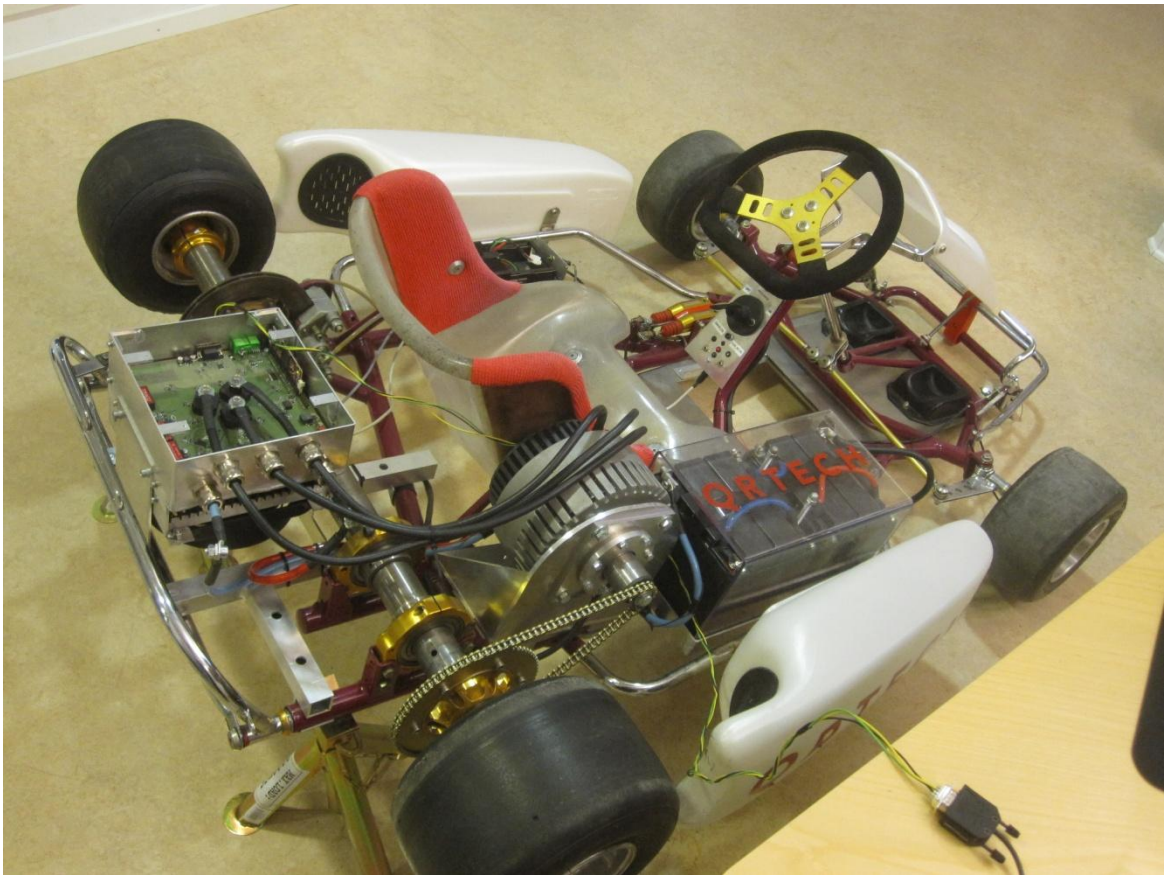
CHALMERS



Implementation of CAN-interface and RTJ heads-up display on an electric go-cart

Master of Science Thesis in the Programme Systems, Control and Mechatronics

JONAS HENNING
JOHAN LOJANDER

Chalmers University of Technology
University of Gothenburg
Department of Signals and Systems
Gothenburg, Sweden, June 2012
Report No. EX032/2012

Implementation of CAN-interface and RTJ heads-up display on an electric go-cart

JONAS HENNING
JOHAN LOJANDER

Examiner: PETTER FALKMAN

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

Cover:
A photo showing the electric go-cart platform controlled via a CAN cable.

Department of Signals and Systems
Gothenburg, Sweden, June 2012

Implementation of CAN interface and RTJ heads-up display on an electric go-cart

Master of Science Thesis in the Programme
Systems, Control and Mechatronics

Jonas Henning
Johan Lojander

June 4, 2012

Abstract

Today's cars are filled with embedded systems making way for advanced safety techniques and comfort-increasing features. As a consequence, software complexity is taken to a completely new level. To facilitate software development, automobile manufacturers have created an open standard called AUTOSAR. It serves as an interface between hardware and software, easing the reuse of applications in ECUs.

Traditionally, real-time applications in automotive industry are developed in low-level languages as C, but with growing complexity these systems tend to be hard to overview. As a reaction to this, a research project called CHARTER was started. It strives to introduce high-level languages like real-time Java (RTJ) and model driven development in critical embedded systems.

This thesis continues the development of an electric go-cart by extending the software functionality with a heads-up display, a cruise controller and a Controller Area Network (CAN) interface for its motor controller.

By adding a CAN interface to the motor controller, it was possible to instead connect the torque and brake pedals to the Vehicle Master Control Unit (VMCU), and let this device demand an adequate torque from the motor controller via CAN.

Aligning with both AUTOSAR and CHARTER, the heads-up display and the cruise controller were developed as AUTOSAR software components in RTJ. The heads-up display provides information about the vehicle to the driver, such as speed and battery status. The driver may also activate the cruise controller and set several options, through the heads-up display.

The future of AUTOSAR looks promising. The advantages of scalability and reusability outperform the overhead in implementation. With RTJ for embedded systems still being in its cradle, performance issues persists. Furthermore, the difficulty to reach the hardware with RTJ occasionally forces the use of Java Native Interface, decreasing the benefits of using Java in the first place. However, the object-oriented approach and the low learning threshold of Java are benefits worth taking into account.

Preface

This master's thesis was written for Chalmers University of Technology, Göteborg, Sweden as part of the master program Systems, Control and Mechatronics. The thesis was performed by Jonas Henning and Johan Lojander at QRTECH, Göteborg. Examiner and supervisor at Chalmers was Petter Falkman at the Signals and Systems (S2) Department. Supervisor at QRTECH was Anders Runeson and Benneth Claesson. We would like to thank our supervisors at QRTECH and Petter Falkman for the dedication and help in our project.

Throughout this report, the master's thesis will be referred to as *the thesis*.

Contents

Acronyms

Name	Description
A/D	Analog/Digital
ADC	Analog (to) Digital Converter
AUTOSAR	AUTomotive Open System ARchitecture
BMU	Battery Management Unit
BSW	Basic Software
CAN	Controller Area Network
CHARTER	Critical and High Assurance Requirements Transformed through Engineering Rigour
CRC	Cyclic Redundancy Check
DLC	Data Length Code
DSP	Digital Signal Processor
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
E/E	Electrics/Electronics
EMU	Engine Master Unit (?)
GUI	Graphical User Interface
HUD	Heads-Up Display
IDE	Integrated Development Environment
JNI	Java Native Interface
JML	Java Modeling Language
JVM	Java Runtime Environment
JVM	Java Virtual Machine
OS	Operating System
PID	Proportional-integral-derivative
PWM	Pulse-Width Modulation
RPM	Revolutions per Minute
RTE	Runtime Environment
RTJ	Real Time Java
SoC	State of Charge
SoH	State of Health
SW-C	Software Component
SysML	Systems Modeling Language
UML	Unified Modeling Language
VM	Virtual Machine
VMCU	Vehicle Master Control Unit

List of Figures

1 Introduction

Today's electrification of cars provides a lot of opportunities. Cars are filled with embedded systems making way for advanced safety techniques and comfort-increasing features. As a result of this, software complexity is taken to a completely new level. Traditionally, real-time applications in automotive industry are developed in low-level languages as C, but with growing complexity these systems tend to be hard to overview.

An ongoing project, named CHARTER (Critical and High Assurance Requirements Transformed through Engineering Rigour), carried out by QRTECH and several other European companies and institutions, serves to use model driven development (e.g. JML/UML) and high-level languages (e.g. real-time Java) to develop software for embedded systems.

To take a leading position in this matter in the automotive industry of the future, QRTECH designed a master thesis serving to develop real-time applications in Java for their electric go-cart. The go-cart has been worked on during other previous masters' theses carried out at QRTECH and serves as a platform for testing and evaluating new techniques, hardware as well as software.

1.1 Background

This section describes the challenges of this project, delimitations of the thesis, a brief introduction to the thesis-requesting company QRTECH as well as previous and related work.

1.1.1 About QRTECH

QRTECH is a product development company with services within software and electronics development. QRTECH also possesses long experience of embedded systems in the automotive industry-dense city of Gothenburg.

For several years QRTECH has worked on an electric go-cart used as a platform for testing automotive industry techniques such as electric motor control, infotainment systems, battery management systems and, concerning the thesis, drive-by-wire and real-time Java software development for critical systems.

1.1.2 Project description

The thesis will further develop the electric go-cart by implementing software functionality. The desired functions are: a heads-up display (HUD) presenting data and status about the vehicle, a cruise controller, an error handling mechanism and a CAN interface for the motor controller. Additionally, the available motor controller needs to be adapted to align with the new CAN interface developed in the thesis. Since this project is a part of

another project called CHARTER (section 2.2), real-time Java will be used for realisation of some software functionalities. In the scope of the thesis also lies a general understanding of the complete system and how its parts work together.

