CHALMERS

Design and implementation of a central control unit in an automotive drive-by-wire system

Master of Science Thesis in Embedded Electronic System Design

Alexandra Angerd
Andreas Johansson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, September 2013

Design and implementation of a central control unit in an automotive drive-by-wire system

Alexandra Angerd
Andreas Johansson

Examiner: Lars Svensson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, September 2013

Abstract

X-by-wire is a technique which is commonly used in aeronautics, but a challenge to face for the automotive industry. The objective of Sigma Kudos, which is a consultancy company in the automotive industry, is to implement a drive-by-wire system in a full-size electric car. As part of the process of developing a prototype, the purpose of this project was to design a central control unit which would bring the prototype one step closer to vehicle implementation. In addition, the central control unit had to support an already existing system, containing throttle, brake, and steering functionality. To achieve this a platform was chosen, consisting of a TMS570 development board from Texas Instruments, and a GPL licensed FreeRTOS software platform from Real Time Engineers Ltd. The outcome of the project was an extensible platform with the potential of supporting full-size vehicles. However, questions arose regarding the use of an open source software platform in a safety-critical system, which is an issue that should be investigated further.

Acknowledgements

We would like to offer our special thanks to David Rydén, our supervisor at Sigma Kudos Engineering Services, for his technical support and for sharing his insights into the automotive industry. We also would like to express our very great appreciation to our examiner Lars Svensson, for his academic support, constructive critique, and proof-reading of the report.

In addition, we wish to thank various people for their contribution to this project; Klas Persson, CEO at Sigma Kudos Engineering Services, who approved and initiated this project; Risat Pathan, for his help regarding response times in real-time systems; and Roger Johansson, for interesting discussions regarding real-time systems.

Alexandra Angerd and Andreas Johansson, Göteborg 4/9/13

Contents

Abbreviations

Acronym	Description
ABS	Anti-lock Braking System
A/D	Analogue-to-Digital
ADC	Analogue-to-Digital Converter
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
AWG24	American Wire Gauge 24, a standardized wire gauge size with a diameter of 0.511 mm
BCD	Binary-Coded Decimal
BSW	Basic Software
CAN	Controller Area Bus
CAN-H	CAN High
CAN-L	CAN Low
CCU	Central Control Unit
CRC	Cyclic Redundancy Check
DMIPS	Dhrystone Million Instructions Per Second
ECC	Error-Correcting Code
ECU	Electrical Control Unit
EMI	Electromagnetic Interference
GIO	General Input/Output
GPL	GNU General Public Licence
HalCoGen	HAL Code Generator Tool
I/O	Input/Output
IDE	Integrated Development Environment
IEC61508	An international standard regarding functional safety in electrical/electronic/programmable safety-related systems
ISR	Interrupt Service Routine
LIN	Local Interconnect Network
MIPS	Million Instructions Per Second
PCB	Printed Circuit Board

Abbreviations, continued

Acronym	**Description**
RC	Radio Controlled
RISC	Reduced Instruction Set Computing
RJ10	A connector with four pins
RJ45	A connector with eight pins
RTE	Runtime Environment
RTI	Real-Time Interrupt
SIL	Safety Integrity Level
SPI	Serial Peripheral Interface
SWC	Software Component
WHIS	Wittenstein High Integrity Systems

1

Introduction

Drive-by-wire is a technique which aims to replace mechanical control systems in vehicles with an electronic equivalent, consisting of Electrical Control Units (ECUs), communication buses, sensors, and actuators. In a strict drive-by-wire system, there is no mechanical connection between the driver's input interface and the actuators which control the speed and direction of the vehicle.

Already a well-used technique in aeronautics such as Airbus A320, X-by-wire is now a challenge to face for the automotive industry. Drive-by-wire in cars may offer a lot of benefits, such as making it easier to integrate driver assistance functions, reduce the repair time, and reduce the weight of the car. It also brings the technology one step closer to the driverless car, where one example is Google's autonomous car [1]. Google's autonomous car is a modified Toyota Prius, which uses a drive-by-wire system. Other car manufacturers, such as Volkswagen and Lexus, also use the drive-by-wire concept, although none of them are strictly drive-by-wire systems as they retain some mechanical links between the input interface and the actuators.

The objective of consulting company Sigma Kudos is to implement a strict drive-by-wire system in a full-size electric car prototype, with the aim to attract new employees at job fairs and similar activities. The basic drive-by-wire prototype should include a solution which handles systems controlling the steering functionality, as well as the throttle and brake functionality. In addition, it has to be possible to add more functionality in the future. When this project began, some development had already been done, and the prototype consisted of two subsystems which operated in parallel.

1.1 Purpose

The two subsystems in the prior prototype contained two ECUs each, where one ECU in each subsystem was a sender ECU, and the other two were receiver ECUs. The system was undersized for use together with a full-size vehicle, as it was implemented in order to

control wheels taken from a radio-controlled car. Thus, the purpose of this project was to prepare the drive-by-wire system for implementation in a full-size vehicle by designing a Central Control Unit (CCU), and thereby increasing the capacity of the prototype.

1.2 Objective

The main objective of the project was to restructure the prior solution by merging the two sender ECUs present in the prior prototype. This process included several milestones, such as choosing a suitable hardware and software platform, assemble the hardware, install the software platform, develop new application software, and make the new platform compatible with the prior prototype.

Furthermore, the solution had to be extensible, both with respect to hardware and software. This means that the capacity of the microcontroller in the CCU must be such that more functionality can be added without a significant performance loss. It also means that the system must be designed to support the possibility to include additional sensors and actuators in a future system.

A secondary objective of this project was to implement a temporary way to monitor the major activities inside the CCU. However, not much effort was spent on this part because in the future, the prototype will include a dashboard which holds this functionality.

1.3 Scope

Because the objective of this project was to reconfigure the existing system architecture, no new functionality was added to the wheel and engine control units. The reconfiguration did not involve replacement of the sensors and actuators used in the prior prototype.

In addition, the project was only supposed to bring the prototype one step closer to a complete product. Therefore, PCB design was avoided by using pre-assembled hardware as far as possible.

Even though fully implemented drive-by-wire systems require the properties of a fault-tolerant and safety-critical system [2], this was omitted from this project. However, the new CCU must have the capacity to implement at least some of these properties in the future.

1.4 Method

The project was carried out at Sigma Kudos Engineering Services. A supervisor at the company monitored the progress during weekly meetings throughout the project. At the meetings, the current status and problems were discussed, and the time plan was evaluated and modified if needed. All expenses of the project had to be approved by the supervisor, with the result that the company was responsible for all decisions regarding the budget.

In order to choose an appropriate platform for the CCU, a pre-study was carried out. The prior prototype was analyzed together with possible solutions found by a literature study, and requirements were determined. When the requirements of the new unit were set, a hardware platform was selected. In addition, based on this platform and the requirements of the software, a suitable software platform was chosen.

When the software and hardware platform had been selected, the implementation phase started. In this phase, the software and hardware were developed in parallel in an agile fashion. At last, an analysis of the CCU was made in order to determine how much resources the implementation consumes.

1.5 Technical background

The modern vehicle of today is equipped with several ECUs. The ECUs are used in driver-to-vehicle interfaces, in actuator control, and in information gathering regarding the current vehicle state. For instance, one ECU may be responsible for displaying the current vehicle speed on the dashboard, while another ECU is responsible for controlling the Anti-lock Braking System (ABS). As seen in Figure 1.1, the ECUs are interconnected via a communication bus, which the ECUs use to send and receive data. Thus, data regarding the state of the ABS is presented on the dashboard, even though the dashboard is not directly connected to the brakes.

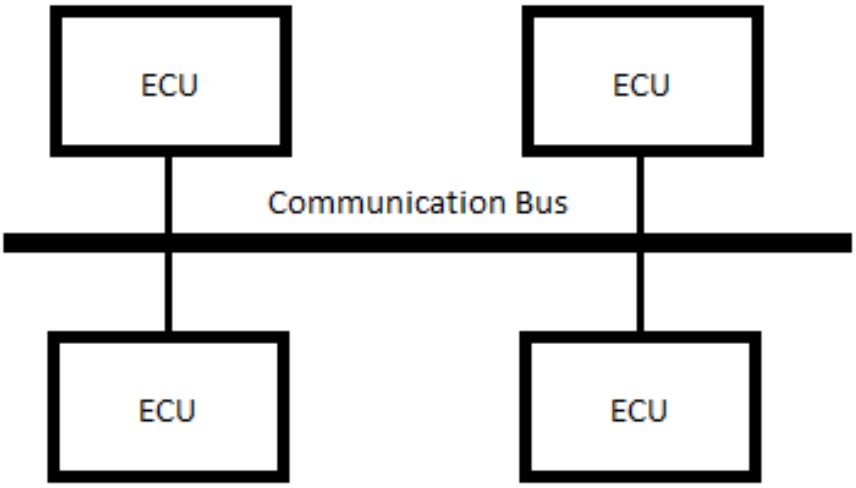

Figure 1.1: An illustration of how the ECUs are interconnected with a communication bus.

The communication bus is based on a communication protocol used by the ECUs. In the automotive industry, mainly two different communication protocols are available: FlexRay and Controller Area Network (CAN). The main differences between the two are that FlexRay offers higher reliability due to its redundancy features, and has a maximum

transmission rate of 10 Mbps, while CAN has a maximum transmission rate of 1 Mbps. However, FlexRay is more expensive than CAN, which makes CAN a more suitable choice in low-budget projects [3]. Because of this, the prototype described in this report is using the CAN communication protocol.

1.5.1 CAN

CAN was designed by Bosch in 1983, and presented in 1984 at the Society of Automotive Engineers (SAE) [4]. The purpose of CAN was to increase the reliability and the fuel-efficiency of the vehicle. Furthermore, it was intended to solve communication problems occurring when multiple ECUs are transmitting simultaneously [5].

A message sent over CAN is called a frame, and consists of several fields. The fields hold information regarding the frame, such as the number of data bytes, the data to be sent, and a frame identifier. The identifier is unique for each type of message, and in addition used for message prioritization. The total length of the frame depends on whether the frame is a base frame, with a length of 11 bytes, or an extended frame, with a length of 29 bytes. The additional 18 bytes in the extended frame are used to extend the number of available frame identifiers, making it possible to include more types of messages on the bus. Figure 1.2 illustrates the different fields in a CAN base frame.

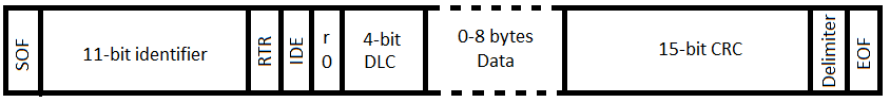

Figure 1.2: Fields in a CAN base frame.

The transmission of a CAN frame is carried out using a differential bus consisting of two lines: CAN high and CAN low. Using a differential bus for communication reduces interference and allows for ground shifting. Furthermore, it allows the ECUs to have different supply voltages, as long as the differential voltage can be achieved in all nodes [6]. Figure 1.3 illustrates CAN high and CAN low during communication.

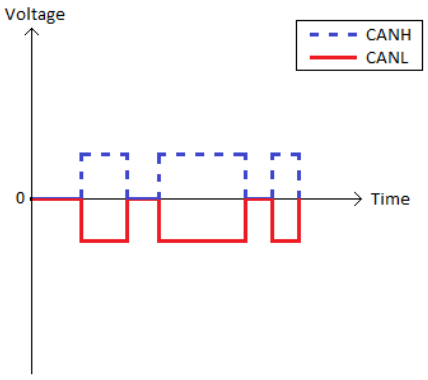

Figure 1.3: Figure of CAN high and CAN low during communication.

When the CAN bus is idle, the signal level of the bus is high. The Start-Of-Frame (SOF) bit signals that a transmission has started, by setting a low signal level on the bus. Following after the SOF bit is an 11-bit identifier, which sets the priority of the frame. A low identifier value on a CAN bus indicates a high priority. This behaviour is illustrated in Figure 1.4, where two ECUs are transmitting at the same time. In this example, the first ECU sends a frame with the identifier value seven, while the second ECU sends a frame with the identifier value six. When this happens, the message of the second ECU will be prioritized over the message of the first ECU. This is because a low signal value is dominant, while a high signal level is recessive [7].

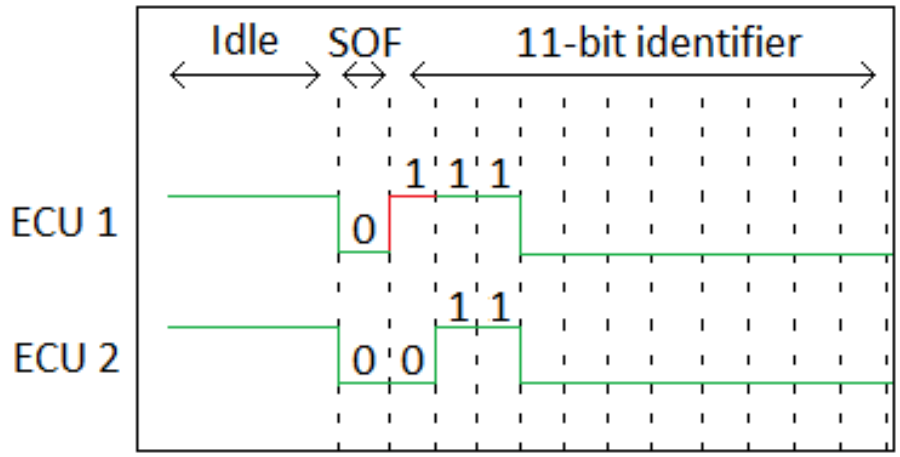

Figure 1.4: A representation of how the signals are interpreted when two CAN messages are sent.

The identifier is followed by a Remote Frame Request (RFR) bit. This bit specifies if the frame requests data or if the frame intends to deliver data, which corresponds to a high or a low value on the bus, respectively [4] [5].

When the RTR is sent, a bit called Identifier Extension (IDE) is transmitted. The IDE informs the receiver about the type of frame being sent. If the IDE bit is set low, the frame is a base frame, and if it is set high, the frame is an extended frame. The IDE bit is followed by a reserved bit, which is set to the low dominant value.

The data is sent after a field called Data Length Code (DLC). This field specifies the number of data bytes, ranging between zero and eight bytes. In order to ensure that the frame is received correctly, a Cyclic Redundancy Check (CRC) is added to the frame. When all fields have been transmitted, End-Of-Frame (EOF) is signaled by setting the signal level high. At this point, the bus is available for a new frame transmission.

1.6 Report disposition

The report is structured as follows: the prior prototype, which was the foundation of this project, is described in chapter 2. In addition, the prior prototype is evaluated in order to find weak spots which can be reinforced by the new CCU prototype. In chapter 3, the selection process of a platform for the CCU is described. Possible hardware and software platforms are discussed, and the best suited candidates are chosen.

Chapter 4 explains the implementation of the new CCU, describing both the functional and the physical architecture. The resulting system is then analysed in chapter 5, where the CPU load, memory utilization, and pin and channel resources are presented. In addition, chapter 5 contains a response time analysis.

The resource analysis, the response time analysis and the problems encountered during the project are discussed in chapter 6. In addition, possible future projects are suggested.

The report ends with a conclusion in chapter 7, which also summarizes the project.

2

The prior prototype

In order to understand the foundation on which this project stands, an evaluation of the prior prototype had to be done. The prototype consists of two separately developed subsystems, tied together with a CAN bus. The first subsystem implements steer-by-wire functionality, and consists of one sender ECU and one receiver ECU. The sender ECU converts an analogue value, corresponding to a steering wheel angle, into a digital value. The value is packed into a CAN message and transmitted on the CAN bus. The receiver ECU reads the CAN message, and sets a wheel position corresponding to the information given by the CAN message.

The other subsystem also consists of one sender and one receiver ECU, and works in a similar manner. However, this subsystem implements the brake- and throttle-by-wire functionality. An overview of the prototype can be seen in Figure 2.1. As a result of the distributed system architecture, all ECUs execute independent of each other, disregarding the CAN bus. Because the CCU replaces the sender ECUs in the prior prototype, the emphasis on the evaluation are placed on the sender ECUs.

All ECUs are built on a microcontroller called ATmega128 [8]. The microcontroller includes an 8-bit AVR RISC-based processor, which is running at a frequency of 16 MHz. The throughput of the processor is about 1 MIPS per MHz. In addition to the microcontroller, a CAN transceiver [9] is available in each ECU. The CAN transceiver can be accessed by the microcontroller via the communication protocol Serial Peripheral Interface (SPI). Hence, the microcontroller is able to transmit and receive messages on the CAN bus by controlling the CAN transceiver via SPI. The CAN transceivers used in the existing system are designed to use CAN 2.0B, and configured to use the maximum transmission rate of 1 Mbit/s.

Moreover, the prototype is powered using a power box which offers two different voltage levels; 5 V with a maximum supply current of 2.1 A, and 12 V with a maximum supply current of 1.2 A. The power is distributed through the system via D-SUB cables. Since the power lines are traversing through all ECUs in the system, the CAN bus is

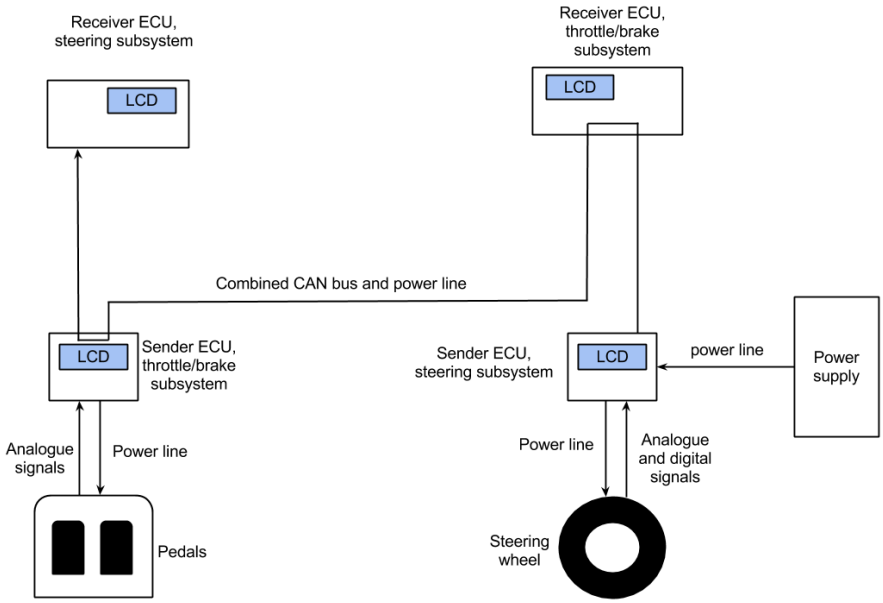

Figure 2.1: A simplified overview of the prior prototype.

distributed using the same D-SUB cables as the power lines.

