DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, SWEDEN 2014

Evaluation of Protocols for Transfer of Automotive Data from an Electronic Control Unit

SAM LÖÖF

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

Evaluation of Protocols for Transfer of Automotive Data from an Electronic Control Unit

S A M   L Ö Ö F

saml@kth.se

Master’s Thesis in Computer Science (30 ECTS credits)
Master Programme in Computer Science (120 credits)
Royal Institute of Technology year 2014
External provider was Scania CV AB
Supervisor at CSC was Sten Andersson
Examiner was Anders Lansner

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Nowadays almost all motorized vehicles use electronic control units (ECUs) to control parts of a vehicle's function. A good way to understand a vehicle's behaviour is to analyse logging data containing ECU internal variables. Data must then be transferred from the ECU to a computer in order to study such data. Today, Keyword Protocol (KWP) requests are used to read data from the ECUs at Scania. The method is not suitable if many signals should be logged with a higher transfer rate than the one used today. In this thesis, communication protocols, that allow an ECU to communicate with a computer, are studied. The purpose of this master's thesis is to examine how the transfer rate of variables from Scania's ECUs to a computer can become faster compared to the method used today in order to get a more frequent logging of the variables. The method that was chosen was implemented, evaluated and also compared to the method used today. The busload, total CPU load and CPU load for the frequency used during the experiments, 100 Hz, was also examined and evaluated.

The experiments performed show that the method chosen, data acquisition (DAQ) with CAN Calibration Protocol (CCP), increased the transfer rate of the internal ECU variables significantly compared to the method using KWP requests. The results also show that the number of signals have a major impact on the busload for DAQ. The busload is the parameter that limits the number of signals that can be logged. The total CPU load and the CPU load for 100 Hz are not affected significantly compared to when no transmissions are performed. Even though the busload can become high if many variables are used in DAQ, DAQ with CCP is preferable over KWP requests. This is due to the great increase in transfer rate of the ECU internal variables and thus a great increase in the logging frequency.

Keywords: Electronic Control Unit (ECU), CAN Calibration Protocol (CCP), data acquisition (DAQ), communication protocols.

Utvärdering av protokoll för överföring av fordonsdata från en styrenhet

Sammanfattning

Nuförtiden används styrenheter (ECUer) för att styra delar av ett fordons funktion i så gott som alla motoriserade fordon. Ett bra sätt att förstå ett fordons beteende är att analysera loggningsdata som innehåller interna styrenhetsvariabler. Data måste då överföras från styrenheten till en dator för att data ska kunna studeras. Idag används Keyword Protocol-förfrågningar (KWP-förfrågningar) för att läsa data från Scanias styrenheter. Metoden är inte lämplig om man vill logga många variabler med en högre överföringshastighet än den som används idag. I detta examensarbete studeras kommunikationsprotokoll som tillåter en styrenhet att kommunicera med en dator. Examensarbetets syfte är undersöka hur överföringshastigheten av variablerna, från Scanias styrenheter till en dator, kan ökas jämfört med den metod som används idag för att få en mer frekvent loggning av variablerna. Metoden som valdes implementerades, utvärderades och jämfördes med metoden som används idag. Busslasten, totala CPU-lasten och CPU-lasten för den frekvens som användes under experimenten 100 Hz har också undersökts och evaluerats.

De utförda experimenten visar att den valda metoden, data acquisition (DAQ) med CAN Calibration Protocol (CCP), ökade överföringshastigheten av de interna styrenhetsvariablerna betydligt jämfört med metoden KWP-förfrågningar. Experimenten visar också att antalet signaler har stor inverkan på busslasten för DAQ. Busslasten är den parameter som begränsar antalet signaler som kan loggas. Den totala CPU-lasten och CPU-lasten för 100 Hz påverkas inte betydligt jämfört med när inga överföringar görs. DAQ med CCP är att föredra framför KWP-förfrågningar även om busslasten blir hög för DAQ då den stora ökningen i överföringshastighet av de interna styrenhetsvariablerna medför en mer frekvent loggning av variablerna.

Nyckelord: Styrenhet (ECU), CAN Calibration Protocol (CCP), data acquisition (DAQ), kommunikationsprotokoll.

Preface

This master’s thesis has been performed as a part of the master programme in computer science at the School of Computer Science and Communication (CSC) at the Royal Institute of Technology (KTH) in collaboration with Scania CV AB at the group ECU System Test (REVT) in Södertälje during Spring 2014.