In Figure 1, an overview of the go-cart system is presented. Before the start of the project, the pedals of the go-cart (Throttle and Brake in Figure 1) were directly connected to the motor controller, affecting output torque of the AC-motor. For details, please refer to section 1.1.3.1. By the end of the thesis, the Vehicle Master Control Unit (VMCU) should request a torque from the motor controller via CAN, according to the pedals connected to the VMCU, see Figure 1. At thesis start, CAN was physically available on the motor controller, but unused. The thesis should design a CAN interface for the motor controller making it possible to request a torque via CAN. The CAN-network is also supposed to include a Battery Management Unit (BMU) responsible for features like measuring cell voltages and current drain as well as providing battery cell balancing functionality.

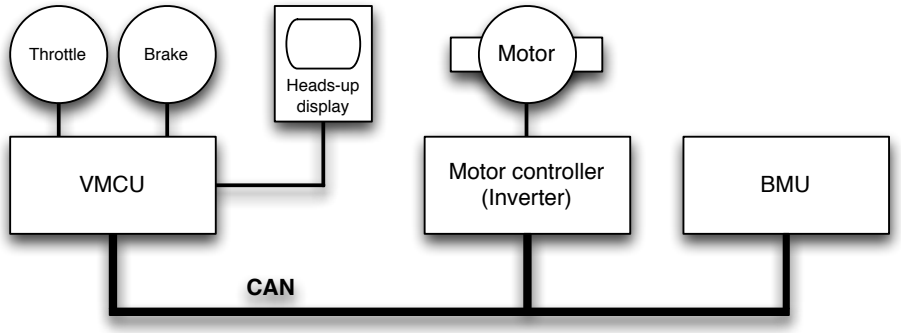


Figure 1: System description of the go-cart.

The CAN interface adaptation makes communication between the motor controller and the VMCU possible in both directions. By installing the VMCU and its display in the dashboard it can also serve as a heads-up display, presenting information such as speed, battery state of charge and engine temperature to the driver.

The design of the graphical user interface (GUI) of the heads-up display is thus part of the thesis and it should display five different views presenting information as:

- Driving View
  - Speed
  - Revolutions per minute (RPM)

  - Battery State of Charge (SoC)
  - Temperature
  - Average Speed
  - Trip-meter

- Battery View
  - Battery State of Charge (SoC)
  - Battery State of Health (SoH)
  - Voltage
  - Current drain
  - Individual Cell Voltage
  - Battery balancing mode

- Diagnostics View
  - Diagnostic Trouble Codes (DTC)
  - Possibility to clear certain or all DTCs

- Properties View
  - Driving mode selection (Normal, Eco, Sport)
  - Charging activation/deactivation
  - GUI-menu position (left/right)

- Demo View
  - QRTECH presentation

Note that development of the BMU itself is not part of the thesis, even though information will be collected from the unit for presentation on the heads-up display.

A cruise controller maintaining a desired speed given by the driver will be developed. The idea is to let the cruise controller create a virtual throttle pedal, sending an appropriate requested torque to the motor controller to maintain a close to constant speed.

Creating error handling functions to detect and react to system malfunction is also part of the thesis. The driver should be alerted of system errors and be able to take stance to and clear these errors. Security measures avoiding critical errors like brake dysfunction or the motor locking on full torque in case of a CAN failure will also be implemented.

1.1.2.1 Delimitation

The work is carried out at QRTECH in Kallebäck, Gothenburg. Focus is software development so hardware should be available and no changes intended. It might be necessary to do some measurements in order to test hardware or verify software results. There is no cost limit. Some complementary items might be purchased after approval of request. The thesis starts the 23rd of January spanning to the 8th of June, i.e. a total of 20 weeks. The final date may be postponed if necessary, but meeting deadlines are wished-for. Desired date of the handover of the planning report is early February, the oral presentation at end of May and the handover of the final report early June.

To limit time requirement for studies of related work and the go-cart system overview, technical details of these were omitted.

The thesis incorporates work with the complete CAN interface on the motor controller. On the VMCU however, only the software components as the HUD and cruise controller are worked with. The lower AUTOSAR layers, mainly the basic software and the RTE-layer are performed by a colleague at QRTECH and are not part of the thesis. Some co-operation for integrating the RTE-layer and the SW-Cs may however be needed.

Visual design of the heads-up display is secondary, focus is instead technology level and a well-working CAN interface including error detection.

1.1.2.2 The go-cart

The QRTECH go-cart was originally a standard go-cart with a combustion engine. In 2010 it was rebuilt for electric drive as a part of a master's thesis in co-operation with Chalmers, where a Simulink model of the electric motor of the go-cart was derived.

The AC-motor controller is based on a Digital Signal Processor (DSP), which can be viewed in Figure 2 as the small red board close to the black cables. The control algorithm was made as a previous master's thesis at QRTECH, described in Section 1.1.3.1. The battery pack is based on a set of car batteries supplying the electric motor 48 volts. The motor drives the rear axis with a constant gear ratio of 13:73. The throttle and brake pedals are connected through wires to sliding potentiometers. The output voltage from the potentiometers is read by the motor controllers analog to digital converters (ADCs).

Furthermore, the go-cart is intended to be rebuilt by the end of the thesis. The Pb-batteries will be changed to a lighter lithium-ion battery pack, the heads-up display and BMU will be installed close to the battery pack. Also, a CAN bus interconnecting the VMCU, DSP and BMU will be added, and the brake and throttle pedals are to be connected to the VMCU,

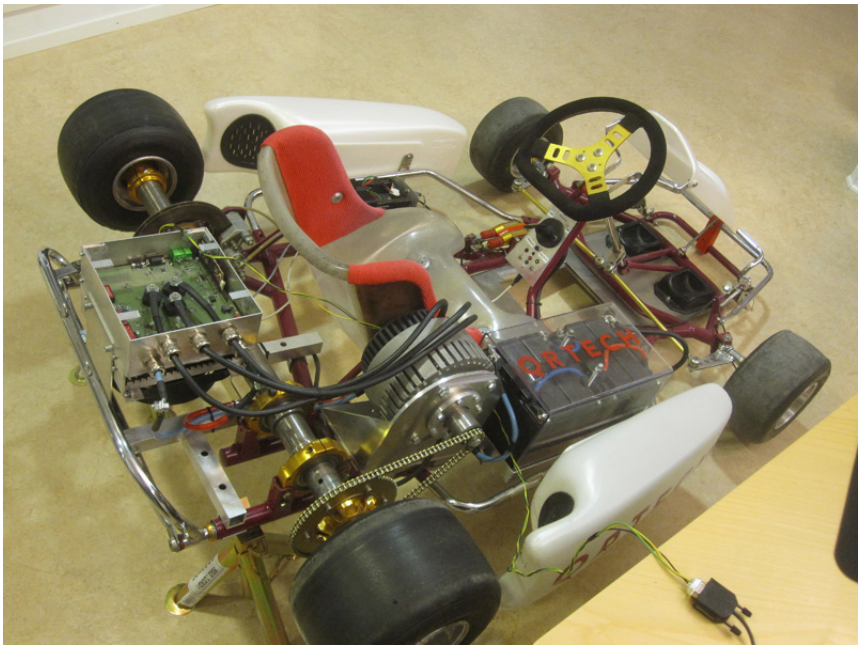


Figure 2: The go-cart before it was rebuilt. Notice the twisted CAN cable used to control the go-cart from the PC.

to support drive-by-wire functionality over CAN.

1.1.3 Related work

Since an insight of the complete system and how its parts works together was desired for in the thesis, this subsection explains previous projects related to the thesis.

1.1.3.1 AC-motor control

The previous master thesis carried out on the go-cart at QRTECH aimed to develop a motor control algorithm for the AC motor. The motor torque is controlled by a PID-regulator on the motor controller. By altering the duty cycle of the Pulse Width Modulation-signals (PWM) affecting the motor current, the motor torque is controlled. Since the motor output axis is directly connected to the rear axis (via fixed gears) the control algorithm is closely connected to the vehicle speed.

Extensive studies of the AC-motor control project have been performed in the thesis. Integration with the motor controller is crucial when moving the pedals to the VMCU. Information in CAN frames received by the motor controller will serve as input to the AC-motor control algorithm. Fur-

thermore, the implementation of a cruise control algorithm relies upon the AC-motor controller for smooth speed control. No report of the master thesis was available at the writing of the thesis, whereof the lack of reference.

1.1.3.2 In-vehicle infotainment demonstrator

The main purpose of Sanell's and Samuelsson's project was to examine the possibility of using available hardware and software when developing infotainment systems for automotive industry and investigate if it could deliver the same performance as custom, self-made systems. The benefit of using already existing hardware and software would be reducing costs.

The work carried out by Sanell and Samuelsson can be seen as the predecessor to the thesis and, naturally, has a lot of similarities. Both trying to develop heads-up displays to present information to the driver on predefined hardware and operating systems. Sanell's and Samuelsson's heads-up display were created using QT (a framework for creating graphical user interfaces) on a MeeGo Linux based system whereas the thesis runs the Ångström distribution on slightly different hardware and with the GUI now developed in Java Swing. While the in-vehicle infotainment demonstrator thesis explored the possibilities of showing information provided via CAN to the driver this thesis actually implements both CAN interface and HUD, on a live platform. (Hans Sanell and Göran Samuelsson, 2011)

2 Theory

The theory section describes the open automotive standard AUTOSAR and the dominating communication protocol in vehicles, Controller Area Network. The in parallel on-going project CHARTER, serving to introduce model driven development in embedded systems, is also considered.

