called *System Basis Chip*(hereafter abbreviated SBC) for the Central Controller Unit. System Basis Chips supplied by Freescale Semiconductor are ICs combining the following functions into a single chip.

- Voltage Regulator
- CAN Transceiver
- External Watchdog

These units are targeted towards the automotive market, where embedded units are often connected using CAN or LIN bus. The watchdog, as well as other configurations, are managed by the microcontroller using a SPI interface. A block diagram of the SBC is shown in Figure 6.4.

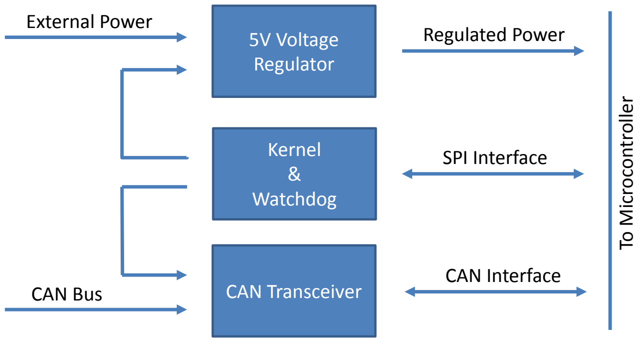


Figure 6.4: Block diagram of system basis chip

The Freescale MC33989 SBC recommended by the prestudy has, however, been updated to a new second-generation model – MC33903. The functionality of this second generation model is, for this purpose, identical to the previous version. The upgrade has mainly been done to reduce chip size, as the same SBC is used for the Wheel Controller Units.

5V regulator

When several units are connected to the same power supply, changes in current load may cause undesirable noise and spikes in the supply voltage. To prevent this from effecting the operation of the CCU, the unit is equipped with an internal voltage regulator, as shown in Figure 6.5. As can be seen from the figure, the voltage regulator is incorporated the SBC.

The MC33903 is available with 3.3V or 5V voltage regulators. The Atmel AVR32 UC3C may also be powered using either 3.3 or 5V. Nevertheless, in this application a 5V power supply was chosen. The signal-to-noise ratio for a signal is defined as

$$SNR = \frac{P_{noise}}{P_{signal}} = \left(\frac{A_{noise}}{A_{signal}}\right)^2 \tag{6.1}$$

where P is the power of each component, and A is the corresponding amplitude. From the formula we can easily see that the SNR increases with the amplitude of the signal squared, when the amplitude of the noise is kept constant. Increasing the system voltage from 3.3 to 5V increases the SNR by a factor of 2.3. A 5V circuit should consequently be more robust against noise. The internal voltage regulator of the MC33903 is capable of delivering a current of maximally 300mA at 5 volts. This should be sufficient for powering the components of the CCU.

As can be seen from the schematic a LED, designated `LED_PWR`, has also been connected to the 5V supply of the CCU. This way, the user can easily verify that the power supply is operative.

CAN Interface

The built in CAN transceiver in the SBC will be used for communication on the Main CAN bus. The CAN transceiver of the MC33903 is rated for bit rates up to 1Mb/s, such as specified in Section 5.2.

Termination of the CAN bus has been implemented at the CAN interface of the SBC. This transceiver allows a so called *split terminaction*, implemented

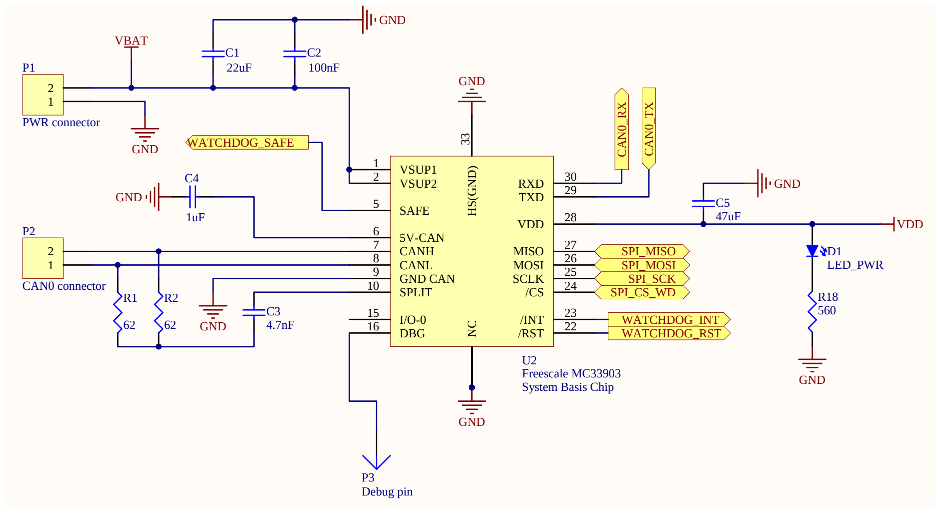


Figure 6.5: System Basis Chip with 5V main power supply

with resistors R1 and R2 in Figure 6.5. Instead of using a typical 120Ω resistor, as is normal on CAN bus, two resistors in series has been used. The middle point of this voltage divider is connected to the SPLIT pin of the SBC. If termination should not be desired at this unit, the termination circuit may be removed altogether.

Watchdog

As described in Section 3.1.4 each module should implement a mechanism for internal error detection. For the CCU, this has been implemented by using an external watchdog circuit built into the SBC. The watchdog may halt the microcontroller in two ways:

Reset: The SBC is connected to the reset pin of the UC3C microcontroller. If the microcontroller fails to refresh the watchdog timer, the reset pin will be pulled low to reset the microcontroller. The watchdog timer is reset using the SPI interface, as will be described in greater detail in Chapter 12.

Power down: After power up the microcontroller has 10 seconds to reset the watchdog timer of the SBC. This initializes the watchdog, which then will have to be reset periodically. If the watchdog is not initialized within the first 10 seconds, the SBC will disable its power supply. The power of the circuit will have to be toggled to re-enable the power supply.

Obviously, the microcontroller will be unable to reset the watchdog periodically during events such as programming and debugging. A special debug mode is therefore implemented in the system basis chip. As can be seen from Figure 6.5 a special debug pin, marked `DBG`, is available. If a voltage between 8-10 volts is applied to this pin, the SBC will enter a so called *debug mode* that disable its watchdog. The SBC may also be set in a special *flash mode* that extends the refresh period of the watchdog timer to approximately 30 seconds.

6.2.3 3.3V Regulator

The current consumed by a SD card may vary from a few milliamperes, to up to 200mA during data writing[9]. A simple voltage divider is therefore not sufficient for powering the SD card. Instead, a conventional linear regulator has been used. The regulator chosen is a 3.3V LDO regulator from Texas Instruments, with model number TLV1117. The datasheet of the voltage regulator[12] specifies a maximal continuous current of 800mA. This is sufficient for powering the SD card. As described in Section 6.2.2, SBC is able to toggle the power supplied to the system. As can be seen from Figure 6.6 the TLV1117 is connected with VDD(5V regulated voltage) as its supply voltage. This allows the SBC to also toggle the power supply to the SD-card.

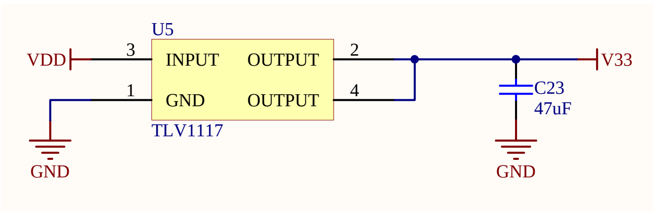


Figure 6.6: 3.3V secondary power supply

6.2.4 CAN Interface

As the CCU must connect to two CAN buses, the transceiver in the MC33903 alone is not sufficient. The prestudy recommended that a MCP2551 transceiver from Microchip should be used to interface the Sensor CAN bus. For the actual implementation, an ADM3051 transceiver from Analog Devices has been used. This transceiver is pin-compatible with the MCP2551, as well as transceivers from Texas Instruments, Maxim and other manufacturers. This ensures freedom to easily change this component if the work is to be continued at a later point of time.

The CCU has not implemented any means for terminating the CAN bus locally at the board. If desired, this may be done with a through-hole resistor when the bus wires are attached to the PCB. The ADM3051 also implements circuitry for slope control. This allows the maximal slew rate of the input signal to be limited, for improved noise rejection. This is particularly useful for applications in extremely noisy environments, or in applications where the use of a shielded cable is not possible.

To adjust the maximal slope, a resistor should be applied between the `RS` port of the ADM3051 and ground. For this particular application a shielded cable has been used, and practical use has not indicated any issues with noise. The slope control has therefore been disabled by grounding the `RS` pin.

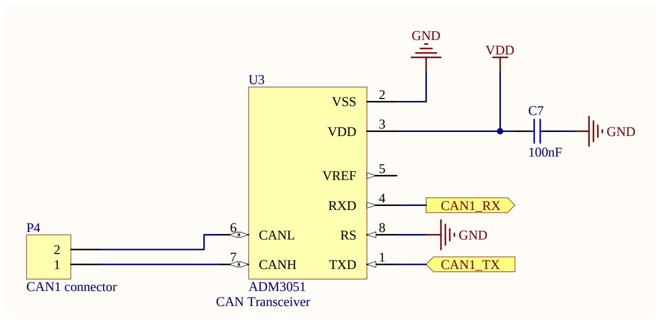


Figure 6.7: Second CAN Transceiver

6.2.5 RS-232 Interface

As can be seen from Figure 6.12 the RS-232 port is connected to a standard DB-9 connector, mounted on the outside of the enclosure of the CCU. It was therefore important to chose a transceiver with ESD protection, so that internal circuits may not be damaged by electrostatic discharge when external equipment is connected. For this application, an ADM202E transceiver from Analog Devices has been selected. The datasheet of this transceiver is available in [13].

The standard signalling levels of RS-232 use both positive and negative voltages. A voltage level between $+3V$ to $+15V$ to ground represents a high bit, while a voltage level of $-3V$ to $-15V$ represents a low bit. The interval $-3V$ to $+3V$ is not valid. As only a single $+5V$ power supply is available in this application it was necessary to choose a transceiver with a built in voltage converter, to eliminate the need for external components. The ADM202E uses signalling levels of -15 and $+15$ volts by using an internal voltage converter. External capacitors(`C18`, `C19`, `C20`, `C21`) are required to drive its internal regulators, as seen in Figure 6.8.

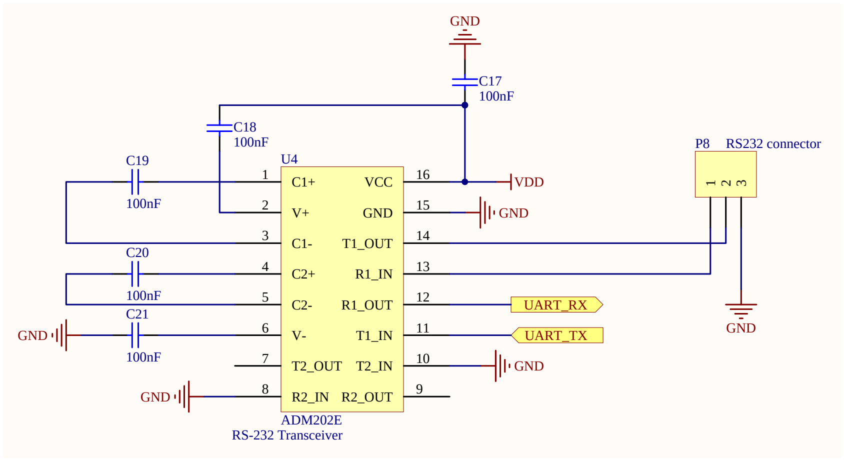


Figure 6.8: RS-232 transceiver

6.2.6 Memory Card Interface

As SD cards support a standard SPI interface, implemented in most microcontrollers, such a connector may be implemented with little need for extra circuitry. The pinout of a SD card is shown in Figure 6.10, while table 6.2 shows the appropriate connection to a SD-card for using SPI. As described in Section 6.2.3 this connection is, however, made more difficult by the fact that the SD card requires 3.3V, as opposed to 5V in the remaining circuit.

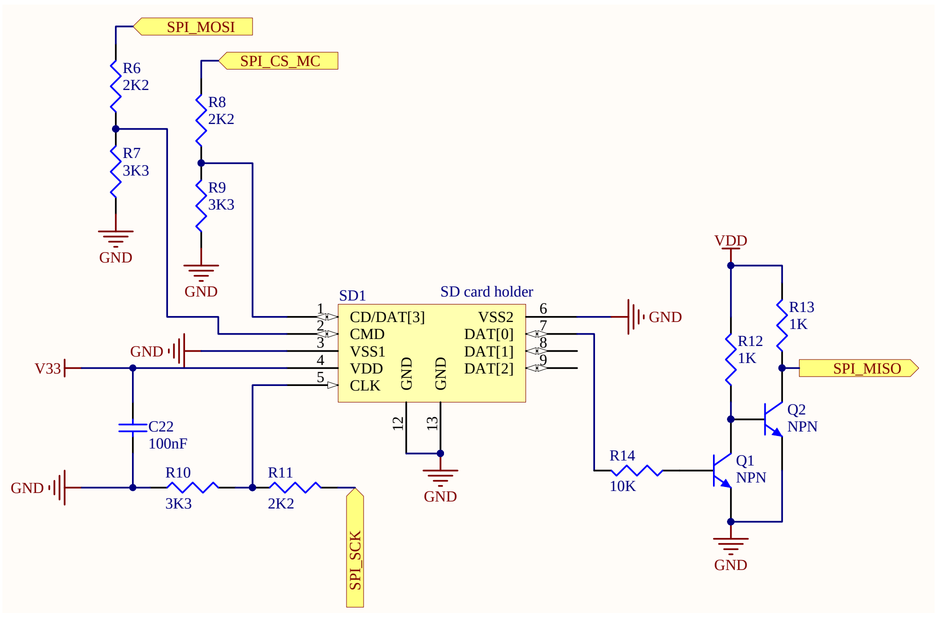


Figure 6.9: SD Card socket

For signals transmitted from the microcontroller to the SD card, a voltage conversion from 5 to 3.3 volts is required. As the current drawn at the SD card is negligible, a simple voltage divider has been used for this purpose. As can be seen from Figure 6.9, voltage dividers has been implemented for the signals `MOSI`, `MISO`, `SCK` and `/CS`. Values of $2.2k\Omega$ and $3.3k\Omega$ has been chosen for the resistors. This gives a resulting voltage and active current as shown below.

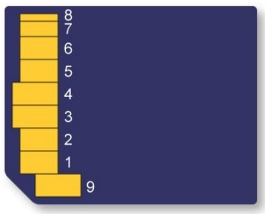


Figure 6.10: SD Card pinout

Pin nr.	SPI Function
1	CS
2	MOSI
3	GND
4	VCC (3.3V)
5	SCLK
6	GND
7	MISO
8	N/C
9	N/C

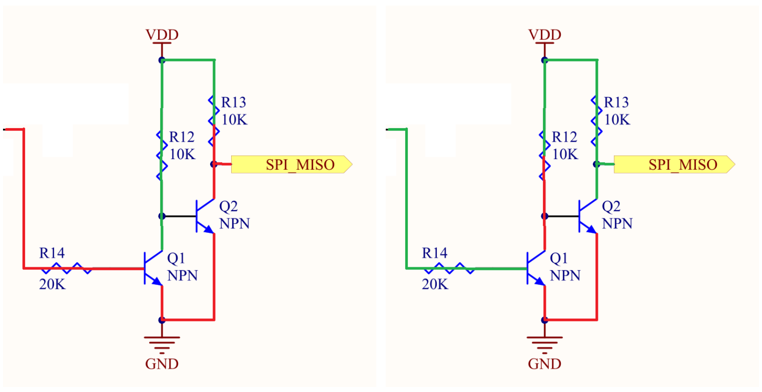
Table 6.2: SPI connections for SD Card

$$V_{div} = 5V \cdot \frac{3.3k\Omega}{2.2k\Omega + 3.3k\Omega} = 3V \tag{6.2}$$

$$I_{on} = \frac{5V}{2.2k\Omega + 3.3k\Omega} = 0.91mA \tag{6.3}$$

As we can see, the loss in the voltage divider is relatively high. There is, however, a trade-off within the voltage divider between minimizing power loss, and a quick slope of the output signal. For this particular circuit a fast response of the output signal was prioritized.

The datasheet of the Atmel UC3C[11] specifies a minimum voltage of $0.7 \cdot V_{cc}$ for a high signal. For a V_{cc} of $5V$ this gives a threshold voltage of $5V \cdot 0.7 = 3.5V$. This means that the microcontroller will not interpret the 3.3V signal from the SD-card as a high logical signal. A means to convert the voltage level of the MISO signal therefore had to be implemented. A detailed view of this circuit is shown in Figure 6.11.



(a) Low input (b) High Input

Figure 6.11: Signal level conversion circuit for SD-card (Red: Logic low Green: Logic high)

Figures 6.11a and 6.11b show the circuit for respectively low and high input. Two NPN transistors have been used in this circuit. The first stage generates an inverted 5V signal by using a pull-up resistor. This signal may be pulled low by activating transistor Q1. To achieve the same logical polarity for input and output, the signal will have to be inverted once more. The second stage is identical to the first stage. The transistor Q2 is trigged by the output of the first stage. Together these signals give a non-inverting signal level conversion. Similarly to the 5 to 3 V signal conversion, low values for the pull-up resistors has been chosen to achieve a fast response for the output signal.

6.3 PCB Design

6.3.1 PCB Design Objectives

The CCU may be mounted freely on the car, according to the wish of the developer. There are consequently no strict external requirements to the

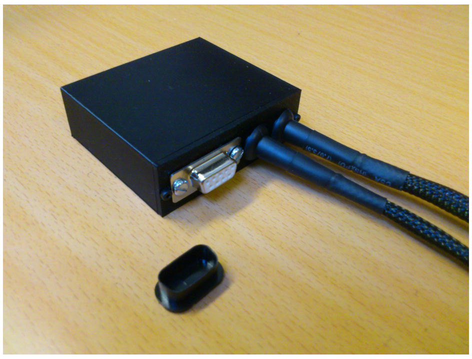


Figure 6.12: Assembled enclosure for Central Controller Unit

design of the CCU. On a racing car the minimization of weight and size is, however, always general targets for any accessories. It has therefore been in focus to keep the final circuit board simple and compact. To protect the module after installation, it must be mounted inside an enclosure. This must be taken into consideration when the PCB is designed.

6.3.2 PCB Design

The Central Controller Unit was the first PCB that was designed for this project. The PCB design was developed with two main revisions that will be shown below.

Initial Prototype Layout

The first prototype PCB for the CCU was designed mainly for the developer to achieve familiarity with PCB design, as well as to achieve some proficiency with the Altium PCB designer. Figure 6.13 compares the designs of the first and the final prototypes. As can be seen the initial prototype allowed a lot of space between components and tracks, for simple routing. This also made it easy to debug the design, and correct the inevitable errors to a first batch prototype.

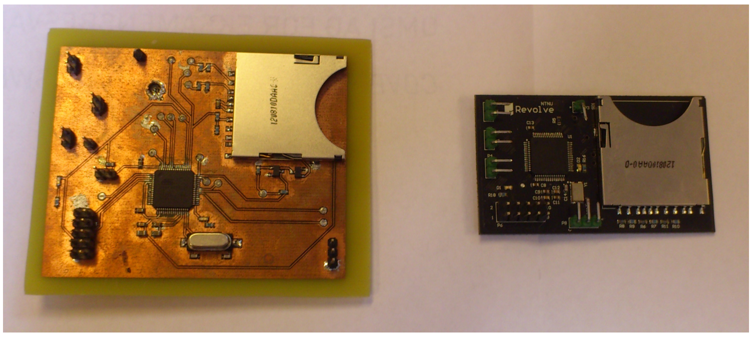


Figure 6.13: Comparison between initial and final prototype board

Final Prototype Layout

While the initial prototype was designed with ease of debugging in mind, it's design was deemed to inefficient to be acceptable for usage on the car. The unit was therefore completely redesigned for the next prototyping stages. The result was a board that occupied less than 30% of the area of the initial prototype.

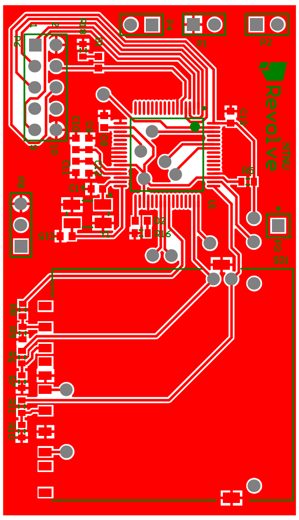
The PCB layers of this final prototype is shown in Figure 6.14. Full page illustrations are also available in Appendix B.1. A close-up illustration of the finished prototype is shown in Figure 8.7.

6.4 Testing of Central Controller Unit Hardware

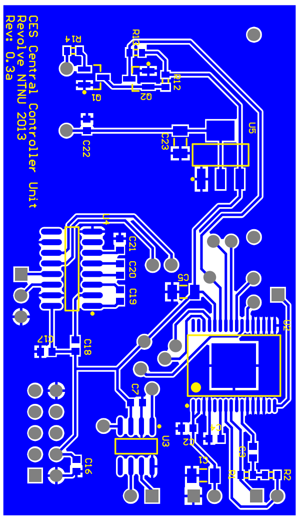
Before the prototypes were taken into use, some elementary hardware tests were performed to verify their functionality. This section will present the tests that were performed, and highlight any errors that were discovered.

6.4.1 Test of Power Supply

The initial test of the power supply performed on the first prototype of the CCU. The first time power was to be applied to the board, a digital power supply with current limiting was used. The current limited was adjusted



(a) Top layer



(b) Bottom layer

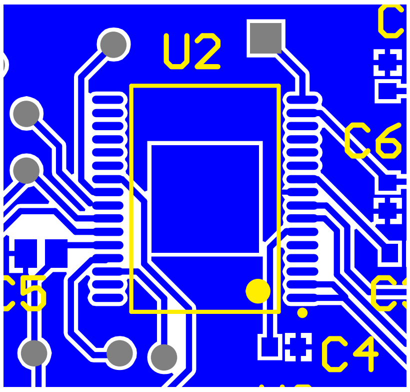
Figure 6.14: Layers of the Central Controller Unit PCB

to a maximum of 100mA, to reduce the risk of damage if there were to be any errors in the design. A voltage of 8V was applied to the input of the voltage regulator; about 1.5V above the under voltage threshold defined in the datasheet of the MC33903 [14, p. 19].

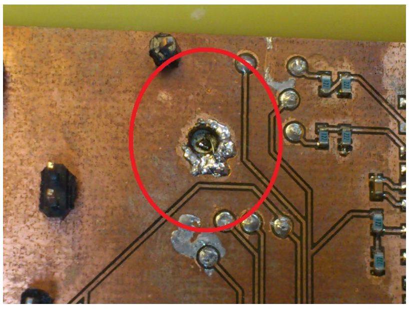
The voltage reading on the output of the voltage regulator was, however, approximately 5.8V. After studying the schematic, the error was identified to be a missing connection to the ground pad, on the underside of the MC33903. The error can be seen in Figure 6.15a. This pad is the only ground supply to the voltage regulator.

To work around this problem without having to produce a new PCB, a hole was drilled from the opposite side of the PCB. A thin wire was then soldered directly from the pad to the ground plane, as seen in Figure 6.15b. This allowed the voltage regulator to work correctly, but was obviously not a stable solution. The error was thus corrected for the following prototypes.

After the 5V regulator had been tested the 3.3V regulator was soldered to



(a) Missing connection to ground pad



(b) Fix of missing connection

Figure 6.15: Error in connection of PSU on initial prototype

the PCB, and tested in the same manner. No errors were found in this test.

6.4.2 Test of CAN Transceivers

Due to a lack of time, the CAN channels of the CCU was not connected until the final prototype PCB had been ordered. The CAN transceivers were tested by connecting a PEAK System PCANUSB USB-to-CAN interface to the transceivers and applying a test message. The USB-to-CAN adapter is shown in Figure 6.17. An oscilloscope was then connected to the `CANIF-RXLINE` pins of the microcontroller, to verify the propagation of the signal. At first attempt there was no signal measured at this pin. After studying the schematic of the CCU is became clear that the connections to the TX and RX pins of the microcontroller had been swapped for both CAN interfaces, as shown in Table 6.3.

To solve this error, the tracks between the microcontroller and the CAN transceivers were cut using a scalpel. Thin strap wire was then used to interchange the paths, as shown in Figure 6.16. A successful test was performed of both CAN transceivers after these corrections. It should be noted that the CAN transceiver of the MC33903 is disabled as default. It

Pin	Correct Function	Anticipated Connection
PB00	`CANIF-RXLINE[1]`	`CANIF-TXLINE[1]`
PB01	`CANIF-TXLINE[1]`	`CANIF-RXLINE[1]`
PD27	`CANIF-RXLINE[0]`	`CANIF-TXLINE[0]`
PD28	`CANIF-TXLINE[0]`	`CANIF-RXLINE[0]`

Table 6.3: CAN controller connections of the Atmel UC3C2512

must be activated using SPI on power up, this transceiver could therefore not be tested until a driver for the SBC had been developed.

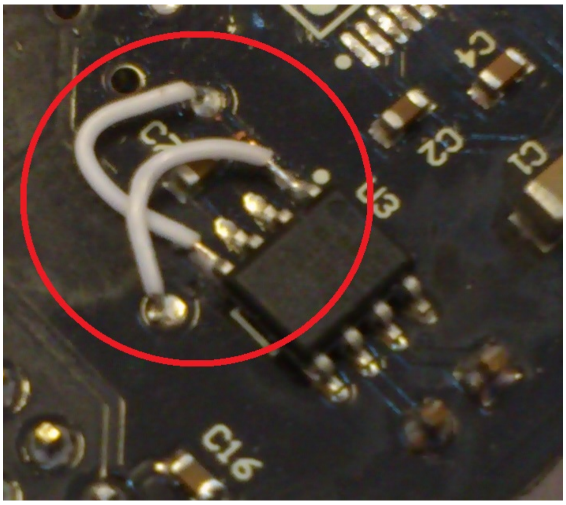


Figure 6.16: Fix of wrong connection to CAN transceivers

6.4.3 Test of RS-232 Transceiver

The RS-232 transceiver was tested initially in the same way as the CAN transceivers. The transceiver was connected to the RS-232 port of a computer, and a terminal emulator was used to generate some test data. An oscilloscope was used to verify the reception of data at `USART0-RXD` at pin 39 of the microcontroller. No errors were discovered during this test.

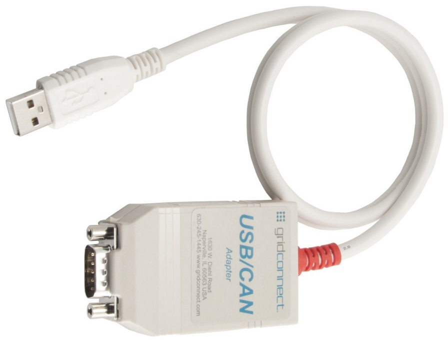


Figure 6.17: PEAK System CAN-to-USB interface[15]

6.4.4 Untested Hardware

As a consequence of limited time, the functionality of the memory card interface and corresponding level conversion circuit has not been tested.

7 Wheel Controller Unit

"There is a computer disease that anybody who works with computers knows about. It's a very serious disease and it interferes completely with the work. The trouble with computers is that you 'play' with them!"

- Richard Feynman

This chapter will present the hardware implementation of the Wheel Controller Units. Initially, the chapter will start by identifying the requirements that apply to this module. After these requirements has been identified, the chapter will present the hardware design that has been prepared for this unit. PCB design and implementation is also part of this chapter, and are presented in the final sections.

7.1 Requirements

7.1.1 Power Supply

Requirements to the Power Supply of the Wheel Controllers are much the same as for the Central Controller Unit. A 5V power supply to drive the electronics should be supplied. For further details, the reader should refer to the requirements identified for the power supply of the Central Controller Unit in Section 6.1.1. The DC motors will be supplied with current directly from the wiring harness. PWM control of the DC motors may generate high frequency noise on the power supply. It is important that this does create disturbances for the power supply to the unit's electronics.

7.1.2 Microcontroller

The primary role of the WCUs microcontroller is to execute two PID controllers that will control the position of the damper's valves. Two PID loops must be implemented for each damper. The computational performance of the microcontroller must sufficiently high to execute these control loops at a frequency exceeding 500 Hz, as specified by the requirements top level requirements in Section 3.1.1.

The microcontroller must also be able to interface the rotary encoders through SPI, and have the ability to generate 4 PWM control signals to control the two H-bridges for the motors. As the available board space for the WCU is very limited, a microcontroller with as little board area as possible should be chosen.

7.1.3 Communication

The WCUs must respond to control signals from the Central Control Unit, transmitted over the interconnecting CAN bus. A CAN channel must therefore be implemented at each WCU according to the specifications defined in Chapter 5.