I would like to thank for all support I have got from different people during this master’s thesis. First and foremost, I would like to take the opportunity to express my deepest gratitude to Lars Broander at Scania for his continuous guidance, support and insightful discussions. I also want to thank Christopher Lidbäck at Scania for valuable input and practical support. Further thanks go to Marcus Bergman at Scania for his support. I would also like to thank my supervisor Daniel Norin at Scania and my supervisor Sten Andersson at KTH.

Acronyms

ACK	Acknowledgement
ASAP	Arbeitskreis zur Standardisierung von Applikationssystemen
ASAM	Association for Standarisation of Automation- and Measuring Systems
CAN	Controller Area Network
CCP	CAN Calibration Protocol
CMD	Command code (for CCP context)
CRM	Command Return Message
CRO	Command Receive Object
CTR	Command counter
DAQ	Data Acquisition
DTO	Data Transmission Object
ECU	Electronic Control Unit
ERR	Error code
KWP	Keyword Protocol
ODT	Object Descriptor Table
PID	Packet ID
VCI	Vehicle Communication Interface
XCP	Universal Measurement and Calibration Protocol

Contents

Chapter 1

Introduction

Scania is a leading manufacturer of heavy trucks, busses and industrial engines and marine engines. This thesis was performed at Scania's Research and Development department at the group ECU System Test that is working with testing functionality for *electronic control units* (ECUs) in trucks and busses. An ECU consists of hardware and software and is used to control a part of a vehicle's function. The ECUs read signals continuously from sensors and these input signals, with information from the vehicle, are used to calculate output signals to actuators. The ECUs use the standard *Controller Area Network* (CAN) in order to communicate with each other.

A vehicle's behaviour can be analysed by studying logging data containing ECU internal variables and this is a good way to understand a vehicle's behaviour. The logging data containing ECU internal variables must be transmitted to a computer in order to log it.

1.1 Background

An emulator is hardware or software used to imitate the functionality of other hardware or software. Scania is using a software emulator to test ECU software. Variables, or signals, can be read directly in an emulator which allows many signals to be logged without using any communication protocol. Scania also performs ECU tests in vehicles and in lab environment with hardware-in-the-loop (HIL) *rigs* that simulates the environment of an ECU. Signals cannot be read directly in rigs or in vehicles. Today, a signal is read in a rig via the protocol Keyword Protocol 2000 (KWP2000). The method of reading variables via KWP is ineffective due to the fact that the computer must first request a variable, then the ECU shall process the request and send a response. A more effective method to transmit signals, from an ECU, would be of great importance.

1.2 Purpose and goal

The purpose of this master's thesis is to examine how the transfer rate of variables from Scania's electronic control units (ECUs) to a computer can become faster compared to the method used today in order to get a more frequent logging of the variables. The method that is chosen will be implemented, evaluated and also compared to the method used today. The busload, total CPU load and CPU load for the frequency used during the experiments 100 Hz, will also be examined and evaluated.

1.3 Delimitations

This study is delimited to communication protocols used in the automotive industry and more precise protocols that are compatible with CAN.

Chapter 2

Theory

This chapter describes the theory of this master's thesis and contains information about what a bus system is and in particular the CAN bus which is used in this project. The chapter also presents what a communication protocol is and in particular the protocols Keyword Protocol 2000 (KWP2000), CAN Calibration Protocol (CCP) and the Universal Measurement and Calibration Protocol (XCP).

2.1 Bus system

A *bus* is a transmission path that connects different components in a network. Messages are picked up at every device attached to the bus. Each device can recognise the messages intended for it [1]. Bus systems used in the automotive industry are for instance Controller Area Network (CAN), FlexRay, Local Interconnect Network (LIN) and Media Oriented Systems Transport (MOST). The bus system of interest in this master's thesis is the Controller Area Network (CAN) that is described in the next section.

2.2 Controller Area Network (CAN)

Controller Area Network (CAN) is an asynchronous, multimaster, broadcast, serial bus protocol for real time control applications with data rates up to 1 Mbit/s in twisted-pair cabling. CAN is primarily used in embedded systems to allow fast and reliable communication between processors. CAN was developed in the 1980s by the German company Robert Bosch GmbH for in-vehicle networks. The primary purpose was to enable more functionality in automobiles. Another reason for developing CAN was to reduce wiring. Before CAN was developed, electronic devices in automotives were connected by point-to-point wiring. This caused more and more wires as more functionality were added and resulted in heavier vehicles, higher wiring costs and harder for manufacturers to troubleshoot vehicles. The wiring cost, the complexity of the system and the wiring weight was reduced when CAN was introduced. A CAN system also has better flexibility since new subsystems can be added to an existing system without modification [2, 3]. CAN is now an international standard (ISO 11898) and a de-facto standard in today's vehicles. CAN has other application fields than in the automotive industry such as in trains, aeroplanes, factory automation, escalators, medical equipment, household applications such as coffee machines and washers [2, 4].