2.1 AUTOSAR

The automotive industry is getting complex and diversified within the field of Electrics/Electronics (E/E) implementation. Therefore automobile manufacturers, suppliers and tool developers have developed an open standard called AUTOSAR, AUTomotive Open System ARchitecture. It is being developed and established as an interface between hardware and software to ease the reuse of applications and other software modules. AUTOSAR also improves flexibility in case of software modifications and eases scalability within and across product lines, cutting costs as a result. (Adam Hulin and Marcus Johansson, 2011)

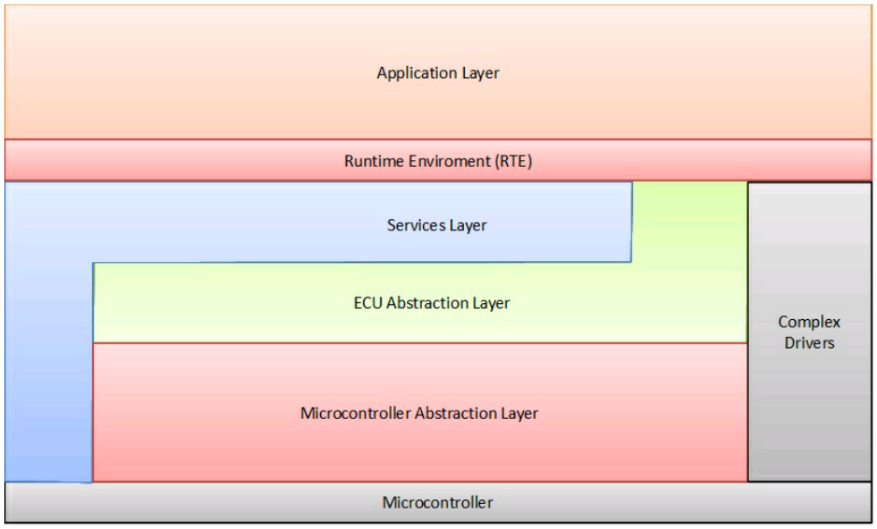


Figure 3: AUTOSAR software architecture including the three layers and the important standardised interfaces. (Johan Elgered and Jesper Jansson, 2012)

AUTOSAR consists of multiple layers, see Figure 3, the basic software layer (BSW), the runtime environment (RTE) layer and the applications layer.

2.1.1 BSW

The lowest layer is called basic software, which consist of Microcontroller Abstration Layer, ECU Abstratcion Layer, Services Layer and Complex Drivers. The Microcontroller Abstraction Layer receives calls from higher levels and forward these calls in a standardised way to the hardware and peripherals. The BSW is the only layer with direct access to the microcontroller. The basic software layer also incorporates the operating system and interfaces to the upper layer, called RTE. (AUTOSAR, 2012)

2.1.2 RTE

The RTE is the layer in between the BSW and the applications layer. The RTE-layer makes communication in underlying layers transparent to the developer programming software components. Since SW-Cs are dependent on the ECU hardware, the RTE has to be customised for the ECU on which it runs. ECU-specific configuration therefore leads to that RTEs differentiate between different ECUs. (Johan Elgered and Jesper Jansson, 2012)

2.1.3 Application layer

The applications layer consists of multiple software components, often referred to as SW-Cs. The SW-Cs comes in two types, application software component type and sensor actuator type. The latter is software representing a sensor or an actuator whereas the former provides actuators with input by collecting sensor data. (AUTOSAR, 2012)

All actual functionality in the vehicle is developed in the software components. The purpose of the BSW and the RTE layers is simplifying and standardising development of the software components, increasing reusability and modularity. These properties makes AUTOSAR cost saving since development and testing effort for software components can be reduced. SW-Cs can be ported to other target platforms without spending time on adaptation to new environments (Joakim Plate and Peter Fridlund, 2011)

The modular concept of course has drawbacks, mainly performance issues due to increased processor and memory usage. Standardised software will always take detours that single-purpose software can avoid. However, with today's increased hardware performance this may be a price worth paying for increased software reusability. (Joakim Plate and Peter Fridlund, 2011)

2.2 CHARTER

Embedded systems are traditionally developed in low-level languages like C or Assembly as these gives the programmer good control of time critical events. However, in big and complex systems the vast number of lines of code can make the system difficult to overview. Therefore a project

called CHARTER (Critical and High Assurance Requirements Transformed through Engineering Rigour) was started, striving to introduce high-level languages and model driven development in critical embedded systems and evaluate performance compared to traditional low-level languages.

QRTECH has good experience of time critical embedded systems, mainly due to involvement in automotive industry, and the company's role as a CHARTER project member is to implement code of high-level languages (especially real time Java) in critical embedded systems, in this case the go-cart.

CHARTER is a ARTEMIS Embedded Computing Systems Initiative project partnership between the European Commission, member states (UK, Sweden, Ireland, Netherlands, Germany) and ARTEMISI (a non-profit Industrial Association).

Partnership members except for QRTECH include universities like Chalmers Univeristy of Technology and Dundalk Institute of Technology, but also companies like Impronova (SWE), aicas (GER) and Atego (UK). (Alec Dorling, 2012)

2.3 Controller Area Network

Controller Area Network (CAN) is a multi-master broadcast serial bus defined by Bosch in the mid-80's. It was initially intended for use in the automotive industry and has been the dominating communication protocol in cars and trucks since the mid-90's. Due to the low cost of CAN controllers it has also become popular in other fields such as industrial automation and equipment for the medical industry. (Computer Solutions Ltd, 2012)

A modern car contains about 50 electrical control units (ECU's), each one often responsible for one feature, for example engine control, airbags, cruise control, doors or mirror adjustment. Actuators and sensors are in turn often connected to the ECU's. The CAN bus connects the ECU's and provides the possibility for the ECU's to communicate with one another. CAN is particular in the sense that nodes are not predefined specific addresses, instead the messages have identifiers. Nodes can therefore listen for specific messages on the bus and ignore those which are not of interest to the node. In the same manner, nodes transmit messages with different identifiers depending on message content and receiver. The message identifier also defines the messages priorities, where a lower identifier corresponds to a high priority and vice versa. (Joakim Plate and Peter Fridlund, 2011)

When the CAN bus is idle, any node on the bus is free to start transmitting. In the case of two nodes starting to send simultaneously, the highest prioritised message (with lowest numerical id) will win the arbitration and fulfil transmission whereas the message with lower priority will sense this, step back and wait for the bus to be free, allowing for the message to be resent after a predefined delay. (Joakim Plate and Peter Fridlund, 2011)

This built in prioritisation of messages is achieved using recessive and dominant bits, '1' being recessive and '0' being dominant. If the bus is idle (no one is sending) the bus is in it's recessive state (high). If a node sends a dominant '0'-bit, it grounds the bus, drawing it down to '0'. So, if a dominant bit is sent on the same time as a recessive bit, the dominant bit is displayed on the bus. The node sending the recessive bit sees that the bus is '0' even if it sent a '1', and a collision is detected. If all nodes have unique identifiers, which is necessary, a single node remains as winner when a full identifier has been transmitted. This node now continues to send its data. (Joakim Plate and Peter Fridlund, 2011)

There are two CAN specifications, basic (standard) CAN and full (extended) CAN. Basic CAN offers a maximum transfer speed of 250 kbit/s and an 11 bit message identifier whereas the full CAN is capable of 1 mbit/s with a 29 bit identifier. These transfer speeds are defined for 50 meter bus length. By lowering speed the bus length can be increased, for example 125 kbit/s at around 500 meter extended CAN. There are a lot of possibilities when choosing physical media for CAN, the most common being twisted pair on a 5V differential signal. The CAN-low (CANL, CAN-) holding -2.5V and CAN-high (CANH, CAN+) holding +2.5V. This makes the physical layer robust even in noisy environments such as vehicles. The bus should be terminated with 120Ω-resistances in both ends between CAN-low and CAN-high to prevent interference due to reflections. A small CAN bus containing three nodes can be seen in Figure 4. (Robert Bosch GmbH, 2012)

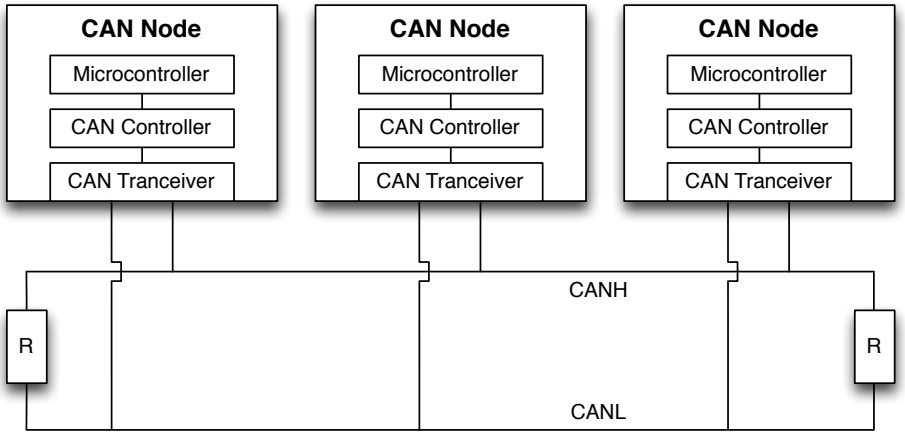


Figure 4: A CAN bus with the two differential signals CANL and CANH, terminated with 120Ω resistances.

As for the physical media, connector types are not yet formally specified and custom designs are common. However, the 9-pin D-sub connector with

the following pin-out, has become popular.