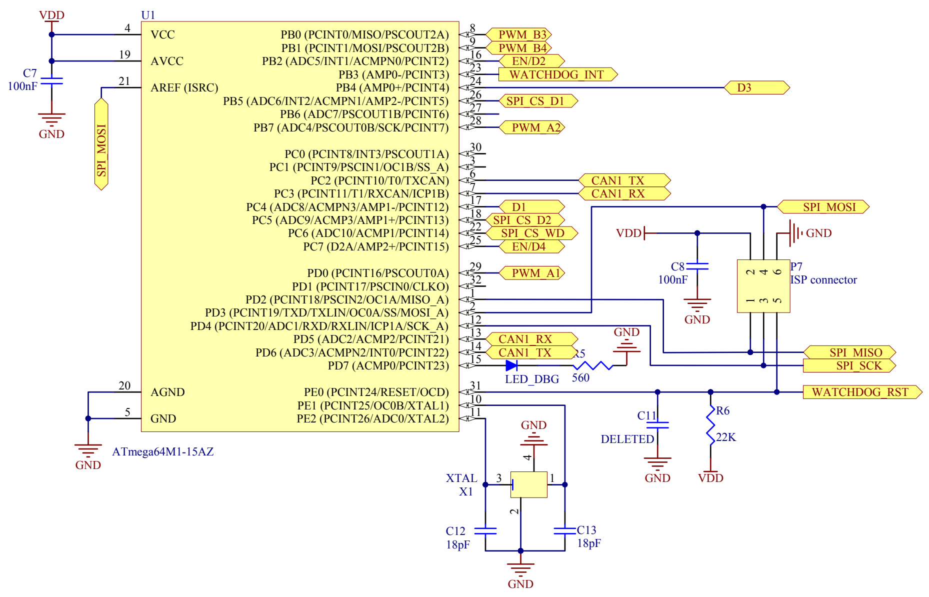
An interface to perform hardware debugging for the unit should also be implemented.

7.1.4 Motor Control

Each Wheel Controller Unit shall control the two valves of a damper. The actuation of the valves are done by two DC motors. Circuitry to control these DC motors must therefore be implemented. The control circuitry must be implemented according to the motor specifications given in the top level requirements, Table 3.1.

7.2 Implementation

7.2.1 Microcontroller



VDD
U1
C7
100nF
GND
SPI_MOSI
VCC
AVCC
AREF (ISRC)
AGND
GND
ATmega64M1-15AZ
PB0 (PCINT0/MISO/PSCOUT2A)
PB1 (PCINT1/MOSI/PSCOUT2B)
PB2 (ADC5/INT1/ACMPN0/PCINT2)
PB3 (AMP0-/PCINT3)
PB4 (AMP0+/PCINT4)
PB5 (ADC6/INT2/ACMPN1/AMP2-/PCINT5)
PB6 (ADC7/PSCOUT1B/PCINT6)
PB7 (ADC4/PSCOUT0B/SCK/PCINT7)
PC0 (PCINT8/INT3/PSCOUT1A)
PC1 (PCINT9/PSCIN1/OC1B/SS_A)
PC2 (PCINT10/T0/TXCAN)
PC3 (PCINT11/T1/RXCAN/ICP1B)
PC4 (ADC8/ACMPN3/AMP1-/PCINT12)
PC5 (ADC9/ACMP3/AMP1+/PCINT13)
PC6 (ADC10/ACMP1/PCINT14)
PC7 (D2A/AMP2+/PCINT15)
PD0 (PCINT16/PSCOUT0A)
PD1 (PCINT17/PSCIN0/CLKO)
PD2 (PCINT18/PSCIN2/OC1A/MISO_A)
PD3 (PCINT19/TXD/TXLIN/OC0A/SS/MOSI_A)
PD4 (PCINT20/ADC1/RXD/RXLIN/ICP1A/SCK_A)
PD5 (ADC2/ACMP2/PCINT21)
PD6 (ADC3/ACMPN2/INT0/PCINT22)
PD7 (ACMP0/PCINT23)
PE0 (PCINT24/RESET/OCD)
PE1 (PCINT25/OC0B/XTAL1)
PE2 (PCINT26/ADC0/XTAL2)
PWM_B3
PWM_B4
EN/D2
WATCHDOG_INT
D3
SPI_CS_D1
PWM_A2
CAN1_TX
CAN1_RX
D1
SPI_CS_D2
SPI_CS_WD
EN/D4
PWM_A1
CAN1_RX
CAN1_TX
LED_DBG
560
SPI_MOSI
GND
VDD
C8
100nF
P7
ISP connector
SPI_MISO
SPI_SCK
WATCHDOG_RST
C11
DELETED
R6
22K
XTAL
X1
C12
18pF
C13
18pF

Figure 7.1: ATmega64M1 Microcontroller

MCU

The Wheel Controller Units are implemented with a different purpose than the Central Controller Unit. The specifications of the Atmel UC3C chosen for the CCU are therefore not strictly ideal for the WCUs. As described in Section 6.2.1, the UC3C was chosen primarily because of its strength with respect to computing power. The Wheel Controller Units will, on the

Unit	Connection
Angular Sensors	SPI
System basis chip	SPI + $\overline{\text{INT}}$ + $\overline{\text{RST}}$
CAN1 Transceiver	CAN RX/TX
Dual H-bridge Control Signal	PWM
Dual H-bridge Enable/Disable Signal	Digital

Table 7.1: Peripheral units connected to WCU microcontroller

other hand, implement PID control algorithms for position control of the damper valves. PID controllers may be implemented easily in software, and a standard 8-bit microcontroller is therefore sufficient for this application.

The prestudy recommended to use an Atmel ATmega64M1 for this purpose. The primary strengths of this microcontroller is its small 32-pin package, and an integrated CAN controller. Which will help reduce the limited board size. The ATmega64M1 akso features a 6 output *power stage controller* and a 16-bit counter, both suitable for generation of PWM control signals.

As described in Section 6.2.1 the WCU will also, as when possible use components with QFP packages to simplify the assembly process for PCB prototyping. The ATmega64M1 is available in a standard QFP package.

Table 7.1 shows the peripherals connected to the microcontroller. Compared to the hardware design of the CCU, there are several units requiring common digital I/O signals. Combined with the lesser number of pins available, this has resulted in very few unused pins.

Oscillator

The datasheet of the ATmega64M1[16, p. 295] specifies a maximum clock frequency of 16MHz for V_{cc} higher than 4.5V. The microcontroller features an internal RC oscillator of 8MHz. To achieve a clock frequency of 16MHz, an internal PLL may be used. The internal RC oscillator is however of limited accuracy. As the Wheel Controller Units shall communicate with peripherals over CAN and run a PID controller, a stable system clock is highly important. As for the Central Controller Unit, an external oscillator is therefore required. Opposed to the UC3C does not allow external oscillators to be used as a source for the PLL, a oscillator with the exact desired clock frequency needs to be used.

In the interest of simplicity it was decided to use the same 16 MHz crystal oscillator as for the CCU. The connection of the oscillator is designated as `X1` on Figure 7.1.

Programming Port

Because of the limited number of pins available, the ATmega64M1 has not implemented a JTAG debugging interface. The controller does, however, implement a standard ISP programming interface, and Atmels debugWire interface for low-level debugging. The ISP interface requires a 6-wire header to be available on the microcontroller, while debugWire requires only a single wire to be connected to the reset pin of the microcontroller in addition to V_{cc} and ground. Because of the limited PCB space available, it would have been beneficial if the ISP interface could be omitted. The megaAVR architecture does, however, require a ISP interface for programming of the microcontrollers fuses. As the unit is in a prototype stadium, it was decided to include a 6-pin header for connection of an ISP programmer. The programming port is denoted `P7` in Figure 7.1. As both the reset, V_{cc} and ground pins are available in the ISP header, this header may also be used for debugging using debugWire.

As can be seen from Figure 7.1 a LED, designated `LED_DBG`, has also been implemented on the PCB for elementary debugging.

SPI Interface

The WCU uses three components that will be interfaced by SPI from the ATmega64M1; the angular sensors and the system basis chip. As can be seen from Figure 7.1, the number of free I/O pins on the microcontroller was very limited. The ATmega64M1 may use two different pinouts for its SPI interface. The two alternative pinouts are shown in Table 7.2. If the table is compared with the microcontrollers schematic, it can be seen that

Function	Pin	
	Regular Pinout	Alternative Pinout
MOSI	PB1	PD3
MISO	PB0	PD2
SCK	PB7	PD4
/SS	PD3	PC1

Table 7.2: Pinout alternatives for SPI on ATmega64M1

the alternative SPI pinout is identical to the ISP programming interface. To not occupy any unnecessary pins it was therefore chosen to use this alternative pinout for the SPI.

7.2.2 System Basis Chip

In the interest of simplicity it was decided to reuse the same Freescale MC33903 System Basis Chip for the Wheel Controller Units. For more detailed information about this component, please refer to Section 6.2.2. As can be seen from Figure 7.2 the connection schematic of the SBC is highly similar to that implemented for the CCU.

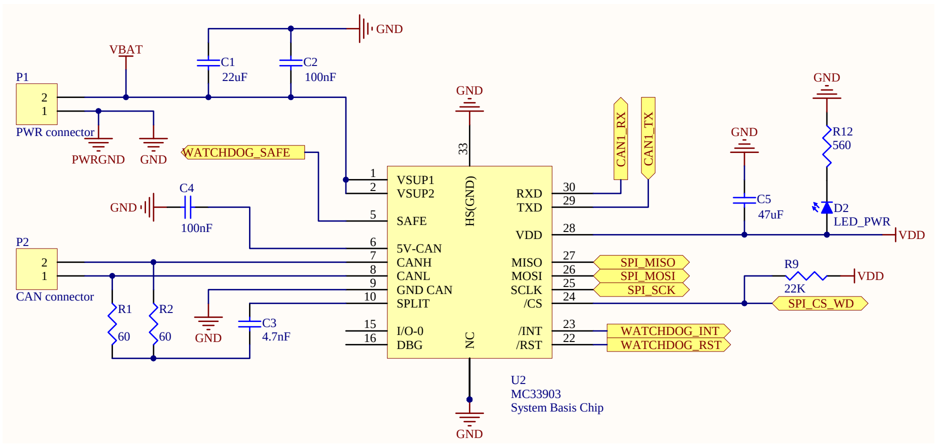


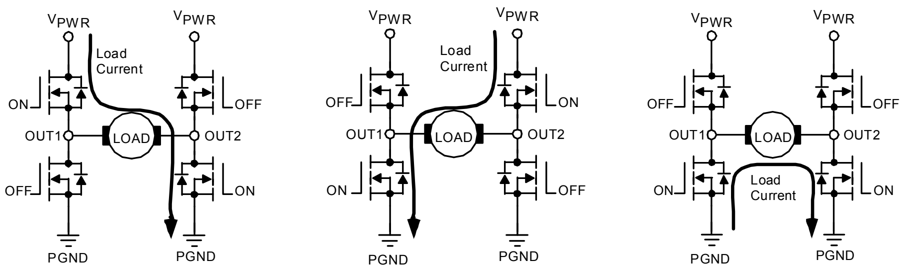
Figure 7.2: System Basis Chip with 5V power supply

As described in Section 7.2.1, the SPI interface of the SBC had to be connected to the alternative set of pins on the ATmega64M1. This connection is shown in Figure 7.1. Because of limited board space, it was decided not to implement a header for the `DBG` pin of the SBC. When needed, during early development stages, a strap wire was soldered between this pin and `VSUP`.

To easily be able to monitor the state of the power supply, a LED designated `LED_PWR`, has been connected to the regulated output.

7.2.3 Motor Control

A MC33932 dual H-bridge has been implemented in the hardware implementation for controlling the DC motors. The basic principle of a H-bridge is illustrated in Figure 7.3.



(a) Forwards (b) Reverse (c) Braking

Figure 7.3: Conceptual illustration of a H-Bridge [17]

By activating a pair of transistors, i.e. as in Figures 7.3a and 7.3b, the motor may be controlled bi-directionally. The motor may be braked by short-circuiting both its terminals to ground, as seen in Figure 7.3c. Furthermore, the torque of the motor may also be controlled, by applying a PWM signal to the transistors.

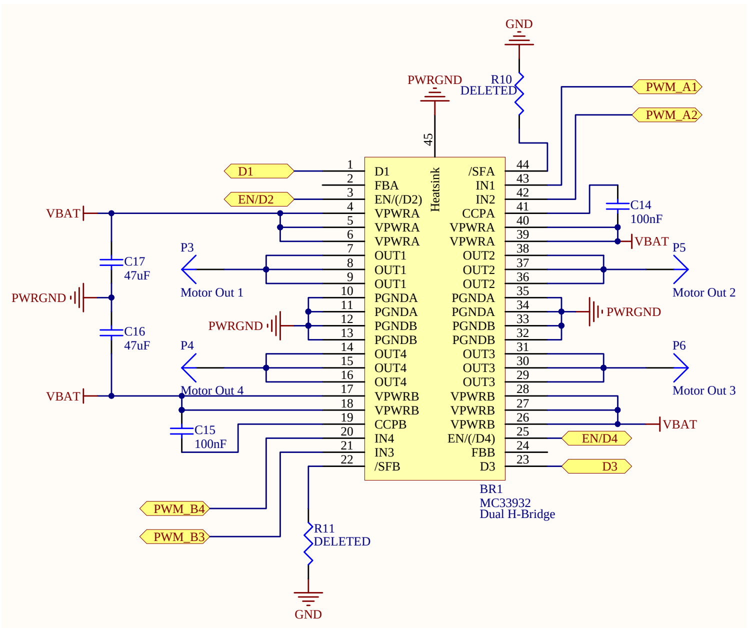


Figure 7.4: Dual H-Bridge motor controller

Dual H-bridge

The schematic of the MC33932 is shown in Figure 7.4. A logic state table extracted from the datasheet of the MC33932 has been presented in Table 7.3. The corresponding logic table for outputs `OUT3` and `OUT4` is identical. As can be seen from the table, four inputs control the operation of each H-bridge. `IN1` and `IN2` activate respectively reverse or forwards current, while EN/$\overline{\text{D2}}$ and D1 may be used to completely disable the operation of the H-bridge. As can be seen from Figure 7.1 these inputs have been connected to regular digital outputs of the microcontroller.

The datasheet of the H-bridge[17] specifies a maximal continuous output current of 5A. This is above the required rating, given in the motor specifications in Table 3.1.

Device State	Input Conditions				Outputs	
	EN/$\overline{\text{D2}}$	D1	IN1	IN2	OUT1	OUT2
Forward	H	L	H	L	H	L
Reverse	H	L	L	H	L	H
Braking	H	L	L	L	L	L
Disable 1 (D1)	H	H	X	X	Z	Z
Sleep mode	L	X	X	X	Z	Z

Table 7.3: Table of logic states for the MC33932 (H: High, L: Low, Z: Floating, X: Don't-care)

Power Stage Controller

The ATmega64M1 has implemented a hardware module called a *Power Stage Controller*(hereafter abbreviated PSC). The PSC is in many ways similar to a regular counter with PWM output, however, it also supports synchronization of the outputs for control of e.g. brushless motors.

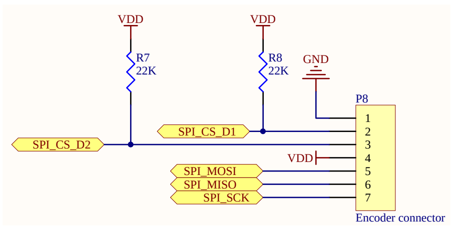
To allow the internal 16-bit timer to be used for other purposes, it was decided to use the PSC for control of the DC motors. As we are controlling a brushed DC motor, only the most basic functionality of the PSC has been used. The MC33932 has four PWM input ports, `IN1` through `IN4`. The PWM inputs has been connected to the outputs `PSCOUT0A`, `PSCOUT0B`, `PSCOUT2A` and `PSCOUT2B` of the microcontroller, as seen in Figure 7.1. Further operation of the PSC will be described in Section 13.3.2.

7.2.4 Wheel Encoder Board

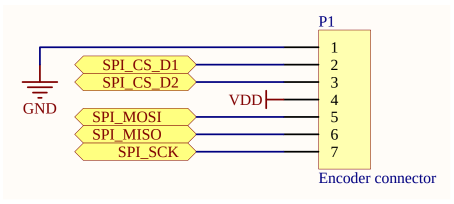
As can be seen in Figure 7.10, the electronics of the Wheel Controller Unit has been divided onto two PCBs. The main board, as seen on top in the picture, includes all the electronics described so far. As position control of the DC motors has been implemented, angular sensor were needed. For this application a through-hole absolute rotational encoder, with PCB mounting has been used. To eliminate the need for gears to connect the sensor, a separate PCB has been designed to be mounted directly under the motors.

Connection Header

To connect the Wheel Encoder Board with the main board of the WCU, traditional header connectors with 2.54mm spacing have been used. This connector will also act as support for the main board, when mounted inside the enclosure. The pinout for each side of the connector is shown in Figure 7.5. The encoders are interfaced by the microcontroller using a SPI interface. In addition to SPI, the header will also supply the Wheel Encoder Board with $+5V$ power, ground, and chip select signals for each of the encoders.



(a) Wheel Controller Board header



(b) Wheel Encoder Board header

Figure 7.5: Wheel Encoder Board connection header

Angular Sensor

As recommended by the prestudy, a Piher MTS-360 has been used to provide positional feedback for the motor controller. This encoder was chosen on the basis of its small size, so that it could be mounted directly under the shaft of each motor.

The Piher MTS-360 encoders are available with wither analogue or SPI interface. To eliminate any inaccuracy from the use of an ADC, and simplify the interface from the microcontroller, it was chosen to use the SPI version of the encoder. This encoder had to be ordered directly from the manufacturer, as it was not stocked by any online electronics suppliers. The available documentation from the manufacturers website was very limited and, at the time the order was placed, outdated. Table 7.4 shows

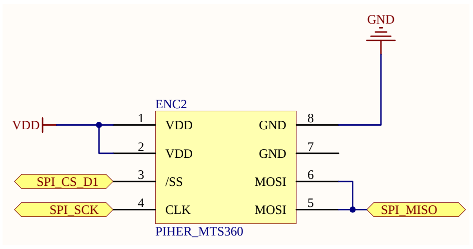


Figure 7.6: Angular Sensor

Pin	**Function**	
	Old	New
1	+5V supply	+5V supply
2		
3	/SS	/SS
4	CLK	CLK
5	MOSI	MOSI
6	MISO	
7	GND	GND
8		

Table 7.4: Comparison of old and new pinout for the MTS-360

the pinout available at the time of ordering, compared to the actual pinout provided in the documentation that followed the sensors at delivery.

As can be seen from the table, the traditional MISO/MOSI SPI interface had been replaced with a single-wire SPI interface with a combined MISO/MOSI line. This line had to be connected as indicated in Figure 7.7. As can be seen, a N-channel FET transistor, connected to the MOSI output of the MCU, is used to pull the combined MOSI/MISO line to ground. The line must also be connected to V_{cc} by a pull-up resistor of $1k\Omega$.

As the ATmega64M1 only has one SPI channel, that will be shared with the SBC, this is likely to cause compatibility issues. While SPI is normally a full duplex bus, the single-wire connection may only be used as half duplex. A way to enable and disable the pull-down transistor thus had to be implemented. This logic mechanism is described in the following section.

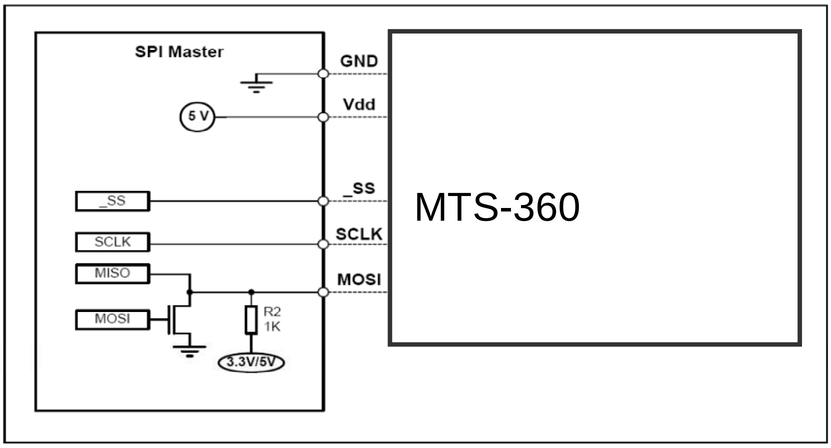


Figure 7.7: Connection diagram for angular sensor[18]

Logic Circuit

As explained in the above section, the MTS-360 requires a single-wire SPI bus to be implemented. The regular transfer principle for a SPI bus is shown in Figure 7.9. Data is shifted simultaneously from the Master to the Slave, and vice versa. This will cause compatibility problems with the communication with the System Basis Chip when the Wheel Encoder Board is connected, as the transistor will pull the MISO line of the two-wire SPI bus to ground whenever MOSI is high – thereby distorting this signal. To solve this problem, the circuit shown in Figure 7.8 was implemented.

`Q1` is a dual small signal N-channel MOSFET. The two internal MOSFETS were connected in series, so that both transistors has to be active for the MOSI/MISO line to be pulled to ground. The first of the transistors were connected directly to MOSI as seen in Figure 7.7. As the pull down mechanism only has to work during communication between the MCU and the encoders, the second transistor may be enabled using the chip select signal for the encoders. As the chip select signals are *active low*, a NAND gate was used to generate the gate signal `G2` for the second transistor.

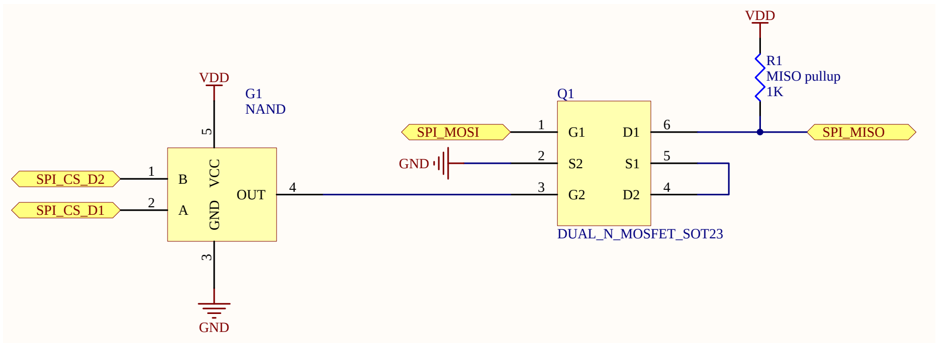


Figure 7.8: Logic circuit for connection of angular sensor

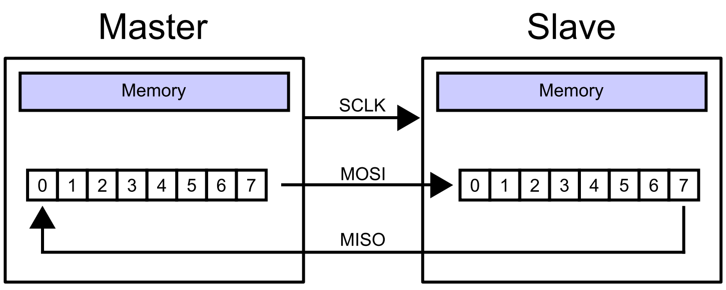


Figure 7.9: Circular data transfer on SPI[19]

7.3 PCB Design

This section will present the hardware design of the Wheel Controller Unit. Two PCBs has been designed; The main Wheel Controller Board and the Wheel Encoder Board. The first part of this section will present some of the requirements that were used to design the PCBs. A description of the PCBs, as well as some special considerations that had to be taken during the design process will be given.

7.3.1 PCB Design Objectives

The main requirements of the Wheel Controller Units, with respect to PCB design, has been in terms of size. As described in the top level requirements, is was recommended that the main Wheel Controller Board of the WCUs should be placed inside the actuator enclosure mounted on the damper. This enclosure protects the electronics from dust and water. Because the lid of the enclosure is made of carbon fibre, the electronics are also protected from external electromagnetic radiation. The suspension placement on the car did, however, heavily limit the outer dimensions of this enclosure. A space of $45 \times 26 \times 20mm$ was made available for the main PCB of the WCU.

As can be seen from the figure the main PCB is mounted directly onto the Wheel Encoder Board PCB using a 90° angled header connector.

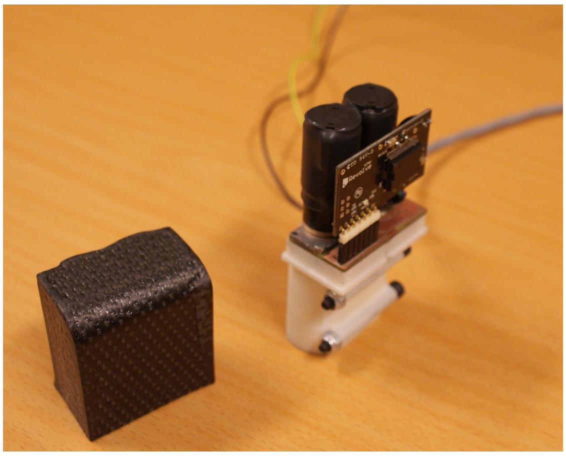


Figure 7.10: Assembled Wheel Controller Unit

7.3.2 PCB Design

Wheel Controller Board

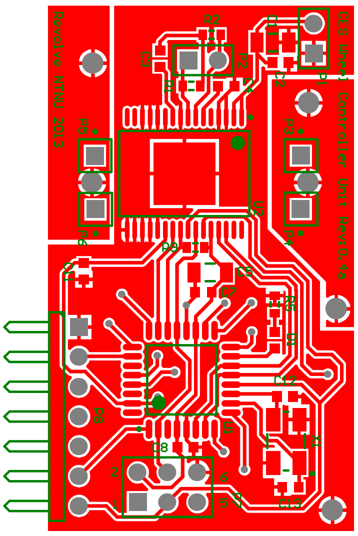
The Wheel Controller Board is the main PCB of the Wheel Controller Units. As described in Section 7.3.1 a space of $45 \times 26mm$ is the maximum permissible area for this printed circuit board. The Wheel Controller Board will supply the DC motors with current. According to Table 3.1 each motor has a stall current of 1.45 amperes. Special considerations therefore has to be taken when the PCB layout is designed to avoid noise and damages to other electronics in the circuits on the same board

As far as possible, the design has tried to separate the high current electronics to a separate layer of the PCB. Because of the small size it was, however, not possible to strictly enforce this rule. Where signal and high current components have had to be placed on the same layer, separate ground planes has been created to avoid undesirable ground currents and noise. Further information about mixed current PCB design may be found in [20]. The top layer has primarily been assigned to the small signal electronics, while the bottom layer has been used for the H-bridge IC. This can be seen in Figure 7.11, as well as in Appendix B.2, where full page illustrations of the circuit design has been included.

Wheel Encoder Board

The Wheel Encoder Board has been designed to fit directly under the mounting bracket of the DC motors. As described in Section 7.2.4, this PCB will be used to mount the angular sensors and their accompanying logic circuitry. As the board will be mounted under the motor mounting bracket, there are fixed size constraints that apply to the PCB. These measurements are shown in Figure 7.12. As can be seen from the Figure, one side of the PCB is partially covered by the metal mounting bracket. This area may therefore not be used for PCB tracks, and should be isolated from the remaining circuit.

As can be seen from Figure 7.13, two holes has been drilled for the motor shafts. Additionally, there are also two holes for the head of the bolt that


(a) Top layer

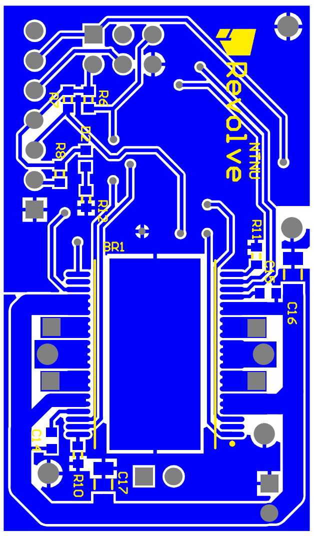

(b) Bottom layer

Figure 7.11: Layers of the Wheel Controller Unit PCB

fix the motor to its mounting bracket. To allow fitment inside the enclosure, all motors must be mounted with the same angle to allow clearance between the motor's connection terminals and the carbon fibre lid. The placement of the mounting holes relative to the terminals is, however, arbitrary. The position of these holes therefore had to be entered manually for each of the Wheel Encoder Boards.

7.4 Testing of Wheel Controller Unit Hardware

Before the wheel controller prototypes were taken into use, some elementary hardware tests were performed to verify their functionality. This section will present the tests that were performed, and highlight any errors that were discovered.

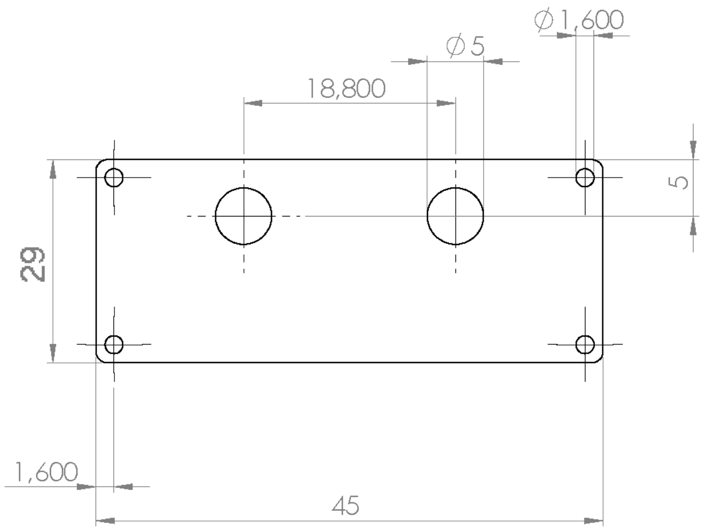


Figure 7.12: Outline of Wheel Encoder Board

7.4.1 Test of Power Supply

The power supply was tested using the same procedure as for the Central Controller Unit. Please refer to Section 6.4.1, for further details. No errors were found during the testing of the power supply for the wheel controller units

7.4.2 Test of CAN Transceiver

The CAN transceiver was tested using the same test procedure as for the CCU. Further details may be found in 6.4.2. Propagation of signals through the transceiver was verified without any issues for the Wheel Controller Units.