CAN is a *peer-to-peer* network which means that the network is a non-hierarchical network and communication is not performed according to the client-server model. Instead all nodes are equally privileged, different nodes are not assigned different roles and a node can function as both a client and a server [5]. CAN is *multimaster* since many nodes can function as the master on the bus. Since CAN is a *broadcast* protocol all nodes in the CAN network will receive all messages sent on the CAN bus. Each node decides if it shall take care of the message or not depending on a message identifier that uniquely defines the content and the priority of the message [6]. In order to determine which message shall be allowed access to the bus when two or more messages want to access the buss simultaneously, bitwise arbitration is performed based on the messages identifier. This will be described in subsection 2.2.1. The different frames are described in subsection 2.2.2, the data frame format in subsection 2.2.3, bit stuffing is described in subsection 2.2.4 and error handling is described in subsection 2.2.5.

2.2.1 Bus arbitration

When a node wants to send a message it checks if the CAN-bus is busy and then writes the data to the CAN bus. If two or more nodes try to transmit messages at the same time the conflict is resolved by using *bitwise arbitration*. This process is based on the possibility of two different bus levels. A bit with value 0 on the CAN bus corresponds to a *dominant bus level* and a bit with value 1 on the CAN bus corresponds to a *recessive bus level*. The CAN bus level will be dominant if any of the nodes output a dominant level. If two or more nodes start to transmit at the same time each bit in arbitration field in the messages are considered until differences appear. The message with bit 1 will be withdrawn. The process continues until there is only one message left which wins the bus access. Since 0 is dominant over 1, messages with identifiers with smaller values will be prioritized over messages with identifiers with bigger values [5, 7].

2.2.2 Frames

CAN have two different message formats. The first is often referred to as the *standard format* and has an 11 bit identifier range. The other format is the *extended format* that uses an address range of 29 bits. The frames will thus differ in the number of bits depending on if the standard or extended format is used.

A frame is a message that is transmitted between nodes. There are four different CAN frames: data frame, remote frame, error frame and overload frame. The *data frame* is used to transmit data from a transmitting device to a receiving device. The data frame will be described in more detail in subsection 2.2.3. A *remote frame* is used by a receiver node to request the transmission of a data frame with the same identifier. An *error frame* is sent when a unit detects a bus error. The *overload frame* is used to delay further frames to let the device process received data. Data and remote frames are separated from preceding frames by a field called *interframe space* [7].

2.2.3 Data frame

A data frame, shown in Figure 2.1, consists of the following fields: start of frame (SOF), arbitration field, control field, data field, cyclic redundancy check field (CRC field), acknowledgement field (ACK field) and end of frame (EOF). The *start of frame* bit is a single dominant bit used to mark the beginning of the data frame. The bus must be idle to start the transmission and will then put the bus low. After that comes the *arbitration field* that includes the identifier. The number of bits in this field does thus depend on if the standard or the extended format is used. The arbitration field also contains a bit, the remote transmission request bit (RTR bit), to distinguish between data and remote frames. The extended format also contains 2 extra bits besides the longer identifier. The next field is the *control field* that have the function to indicate if the frame is a standard or extended frame and also express the length of the data field. Thereafter comes the *data field* that can contain 0 – 8 bytes data and thereafter the *CRC field* which is a checksum used for error checks in the CAN communication. The next field is the *ACK field* that is used to acknowledge the reception of a message and assures that at least one node has received the message correct. The last 7 bits in a data frame are the *end of frame* bits that are recessive and used to mark the end of the data frame [7, 8].

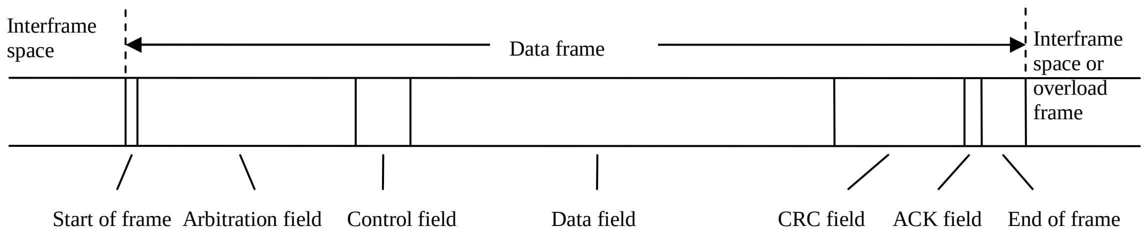

Figure 2.1 CAN data frame format.