- pin2: CAN-low (CAN-)
- pin3: GND (Ground)
- pin7: CAN-high (CAN+)
- pin9: CAN V+ (Power)

CAN offers four different types of frames, remote frame, error frame, overload frame and data frame. The data frame comes in two formats, basic and extended. The extended data frame is the only one used in the thesis and its frame layout is showed in Table 1:

Name	**Bit-length**	**Description**
SOF	1	Start-of-frame.
ID part 1	11	First part of message identifier, also represents message priority
SRR	1	Recessive (0)
IDE	1	Identifier extension bit
ID part 2	18	Second part of message identifier, also represents message priority
RTR	1	Dominant (0) (see Remote Frame below)
Reserved bits	2	Reserved bits, dominant (0)
DLC	4	Data length code, number of bytes of data (0-8)
Data	0–64	Data to be transmitted (bytelength corresponds to DLC field)
CRC	15	Cyclic Redundancy Check, Checksum
CRC delimiter	1	Always (1)
ACK slot	1	Transmitter sends recessive (1), any receiver ACKs with dominant (0)
ACK delimiter	1	Must be recessive (1)
EOF	7	End-of-frame, must be recessive (1)

Table 1: The layout of the CAN data frame.

Normally CAN frames are sent cyclically, messages can for example be sent every 10 or 100 ms. Using this manner, errors can be detected. If a specific message stops arriving, nodes will realise there is an error occurrence, making adequate actions. Also, bus overload can be avoided since bus load is kept close-to constant if all messages ”always” are sent with a specific cycle time. CAN buses in automotive industry often have a bus load around 70-80%.

CAN is well suited in vehicles for a lot of reasons. It is cheap and reliable in harsh and noisy environments. Being message based also favours CAN when deciding for a network type for the automotive industry, since certain data often is relevant for several nodes in the vehicle and data consistency

is important. CAN also has the ability to automatically drop faulty nodes on the bus to prevent these from bringing the network down. This fault confinement ensures bandwidth for critical transmissions. (Corrigan, 2002, revised 2008)

Another commonly used construction of CAN frames in the automotive industry is the usage of the update bit. It is a single bit which is set to 1 if the corresponding signal has been updated, assuring the receiver that the transmitter successfully managed to send a new updated value of the signal in the most recent frame. (Joakim Plate and Peter Fridlund, 2011)

2.4 PID control

The Proportional-Integral-Derivative controller (PID controller) is a commonly used controlling mechanism within industry and other control intensive environments. It uses an error (calculated as the difference between a defined setpoint value and the output signal from the process) to adjust the input signal to the process which it controls. The PID-controller is known as three-term control since it uses a proportional gain for the present error, integral gain for the summation of the past errors and a derivative gain for the "future" error, based on the current rate of change. The sum of the three terms multiplied with the error adjusts the input signal to the process. Sometimes, one or two of the terms are discarded, since only one or two of the terms are required to control the system. This is achieved by setting the term(s) to discard equal to zero and the result is for example a P- or PI-controller. (Lennartsson, 2000)

2.5 Hardware Specifications

This section covers technical details about the electronics used in the thesis. Other hardware, such as the motor and the batteries, are not described in the section. However, it is inevitable not to mention them, since these are controlled by the electronics.

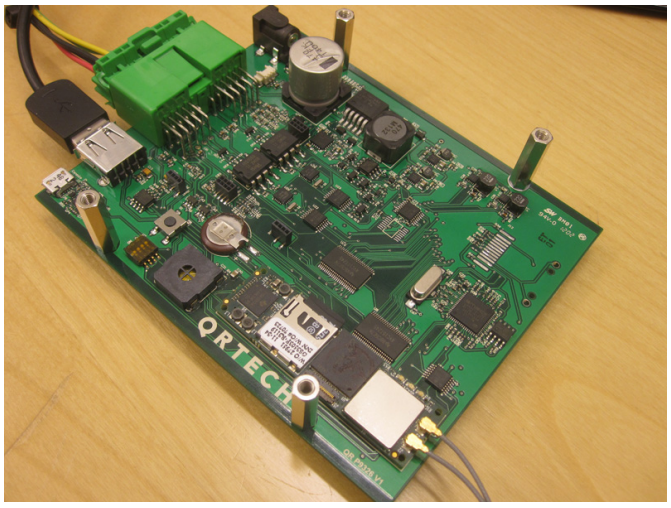
2.5.1 VMCU and BMU

The hardware used as VMCU, illustrated in Figure 5, is developed by QRTECH. It is based on the Gumstix Overo Earth with a 720 MHz ARM Cortex-A8 CPU and 256MB RAM with a WLAN-controller, but with some added features

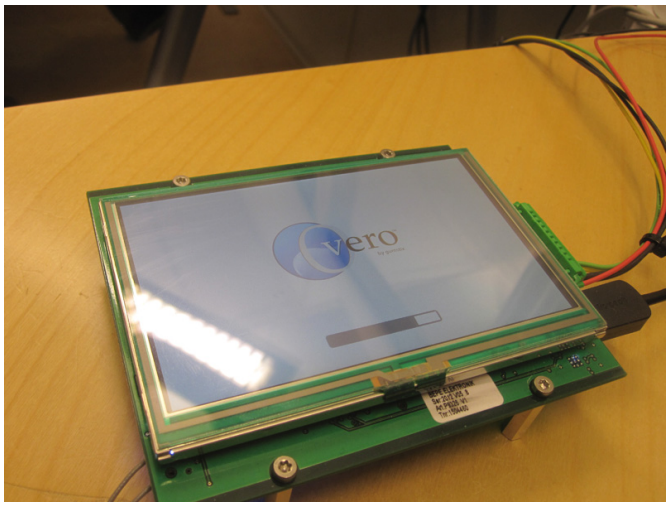
- 5" resistive touch display
- Texas Instruments CAN controller ISO1050TI

The VMCU is running the lightweight Linux distribution Ångström. The Ångström community originates from the OpenEmbedded, OpenZaurus and OpenSimpad projects aiming to develop a stable, user friendly Linux distribution for embedded devices. (koen, 2011)

Using a Linux based OS like Ångström involves a lot of advantages as compared to a self-developed one, the largest being cutting development costs.



(a) The backside of the VMCU.



(b) The VMCU booting up the Ångström Linux distribution.

Figure 5: The ARM-based VMCU board. Notice the SD card holder containing the Ångström OS

The BMU consist of the same hardware, running the same operating systems, as the VMCU. The difference is that it lacks the 5” display.

2.5.2 Motor controller

The Texas Instruments developed Digital Signal Processor (DSP) F28335 Delfino is equipped with a 32 bit 150 MHz CPU, 3.3 V I/O design and two CAN channels. It has 68 kB RAM and 512 kB Flash memory. Delivered with the DSP was also a docking station, which is a small motherboard giving access to all GPIO and ADC signals of the DSP.(TI and Community contributors, 2012*b*)

The DSP mounted in the slot on the control board of the electric go-cart, can be seen in Figure 6.

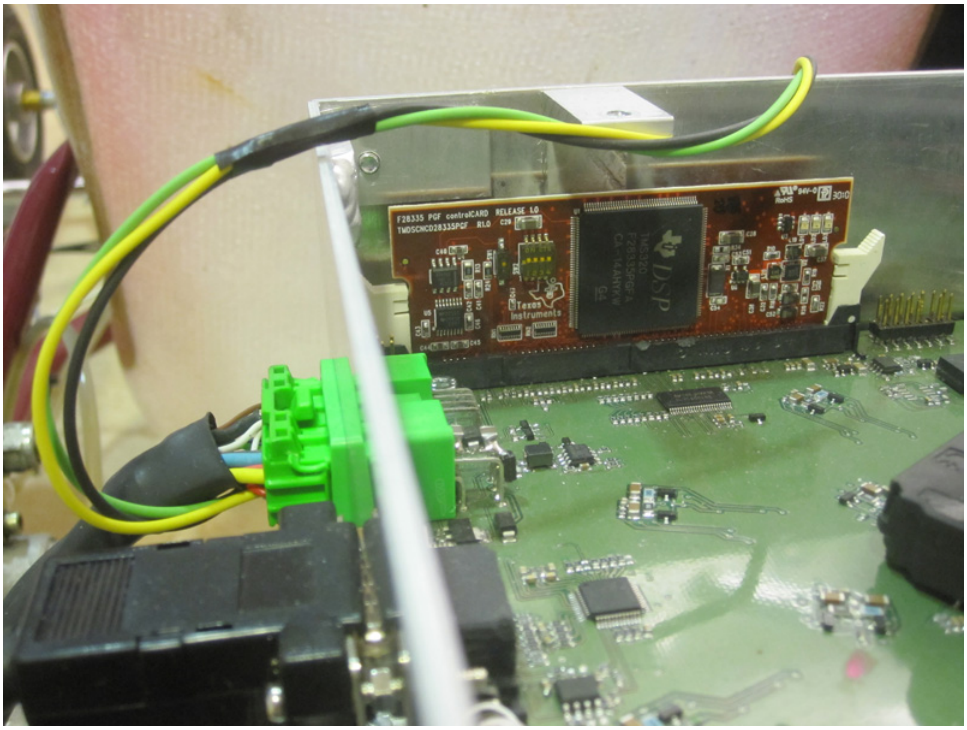


Figure 6: The Digital Signal Processor (F28335) by Texas Instruments.

3 Method

In this section the methods used through out the thesis are motivated and described briefly.

At the beginning of the thesis, the hardware and the main goal were known, making the waterfall model suitable to use. It consists of different levels, where each level represent one phase of the thesis. Due to the phases, the waterfall model is said to be static, aggravating for adoptions to new requirements or features. To avoid the stiffness, the waterfall model is used in combination with the agile modelling technique, which increases the dynamics and allows for features to be added or removed during the progress of the thesis.