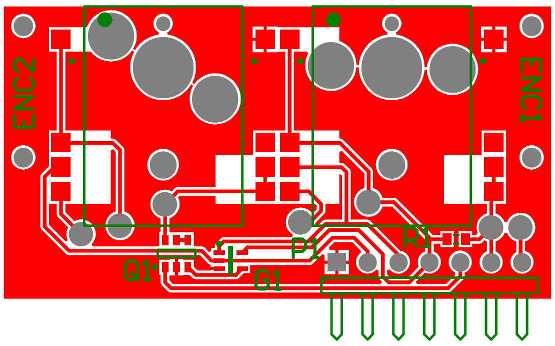


Figure 7.13: Top layer of Wheel Encoder Board

7.4.3 Test of H-bridges

The H-bridges could not be tested sufficiently without an operating microcontroller. A simple test program therefore had to be implemented for the WCU. This program manually applied the necessary signals to drive enable the outputs for forwards and reverse, according to Table 7.3. An 8V input was applied to the circuit.

At the first test no output signal was detected at outputs `OUT1` and `OUT2`, while a correct output signal of successively $+8V$ and $-8V$ was measured between outputs `OUT3` and `OUT4`. Measurements on the H-bridge inputs `IN1` and `IN2` indicated that the enable signals from the microcontroller did not propagate to these pins. A short circuit was also detected between these inputs and ground. After studying the schematic, it was found that these lines lie very close to the ground pad of the H-bridge. This is indicated in Figure 7.14. The cause of the failure was suspected to be that too much solder paste had been applied when the circuit was assembled. Testing of this H-bridge was therefore halted, until a new prototype had been assembled. The test of the second prototype showed no errors.

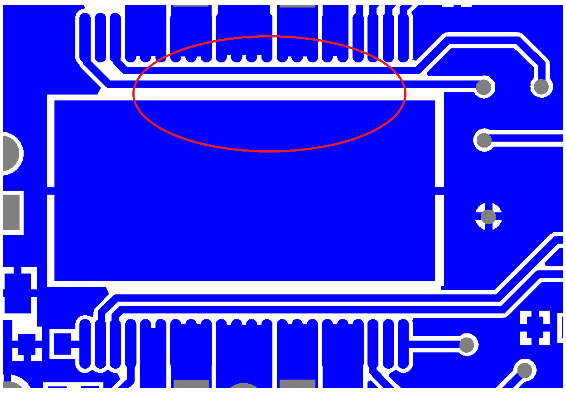


Figure 7.14: Probable cause of malfunctioning H-bridge channel

8 Production, Assembly and Installation

"Of course I do not look busy. I did it right the first time!"

- Anonymous

8.1 Production and Assembly of Units

The first prototypes of the units for the CES system has to be produced and assembled locally at the university. An LPKF ProtoMat circuit board plotter, as shown in Figure 8.1, was used for production of the PCBs. The assembly of the units then had to be done manually, using a traditional soldering iron.

When the final prototypes were to be produced a more professional result was, however, desired. Through Revolve, a deal was arranged with electronics producer SimPro, at Løkken Verk to manufacture and assemble the CES units. This section will give a brief introduction to how the units were assembled.

8.1.1 The PCB Panel

The production of the CES units were done in cooperation with several other units for the electronics group of Revolve. Multiple PCBs were collected to create a so called *panel*, as shown in Figure 8.2. All units on

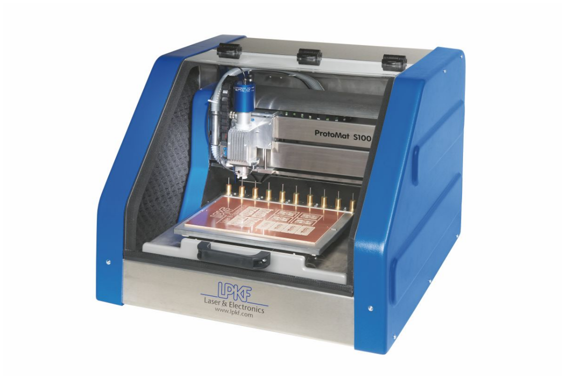


Figure 8.1: LPKF ProtoMat circuit board plotter

the panel were assembled at once, before the panel was split in to individual PCBs at the end of the assembly process. Notice that the PCB has been painted wherever there are no pads for the components. This will prevents short circuits and corrosion, that unpainted PCBs are prone to.

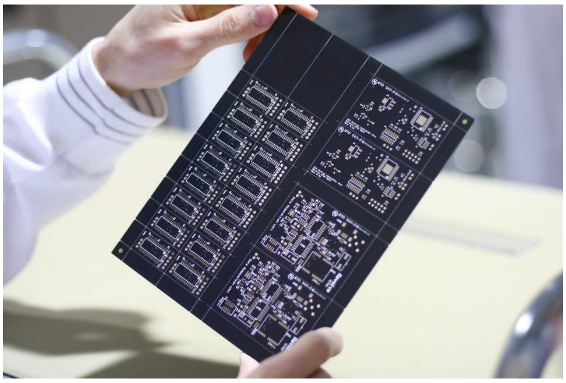


Figure 8.2: A professionally manufactured PCB panel

8.1.2 Application of Solder Paste

The huge majority of components used on the PCBs were surface mounted. Only these components were assembled during this process. Before components could be placed on the PCB, solder paste had to be applied. The solder paste also acts as a "glue" that hold the components to the PCB.

To only apply the solder paste where there are pads for components, a *stencil* is used, as shown in Figure 8.3. A stencil is a metal sheet, where holes have been etched at the exact places of the pads of the PCB.

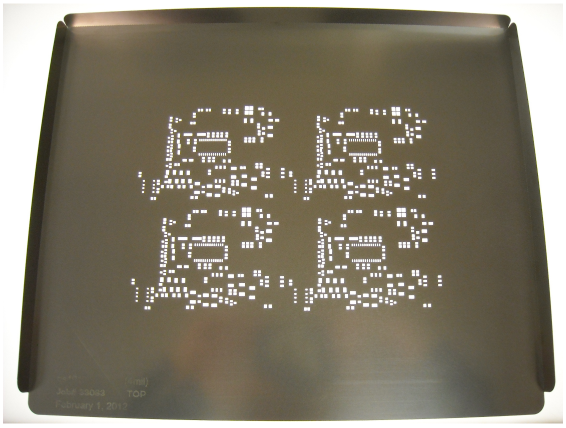


Figure 8.3: Stencil for application of solder paste[21]

The stencil was aligned with the PCB panel using automated machinery, before the solder paste was applied with high pressure by an automated roller, as seen in Figure 8.4.

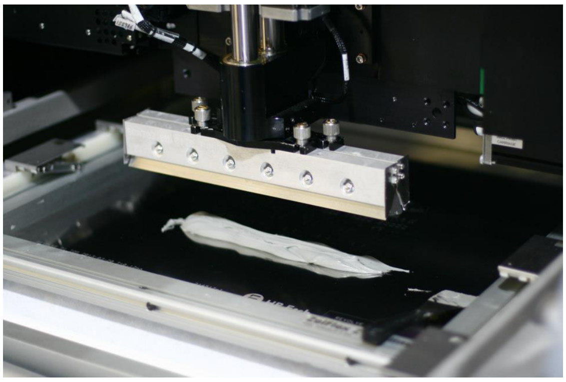


Figure 8.4: Solder paste was applied automatically to PCB panel

8.1.3 Component Placement

For larger batches of PCBs it is common to use automated, so called pick-and-place machinery, to place components on the panel. For smaller batches, setup of these machines are, however, too costly. Components were therefore placed with conventional tweezers, as shown in Figure 8.5.

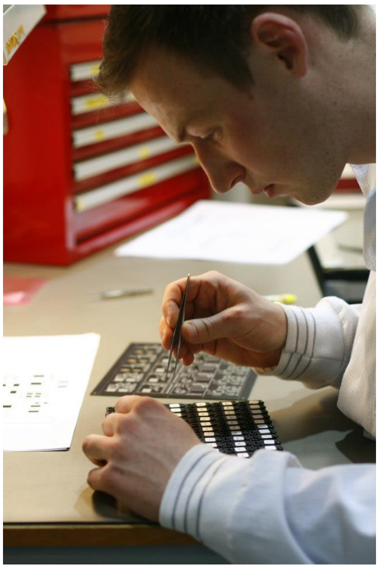


Figure 8.5: Manual placement of components on circuit board

8.1.4 Baking

Correct temperature is critical when components are soldered to the PCB. A too high temperature will damage the components, while a too low temperature will not allow the solder paste to melt properly. This will produce solder joints with bad contact. The panels were therefore baked in a specialized oven. Figure 8.6 shows the baking oven used at SimPro. The length of the oven is approximately 5 meters. Different zones inside the oven apply different temperatures. A conveyor belt transported the panels through the oven, allowing gradual heating and cooling.

Figure 8.6: The PCBs were soldered in an automated soldering machine

After the soldering process the assembly was complete. The panel was broken into several PCBs. Figure 8.7 shows a finished PCB for the Central Controller Unit.

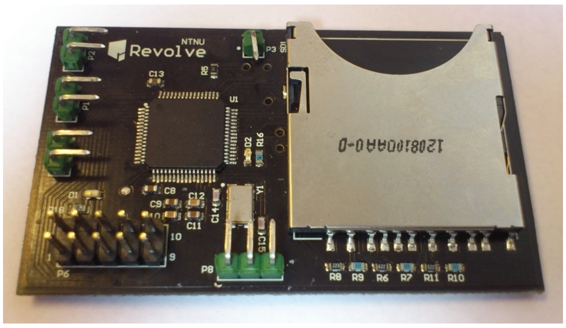


Figure 8.7: Finished Central Controller Unit PCB

8.2 Installation in Car

The system has been installed in the car, for further performance testing. The electronics groups has provided the necessary wiring harness in the car, to connect the various modules. The wiring harness will both provide CAN bus and 12V power supply from the battery.

Pin	Connection
1	Ground
2	12V
3	CAN High
4	CAN Low

Table 8.1: Pinout for wiring harness connectors

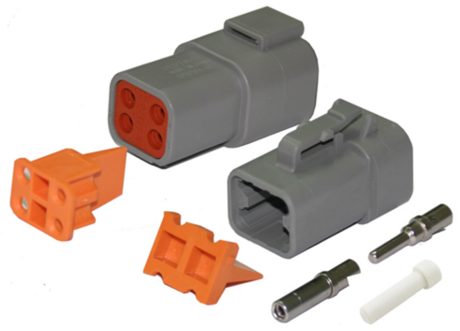


Figure 8.8: Connectors used for connection to the harness

To connect the units to the wiring harness, a suitable connector was needed. The connectors chosen were of the DTM series by Deutsch Connectors. A picture of the connector is shown in Figure 8.8. A 4-pin version of this connector was chosen for all units that would be connected to the wiring harness. This would allow easy connection of all units. A common pinout for the connectors was defined, as in Table 8.1.

9 Discussion on the Hardware Design

"Never trust a computer you can't throw out a window."

- Steve Wozniak

9.1 Central Controller Unit

9.1.1 Evaluation of the Hardware Design

This part has presented the design of a Central Controller Unit, to be used in the CES system. Based on recommendation from the prestudy, various components were selected and a hardware design was implemented.

To a large degree it was possible to reuse the components that had been chosen in the prestudy. Due to availability interchangeable components were, nevertheless, used for come of the components. Most of the chosen components were able to be integrated easily in the circuit. In addition to the features recommended by the prestudy, it was chosen to implement a memory card connector in the circuit. While the difference in signalling levels did somewhat complicate this process, it has allowed a large amount of flexibility for future use.

The first revision of the PCB design was laid out on a large PCB giving good space for debugging, and any corrections needed, such as the missing connection to the system basis chip. This did, however, require a lot of extra effort when the system had to be redesigned for the final prototype. The effort for the redesign was, however, deemed to be worthwhile as the board size was reduced with about 70%. A much more professional layout was also achieved.

Thanks to invaluable help from SimPro, it was possible to implement a final prototype with professionally manufactured PCB and assembly of components. This made further development of software much easier, as the early prototypes had minor defects and unstable tendencies.

The various modules of the circuit were tested after the PCBs had been assembled. Inevitably, there were some errors that had to be corrected. Due to limited time, it was necessary prioritize development of a circuit for the Wheel Controller Units. All modules were therefore not tested before after the final prototypes had been assembled. The error in connection of the CAN transceivers were therefore not corrected on time. The memory card circuit has nor been tested, as it was not necessary to achieve the level of software functionality intended for this assignment.

9.1.2 Further Work

There are several improvements that could be done to the hardware design of the CCU. Below follows a categorized list.

System Basis Chip

The system basis chip provided an easy way to integrate a power supply, CAN transceiver and watchdog functionality into the circuit. As will be remarked in the software discussion, development with the SBC has, nevertheless, been rather cumbersome. While not tested, an internal watchdog module is also available in the UC3C that may eliminate the need for an external watchdog. Future revisions should consider replacing the SBC for a conventional linear regulator and CAN transceiver. A regulator with higher current rating should be considered. Future revisions should also include a diode to protect the circuit from polarity inversion on the power connector.

3.3V Power Supply

The 3.3V power supply has been connected in cascade with the main power supply of the circuit. This is strictly not necessary, and could be avoided in later revisions. This would reduce the load on the main power supply.

CAN Transceiver

For any future hardware revisions the error in connection of the CAN transceivers should be corrected.

SD Card Connector

This module has not been tested for this assignment. For further development this module should be tested, and the implemented concept for signal level conversion should be verified.

9.2 Wheel Controller Unit

9.2.1 Evaluation of the Hardware Design

This part has presented the design and implementation of a Wheel Controller Unit to be used in the CES system. As with the Central Controller Unit, the majority of parts recommended in the prestudy could be used for development of a hardware design. Only one hardware design was implemented for the Wheel Controller Units, however, this design was gradually improved throughout the various prototype revisions. The design of a PCB was made rather difficult because of the strict size constraints that applied to the WCU.

The final prototypes of the WCUs was also manufactured by SimPro. As the first prototype of the WCU was designed after the initial prototype of the CCU had been tested, some of the elementary errors and flaws had already been addressed.

All modules of the of the WCU PCBs has been tested. As opposed to the CCU, all modules were required to work for the unit to fulfil the requirements defined in Section 7.1.

9.2.2 Further Work

As with the CCU there are several changes that could be done to improve the hardware design of the CCU. A list is given below.

System Basis Chip

The system basis chip should be considered replaced with a conventional linear regulator and CAN transceiver if allowed by the available board space. See Section 9.1.2 for more details.

Motor Noise Filtering

Although it has not been registered as a problem it is common practice to connect a ceramic capacitor across the terminal of DC motors to provide noise filtering. This should be considered for future hardware revisions.

Programming Port

The current design uses a standard header with 2.54mm pitch for the ISP programming port. Unfortunately, this header will not fit inside the finished enclosure. It thus had to be removed from the final prototypes. Future revisions should seek to implement an alternative connector for the programmer.

Part III

Software

10 Software Introduction

"People who are really serious about software should make their own hardware."

- Alan Kay, Pioneer Computer Scientist

The development of software for the CES system has been a substantial part of the work with this assignment. After a prototype hardware design had been implemented, it was necessary to develop a baseline software implementation for testing and evaluation of the system.

The development of software may be divided into several phases, the first of which is the *specification* phase. This phase will identify the requirements that apply to the software, so that these may be addressed during the design phase of the project. As this masters thesis has used a prestudy[1] by the same author as a basis, specifications from this document has been used throughout this phase.

While the prestudy gave several suggestion for the hardware design of the system, there were fewer suggestions in terms of software design. As the CES system has been implemented using a distributed architecture, over several modules, it has been an additional challenge to identify clear interfaces between the modules, and to keep the design of the software homogeneous. It has also been important to achieve a clear interface towards the other modules of the vehicle, that have been implemented by other team members. A common CAN message standard will therefore be presented in Section 11.2.

Based on the software design, a software *implementation* has been developed for both the Central and Wheel Controller Units. The software may be divided in a hierarchical structure, as shown in Figure 10.1.

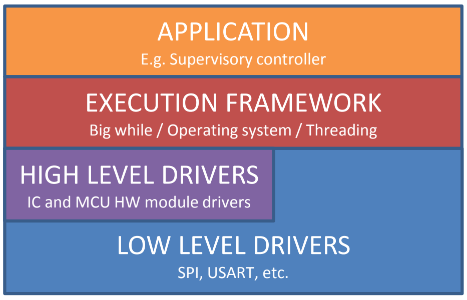

APPLICATION
E.g. Supervisory controller
EXECUTION FRAMEWORK
Big while / Operating system / Threading
HIGH LEVEL DRIVERS
IC and MCU HW module drivers
LOW LEVEL DRIVERS
SPI, USART, etc.

Figure 10.1: Hierarcical structure of embedded software

The first part of the development phase was much targeted towards the two bottom layers shown in the figure. Low level drivers had to be developed to utilize the hardware modules of the units. Higher level drivers also had to be implemented to interface external ICs, and internal modules such as the CAN controller. During software development it has been important to create an efficient framework for the system's main control algorithm, so that development may easily be continued at a later point of time.

The development of the supervisory control algorithm for the Central Controller Unit, is itself not part of this thesis. Collaborative work with other team members of Revolve has allowed implementation of a prototype control algorithm that may be used for testing. The target of the design has, however, been to allow execution of a more advanced algorithm. While the prestudy evaluated the possibility of using Mathworks Embedded Coder for implementation for controller development, this possibility has not been pursued further in this thesis.

All software development has been done in the language C. The Atmel Studio 6 IDE has been used throughout the project. Table 10.1 shows a list of the exact versions of the development tools used for this project. This should allow the reader to be able to accurately reproduce the findings presented in this thesis. Complete program code for this assignment is available in the digital attachments.

Component	**Version**
Atmel Studio	6.1.2562
AVR 8-bit Toolchain	3.4.2.939
8-bit GCC	4.7.2
AVR32 Toolchain	3.4.2.435
32-bit GCC	4.4.7
Atmel Studio Framework	3.3

Table 10.1: Software versions for development tools

11 Communication Protocol

"Design is not just what it looks like and feels like. Design is how it works!"

- Steve Jobs

11.1 Requirements for the Communication Protocol

The CES system has been distributed into five units: One *Central Controller Unit*, and four *Wheel Controller Units* – one for each wheel. The purpose of the Wheel Controller Units is clearly different from the purpose of the Wheel Controllers, and the software developed is also fundamentally different, as will be explained in chapters 12 and 13. The operation of these two units are, however, closely linked. The WCUs will continuously read supervisory control signals and respond to system mode changes from the CCU. As described in Section 3.1.4, the system should also error detection mechanisms that will be based on communication between the units.

It was previously decided that CAN bus should be used as the physical and transfer layer of this protocol. The CAN standard does, however, not implement the higher protocol layers. This allowed a large degree of freedom when the messaging protocol for the CES system was implemented. Specific messages has therefore been implemented to best serve the purpose of this system.

Moreover, the CES system uses the same CAN bus as several other embedded systems of the car. It is therefore important for all these systems implement a common addressing protocol, that allows them to communicate and address each other without collision. This will be further described in Section 11.2.

11.2 Revolve CAN Addressing Protocol

On a CAN bus, all messages are transmitted by broadcasting. All units may read the content of a message sent on the bus. To differentiate between various messages, all messages must be assigned an identifier. Identifiers may be allocated freely, according to the wishes of the developers. It is, however, crucial that all units share a common definition of how the identifiers should be used. In Revolve, there has been several developers working on various electronic systems for the car. It was therefore apparent that a common addressing protocol had to be agreed on.

CAN allows two types of identifiers to be used for messaging. A standard 11-bit identifier, or an extended 29-bit identifier. The 11-bit identifier will allow a number of $2^{11} = 2048$ different message types to be sent. This was deemed to be more than sufficient for the anticipated use on the car. The 11-bit address space has been divided as illustrated in Figure 11.1.

BIT 10...8	BIT 7...3	BIT 2...0
MESSAGE FUNCTION	SYSTEM GROUP	MODULE NUMBER

Figure 11.1: Adressing protocol for CAN messages

11.2.1 Message Function

Embedded systems may have different requirements with respect to real time properties and latencies. This does, of course, also apply to the communication in-between such systems. CAN uses the message identifier for arbitration on the bus. The assigned identifiers will therefore influence

Adress [Bytes 10..8]	**Message Type**
0x000	Engine and Control Unit Messages
0x200	Priority Message
0x400	Command Message
0x600	Data Message

Table 11.1: Table of message types

which message is given priority. A lower identifier gives higher priority on the CAN bus. Given the many developers working on the various embedded systems on the car, it was not practically possible to enforce a rate-monotonic scheduling scheme for identifier allocation, such as proposed in the prestudy.

The addressing scheme will use the three most significant bytes to indicate what sort of function a message has. Table 11.1 shows a list of the message types that has been defined.

As can be seen from the table, all priority messages will be given precedence over command and data messages. Messages used for engine and gear management has been assigned a separate category, as they are crucial to the fundamental operation of the car.

11.2.2 Group Identifiers

To distinguish messages originating from various systems, each subsystem has been assigned its own group identifier. These group IDs determine the 5 middle bytes of the message identifier. Most CAN controllers allow so-called masking of identifiers. This means that messages may be filtered on reception by a subset of bytes in their identifier. If a unit is interested in a message from a particular subsystem, it may apply a filter that allows only those messages. A complete list of group identifiers is given in Table 11.2. The allocation of identifiers has been ordered according to the anticipated real-time requirements of the various systems.

Adress [Bytes 7..3]	Group
0x08	**Gear Control**
0x10	Engine Auxiliary
0x18	**CES Suspension**
0x20	**Dashboard**
0x28	Steering Wheel
0x30	Variable Intake
0x38	**Engine Control Unit IO**
0x40	Inertial Navigation System Sensors
0x48	**Brake Pressure Sensors**
0x50	Exhaust Gas Temperature Sensors
0x58	**Steering Angle Sensors**
0x60	Suspension Damper Displacement Sensors
0x68	Lab Beacon Sensors
0x70	Telemetry System

Table 11.2: Table of module groups (Applicable groups in bold)

11.2.3 Module Numbers

Several of the systems on the car consist of more than one unit. An example is the CES system described in this thesis. A way to distinguish between these units is therefore needed. The addressing protocol has allocated the three least significant bytes for this purpose. This allows a total of 8 units per subsystem.

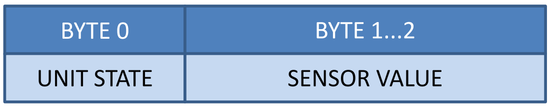
As can be seen from Table 11.3, one module number has been assigned to each of the physical units. There are, however, also messages that has been assigned special identifiers. The Setpoint Control messages are broadcasted at a high frequency. To allow other units to distinguish these messages from the remaining messages from the CCU, they have been assigned special identifiers.

Adress [Bytes 2..0]	Module Number
0x0	Central Controller Unit
0x1	Front Setpoint Control Message
0x2	Rear Setpoint Control Message
0x3	N/A
0x4	Front Left Wheel Controller Unit
0x5	Front Right Wheel Controller Unit
0x6	Rear Left Wheel Controller Unit
0x7	Rear Right Wheel Controller Unit

Table 11.3: List of module codes for CES system

11.3 Reception of Sensor Data

The ability to receive current sensor data is one of the most important properties of the CES system. As indicated in Section 3.1.2, all sensor data will be sent digitally over the Sensor CAN Bus. Most sensor data, e.g. steering wheel position and brake pressure, will be transmitted by dedicated sensor CAN-modules, that transmit data on the form indicated in Figure 11.2. Values are transmitted as a singed 16-bit value. Additionally, a status byte is also included in the message. The coding of unit states are common for all modules, and is shown in Table 11.5. If an error should occur within a sensor CAN module, any listening units may be notified by reading this byte.



BYTE 0	BYTE 1...2
UNIT STATE	SENSOR VALUE

Figure 11.2: Structure of WCU State Message

11.4 Controlling the System

11.4.1 Dashboard Control

As described in Section 3.1.5, the CES system may be controlled by using both the steering wheel, and the dashboard of the car. Primary mode changes, such as activation and deactivation, are controllable by a three position switch on the dashboard. Switch positions will be transmitted as a single message from the dashboard. The structure of this message is shown in Figure 11.3. The dashboard state message will be broadcasted periodically, as well as on any changes, on the Main CAN Bus.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1..0
AUTO GEAR	IGNITION	TRACTION MODE	DATALOG MODE	START BUTTON	CLUTCH MODE	SUSPENSION MODE

Figure 11.3: Structure Dashboard State Message

11.4.2 System Command Messages

For changing more advanced parameters of the system, e.g. using the OLED interface in the steering wheel, a command message structure has been implemented. The structure of this message is shown in Figure 11.4. As can be seen, the first byte is used to specify a command ID, while remaining bytes are used freely depending on the command type. Table 11.4 shows a list of commands that has been implemented for the CES system. All commands in the list use a command content of 16 bits, i.e. 2 bytes, to represent a damping coefficient.

BYTE 0	BYTE 1...7
COMMAND ID	COMMAND CONTENT

Figure 11.4: System Command Message

ID	**Action**
0x21	Change manual mode front left compression coefficient
0x22	Change manual mode front left rebound coefficient
0x23	Change manual mode front right compression coefficient
0x24	Change manual mode front right rebound coefficient
0x25	Change manual mode rear left compression coefficient
0x26	Change manual mode rear left rebound coefficient
0x27	Change manual mode rear right compression coefficient
0x28	Change manual mode rear right rebound coefficient

Table 11.4: System Command IDs implemented for the CES system

State byte	**State name**
0x00	**Error**
0x01	**Operative**
0x02	**Standby**
0x03	Fault
0x04	**Manual**

Table 11.5: Table of system states (Applicable states in bold)

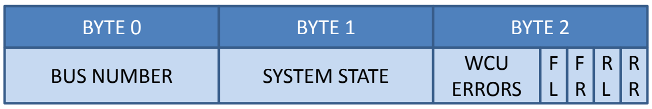
11.5 Unit State Messages

Error detection and fail-safe functionality will to a large degree rely on the CCU and WCUs ability to monitor each other. The Central Controller Unit must also be able to broadcast the current state of the system to the wheel controllers, so that they may respond to any state changes.

For these purposes, state messages has been implemented for each unit, as will be described in sections 11.5.1 and 11.5.2. To achieve a unified representation of states across all units, a common coding for states has been implemented. This coding is shown in Table 11.5.

11.5.1 Central Controller State Message

The Central Control Unit will broadcast a state message on both CAN buses. On the Sensor CAN Bus, this message will be used by the Wheel Controller Units to determine the state of the system. On the Main CAN Bus, this message will be used by the dashboard to indicate failure if the system is in an erroneous state. The structure of the message is shown in Figure 11.5. The State Message will be transmitted as a data message with the ID of the CCU.



BYTE 0	BYTE 1	BYTE 2				
BUS NUMBER	SYSTEM STATE	WCU ERRORS	F L	F R	R L	R R

Figure 11.5: Structure of CCU State Message

As this message is transmitted on both buses, the first field will be used to indicate which bus the message has originated from. The second field of the message will indicate the current state of the system. This field will be used by the WCUs to determine the state of the system. The CCU state message has also implemented error flags for each of the WCUs. This field may be used when debugging the system.

It is also important to be able to externally monitor the behaviour of the system. The state and any error flags will therefore be logged continuously by the data acquisition system of the car.

11.5.2 Wheel Controller State Message

The Wheel Controller Units will also transmit a state message on the Sensor CAN Bus. The ID of these status messages will be the identifier allocated to each WCU in Table 11.3. The structure of this message is shown in Figure 11.6. As can be seen from the figure, the first field of the message will indicate the state of the originating WCU. This value will be used by the CCU for error detection.

Additionally, representations of the valve positions has been included for debugging purposes. These values has been coded as signed 8-bit integers.

BYTE 0	BYTE 1	BYTE 2
UNIT STATE	COMPRESSION VALVE POSITION	REBOUND VALVE POSITION

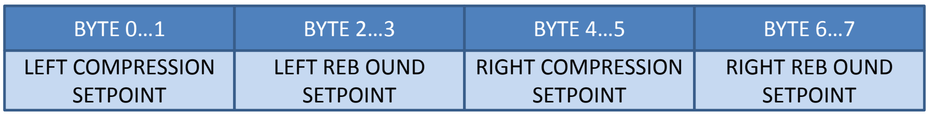
Figure 11.6: Structure of WCU State Message

11.6 Setpoint Control Messages

Perhaps the most important of all the messages used in the CES system is the Setpoint Control Messages. These messages will transmit supervisory control signals to each Wheel Controller Unit, in the form of setpoints for the valve PID controllers. As will be described in Chapter 12, no closed loop control of the suspension has yet been implemented for the system. It has, however, been the target to design a system that will allow a high bandwidth closed loop controller to be implemented at a later stage.

There will always be a certain overhead associated with transmission of data over CAN. The meta-data transmitted in a CAN frame is however of fixed length. More data per frame will thus lead to a lower overhead during transmission. As indicated in Section 2.1.3, a high bandwidth controller may require a frequency of up to 500 Hz. It is therefore desirable to minimize the overhead of each message.