2.1 The steering subsystem

The physical architecture of the steering system consists of a steering wheel, a sender ECU, and a receiver ECU. The receiver ECU is directly connected to a pair of small wheels controlled by a servomotor. In order to communicate with the sender ECU, a D-SUB connector is mounted on the receiver ECU. In addition, it also controls a display. Figure 2.2 illustrates the receiver ECU.

The sender ECU, illustrated by Figure 2.3, includes two D-SUB connectors and one RJ45 connector. The first D-SUB connector is a power connector which receives a 12 V source and a 5 V source from the power box. Both of the power sources are redirected to the second D-SUB connector, the system connector, which in addition to the power sources also carries a CAN-H signal and a CAN-L signal. The system connector connects the sender ECU with the rest of the system, and thus distributes both the power sources and the CAN signals to the receiver ECU.

The RJ45 connector is used to connect the steering wheel to the sender ECU. In order to make it possible to read the angle of the steering wheel, an integrated circuit capable of measuring linear or rotary motion [10] is used. In this circuit, a magnet is used to generate signals to make it possible to read the direction of the motion. Hence, this circuit will be called the magnet sensor from this point forward. The relevant output signals from the magnet sensor which are used by the sender ECU are listed in Table 2.1. In addition to these signals, the RJ45 connector also carries the power supply for

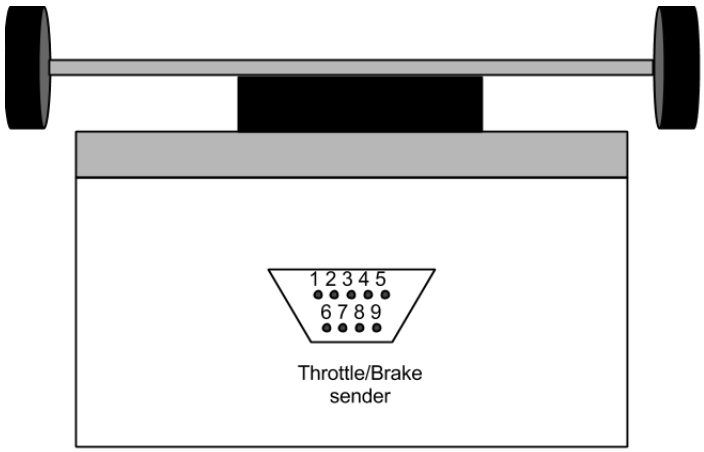

Figure 2.2: Connector and wheels of the steering receiver ECU.

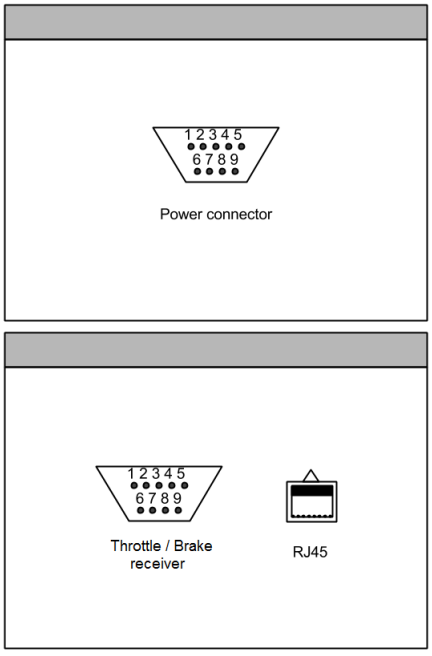

Figure 2.3: The connectors on the sender ECU of the steering subsystem.

Table 2.1: Output signals from the magnet sensor which are used by the sender ECU.

Pin name	**Type of signal**
A	Digital
B	Digital
Index	Digital
A0	Analogue

Table 2.2: Structure of the CAN message from the sender unit in the steering subsystem.

CAN message configuration	**Value**
Data (field 0)	Quadrature counter
Identifier	0x12C
Length	1 byte

the magnet sensor.

To decide the rotation of the steering wheel, the A and B signals are two pulse trains which are phase-shifted by 90 °. If A leads B, the direction is clockwise. Similarly, if B leads A, the direction is counter-clockwise. Each pulse from the A and B signal are called a quadrature pulse, and a single index pulse is generated every 40 quadrature pulse. The index signal can be used to define a start position. The analogue signal A0 is used to calibrate the distance between the magnet and the sensor. If the value of analogue signal is out of bounds, the digital signals may be too weak to read.

The responsibility of the sender ECU in this subsystem is to measure the steering wheel angle, show the angle at an LCD display, and transmit a CAN message to the receiver ECU. The software application of the sender ECU consists of an initiation phase and an infinite loop. In the initiation phase, all necessary initiations are made. An important part of this phase is to set a home position for the magnetic sensor. This makes it possible for the software to create a starting point for the magnetic sensor, and calculate the current position of the steering wheel.

After the initiation, the infinite loop is started. Within this loop, the software switches between three different functions, simulating a real-time behaviour where each task is given two milliseconds each. Although the software switches between three different tasks, only two of them are implemented, namely a CAN task and an LCD task. The LCD task converts a quadrature counter value, which is described below, to an ASCII value, and puts the steering rate and direction onto the display.

The CAN task creates a new CAN message, with the configurations shown in Table 2.2. After making sure that the CAN buffer is free, it puts the quadrature counter value into the data field of the CAN message and sends the message.

The value of the quadrature counter is initially set to 110. This value changes when

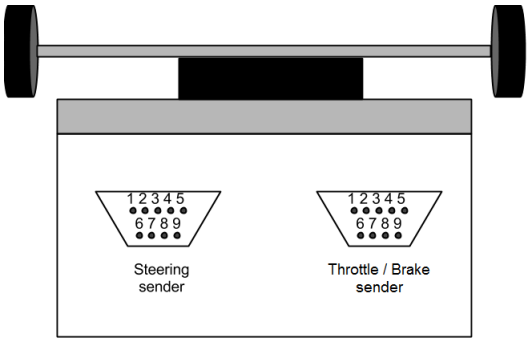

Figure 2.4: Connectors and wheels of the throttle and brake receiver ECU.

the A signal has a positive edge, as the A signal is used to generate interrupts. The interrupt handler reads the B signal from the magnet sensor. If the B value is equal to zero, the quadrature counter is increased. If the B value is greater than zero, the quadrature counter is decreased. In practise, this means that a quadrature counter value of 110 corresponds to a 0 degree steering wheel angle (straight forward), while a value greater than 110 corresponds to a clockwise rotation. A value less than 110 corresponds to a counter-clockwise rotation.

In parallel with the sender ECU, the receiver ECU waits for a CAN message to arrive. This is done in a busy-wait fashion. When it arrives, the value of the quadrature counter is converted into a BCD value which is written to the display attached to the receiver ECU. In addition, it scales the steering wheel angle ($-180\,^\circ$ to $180\,^\circ$) into an angle which makes sense for the wheels ($-44\,^\circ$ to $44\,^\circ$) and instructs the wheels to take the correct position.

2.2 The throttle and brake subsystem

Like the steering subsystem, the brake and throttle subsystem also consists of a sender ECU and a receiver ECU. However, instead of a steering wheel, two pedals are used as input sensors, and a toggle switch is used as a gear shift. The receiver ECU communicates with the system via two D-SUB connectors. The reason why it needs two D-SUB connectors is to make it possible for the CAN and power signals to propagate in the system. Like the receiver ECU in the steering system, the brake and throttle receiver ECU also controls a display and two wheels. The receiver ECU is illustrated in Figure 2.4.

The sender ECU, illustrated by Figure 2.5, includes two D-SUB connectors and one RJ10 connector. The D-SUB connectors are shorted with each other inside the interface box, in such a way that the pins of the first connector is connected with the corresponding pins of the second connector. This setup makes it possible for the power lines and the CAN bus to traverse to the next node in the system. The pin configuration of the D-SUB cables is the same as in the steering sender ECU, and can be found in Appendix A.

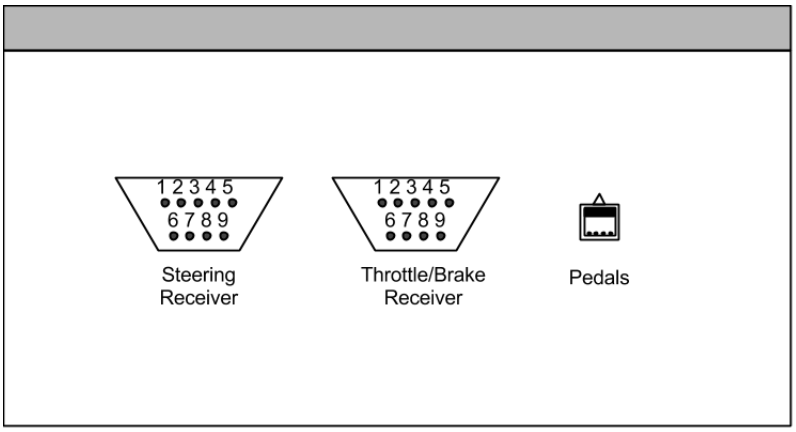

Figure 2.5: Connectors mounted on the throttle and brake sender ECU.

Table 2.3: Output signals from the pedals that are used by the sender ECU.

Pin name	**Type of signal**
Throttle value	Analogue
Brake value	Analogue

The RJ10 connector is used in order for the ECU to read the position of the pedals. The pedals are connected to one potentiometer each, which makes it possible to change the voltage level that is carried by the RJ10 cables. Thus, a certain voltage level corresponds to a certain pedal position.

Two output signals from the RJ10 connector are relevant for the software application in the sender ECU, and those are shown in Table 2.3. In addition to the pedals, the software application uses the toggle switch to decide the gear position (drive or reverse). The toggle switch sends out a single digital signal, which is high if the direction is set to reverse, and low if the direction is set to drive.

The software of the sender ECU consists of an initiation phase, where the LCD, the ADC, and the CAN peripherals are initiated. In addition, an A/D conversion is started. This is followed by a loop which works in a busy-wait fashion. At the beginning of the loop, a CAN-message is sent. After that, the LCD is updated if a counter has reached the value of 10 000 (every 10 000th loop revolution). This LCD routine only updates the default information on the display.

The next step is to check whether a new A/D value, coming from the throttle and brake signals, has arrived or not. A checkbit is used in order to know if a new value has arrived. When the A/D-conversion is finished, an interrupt is sent, and an interrupt handler sets the checkbit. When the checkbit is set and a new A/D value has arrived, the value is converted into a percentage. The software checks if the value came from the throttle or the brake by identifying the latest used A/D converter. The percentage, and

Table 2.4: Structure of the CAN message from the sender unit in the throttle and brake subsystem.

CAN message configuration	**Value**
Data (field 0)	Brake percentage
Data (field 1)	Throttle percentage
Data (field 2)	Direction (drive (0xFF) or reverse (0x00))
Identifier	0x01
Length	3 bytes

the direction (forward or reverse) if the value came from the throttle pedal, is written to the display and also put into a CAN message. The percentage value in the display is only updated every 150th loop revolution, while the CAN message, with the configuration shown in Table 2.4, is updated and transmitted every loop revolution. When the CAN message is transmitted, the A/D converter is restarted and the checkbit is unset. During one second the busy-wait loop laps about 150 times [11], which gives a loop period of approximately 6.7 milliseconds.

As for the steering subsystem, the receiver ECU of the throttle and brake subsystem executes in parallel with the sender ECU. After some initiations are made, a busy-wait loops waits for a CAN message to arrive. When a message arrives, throttle signals are sent to the wheels, and the brake value is sent to a servomotor. Also, the software reads the direction bit and sets an H-bridge to the correct configuration to make the wheels spin in the desired direction.

2.3 Evaluation

The system is functional, but several components could be modified to optimize the functionality, and to prepare the system for industrial implementation.

2.3.1 Distribution of CAN bus and power

The physical architecture solution is sufficient for the prior prototype, but if future modifications are to be done, several parts of the hardware need to be altered. One reason for this is that the distribution of the CAN network and the power sources is built as a chain, where each ECU has to be connected in a particular order. If they are not connected in the correct order, or if an ECU is disconnected, the consequence may be voltage drops, or that the CAN and power sources do not reach all ECUs. It may also result in interference on the CAN bus, as the bus terminations may be positioned at inappropriate places.

The unit that acts as a power supply includes two different power sources; a 5 V source with a maximum current of 2.1 A, and a 12 V source with a maximum current of 1.2 A. Both of them are distributed via a D-SUB cable along with ground. In addition, the 5 V supply is distributed via two plug cables, as the amount of current drawn by the components in the system is too high for the D-SUB cable to carry.

The power sources given by the central power supply are sufficient to support the prior prototype. However, if nodes with higher power consumption are added, neither the power supply nor the distribution are suitable as is. As the power supply limits the amount of voltage levels and current available, the D-SUB cables have a small cross-sectional area which limits the amount of current that can be distributed. Thus, in the future, both the power supply and the distribution method have to be upgraded.

The CAN bus is distributed using the same D-SUB cables as the power sources. This is not the ideal case since cables used for communication are commonly divided in twisted pairs, as illustrated by Figure 2.6, to cancel out electromagnetic interference (EMI). Since the prior prototype is small, this does not cause immediate problems. However, when the prototype is installed in an actual vehicle, the electromagnetic interference will be at a much higher level, which demands the cables to be arranged as twisted pairs.

Figure 2.6: Twisted pair.

In addition, EMI that interferes with the CAN bus may occur when the power supply

is upgraded to a more powerful alternative, as the amount of current in the cables will be higher. Thus, it is suitable to separate the CAN bus and the cables that carry the power.

2.3.2 Application code

The software of the ECUs have been developed ad-hoc with busy-wait loops. This results in a software architecture which is hard to overview, and it is also complex to add new functionality to the ECUs without disturbing the existing functionality. Even though the busy-wait loops provide a sufficient response time for the prior prototype, this might not be the case if additional functionality is added.

In the new prototype, the sender units will be merged into a centralized unit, and new functionality will be added in the future. This results in a more complex application code which is hard to maintain, alter, and analyze if built ad-hoc, and the need of a structured software platform is clear. Furthermore, in order to provide a way to speed up the execution, parallelization of the functionality is needed.

Also, the CAN identifiers are set arbitrarily. A systematic way of assigning identifiers would make the system easier to understand, especially when it grows larger. Furthermore, CAN identifiers are used to assign priority to the messages. This means that if the identifiers are set arbitrarily, the inherent priorities may cause problems for critical systems, as non-critical systems may have a higher priority. However, it is not possible to assign new identifiers during this project. This is because the functionality of the CCU in this project must be compatible with the old system, and hence, the CAN messages sent out by the CCU must look like the ones transmitted from the sender units. The task of assigning more suitable identifiers is therefore omitted until the receiver units are replaced.

In addition to the arbitrary identifiers, the speed of the CAN communication is set to 1 Mbit/s. That speed is not necessary in the prior prototype, and could be optimized in a later stage. However, as the future prototype might be altered in a way that puts a high load on the bus, this optimization is omitted until the system is complete.

3

Choice of platform

There are three main requirements on the new central unit. First, the system needs to include the old functionality of the two senders from the previous subsystems. Second, it should be extensible, both with respect to hardware and software. This means that the hardware platform needs to include a lot of resources to make it possible to add more sensors, actuators, and buses. This also means that the software must be structured in such a way that it is easy to add new functionality without the need of rewriting large parts of the existing code. Third, the new unit needs to be easy to understand in order to make it easy to use and alter it in the future.

In order to fulfill these requirements, both a hardware platform and a software platform had to be chosen. This was carried out by setting more detailed requirements for each platform, finding candidates, and choosing the ones that best fit the set requirements.

3.1 The hardware platform

The scope of this project states that PCB design shall be avoided. Therefore, the best solution is a development board already containing as much needed resources as possible. The resources required by the sensors and means of communication for the prior prototype are four digital I/O-pins, three ADC channels, and one CAN channel. Thus, more than four I/O-pins and three ADC channels are required in order to connect additional sensors to the system. Because it is impossible to guess how many I/O pins that might be needed for future sensors, as many I/O pins and ADC-channels as possible is preferred.

The prior prototype requires one CAN channel, which is sufficient to communicate with a lot of ECUs. However, one additional CAN channel is preferred to make it possible to physically separate future critical and non-critical CAN messages, as well as to offer

a way to implement redundancy of the CAN-bus. Also, a Local Interconnect Network (LIN) channel might be useful to relieve the CAN bus from communication that is less time-critical than the communication between the ECUs.

In addition to the these requirements, the microcontroller of the chosen development board has to be powerful enough to support both the functionality of the sender ECUs in the prior prototype, but also have the capacity to support a lot of new functionality. This includes both the speed of the CPU as well as the size of the internal memory.

3.1.1 Candidates

When the requirements were set, possible hardware platforms were considered. Several development boards were discovered, but only a few were investigated in more detail. The ones that best fitted the requirements are presented below.