3.1 The waterfall model

The roots of the waterfall model stretches back to the 1970s, but it is still used widely due to its simplicity, and wherefore it is also used in the thesis. As can be seen in Figure 7, the waterfall model is divided into different phases. In the first phase is the requirements (which originates from the end user's, in the thesis case QRTECH's goal) defined as a set of functions and constraints. A validation of these as well as the possibility of incorporating them are also done in the first phase. In the second phase, the requirements are used to make an overall system design, giving an idea on how the system is going to work and look like for the end user. In the third phase, the implementation and unit testing phase, the system is realized as a set of independent units/modules (heads-up display, cruise controller, CAN interface etc). In the fourth phase, the units/modules developed in the previous phase are put together to make a complete system. Checks and testing are performed to verify that the units/modules coordinates properly with each other and that the entire system works as specified. If everything works properly, the system is delivered to the end user and might be maintained (if agreed upon at the beginning of the project) when necessary. (Nilesh Parekh, 2011)

3.2 Agile model

After a while it was realized that the stiffness of the waterfall model needed to be complemented with a more volatile method. Agile modelling is a modelling technique which developed in the 1990s as a reaction to the existing heavy regulating methods, serving to make projects open to inputs and to 'move quickly'. Less focus is put on documentation between different phases, and instead is the development allowed to take sudden turns and changes. A close interaction with the end user through the whole developing process is often significant, which usually lead to a lot of prototypes before the final

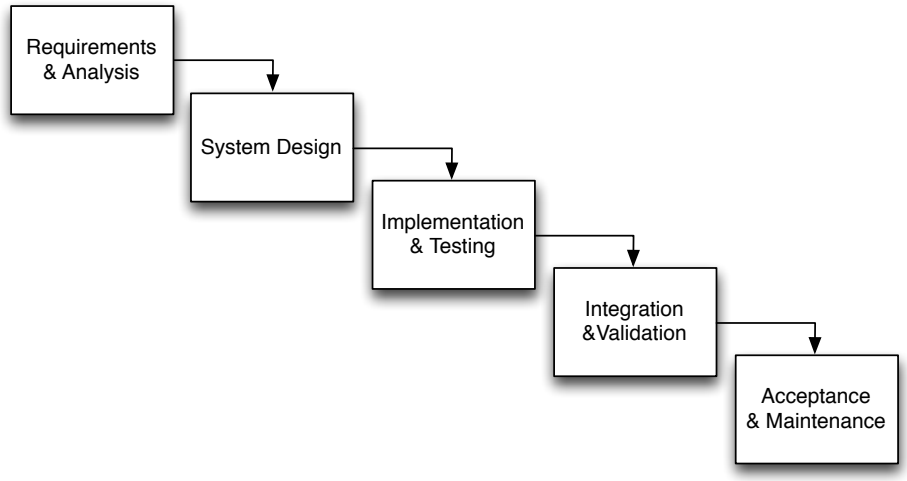


Figure 7: The waterfall model.

product is reached. (Gray Pilgrim, 2012)

3.3 Working process

Throughout the realization of the thesis the methods mentioned above was followed. Utilizing the thesis specification in combination with discussions and ideas from colleagues at QRTECH as a starting point, an overall thought of the system was defined. In parallel was previous work studied to widen the horizon and to get understanding of existing implementations on the go-cart.

During the development of software components, testing environments (consisting of equivalent hardware that is available on the go-cart) was set-up at the bench, allowing for verification throughout the process. Hopefully this approach resulted in elimination of some early teething troubles. After completion of the individual software components, they were tested on the blocked up go-cart one-by-one and finally altogether composing the entire system.

3.4 Development Environments

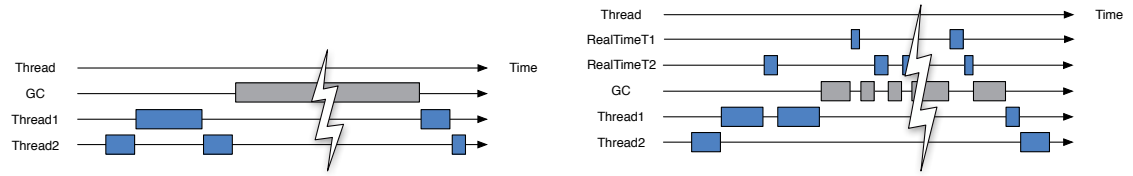
This section describes the different software used to simplify the work throughout the thesis. Also, an description of the test-bench set-ups, is given.

3.4.1 Eclipse Java IDE

Eclipse IDE for Java Developers is an Integrated Development Environment (IDE) featuring code-completion, syntax highlighting and search functionalities as well as giving the possibility to test-run the class files in the Oracle Java Runtime Environment (JRE) with a simple click, making development and testing effective and easy. (The Eclipse Foundation, 2012)

3.4.2 JamaicaVM

JamaicaVM is an application written by the CHARTER member aicas. Aligning well with CHARTER's aim, to introduce Java in embedded systems, JamaicaVM is designed with this in mind. JamaicaVM is a virtual Java machine and a build environment for real-time Java. The use of three different kinds of threads, RealtimeThread, NoHeapRealtimeThread and Thread, ensures that execution performs optimally. JamaicaVM does also support real-time garbage collection, allowing for threads of type RealtimeThread and NoHeapRealtimeThread to interrupt the garbage collection mechanism to met desired real-time criterion, see Figure 8. JamaicaVM is available for three host systems, Windows, Solaris and Linux. From there, it compiles and builds Java code to produce binaries which executes on several CPU's, including x86, PowerPC and ARM. (aicas GmbH and aicas incorporated, 2012)



(a) With classic Java, threads can be interrupted by the garbage collector mechanism.

(b) With RTJ, real-time threads can interrupt the garbage collector mechanism.

Figure 8: Illustrating the difference of how the garbage collector works with Java and RTJ.

3.4.3 Code Composer Studio 5

The C-code for the TI DSP F28335 in the dock (described in 2.5.2) was developed under Windows XP in a Eclipse-based development platform called Code Composer Studio 5 (CCS5), provided by Texas Instruments. In the same way as the Eclipse Java IDE, CCS5 features code-completion, syntax highlighting and search functionalities but also provides support of breakpoint debugging, machine code generation and erasing/writing of flash

through USB. A drawback is that CCS5 took a few minutes to build the machine code and flash the non-volatile memory, making the development cumbersome. (TI and Community contributors, 2012*a*)

3.4.3.1 COM-configurator

QRTECH has an in-house developed program called QRtech COM Configurator, capable of generating CAN related C-code from an imported XML-file. The imported XML-file is derived from CANdb++ Editor, a third party software developed by Vector. CANdb++ Editor allows for design of CAN frames, e.g. selecting not only which signals they should consist of, but also bit-length and parameters like offset and type. It does also come with the possibility to set several options, such as how often a frame should be transmitted or if a signals should have an update bit or not. (Vector Informatik GmbH, 2012)

3.4.3.2 CANoe

CANoe is a PC software used to analyse a CAN network and its nodes. Nodes, which should be a part of the network, can be simulated to test the behaviour of all ready existent nodes. By importing a CAN database, frames on the CAN bus can be monitored and transmitting frames can be tailor-made through simulated nodes, even under runtime. CANoe also features bus load calculation and the possibility to draw signal graphs. (Vector Informatik GmbH, 2012)

The software uses a small box called CANcaseXL which is connected to the host machine through USB. The CANcaseXL is in its turn connected to the CAN network using a twisted pair cable with D-SUB9 connectors. In Figure 9 CANoe is used to simulate the VMCU sending the frame containing the vehicle state and requested torque to the DSP, on the go-cart. The green graph shows the actual torque generated by the motor and the black graph shows the vehicle speed. In the lower-left quarter of the screen-shot, the transmitting frame can be edited. In the lower-right quarter, all frames and its signals on the bus can be viewed.

3.4.4 Test-bench

Two general test-bench set-ups were used in the thesis. The set-up represented by the dashed box in Figure 10 was used for developing the CAN interface for the motor controller. On a PC running Windows, code was edited, built and uploaded (via USB) to the DSP with Code Composer Studio 5. The PC was also running CANoe and a CANcaseXL was connected to the PC via USB. With a twisted pair CAN cable with DSUB-9 connectors