2.2.4 Bit stuffing

CAN uses a method called *bit stuffing* in order to keep the receiving nodes synchronized. The method is applied for the fields SOF to the CRC field for data and remote frames. When more than 5 consecutive bits of the same polarity is detected by a transmitter one extra bit of inverted polarity is added after the fifth bit of equal polarity. The receivers *de-stuff* the received data and remote frames to retrieve the original content. This is done by removing the bit that follows a sequence of five bits of the same polarity within the frame sequence coded by bit stuffing [4, 9].

2.2.5 Error detection and error management

The CAN standard defines five error detection mechanisms. These contribute to CANs high level of error resistance and reliability. CAN also defines three error states. These are used by the CAN nodes and allow nodes to distinguish between permanent failures and sporadic disturbances. The error states represent the healthiness of the nodes in the network [2]. The error detection mechanisms will be described first and then the different error states.

CAN uses the error detection mechanisms bit check, stuff rule check, cyclic redundancy check, frame check and acknowledgement check. The error detection mechanism *bit check* is used by the transmitter when it is sending a message. The transmitter monitors the bus to check that the value monitored does not differ from the sent value. If a bit error is detected, an error frame is sent and the original frame is resent. The bit check rule has an exception which is when a recessive bit is sent during the arbitration field or the acknowledgement slot. The error detection mechanism *stuff rule check* is used by each node that check that there are at most 5 consecutive bits of same polarity for the SOF to CRC field for a data or remote frame. If a node detects 6 consecutive bits of same polarity, an error frame is generated and the message is resent. The *cyclic redundancy check* (CRC) is a mechanism that detects errors with a very high probability. The transmitter calculates the CRC value that is added to a message in the CRC field. The receivers calculate the CRC value in the same way as the transmitter and each node compares the calculated CRC value with the CRC value in the CRC field. If the CRC values do not match for one or more nodes, the message is resent. The detection mechanism *frame check* is used by every node to detect illegal bits in the fixed-form bit fields. The fixed-form bit fields are the recessive delimiter bits. A receiver that monitors a dominant bit for the last end of frame bit is not considered as a form error. The last detection mechanism is the *acknowledgement check* that is used by the transmitters to check if the ACK slot, part of the ACK field, contains a dominant bit. The dominant bit is sent by a receiving node as confirmation that the CRC check was successful. There is an acknowledgement error if no dominant bit is monitored during the ACK slot which means that no nodes have retrieved a valid message correctly. The original message is resent for all possible errors detected by the detection mechanisms described above [4, 7, 9].

The different error states a CAN node can have are error active, error passive and bus off. The error state is determined by two counters. There is one counter for errors occurred for transmitting frames, the *transmit error counter* (TEC), and one counter for errors occurred for receiving frames, the *receive error counter* (REC). An error detected by the transmitter will increase the TEC and an error detected by the receiver will increase the REC. The TEC is decreased if no error is detected when sending a frame and analogously the REC is decreased if no error is detected when receiving a frame. The counters usually increment faster compared to when they decrement. A node is in the error state *error active* if both counters have values less than 128. A node in error active state can normally participate in the bus communication. A node enters the state *error passive* when one of the error counters exceeds 127. The node can then still participate in the bus communication but will have to wait a certain time before further transmissions can take place when an error is detected. The third state is *bus off* which occurs when the TEC is greater than 255. The node is not allowed to participate in the bus communication in bus off state. A node in the bus off state can re-enter the error active state which is accomplished by resetting the error counters to zero, reconfiguration and then wait for 128 sequences of eleven consecutive recessive bits. The reason for waiting is to ensure that the node cannot disturb the bus communication immediately after the reset [2, 7].

2.3 Communication protocols

Communication protocols are formal descriptions of message formats and rules that allow exchange of messages in or between computing systems. A communication protocol may e.g. cover authentication, connection establishment, error detection and error correction [10, 11]. The protocols of interest in this project are the protocols Keyword Protocol 2000 (KWP2000), CAN Calibration Protocol (CCP) and the Universal Measurement and Calibration Protocol (XCP) which are described in the following subsections.