The AVR UC3 Evaluation Kit

The AVR UC3C Evaluation Kit [12] is a development board designed to be able to act as a prototype for motor control. It is built with the microcontroller AT32UC3C0512C [13], which has a 32-bit AVR processor with a performance of 1.49 DMIPS [1] per MHz, running at a maximum frequency of 66 MHz. It has several digital I/O pins and 16 analogue input channels with a resolution of 12 bits. In addition, the microcontroller is equipped with two CAN channels and five LIN channels, 512 KB of internal flash and 256 KB of internal RAM. With these specifications, the microcontroller itself fulfills all the requirements listed for the hardware platform.

The kit is designed in a way which makes is possible for the user to test all the functionality in the microcontroller, using only the evaluation board. It is equipped with a lot of additional hardware, such as an audio jack, a microphone, LEDs, push buttons, a capacitive touch screen, and several other interfaces. The available CAN channels in the microcontroller are connected to CAN controllers and external connectors. In addition, two of the five LIN channels are available for use on the evaluation board.

The TMS570 Hercules Development Kit

Texas Instruments offers a microcontroller called TMS570LS3137 [14]. It is built around a 32-bit ARM Cortex-R4F microprocessor, which is running at a maximum frequency of 180 MHz, performing 1.6 DMIPS per MHz. It is designed to be used in safety-critical systems, such as braking systems and electric power steering, and uses two lock-stepped CPUs with error detection logic. The microcontroller is provided with three CAN channels, two FlexRay channels, two analogue-to-digital converters, one ethernet channel, and one LIN channel. In addition, it has several digital I/O-pins, and two 12-bit ADC modules, where the first module supports 24 channels, and the second module supports 16 channels, all of which are shared with the first module.

[1] Dhrystone MIPS, a benchmark for measuring general processor performance.

The development board, TMDX570LS31HDK [15], consists of the TMS570LS3137 microcontroller and peripheral hardware. Two of the CAN channels in the microcontroller are routed to CAN a controller and external connectors. Some of the digital I/O pins are connected to LEDs and pushbuttons on the board, but most of them are routed to extension connectors available for prototyping and custom made hardware. Most of the analogue channels are also routed to the extension connections. However, two of them are used for hardware mounted on the development board. Three extension connectors are available, consisting of a total of approximately 160 pins, directly connected to the microcontroller. This means that the user is free to use the same type of board in several projects with different objectives.

The TMS470M Hercules Development Kit

In addition to the microcontroller used in the TMS570 Hercules kit, Texas also offers a smaller and less complex microcontroller called TMS470MF066 [16], which has a 32-bit ARM Cortex-M3 processor, running at a maximum frequency of 80 MHz and performs 1.2 DMIPS per MHz. As for the TMS570 kit, it is designed for use in electric power steering and braking system. It is equipped with several digital I/O pins and one 16 channel analogue-to-digital converter with 10 bits resolution. Also, two CAN channels and two LIN channels are available, as well as 640 KB of internal flash and 64 KB internal SRAM.

The microcontroller can be used in development purposes, by using the development board TMDX470MF066HDK [17]. On the board, both of the CAN channels are connected to CAN controllers available through on-board connectors. Most of the digital I/Os and all of the analogue channels are connected to extension connectors, which makes it possible to connect external custom made hardware. As for the TMS570 kit, the board is designed to fit different kinds of project areas.

The LPC1768 Evaluation Board

The LPC1768 Evaluation Board [18] is equipped with a LPC1768 [19] microcontroller, designed by NXP. The LPC1768 is built on a 32-bit ARM Cortex-M3 processor, running at a maximum frequency of 100 MHz, performing 1.5 DMIPS per MHz. Moreover, it offers 512 KB of flash memory, and provides two CAN channels, a 12-bit analogue-to-digital converters with eight channels, a 10-bit digital-to-analogue converter, and one ethernet channel. LPC1768 is designed for use in applications such as alarm systems, lighting, motor control, and industrial networking.

LPC1768 Evaluation Board has connectors mounted for use of both CAN channels. The ethernet channel is connected to an RJ45 connector. Also, a prototyping area is available in order for the user to connect external custom designed hardware. Most of the digital and analogue I/O pins are routed to the prototype area. Moreover, a graphical LCD with a size of 240x320 pixels is mounted on the board.

3.1.2 Discussion of choice

In order to choose the best possible hardware candidate for the project, several aspects need to be considered. First, the requirements of the hardware platform must be fulfilled. Second, the hardware should be flexible for future modifications. For example, safety is one area which will probably be implemented in upcoming projects. Thus, an already safety-critical platform is preferable.

All the hardware candidates would function as a CCU. However, some boards are more suited than others, since they are intended to be used in different application areas. Therefore, the candidates were analyzed and compared against the hardware requirements. When doing so, some of the candidates were removed as possible choices. The LPC1768 is the only board which does not fulfill all of the requirements. Due to the fact that there is no possibility to use LIN on the board, the other alternatives are better suited. Atmel's development board, AT32UC3C-EK offers several LIN channels, dual CAN channels, and a powerful microprocessor. Although, the number of free I/O pins is not as high as on the other boards. Thus, AT32UC3C-EK was also omitted.

The Texas Instruments boards are similar in their physical architecture and equipped with the same types of components. Also, they are designed to be used in the same types of application areas, and either one of them would be a good choice. However, the microcontroller mounted on the TMS570 board provides two CPUs in lock-step. Since this feature adds safety features to the system, the lock-stepped CPUs are helpful in upcoming projects covering fault-tolerance. Thus, the TMS570 board was chosen to be the hardware platform of the CCU. This choice meant that a separate LCD display had to be purchased in addition to the development board, as it does not include an integrated display.

3.2 The software platform

As the two subsystems execute in parallel of each other, the software platform has to have a real-time operating system. The real-time behaviour is suitable because of many reasons, where the most apparent one is the fact that this system is meant to be used in a context where timing is important. A drive-by-wire system would be useless if the wheels do not respond to the sensors in the steering wheel or in the pedals in a given amount of time. A real-time system offers tools to make it possible to schedule the different tasks and make sure they finish in time. Furthermore, a real-time system often offers mutual exclusion mechanisms that simplifies the sharing of resources.

With the use of an operating system, the hardware-specific code becomes separated from the code that controls the actual functionality. This makes it easier both to identify the application code, and to add more functionality without rewriting hardware-specific code.

The idea of separating the application code from the hardware-specific code has been adopted by the Automotive Open System Architecture (AUTOSAR) standard. AUTOSAR is a development partnership of many companies that are connected to the automotive

industry. The partnership works together to establish an open industry standard for automotive electrical and electronic architectures. In order to separate the hardware-specific code from the application code, AUTOSAR consists of three main layers: Application Software, Runtime Environment (RTE), and the Basic Software (BSW) [20]. While the BSW and RTE are hardware-dependent, the Software Components (SWC) in the Application Software are independent of the specific hardware architecture. This is achieved with well-defined AUTOSAR interfaces, which make it possible to move SWCs from one ECU to another without re-writing the code.

Because AUTOSAR is a standard developed by the automotive industry, as well as a standard developed to separate the hardware-specific layers from the application-specific layers, it would be suitable for the software platform to support AUTOSAR. This is however not a requirement.

Sigma Kudos wants to keep the costs of the product down, and thus, the software platform needs to be cheap, or preferably follow an open source license. In addition, it is required that the platform is compatible with the target hardware platform, as the time frame of this project does not allow the work of porting an incompatible software platform.

Also, it would be suitable if the software platform contains some fault-tolerant and safety-critical properties, as this will be needed in the future.

3.2.1 Candidates

Candidates containing real-time properties and support for the microprocessor were chosen to be studied in more detail. Here, the four most promising candidates are described.

Arctic Core

Developed by Arccore, Arctic Core [21] is a software platform that is based on the AUTOSAR standard. The platform includes a real-time operating system, support for communication standards such as CAN and LIN, as well as drivers for a number of different microcontrollers.

Because Arctic Core is AUTOSAR compliant, its operating system is built on the OSEK OS standard [22]. The OSEK OS is a single-processor operating system that supports event-driven control systems. It uses static allocation of tasks, resources and services, and is capable of running on read-only memory. The scheduler is priority-based, and non, full or mixed preemptive scheduling may be used depending on the requirements of the system. It also provides protection mechanisms for resources, in order to guarantee mutual exclusion.

Arctic Core supports a number of different architectures, including ARM Cortex R4, which is the architecture of the chosen hardware platform. It also exists a port to the TI development board TMDX570LS20SMK, which is similar to the chosen TI development board TMDX570LS31HDK. However, some porting has to be done in order to be able to use Arctic Core.

For this platform, Arccore offers a dual license. All source code for Arctic Core that is available for download is licenced under GNU General Public License (GPLv2), as well as a commercial licence with the GPLv2 removed. Besides Arctic Core, Arccore also offers a set of tools available for Arctic Core. They include among others an IDE, a BSW builder, an RTE builder, and a SWC builder.

FreeRTOS

FreeRTOS is developed by Real Time Engineers Ltd [23], and is distributed under the GPL license. As for Arctic Core, Real Time Engineers Ltd also offers a commercial licence for FreeRTOS, named OpenRTOS. The main difference between these two alternatives is that OpenRTOS has a warranty and legal protection provided. Also, when using FreeRTOS, any changes made on the kernel has to be published. This is not the case with OpenRTOS.

FreeRTOS is designed to be small, simple, and easy to use, and according to the developers, a typical kernel library image is between 4 KB and 9 KB. The source code is easy to port, and officially supports 33 embedded system architectures, including ARM Cortex R4.

The scheduler of FreeRTOS can be either non-preemptive or preemptive, and uses priority scheduling. Synchronization mechanisms such as queues, mutexes, and semaphores are also included. In addition, FreeRTOS has a collaboration with Texas Instruments, and it is possible to generate FreeRTOS with the HAL Code Generator Tool (HalCoGen), which also generates drivers compatible with the TMS570 microcontroller.

SafeRTOS

Derived from FreeRTOS, SafeRTOS is a operating system from WITTENSTEIN high integrity systems (WHIS) [24]. Although FreeRTOS and SafeRTOS share a functional model and run on the same class of microcontrollers, they are not the same. SafeRTOS only offers commercial licensing, is especially developed for critical applications, and is developed in compliance with the IEC61508 SIL3 standard. Like FreeRTOS, it offers a small footprint of typically 7 KB to 14 KB. It also supports the TMS750 microcontroller.

μC/OS-II

Developed by Micrium for safety-critical systems, and compliant with the IEC61508 SIL3 and SIL4 standard, μC/OS-II [25] is a real-time kernel that is scalable and portable. Its typical footprint is 5 KB to 25 KB. It uses a preemptive scheduler and allows one task per priority level. Synchronization mechanisms such as semaphores, mailboxes, and queues are also available. The kernel supports the ARM Cortex R4 architecture, but not the TMS570 microcontroller. Due to this, some porting is required in order to fit the hardware platform.

3.2.2 Discussion of choice

From a safety perspective, SafeRTOS and μC/OS-II are the strongest candidates. They both comply with the IEC61508 standard, and are developed for safety-critical systems. However, both of them require a commercial license, resulting in costs which fall outside the scope of this project. Thus, SafeRTOS and μC/OS-II are excluded, leaving FreeRTOS and Arctic Core as possible candidates.

Arctic Core and FreeRTOS are different at many points. Compliant with AUTOSAR, Arctic Core is standardized for the automotive industry. This makes Arctic Core a future-proof choice, as many car manufacturers are involved in the AUTOSAR project. FreeRTOS does not rely on any standard, and thus, Arctic Core is more in line with the automotive industry.

Furthermore, AUTOSAR is developed with distributed systems in mind. The modular structure of AUTOSAR makes it possible to reuse application code, and also move applications from one ECU to another one. This may be used at a later development stage of the prototype in order to optimize the physical placement of the functionality. This assumes however that all ECUs in the system are following the AUTOSAR standard.

Even though Arctic Core has many advantages, its complexity give rise to some disadvantages. The GPL distribution of the platform only includes basic functionality, and is merely a template. Without the additional tools offered by Arccore, many parts, including the RTE, have to be implemented manually. To do this, including connecting the SWCs to the RTE in a way that is compliant with AUTOSAR, requires good knowledge of AUTOSAR, and is also time consuming. In addition, some porting of the existing versions of Arctic Core is needed in order to make the platform compatible with the selected hardware. These manual alterations of the existing platform code are too time consuming to fit the time scope of this project. A way to work around most of these alterations would be to buy the necessary tools from Arccore, but the budget of the project prevents this purchase.

Furthermore, an objective of this project is to make the new platform easy to use. If the tools which make the platform easy to use cannot be obtained, the threshold of using and understanding the platform will be large.

As a contrast, FreeRTOS is developed to be easy to understand, without the need of additional tools. Most of its complicated features are explained on their homepage, and manuals which are in the scope of the budget are available for purchase. Despite the fact that FreeRTOS is light-weight and less modular than Arctic Core, it includes all required functionality. Tools included with the selected hardware platform are also compatible with FreeRTOS, which makes it easy to adapt the operating system to the needs of the system.

Even though Arctic Core is a better theoretical choice, its drawbacks on the practical side and the scope of this project makes FreeRTOS a more suitable choice. Because of this, FreeRTOS was chosen to be implemented as the software platform.

4

Implementation

When the decision of hardware and software platform had been made, the unit was implemented with the aim to make it easy to understand the physical architecture as well as the functional architecture. It is important that it is possible to connect the CCU to the prior prototype, but it should also be easy to restructure the physical architecture to fit another system. This is due to the fact that the components from the prior prototype will be replaced at a later stage.

In order to satisfy the requirement that the new system should be backward compatible with the prior prototype, a remap of all the signals was necessary. This was done using a separate interface box. Inside the interface box, signals from the sensors, the display, the power supply and the receiver ECUs are bundled together, forming a single connector to the CCU enclosure. A simplified view of the physical architecture is shown in Figure 4.1.

When the signals from the sensors arrive at the development board, the real-time system reads the signals, processes them, and transmits appropriate CAN messages to the receiver ECUs. The real-time system also updates the LCD in the display box with appropriate values. In addition, the display box contains a toggle switch to represent the reverse and forward gear. A digital signal from the toggle switch tells the real-time system which gear is currently active.

4.1 Functional architecture

The responsibility of the CCU is to act as a gateway between the sensors, the display and the CAN bus. The CCU has to interpret the signals from the sensors, and convert them into something that makes sense for the receiver ECUs and the display. When this is done, the CCU must pass the interpretations forward. A simplified functional architecture is illustrated by Figure 4.2.

As the FreeRTOS API is based on tasks, the functional architecture needs to be

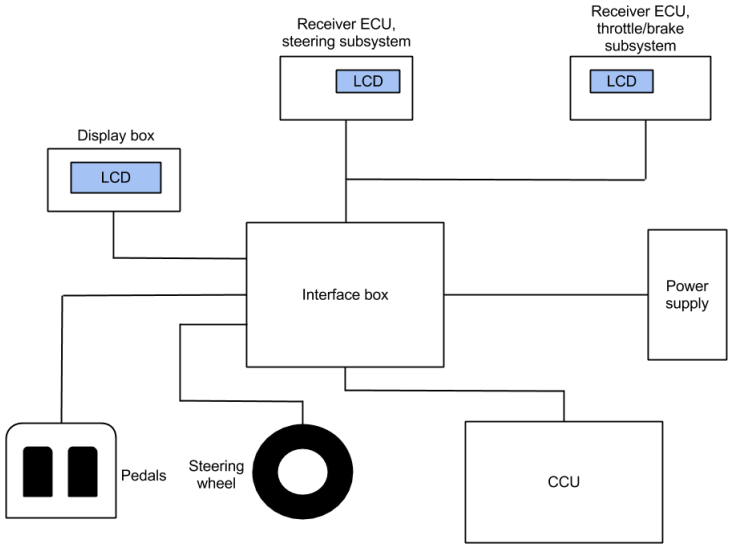

Figure 4.1: A simplified view of the new physical architecture.

allocated into well-defined tasks which communicate with each other. Also, because the different tasks are being executed in a pseudoparallel manner, all common resources that require mutual exclusion must be identified and protected in such a way that only one task at a time is able to access the resource.

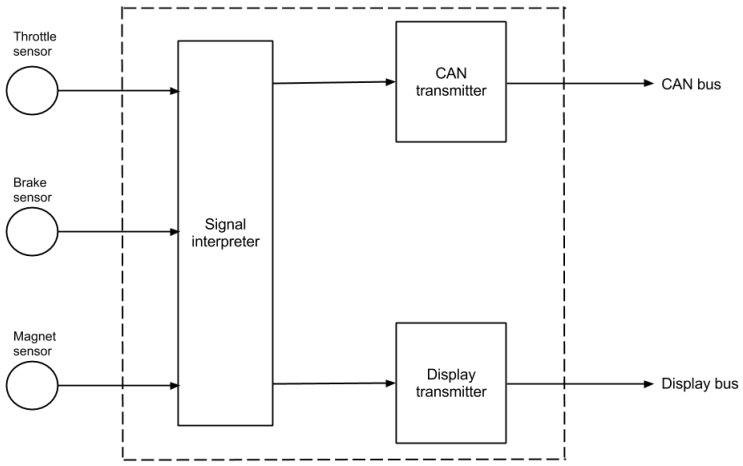

Figure 4.2: A simplified functional architecture of the CCU.

The task allocation must also take into account that the three different sensors transmit their signals in different ways. The throttle and brake sensors each transmit an analogue signal which can be read at any time. However, the magnet sensor attached to

Table 4.1: An overview of the different tasks and how they are triggered.

Task name	**Functionality**	**Activation**
init_system_task	Initiates peripherals.	Single
throttle_task	Reads current value of the throttle sensor from the A/D converter, interprets it, and passes it forward to the LCD_task and CAN_task.	Periodic
brake_task	Reads current value of the brake sensor from the A/D converter, interprets it, and passes it forward to the LCD_task and CAN_task.	Periodic
steer_task	Reads the A and B signal from the magnet sensor, interprets them, and passes it forward to the LCD_task and CAN_task.	Interrupt
LCD_task	Reads interpreted values and writes them on the display.	Periodic
CAN_task	Reads interpreted values, puts them in CAN messages, and transmits them.	Periodic

the steering wheel transmits digital signals which need to be read using interrupts.