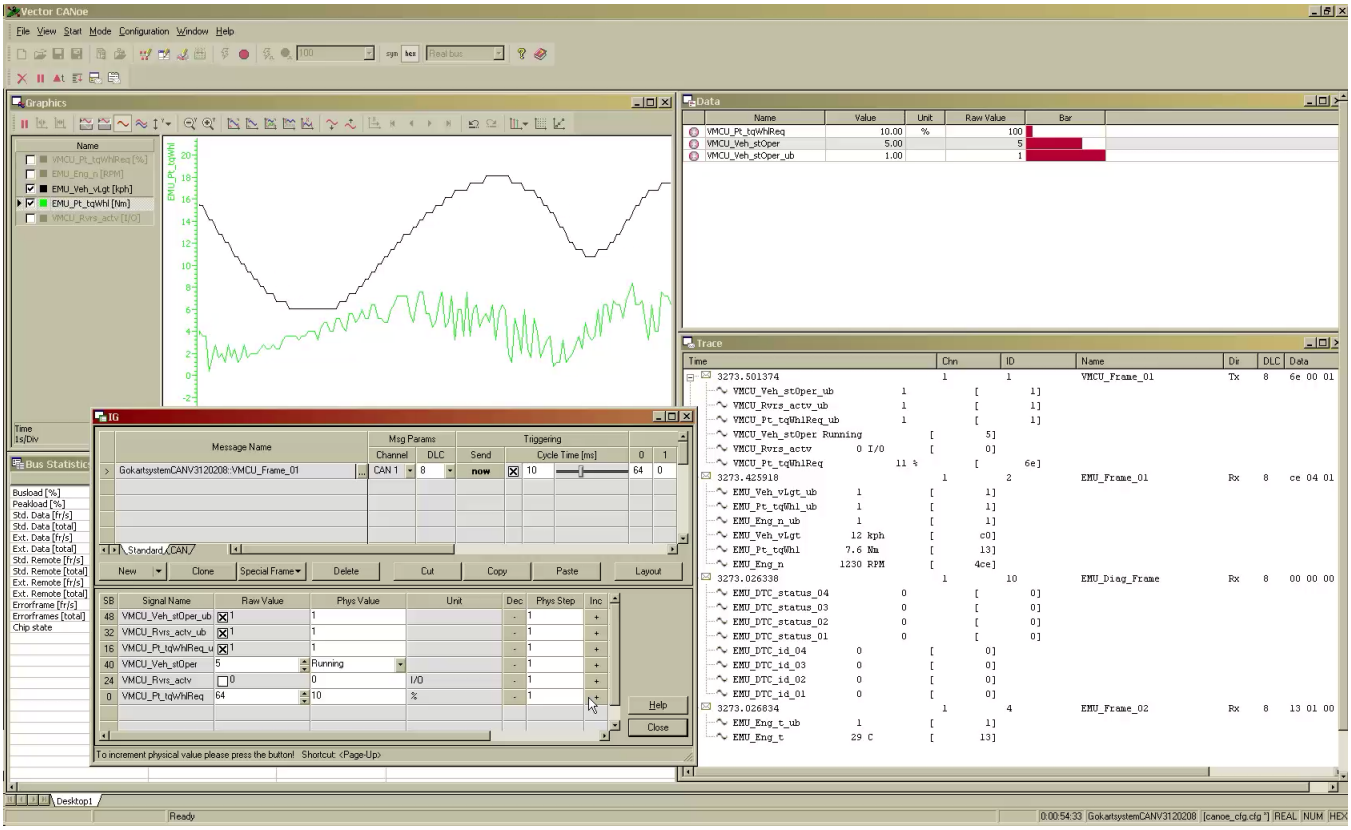


Figure 9: Here CANoe is used to simulate the VMCU node when developing the CAN interface on the motor controller.

the DSP's CAN controller was interconnected to the CANcaseXL. Furthermore, especially in the first half of the thesis, an oscilloscope was used to monitor the duty cycle of the PWM-signals from the motor controller.

The dashed-dotted box in Figure 10 represents the heads-up display test-bench. The GUI was designed on Windows XP in Eclipse Java IDE and test run with the Java Runtime Environment (JRE). When testing on target hardware, the VMCU, JamaicaVM on Ubuntu Linux 11.10 was used to cross-compile and build the binary. The executable was then transferred from the PC to the VMCU via the intranet.

With a complete AUTOSAR architecture in place on the VMCU, the CAN controller was tested by connecting it to the CANcaseXL in the same manner as for motor controller development. However, this time CANoe simulated the CAN controller of the DSP or the actual DSP itself was connected.

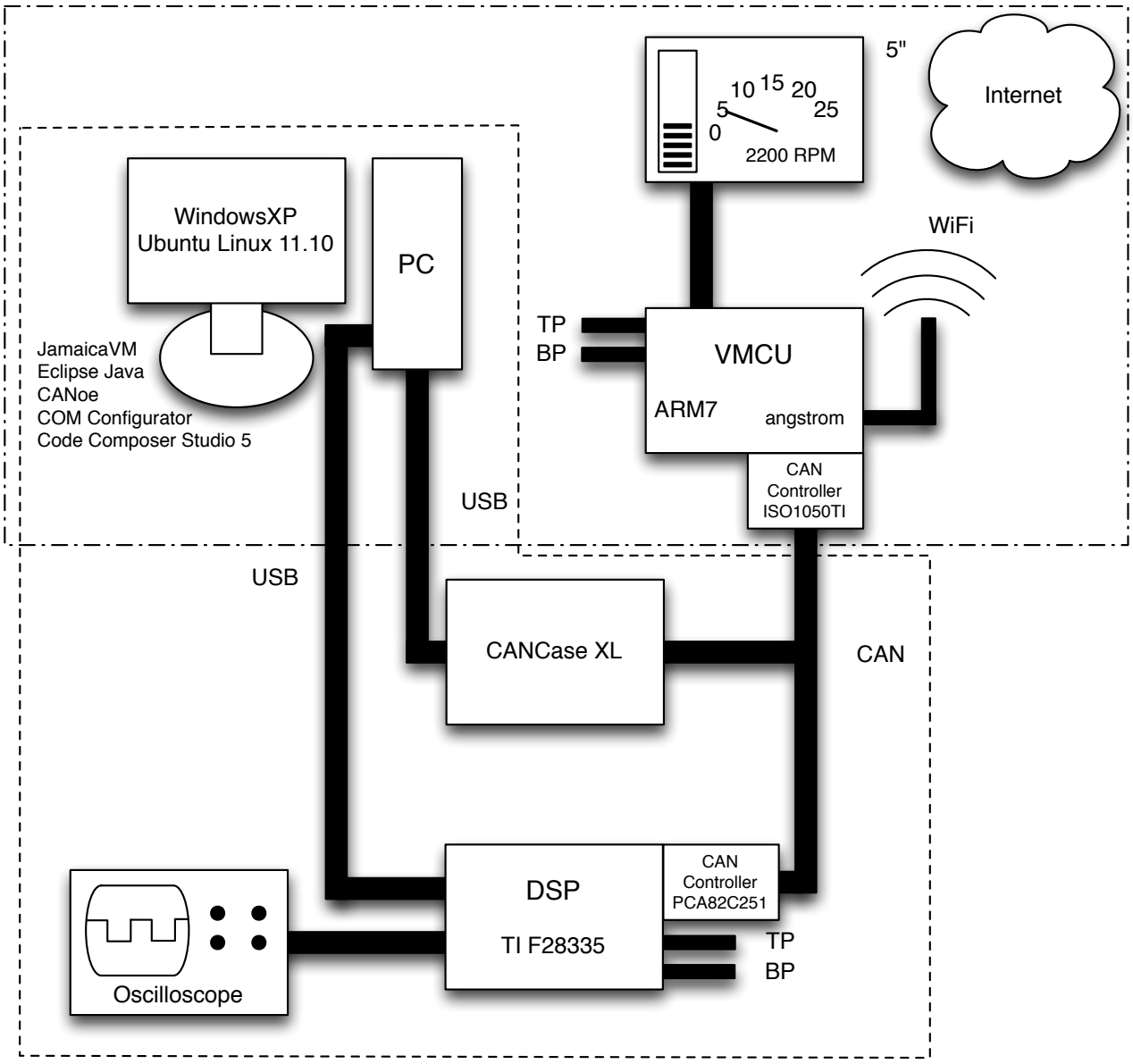


Figure 10: The test bench set-up.

4 Implementation

This section describes the design of the AUTOSAR software components, the heads-up display and the cruise controller. The implementation of the CAN interface on the motor controller is also treated.

4.1 Heads-up Display

The graphical user interface (GUI) for the heads-up display was developed in Java using the Java libraries awt and Swing. (Skansholm, 2005) The use of more extensive graphical frameworks such as Java FX was desired but discarded due to low or non-existent Ångström support.

A non-complete but working RTE-layer, together with a test interface for setting variables/signals in the RTE-layer, was early integrated with the heads-up display, allowing for testing and validation throughout the process. (For a reminder of RTE, see Section 2.1.2)

The heads-up display consists mainly of Java Swing components, such as JPanels and JLabels. To place the components in the JFrame, a combination of the classes FlowLayout and TableLayout were used. TableLayout is a custom non-standard Java class which allows for exact positioning in a grid-like structure, making the placement of components easier compared to using the default layouts in Swing (`http://java.sun.com/products/jfc/tsc/articles/tablelayout/`).

At first, vehicle data such as speed, temperature, revolutions per minute and state of charge, was presented to the driver using JPanels and JLabels. The speed is also displayed in a more convenient way, using the drawline component in Swing's 2D Graphics library, which was used to draw a needle in a velocimeter. To create the velocimeter, eight JLabels (each containing digits representing a speed) were placed in a third of a circular arc, with an equal distance between each other, see Figure 11a.

The drawline method paints a line between two defined coordinates. Thus, by calculating new coordinates for one of endpoint of the line, it is possible to make the needle rotate and follow the circular arc according to the changes in speed.

To make the heads-up display more attractive, an external font was used. The font can be described as having a retro-digital-watch-look, based on a seven segment display. It had also to be monospaced to avoid movements of text, since the letters did not have the same width. The font used is DS-Digital and is a shareware font created by Dusit Supasawat (`http://www.dafont.com/ds-digital.font`).