2.3.1 Keyword Protocol 2000 (KWP2000)

Keyword Protocol 2000, or *KWP2000*, is a diagnostic communication protocol that is standardized in ISO 14230. The standard is divided into different parts that define the physical layer, the data link layer and the application layer. The communication between a client (the diagnostic device) and a server (e.g. an ECU) is carried out over e.g. a CAN network. The client performs request to the slave to either request data or to let the slave perform certain operations. KWP2000 can be used for e.g. reading or clearing trouble codes, reading signals, transfer data to the server and set server parameters. A request message contains a *service identifier* and parameters related to that service. The service identifier is used in the server to identify which action to perform. The response message also contains the service identifier but is used to inform the client if the operation was successful or not. The term *positive response* is used for a successful operation and the term *negative response* is used for an unsuccessful operation or an operation not completed in time. For each sent request the corresponding response message must arrive before the next request can be sent. An example of a service is *readDataByCommonIdentifier* that is used to requests data record values from the slave by specifying the parameter *common identifier* (CID) that corresponds to a signal in the slave [12, 13].

2.3.2 CAN Calibration Protocol (CCP)

CAN Calibration Protocol (CCP) is a software interface for communication between a development tool and an ECU. The protocol is CAN based and can be used for measurement, calibration and flash programming. It is developed by the ASAP task force (Arbeitskreis zur Standardisierung von Applikationssystemen; English translation: Standardization of Application/Calibration Systems task force) that was founded by the automotive companies Audi, BMW, Volkswagen, Mercedes-Benz and Porsche. The organisation was renamed and is now called ASAM (Association for Standardization of Automation and Measuring Systems). The purpose of the task force is to standardize systems for calibration, measurement and diagnostic purposes to provide compatibility between software and hardware [14].

The CCP protocol is described in detail in the following subsections starting with the CCP dialogue in subsection 2.3.2.1 that portrays how a tool and an ECU can communicate with each other. The next subsection 2.3.2.2 describes session log-in which is a process containing certain CCP commands that shall be performed in a certain order before any other CCP commands are used. The two subsections thereafter 2.3.2.3-2.3.2.4 describe CCP commands used to perform *data acquisition* (*DAQ*) from an ECU to a tool. The first subsection of these two portrays how to initialize and start the DAQ and the second how to stop it.

2.3.2.1 CCP dialog

CCP dialog is master-slave based. The master initiates the dialog by sending information, commands and parameters, to the slave via the CAN bus. The slave is then responsible to answer the master with data or an acknowledgement according to the CCP standard [15]. The master is often a development tool and the slave is an ECU. There are only two different types of messages for CCP, each identified by their unique identifiers (addresses). Since CCP is a CAN based protocol each message is 8 bytes long. A message sent from the tool to the ECU is called a *Command Receive Object* (CRO) message that contains a command code CMD, a command counter CTR and parameters as shown in Figure 2.2.

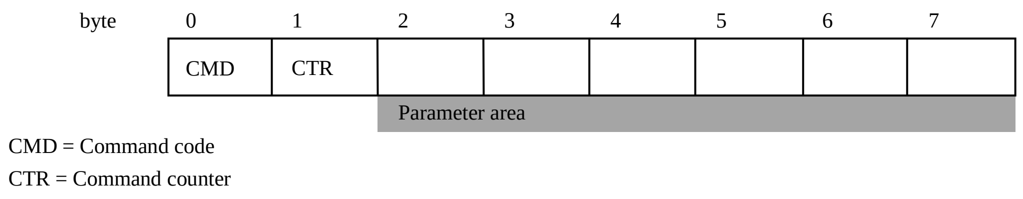

Figure 2.2 Structure for the CRO message.

The command code is used in the ECU to inform the ECU which command to perform. The command counter is used to keep track of the sent CRO from the master's point of view and the parameters are specific for each command. The command code and the commands' corresponding parameters are defined in the ASAP standard for CCP. When the ECU has performed the actions, requested by the tool, it will send a reply message to the tool. A message sent from the ECU to the tool is called a *Data Transmission Object* (DTO) message. There are 3 types of DTOs, Command Return message (CRM), Event message and Data Acquisition (DAQ) message, which will be described in the 3 next subsections. Thereafter there is a short subsection about the *CCP driver,* a piece of code in the ECU that receives commands, processes them and gives appropriate answers and after that a subsection that constitutes a CCP command example.