In addition, to make it possible for the microcontroller to interact with all required ports on the development board, peripheral drivers for the involved modules have to be initiated. The required modules are the CAN module to send CAN messages, the A/D converter to interpret the signals from the throttle and brake sensors, and the general input/output (GIO) module to read the signals from the magnet sensor, and also to control the LCD display.

A task allocation which takes all these observations into account is shown in Table 4.1. Six tasks are identified, where four are periodic, one is interrupt driven, and one is only executed once.

Before any of the tasks are created, the peripheral drivers that have the closest relation with the hardware are initiated. When this is done, the init system task initiates everything that has to do with the functionality, such as the LCD and the home position of the steering wheel, before it spawns all other tasks. When the other tasks are spawned, the init system task is terminated, and the system is online. This behaviour is shown in Figure 4.3, where the system is divided into two phases: the top one, which is the initiation phase and consists of peripheral initiations and the init system task, and the bottom one, which is the periodical phase and consists of all other tasks.

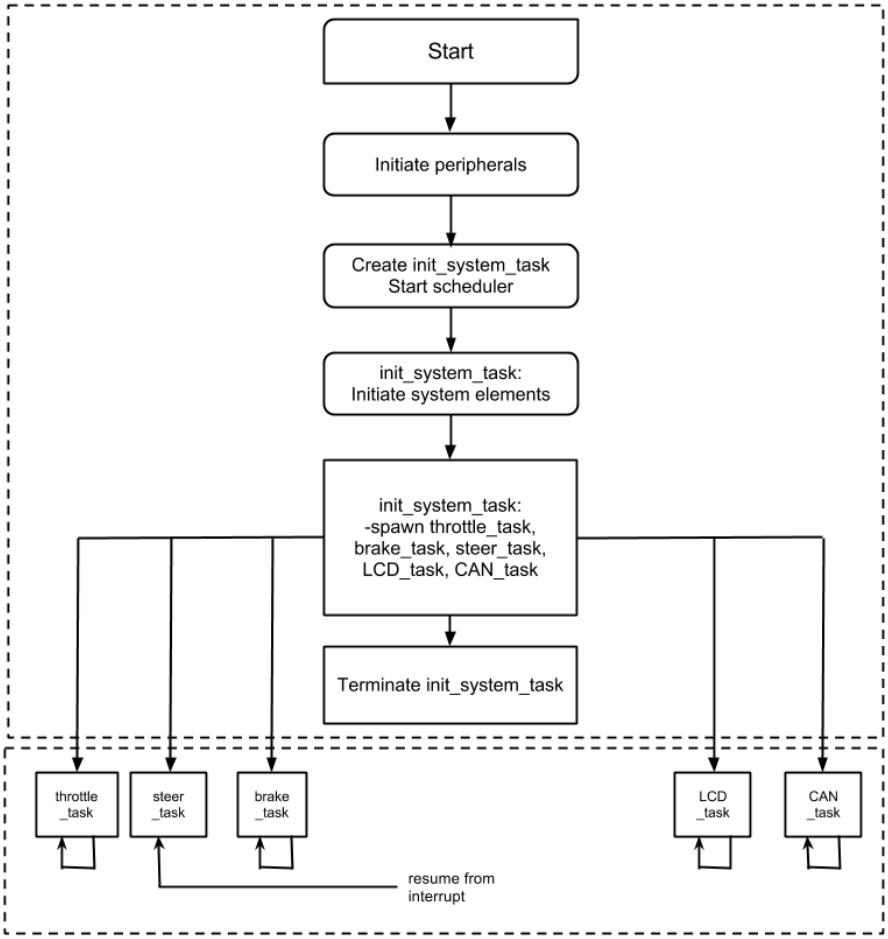

Figure 4.3: An overview flowchart of the functionality of the system.

There are four different kinds of resources available to the tasks in the periodical phase. The first kind is the ADC module, which is used to convert analogue signals from the pedals. This resource is used in the throttle task and interrupt task, and thus requires mutual exclusion. This is achieved using semaphores, which is closer described in subchapter 4.1.2.

The second and third kinds of resources available are the LCD display and the CAN module. However, these resources are only used by one task each; the LCD task and the CAN task, respectively. Because of this, no other task will interfere during the use of the resources, and they do not have to be protected. However, if other tasks which require access to these resources should be added, they need to be protected by a mutual exclusion mechanism.

The fourth kind of resource is variables which are used for communication between the tasks that read the sensors, and the tasks that use the values of the sensors. As seen in Figure 4.4, the throttle, brake, and steering tasks write values to one variable each.

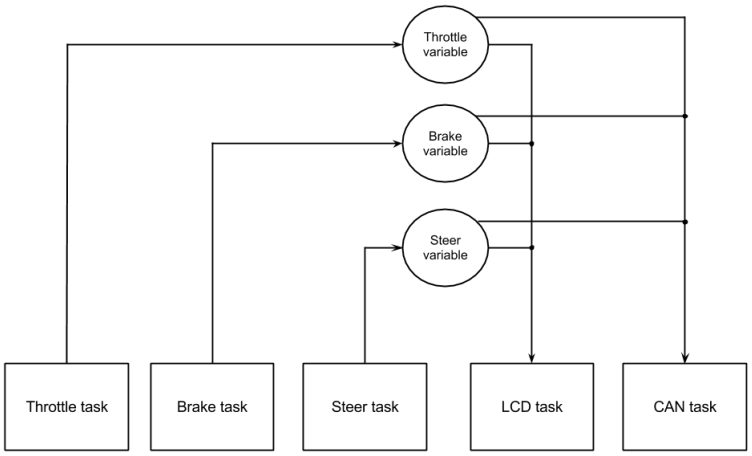

Figure 4.4: How the variables are used in the system.

These values are then read by the LCD and the CAN task. This means that all variables have one task which writes to them, and two tasks which read from them, which means that no mutual exclusion mechanisms are needed. However, as for the LCD and the CAN resources, protection mechanisms have to be included if the variables are to be changed by more than one task.

As an overview of the tasks and how they communicate now has been given, the subchapter continues by explaining the different tasks in further detail.

4.1.1 Initiation

The initiation of the system is divided into two parts, as the peripherals for the GIO module, the ADC module, and the CAN module can be executed before the scheduler starts. However, the initiation of the LCD requires timing behaviour as a certain time has to pass between commands to the display are sent [26]. To simplify the implementation and make use of the functions from the FreeRTOS API, the initiation of the LCD takes place in the initiation task. In order to follow the same pattern and get a structured flow, all initiations directly linked to the functionality are made in the initiation task. This include initiation of variables, finding a home position for the steering wheel, and spawn the interrupt driven and periodic tasks. A flowchart of the initiation task is shown in Figure 4.5.

The execution of the initiation task begins with an initiation of the LCD. This includes writing basic information to the LCD, which consists of a static template. This template, as seen before the LCD task is spawned, is illustrated in Figure 4.6. The patterns of the six positions at the top-right hand of the screen forms the company logotype. These patterns are saved in the memory of the LCD module during its initiation phase.

After the LCD is initiated, variables for the throttle value, the brake value and the quadrature value are initiated to be zero, zero, and 110 respectively. These variables are explained in more detail later in this subchapter. In this phase, also semaphores used

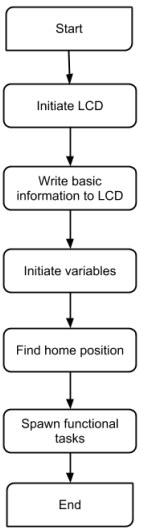

Figure 4.5: A flowchart of the initiation ask behaviour.

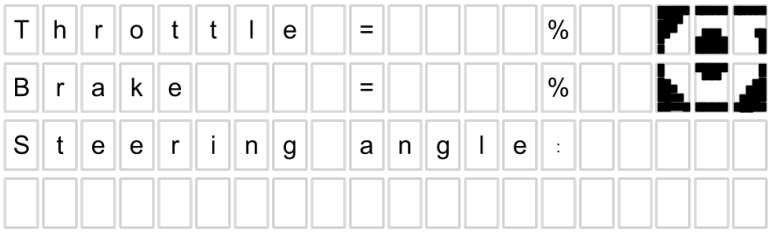

Figure 4.6: An illustration of the basic information template of the LCD display.

for the mutual exclusion mechanism for the ADC module is initiated.

The task finds a home position for the steering wheel by using a busy-wait loops which polls the A, B, and index signal from the magnet sensor. When all these three signals are high, a home position has been found and the busy-wait loop is terminated. When this happens, the system is fully initiated, and the functional tasks are spawned. After this, the initiation task is no longer needed, and hence, the task terminates itself.

4.1.2 The brake and throttle functionality

As observed in Table 4.1, the brake task and throttle task perform similar execution, and may be described by the same flowchart, as seen in Figure 4.7. The difference between them is that they read from different ADC channels, in order to get a value for the brake pedal or the throttle pedal. When the ADC value from a channel, either the one used for throttle or the one used for brake, is read, it is compared to some constants. As the ADC module is set to 12-bit mode, and the result of the A/D conversion returns a

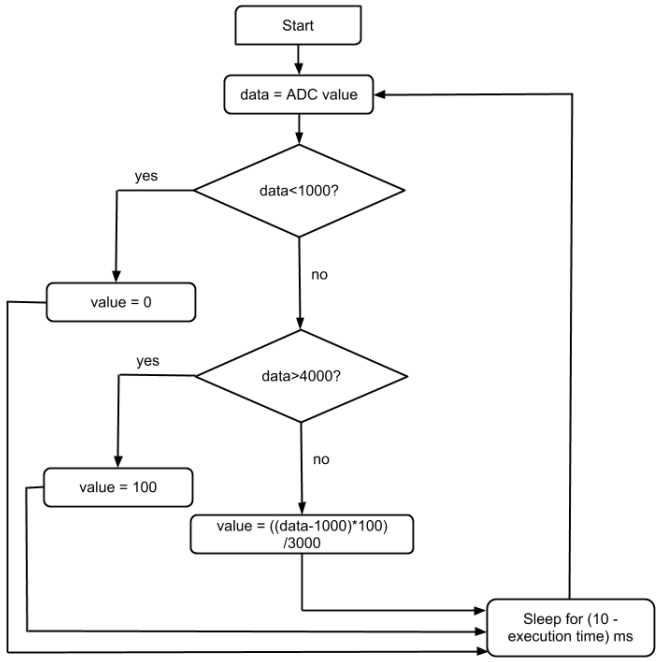

Figure 4.7: A flowchart of the behaviour of the throttle and brake tasks.

value between 0 and 4096. Empirical test cases showed that for A/D results lower than 1000, the brake and throttle percentage values should be 0%. Similarly, an A/D result higher than 4000 indicates a full brake force of 100%. In cases between 1000 and 4000, the value of the brake force is linearly calculated as a percentage value. The calculated percentage value is placed inside the variable that can be read by the LCD task and the CAN task. When this has been done, the tasks are suspended for about 10 milliseconds, before they are awoken and the procedure repeats itself.

It is important to note that both the throttle task and the brake task use the ADC module. When the ADC module has begun converting a given channel, it cannot be used by another task before it has returned a value to the task who accessed the ADC module first. This means that mutual exclusion is needed for the ADC module. To solve this, a mutex semaphore is used. In order to start a new A/D conversion, the task has to obtain a semaphore. If the semaphore is obtained, the task is allowed to start the A/D conversion. When the conversion is finished, the semaphore is released. If the semaphore cannot be obtained, however, the task is blocked until the semaphore is free.

4.1.3 The steering functionality

When it comes to the steering functionality, the inputs from the magnet sensor require interrupts. This is because, as explained in chapter 2, the signals A and B are two pulse trains which are shifted 90 °. Depending on if A leads B or vice versa, the steering wheel turns in different directions. A high-triggered interrupt is set on signal A. If B is zero

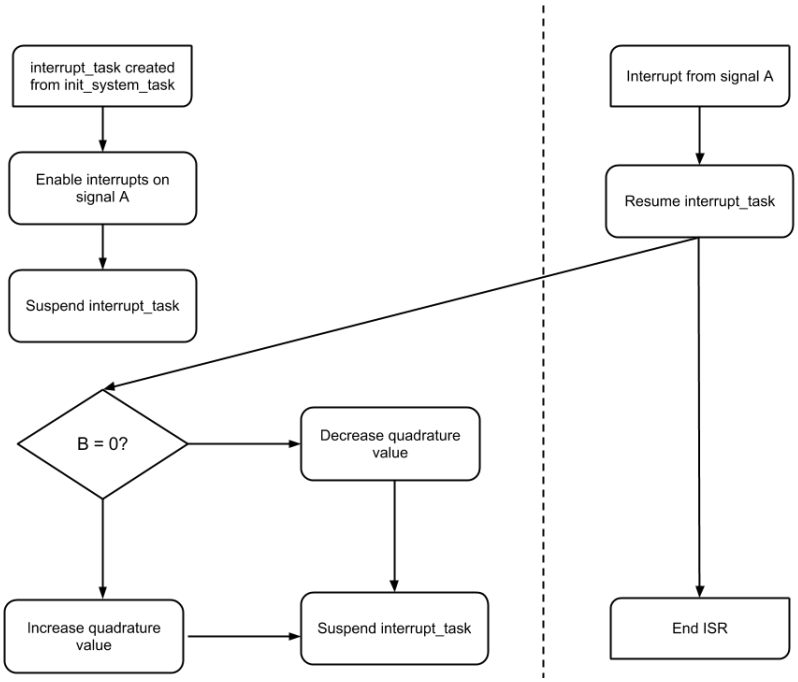

Figure 4.8: Illustration of the steering task behaviour, with the interrupt task on the left side, and the ISR on the right side.

when the interrupt is triggered, the steering wheel was turned clockwise. Conversely, if B is one, the steering wheel was turned counter-clockwise.

In order to know in what direction and with what angle the steering wheel is turning, a variable called the quadrature value is initiated to be 110, which marks the middle of a scale ranging from 0 to 220. This is set at the home position in the initiation phase. This variable is increased by one if the steering wheel is turned clockwise, and decreased by one if the steering wheel is turned counter-clockwise. In this way, values above 110 indicates a clockwise turn, while values below 110 indicates a counter-clockwise turn.

Unfortunately, FreeRTOS does not include an integrated interrupt handler. In order to integrate the interrupts with the operating system, the system may either use semaphore calls, queue calls or resume task-calls from the ISR. In this case, it makes most sense to use the resume-task alternative. This means that the steering task consists of one task named interrupt task, as well as an ISR handler. The connection between them is illustrated in Figure 4.8.

The initiation spawns the interrupt task, which enable interrupts on signal A before it suspends itself. When an interrupt from signal A is handled by the ISR, the ISR awakens the interrupt task, which enters an infinite loop. The interrupt task checks the value of B, corrects the quadrature value, and falls back into suspended mode.

4.1.4 The CAN task

The variables written by the throttle, brake, and steering task are read by the CAN task and transmitted onto the CAN bus. This is done periodically, with the same frequency as the throttle and brake tasks. The use of a variable as the link between the CAN task and the other tasks ensures that always the latest recorded value is transmitted to the receiver ECUs. Another way to implement the communication between the tasks could be to introduce a queue on which the tasks append the variables. This solution, however, may lead to the result that the CAN task transmits outdated values if the CAN module of the microcontroller is slow. In addition, if the task is activated by the presence of new values in the queue, it may be the case that the CAN task consumes a large part of the total execution time of the CPU. To avoid this, only the latest updates of the values are transmitted every time the task is executing.

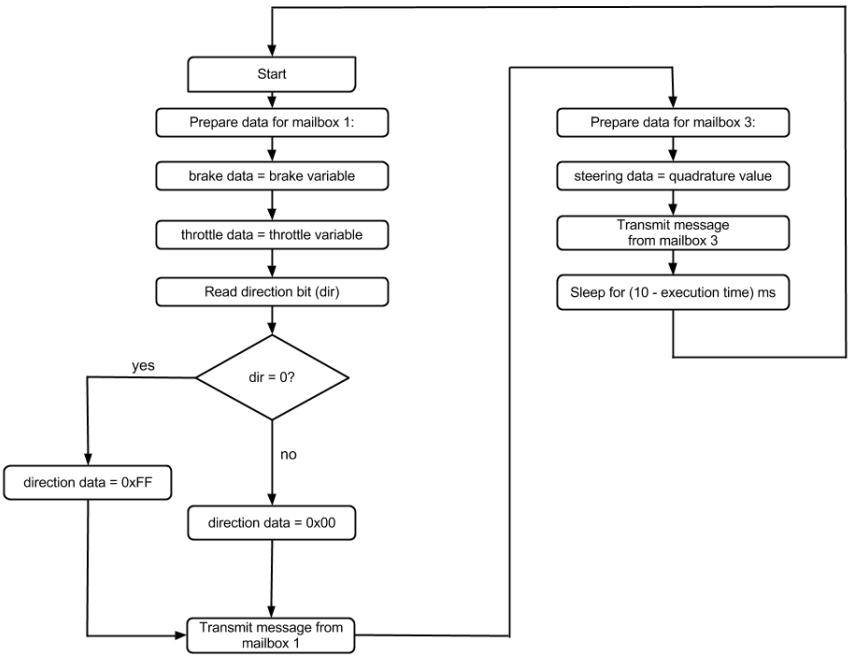

Figure 4.9: A flowchart of the CAN task functionality.

As seen in Figure 4.9, the CAN module of the microcontroller uses two different mailboxes to handle the different messages. The CAN module of the microcontroller is structured in such a way that mailboxes with static configuration are initiated together with the CAN module. The configurations include, among others, the possibility to set an identifier, a mask, if the mailbox handle transmit or receive functionality, and a data size. To transmit a message with the specified configuration, the application code has to specify what CAN channel, what mailbox, and what data to use. This structure simplifies the handling of many messages, as the application code only has to specify the mailbox of the message, instead of other configurations. The settings of the mailboxes

Table 4.2: The configuration of the mailboxes used in the CAN task. Mailbox number one is used to transmit throttle and brake messages, while mailbox number three is used for the steering messages.