Each of the setpoint values transmitted to the WCUs will be of 16-bit length, i.e. two bytes. A total of four setpoints may therefore be transmitted in a single message. As can be seen from Table 11.3, two messages has been implemented for the front, and rear axle setpoints. This will allow all 8 bytes of a CAN message to be utilized, thus giving the least possible overhead for messages. The structure of the Setpoint Control Messages has been shown in Figure 11.7.



BYTE 0…1	BYTE 2…3	BYTE 4…5	BYTE 6…7
LEFT COMPRESSION SETPOINT	LEFT REB OUND SETPOINT	RIGHT COMPRESSION SETPOINT	RIGHT REB OUND SETPOINT

Figure 11.7: Structure of Setpoint Control Messages

12 Central Controller Software

"If your culture doesn't like geeks, you are in real trouble."

- Bill Gates

12.1 Requirements

12.1.1 Execution of Controller Algorithm

An important role of the Central Controller Unit is to execute the main control algorithm of the CES system. The software of the CCU must therefore implement a framework, that allows the execution of a controller algorithm with accurate period. It must also be possible to work on this algorithm without in-depth knowledge of the remaining system.

The software of the system must be designed in such a way that it will allow the implementation of complex controller algorithms, such as described in Section 2.1.3.

12.1.2 Control of Wheel Controller Units

For the CES system to work, the CCU must obviously be able to send supervisory control signals to the Wheel Controller Units. This should be done according to the message standard defined in Section 11.6. The software must be able to send these setpoints with a frequency equal to the execution of the main controller algorithm.

12.1.3 State Control

The Central Controller Unit will be responsible for managing the state of the CES system. This state will be determined by a switch located on the dashboard of the car. The Wheel Controller Units will in turn determine their state by messages sent from the CCU. These message must therefore be sent periodically. State messages should be sent on the form specified in Section 11.5. This section also specifies a common representation of states for all systems on the car. The CCU should manage its states according to this representation.

12.1.4 Gathering of Sensor Data

The CCU will read the necessary sensor data from the CAN bus of the data acquisition system. To allow implementation of high bandwidth control algorithms, the CCU must be able to read sensor data sent at a high frequency, typically above 200 Hz per message. Reception of these messages must be done without disturbing the flow of the remaining program.

12.1.5 Run-time Debugging

An RS-232 interface has been implemented for the CCU, to allow run-time debugging. A driver must be implemented for this interface, allowing flexible use in the program code.

12.1.6 Error Detection

As described in Section 3.1.4, a method for error detection must be implemented in the CES system. The Central Controller Unit will be responsible

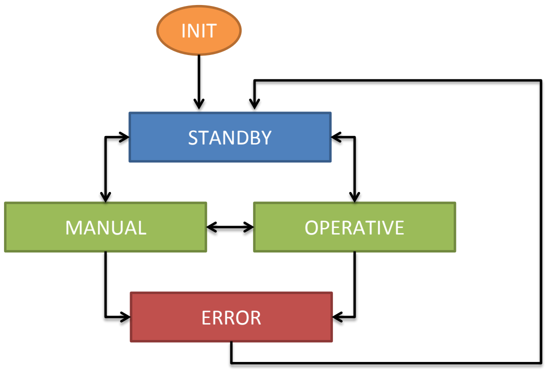


Figure 12.1: State diagram for Central Controller Unit

for monitoring the state of the Wheel Controller Units. This should be done by monitoring the reception of unit state messages. If any errors should be indicated, or the reception of messages ceases, the system must be put in a fail-state. The system must also be able to interface the external watchdog, as described in Section 6.2.2. The watchdog must be reset with a period of maximally 256ms.

12.2 Software Architecture

This section will give a description of the software architecture chosen for the Central Controller Unit.

12.2.1 State Management

As described in Section 12.1.3 the CCU is responsible for managing the state of the CES system. A list of applicable states for the system is found in Table 11.5. The state of the unit will determine the operating mode of the system. Figure 12.1 shows a state diagram for the CCU. A description of the various system states follows below.

Standby

Once the system is powered up it will enter a default mode where all control is suspended, until the Central Controller Unit initializes the mode of the system. In the *standby* mode all units operate independently, and the wheel controller units will assume a locally defined position. If the driver wishes to disable the system, this state should suspend all supervisory control of the wheel controller units.

Manual

As discussed in 2.1.3 there are several control strategies that could be applied to an adaptive suspension system. The simplest way a such a system may be controlled is, however, by manual adjustment of the control parameters. Such a mode has also be implemented for the system in this mode. When in the *manual* mode, the driver may change the static settings for the suspension system. No dynamic adjustment of the suspension will be enabled. The CCU will instead send a static control signal to the Wheel Controller Units. Error detection will be enabled to monitor the state of the system.

Operative

In *operative* mode, the dynamic controller algorithm implemented in the CCU is activated. Further details about this algorithm may be found in Section 12.6.1. Based on input from sensor data the CCU will continuously send control signals to the wheel controller units, with a predefined frequency. As in manual mode, error detection the system will be activated so that the system may fail safely.

Error

To achieve the desired level of fail-safe functionality for the system, the units need to be able to detect internal and external errors. Once an error is detected, the system will suspend active control to isolate any failures and give predictable behaviour. This will be triggered by the system going into the *error* state. The state of the CCU will be continuously broadcasted in the unit state message described in Section11.5. The Wheel Controller Units will, thus, be alerted of the error.

Once in an error state, it is not possible to exit this state until the error condition has been reset. Error must be reset by putting the system into standby mode before a new mode is selected.

12.2.2 Operating System

As was seen from Section 12.1, the CCU is responsible for executing several functions that are more or less independent of each other. The responsibilities of the CCU may be divided into three primary groups:

- Controller execution
- System management and monitoring
- Data acquisition

For the system to function as intended, it is important that all these tasks are executed periodically. It was therefore decided to base the software architecture for the Central Controller Unit around an operating system. The use of an OS was also recommended by the prestudy, for similar reasons. As the author was already familiar with the FreeRTOS embedded real-time OS from the experiments performed in the prestudy it was decided to use this OS for implementation of the CCU. The rest of this section will give an introduction to FreeRTOS and present some of its main features.

FreeRTOS

FreeRTOS is an embedded real-time operating system that may be easily deployed on embedded systems. The base implementation of FreeRTOS features some of the most elementary, nevertheless effective, mechanisms of an operating system. FreeRTOS is distributed as several libraries, written in C, and may easily be included in a software project.

Scheduler/Kernel	236 bytes
Each queue	76 bytes + storage of elements
Each task	64 bytes + task stack size

Table 12.1: RAM usage by applicable freeRTOS features

As the CCU will have to perform several concurrent processes, a division into tasks can be used to control the execution of the program. Chapter 7 of the prestudy also performed a simple evaluation of the real-time performance of freeRTOS, and found them to be satisfactory. For this application, elementary features such as scheduling of tasks and inter-task communication has been the most important. Table 12.1 shows a the required RAM for the freeRTOS features that has been used in this application. Within reasonable usage, this should be negligible compared to the total size of a program.

Kernel

freeRTOS is a micro-kernel operating system, meaning that the kernel only includes the very basic features of the OS. All other services are added as modules. Since the UC3C devices does not implement a memory manager, the role of the kernel is limited to the execution of the scheduler. freeRTOS features a real-time task scheduler with support for preemption. The scheduler itself is started by calling the function `vTaskStartScheduler()`.

Tasks

One of the primary strengths of FreeRTOS is its ability to run several concurrent tasks. A task is created by the implementation of a task function, as shown in Listing 12.1. There are few restrictions to the form of a task function, however, it is important to notice that a task will terminate once its function is finished.

The listing also shows how a task is created by using the function `xTaskCreate()`. This function will add the task function, referenced by the function pointer in its first argument, to the scheduler. The remaining parameters define respectively the task's name, maximal stack size, parameters passed to the task, scheduler priority and a handle pointing to the task.

Listing 12.1: Example definition and dispatch of a FreeRTOS task

```c
void vATask( void *pvParameters ){
  // Task content
}

int main(void){
  ...
  // Send task to the scheduler
  xTaskCreate(vATask, (signed char*)"TASKNAME", STACK_SIZE, &
      taskParameters, tskPRIORITY, NULL);
  // Start scheduler. All tasks are dispatched
  // Execution of main function will stop here
  vTaskStartScheduler();
  ...
}
```

Another important property of a real-time system is its ability to execute tasks periodically. FreeRTOS implements two mechanisms to suspend a task for a specified period of time:

- `vTaskDelay(portTickType xTicksToDelay)` may be called from inside a task to suspend the task for a specified number of *ticks*
- `vTaskDelayUntil(portTickType *pxPrevWakeTime, portTickType xTime)` may be called from inside a task to suspend the number task for a number of *ticks* relative to a specified time specified in the parameter `pxPreviousWakeTime`. This function is very useful for creating periodic execution of tasks.

Queues

Another important feature of freeRTOS is its implementation of queues. Queues may, for instance, be used for safe communication between tasks. For the software of the CCU, queues has been particularly useful to store incoming CAN messages before they are processed by the application.

Listing 12.2: Example initialization and usage of a FreeRTOS queue

```
// Global declaration of queue handle
xQueueHandle xQueue;

// Task initializes and posts to queue
void vATask( void *pvParameters ){
  ...
  xQueue = xQueueCreate( 16, sizeof( uint8_t ) );
  uint8_t valueToSend = 42;
  ...
  while(1){
      ...
      // Post variable to queue
      // Wait for 10 ticks if queue is full
      xQueueSend( xQueue, ( void * ) &valueToPost, ( portTickType )
           10 );
  }
}

// Task receives values from queue
void vATask( void *pvParameters ){
  ...
  // Initialize buffer for read messages
  uint8_t rxedMessage;
  ...
  while(1){
      // Read variable from queue to buffer
      // Wait for 10 ticks if no message on queue
      xQueueReceive( xQueue, &( rxedMessage ), ( portTickType ) 10
          )
      ...
  }
}
```

Listing 12.2 shows an elementary example of how a queue may be initialized and used to send data from one task to another. Once a queue has been initialized, data is sent and read from the queue simply by using respectively the `xQueueSend()` and `xQueueReceive()` functions.

12.2.3 ASF

The Atmel Studio Framework, abbreviated ASF, is a software library providing drivers and software libraries for several of Atmel's Microcontrollers. This framework is particularly useful for the AVR32 UC3 architecture, which in general require somewhat more complicated drivers than the traditional 8-bit architecture. The structure of ASF is shown in Figure 12.2.

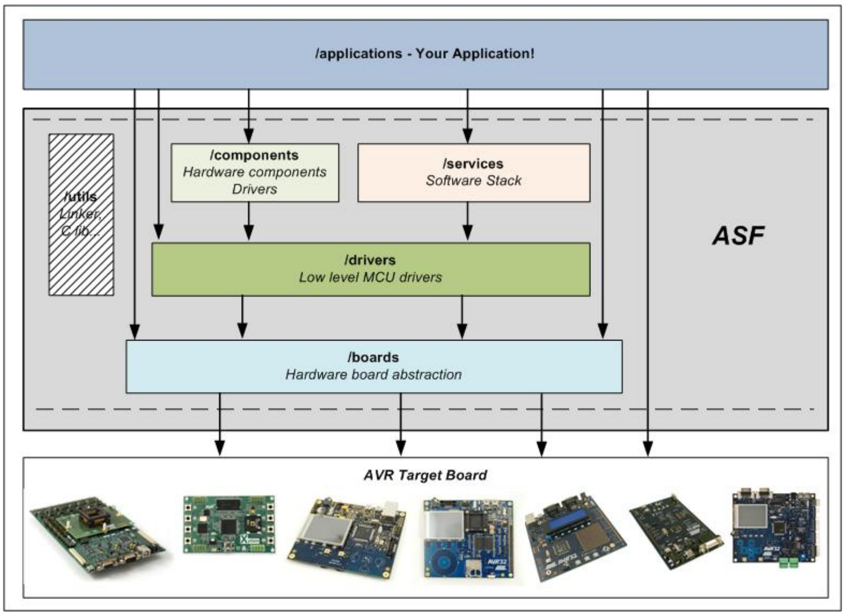


Figure 12.2: Atmel Software Framework modules structure [22]

To shorten the development time of the CCU, libraries has been used from ASF where available. Below is a list of drivers and libraries that has been used for the development of the software for the Central Controller Unit.

SPI Master API This is an abstractive library for SPI communication. The API may be used for several of Atmel's devices, allowing the same program code to be used across devices. The API links further to an ASF driver for SPI on the particular unit.

System Clock Control This library allows the programmer to easily control the system clocks of the microcontroller, from within the application. Clock sources, PLL multipliers, etc. may be altered directly from the application code. This is a common library that links directly to the a particular driver for the microcontroller used.

GPIO Driver This library allows the microcontroller to interface GPIO resources of the microcontroller.

CAN Software Stack This library implements a complete software stack for CAN communication. Messaging on the CAN interfaces may be done with simple software calls. The library interfaces a driver for the CAN controller of the microcontroller.

FreeRTOS A preconfigured version of the embedded real-time OS is available in the ASF. Further description of FreeRTOS is available in 12.2.2.

USART Driver This library implements a driver for the USART interface of the microcontroller. Commands are provided for initializing the interface, as well as basic transmission and reception of characters.

Stdio Redirection This library supports redirection of the stdio library to the buffers of the USART channel of the microcontroller. This library also links to the particular USART driver for the microcontroller used.

12.3 CCU Drivers

Although the libraries of the Atmel Software Framework has been used extensively for this application, it has also been necessary to implement drivers for the external components of the unit. Some high level abstraction libraries has also been implemented to simplify the code. This section will present each of these drivers.

12.3.1 CAN Driver

The CAN software stack available in ASF has been used to implement CAN communication for the Central Controller Unit. This software stack allows sending and receiving of messages by simple function calls. It also defines the necessary structures for CAN messages and so-called *mailboxes*(MOBs). Listing 12.3 shows the definition of the MOB and message structures used by the CAN software stack.

Listing 12.3: Declaration of data structures for CAN MOB and message

```
typedef struct
{
  union{
    struct{
      U32   id          : 32;
      U32   id_mask     : 32;
    };
    struct{
      // Details of content omitted
    };
  };
 Union64 data;
} can_msg_t;

typedef struct{
  U8 handle;
  can_msg_t *can_msg;
  U8 dlc;
  U8 req_type;
  U8 status;
}can_mob_t;
```

The `can_msg_t` structure is used to save the content of a single message. It also specifies a message identifier, and in case of reception an ID filtering mask. The `can_mob_t` structure is used to control the transmission of reception of a CAN message. This structure is passed to the `can_rx()` and `can_tx()` functions of the software stack, to register the message for respectively reception or transmission. A *handle* is used to identify the message type upon reception, or when the message has been sent.

As explained in Chapter 11, the CCU will have to process several types of CAN messages. MOB and message structures has therefore been defined for all these types of messages. A separate library has been defined for this purpose, named `ccu_can.c`. This library also includes a necessary function to initialize the CAN controllers of the CCU. The initialization of the CAN controllers is shown in Listing 12.4 and 12.5. Program code for the declaration of the applicable message structures has been included in the appendices, in Listing C.1.

Listing 12.4: Initialization sequence for CAN interfaces - main.c

```
// Initialize CAN channels
ccu_can_init();
// Initialize CAN controllers with ASF library
can_init(0, ((U32)&mob_ram_ch0[0]), CANIF_CHANNEL_MODE_NORMAL,
    can_out_callback_channel0);
can_init(1, ((U32)&mob_ram_ch1[0]), CANIF_CHANNEL_MODE_NORMAL,
    can_out_callback_channel1);
```

Listing 12.5: Initialization of CAN interfaces - ccu_can.c

```
void ccu_can_init(void){
  // Assigns a clock source to the CAN controllers.
  init_sys_clocks();

  // Assign GPIO to CAN.
  gpio_enable_module(CAN0_GPIO_MAP, sizeof(CAN0_GPIO_MAP) / sizeof(
      CAN0_GPIO_MAP[0]));
  gpio_enable_module(CAN1_GPIO_MAP, sizeof(CAN1_GPIO_MAP) / sizeof(
      CAN1_GPIO_MAP[0]));

  // Create queues for handling received messages
  // Reception of WCU state message
  ch1_wcustate_rx_receivequeue = xQueueCreate(8,sizeof(can_msg_t));
  // Reception of dashboard messages
  ch0_dashboard_rx_receivequeue = xQueueCreate(8,sizeof(can_msg_t))
      ;
  // Reception of system command messages
  ch0_extcmd_rx_receivequeue = xQueueCreate(8,sizeof(can_msg_t));
  // Reception of ECU data messages
  ch0_ecu_rx_receivequeue = xQueueCreate(16,sizeof(can_msg_t));
  // Reception of sensor data messages
  ch1_sensor_rx_receivequeue = xQueueCreate(16,sizeof(can_msg_t));
}
```

12.3.2 SPI Driver

As described in Section 12.2.3, a library from ASF has been used for implementing SPI communication. However, as this driver is used repeatedly throughout the code, another layer of abstraction has been implemented by a simple library. An identical library has been implemented for the Wheel Controller Unit(refer to Section 13.3.1), so that drivers for SPI components may be reused. Function declarations for this library is shown in Listing 12.6. `spi_init()` – a function to initialize the SPI interface with predefined settings has also been implemented.

Listing 12.6: Function Prototypes - ccu_spi.h

```
//--- FUNCTION PROTOTYPES
void spi_init(void); // Init SPI with predef. settings
void spi_sbc_select(void);    // Select SBC chip on SPI
void spi_sbc_deselect(void); // Deselect SBC chip on SPI
uint8_t spi_txrx_byte(uint8_t sendbyte); // SPI TX and RX byte
void spi_tx_byte(uint8_t sendbyte); // SPI transmit byte
```

12.3.3 System Basis Chip Driver

As described in Section 6.2.2, a System Basis Chip including a 5V power supply, a CAN transceiver and an external watchdog, has been used in the hardware design. Drivers had to be developed for SBC to perform as expected.

Figure 12.3 shows a simplified version of the state diagram from the datasheet[14, p. 41]. As can be seen from the figure, the SBC will go into a initialization-mode at power up. If the watchdog is not initialized within

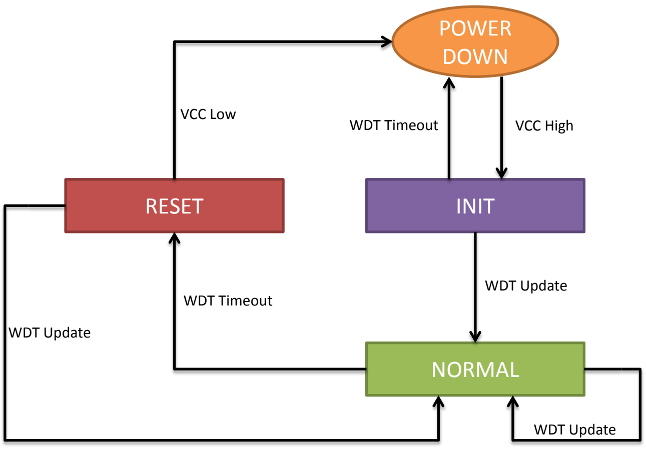


Figure 12.3: Simplified state diagram for the MC33903

the first 10 seconds, the power supply to the circuit will be turned off. To initialize the watchdog, a WDT Refresh command must be sent over SPI. After the watchdog has been initialized these messages must be sent every $256ms$ at a minimum. If a message fails to arrive, the microcontroller will be reset.

While the SBC is in init-mode, its internal settings may be configured. For this application, it was necessary both to enable power to the CAN transceiver, and to enable the transceiver itself. The watchdog was also reconfigured to operate in it's simplest mode, so that it may be reset with a single command.

Listing 12.7 shows the content of the driver that has been implemented for the SBC. Comments in the listing indicate the purpose of each SPI command. At the end of the initialization sequence, messages are printed to the console for debugging.

Listing 12.7: Function Prototypes and Variables - sbc_mc33903.c

```c
volatile void sbc_init(void){
  volatile uint8_t readbyte[10] = { 0 };

  //SET WATCHDOG CONFIGURATION
  spi_sbc_select(); // Set CS for SBC low(active)
```

```
    spi_txrx_byte(0b01001100);
    spi_txrx_byte(0b00000001);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    delay_ms(1);

    // ENABLE INTERNAL 5V SUPPLY TO CAN TRANSCEIVER
    spi_sbc_select(); // Set CS for SBC low(active)
    readbyte[0] = spi_txrx_byte(0b01011110);
    readbyte[1] = spi_txrx_byte(0b00001000);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    delay_ms(1);

    // SET CAN TRANSCEIVER TO ON
    spi_sbc_select(); // Set CS for SBC low(active)
    readbyte[2] = spi_txrx_byte(0b01100000);
    readbyte[3] = spi_txrx_byte(0b11000000);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    delay_ms(1);

    // READ CAN STATUS
    spi_sbc_select(); // Set CS for SBC low(active)
    readbyte[4] = spi_txrx_byte(0b00100001);
    readbyte[5] = spi_txrx_byte(0b00000000);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    delay_ms(1);

    //EXIT INIT MODE
    spi_sbc_select(); // Set CS for SBC low(active)
    readbyte[6] = spi_txrx_byte(0b01011010);
    readbyte[7] = spi_txrx_byte(0b00000000);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    delay_ms(1);

    // READ MODE
    spi_sbc_select(); // Set CS for SBC low(active)
    readbyte[8] = spi_txrx_byte(0b11011101);
    readbyte[9] = spi_txrx_byte(0b10000000);
    spi_sbc_deselect(); // Set CS for SBC high(deactive)

    // PRINT CONTENT OF MESSAGES
    debugPrintf(4, "\r\nSBC Initialized. Printing SPI data:\r\n");
    debugPrintf(4, "MSG1: %x\r\n",((readbyte[0]<<8)+readbyte[1]));
    debugPrintf(4, "MSG2: %x\r\n",((readbyte[2]<<8)+readbyte[3]));
    debugPrintf(4, "MSG3: %x\r\n",((readbyte[4]<<8)+readbyte[5]));
    debugPrintf(4, "MSG4: %x\r\n",((readbyte[6]<<8)+readbyte[7]));
    debugPrintf(4, "MSG5: %x\r\n\r\n",(readbyte[8]<<8)+readbyte[9]);
}
```

```

// Send Refresh Command to SBC Watchdog. Must be sent every 256 ms
volatile void sbc_wdt_refresh(void){
  //REFRESH WDT
  spi_sbc_select(); // Set CS for SBC low(active)
  spi_txrx_byte(0b01011010);
  spi_txrx_byte(0b00000000);
  spi_sbc_deselect(); // Set CS for SBC high(deactive)
}
```

12.3.4 USART Driver

As defined in Section 12.1, a simple way to allow debugging of the software must be implemented. The hardware implementation addressed this requirement in Section 6.2.5 by implementing an RS-232 interface that may be used as a debug console for the unit. A driver is, however, required to utilize the potential of this console.

ASF also provides a library to redirect input and output from the standard `stdio` library, to the buffers USART controller. In this way functions, such as `printf()` and `scanf()`, may be used to correspondingly write or read strings of data from the interface. A specific library for the CCU has however been implemented with some useful functions. This library is shown in Listing 12.8.

The `debugPrintf()` function is a wrapper function for the regular `printf()` function of the stdio library. This function allows the programmer to define a *debug level* for each print command. Only commands with a debug level higher or equal to the DEBUG_LEVEL constant will be printed.

Listing 12.8: Function Prototypes and Variables - ccu_usart.h

```
#define DEBUG_LEVEL 4

// Initialize USART with predef. properties and redirect stdio
void ccu_usart_init(void);

// Calls 'printf()' if define DEBUG_LEVEL > argument debugLevel
void debugPrintf(uint8_t debuglevel, const char* format, ...);
```

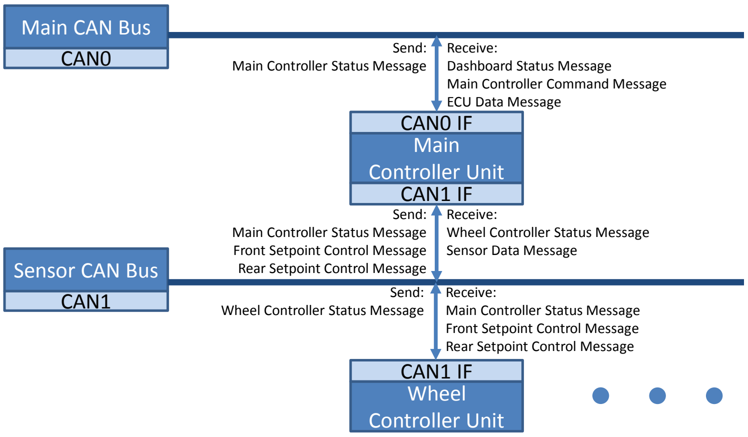

Main CAN Bus
CAN0
Send:
Main Controller Status Message
Receive:
Dashboard Status Message
Main Controller Command Message
ECU Data Message
CAN0 IF
Main Controller Unit
CAN1 IF
Send:
Main Controller Status Message
Front Setpoint Control Message
Rear Setpoint Control Message
Receive:
Wheel Controller Status Message
Sensor Data Message
Sensor CAN Bus
CAN1
Send:
Wheel Controller Status Message
Receive:
Main Controller Status Message
Front Setpoint Control Message
Rear Setpoint Control Message
CAN1 IF
Wheel Controller Unit

Figure 12.4: Overview of can messages used by the CES system

12.4 CAN Message Handling

As described in Section 12.2.2, freeRTOS supports the implementation of message queues. This feature has been particularly useful for handling reception of messages in the CCU. Figure 12.4 shows an overview of the messages that are used by the CES system. As CAN be seen from the figure, several different types of messages will have to be handled by the CCU.

A message queue has been implemented for each message type. Once a message is received, the implemented callback for the CAN controller function adds the message to the corresponding message queue. Figure 12.6 shows gives a graphical illustration of the message queues implemented. An extract of the callback function, showing the handling of a sensor data message, is shown in Listing 12.9.

Listing 12.9: Excerpt of CAN1 callback function - main.c

```c
// CALLBACK FUNCTION FOR CAN1 CONTROLLER
volatile void can_out_callback_channel1(U8 handle, U8 event){

  // Check if message is sensor data message
  if (handle == ch1_sensor_rx_msg.handle){
    // Read message to buffer
    ch1_sensor_rx_msg.can_msg->data.u64 = can_get_mob_data(1,handle
        ).u64;
    ch1_sensor_rx_msg.can_msg->id = can_get_mob_id(1,handle);
    ch1_sensor_rx_msg.dlc = can_get_mob_dlc(1,handle);
    ch1_sensor_rx_msg.status = event;

    // If message has correct length, add to queue
    if(ch1_sensor_rx_msg.dlc == 3){
    xQueueSendToBackFromISR(ch1_sensor_rx_receivequeue,
        ch1_sensor_rx_msg.can_msg, NULL);
    }

    // Prepare for receiving new message
    can_rx(1,
    ch1_sensor_rx_msg.handle,
    ch1_sensor_rx_msg.req_type,
    ch1_sensor_rx_msg.can_msg);
  }
      // REST OF FUNCTION OMMITTED FROM EXAMPLE!
}
```

12.5 CCU Tasks

To utilize the potential of freeRTOS, the system has been divided into multiple tasks. This way, the various functions of the CCU may be executed independently of each other. Figure 12.5 shows an overview of the tasks that have been implemented. Below follows a description of each of the tasks.

12.5.1 Management Task

The Management Task does, as the name indicates, manage the operation of the Central Controller Unit, and the CES system as a whole. This includes the processing of commands and messages from the dashboard. Monitoring of the state messages from the WCUs has also been implemented in this task. Figure 12.6a the message queues implemented for reception of CAN messages in this task.

A timer has been implemented for monitoring each WCU. Status messages from the WCUs are registred continously. Every time a message is received,

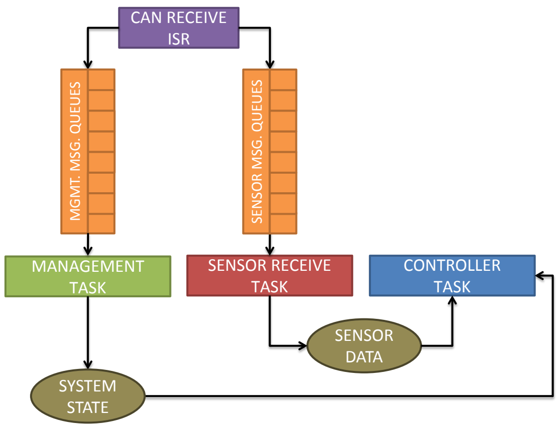


Figure 12.5: Overview of tasks on the CCU

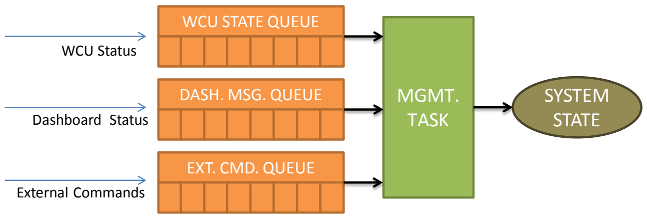
the timer is reset. If a timer expires, the system is put into an error state, according to the description in Section 12.2.1. The management task also transmits the status messages of from the CCU. The state of the system, as well as any error flags from the WCUs, will be sent in these messages.