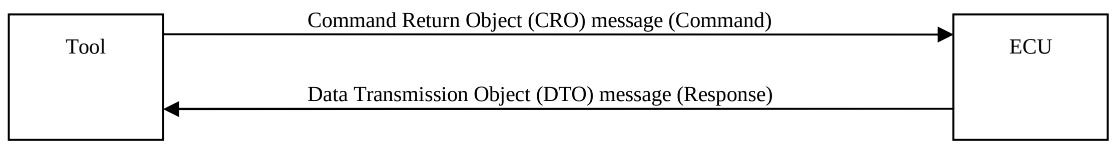
Tool
Command Return Object (CRO) message (Command)
ECU
Data Transmission Object (DTO) message (Response)

Figure 2.3 Basic CCP dialog.

A basic CCP dialogue is shown in Figure 2.3. CCP commands are generic which means that they are not node specific. This implies that ECU must be specified, which is done by specifying the *station address* for the ECU. The address is unique for the ECU and is used as an argument for the CCP command CONNECT. This is the first CCP command that can be performed. All other commands are ignored until the CONNECT command is sent to the ECU. The connection persists until a DISCONNECT command is sent from the master to the ECU or if the master connects to another ECU [16, 17].

Command Return message (CRM)

A *Command Return Message* (CRM) is an answer to a CRO message. It contains information that indicates if the command succeeded and can also contain requested data, specified in the CRO message. More specifically, a CRM message contains a packet identifier (PID) with value 0xFF. The packet identifier is the first byte in the message and has different values for different DTO messages. It is used to identify which DTO message is sent from the ECU. The second byte for the CRM message is an error code (ERR) which indicates if the performed action was successful or if it failed. The third byte is the command counter from the CRO message which is used by the master to identify which CRO message this was an answer to. The remaining 5 bytes constitute a data area that contains different information depending on the command sent from the tool. The structure of the CRM message can be seen in Figure 2.4 [16, 17].

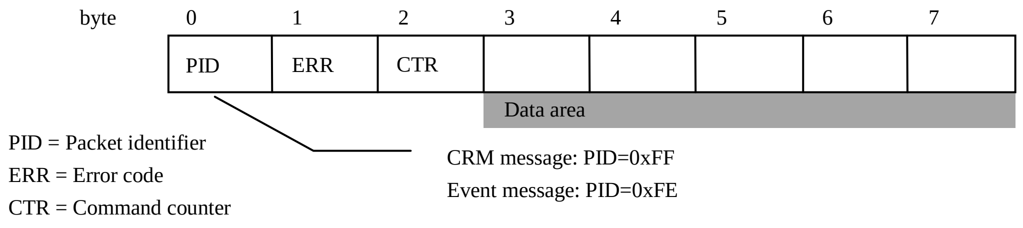

Figure 2.4 Structure for the CRM and for the event message.

Event message

The *event message* is used to notify the master of ECU internal changes. This can e.g. be used to detect errors in the ECU which can inform the master about the error and thus allow the master to take necessary actions such as error recovery. The structure is the same as for CRM message but the packet identifier is 0xFE [16, 17].

Data Acquisition (DAQ) message

The *data acquisition* (DAQ) messages are used to send measurement data from the ECU to the master. The data can be sent periodically or as a consequence of a particular event. Synchronous DAQ is shown in Figure 2.5.

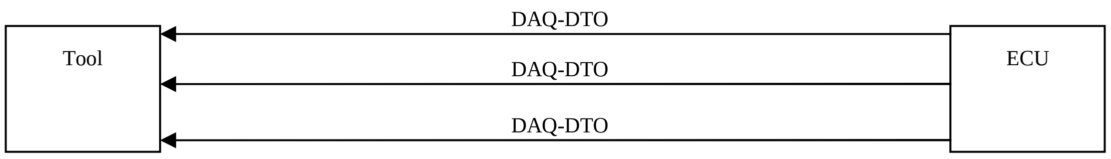

Figure 2.5 Synchronous DAQ-DTO transmission.

Before any data can be transmitted with DAQ from an ECU, the DAQ process must be initialized. In this process information is sent from the master to the ECU about which data in the ECU shall be included in the data acquisition. The initialization will be described in more detail in section 2.4.3. When the initialization is done, data can be sent synchronously to the tool from the ECU. This assumes that the ECU has support for specific DAQ commands and that the ECU has many *DAQ lists*. A DAQ list contains all measurement data for a specific time period or for a specific event. A list consists of up to 254 *Object Descriptor Tables* (ODTs), each containing 7 pointers that are used to point out memory addresses, in the ECU memory, for signals to be sent via DAQ. When information is sent to the master, it is packed in CAN messages called DAQ-DTOs. A *DAQ-DTO message* contains a Packet Identifier (PID), byte 0, and the remaining bytes can be packed with data which is shown in Figure 2.6. The organization of DAQ-lists and ODTs are shown in Figure 2.7. PIDs are used to identify the data in the DAQ-DTO messages, since a DAQ-DTO message corresponds to a certain ODT [16, 17].