Mailbox number	**Standard/ Extended identifier**	**Identifier**	**Mask**	**Transmit/Receive**	**Size (bytes)**
1	Standard	0x01	0x7FF	Transmit	3
3	Standard	0x12C	0x7FF	Transmit	1

used in this task is specified in Table 4.2, and are chosen to fit the prior prototype.

When the data from the throttle, brake, and steering task are added to the messages, and the messages are transmitted, the CAN task is suspended for approximately 10 milliseconds.

4.1.5 The LCD task

In addition to the CAN task, also the LCD task uses the three variables updated by the throttle, brake and steering task. The LCD task is periodic, but in contrast to the other periodic tasks which has a period of 10 milliseconds, the LCD task has a higher periodic time of 250 milliseconds. This is because it is not necessary to update the LCD as often as it is necessary to update the sensor information and the CAN messages. A flowchart of the LCD task is available in Figure 4.10.

In order to convert the quadrature value given from the interrupt task, Equation 4.1 is used.

$$Q_{degree} = C \times (Q_{value} - Q_{zero}) \tag{4.1}$$

Q_{value} is the value updated by the interrupt task, while Q_{zero} is the quadrature value where the neutral position is, which is placed slightly to the right from the middle point of the steering wheel. The constant C is a scale factor which scales the number of steps in one revolution of the magnet sensor to the corresponding degree value of one step.

After the execution, when the task is suspended, the information on the display is updated with the latest values. Figure 4.11 gives an example on how this can look. In this example, the vehicle has a throttle value of 35%, the gear is in drive mode, and the direction is 22 ° to the right.

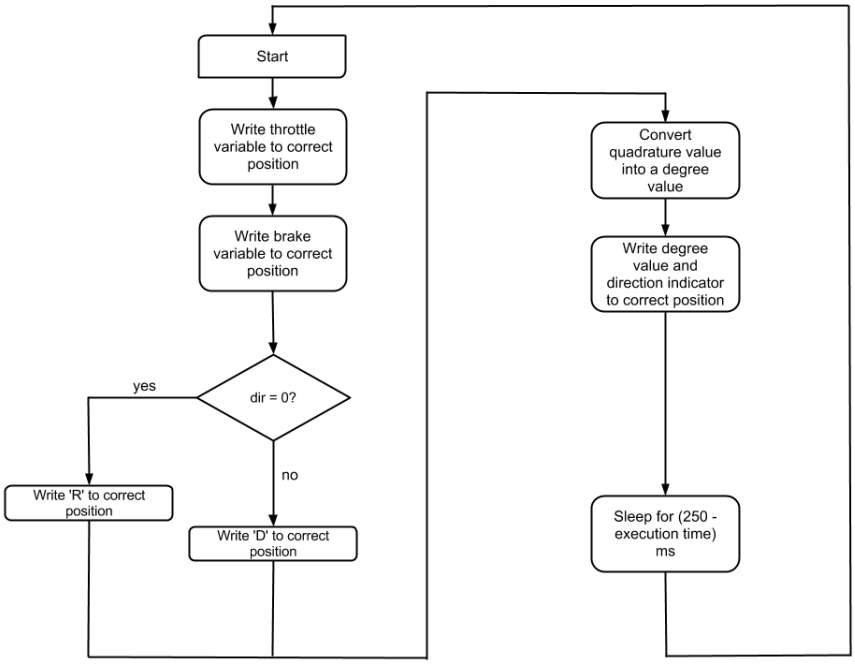

Figure 4.10: A flowchart of the LCD task.

Figure 4.11: An example of the LCD display during execution.

4.2 Physical architecture

As seen in Figure 4.1, the hardware in the system consists of several components; a power supply, a CCU, a display box, an interface box, the steering receiver ECU, and the throttle and brake receiver ECU. The power supply and the receiver ECUs consist of the same modules as in the prior prototype, why they are not described in detail. However, the CCU box, the display box, and the interface box were developed during this project and are described in this subchapter.

4.2.1 The display box

The display box consists of an LCD display which shows the user relevant system information, and a gear shift which makes it possible for the user to change the direction of the driving wheels. Figure 4.12 illustrates the connection and the components of the display box.

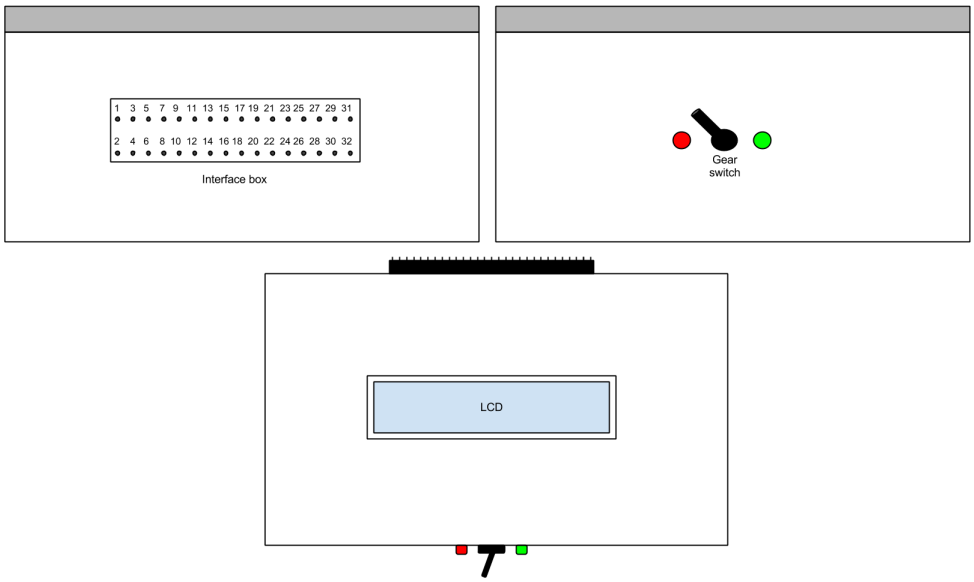

Figure 4.12: All sides of the display box.

The display is a four-row, twenty character-wide, alphanumeric display, designed by Display Elektronik [26]. It has eight data bits which are used for initiation, commands and data communication. It is possible to operate the display in either eight-bit mode or four-bit mode. Since the requirement specification of this project states that the CCU shall have several free I/Os, the four-bit mode was chosen. In addition to the four data bits, three LCD control signals need to be connected to the CCU, resulting in a utilization of seven pins.

In order for the LCD to properly display its content to the user, the backlight and the contrast of the display need to be calibrated. Both are adjusted using analogue voltages. The voltage level applied for the backlight configuration is directly proportional to the

backlight level of the display. For simplicity reasons, the backlight input was connected to the supply voltage of the LCD, resulting in a maximum level of backlight. However, the same method is not applicable for the contrast configuration, because the contrast varies depending on the ambient temperature. Thus, a circuit consisting of a potentiometer was installed in order for the user to adjust the contrast when needed.

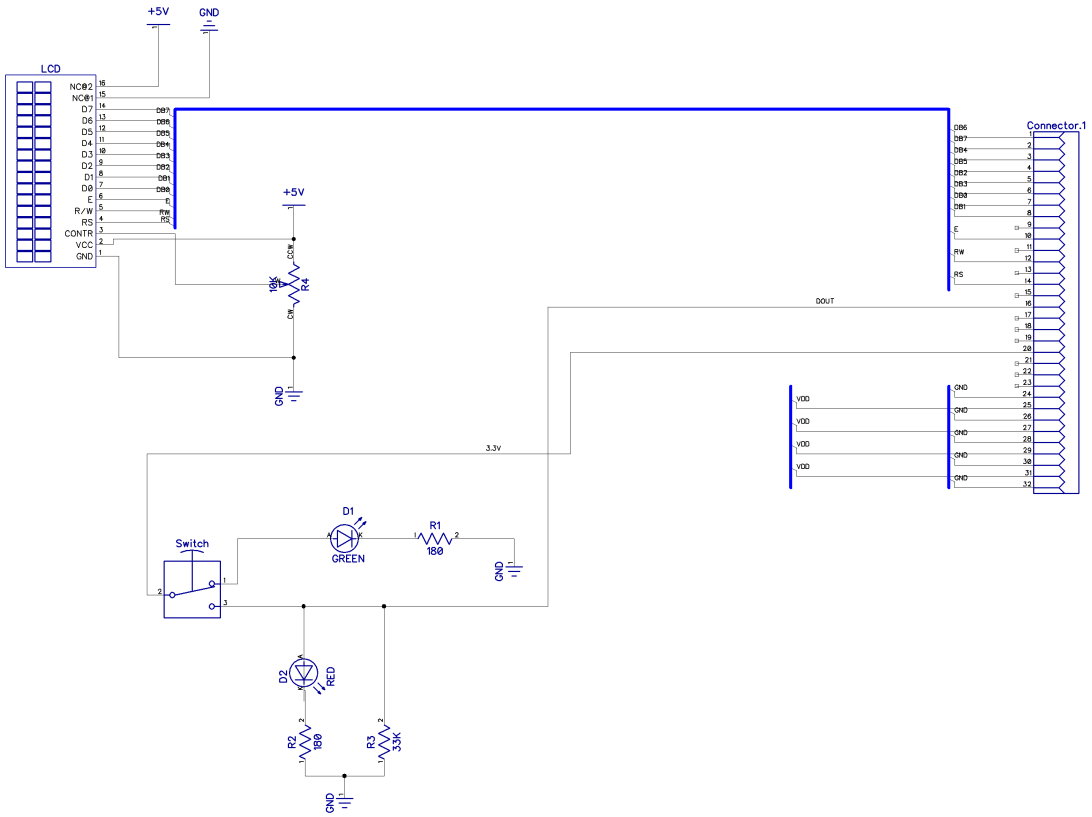

Figure 4.13: Schematic view of the display box.

The gear switch positioned on the display box makes it possible for the user to shift between drive and reverse. The shift mode is represented by a green LED, positioned on the right side of the switch, and a red LED, positioned on the left side of the switch. When the green LED is lit, the gear is set to drive, and when the red LED is lit, the gear is set to reverse. The CCU determines which gear is set by reading the digital representation of the voltage applied over the red LED.

When the gear is set to reverse, current flows through the red diode to ground. In order to limit the current and protect the LED from breakage, a resistor is positioned between the diode and ground. In addition, there is one extra resistor positioned in parallel with the red diode. This resistor acts as a pull-down resistance when the gear switch is set to drive. If no pull-down resistance is present, the gear signal might be set to either a logical one or a logical zero, as the pin is floating. When the gear switch is

set to drive, current will flow through the green LED. Hence, an additional resistor is needed between the green LED and ground to protect the LED from breakage. Figure 4.13 illustrates the internal connections of the display box.

4.2.2 The CCU box

The CCU box holds the Hercules development board. On the back of the development board three expansion connectors are positioned, all consisting of pin sockets routed to the microcontroller. These are used in order to connect the board to the rest of the system. Figure 4.14 illustrates the connectors mounted on the back of the development board.

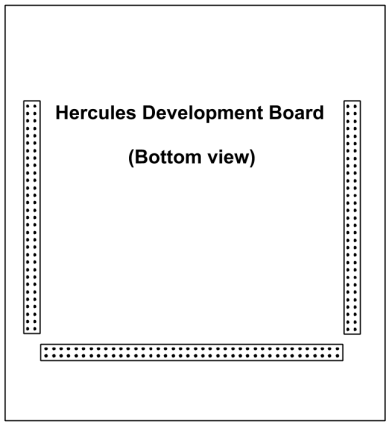

Figure 4.14: Expansion connectors mounted on the back of the development board.

In addition to the development board, the box contains three industrial connectors which are mounted on the sides of the box. One of them powers the CCU and connects it to the rest of the system. The other two are left unconnected, but are available for use in upcoming projects. Furthermore, a mini-USB-to-USB cable is connected to the board and extended through the box, where it can be used to program the CCU. Also, an ethernet connector is mounted on the box, which is extended from the CCU. The ethernet connector is currently unused, but is available to be used during upcoming projects. Figure 4.15 illustrates the connectors positioned on the CCU box.

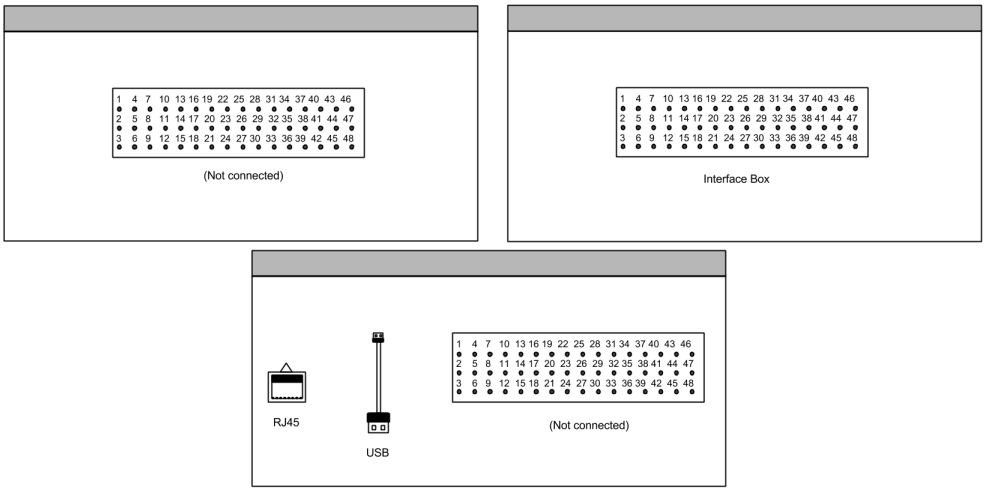

Figure 4.15: The connectors of the CCU box.

4.2.3 The interface box

The interface box was created in order to connect the CCU with the rest of the system. It consists of several different connectors, making it possible for all components of the system to be interconnected. Figure 4.16 illustrates the interface box and its different connectors. The pin configuration and the pin setup of all the connectors can be found in Appendix A.

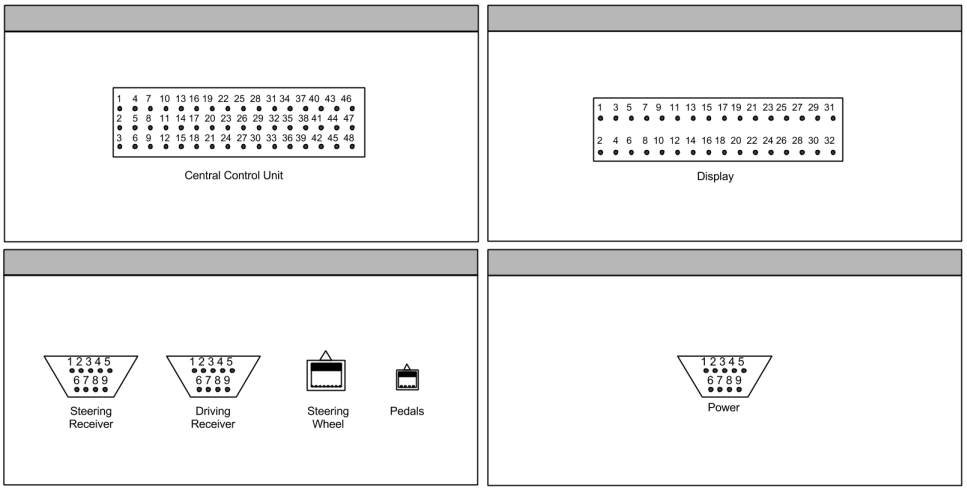

Figure 4.16: The four sides of the interface box.

There are two industrial connectors positioned on the interface box: one 48-pin connector and one 32-pin connector. The 48-pin connector powers the CCU and connects it to all system components, including the CAN bus, while the 32-pin connector powers the

display box and connects it to the CCU. Furthermore, there are three D-SUB connectors mounted on the interface box. Two of them power the receiver ECUs and connect them to the CAN bus. The third D-SUB connector is used to distribute the power supplies from the power box to the system. It delivers the 5 V source, the 12 V source, and ground to the interface box, which in turn distributes the power nets to the rest of the system.

In addition to the industrial connectors and the D-SUB connectors, one RJ10 connector and one RJ45 connector are mounted on the interface box. The RJ10 connector is used in order for the pedals to be connected to the CCU, while the steering wheel is connected to the CCU via the RJ45 connector.

5

Resource utilization and response time analysis

As the CCU developed in this project will be the starting point for other projects, it important to analyze how much resources the current implementation use, and thus, how much additional functionality that could be added. The resources include the physical I/O pins, the A/D channels, and the CAN channels, as well as the processor and the memory.

Another important aspect of the system is the response time, which is the time it takes from a sensor value is given until an actuator responds to the input given from the sensor. This is important because an automotive system would be useless unless the throttle, brake and steering functions respond in a given amount of time from the pedals are pressed or the steering wheel is turned. This is explained in further detail in this chapter.

5.1 Processor load

In order to analyze the tasks and the processor load, a Tracealyzer from Percepio [27] was used. The tool is developed for FreeRTOS, and makes it possible to view how the tasks behave during execution. Figure 5.1 shows a Gantt chart extracted by the Percepio tool, recoding just over the first second of the execution.

The chart shows that the tasks behave as expected: at the beginning, only the initiation task is active. After all the initiations are made, the initiation task spawns the other tasks, where the throttle, CAN, and brake tasks display the same periodic behaviour. The LCD task also shows a periodic behaviour, but with a longer period than the other tasks. In addition, the chart shows the aperiodic behaviour of the interrupt task, which is driven by an arbitrary triggered interrupt signal.