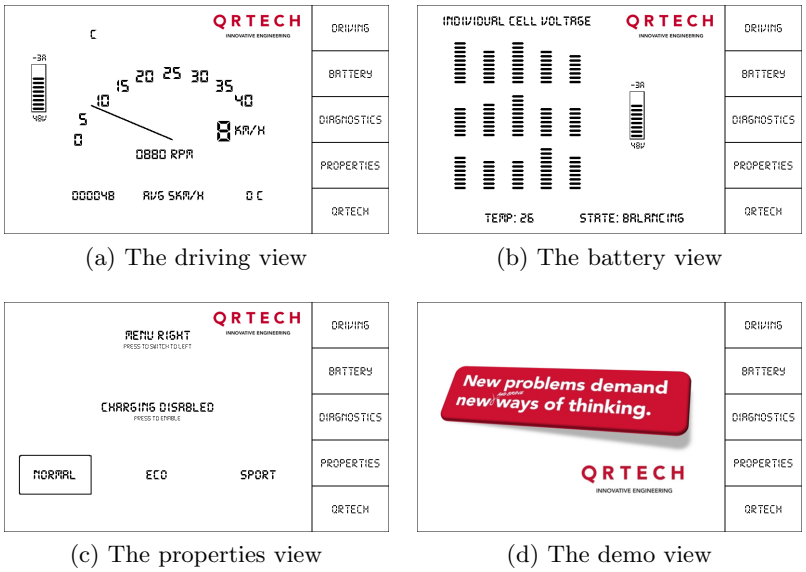
The battery indicators used in the driving view (Figure 11a) and the battery view (Figure 11b), are based on another 2D Graphics GUI Swing component, namely the drawrect. An instance of this component creates the rectangle representing the border of the battery. To indicate the state

of charge, the standing rectangle is filled with the corresponding number of underlines stapled one another, see Figure 11a.

Since the needle, especially when moving, looked jagged, the Java rendering hints parameters were changed (for example enabling anti-aliasing) to improve the appearance of the graphical representation. The battery indicators also used this setting.

The heads-up display, as explained in 1.1.2, consists of five views: driving, battery, diagnostics, properties and demo. The option to switch between different views is available through the menu to the right. It contains five JLabels, each with an attached ActionListener, setting the visibility of the pressed view to true and all others to false when clicked. It also prohibits unnecessary operations to be made on views that are not visible, increasing performance.

A picture containing the views can be seen in Figure 11.



(a) The driving view

(b) The battery view

(c) The properties view

(d) The demo view

Figure 11: The different views of the heads-up display.

The driving view is the main view providing information to the driver about the vehicle's current state. It shows the speed, both with the velocimeter and a numerical value, the average speed, the total distance driven, revolutions per minute, the motor temperature and power status, such as current drain and state of charge. See Figure 11a.

The battery view presents the state of the individual cells in the battery and general battery status, such as state of charge and temperature, see Figure 11b.

The diagnostics view shows a button, providing the possibility to clear DTC:s after the driver has been informed about the occurrence of the error.

In the properties view, the driver can select which driving profile to use, enable or disable battery charging and decide if the menu should be placed to the left or to the right. See Figure 11c.

In the demo view, the driver is shown a brief presentation about QRTECH. A power point presentation was segmented into pictures which are displayed with the media player delivered with Ångström, mplayer, showing each slide for 5 seconds. See Figure 11d.

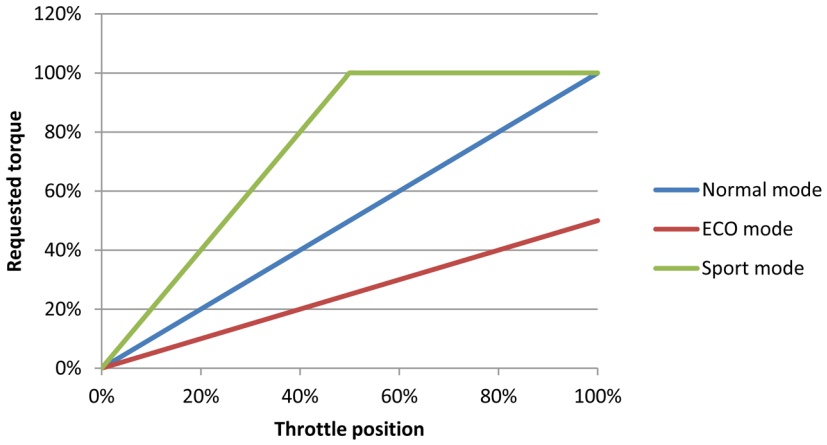


Figure 12: Mapping of throttle pedal position to the reqeusted torque, for the three different driving profiles.

Another feature requested by QRTECH, is the option to choose between different kinds of driving profiles. Therefore, three profiles are built into the go-cart, Normal mode, where the throttle position is mapped 1:1 to the torque request, ECO mode, where the requested torque is mapped 1:2 to the throttle position, and Sport mode, where the request torque is mapped 2:1 to the throttle position and set to maximum torque when the pedal is pressed halfway or more, see Figure 12. The profile selection is made in the properties view.

4.2 Cross-compilation for ARM with JamaicaVM

When a fairly complete version of the heads-up display was ready it was compiled and built using JamaicaVM. This was done under Ubuntu 11.10 with the JamaicaVM cross-compiler for ARM7, referring to the class containing the main method, using the command:

```
jamaicac -useTarget linux-arm-le -verbose -d ../classes
RTE/Rte.java
```

which compiles the Java code, essentially building class files in the same manner as the standard Java compilation command (javac). Typing:

```
jamaicabuilder -target=linux-arm-le -XstaticLibraries "stdc++"
-verbose=1
-resource+=ds-digi.ttf:ds-digii.ttf:ds-digib.ttf:ds-digit.ttf
-object=BSW/SocketCAN/BSW_SocketCAN_SocketCAN.o RTE/Rte
```

then uses the class files to generate C-code which is built to a binary executable on the ARM platform. Note how the custom fonts are added to the binary as resources and how the JNI coded in C++ is passed to jamaicabuilder using the -object flag. Since jamaicabuilder needs a few minutes to build the 18MB binary, the heads-up display was developed and tested as far as possible in the Java virtual machine (JVM) before building with jamaicabuilder. However, when a binary was built, it was transferred to the VMCU using its wireless internet connection. On the target system it was run (after setting execution flags) with:

```
chmod +x Rte
./Rte
```

4.2.1 Profiling

It was discovered that execution of the binary generated by JamaicaVM was a lot heavier than running the class-files in the JRE. The solution was found in the profiling feature of JamaicaVM. By calling the profiling flag when running jamaicabuilder, a special version of the binary is generated:

```
jamaicabuilder -target=linux-arm-le -XstaticLibraries "stdc++"
-profile -verbose=1
-resource+=ds-digi.ttf:ds-digii.ttf:ds-digib.ttf:ds-digit.ttf
-object=BSW/SocketCAN/BSW_SocketCAN_SocketCAN.o RTE/Rte
```

When running this version on the target system (ARM), the program checks which parts of the program that is slow and/or uses a lot of CPU cycles and from this information generates a so called profile. The binary is then rebuilt passing the generated profile to jamaicabuilder using the -useProfile flag. Setting options such as number of threads, heap size and how much of the code that is compiled also increases execution speed.

```
jamaicabuilder -target=linux-arm-le -XstaticLibraries
"stdc++" -useProfile ../RTE.Rte.prof -numThreads 20
-percentageCompiled 50 -heapSize=192M -verbose=1
-resource+=ds-digi.ttf:ds-digii.ttf:ds-digib.ttf:ds-digit.ttf
-object=BSW/SocketCAN/BSW_SocketCAN_SocketCAN.o RTE/Rte
```

The new binary is now tailor-made for maximal performance with satisfying results.

4.3 Motor controller and CAN interface

There are two primary parts of the DSP program: the PWM-interrupt and the software interrupt.

The PWM-interrupt serves to control the electric motor, it is triggered at a 20 kilohertz rate and is essentially the work of the previous master's thesis on the go-cart at QRTECH (see section 1.1.3.1). Its purpose is to read sensors providing information such as winding currents and temperature as well as reading the throttle and brake pedals and from this data react and control the current supplied to the motor and its fan by altering the duty cycle of PWM-signals.

In the thesis a new version of this routine was created. In the new version update bits in the CAN frames are set when signals such as motor speed and temperature are recalculated. Furthermore, the throttle and brake pedals are not read by the analog/digital converter measuring potentiometers. Instead, the variable taking care of the requested torque is set by reading an incoming signal in a CAN frame.

Normally, when the PWM-interrupt is not executing, the DSP is busy waiting in the main function. However, every tenth millisecond a software triggered CPU-timer permits the DSP to execute code which purpose is to serve the CAN controller. The CPU-timer allows for well-timed periodic reception and transmission of CAN frames. Also, no code serving CAN communication is executed during the PWM-interrupt and thus not delaying the PWM-signal generation to the motor. This is important since motor control performance is dependent on an agile PWM-routine.

When the software interrupt is triggered, the DSP reads the update bits of all signals received. If the update bit is set, the value of the corresponding CAN signal is copied to an on-memory allocated variable. However, if a maximal number of update bits have been read in its low state, an error is triggered.

When the variables have been updated, the routine checks the state of the vehicle operating state variable and takes appropriate action. There are four states:

- Shutting down. CAN transmission is stopped, however, the DSP is still monitoring the CAN bus.
- Initiating. CAN communication is triggered and the DSP starts transmitting frames.
- Charging. The batteries are charged by the BMU.

- Running. The DSP packs the transmitting frame with vehicle data, i.e. vehicle speed, motor RPM, motor torque (current) and temperature and tells the CAN controller that data is packed and ready to be transmitted.

By reading an input pin called CAN-ON, the DSP decides which version of the code to run. If CAN is off, the old interrupt routine (reading the throttle and brake pedals via A/D-converter) is used and the software interrupt does nothing. Instead, if CAN is on, the new altered version of the PWM-interrupt is used and the software interrupt takes care of CAN reception/transmission as described above.