12.5.2 Controller Task

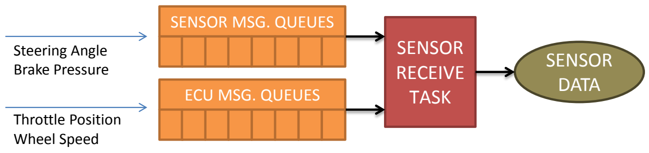
The Controller Task implements a framework for execution of the main controller algorithm of the CES system. The controller algorithm is executed with a fixed period, which for testing purposes has been set to 10ms. After each iteration of the control algorithm, the task will transmit CAN Setpoint Control Messages to the Wheel Controller Units, as defined in Section 11.6. Program code for the task function `vControllerTask()` is shown in Appendix C.1.2. Furthermore, the controller loop function will be studied in Section 12.6.1.

12.5.3 Sensor Receive Task

As defined in the requirements for the CCU, the unit must be capable of efficiently processing sensor data that is received on the Sensor Can Bus.



(a) Management Task



(b) Sensor Receive Task

Figure 12.6: Reception of CAN messages for system threads

A separate task has therefore been defined for this purpose. As opposed to the controller task and the management task, that should be executed with a fixed period, the *sensor receive task* should process incoming messages as quickly as possible. For testing purposes, the period of this function has nevertheless been limited to 1ms. Figure 12.6b shows the flow of sensor data messages received from the ECU and the CAN sensor modules. Two queues have been implemented to handle the reception of these messages.

Each queue is read once for every time the task is executed, to achieve a predictable execution time. Assuming that the sensor data messages are sent periodically, the rate of these messages for each message type assigned to a queue is limited by equation 12.1.

$$f_{max_{msg}} = \frac{1}{P_{task} \cdot N_{msg}} = \frac{1}{N_{msg}} Khz \tag{12.1}$$

Where P_{task} and N_{msg} is respectively the period of the task, and the number of messages assigned to the queue. As was seen in Figure 12.6b, each queue was assigned with two message types, giving a threshold of 500Hz each. This should be sufficient given the requirements in Section 12.1.

12.6 CCU Application

The previous sections of this chapter has presented the drivers and framework necessary to execute the main controller algorithm for the CES system. The development of a supervisory control algorithm has not been part of this assignment. However, work done by other team members of Revolve has given a prototype algorithm that can be used for evaluating the performance of the system. This section will give a brief introduction to the control algorithm that has been implemented.

12.6.1 Supervisory Controller

A library with several functions has been implemented for managing the Main Controller of the CES system. Prototypes of the applicable functions are shown in Listing 12.10. Two of these functions are of particular importance. The function `ccu_controller_loop()` is executed periodically to generate new setpoints for the wheel controller units. The `ccu_controller_init()` is used to initialize the controller with a particular mode.

Listing 12.10: Function prototypes for CES main controller - ccu_controller.h

```c
// Get the current mode of the controller
int8_t ccu_controller_getControlMode(void);
// Initialize the controller, or change mode
void ccu_controller_init(uint8_t controllerMode);
// Update static settings for MANUAL mode
void ccu_controller_changeStaticParam(uint8_t position, uint16_t
    damperParameter);
// Execute controller loop
void ccu_controller_loop(void);
```

The prototype controller algorithm implemented for the CES system does not implement a closed loop controller. Instead, fuzzy logic has been used to determine certain known states of the vehicle. This has been done to achieve a system that may be easily tuned, and tested. The implemented vehicle-states are shown in Listing 12.11.

Optimal values for all damping parameters within a state have been calculated using suspension simulations. These predefined damping coefficients have been implemented in the array `ccu_damper_coeff_setpoints`. Once a coefficient has been determined from the vehicle-state array, it must be converted to a position reference for the wheel controller. This is done using the function `damper_compression_coeff_to_position()`. The output variables are stored in a data structure that is read by the Controller Task. The conversion function uses a conventional lookup table to convert between the ranges of values.

Listing 12.11: Fuzzy logic evaluation of vehicle state - ccu_controller.c

```c
// Begin with default state STRAIGHT
VehicleState_t currentVehicleState = STRAIGHT;

if (ccu_sensorValues.gearSignal){
  currentVehicleState = GEAR_SHIFT;
}
if(ccu_sensorValues.brakePressure > BRAKE_PRESSURE_THRESHOLD){
  currentVehicleState = BRAKING;
}
if(ccu_sensorValues.throttlePosition > THROTTLE_THRESHOLD){
  currentVehicleState = ACCELERATING;
}
if(ccu_sensorValues.steeringPosition < RIGHT_TURN_THRESHOLD){
  currentVehicleState = RIGHT_TURN;
}
if(ccu_sensorValues.steeringPosition > LEFT_TURN_THRESHOLD){
  currentVehicleState = LEFT_TURN;
}

// Set damping coefficients to output struct
ccu_controlVariable_output.flCompression =
    damper_compression_coeff_to_position(ccu_damper_coeff_setpoints
    [currentVehicleState][0]);
// Following lines omitted...
```

13 Wheel Controller Software

"Computers are like Old Testament gods; lots of rules and no mercy."

- Joseph Campbell

13.1 Requirements

13.1.1 Valve Control

The Wheel Controllers are responsible for actuation of the damper valves. A PID controller must be implemented to allow adjustment of these valves. Simulations by the team members responsible for the main control algorithm has verified that the cycle time of the PID controller must be 2ms at a minimum, to achieve required bandwidth and response time for the main control algorithm to perform correctly.

To implement a PID controller, the Wheel Controller Unit must also be able to read values from the angular sensors. Generation of PWM signals is also required to drive the H-bridge implemented in the circuit. As the angular sensor is only absolute within one rotation, the WCU must be able to calibrate the position of the valve once it is powered up.

13.1.2 Reception of Control Signals

The WCUs do not implement a controller for the adaptive suspension themselves, but are dependant on receiving setpoints from the Central Controller Unit. The WCUs are also dependant on monitoring the state of the CCU, to determine which state they should be in. The reception of messages must be implemented according to the definitions given in Chapter 11.

13.1.3 Error Detection

Like the CCU, the Wheel Controller Units must also participate in error detection for the system. If a WCU fails to receive the unit state messages sent from the CCU, it should enter an error state. The WCUs must also send periodical unit state messages to the CCU.

13.1.4 Fail-Safe

As described in the top level requirements, the CES system must implement fail-safe functionality. The Section 13.1.3 describes how the Wheel Controller Unit may detect an error in the system. If the unit goes into an error state, it is important that the car will behave in a predictable manner. All damper control must be suspended. To exit the error state, the system should first have to be disabled.

13.1.5 Bootloading from CAN

The wheel controller units should be placed inside a sealed enclosure mounted on each damper. It may therefore be cumbersome to access the normal programming port of the unit. To avoid this issue, a boot loader should be implemented. While the prestudy suggested that the WCUs should download their firmware from the CCU, this is deemed to require too much effort. The bootloader should therefore implement a bootloader supporting Atmel Flip, as descibed in [23].

13.2 Software architecture

13.2.1 Big-While

The software of the Wheel Controller Unit is significantly less complex than for the Central Controller Unit. Its main task is to run a PID controller for each valve of the damper. Execution of the controllers should happen with a fixed period. Reading of valve position and application of new control output to the motors is done every time the controller loop is executed.

In addition to the valve controller, the WCU must respond to and send CAN messages which should be handled as quickly as possible. Given these requirements, it was not found the usage of an operating system could not be justified. The prestudy therefore suggested a traditional *big-while* architecture, which have been adopted for this system.

The big-while structure has been centred around the `main.c` file of the system. Figure 13.1 shows the top level structure of the WCU software.

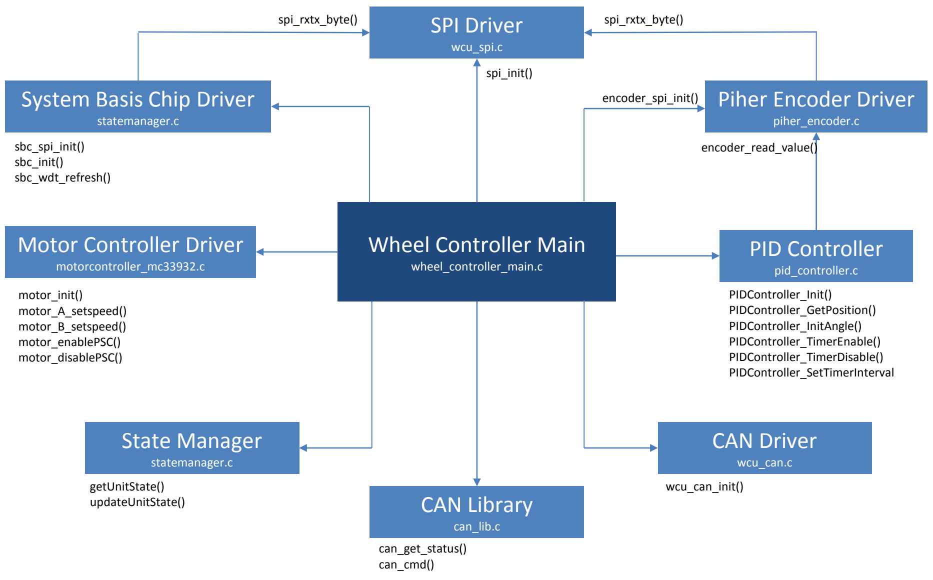


Figure 13.1: Top level structure of WCU software

13.2.2 State Management

The Wheel Controller Units will adopt the same states as was described in Section 12.2.1 for the Central Controller Unit. Please refer to this section for further details. Unit State Messages will be transmitted from the CCU to the WCUs as indicated in Section 11.5.

13.3 WCU Drivers

13.3.1 SPI Driver

SPI communication on the 8-bit AVR architecture is done simply by writing and reading the `SPDR` register of the microcontroller. Before the interface can be used there is, however, a need to configure the interface with the desired configuration. Te be able to reuse drivers for SPI components from the Central Controller Unit, it was also desirable to implement a simple driver for the SPI interface. Listing 13.1 shows the function prototypes for this driver. As can be seen, the receive and transmit functions are identical to those shown in Section 12.3.2.

Listing 13.1: Function Prototypes and Variables - wcu_spi.h

```
// Initialize SPI channel with predefined configuration.
void spi_init();

// Transmit single byte on SPI
volatile uint8_t spi_txrx_byte(uint8_t sendbyte);

// Transmit AND read single byte on SPI
volatile void spi_tx_byte(uint8_t sendbyte);
```

13.3.2 Motor Driver

A dual H-bridge chip was chosen to drive the DC motors used for valve actuation in the Wheel Controller Unit. The H-bridge was paired with the so-called *power stage controller* of the ATmega64M1 microcontroller. The PSC is in many ways identical to a conventional counter, so the output of the PSC may be used to generate a PWM signal. To allow simple usage of the motor driver circuitry, a library has been implemented. Two PSCs, one for each motor, has been used. Each PSC has implemented two outputs that has been assigned to forwards and reverse operation of one motor. Figure 13.2 shows the operation cycle of a PSC in *centred mode*. This implies that the counter is alternating between incrementation and decrementation.

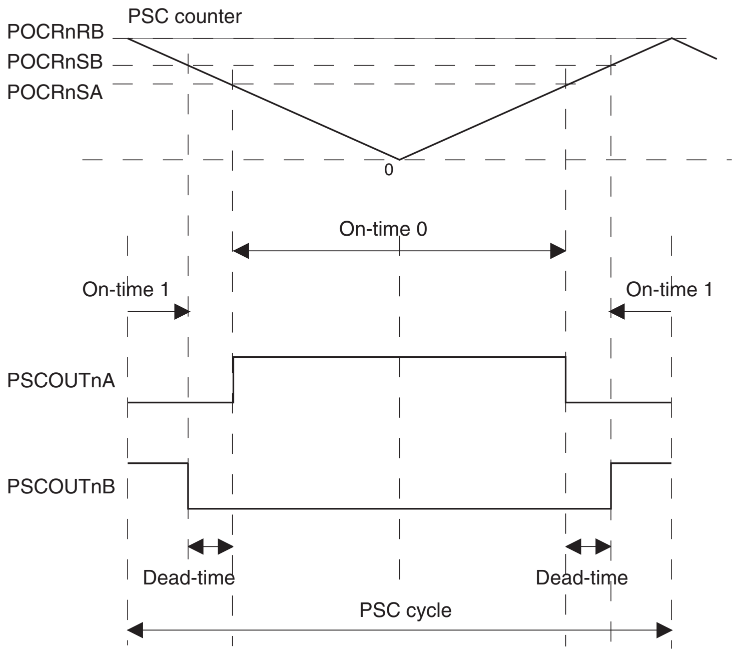


Figure 13.2: Cycle of Power Stage Controller [16]

The register `POCRnRB` adjusts the compare value at which the counter will start decrementing. Registers `POCRnSA` and `POCRnSB` adjust the compare value at which the outputs will be toggled. This gives the following formulas for computing the PSC duty cycle.

$$\text{Duty cycle } \texttt{PSCOUTnA} = \frac{2 \cdot \texttt{POCRnSA}}{2 \cdot (\texttt{POCRnRB} + 1)} \tag{13.1}$$

$$\text{Duty cycle } \texttt{PSCOUTnB} = \frac{\texttt{POCRnRB} - \texttt{POCRnSB} + 1)}{2 \cdot (\texttt{POCRnRB} + 1)} \tag{13.2}$$

$$\text{Cycle time } \texttt{PSCn} = \frac{2 \cdot (\texttt{POCRnRB} + 1)}{f_{pscclk}} \tag{13.3}$$

The datasheet of the H-bridge[17] has a specified maximal PWM frequency of $11KHz$. A value of $8KHz$ was chosen for the PWM frequency for the motors. This could conveniently be achieved using the CPU clock with no scaling and `POCRnRB` $= 1000$. Given the above equations, range of the values `PSCOUTnA` and `PSCOUTnB` will be $(-1000 \rightarrow +1000)$. Listing 13.2 shows the program code the the H-bridge library. The functions `motor_(A/B)_setspeed()` are used to set speeds for the motors. Functions to enable, disable, and initialize the motor controller circuitry has also been implemented.

Listing 13.2: Motor control library - hbridge_mc33932.c

```
// Set speed for motor A
void motor_A_setspeed(int16_t speed){
  if(speed>1000){
    speed = 1000;
  } else if(speed<-1000){
    speed = -1000;
  }
  POCR0SA = fmax(0,speed);
  POCR0SB = 1001+fmin(0,speed);
}

// Set speed for motor B
void motor_B_setspeed(int16_t speed){
  if(speed>1000){
    speed = 1000;
  } else if(speed<-1000){
    speed = -1000;
  }
  POCR2SA = fmax(0,-speed);
  POCR2SB = 1001+fmin(0,-speed);
}

// Initialize PSC with predefined settings
void motor_init(char en_a, char en_b){
  // Activate PSC0
  DDRB |= (1 << PORTB7); // Set PWM ports to output PSCOUT0B
  DDRD |= (1 << PORTD0); // Set PWM ports to output PSCOUT0A
  DDRB |= (1 << PORTB2); // Set ENABLE ports to output
  DDRC |= (1 << PORTC4); // Set DISABLE ports to output
  PORTC &= ~(1 << PORTC4); // Set DISABLE signal low

  // Activate PSC2
  DDRB |= (1 << PORTB0); // Set PWM ports to output PSCOUT2A
  DDRB |= (1 << PORTB1); // Set PWM ports to output PSCOUT2B
  DDRC |= (1 << PORTC7); // Set ENABLE ports to output
  DDRB |= (1 << PORTB4); // Set DISABLE ports to output
  PORTB &= ~(1 << PORTB4); // Set DISABLE signal low

  // Disable both outputs
  POC = 0b00000000;
  if(en_a){
    PORTB |= (1 << PORTB2); // Set ENABLE signal to H-bridge
    POC |= 0b00000011; // Enable PSC outputs 0A & 0B
  }
  if(en_b){
    PORTC |= (1 << PORTC7); // Set ENABLE signal to H-bridge
    POC |= 0b00110000; // Enable PSC outputs 2A & 2B
  }

  PSYNC = 0b00000000; // Set no sych. for ALL outputs
  POCR0RA = 0;
  POCR2RA = 0;
  POCR_RB = 1000;

  PCNF = 0b00011100;  // No update lock, select centered-mode, and
      outputs active high
  PMIC0 = 0b10100000; // Disable all inputs to PSC
  PMIC1 = 0b10100000;
  PMIC2 = 0b10100000;
  PCTL = 0b00000000; // CLK is FCPU, no scaling
}

// Enable motor outputs
void motor_enablePSC(void){
  PCTL |= 0b00000001; // Enable PSC
  motor_A_setspeed(0);
  motor_B_setspeed(0);
}

// Disable motor outputs
void motor_disablePSC(void){
  PCTL &= ~0b00000001; // Disable PSC
}
```

13.3.3 System Basis Chip Driver

As described in Section 13.3.1, identical libraries for SPI communication has been implemented for the Wheel Controller Units as for the Central Controller Unit. The usage of the SBC in the WCUs is also equal to that in the CCU. The SBC driver for the CCU, shown in Listing 12.7, could therefore be reused for the WCUs by only removing the printout to console.

13.3.4 Piher Encoder Driver

The angular sensors are one of the most important parts of the Wheel Controller Units. Although there were several difficulties with the hardware implementation of the sensors, their implementation in software was relatively easy. Figure 13.3 shows the frame for reading an angle over SPI. As an inverting transistor has been used in the logic circuit described in Section 7.2.4, the data bits sent on MOSI had to be be inverted. The angle of the sensor is transmitted in the third and fourth byte of the frame. The fifth and sixth bytes transmit the same data with bitwise inversion. It is, however, not necessary to transmit bytes 5 to 10 of the frame. The implemented driver therefore stops transmission after the angle data has been received, to reduce utilization of the SPI bus. The angle value is encoded as a 12-bit unsigned value, however, only the range from 10% to 90% is used. The function `encoder_read_value()` will therefore scale the value to the range of a 16-bit unsigned integer to simplify further use.

Listing 13.3: Function Prototypes - piher_encoder.h

```c
// Initialize SPI Channel for Using Piher Encoder.
void encoder_spi_init(void);
// Read Encoder Value by SPI. Value is scaled from 0-2^16
volatile int16_t encoder_read_value(uint8_t chan);
```

13.3.5 CAN Communication

The built in CAN controller of the ATmega64M1 is by far its most advanced hardware module. A library was therefore needed to use this controller

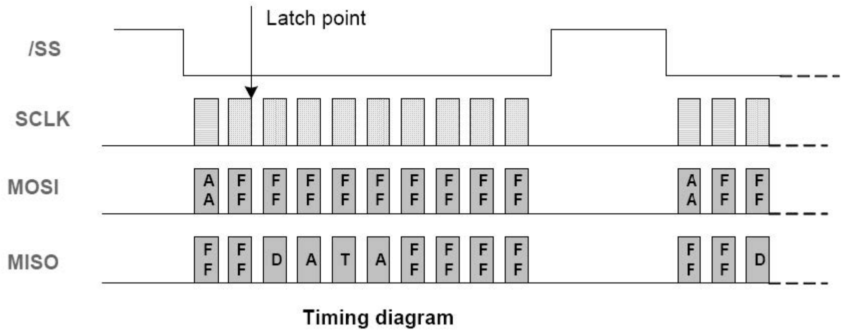


Figure 13.3: SPI frame for Piher MTS-360 encoder

efficiently. As will be described in Section 13.6, Atmel has published a CAN bootloader compatible with the ATmega64M1. The source code for the bootloader also includes a simple CAN driver, that has been adapted for several earlier projects in Revolve. To minimize the development time, this driver has been adopted for this project. A detailed description of this library will not be given, as it has not been developed for this thesis. A top level introduction will, however, be given to familiarize the reader with it's usage.

Atmel CAN Library

The Atmel CAN library has a similar structure to that provided by the ASF for the UC3C. A so called *command structure* is used to control the transmission or reception of a message. The `st_cmd_t` structure is shown in Listing 13.4. Different command types are defined in `can_cmd_t`, such as `CMD_TX_DATA` for sending a can message with data payload, and `CMD_RX_DATA_MASKED` for receiving a data CAN message with a defined identifier mask. The command structure also includes a pointer to a data array where the data content is stored.

The function `can_cmd (st_cmd_t *)` is used to execute the command defined in the structure. No interrupts have been implemented for this simple library.

Listing 13.4: Function Prototypes and Variables - can_lib.h

```
// Return values for can_cmd() function.
#define CAN_CMD_REFUSED   0xFF
#define CAN_CMD_ACCEPTED  0x00

// Return values for can_get_status function.
#define CAN_STATUS_COMPLETED     0x00
#define CAN_STATUS_NOT_COMPLETED 0x01
#define CAN_STATUS_ERROR         0x02

// This enumeration is used to select an action for a specific
    message
typedef enum {
  CMD_NONE,
  CMD_TX,
  CMD_TX_DATA,
  CMD_TX_REMOTE,
  CMD_RX,
  CMD_RX_DATA,
  CMD_RX_REMOTE,
  CMD_RX_MASKED,
  CMD_RX_DATA_MASKED,
  CMD_RX_REMOTE_MASKED,
  CMD_REPLY,
  CMD_REPLY_MASKED,
  CMD_ABORT
} can_cmd_t;

// This union defines a CAN identifier
typedef union{
  U32 ext;
  U16 std;
  U8  tab[4];
} can_id_t;

// This structure allows to define a specific action on CAN network
    .
typedef  struct{
  U8          handle;  // Handle to slot in MOB/Mailbox
  can_cmd_t   cmd;     // Command type
  can_id_t    id;      // ID of CAN message in pt_data
  U8          dlc;     // DLC of CAN message in pt_data
  U8*         pt_data; // Pointer to array of message data bytes
  U8          status;  // Message status
  can_ctrl_t  ctrl;    // Message control variables
} st_cmd_t;

//--- Function declarations

// Initialize can controller in given mode.
extern U8 can_init(U8 mode);
```

```

// Issue a command to the CAN controller. Returns state.
extern U8 can_cmd (st_cmd_t *);

// Get status for previous command from the CAN controller.
extern U8 can_get_status (st_cmd_t *);
```

WCU CAN Driver

To allow reception of the various CAN messages defined in Chapter 11, several command structures has been defined. These are shown in Listing 13.5. As we can see, the required number of message handlers has been drastically reduced from the CAN driver of the Central Controller Unit. A function for simply initializing the CAN interface has also been implemented in this library.

Listing 13.5: Function Prototypes and Variables - wcu_can.h

```
//--- VARIABLE DEFINITIONS
U8  cmd_msg_rx_data[8];
U8  ccustate_msg_rx_data[8];
U8  wcu_msg_data[8];

st_cmd_t cmd_msg_rx;
st_cmd_t ccustate_msg_rx;
st_cmd_t wcu_msg_tx;

//--- FUNCTION PROTOTYPES
// Initialize CAN controller with predefined settings
void wcu_can_init(uint8_t thisunit);
```

13.3.6 Timer Driver

To achieve accurate periodic execution of the controller algorithm, a hardware timer may be used. The ATmega64M1 has features a 16-bit hardware timer that has been used for this application. The timer has been run in a so called CTC mode – Clear Timer on Compare. This means that the counter is reset every time the timer reaches its compare value. A simple library has been implemented to control the timer. Function prototypes are shown in Listing 13.6.

Listing 13.6: Function Prototypes and Variables - wcu_timer.h

```c
void WCUTimer_Enable(void); // Enable timer operation
void WCUTimer_Disable(void); // Disable timer operation
// Set given interval(ms) for timer
void WCUTimer_SetInterval(uint16_t interval);
```

13.4 WCU Application

13.4.1 State Manager

A library has been implemented to manage the state of each Wheel Controller Unit. The library includes functions to set and read the state if the WCU. The states of the WCU has been defined according to Table 11.5. The library will also handle enabling and disabling of PWM signals to the DC-motors, depending on the chosen mode. Program code for the library is shown in Listing 13.7.

Listing 13.7: Function Prototypes and Variables - statemanager.c

```c
// Local variable to store current state
static int8_t unitState = -1;

// Set new state for unit
void updateUnitState(int8_t newState){
  if(newState != unitState){
    pid_Reset_Integrator(&pidData[0]);
    pid_Reset_Integrator(&pidData[1]);
    // Update with new state. If system is in ERROR, only allow
        change to DISABLED
    if ( (unitState != CONTROLMODE_ERROR) || (unitState ==
        CONTROLMODE_ERROR && newState == CONTROLMODE_STANDBY) ){
      unitState =  newState;
    }
    //System is changing to a normal state
    //(DISABLED enables PSC temporarily: will be disabled in MAIN
        LOOP once position is reached)
    if( (unitState==CONTROLMODE_STANDBY) || (unitState==
        CONTROLMODE_MANUAL) || (unitState==CONTROLMODE_OPERATIVE) )
        {
      motor_enablePSC();
    }
    // If the system is in ERROR, disable motors immediately
    else if(unitState == CONTROLMODE_ERROR){
      motor_disablePSC();
    }
  }
}

// Read current unit state
int8_t getUnitState(void){
  return unitState;
}
```

13.4.2 PID Controller

The PID controllers are arguably the most important feature of the Wheel Controller Software. This section will cover how these PID controllers have been implemented in software, as well as any supporting libraries.

The implementation of the PID controller itself has been done using a library published by Atmel, for their 8-bit microcontrollers[24]. This has been done to reduce development time for the system. Some modifications to the code has, however, been necessary. As this library has not been developed for this thesis, the focus of this section will lie on the modifications done to the library, and the challenges of implementing a PID controller on a 8-bit architecture.

Atmel PID Library

One of the main challenges of implementing a PID controller on the AVR architecture is its limitation in variable resolution. Floating point performance on 8-bit microcontrollers is very limited, and the implementation must consequently be limited to integers. The Atmel PID library is implemented using mainly 16-bit singed integer values. This was deemed to be sufficient angular resolution for the WCU.

Listing 13.8 shows declaration of necessary structures and function prototypes for the Atmel PID library. The structure `pidData_t` is used to store all parameters for one PID controller. This includes tuning constants as well as previous system states for computation of integral and derivative terms.

The operation of the controller is managed by the use of only three functions. The function `pid_Init()` is used to initialize the values of the `pidData_t` structure for the applicable controller. New output variables are generated by running the PID loop function `pid_Controller()`. The function `pid_Reset_Integrator()` may be used to zero the error sum, if an integral term is used.

The damper valves have proven themselves to require more force to be turned in the closing direction, than in opening direction. This made tuning of the standard Atmel PID controller difficult. A logic case was therefore implemented to allow different proportional gains to be applied for positive and negative errors.

Listing 13.8: Function Prototypes and Variables - pid.h

```c
//--- CONSTANT DEFINTIONS
// Scaling factor for PID variables
#define SCALING_FACTOR  10

// Treshold to neglige small output values
#define CONTROLTRESHOLD_DISABLE 0xF0

// Maximum value of variables
#define MAX_INT         INT16_MAX
#define MAX_LONG        INT32_MAX
#define MAX_I_TERM      (MAX_LONG / 2)

//--- VARIABLE DEFINITIONS
// Setpoints and data used by the PID control algorithm
typedef struct{
  int16_t lastProcessValue; //! Last process value(for D-term)
  int32_t sumError;     // Summation of errors(for I-term)
  int16_t P_Factor_Pos;   // The Proportional tuning constant(/
      SCALING_FACTOR)
  int16_t P_Factor_Neg;   // The Proportional tuning constant(/
      SCALING_FACTOR)
  int16_t I_Factor;     // The Integral tuning constant(/
      SCALING_FACTOR)
  int16_t D_Factor;     // The Derivative tuning constant(/
      SCALING_FACTOR)
  int16_t maxError;     // Maximum allowed error (overflow
      protection)
  int32_t maxSumError;    // Maximum allowed sumError
} pidData_t;

//--- FUNCTION PROTOTYPES
void pid_Init(int16_t p_factor_pos, int16_t p_factor_neg, int16_t
    i_factor, int16_t d_factor, pidData_t *pid);
```

```
int16_t pid_Controller(int16_t setPoint, int16_t processValue,
    pidData_t *pid_st);
void pid_Reset_Integrator(pidData_t *pid_st);
```

WCU PID Library

The WCU PID library has been implemented to define the necessary data structures and abstractive functions for usage of the Atmel PID library.

Listing 13.9 shows the header file for this library. As can be seen, this file defines the tuning parameters that are applied when the PID controller is initialized. Furthermore, the array `pidDefaultPosition` is used to define the static position that unit will assume when it is in standby mode. Finally, a function `void PIDController_Init()` has been implemented to initialize both PID controllers with the defined settings.