However, when examining the periods of the tasks closer, all periods are half the

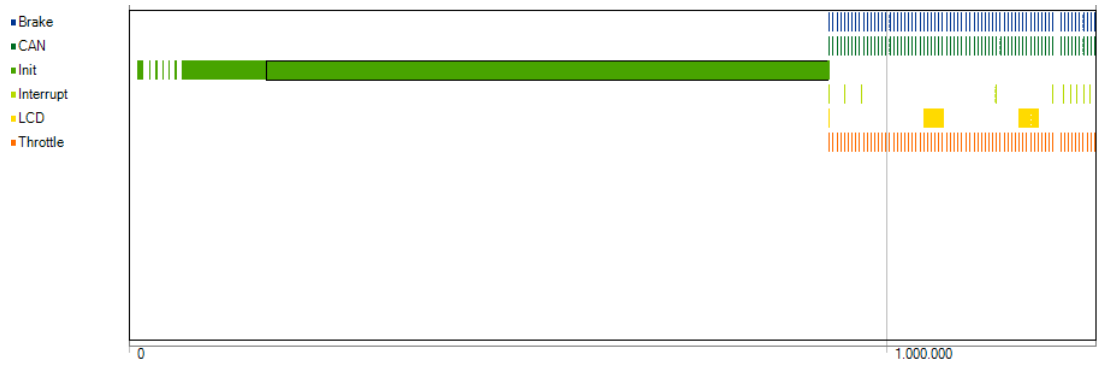

Figure 5.1: A Gantt chart of the tasks in the application code.

length they should be. The error is consistent: the brake, CAN, and throttle tasks, which all should have a period of ten milliseconds, appear to have a period of five milliseconds. In addition, the LCD task assigned to have a period of 250 milliseconds appears to have a period of 125 milliseconds. In order to locate the source of the error, the CAN task is examined closer using Figure 5.2.

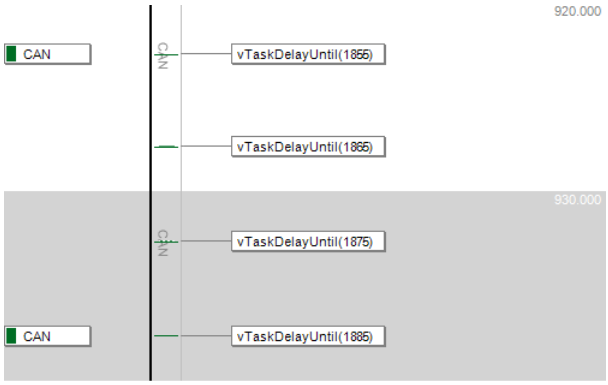

Figure 5.2: The periodic behaviour of the CAN task.

The numbers to the right in Figure 5.2 shows the time base in milliseconds, and the separation between each CAN task instance is five milliseconds according to the time base. The periodic behaviour is created by the function vTaskDelayUntil(), which is executed each instance. The input parameter of the function is given in ticks and corresponds to the internal operating system time when the task should be awaken. In this application code, the configuration should be set such that one tick corresponds to one millisecond. Here, the error is evident: as the separation between the input parameters are ten ticks, the analysis tool should show a period of ten milliseconds instead of five milliseconds. Supposing that either the tick base or the analysis tool is the source of the error, both of them have to be investigated.

To generate ticks, the operating system uses the real-time interrupt (RTI) module integrated with the microcontroller, where the frequency of the RTI clock source is 90

MHz. This clock source is in turn used to generate a frequency for a free running counter which is set by using a prescaler on the clock source. In the case of FreeRTOS, the prescaler is set to be two, which gives a free running counter frequency of 45 MHz. In order to generate interrupts, one compare register and one update compare register are set. The compare register compares its value with the free running counter. If they match, an interrupt is flagged and the value of the update compare register is added to the compare register, giving a tick frequency which match the value of the update compare register.

The values of the compare and the update compare registers are dependent on two configurations set by the FreeRTOS user, namely the frequency of the RTI clock source, configCPU_CLK_HZ, and the tick rate, configTICK_RATE_HZ. The values of the compare register, c_{reg}, and the update compare register, uc_{reg}, are initially set to the same value, as shown in Equation 5.1. Here, f_{clk} is equal to configCPU_CLK_HZ, and f_{tick} is equal to configTICK_RATE_HZ.

$$c_{reg} = uc_{reg} = \frac{f_{clk}/2}{f_{tick}} \tag{5.1}$$

In this project, configCPU_CLOCK_HZ is set to be 90 MHz in order to match the RTI clock source, and configTICK_RATE_HZ is set to 1000 Hz. This gives that the value of the compare and update compare registers are $\frac{\frac{90\times10^6}{2}}{1000} = \frac{45\times10^6}{1000} = 45000$, and thus, that one tick for the operating system corresponds to 45 000 ticks for the free running counter. As the frequency of the free running counter is 45 MHz, corresponding to a period of 22.22 ns, one tick for the operating system corresponds to 22.22 ns $\times 45000 = 0.9999$ ms. The period in ticks for the CAN task in Figure 5.2 is derived to be ten ticks, which corresponds to 0.9999 ms $\times 10 = 9.999$ ms ≈ 10 ms.

The reasoning above points to the fact that it is not the operating system settings that is the cause to the incorrect period. Thus, the source of the error is probably embedded in the Tracealyzer.

The Tracealyzer tool consists of a graphical interface which interprets data given by a trace recorder library integrated with FreeRTOS. The trace recorder library includes functions which have to be added to the application code in order to trace the tasks. During execution, trace data consisting of events is stored in a RAM buffer of the microcontroller. After execution, the data contained in the RAM buffer can be saved into a text file and uploaded to the graphical interface.

The only hardware dependency of the recorder library is a hardware timer port which is needed in order to get accurate timestamps of the events. The hardware timer port needs to be configured to fit the target microprocessor. Although a port is available for the TMS570 microcontroller, it is unofficial and not yet verified by Percepio, why there are reasons to believe that the port is the cause to the incorrect periods.

There are two macros in the recorder library port which could be the source of the error, namely HWTC_COUNT and HWTC_PERIOD. The first macro should correspond to the current value of the counter, and is expected to be reset each tick interrupt. The second macro should correspond to the number of increments of HWTC_COUNT

between two tick interrupts. In the unofficial port, HWTC_COUNT and HWTC_PERIOD are set as shown in Equation 5.2 and 5.3, respectively. In these equations, H_C is equal to HWTC_COUNT, and H_p is equal to HWTC_PERIOD.

$$H_c = (R_{free} - (R_{comp} - R_{ucomp})) \tag{5.2}$$

$$H_p = R_{ucomp} \tag{5.3}$$

R_{free} is the value of the free running counter, while R_{comp} corresponds to the compare register, and R_{ucomp} corresponds to the update compare register. Because HWTC_COUNT is expected to reset after each tick interrupt, it is calculated as the value of the free running counter subtracted by the difference between the compare register and the update compare register. As R_{comp} contains the sum of the number of ticks since the last interrupt and the update compare register, HWTC_COUNT results in the correct value. Also the HWTC_PERIOD is configured correctly, as the update compare register contains the number of ticks between two interrupts.

As the macros seems to be correctly configured, no errors are to be found in the port. This means that the source of the error is yet to be discovered. However, even though the error remains and affects the credibility of the results, some results from the Tracealyzer are still useful. As an example, it is possible to extract the execution times of the different tasks. Assuming the error affecting the timestamps of the periods also affects the timestamps used to extract the execution time, the raw execution time data should be multiplied with two in order to get the actual execution time. Both the raw average execution time and the actual average execution time are summarized in Table 5.1 .

As expected, the brake and throttle tasks have a similar average execution time. This is since these two tasks are almost identical. The interrupt task has the lowest execution time, while the LCD task has a significant higher execution time than all the other tasks. This is also expected, as the LCD task has to execute a lot of commands in order to communicate with the display, where one character is written at a time.

Table 5.1: The tasks and their average execution time.

Task name	**Raw average execution time** (μs)	**Actual average execution time** (μs)
brake	26	52
throttle	27	54
CAN	15	30
LCD	742	1484
interrupt	10	20

In addition to the execution times, it is possible to extract the CPU load from the Tracealyzer, as shown in Figure 5.3. The different bar areas are divided in time units,

and each time unit shows the CPU load contribution of each task during that time. The green bars show the initiation task, and especially the behaviour of the busy-wait loop implemented for the home position finder. During the home position finder phase, the CPU load is 100%. The reason for this is that the home position finder is polling a pin value indefinitely until it has a desired value, blocking everything else from the CPU. However, after the home position is found, the CPU load is heavily reduced. The CPU load about seven seconds after the termination of the initiation task is shown in Figure 5.4.

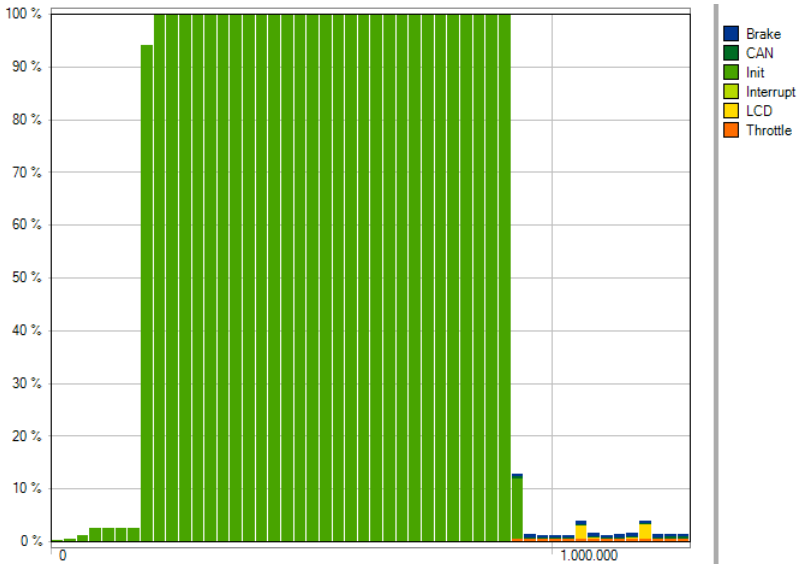

Figure 5.3: The CPU load of the first part of the execution.

Using the assumption that all timestamps of the recorder data are scaled the same way, the CPU load in Figure 5.4 should be relatively correct. This is because the CPU load describes how much the CPU is working, and with the CAN task as an example, working 26 microseconds out of 5000 microseconds twice is the same as working 52 microseconds out of 10 000 microseconds. In addition, the pattern shown in Figure 5.4 should be correct, as all timestamps assumes to be scaled by the same factor. However, the execution load peaks does probably not correspond to the actual load. Therefore, Figure 5.5 which shows that the average load is about 2%, should be approximately the same as the actual average execution load.

5.2 Memory utilization

The microcontroller has a total memory space of 4 GB. This space is divided into several regions, most reserved for system modules, ECC, CRC, peripherals, and similar. The internal memory regions available for applications is mainly a 3 MB flash and a 256 KB RAM. In addition, the development board includes an external SDRAM of 8 MB,

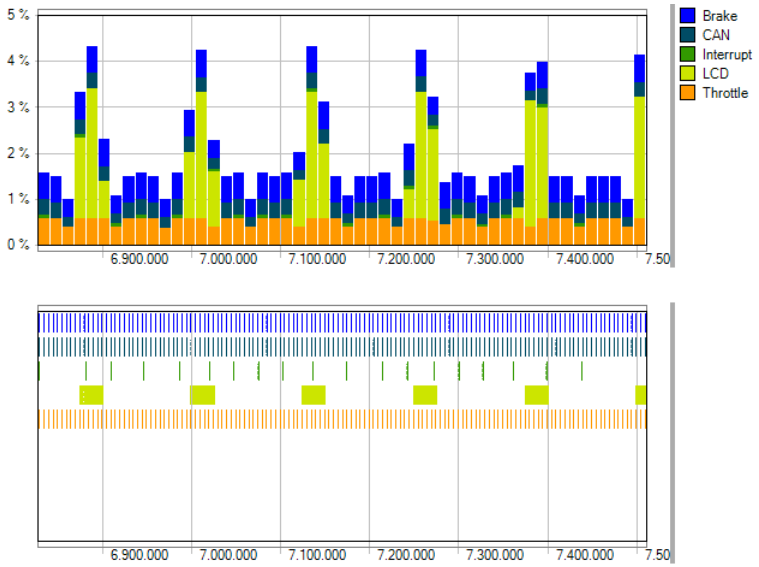

Figure 5.4: The CPU load after seven seconds of execution.

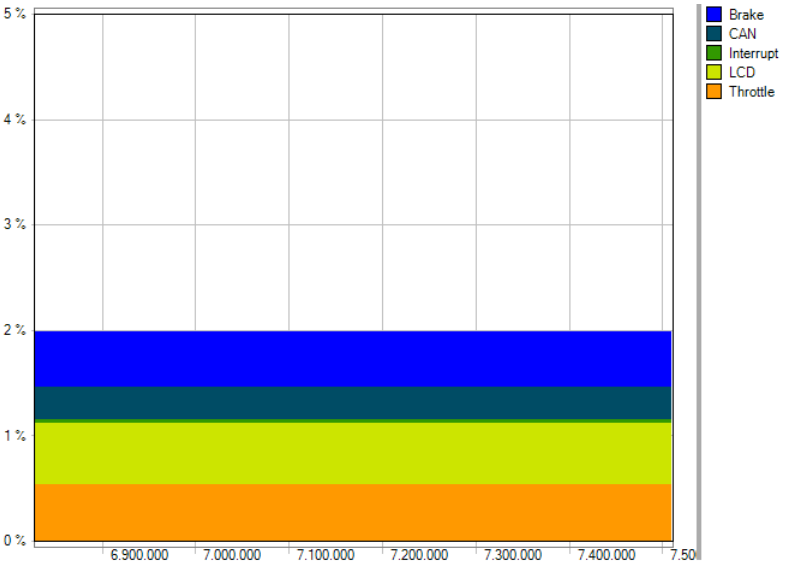

Figure 5.5: The average CPU load after seven seconds of execution.

accessible through an external memory interface located in the memory space. However, the external SDRAM is not used in this project, and therefore omitted from this analysis.

The flash is divided into two separate flash banks, each 1.5 MB. In the first flash memory bank, the kernel code and the application code are placed. The kernel code consists of functionality specific for FreeRTOS, such as task handling, queue handling, the scheduler, and the code necessary to port the RTOS to the target microcontroller, as well as the startup routine. The application code represents everything that has to do with the functional behaviour, such as the main function and the peripheral drivers.

The RAM region contains all variables and data used by the application and the kernel, including the heap for the kernel.

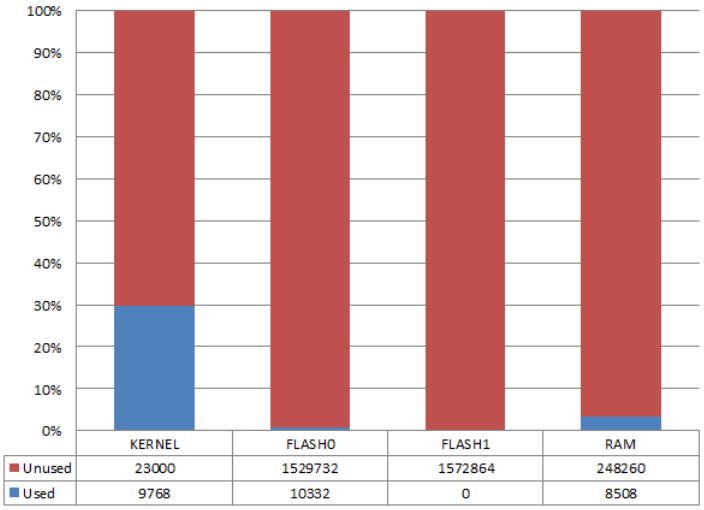

Figure 5.6: Memory usage for the kernel, the flash, and the RAM regions. The sizes are given in bytes.

Figure 5.6 shows the memory usage for the flash and RAM regions. Even though the kernel addresses are placed inside a configurable segment of the flash region, the system perceives this as a separate area. Thus, the figure handles the kernel and the flash as separate regions.

Figure 5.6 also shows that the size of the kernel code is 9 768 bytes, and uses approximately 30% of the total memory area. It also shows that the application code of 10 332 bytes takes up less than 1% of the first memory bank, while the entire second flash bank is available. Furthermore, 8 508 bytes of the RAM are occupied, corresponding to approximately 3% of the total RAM. Seeing the kernel as a part of the flash region (bank 0 and bank 1), the summarized result of Figure 5.6 can be seen in Table 5.2.

5.3 Pin and channel resources

The pins and the channels of the development board make it possible for the microcontroller to interact with its environment. The pins and channels currently used by the

Table 5.2: A summary of the memory utilization.

	Flash (both banks)	**RAM**
Used space (bytes)	20 100	8508
Total space (bytes)	3 352 696	256 768
Free space (%)	99.4	96.69

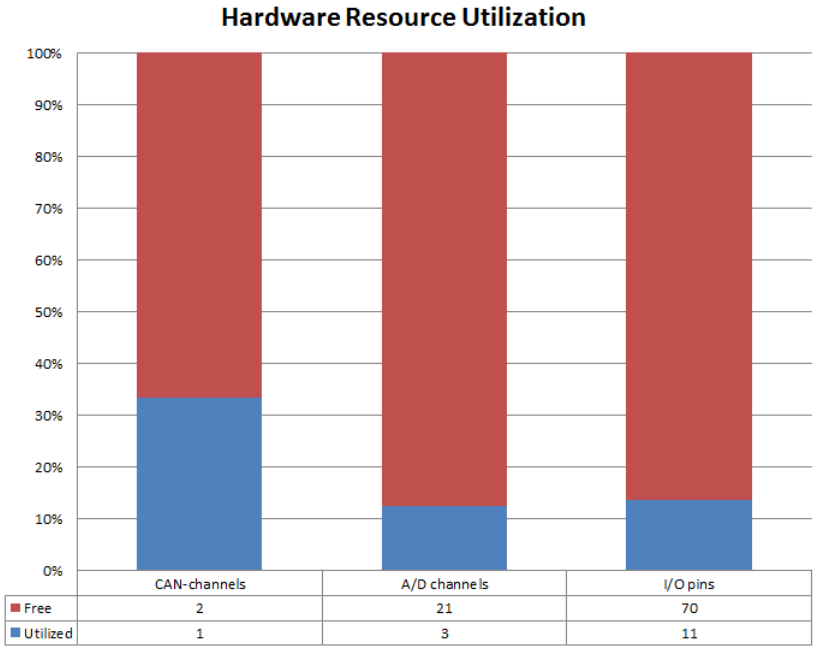

Figure 5.7: The hardware resource utilization of the I/O pins, the A/D channels, and the CAN channels..