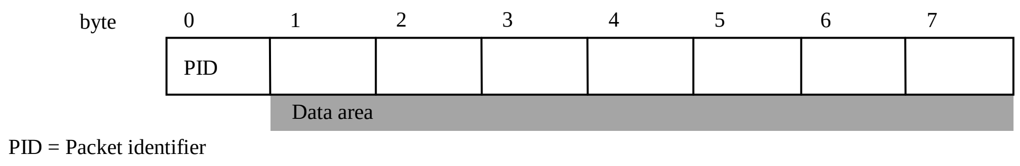

Figure 2.6 Structure for the DAQ-DTO message.

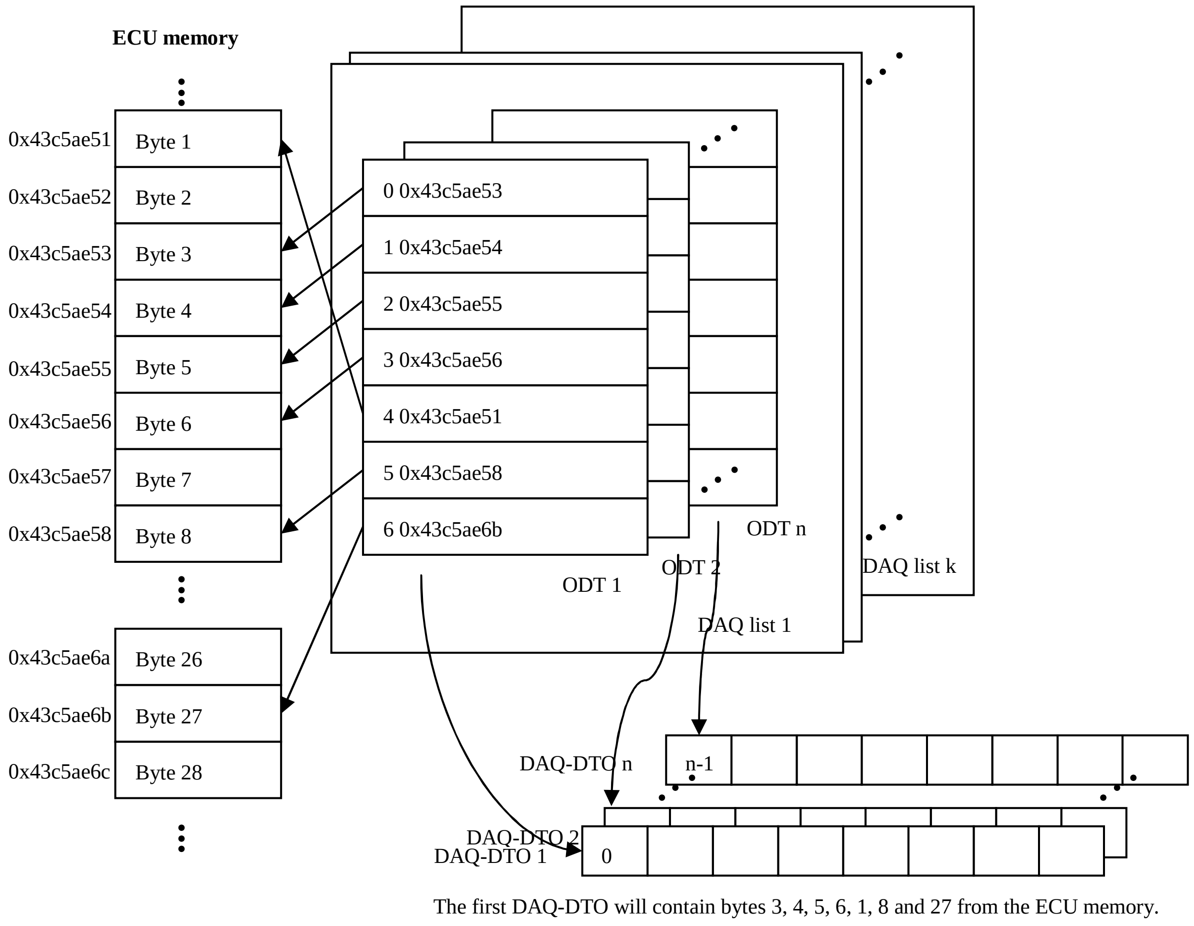

Figure 2.7 ECU memory, DAQ list- and ODT-organization.