Listing 13.9: Function Prototypes and Variables - pid_controller.h

```
//--- CONSTANT DEFINITIONS
// P, I and D parameter values (Different K_P for positive and
    negative error)
// Right controller parameter values
#define R_K_P_POS     8
#define R_K_P_NEG     12
#define R_K_I         0 // Only P-Controller
#define R_K_D         0
// Right controller parameter values
#define L_K_P_POS     8
#define L_K_P_NEG     12
#define L_K_I         0 // Only P-Controller
#define L_K_D         0

//--- VARIABLE DEFINITIONS
static const uint16_t pidDefaultPosition[4][2] = { {21417, 19179},
    {21417, 19179}, {21417, 19179}, {21417, 19179} };

// Struct for controller variables
typedef struct{
  int16_t   referenceValue;
  int16_t   measurementValue;
  int16_t   controlInput;
} pidControlVariables_t;

// Variables for WCU PID controllers
extern pidControlVariables_t controlVariables[2];
```

```c

// Parameters for regulator
extern pidData_t pidData[2];

// Global flags to indicate PID timeout
extern char timerflag;
extern uint8_t controllerloop_cnt;

//--- FUNCTION PROTOTYPES
void PIDController_Init(void);
int16_t PIDController_GetPosition(uint8_t chan);
int16_t PIDController_ValveGotoMax(uint8_t chan);
void PIDController_InitAngle(uint8_t chan);
```

Angle Calculation

As described in Section 3.1.1 the valves of the Öhlins TTX25 damper require 4.5 turns rotation to be adjusted from fully closed, to fully open. The Piher MTS-360 angular sensor is only absolute within one rotation. A mechanism to calibrate and calculate the total angle has therefore been implemented in the WCU PID library.

Listing 13.10 shows the function to read the angle of the valve. As can be seen, the `encoder_read_value()` function from the encoder driver is used to get the angular value from the sensor. To keep track of the actual angle of the valve, a *turn counter* has been implemented. If the angular value of the sensor wraps around between two samples, a turn detected. This is done using the principle shown in Figure 13.4. If a transition is detected from the red to the green area, or vice versa, the turn counter is respectively incremented or decremented.

The expression evaluated for calculating the position of the valve is at the bottom of Listing 13.10. As the the angular sensor driver uses the range of a unsigned 16-bit integer, and the PID controller uses singed 16-bit integers, the value must be scaled. For simplicity, the value range of the PID controller is scaled to 5 rounds in both positive and negative direction.

Listing 13.10: Reading of valve position - pid_controller.c

```c
int16_t PIDController_GetPosition(uint8_t chan)
{
```

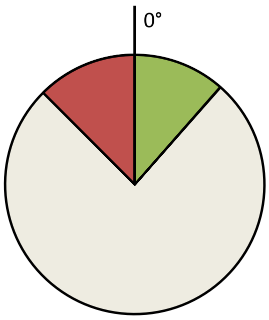


Figure 13.4: Wraparound areas for valve turn counter

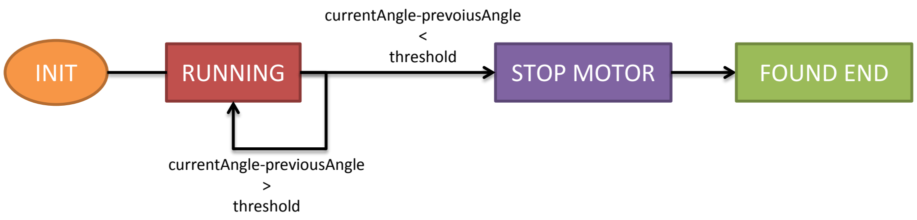


Figure 13.5: Calibration routine for Wheel Controller Unit valves

```
    // Read encoder value with calibration parameters
    uint16_t temp =  2*(uint16_t)encoder_read_value(chan) - 2*(
        uint16_t)angleData[chan].calibrationAngle;

    //...OMITTED LOGIC FOR VALVE COUNTER...

    return (UINT16_MAX/10)*angleData[chan].turnCounter + (temp/10);
        // Divide by two, and scale for 5 rounds
}
```

Every time power is applied to the WCU, the position of the valve will be calibrated. The calibration routine for function `PIDController_ValveGotoMax()` is shown in Figure 13.5. The calibration is done by running the motor until the valve is fully closed. The end position is determined by monitoring the change in angle between each sample. If little change is detected, an end stop is indicated.

Once the calibration is finished, the function `PIDController_InitAngle()`

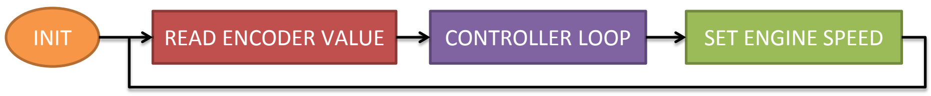


Figure 13.6: Execution sequence for valve controller

may be called to zero the turn counter, and register a turn offset.

13.5 Big-While Loop

As described in Section 13.2.1, a *big-while* architecture has been adopted for the Wheel Controller Unit. Figure 13.6 shows the execution sequence for the valve controller. The controller sequence is executed periodically, using a hardware timer, as defined in Section 13.3.6.

A heavily reduced version of the main execution loop is shown in Listing 13.11. The structure of execution has, however, been withheld to shown the overall principle of this function. The big-while loop can be studied in it's in its entirety in Appendix C.2.1.

Listing 13.11: Heavily reduced big-while loop - wheel_controller_main.c

```
// Timer interrupt to signal the execution interval
ISR(TIMER1_COMPA_vect){
  timerflag = 1;
  TIFR1 = (1 << OCF1A); // clear the CTC flag
}

//...

int main(void){
  // System is initialized
  //...
  while(1){
    //Check for CAN messages
    //...
    if(timerflag){

      //--- RESET TIMER FLAG ---
      timerflag = 0;

      //--- GET VALVE POSITION ---
```

```
        controlVariables[0].measurementValue =
            PIDController_GetPosition(ENC1);
        controlVariables[1].measurementValue =
            PIDController_GetPosition(ENC2);

        //Actions of other states omitted
        //...
        if( (getUnitState() == CONTROLMODE_MANUAL) || (getUnitState()
             == CONTROLMODE_OPERATIVE) ){
          //--- RUN PID LOOP ---
          controlVariables[0].controlInput =
          pid_Controller(controlVariables[0].referenceValue,
              controlVariables[0].measurementValue, &pidData[0]);
          controlVariables[1].controlInput =
          pid_Controller(controlVariables[1].referenceValue,
              controlVariables[1].measurementValue, &pidData[1]);
        }

        //Broadcast unit state message
        //...
      }

      //--- CHECK IF PWM PERIOD IS FINISHED
      if ( (PIFR&(1 << PEOP))==1 ){
        PIFR |= (1 << PEOP);

        //--- SET NEW MOTOR SPEED ---
        motor_A_setspeed(controlVariables[0].controlInput/33);
        motor_B_setspeed(controlVariables[1].controlInput/33);
      }
    }
}
```

13.6 CAN Bootloader

Section 13.1.5 defined a requirement that a CAN bootloader should be deployed on the Wheel Controller Units. Atmel has published a finished solution in [23] that may be used for this purpose. This bootloader has been customized for use in several previous development in Revolve, and could therefore be deployed with only minor modifications to the WCUs. This section will give an introduction to the Atmel CAN Bootloader and present how it has been deployed on this system.

13.6.1 Atmel CAN Bootloader

The Atmel CAN Bootloader allows the program of 8-bit AVR microcontrollers to be reprogrammed using CAN bus. This is particularly useful for units that are mounted in areas with poor accessibility. All units connected to the CAN bus may be programmed individually. The Atmel CAN Bootloader imitates the ISP protocol normally used by regular programming adapters.

Once device is powered up, the processor starts execution of the bootloader. The bootloader will wait for a defined time for a CAN command to indicate the start of a flash session. If no flash session is initiated the CPU will jump to execution of the stored application.

13.6.2 System Basis Chip Flash Mode

The watchdog of the System Basis Chip used in this unit must be reset every 256ms during execution. This may not be possible during programming and debugging of the unit. To get around this problem, the SBC may be put into a so called *flash mode*. This extends the period of the watchdog up to 30 seconds.

As no previous devices had required this functionality from the bootloader, it had to be implemented for the WCUs. The selection of flash mode is done with an SPI command, such as the remaining configuration of the System Basis Chip. The driver implemented for configuration of the CCU and WCU applications,as shown in 12.3.3, could therefore be reused with minor modifications.

13.6.3 Atmel FLIP

Obviously, some computer software is required to communicate with the bootloader. This assignment has used the FLIP tool that is available freely from Atmel. FLIP allows tunnelling of the ISP programming protocol through both CAN, USB and RS-232. To connect the computer to the CAN bus an interface, typically a USB to CAN interface, is needed. FLIP has implemented support for several of the most popular interfaces on the market. For this application a PCAN-USB adapter from PEAK-System has been used. A picture of the adapter is Shown in Figure 6.17. More details on how the bootloader is configured and used is available in [23].

14 Discussion on the Software Design

"To err is human - and to blame it on a computer is even more so."

- Robert Orben

This part has described the implementation of software for the CES system. The primary goal for this assignment has been to implement a system with baseline functionality. It has been important to create a software framework for the system that may be used for further development of software on both units.

14.1 Communication Protocol

Chapter 11 presented the design of a communication protocol that could be used for the system. The requirements that were identified for the communication protocol has to a large extent been addressed by the current implementation. Furthermore, the communication protocol that has been implemented has proven itself to work as intended during testing.

The common addressing protocol that was developed for the system has undoubtedly simplified the integration of all the systems on the car. Communication between the various systems has been possible without difficulty as all units has used the same coding of addresses.

14.2 Central Controller Unit

14.2.1 Evaluation of the Software Implementation

Chapter 12 gave a presentation of the design and implementation of software on the CCU. This unit has, by far, required the most effort for software development in the system.

Given the requirements that were identified for this unit it was decided to use a operating system with support for multiple tasks. While a simpler and more basic implementation could have been possible, it has been a goal to use a software design that may easily support additions. The current program structure also allows the user to modify the controller algorithm without in-depth knowledge of the remaining system.

Because of the limited time frame for this assignment, functions to implement baseline functionality and testing of these has been prioritized. Convenience functions such as memory card for storage of log files and bootloading of software has not been addressed.

Ready-made libraries from the Atmel Software Framework was used extensively throughout the development. This eliminated the need for implementation of most drivers and abstraction libraries. The documentation for the ASF libraries are, however, in many cases very limited, and a lot of time has been lost in the process of trying to understand how libraries should be used.

As mentioned in the hardware discussion, development using the SBC has also been somewhat cumbersome. In early development stages the watchdog had to be disabled, as no driver was yet available. The watchdog may only be disabled using a hardware connection, which require 8-10V. Testing with the power supply of the car was therefore not possible with this connection.

14.2.2 Further work

There are several issues that may be addressed for future development of the CCU software. A list follows below.

System Basis Chip

As described in Section 9.1.2, the SBC does add unnecessary complexity of the system. It should therefore be considered replaced in future hardware revisions. The UC3C has implemented an internal watchdog timer that may be used as a substitute.

SD Card

Because of a limited time frame, no software has been implemented for the SD card. Future developers should consider implementing support for storing configuration parameters, system logs and loading programs from the SD card.

CAN Bootloader

Bootloading from CAN bus has proven to be a very valuable tool, during development of the Wheel Controller Units. A similar bootloader should be considered for implementation for the CCU, as an alternative to bootloading from the SD card.

14.3 Wheel Controller Unit

14.3.1 Evaluation of the Software Implementation

In Chapter 13 the implementation of software for the Wheel Controller Units were presented. The software implemented for these units is significantly simpler, compared to the CCU, and a traditional big-while software architecture was therefore sufficient for this unit.

Libraries to implement CAN communication and PID control were available from Atmel. Both libraries has been used with success, and were found to be easy to use. As the valve of the dampers required more force to be closed, rather than opened, modifications were done to the standard PID controller. This allowed different proportional constants to be used for negative and positive errors. Due to the friction in the valves, a rather large duty cycle was required for the motors to turn the valve at all. A large proportional constant therefore had to be used. With such a large P-constant, the controller was found to work satisfactory as a pure proportional controller. However, some stationary deviation have had to be tolerated.

Access to program the units on the car was rather poor. The usage of a bootloader has therefore been invaluable during the late development phases.

14.3.2 Further Work

There are several issues that may be addressed for future development of the WCU software. A list is given below.

System Basis Chip

As for the Central Controller Unit, development with the SBC has been rather cumbersome. Refer to Section 14.2.2 for more information. The ATmega64M1 also features an internal watchdog timer that may be used as a substitute.

PID Controller

Due to the limited time frame for this assignment, only basic tuning of the PID controller done. As presented in the results, the current response of the controller is acceptable. However, further tuning of the PID controller should be able to increase the performance of the system.

Part IV

Test Results, Discussion and Conclusion

15 Test Results

"The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency."

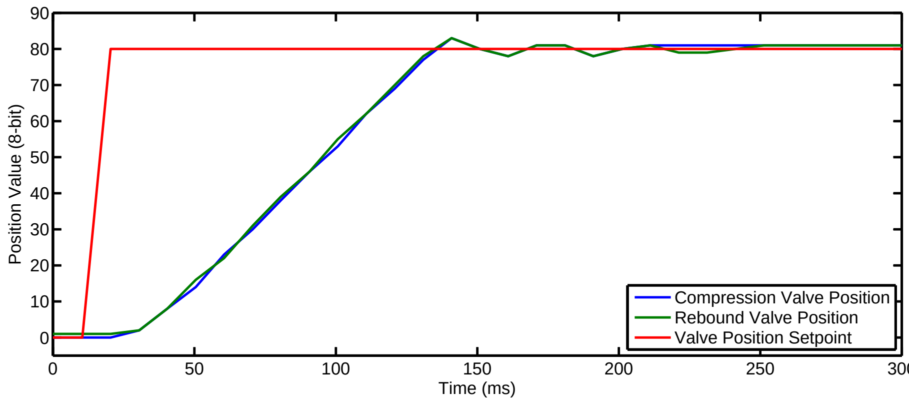
- Bill gates

This section will present procedures and tests that have been conducted to evaluate the performance of the system. Because of the limited time frame for the assignment, there has been little time available for testing. An elementary test of the step response of the Wheel Controller Units is nevertheless presented below.

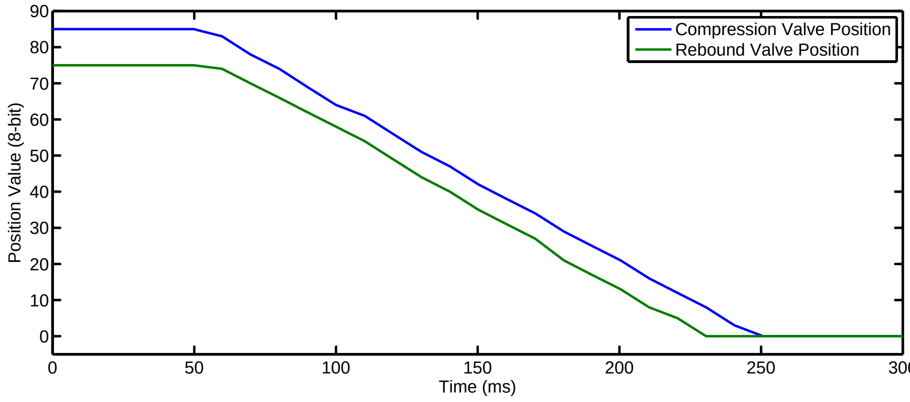
15.1 Unit Tests

15.1.1 WCU Setpoint Response Test

To study the response time of an assembled Wheel Controller Unit, a conventional step response test has been performed. A PEAK System PCAN-USB interface was used to imitate the Unit State and Setpoint Control Messages from the CCU. The Wheel Controller Unit was set in the Operative state while the test was performed. The setpoint values for rebound and compression were kept constant, before a step response was applied. Figures 15.1a and 15.1b shows plots of the step response, for respectively an opening and closing valve.



(a) Opening valve



(b) Closing valve

Figure 15.1: Step responses of WCU PID controller

As can be seen from the plots, both responses show little oscillations, with the opening valve beeing slightly underdamped. The position values were scaled to the range of an 8-bit signed integer. The range of the PID controller set to 5 turns in both positive and negative direction for the WCU. The step thus represented a change of 5 Turns $* \frac{80}{128} = 3.125$ Turns.

Opening valve speed	$11.2°/ms$
Closing valve speed	$5.45°/ms$

Table 15.1: Measured angular speeds for valve adjustment

It is clearly visible from the figures that the response of the opening valve is significantly faster than for the closing valve. Furthermore, the acceleration time can be seen as negligible compared to the total adjustment time. The motors reach a stationary speed in both directions. Stationary angular speeds, calculated from measurements on the plots, are shown in Table 15.1.

15.2 Evaluation of Test Results

The time frame of the assignment has unfortunately not allowed a lot of testing to be done. The tests that have been conducted has, however, shown very positive results so far. The WCUs have been tested in assembly with the dampers. As was presented in Section 16 the current proportional controller implemented for the damper has allowed efficient position control of the valves. The measurements from the WCU setpoint response test showed that a slightly under-damped response was achieved. An angular speed of between $5.45°/ms$ and $11.2°/ms$ was on par with what could be expected from the motor, given the following nominal motor speed of

$$1522rpm = 0.254r/ms = 9.132°/ms$$

16 Overall Discussion

"I don't aspire to be like other drivers - I aspire to be unique in my own way."

- Lewis Hamilton, Formula One Driver

16.1 Evaluation of Solutions

Throughout this project, a distributed embedded system for control of an adaptive suspension system has been designed. The previous prestudy by the same author had given recommendations for a top level architecture for the system. The proposed architecture was evaluated to be suitable for further development.

Perhaps the most important design choice carried over from the prestudy was the decision to implement a distributed system. The division of the system into a Central Controller Unit and Wheel Controller Units has allowed the functionality of the system to be placed throughout the car – where it is most appropriate. By implementing PID controllers locally at each wheel, a stable and robust solution has been implemented. It has also been possible to interconnect the system using a single bus cable, that could be shared with other systems.

A complete set of prototype hardware has been implemented. The prestudy had already prepared several suggestions for components that could be used for the hardware design. These were to a large degree found to be appropriate, and used for further development. There was, inevitably, several modifications necessary to the initial prototypes for them to work as intended. An updated hardware design was therefore implemented with modifications that were possible within the time frame of the project. This design was assembled at SimPro at Løkken Verk, giving a very professional result. While the quality of the PCBs made locally at the university were of somewhat varying quality, there has been no fault on the units that were assembled at SimPro.

The final prototypes of the Wheel Controller Units were able to meet the external size requirements that applied to the unit. It did, however, require a lot of work to get a satisfactory hardware design within these measurements. This difficulty has to a large degree come as a consequence of the current drawn by the DC motors. This posed additional requirements for the design, to separate signal and power parts of the circuit. The final design of the Central Controller Unit was also within the expectations to size. An enclosure has been purpose designed with help from other team members of Revolve, and manufactured using 3D printing technology at the Department of Engineering Cybernetics.

Because of the limited time frame for this assignment it has been necessary to limit the software implementation to a baseline system. Functions that have been essential for the system to be working as a whole has been prioritized, while several more advanced features have been left to future developers. For the Central Controller Unit this implied that features such as the SD card interface has not been tested. Requirements to the Central Controller Unit yielded a decision to use an embedded operating system to achieve the desired system performance. The program was divided into several tasks that may be executed concurrently.

To shorten the development time it was chosen to use drivers and libraries from the Atmel Software Framework when possible. Because of lack in documentation the provided software did, however, require a lot of work before intended functionality could be achieved.

For the CCU it has also been of particular interest to develop a framework that allowed further implementation of the main controller algorithm to be done by developers without extensive knowledge of the remaining software. The division of the program into several tasks has made it possible to develop this functionality separately from the rest of the system.

As the Wheel Controller Units required significantly simpler software than the CCU, a traditional big-while architecture was chosen for these units. Several drivers had to be implemented to create the necessary level of abstraction from the hardware components. A library provided by Atmel has been used to implement PID controllers for the wheel controller units. Some modifications were, however, required to achieve the desired level of performance.

The restricted access to the dampers was also much helped by the implementation of a bootloader, that allowed the units to be programmed through CAN bus.

16.2 Further Work

The implemented system has provided Revolve with a prototype system that may be used for evaluation of on-car performance and concept. As was discussed in Section 9 there are no critical need for further improvements of the hardware design, nevertheless there are several details where the system could benefit from improvement.

The software implementation of this assignment has, as previously indicated, focused on implementing a baseline functionality for the whole system. Due to a limited time frame, the performance of the system has, however, not been studied sufficiently. Further testing of the implemented system to reveal any potential weaknesses should be assigned top priority for further development.

There are also several convenient functions such as saving of log data, bootsloading and unit configuration via the SD card could definitively simplify the operation of the system. Further enhancements of the system's performance should also be simple, given the execution framework and communication protocol available for both units.

17 Conclusion

"Experience: that most brutal of teachers. But you learn, my God do you learn."

- C.S. Lewis

This assignment has included the design, implementation and testing of an embedded system for electronically adjustable suspension system on a Formula Student racing car. The system has been implemented so that it may be deployed for testing on Revolve NTNU's 2013 car – the KA Aquilo R.

The project started by evaluating the top level system architecture and corresponding requirements, that had previously been during the prestudy for the project. A distributed architecture was found to be appropriate for further development. The decision to use a distributed architecture has allowed the implementation of a robust solution, with error detection. It has also been possible to interconnect the system using a single bus cable, that could be shared with other systems, thus saving weight.

A complete set of prototype hardware has been implemented. The final prototype designs were produced and assembled at electronics producer SimPro. These units have proven to be stable hardware prototypes, that could be used for software development, and installed on the car for evaluation of performance. Possible improvements in the hardware design has been noted, for further development.

Figure 17.1: The 2013 KA Aquilo R

A working software implementation has been implemented for all units, as well as a messaging protocol that allows all systems of the car to be communicate.

Requirements to the Central Controller Unit yielded a decision to use an embedded operating system, to achieve the desired system performance. The program was divided into several tasks that may be executed concurrently. The implemented design has also been designed to accommodate a more advanced control algorithm for future development. Overall, the program design of the CCU has given a system that allows work on the control algorithm to be continued, without extensive knowledge of the remaining software design.

The requirements to the software of the WCU did not justify the implementation of an operating system. A traditional big-while architecture was therefore adopted for the software design of this unit. This allowed a simple software structure, with stable execution pattern. The performance of this software has shown itself to allow efficient position control of the damper valves.

Some elementary testing of the modules has also been done, to verify the functionality of the system. Although the time frame has limited the amount of available time for testing, the current results has given promising indications. Furthermore, experience with the system during development has indicated a stable system.

Overall, the implemented system has provided Revolve with a prototype system, that may be used for evaluation of on-car performance and concept. Further enhancements of the system's performance should also be simple, given the execution framework and communication protocol available for both units.

Based on these findings it is therefore recommended that the work is continued by next years team members, by implementing the suggestions given for further development.

Bibliography

[1] Håkon Devold. “Design of an Embedded Control System for Electronically Adjustable Suspension in a Formula Student Racing Car”. Norwegian University of Science & Technology, 2012.

[2] R. Bosch. *CAN specification version 2.0.* Sept. 1991.

[3] Formula SAE. *2013 Formula SAE Rules.* 2012.

[4] Junli Wu, Xiaoming Chen, and Huijun Gao. “H filtering with stochastic sampling”. In: *Signal Processing* 90 (4 2010), 1131–1145.

[5] S.M. Savaresi, C. Poussot-Vassal, C. Spelta, et al. *Semi-Active Suspension Control Design for Vehicles.* Elsevier Science, 2010.

[6] Maxon Motor. *Maxon product catalogue 2012.* 2012.

[7] Microchip Technology Inc. *AN228 - A CAN Physical Layer Discussion.* 2002.

[8] Alpha Wire. *Master Catalog.* 2011.

[9] Wikipedia. *Secure Digital.* 2013. URL: `http://en.wikipedia.org/wiki/Secure_Digital`.

[10] Oyvind Netland and Amund Skavhaug. “Adaption of MathWorks Real-Time Workshop for an Unsupported Embedded Platform”. In: *36th EUROMICRO Conference on Software Engineering and Advanced Applications(SEAA).* 2010.

[11] Atmel Corporation. *32-bit AVR Microcontroller AT32UC3C Datasheet.* 2012.

[12] Texas Instruments. *Datasheet TLV1117.* 2013.

[13] Analog Devices. *Datasheet ADM3051.* 2011.

[14] Freescale Semiconductor. *Technical Datasheet, SBC Gen2 with CAN High Speed and LIN Interface, MC33903_4_5.* 2011.

[15] *Gridconnect: PCAN-USB.* 2013. URL: `http://gridconnect.com/can-usb.html`.

[16] Atmel Corporation. *8-bit Atmel megaAVR Microcontroller ATmega64M1 Datasheet.* 2012.

[17] Freescale Semiconductor. *Technical Data, 5.0A Throttle Control H-Bridge, MC33932.* 2012.

[18] Piher Sensors & Controls S.A. *MTS-360, Through shaft contactless sensor.* 2012.

[19] *Wikimedia Commons.* 2012. URL: `http://commons.wikimedia.org`.

[20] e2v semiconductors. *Design Considerations for Mixed-Signal*. 2009.
[21] *Prototype SMT Stencil*. 2013. URL: `http://pcbrework.wordpress.com/2012/08/11/prototype-smt-stencil/`.
[22] Atmel Corporation. *Atmel AVR4030: Atmel Software Framework - Reference Manual*. 2012.
[23] Atmel Corporation. *AVR076: AVR CAN - 4K Boot Loader*. 2009.
[24] Atmel Corporation. *AVR221: Discrete PID controller*. 2006.