CCU form three types of resource groups: A/D-channels, CAN-channels, and digital I/O pins. Figure 5.7 shows the current hardware resource utilization of those groups.

Table 5.3: A summary of pin- and channel utilization.

	CAN channels	**A/D channels**	**I/O pins**
Utilized resources	1	3	11
Total resources	3	24	81
Available resources (%)	67	87.5	86.4

The numbers of total resources presented in Figure 5.7 and Table 5.3 are based on the pin mapping between the external industrial connectors and the extension connectors

mounted on the back of the development board. All pin and channel resources available from the development board are not connected to the industrial connectors, and it is possible to change the pin mapping as required. This is why the resource utilization analysis shown in Figure 5.7 is only valid when applying the current pin mapping. A detailed specification of the pin mapping can be found in Appendix A.

5.4 Response time analysis

Since a drive-by-wire system uses electrical connections instead of a mechanical equivalent, there have to be certain timing requirements on the system in order for it to appear to be as responsive as when using mechanical components. When turning the steering wheel or hitting the brake, the corresponding actuator must respond in a certain amount of time, which is determined by the timing constraints set on the system. One way to determine response time constraints of a brake-by-system is showed by the TIMMO-2-USE (Timing Models - Tools, algorithms, language, methodology, and use) project [28], started by ITEA2 (Information Technology for European Advancement 2) in 2010. The goal of the project was to decrease the time-to-market and reduce the development risks by designing automated development steps, and making it possible to specify timing constraints in AUTOSAR based systems.

During the TIMMO-2-USE project, a brake-by-wire validator [29] was developed. The validator was designed to control an ABS, which in turn controls the brakes. As this system is similar to the brake functionality in the CCU, the method used in the TIMMO-2-USE project can be adapted to the functionality of the prototype developed in this project.

The timing constraints used for the brake-by-wire validator are based on a distance requirement, stating that the actuator shall alter the brake within 5 meters after the brake pedal's position is changed. Thus, the time is multiform and the timing constraints are dependent on the speed of the vehicle. By knowing the distance constraint, the maximum accepted response time at a given velocity can be calculated using Equation 5.4.

$$t = \frac{s}{v} \tag{5.4}$$

In the CCU prototype, there are several components between the brake pedal and the actuator that will add time delays to the response time. For this analysis, three different kinds of time delays will be included: execution times, periodicity of tasks, and communication delays. The propagation delay in the wires are to be seen as negligible, due to the fact that they are several orders of magnitude below the other delays. Furthermore, since the prototype is of the size to fit in an RC car, one analysis will cover a low speed vehicle driven by the currently used DC-motor, while a second analysis will cover a full-sized vehicle travelling at high speeds.

In order to calculate the maximum speed of an RC car using the drive-by-wire system as is, several aspects need to be considered: the perimeter of the wheels, the inertia of the

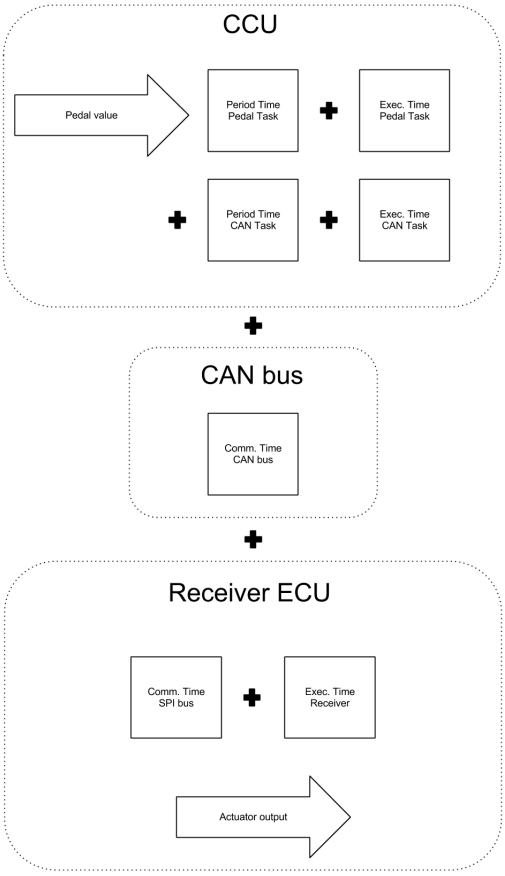

Figure 5.8: Delays affecting the brake-by-wire response time.

wheels, the torque of the DC-motor, and the total weight of the vehicle. However, because several of these variables are unknown, the maximum velocity of an RC-car driven with this kind of motor is estimated to be 15 km/h. With the velocity determined, the time constraint for a low speed vehicle can be calculated to be 1200 ms. The response time can be calculated by adding all the worst case delays affecting the signal propagation time. The different delays are illustrated in Figure 5.8.

When the brake pedal's position is changed, the new position is read by the CCU. As the reading of the pedal's position is done using a periodic task, the worst case delay when reading a new value is equal to the periodicity of the task plus the execution time of the task. After the value has been read it is passed to the CCU's CAN module, to be sent over the CAN bus to the receiver ECU. Since there is a task in the CCU responsible of sending CAN messages, a delay equal to the periodicity of the CAN task is added to the response time, along with the execution time of the task. Furthermore, the CAN module in the CCU has to analyze and structure the message before sending it, which adds a delay to the response time equal to the execution time of the CAN module [14]. However, since the execution time of the module is in the range of nanoseconds, the delay

Table 5.4: Response time and signal path delays of the brake-by-wire functionality.

	Delay	**Time**
CCU related	Period time of pedal task	$10ms$
	Execution time of pedal task	$52\mu s$
	Period of CAN task	$10ms$
	Execution time of CAN task	$30\mu s$
Other	Communication time on the CAN bus	$112\mu s$
	Communication time on the SPI bus	$112\mu s$
	Execution time of loop revolution	$7ms$
	Final response time	$27.306ms$

can be neglected in the final response time. When the CAN module is ready, the message is sent with a speed of 1 Mbps over the CAN bus, which results in a communication delay depending on the number of bits sent. Since three data bytes are included in the CAN frame sent to the receiver ECU, a total of 112 bits will represent the CAN message.

When the message is sent by the CCU, the CAN module mounted in the receiver ECU has to be executed in order for the message to be received. Since the execution time of the CAN module mounted on the sender side is neglected, it can be assumed that the execution time of the CAN module mounted on the receiver side can be neglected as well. Furthermore, the CAN module forwards the message to the microcontroller controlling the brake actuator in the receiver ECU. The communication is performed over an SPI bus operating at 1 Mbit/s, which adds one additional communication delay to the response time. The bit rate of the SPI communication was set during the development of the prior prototype [11].

Finally, an execution time of the microcontroller controlling the actuator will be added to the response time. Since the receiver ECU is executing a busy-wait-loop, the worst case delay of the actuator controller is equal to the execution time of one loop revolution. There is no documentation regarding the execution time of the receiver ECU, but the main loop in the throttle and brake sender ECU is configured to be run 150 times per second [11]. Assuming the same amount of instructions in the application code of the sender ECU and the receiver ECU, the execution time of the ECUs are estimated to be equal. Table 5.4 shows the delays affecting the signal propagation, along with the final value of the response time.

As seen in Table 5.4, the response time is estimated to be 27.306 ms, and the response time constraint of 1200 ms is met. However, the intention is not to drive an RC car, but to drive a full-sized car travelling at speeds higher than 15 km/h. The maximum speed used in the response time analysis during TIMMO-2-USE was set to 130 km/h. Adopting this speed together with a distance constraint set to 5 meters, the timing constraint is equal to 138 ms. Since the same signal path is used in this case, the response time of the system is still 27.306 ms. Thus, the system fulfills the timing requirements set when

travelling at speeds up to 130 km/h.

The response time analysis conducted during the TIMMO-2-USE project covered only the brake-by-wire functionality - not the steer-by-wire functionality. Although, since the importance of steering is equal to the importance of braking, it is reasonable to assume that the distance constraint used in the brake-by-wire response time analysis also can be used in a response time analysis of the steer-by-wire functionality. Thus, the timing constraints from the brake-by-wire analysis can be used for the steer-by-wire analysis. However, the signal path of the steer-by-wire functionality differs from the brake-by-wire functionality, why a new response time needs to be calculated. Instead of reading the angle of the steering wheel periodically, an interrupt is flagged when the angle is altered. The actual reading is then performed using an interrupt routine in the CCU. Since several instructions need to be run to read the steering wheel angle, an execution time delay is added to the response time. When the steering wheel angle is read by the CCU, the same signal path as used in the brake-by-wire functionality is applied. However, the number of data bytes is set to one in this case, resulting in a CAN frame consisting of 96 bits. Figure 5.9 shows the signal path components affecting the steer-by-wire response time, while Table 5.5 specifies the propagation delays of the signal, along with the total response time of the steer-by-wire functionality.

Table 5.5: Response time and signal path delays of the steer-by-wire functionality.

	Delay	**Time**
CCU related	Execution time of interrupt routine	$20\mu s$
	Period of CAN task	$10ms$
	Execution time of CAN task	$30\mu s$
Other	Communication time on the CAN bus	$96\mu s$
	Communication time on the SPI bus	$96\mu s$
	Execution time of actuator controller	$7ms$
	Final response time	$17.242ms$

With a final response time of 17.242 ms, the steer-by-wire functionality fulfills the timing constraints, both when used in an RC car travelling at 15 km/h, and when used in a vehicle travelling at 130 km/h. Due to the fact that both the brake-by-wire functionality and the steer-by-wire functionality meets the corresponding timing constraints, it is feasible to say that the drive-by-wire system fulfills the timing requirements determined using the distance constraints set during the TIMMO-2-USE project.

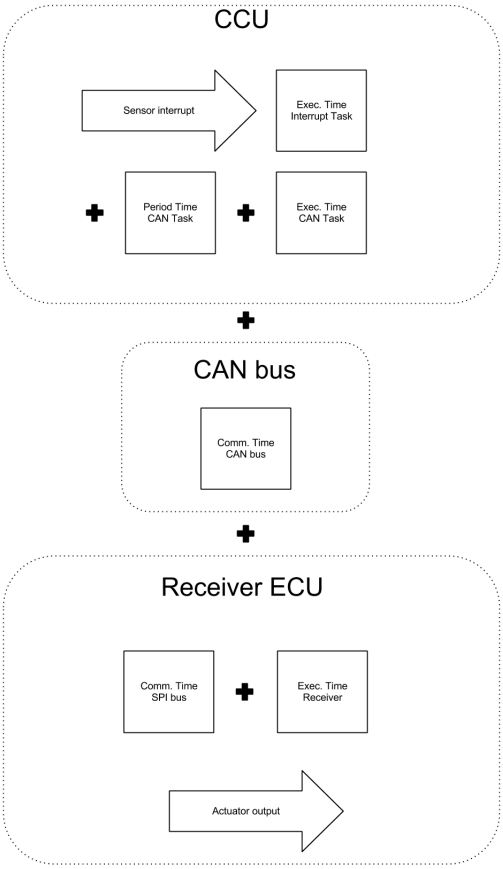

Figure 5.9: Delays affecting the steer-by-wire response time.

6

Discussion

The designed platform is working well with the current implementation, as well as with the subsystems of the prior prototype. However, there are some issues which have to be considered in the future, when the platform is operating in another system.

6.1 The choice of platform

As seen in chapter 5, the pin and channel utilization of the CCU is low, as 13% of the available I/O pins, 13% of the A/D channels, and 33% of the CAN channels are used by the implementation. In addition, there are other hardware resources such as one LIN channel, two Flexray channels, and one ethernet channel available for use. The low pin and channel utilization gives a good basis for future projects to add more features to the unit.

In addition to the pin and channels, a prerequisite for adding more features is that the microcontroller can handle the load. The current implementation uses on average about 2% of the capacity of the CPU, which gives room for more demanding implementations. Also the memory utilization is low, with a total of about 1% usage of the flash memory, and 3.5% usage of the RAM.

Even though the flash memory utilization is low, questions arise regarding the size of the kernel library image of the FreeRTOS implementation. According the FreeRTOS developers, a typical kernel library image is between 4 KB and 9 KB, but the footprint of the FreeRTOS as seen in chapter 5 is almost 10 KB. Logically, this should not be the case, especially because some of the FreeRTOS features such as queues are not used in this particular implementation. According to the FreeRTOS FAQ site [30], this phenomenon could be explained by the additional libraries included in the build. In this case, those libraries consist of files generated by HalCoGen. By removing unnecessary functionality from the HalCoGen build, the footprint may be reduced. In addition, the size of the

FreeRTOS footprint depends on the instruction set of the target microcontroller, as well as the used compiler. The instruction set is static, but it may be possible to reduce the footprint by using another compiler. However, the size of the footprint is not critical in this implementation, but it may be necessary to optimize it in the future.

As for the flash memory, the usage of the RAM is low and fits well within the limits of the total RAM. However, if optimization is needed in a future project, the FreeRTOS FAQ site also give some hints of how to reduce the RAM usage. This can be done by minimizing the stack size for the main function and the tasks as well as recover the stack of the main function, as it is not required when the scheduler has started. In addition, the usage of RAM can be reduced by rationalising the number of tasks. As an example, it may be convenient to merge the throttle task and brake task as they work in a similar manner.

Even though it is possible to optimize the memory usage, the TMS570 development board and FreeRTOS works well together, and offers a platform which is prepared to be extended with additional features. However, there are some limitations directly connected to FreeRTOS which may induce problems in the future.

One limitation is the absence of stable diagnostic tools. The only existing diagnostic tool is the Tracealyzer from Percepio, which is not officially supported by the chosen platform, and may create some confusion regarding the interpretation of the results. This can be handled when the task and semaphore count is as small as it is in the current implementation. However, when more functionality is added, it may be harder to interpret and analyze the behaviour of the system. If the running system can not be analyzed, it may be hard to manage. This is not desirable, as the CCU should be able to handle real-time systems with many more features than the current implementation.

Another limitation is the open-source license, which also can be considered as a strength. While the distribution of the open-source code is free to use with full insight into the kernel code, it also limits both the available diagnostic tools, as described above, and the support of the product. A product with a commercial licence probably has support given by e-mail or telephone, where the responsibility to solve issues is placed on the company that provides the product, rather than the end user. In addition, a company that offers a product is probably more prone to provide first-hand, fully supported, diagnostic tools.

The choice between a commercial and an open-source license is a trade-off, depending both on how far the open-license product is developed, as well as the knowledge of the user. It also depends on the purpose of the project in which the product will be used. In this case, the end result of the project will be a component placed in a safety-critical environment, why it may be safer to invest in a commercial product with integrated support for safety-critical systems. However, if the safety-critical requirements and the verification requirements on the end system are low, FreeRTOS will suffice.

6.2 Future improvements

The CCU prototype is a stepping-stone in the process of adapting the system for implementation in a vehicle. However, there are still a lot of work left before the system is ready. This subchapter describes improvements that cover some immediate weaknesses of the current system.

6.2.1 Develop system guidelines

The current prototype is built on prototypes developed by prior project groups, but no standardizations have been done regarding the complete system. Seeing the project from a future system perspective, the system will be hard to both overview and update if ECUs developed by different groups are designed ad-hoc, without any guidelines to follow. To solve this problem, it would be necessary to develop system guidelines before the existing sensors and actuators are replaced, and before any additional functionality is added to the system. The system guidelines could consist of a standardized way of assigning CAN identifiers as well as which kind of communication bus to use, which kind of software platform that should be used for the ECUs, and general requirements on the hardware for the ECUs.

The process of developing guidelines of this kind would require a good insight in the resulting system, as well as knowledge of the target vehicle. Thus, it would be necessary to do an investigation of the target vehicle and a specification of the main components in the drive-by-wire system before developing the system guidelines. The meaning of this is that the guidelines, the investigation, and the specification should be components of the project following this one.

6.2.2 Replace the sensors, the receiver ECUs, and the display

The current subsystems are using pedals and steering wheels adapted for the gaming industry, and their sensors are not as accurate as they have to be in an actual vehicle. This is particularly evident when it comes to the magnet sensor in the steering wheel, which gives a value relative to the home position calibrated by the software. As there are several home positions indicated by the same value of the output signals, there is no way for the software to decide where the calibrated home position actually is. This could mean that the wheel angle does not correspond to the steering wheel angle, even though the system is free of errors. To avoid this problem, the sensor for the steering wheel should give an absolute value for each of the recordable steering angles.

As the receiver ECUs are adapted to wheels from an RC car, they are undersized for use in a full-size car. The receiver ECUs has to be adapted to fit the wheel system of an actual vehicle, which means that both the hardware and the software of the ECUs has to be replaced.

In addition, the display used in this project shows only the most basic information on an alphanumeric display. When the sensors and the receiver ECUs have been replaced, there is a need for a more sophisticated way to display information such as speed, fuel

level and warnings. Also, the ways of indicating and shifting gear have to be replaced, as the current solution consist of a toggle switch attached to the display box.

Because the current system uses D-SUB cables to distribute the power nets, there might occur voltage drops in the subsystems. This is because the D-SUB cables have a small cross-sectional area, which in turn leads to high wire resistance. During the project, the D-SUB cable carrying the power nets from the power box to the interface box had to be shortened. This was done since only one subsystem could be run at a time, due to voltage drops in the system. When shortening the D-SUB cable, the voltage drops were reduced and both subsystems were able to run simultaneously. However, to increase the stability of the system, the power supplies need to be distributed using cables suitable for power management, with larger cross-sectional area.