4.3.1 CAN bus performance

During normal execution of the DSP and the VMCU, it was observed with help from CANoe, that the CAN bus load was around 35%. At the same time, a stress test was performed by decreasing the periodicity of certain frames, resulting in a bus load of 80% (which is common in automotive industry), without any peculiar behaviour.

4.4 Cruise controller

To control the PWM-signal to the motor, a PID is used. The cruise controller was implemented by adding a second PID to control the input to the motor PID.

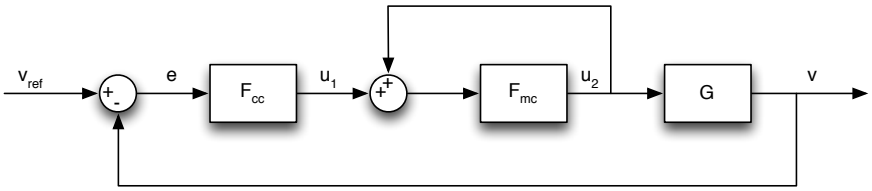


Figure 13: Block diagram of the go-cart including the cruise controller.

The original controller for the motor is illustrated as the block F_{mc} in Figure 13. Its input signal is a torque request, which is read from the CAN bus. When the cruise controller is active, the reference value is not the torque but instead a by the driver defined speed, v_{ref}. Therefore, an extra PID block is added before the original system to adjust for the error between v_{ref} and v. If the go-cart moves too slowly, the F_{cc} regulator has to increase the requested torque (u_1) to accelerate the go-cart, and vice versa.

If the actual speed equals the desired speed, the motor controller reuses the latest torque request as its input, since this torque request resulted in the desired speed.

The integration of a cruise controller was first made on the DSP, even though it is supposed to be present on the VMCU at the end of the thesis. This, as the reconstruction of the go-cart was delayed (thus making it available for testing longer than expected), and the fact that the CAN interface still was unavailable on the VMCU. Therefore was the PID implemented directly on the DSP, as it ought to behave in the same way as if it would have been implemented on the VMCU. The regulator parameters for the PID-controller was selected using manual tuning. K_I and K_D was zeroed and K_P increased until the output started to oscillate. Then K_P was halved and K_I increased until the error offset was corrected in a few seconds. D was not needed and ignored. After a final optimisation by trial and error, the parameter selection was finally $K_P = 0.6$, $K_I = 0.005$ and $K_D = 0.0$ for the F_{cc}.

Having the go-cart and a VMCU with working CAN interface accessible again, the DSP cruise controller implementation was ported from C to real-time Java and implemented as an AUTOSAR software component, running on the VMCU. At the same time, new features were added. For example, the speed of the go-cart must exceed 4 km/h to be able to activate the cruise controller at all. Also, with the cruise controller activated, it is possible to overtake another go-cart using the throttle pedal to exceed the cruise controller reference speed. When the overtake is done and the throttle pedal released, the cruise controller continues to control the speed, using the earlier reference speed. Of course, pressing the brake pedal deactivates the controller.

5 Discussion and Conclusion

This chapter presents the final conclusions and discusses problems encountered during the thesis. It explains the outcome and argues the choices made and how issues have been treated and solved. Finally it proposes some subjects suitable for future work.

5.1 Discussion

At the beginning of the thesis, the heads-up display was developed on an unintended platform, resulting in performance issues when later executed on the target platform. Painting and repainting graphics demanded too much of the VMCU, preventing real time processes to be executed in time. To handle this, the update time of the graphical user interface had to be increased (updated less frequently). Also, more sophisticated methods used to improve the appearance of the heads-up display were discarded. It can be added that, according to the developers of JamaicaVM, techniques such as anti-aliasing are slow in JamaicaVM which also supports the decision to discard it.

After integrating the heads-up display with the RTE-layer and the software components it came with, performance was yet again an issue. In contrast to the very deterministic digital signal processor, the behaviour of the VMCU was inconsistent. Activating the CAN interface made the GUI slow and unresponsive. Analysing the bus it could be seen that the frames sent from the VMCU was forming a queue and was therefore delayed. Also, stressing the VMCU by frequently changing view on the heads-up display sometimes caused the CAN communication to lag behind too much. The problem was avoided by increasing the periodicity of the transmitting frame from 10 milliseconds to 20 milliseconds, giving the VMCU more time for repainting the GUI. Increasing the periodicity comes with an additional domino effect. Since certain frames are transmitted less frequently, the data they contain do not need to be updated and calculated as often as before, lowering the load of the VMCU even more.

Error handling, or more specifically, diagnostic trouble codes (DTCs) was part of the thesis description. The codes was to be displayed in a for the purpose dedicated view, the diagnostics view, indicating errors present on the CAN bus nodes. The diagnostic view should also have offered the possibility to clear these errors. Unfortunately, these features had to be set aside, since the work with the performance issues of the VMCU turned out to be time consuming. Even though the DTCs missed out, some other safety features have been built in. For example if the CAN bus for some reason is broken between the VMCU and the DSP, the DSP recognises this circumstance and sets the requested torque to zero, shutting down the motor. In the same manner, the go-cart motor is shut down if too many frames

arrives without the update bit being set.

Unfortunately was the go-cart reconstruction not completed before the end of the thesis, preventing the go-cart being tested on the ground. Even though the cruise controller behaved as expected when the go-cart was blocked up, it is hard to tell how it would behave with the go-cart on the ground with load (driver). Hopefully, only minor control parameter adjustments of the cruise controller is enough to adapt it to the new environment.

5.2 Conclusion

The thesis resulted in the electric go-cart being equipped with a VMCU, with a heads-up display presenting data to the driver, a drive-by-wire feature via CAN and a cruise controller making it possible to maintain a desired speed.

The fact that a JNI had to be written in C++ to be able to access the VMCU CAN controller at all, is of course a general drawback with (real-time) Java. It makes development more complex, adding another programming language to the project. AUTOSAR does serve as a specification for how interfacing between software layers is treated to extend reusability, it does not specify which language to use for implementing these layers. However, it can be concluded that a RTJ approach is possible when implementing AUTOSAR even though some compromising may be necessary when working close to hardware.

It can be concluded that there is space for improvements on the VMCU. Both the heads-up display and the CAN communication on the VMCU can be optimised for performance and stability. Regarding the DSP, it works in a deterministic manner, meeting its deadlines.

Comparing the actual time spent with the Gantt-chart, (see Appendix A), it can be concluded that the time used for the heads-up display was extended, stretching almost the whole thesis. In contrary, the work with the CAN interface and the motor control was finished in a few weeks, while being planned for more than ten weeks. Unfortunately, the DTCs had to be left out for future work.

References

Adam Hulin and Marcus Johansson (2011), Test application generator for autosar systems, Master's thesis, Department of Computer Science and Engineering at Chalmers University of Technology.

aicas GmbH and aicas incorporated (2012), "Jamaicavm—java technology for realtime", *Collected 2012-04-27* . http://www.aicas.com/.

Alec Dorling (2012), "Charter project", *Collected 2012-01-20* . http://charterproject.ning.com/page/charter-project.

AUTOSAR (2012), "About autosar", *Collected 2012-02-03* . http://www.autosar.org/.

Computer Solutions Ltd (2012), "Can - a brief tutorial", *Collected 2012-03-05* . http://www.computer-solutions.co.uk/info/Embedded_tutorials/can_tutorial.htm.

Corrigan, S. (2002, revised 2008), "Introduction to the controller area network (can)", *Application Report* , p. 14.

Gray Pilgrim (2012), "Waterfall model vs agile", *Collected 2012-01-17* . http://www.buzzle.com/articles/waterfall-model-vs-agile.html.

Hans Sanell and Göran Samuelsson (2011), In vehicle infotainment demonstrator, Master's thesis, Department of Computer Science and Engineering at Chalmers University of Technology.

Joakim Plate and Peter Fridlund (2011), Xcp over can and ethernet on autosar, Master's thesis, Department of Computer Science and Engineering at Chalmers University of Technology.

Johan Elgered and Jesper Jansson (2012), Autosar communication stack implementation with flexray, Master's thesis, Department of Computer Science and Engineering at Chalmers University of Technology.

koen (2011), "The Ångström distribution introduction", *Collected 2012-04-04* . http://www.angstrom-distribution.org/.

Lennartsson, B. (2000), *Reglerteknikens grunder*, 4:7 edn, Studentlitteratur.

Nilesh Parekh (2011), "The waterfall model explained", *Collected 2012-03-19* . http://www.buzzle.com/editorials/1-5-2005-63768.asp.

Robert Bosch GmbH (2012), "What is can?", *Collected 2012-03-07* . http://www.semiconductors.bosch.de/en/ipmodules/can/whatiscan/whatiscan.asp.

Skansholm, J. (2005), *Java direkt med Swing*, 5:4 edn, Studentlitteratur.

The Eclipse Foundation (2012), "Eclipse ide for java developers", *Collected 2012-05-02* . `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/indigosr2`.

TI and Community contributors (2012*a*), "Code composer studio (ccstudio) integrated development environment (ide) v5", *Collected 2012-05-04* . `http://www.ti.com/tool/ccstudio`.

TI and Community contributors (2012*b*), "Tms320f28335", *Collected 2012-02-10* . `http://www.ti.com/product/tms320f28335`.

Vector Informatik GmbH (2012), "The development and test tool for can, lin, most, flexray, ethernet, wlan and j1708", *Collected 2012-02-28* . `http://www.vector.com/vi_canoe_en.html`.

A Gantt chart