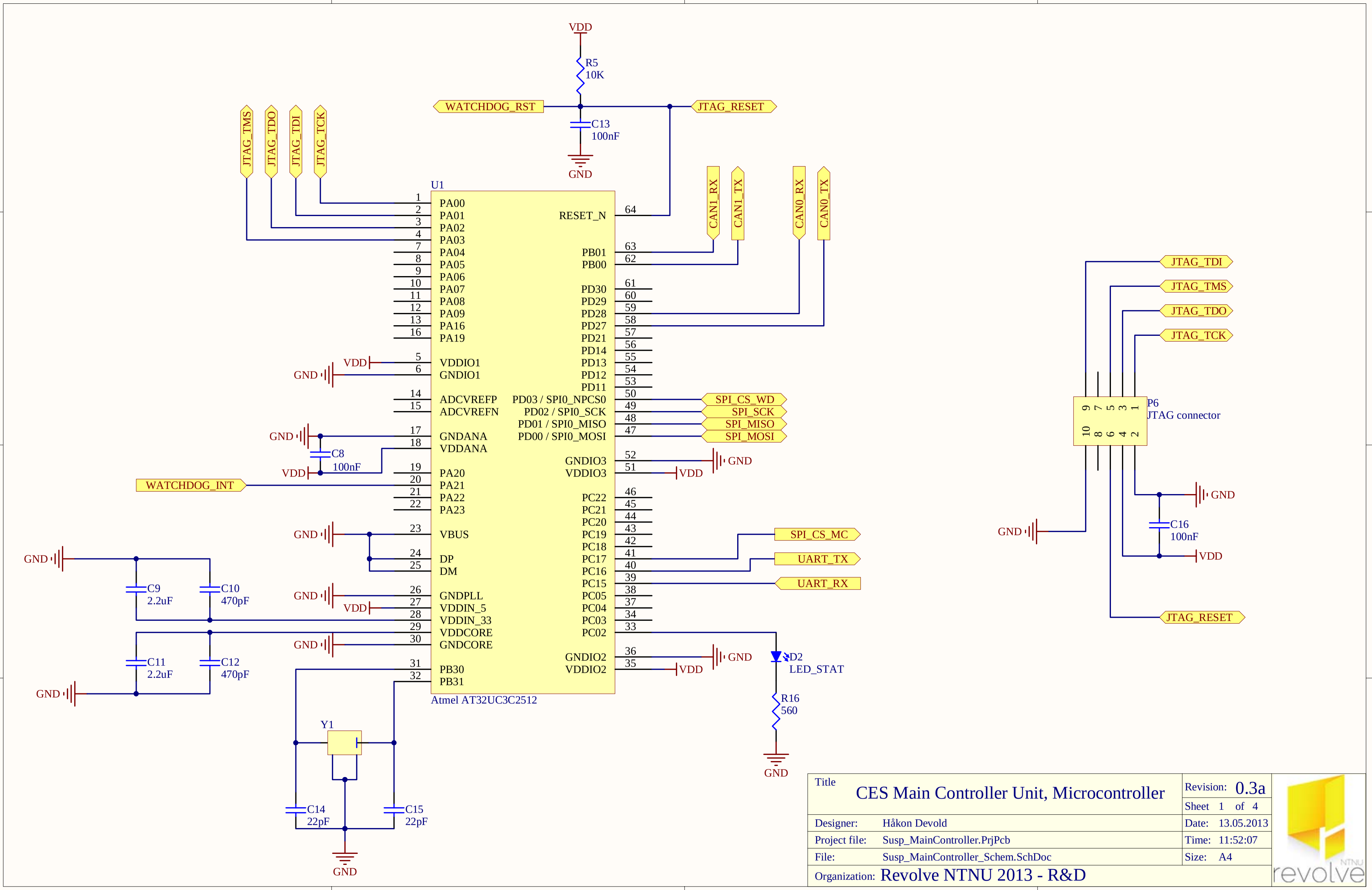
Part V

Appendices

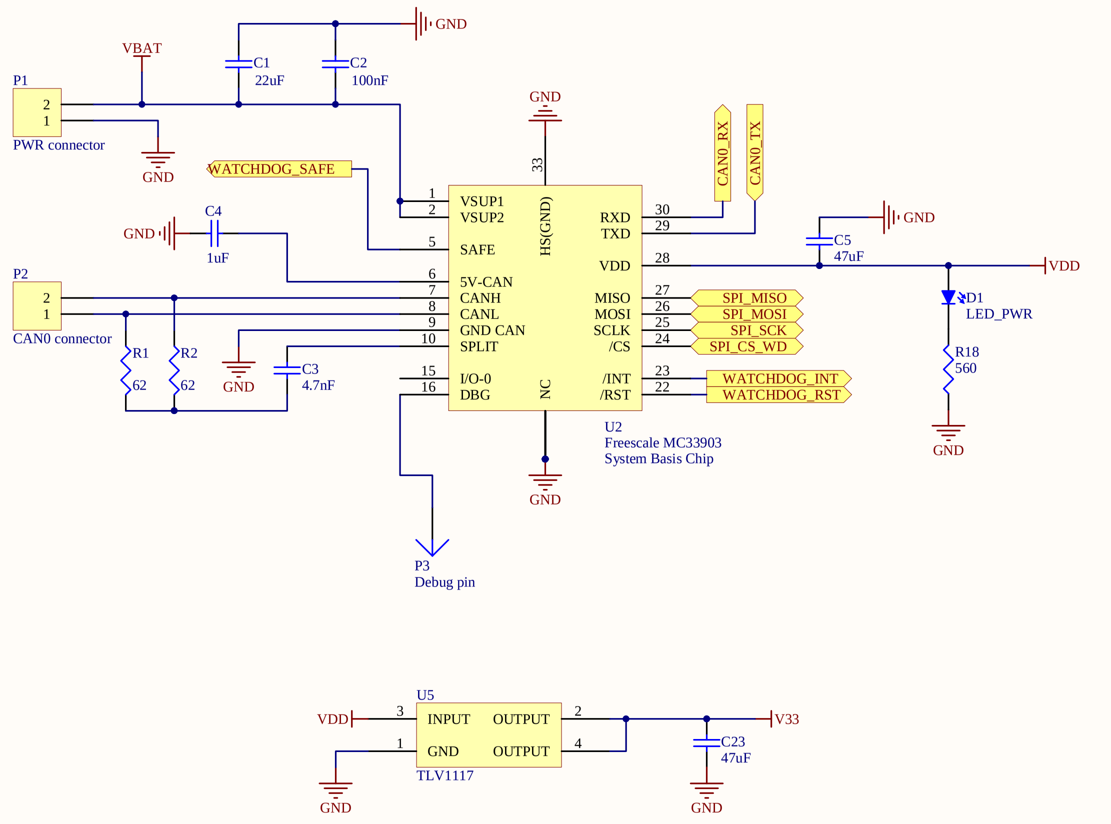
A
Schematics

A.1 Central Controller Unit Schematics

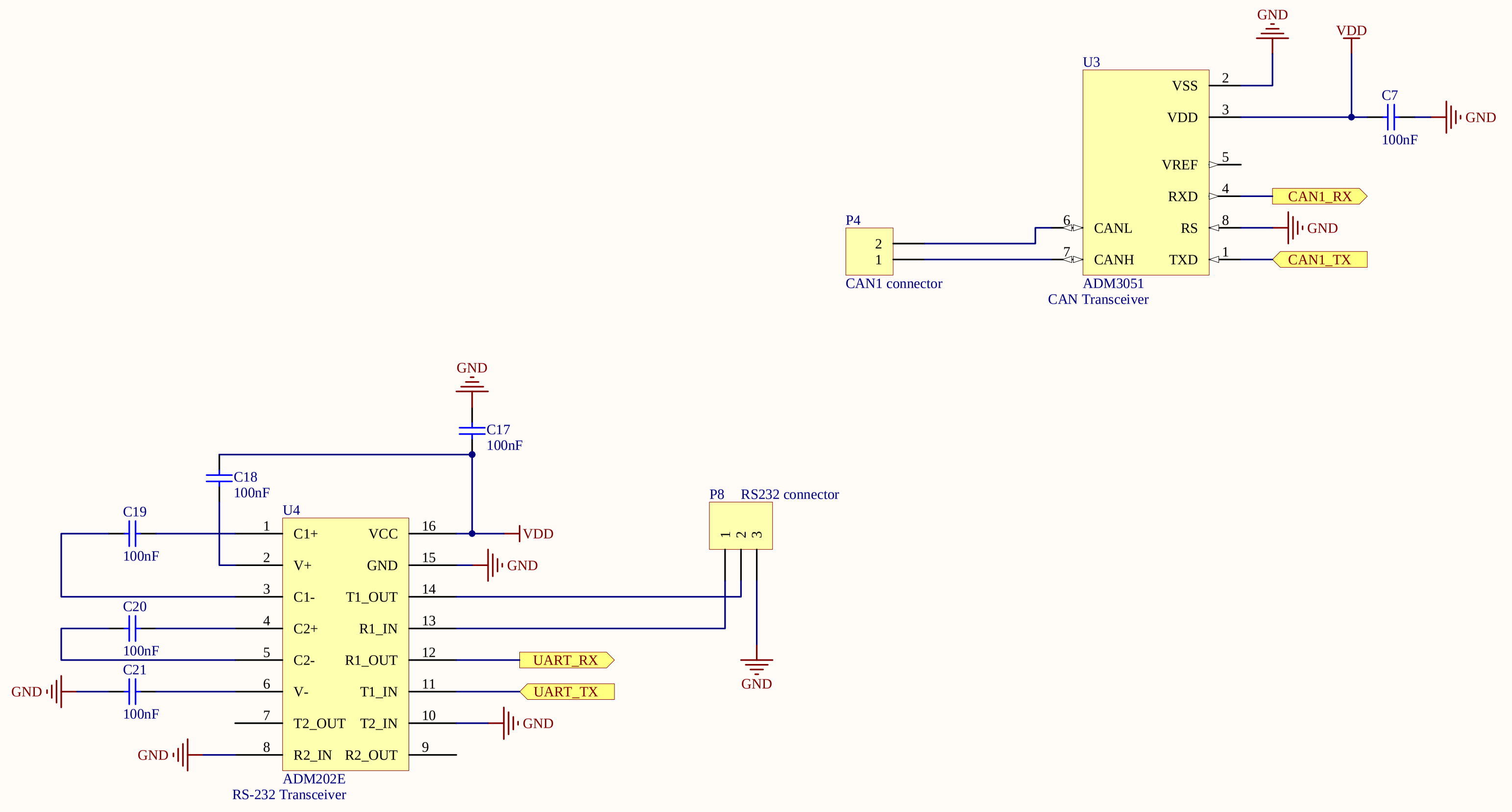
1. Central Controller Unit - Microcontroller
2. Central Controller Unit - Power Supply
3. Central Controller Unit - Peripherals
4. Central Controller Unit - SD Card Connector



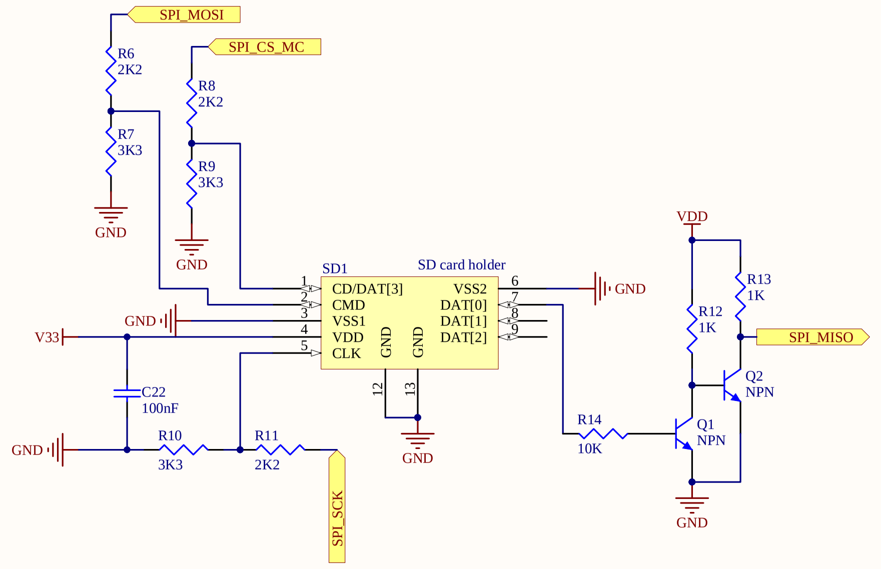
Title	CES Main Controller Unit, Microcontroller	Revision:	0.3a
		Sheet 1 of 4	
Designer:	Håkon Devold	Date:	13.05.2013
Project file:	Susp_MainController.PrjPcb	Time:	11:52:07
File:	Susp_MainController_Schem.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		



Title	CES Main Controller Unit, Power Supply	Revision:	0.3a
		Sheet 2 of 4	
Designer:	Håkon Devold	Date:	13.05.2013
Project file:	Susp_MainController.PrjPcb	Time:	11:51:20
File:	Susp_MainController_PSU_Schem.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		



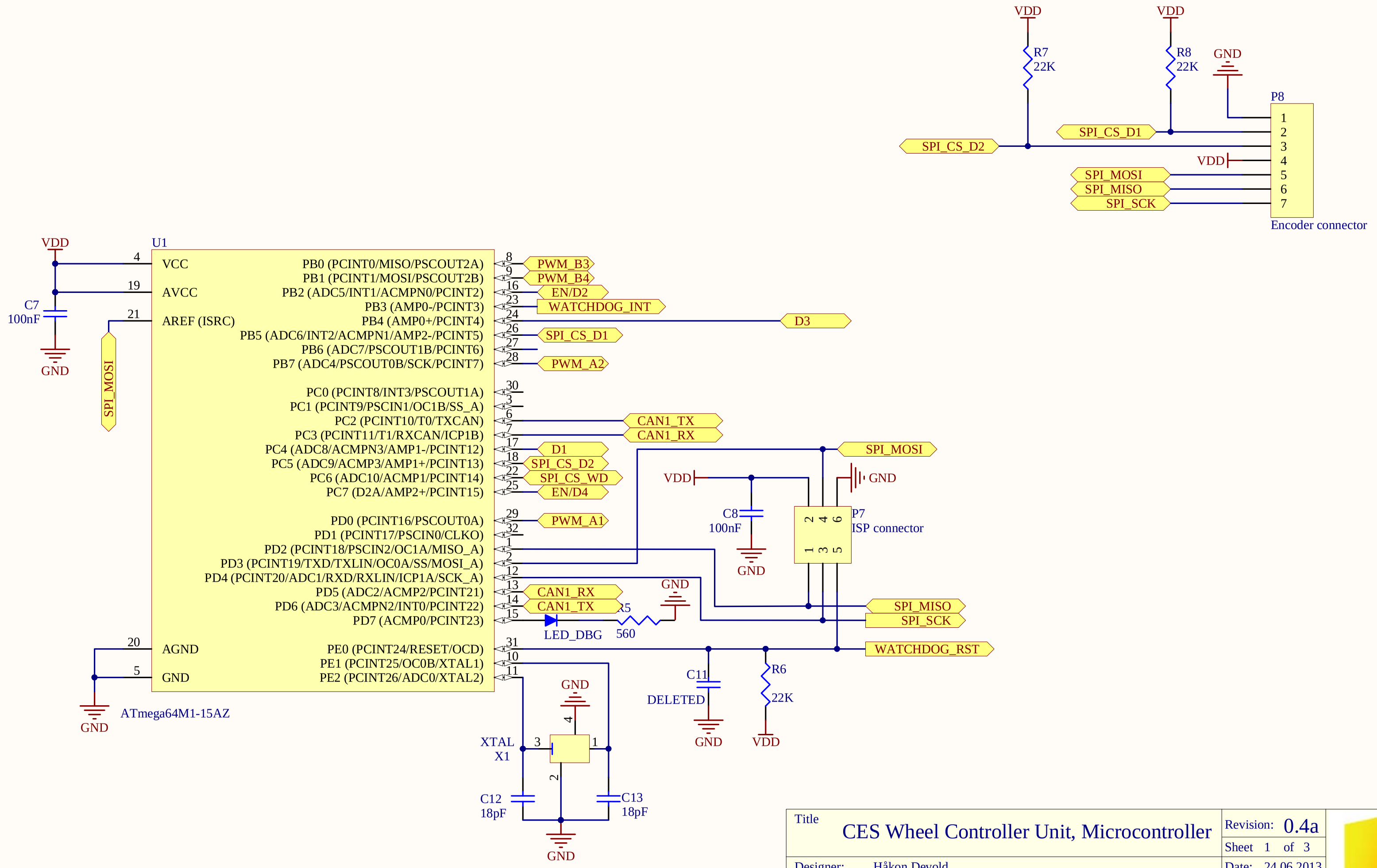
Title	CES Main Controller Unit, Peripherals	Revision: 0.3a
		Sheet 3 of 4
Designer:	Håkon Devold	Date: 13.05.2013
Project file:	Susp_MainController.PrjPcb	Time: 11:50:22
File:	Susp_MainController_Peripheral_Schem.SchDoc	Size: A4
Organization:	Revolve NTNU 2013 - R&D	



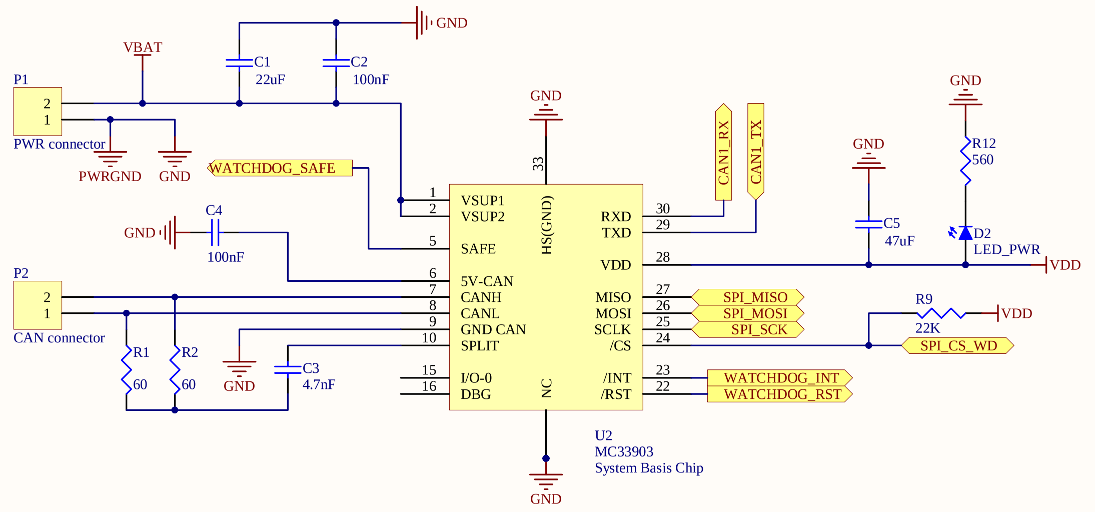
Title	CES Main Controller Unit, SD Card Holder	Revision:	0.3a
		Sheet 4 of 4	
Designer:	Håkon Devold	Date:	23.06.2013
Project file:	Susp_MainController.PrjPcb	Time:	20:31:06
File:	Susp_MainController_MMC_Schem.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		

A.2 Wheel Controller Unit Schematics

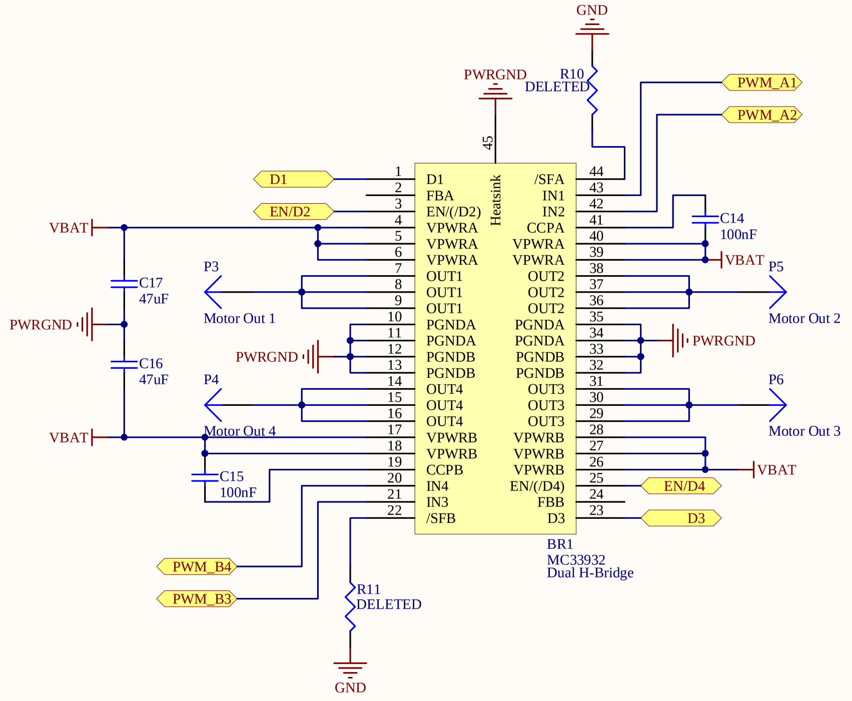
1. Wheel Controller Unit - Microcontroller
2. Wheel Controller Unit - Power Supply
3. Wheel Controller Unit - H-Bridge



Title	CES Wheel Controller Unit, Microcontroller	Revision:	0.4a
		Sheet 1 of 3	
Designer:	Håkon Devold	Date:	24.06.2013
Project file:	*	Time:	21:00:30
File:	main_schematic.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		



Title	CES Wheel Controller Unit, Power Supply	Revision:	0.4a
		Sheet 2 of 3	
Designer:	Håkon Devold	Date:	13.05.2013
Project file:	proj_2.PrjPCB	Time:	12:27:36
File:	main_schematic_psu.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		

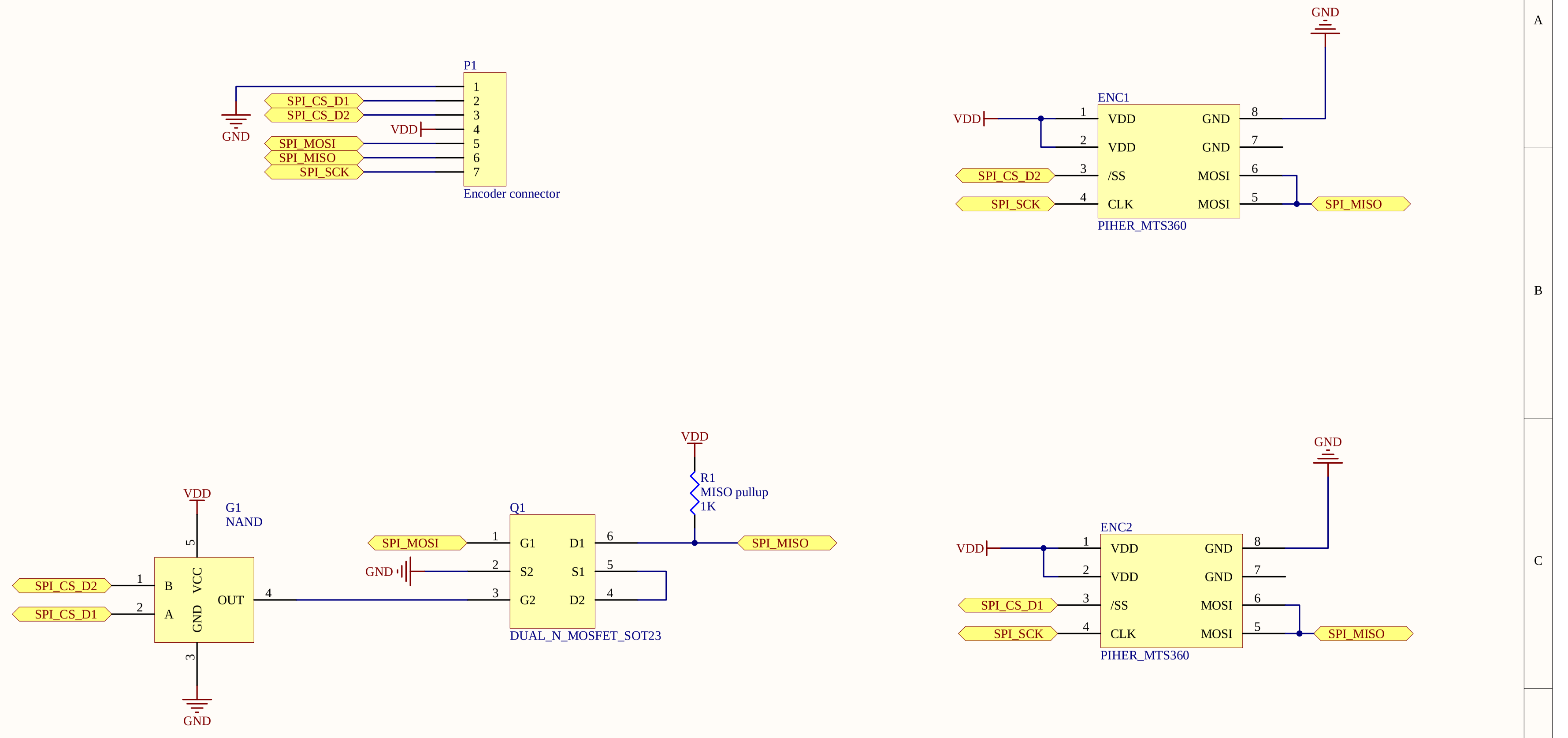


Title	CES Wheel Controller Unit, H-Bridge	Revision: 0.4a
		Sheet 3 of 3
Designer:	Håkon Devold	Date: 13.05.2013
Project file:	proj_2.PrjPCB	Time: 12:26:54
File:	main_schematic_hbridge.SchDoc	Size: A4
Organization:	Revolve NTNU 2013 - R&D	

revolve NTNU

A.3 Wheel Encoder Board Schematics

1. Wheel Controller Unit - Encoder Board

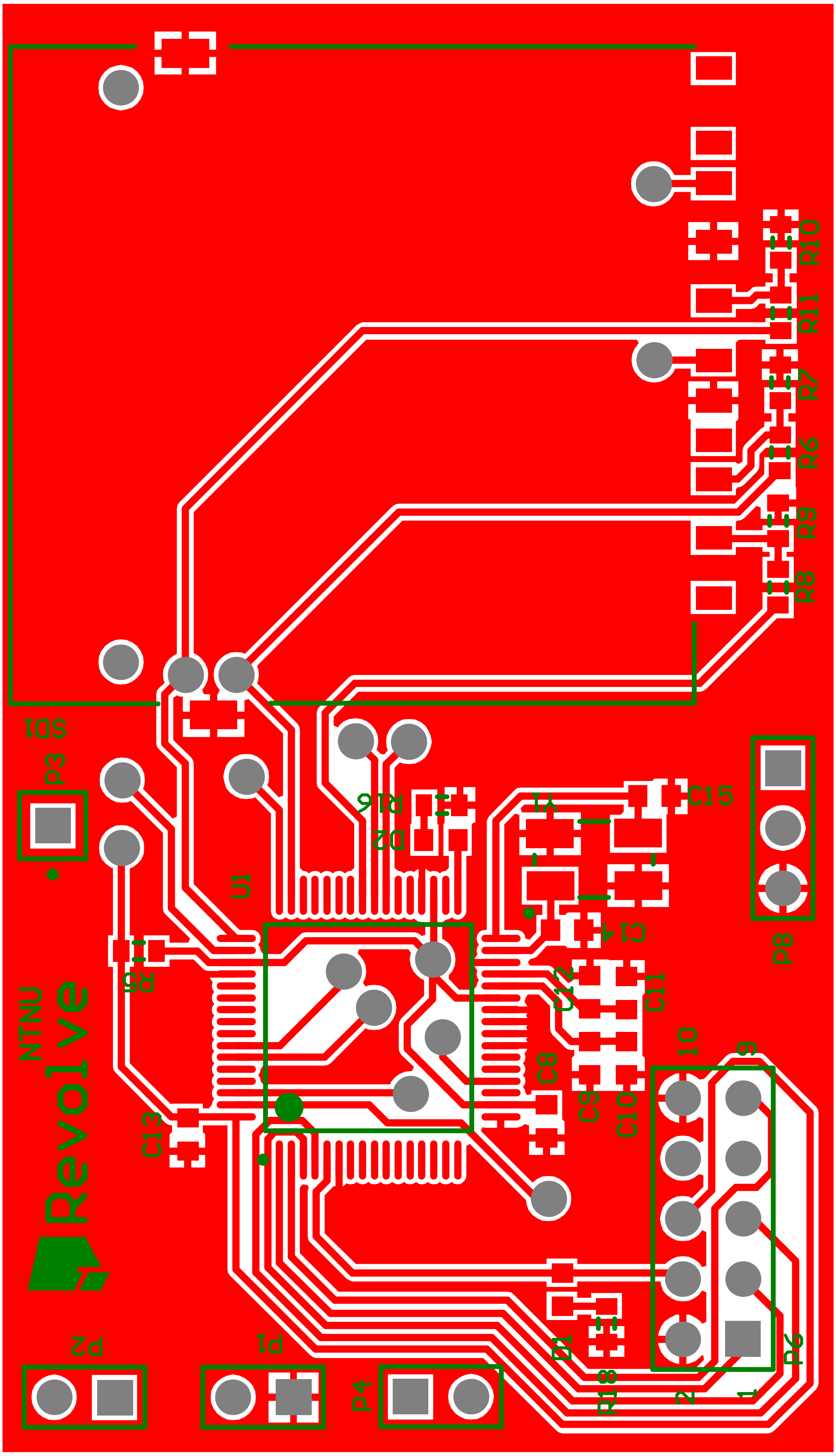


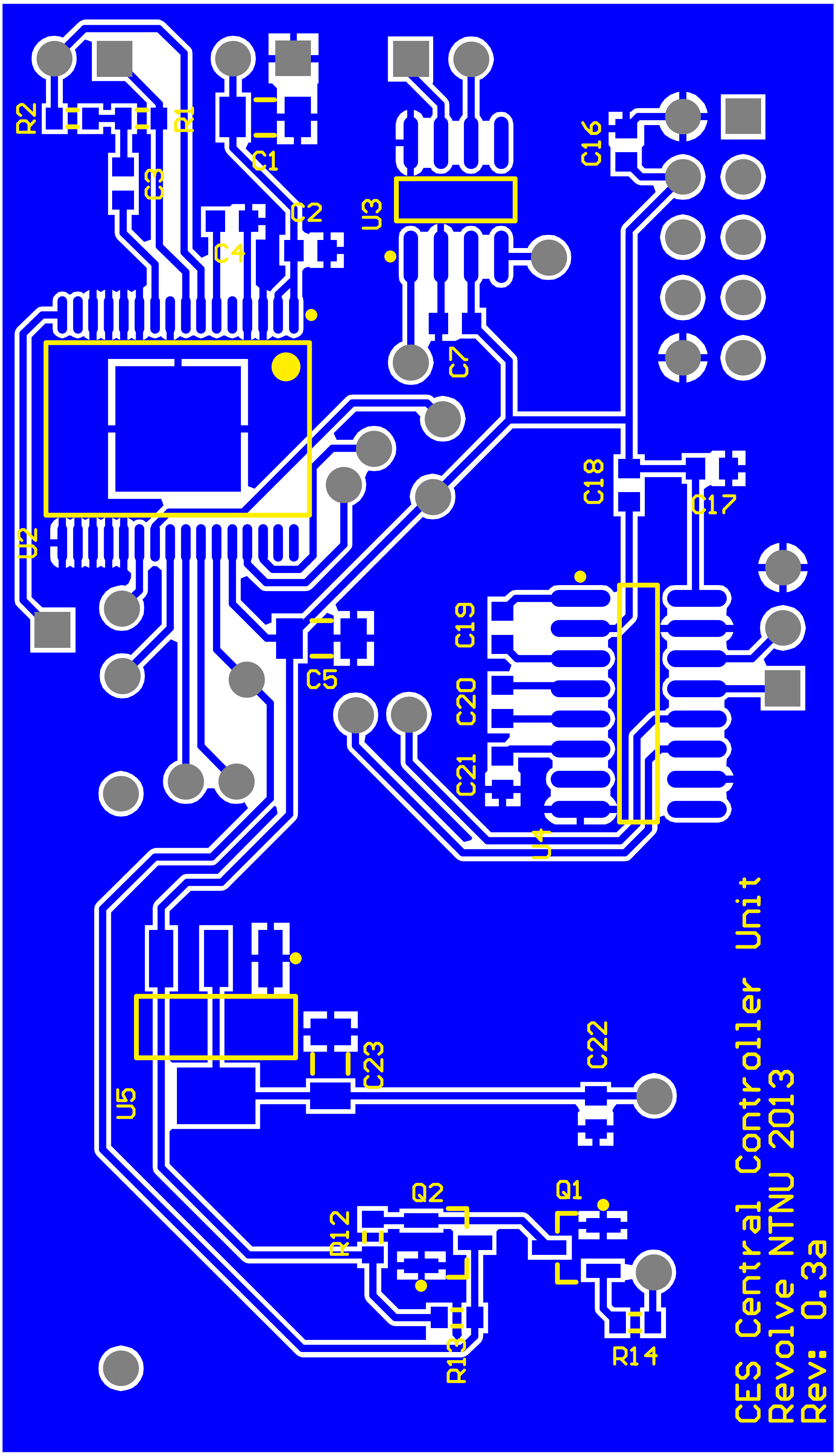
Title	Wheel Encoder Board	Revision:	0.4a
		Sheet 1 of 1	
Designer:	Håkon Devold	Date:	13.05.2013
Project file:	Susp_EncoderBoard.PrjPCB	Time:	12:04:06
File:	encoderboard_schematic.SchDoc	Size:	A4
Organization:	Revolve NTNU 2013 - R&D		