In addition to the power management, also the CAN bus is distributed using D-SUB cables. Since the internal D-SUB wires are not arranged in twisted pairs, electromagnetic interference will be present during communication on the CAN bus. In addition, the power nets may interfere with the communication lines, as the subsystems are powered using the same D-SUB cables that hold the CAN bus. To reduce electromagnetic interference on the CAN bus in future projects, the power nets and the CAN bus should be distributed independently of each other.

Currently, the power box offers two different power supplies: one 12 V source with a maximum current of 1.2 A, and one 5 V source with a maximum current of 2.1 A. These supplies are sufficient when running the current system. However, if new features are added, there might be a need for additional power supplies. If so, the amount of current currently available to be drawn from the power supplies will have to be increased. Thus, the entire power box may have to be redesigned or substituted.

6.2.3 Implement safety-critical properties

An important aspect of automotive drive-by-wire systems, omitted from both this and previous project, is that it is closely related to the safety of humans. If an error occurs somewhere in the system, it could in worst case mean the injury or even death of several persons. This means that safety checks have to be added to the system to make sure that everything works as expected. Some safety-critical properties are already present in the microcontroller of the CCU, such that lock-stepped CPUs and memory checks, but these properties are only useable if the software of the CCU is correct and reliable. Because of this, it is important to start with the safety-critical aspects of the software. This can be done by testing and verifying the code, but also by using methods such as N-version programming.

Besides the CCU, safety-critical properties in any form are also absent in the rest of the system. One way enhance system safety is to make the system fault-tolerant as far as possible. This means that the system has to work passably even if any component fails. In practise, this could mean that every node has at least two ECUs working in parallel of each other, as well as redundant sensors, power supplies, and communication buses. However, as the cost of the system will grow larger with redundant hardware, an assessment has to be made whether the increased safety is worth the cost or not.

7

Conclusion

The purpose of this project was to prepare a drive-by-wire prototype for implementation in a full-size electric car, mainly by merging two parallel ECUs into one CCU. By proposing and assemble a platform for the CCU based on a TMS570 development board and FreeRTOS, and also implementing the prior functionality on the CCU, the project has reached the goal of merging the parallel ECUs. Furthermore, a response time analysis shows that the CCU has the potential to support full-size cars travelling at high speeds.

In addition to the CCU, an interface box and a display box were developed during the project. While the display box is a temporary solution for indicating the major activities inside the CCU, the interface box was developed to connect the prior system to the new CCU. As for now, these solutions works well, but the interface box has to be replaced when new physical components are added. Also, the display box is meant to be replaced with a dashboard.

A requirement for the CCU was that the solution had to be extensible, both with respect to hardware and software. The current software application uses a low amount of both pins and channels, as 13% of the I/O pins, 13% of the ADC channels, and 33% of the CAN channels are utilized. In addition, there are a lot of communication resources not used at all by the current application, such as two Flexray channels, one LIN channel, and one ethernet channel. The low pin and channel utilization give opportunities for following project groups to extend the current implementation. This is also the case with the CPU load and the memory utilization, as the average CPU load is 2%, the flash memory utilization is 1%, and the RAM memory utilization is 3.5%. Furthermore, the current user-dependent configurations of FreeRTOS are not optimized. This means that it is probably possible to scale down the kernel library image size and the RAM usage, making the flash and RAM memory utilization even lower. This may be necessary in the future, when the space requirement of memory increases.

The CPU load analysis shows that it is possible to add more functionality to the

CCU without a significant performance loss. The software platform contributes to this, as the RTOS includes a scheduler which helps solving timing issues. In addition, it is easy to add new functionality to the CCU. However, knowledge of real-time systems is a prerequisite to understand how to structure the tasks, and how to identify critical sections of the tasks. This is necessary to create a reliable system which behaves as expected.

The objectives of this project were reached, but the choice of using FreeRTOS may cause problems in the future. The main reason for choosing FreeRTOS instead of Arctic Core, SafeRTOS, or μC/OS-II was the cost, and the fact that FreeRTOS has a GPL licence. The result of this is however that the software platform does not support any stable analysis tools, and the product support is limited to an online forum. Furthermore, a disadvantage with FreeRTOS is that it does not include any safety-critical properties, which may be needed in the future.

The sum of these disadvantages is that it may be necessary to update the software platform if safety-critical properties and verification are important factors in future projects. This is most probably the case, as the target environment for the CCU is safety-critical.

During the project, the need for developing system guidelines was identified. Previous components have been developed ad-hoc, resulting in a system which is hard to overview. To make it possible to make good decisions regarding how to develop new ECUs and update old ECUs, general system guidelines need to be developed. Also, a decision on the target vehicle has to be taken to move the prototype forward for vehicle implementation.

Bibliography

[1] AsiaNet Pakistan (Pvt) Ltd, "Google brings driverless cars," p. 45, October 2012.

[2] R. Isermann, R. Schwarz, and S. Stölzl, "Fault-Tolerant Drive-by-Wire Systems," *IEEE Control. Syst. Mag.*, vol. 22, pp. 64–81, Oct. 2002.

[3] Z. Rui, Q. Gui-he, and L. Jia-qiao, "Gateway system for CAN and FlexRay in automotive ECU networks," in *Information Networking and Automation (ICINA), 2010 International Conference on*, vol. 2, 2010, pp. V2–49–V2–53.

[4] R. Bril, A. Burns, R. Davis, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 2007.

[5] Pazul, K, "Controller Area Network (CAN) Basics," 1999. Accessed August 11, 2013. [Online]. Available: http://www.ee.uidaho.edu/ee/classes/ECE341/refs/AN713%20-%20Controller%20Area%20Network%20%28CAN%29%20Basics.pdf

[6] Blackman, J and Monroe, S, "Overview of 3.3V CAN (Controller Area Network) Transceivers," 2013. Accessed August 11, 2013. [Online]. Available: http://www.ti.com/lit/an/slla337/slla337.pdf

[7] W. Voss, *A Comprehensible Guide To Controller Area Networks.* Copperhill Technologies Corporation, 2005.

[8] Atmel, *8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash*, June 2011. Accessed August 1, 2013. [Online]. Available: http://www.atmel.com/Images/doc2467.pdf

[9] Microchip, *MCP2515 - Stand-Alone CAN Controller With SPI Interface*, May 2007. Accessed August 1, 2013. [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/21801e.pdf

[10] Austria Microsystems, *AS5304/AS5306 - Integrated Hall ICs for Linear and Off-Axis Rotary Motion Detection*, Accessed August 1, 2013.

[11] D. Robertsson and S. Rhodin, "Prototype of a Drive by Wire system," B.S. thesis, Chalmers Univ. of Tech., Gothenburg, 2012.

[12] Atmel, *AVR32919: AT32UC3C-EK User Guide*, September 2010. Accessed August 1, 2013. [Online]. Available: http://www.atmel.com/Images/doc32151.pdf

[13] ——, *AVR32UC Techical Reference Manual*, March 2010. Accessed August 1, 2013. [Online]. Available: http://www.atmel.com/Images/doc32002.pdf

[14] Texas Instruments, *TMS570LS3137 16/32-Bit RISC Flash Microcontroller*, November 2012. Accessed August 1, 2013. [Online]. Available: http://www.ti.com/lit/ds/symlink/tms570ls3137.pdf

[15] ——, *TMS570LS31x Hercules Development Kit (HDK) User's Guide*, September 2012. Accessed August 1, 2013. [Online]. Available: http://www.ti.com/lit/ug/spnu509a/spnu509a.pdf

[16] ——, *TMS470MF06607 16/32-Bit RISC Flash Microcontroller*, January 2012. Accessed August 1, 2013. [Online]. Available: http://www.ti.com/lit/ds/symlink/tms470mf06607.pdf

[17] ——, *TMS470M Hercules Development Kit (HDK) User's Guide*, April 2013. Accessed August 1, 2013. [Online]. Available: http://www.ti.com/lit/ug/spnu510/spnu510.pdf

[18] Keil, *MCB1700 User's Guide*, April 2013. Accessed August 1, 2013. [Online]. Available: http://www.keil.com/support/man/docs/mcb1700/mcb1700_intro.htm

[19] NXP, *Microcontroller LPC1768*, August 2012. Accessed August 1, 2013. [Online]. Available: http://www.nxp.com/documents/data_sheet/LPC1769_68_67_66_65_64_63.pdf

[20] AUTOSAR GbR, *AUTOSAR Technical Overview (Version 2.2.2)*, August 2008. Accessed August 1, 2013. [Online]. Available: http://www.autosar.org/

[21] Arccore, "Arctic Core," 2013. Accessed August 1, 2013. [Online]. Available: http://www.arccore.com/products/arctic-core/

[22] OSEK/VDX, *OSEK/VDX Operating system (Version 2.2.3)*, February 2005. Accessed August 1, 2013. [Online]. Available: http://www.osek-vdx.org/

[23] Real Time Engineers Ltd., "FreeRTOS," 2013. Accessed August 1, 2013. [Online]. Available: http://www.freertos.org/

[24] WITTENSTEIN High Integrity Systems, "SafeRTOS," 2013. Accessed August 1, 2013. [Online]. Available: http://www.highintegritysystems.com/

[25] Micrium, "μC/OS-II - The real time kernel," 2013. Accessed August 1, 2013. [Online]. Available: http://micrium.com/

[26] Display Elektronik, *LCD Module DEM 20486 SBH-PW-N*, June 2009. Accessed August 1, 2013. [Online]. Available: http://www.display-elektronik.de/DEM20486SBH-PW-N.PDF

[27] Percepio, "FreeRTOS+Trace - Tracealyzer for FreeRTOS," 2013. Accessed August 1, 2013. [Online]. Available: http://percepio.com/

[28] ITEA2, "TIMMO-2-USE," 2011. Accessed August 1, 2013. [Online]. Available: http://www.timmo-2-use.org

[29] TIMMO-2-USE, "Brake-By-Wire Validator," 2012. Accessed August 1, 2013. [Online]. Available: http://www.timmo-2-use.org/deliverables/TIMMO-2-USE_BBW.pdf

[30] Real Time Engineers Ltd., "FreeRTOS FAQ - Memory Usage and Boot Times," 2013. Accessed August 1, 2013. [Online]. Available: http://www.freertos.org/FAQMem.html

A

Pin mappings

A.1 Pin setup and pin configurations of the D-SUB connectors

Two different D-SUB pin configurations are available in the system. One of the configurations is used in the D-SUB connectors between the power supply and the interface box, while the other configuration is used in the D-SUB connectors that connects the receiver ECUs with the interface box. The configurations can be found in Table A.1 and A.2, respectively. Both of the pin configurations are based on the same pin setup, which can be seen in Figure A.1.

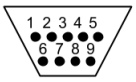

Figure A.1: Pin setup of the D-SUB connectors.

Table A.1: Pin configuration for the power box connector.

Connector pin	**Description**
1	-
2	-
3	5 V Power source
4	12 V Power source
5	GND
6	-
7	5 V Power source
8	12 V Power source
9	GND

Table A.2: Pin configuration for the receiver ECU connectors.

Connector pin	**Description**
1	GND
2	-
3	-
4	CAN L
5	CAN H
6	GND
7	12 V Power source
8	5 V Power source
9	-

A.2 Pin setup and pin configuration of the RJ10 connector

The throttle and brake pedals are connected to the interface box through an RJ10 connector. The pin setup of the RJ10 connector can be seen in Figure A.2, while the pin configuration of the RJ10 connector is specified in Table A.3.

Figure A.2: Pin setup of the RJ10 connector.

Table A.3: Pin configuration for the RJ10 connector.

Connector pin	**Description**
1	Analogue throttle value
2	3.3 V from Hercules board
3	Analogue brake value
4	GND

A.3 Pin setup and pin configuration of the RJ45 connector

As with the pedals, the steering wheel is connected to the interface box. Since the magnet sensor mounted inside the steering wheel needs to be connected using a minimum of seven pins, an RJ45 connector consisting of eight pins was used to connect the steering wheel to the interface box. The pin setup of the RJ45 connector can be seen in Figure A.3, while the pin configuration is specified in Table A.4.

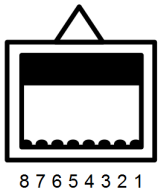

Figure A.3: Pin setup of the RJ45 connector.

Table A.4: Pin configuration for the RJ45 connector.

Connector pin	**Description**
1	5 V Power source
2	A
3	Signal
4	B
5	Index
6	GND
7	A0 (Analogue)
8	GND

A.4 Pin setup and pin configuration of the 48-pin industrial connector.

The CCU is connected to the rest of the system via the interface box through a 48-pin industrial connector. It is possible to interchange the cables in the connector, which makes the system flexible for additional features and changes. The pin setup of the 48-pin industrial connector is illustrated in Figure A.4, while the pin configuration of the connector is specified in Table A.5. The pin configuration table specifies the pin mapping between the extension connectors mounted on the Hercules development board and the external connector mounted on the side of the CCU box. Furthermore, it specifies the mapping between the pins of the development board and the signals in the system.

Figure A.4: Pin setup of the 48-pin industrial connector.

Table A.5: Pin configuration for the 48-pin industrial connector.

External connector pin	Hercules extension connector	Hercules connector pin	Description	Signal
1	Header socket (Bottom,BottomView)	1	12 V Power source	12 V Power source
2	CAN1 terminal block	1	CAN1 TX	CANH
3	CAN1 terminal block	3	CAN1 RX	CANL
4	Header socket (Bottom,BottomView)	3	12 V Power source	-
5	Header socket (Bottom,BottomView)	6	LIN TX	-
6	Header socket (Bottom,BottomView)	5	LIN RX	-
7	Header socket (Bottom,BottomView)	13	FRAY1 RX	-
8	Header socket (Bottom,BottomView)	15	FRAY1 TX	-
9	Header socket (Bottom,BottomView)	17	FRAY1 EN	-
10	Header socket (Bottom,BottomView)	20	GIOA[0]	-
11	Header socket (Bottom,BottomView)	19	GIOA[1]	-
12	Header socket (Bottom,BottomView)	22	GIOA[2]	-
13	Header socket (Bottom,BottomView)	21	GIOA[3]	-
14	Header socket (Bottom,BottomView)	24	GIOA[4]	SWITCH
15	Header socket (Bottom,BottomView)	23	GIOA[5]	A (Digital)
16	Header socket (Bottom,BottomView)	26	GIOA[6]	B (Digital)
17	Header socket (Bottom,BottomView)	25	GIOA[7]	Index
18	Header socket (Bottom,BottomView)	28	GIOB[0]	LCD_E
19	Header socket (Bottom,BottomView)	27	GIOB[1]	-
20	Header socket (Bottom,BottomView)	30	GIOB[2]	LCD_RS
21	Header socket (Bottom,BottomView)	29	GIOB[3]	LCD_RW
22	Header socket (Bottom,BottomView)	32	GIOB[4]	LCD_DB4
23	Header socket (Bottom,BottomView)	31	GIOB[5]	LCD_DB5
24	Header socket (Bottom,BottomView)	34	GIOB[6]	LCD_DB6
25	Header socket (Bottom,BottomView)	33	GIOB[7]	LCD_DB7
26	Header socket (Bottom,BottomView)	38	NHET1[0]	-
27	Header socket (Bottom,BottomView)	37	NHET1[1]	-
28	Header socket (Bottom,BottomView)	40	NHET1[2]	-
29	Header socket (Bottom,BottomView)	39	NHET1[3]	-
30	Header socket (Bottom,BottomView)	42	NHET1[4]	-
31	Header socket (Bottom,BottomView)	41	NHET1[5]	-
32	Header socket (Bottom,BottomView)	44	NHET1[6]	-
33	Header socket (Bottom,BottomView)	43	NHET1[7]	-
34	Header socket (Bottom,BottomView)	46	NHET1[8]	-
35	Header socket (Bottom,BottomView)	45	NHET1[9]	-
36	Header socket (Left,BottomView)	32	AD1IN[0]	Throttle
37	Header socket (Left,BottomView)	31	AD1IN[1]	-
38	Header socket (Left,BottomView)	34	AD1IN[2]	Brake
39	CAN1 terminal block	2	GND	GND
40	Header socket (Left,BottomView)	33	AD1IN[3]	-
41	Header socket (Left,BottomView)	36	AD1IN[4]	A0
42	Header socket (Bottom,BottomView)	35	GND	GND
43	Header socket (Left,BottomView)	35	AD1IN[5]	-
44	Header socket (Left,BottomView)	38	AD1IN[6]	-
45	Header socket (Bottom,BottomView)	36	GND	GND
46	Header socket (Left,BottomView)	37	AD1IN[7]	-
47	Header socket (Left,BottomView)	61	ADREFHI	3.3 V Ref.
48	Header socket (Bottom,BottomView)	4	GND	GND

A.5 Pin setup and pin configuration of the 32-pin industrial connector

The 32-pin industrial connector is used to connect the display box, containing the LCD and the gear switch, to the rest of the system. When installed, the CCU can update the LCD and read the position of the switch, via the interface box. The pin setup of the 32-pin industrial connector is illustrated in Figure A.5, while the pin configuration is specified in Table A.6.

Figure A.5: Pin setup of the 32-pin industrial connector.

Table A.6: Pin configuration for the 32-pin industrial connector.

Connector pin	**Signal**
1	LCD_DB6
2	LCD_DB7
3	LCD_DB4
4	LCD_DB5
5	LCD_DB2
6	LCD_DB3
7	LCD_DB0
8	LCD_DB1
9	-
10	LCD_E
11	-
12	LCD_R/W
13	-
14	LCD_RS
15	-
16	SWITCH
17	-
18	-
19	-
20	-
21	-
22	-
23	-
24	GND
25	5 V Power source
26	GND
27	5 V Power source
28	GND
29	5 V Power source
30	GND
31	5 V Power source
32	GND