B
PCB Layouts

B.1 Central Controller Unit PCB

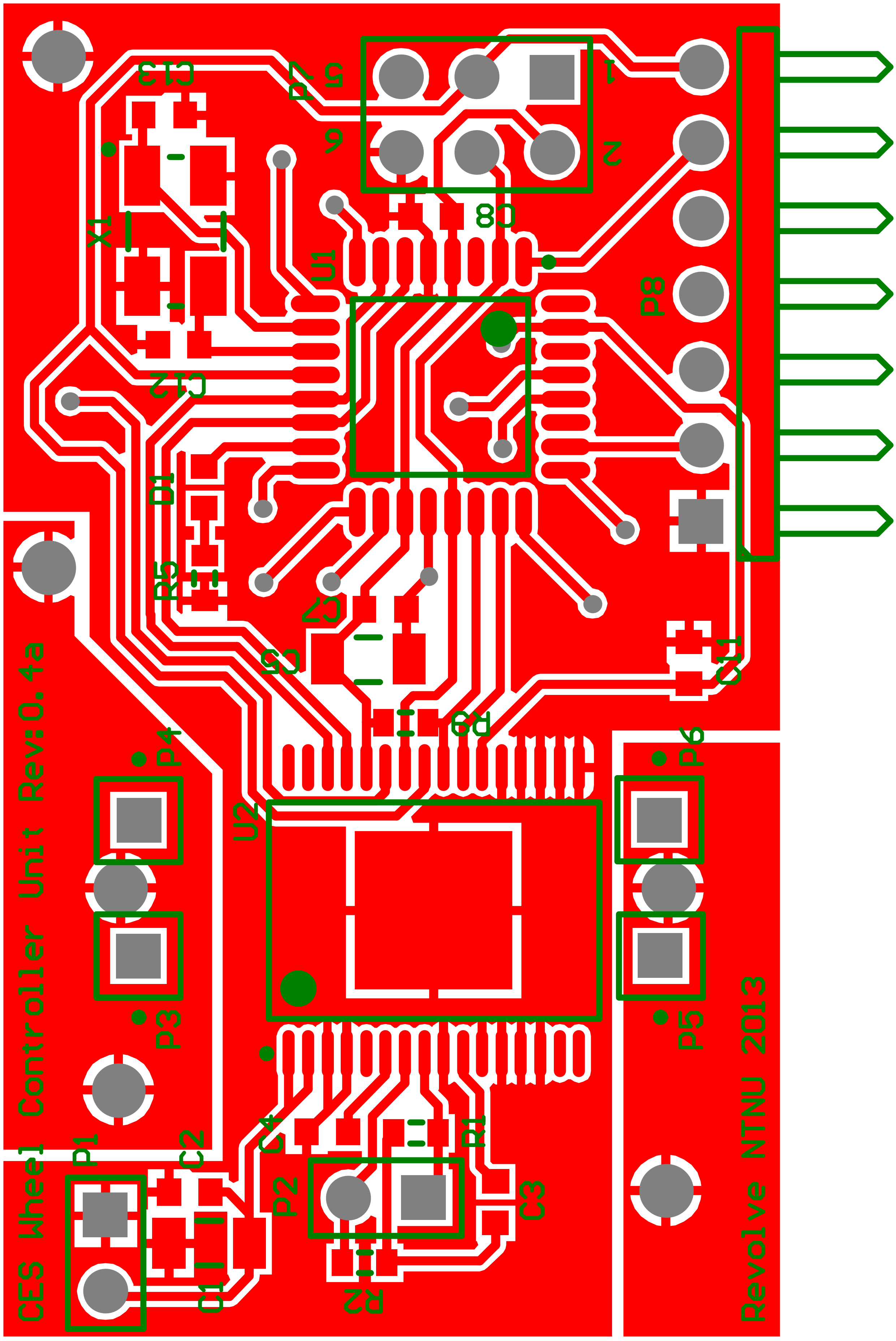
1. Central Controller Unit - Top Layer
2. Central Controller Unit - Bottom Layer

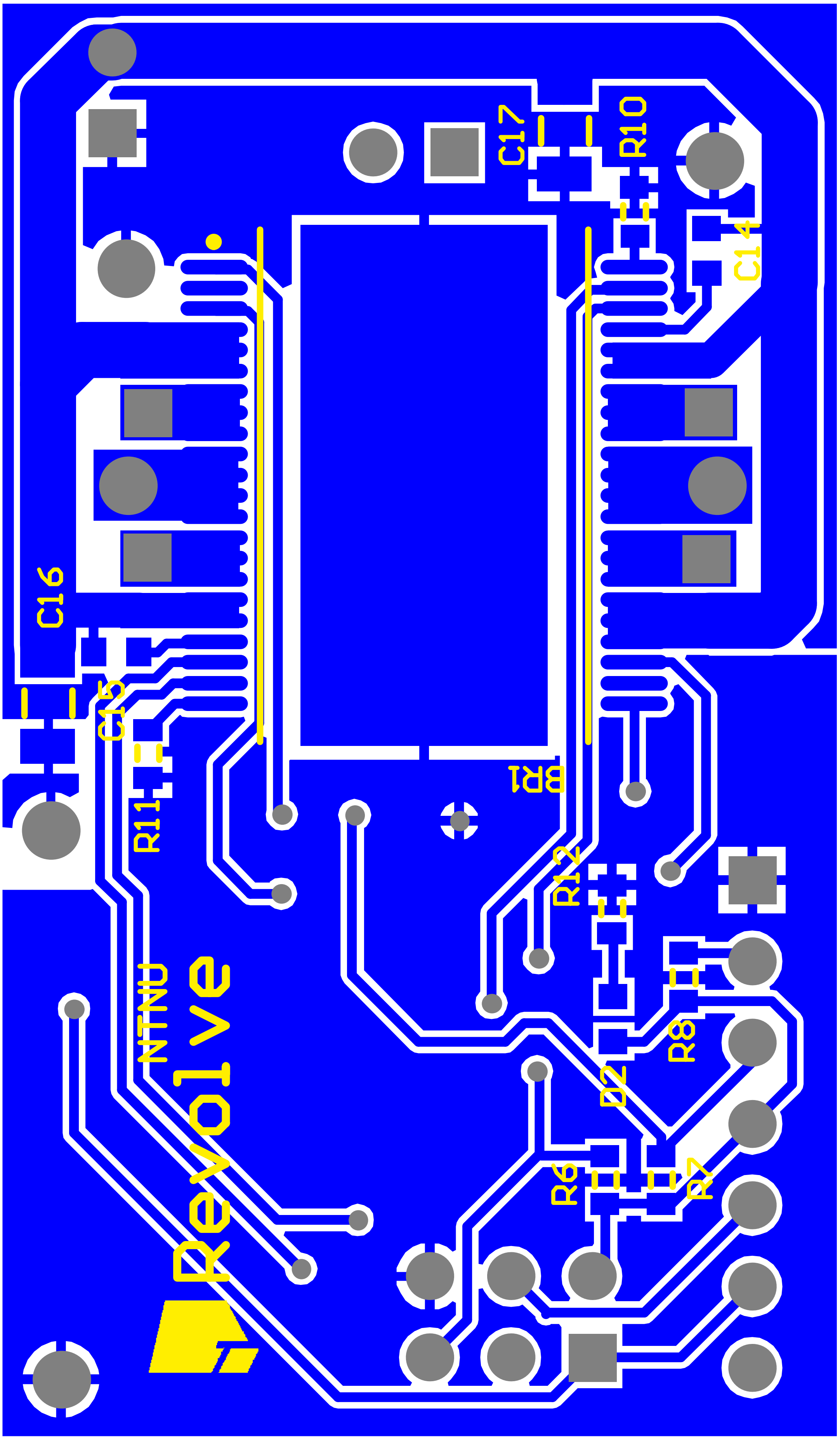

SD1
P3
U1
R16
D2
Y1
C15
C14
C12
C11
C8
C9
C10
R5
C13
NTNU
Revolve
P2
P1
P4
D1
R18
P6
P8
R10
R11
R7
R6
R9
R8
10
9
2
1


R2
R1
C1
C3
C2
C4
U3
C7
C16
U2
C18
C17
C19
C5
C20
C21
U4
C23
C22
U5
Q2
Q1
R12
R13
R14
CES Central Controller Unit
Revolve NTNU 2013
Rev: 0.3a

B.2 Wheel Controller Unit PCB

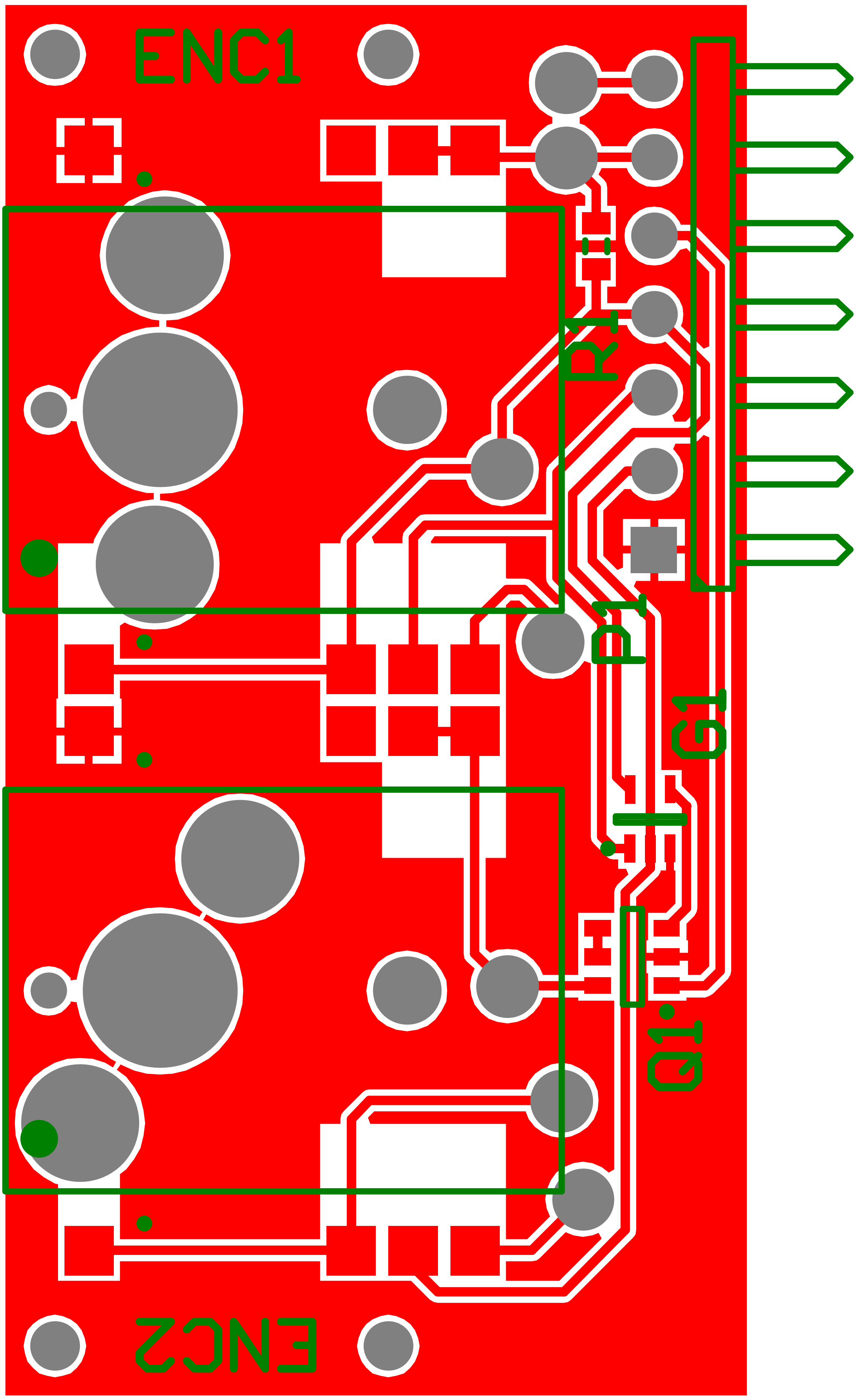
1. Wheel Controller Unit - Top Layer
2. Wheel Controller Unit - Bottom Layer

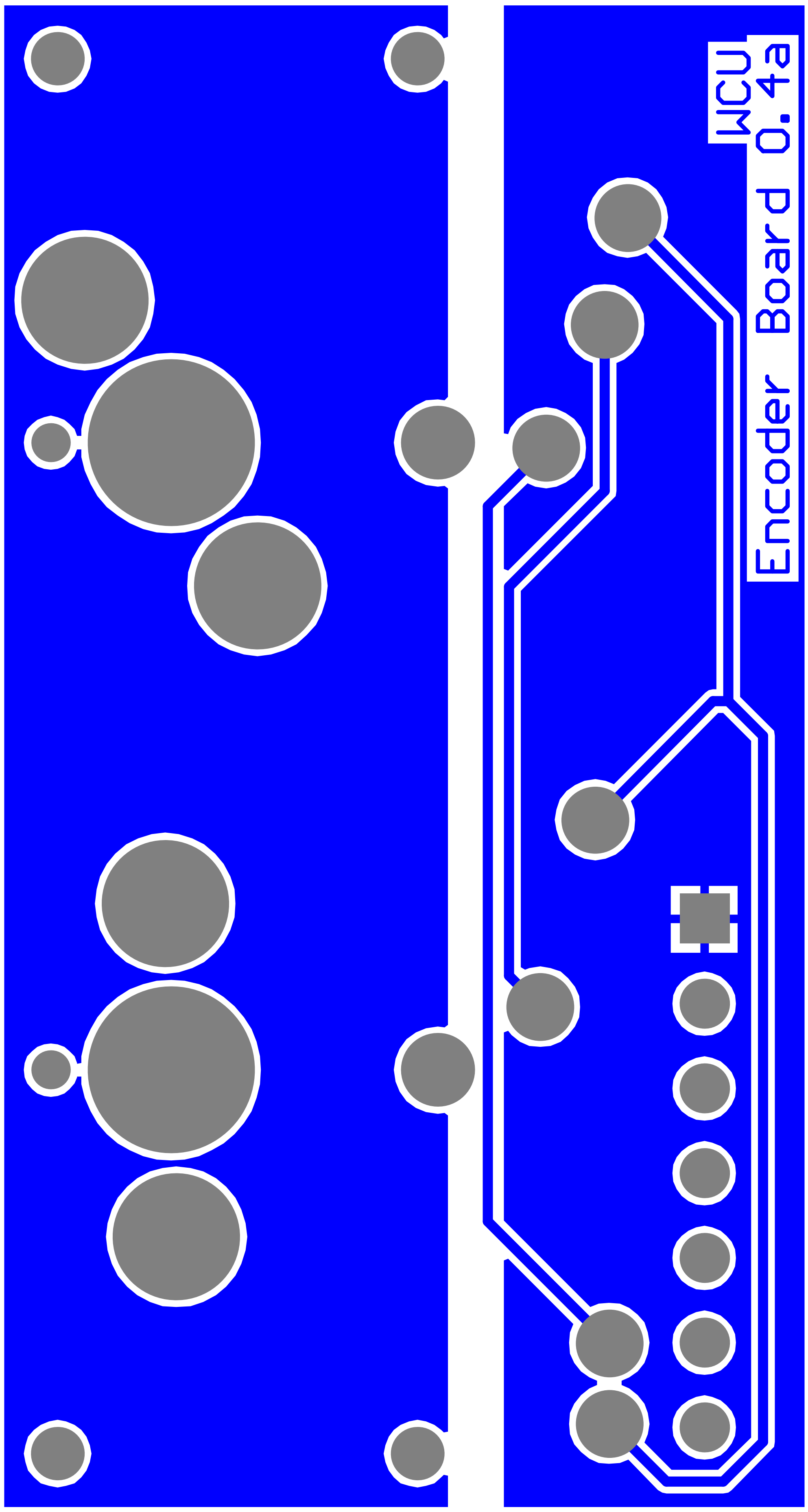

CES Wheel Controller Unit Rev:0.4a
Revolve NTNU 2013
U1
U2
X1
D1
P1
P2
P3
P4
P5
P6
P7
P8
C1
C2
C3
C4
C5
C7
C8
C11
C12
C13
R1
R2
R5
R9
1
2
5
6


C17
R10
C14
C16
C15
R11
BR1
R12
R8
D2
R6
R7
NTNU
Revolve

B.3 Wheel Encoder Board PCB

1. Wheel Encoder Board - Top Layer
2. Wheel Encoder Board - Bottom Layer


ENC1
R1
D1
G1
Q1
ENC2


WCU
Encoder Board 0.4a

C Program code

Complete program code for all software is available in the digital attachments

C.1 Central Controller Unit

C.1.1 ccu_can.h

Listing C.1: Function Prototypes and Variables - ccu_can.h

```
// Local allocation for MOB buffer in HSB_RAM
extern can_msg_t mob_ram_ch0[NB_MOB_CHANNEL] __attribute__ ((
    section (".hsb_ram_loc")));
extern can_msg_t mob_ram_ch1[NB_MOB_CHANNEL] __attribute__ ((
    section (".hsb_ram_loc")));

// ---------------------------------------------------
// CAN Message Definition: Tx Messages

    // CCU periodic status broadcast messages with own ID
    extern can_msg_t msg_ch0_tx_statusdata_sot;
    extern can_mob_t ch0_tx_statusdata_msg;

    extern can_msg_t msg_ch1_tx_statusdata_sot;
    extern can_mob_t ch1_tx_statusdata_msg;

    // Define message to send front damper set points
    extern can_msg_t msg_ch1_tx_frw_sot;
    extern can_mob_t ch1_tx_frw_msg;

    // Define message to send rear damper set points
    extern can_msg_t msg_ch1_tx_rrw_sot;
    extern can_mob_t ch1_tx_rrw_msg;


// -----------------------------------------------------
// CAN Message Definition: Rx Messages

  // Event RX messages

    // Message to check for steering wheel updates
    extern can_msg_t msg_rx_ch0_dashboard_listening;
    extern can_mob_t ch0_dashboard_rx_msg;
    extern xQueueHandle ch0_dashboard_rx_receivequeue;

    // Message to check for ext. commands sent to CCU ID
    extern can_msg_t msg_rx_ch0_extcmd_listening;
    extern can_mob_t ch0_extcmd_rx_msg;
    extern xQueueHandle ch0_extcmd_rx_receivequeue;

  // WCU state RX messages

    // Message to check state messages from WCUs
    extern can_msg_t msg_rx_ch1_wcustate_listening;
    extern can_mob_t ch1_wcustate_rx_msg;
    extern xQueueHandle ch1_wcustate_rx_receivequeue;

  // Sensor RX messages

    // Msg. to check for sensor data from the ECU (Ext.ID)
    extern can_msg_t msg_rx_ch0_ecu_listening;
    extern can_mob_t ch0_ecu_rx_msg;
    extern xQueueHandle ch0_ecu_rx_receivequeue;

    // Message to check for sensor data(0x600)
    //from brake pressure sensor(0x48) and st. angle(0x58)
    extern can_msg_t msg_rx_ch1_sensor_listening;
    extern can_mob_t ch1_sensor_rx_msg;
    extern xQueueHandle ch1_sensor_rx_receivequeue;

// -----------------------------------------------------
// Function Prototypes

void ccu_can_init(void);
```

C.1.2 controllertask.c

Listing C.2: Controller Task Function - controllertask.c

```c
void vControllerTask(void *pvParameters){

  // Define period for task execution as 10 ms
  portTickType vCh1SendTask_Delay = (portTickType)(10) /
      portTICK_RATE_MS;
  portTickType vCh1SendTask_LastWakeTime = xTaskGetTickCount();

  // Allocate mailbox for CAN messages
  ch1_tx_frw_msg.handle = can_mob_alloc(1);
  ch1_tx_rrw_msg.handle = can_mob_alloc(1);
  int8_t controller_currentmode = 0;

  while(1){

    // Generate WCU setpoints
    ccu_controller_loop();

    // Check current system mode
    controller_currentmode = ccu_controller_getControlMode();

    // If control is suspended, send static value
    if( (controller_currentmode == CONTROLMODE_DISABLED) | (
        controller_currentmode == CONTROLMODE_FAILURE) ){

      ch1_tx_frw_msg.can_msg->data.u64 = 0x00;
      ch1_tx_rrw_msg.can_msg->data.u64 = 0x00;
    }
    // If system control is active, send control variables
    else if((controller_currentmode == CONTROLMODE_MANUAL)||(
        controller_currentmode == CONTROLMODE_CONTINOUS)){

      // Set content of front wheel setpoint message
      ch1_tx_frw_msg.can_msg->data.u16[0] =
          ccu_controlVariable_output.flCompression;
      ch1_tx_frw_msg.can_msg->data.u16[1] =
          ccu_controlVariable_output.flRebound;
      ch1_tx_frw_msg.can_msg->data.u16[2] =
          ccu_controlVariable_output.frCompression;
      ch1_tx_frw_msg.can_msg->data.u16[3] =
          ccu_controlVariable_output.frRebound;

      // Set content of rear wheel setpoint message
      ch1_tx_rrw_msg.can_msg->data.u16[0] =
          ccu_controlVariable_output.rlCompression;
      ch1_tx_rrw_msg.can_msg->data.u16[1] =
          ccu_controlVariable_output.rlRebound;
      ch1_tx_rrw_msg.can_msg->data.u16[2] =
          ccu_controlVariable_output.rrCompression;
      ch1_tx_rrw_msg.can_msg->data.u16[3] =
          ccu_controlVariable_output.rrRebound;
    }

    // Post messages for transmission
    can_tx(1,
    ch1_tx_frw_msg.handle,
    ch1_tx_frw_msg.dlc,
    ch1_tx_frw_msg.req_type,
    ch1_tx_frw_msg.can_msg);

    can_tx(1,
    ch1_tx_rrw_msg.handle,
    ch1_tx_rrw_msg.dlc,
    ch1_tx_rrw_msg.req_type,
    ch1_tx_rrw_msg.can_msg);

    // Execute task with fixed period
    vTaskDelayUntil( &vCh1SendTask_LastWakeTime, vCh1SendTask_Delay
        );
  }
}
```

C.2 Wheel Controller Unit

C.2.1 Big-while Loop

Listing C.3: Main execution loop - wheel_controller_main.c

```c
// INITIALIZATION FINISHED. ENTERING WHILE LOOP.
  while(1)
  {
    //--- LOOK FOR NEW CAN SETPOINT MESSAGE
    if(can_get_status(&cmd_msg_rx) == CAN_STATUS_COMPLETED){

        //--- READ CAN MESSAGE TO BUFFER ---
        if ((cmd_msg_rx.id.std==0x419 || cmd_msg_rx.id.std==0x41A
            ) && cmd_msg_rx.dlc == 8)
        {
          //--- RECEIVE NEW SETPOINT MESSAGES ---
          if( ((wcu_hwdef_thisUnit == CANR_GRP_SUS_ID+
              CANR_MODULE_SUSP_WCUFL)&&(cmd_msg_rx.id.std==0x419)
              ) || ((wcu_hwdef_thisUnit == CANR_GRP_SUS_ID+
              CANR_MODULE_SUSP_WCURL)&&(cmd_msg_rx.id.std==0x41A)
              ) ){
            controlVariables[0].referenceValue = ( (int16_t)(
                cmd_msg_rx_data[0] << 8) + (int16_t)
                cmd_msg_rx_data[1] );
            controlVariables[1].referenceValue = ( (int16_t)(
                cmd_msg_rx_data[2] << 8) + (int16_t)
                cmd_msg_rx_data[3] );
          }
          if( ((wcu_hwdef_thisUnit == CANR_GRP_SUS_ID+
              CANR_MODULE_SUSP_WCUFR)&&(cmd_msg_rx.id.std==0x419)
              ) || ((wcu_hwdef_thisUnit == CANR_GRP_SUS_ID+
              CANR_MODULE_SUSP_WCURR)&&(cmd_msg_rx.id.std==0x41A)
              ) ){
            controlVariables[0].referenceValue = ( (int16_t)(
                cmd_msg_rx_data[4] << 7) + (int16_t)
                cmd_msg_rx_data[5] );
            controlVariables[1].referenceValue = ( (int16_t)(
                cmd_msg_rx_data[6] << 7) + (int16_t)
                cmd_msg_rx_data[7] );
          }
        }
        can_cmd(&cmd_msg_rx); // Receive new message
    }
    //--- LOOK FOR NEW CAN UNIT STATE MESSAGE
    if(can_get_status(&ccustate_msg_rx) == CAN_STATUS_COMPLETED){

        //--- READ CAN MESSAGE TO BUFFER ---
        if ( ccustate_msg_rx.id.std== (CANR_FCN_DATA_ID|
            CANR_GRP_SUS_ID|CANR_MODULE_SUSP_CCU) &&
            ccustate_msg_rx.dlc == 3 && ccustate_msg_rx_data[0]
            == 1 )
        {

          // Update system state from message if not equal to
              current state
          updateUnitState( ccustate_msg_rx_data[1] );

          // Reset CCU watchdog timer
          ccuWatchdogTimeoutCounter = 0;
        }

        can_cmd(&ccustate_msg_rx);
    }

    //--- IF HARDWARE TIMER HAS EXPIRED, RUN CONTROLLER LOOP
    if(timerflag){

        //--- RESET TIMER FLAG ---
        timerflag = 0;

        //--- GET VALVE POSITION ---
        controlVariables[0].measurementValue =
            PIDController_GetPosition(ENC1);
        controlVariables[1].measurementValue =
            PIDController_GetPosition(ENC2);


        if( (getUnitState() == CONTROLMODE_MANUAL) || (
            getUnitState() == CONTROLMODE_OPERATIVE) ){
          //--- RUN PID LOOP ---
          controlVariables[0].controlInput = pid_Controller(
              controlVariables[0].referenceValue,
              controlVariables[0].measurementValue, &pidData[0]);
          controlVariables[1].controlInput = pid_Controller(
              controlVariables[1].referenceValue,
              controlVariables[1].measurementValue, &pidData[1]);
        }
        else if( getUnitState() == CONTROLMODE_STANDBY ){
          //--- SET IN PRECONFIGURED POSITION
          controlVariables[0].controlInput = pid_Controller(
              pidDefaultPosition[wcu_hwdef_thisUnit-
              CANR_GRP_SUS_ID-CANR_MODULE_SUSP_WCUFL][0],
              controlVariables[0].measurementValue, &pidData[0]);
          controlVariables[1].controlInput = pid_Controller(
              pidDefaultPosition[wcu_hwdef_thisUnit-
              CANR_GRP_SUS_ID-CANR_MODULE_SUSP_WCUFL][1],
              controlVariables[1].measurementValue, &pidData[1]);

          //--- IF RETURNED TO PREDEF POSITION DISABLE
```

```
      if ( abs(pidDefaultPosition[wcu_hwdef_thisUnit-CANR_GRP_SUS_ID-CANR_MODULE_SUSP_WCUFL][0]-controlVariables[0].measurementValue)<CONTROLTRESHOLD_DISABLE
        && abs(pidDefaultPosition[wcu_hwdef_thisUnit-CANR_GRP_SUS_ID-CANR_MODULE_SUSP_WCUFL][1]-controlVariables[1].measurementValue)<CONTROLTRESHOLD_DISABLE ){
        motor_disablePSC();
      }
    }
    else{
      //--- IF IN OTHER STATE GENERATE NO OUTPUT
      controlVariables[0].controlInput = 0;
      controlVariables[1].controlInput = 0;
    }

    //--- RETURN CAN MESSAGE ---
    if(controllerloop_cnt%10 == 0){
      led_toggle();
      wcu_msg_tx.dlc = 5;
      wcu_msg_data[0] = getUnitState();                  // SEND STATE VALUE
      wcu_msg_data[1] = (controlVariables[0].measurementValue>>8);  // SEND ENCODER 0 MEASUREMENT VALUE
      wcu_msg_data[2] = (controlVariables[1].measurementValue>>8);  // SEND ENCODER 0 MEASUREMENT VALUE
      wcu_msg_data[3] = (controlVariables[0].controlInput>>8);    // SEND ENCODER 1 MEASUREMENT VALUE
      wcu_msg_data[4] = (controlVariables[1].controlInput>>8);    // SEND ENCODER 1 MEASUREMENT VALUE

      while(can_cmd(&wcu_msg_tx) != CAN_CMD_ACCEPTED);
      while(can_get_status(&wcu_msg_tx) == CAN_STATUS_NOT_COMPLETED);

      if (controllerloop_cnt>=100)
      {
        sbc_wdt_refresh();
        controllerloop_cnt = 0;
      }

      if (getUnitState() == CONTROLMODE_MANUAL || getUnitState() == CONTROLMODE_OPERATIVE){
        ccuWatchdogTimeoutCounter++;
      }

      if (ccuWatchdogTimeoutCounter > CCUWATCHDOG_TIMEOUTTRESHOLD)
      {
        updateUnitState(CONTROLMODE_ERROR);
      }
```

```c

          }

          //--- INCREMENT LOOP COUNTER
          controllerloop_cnt++;
      }

      //--- CHECK IF PWM PERIOD IS FINISHED
      if ( (PIFR&(1 << PEOP))==1 ){
        PIFR |= (1 << PEOP);

        //--- SET NEW MOTOR SPEED ---
        motor_A_setspeed(controlVariables[0].controlInput/33);
        motor_B_setspeed(controlVariables[1].controlInput/33);
      }
    }
}
```