CCP driver

A *CCP driver*, a piece of software in the ECU, is used to process the requests from the master and respond to the master with the correct answer. The CCP implementation is divided into two parts: a *command processor* that processes the CROs and answer with appropriate CRMs and a *DAQ processor* that is responsible to send measurement data at appropriate time [14, 18]. The CCP driver implementation is not described in this report.

Example of CCP command: CONNECT

Table 2.1 describes the CRO message and Table 2.2 describes the CRM message for the CONNECT command which establishes a logical point-to-point connection with a specified slave station. The tables specify what parameters are expected for the command and how to interpret the answer from the ECU.

Table 2.1 Structure of CRO message for the CONNECT command.

Position	**Type**	**Description**
0	byte	CMD = 0x01
1	byte	CTR
2-3	word	Station address (Intel format; Little-Endian)
4-7	bytes	Don't care

Table 2.2 Structure of CRM message for the CONNECT command.

Position	**Type**	**Description**
0	byte	PID = 0xFF
1	byte	Command Return Code (Error code, ERR)
2	byte	CTR
3-7	bytes	Don't care

Figure 2.8 shows an example usage of the CRO for the CONNECT command. Byte 0 with value 0x01 indicates that this is a CONNECT command. The command counter is 0x46 and the station address is 0x2236 and due to the station address shall be on Intel format 0x36 is the value of byte 2 and 0x22 is the value of byte 3. Byte 4-7 can be ignored.

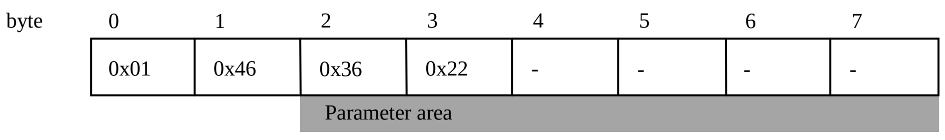

Figure 2.8 Example of CRO message for the CONNECT command.

Figure 2.9 shows the CRM message for the CRO message above. Byte 0 indicates that it is a CRM message (0xFF), byte 1 (0x00) that it is an acknowledgement (command successfully performed), byte 2, the command counter, that this is a answer to the CRO message with CTR=0x46 [17].

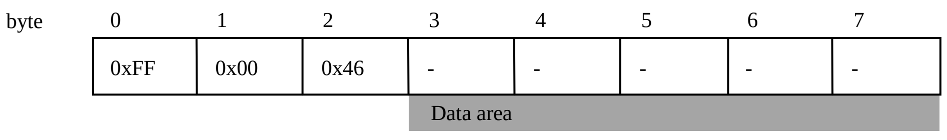

Figure 2.9 Example of CRM message for the CONNECT command.

2.3.2.2 Session log-in (Cold start)

Session log-in or cold start is done to create a new logical point-to-point connection between the master and the slave and also to perform initialization commands. The procedure is shown in Figure 2.10. Each command represents both a CRO message and a CRM message. In more detail, session log-in starts with the connect command CONNECT. The parameters for the CONNECT CRO message are the CMD, the CTR and the station address. The CMD and CTR are used for all CRO commands. The station address specifies which ECU to establish a connection with. Thereafter the GET_CCP_VERSION command is used where the desired CCP version is specified. The slave will answer with the implemented version and if this matches the desired CCP version the communication will proceed. The next command is EXCHANGE_ID which performs automatic session configuration. This command returns a resource protection mask that indicates which resources are protected and which are not. A resource with protection status locked must be unlocked in order to use it. This is done by first getting a key for the resource by passing the information of which resource should be used to GET_SEED. The answer from the ECU contains a ‘key’ or ‘seed data’ that is used to unlock the protection by using the UNLOCK command. An example of a resource is DAQ. The GET_SEED and the UNLOCK command shall be performed for each protected resource that is going to be used. The next command is SET_S_STATUS that is used to set the session status. It is recommended to include this command in the login session to initialize the status bits [15, 17].

```
Session log-in(R)
1  CONNECT                Establish a logical point-to-point connection with a slave station
2  GET_CCP_VERSION        Check CCP version used in slave device
3  EXCHANGE_ID            Automatic session configuration
4  for requested resource $r$ in R
5      if resource $r$ is locked
6          $k_r$ = GET_SEED(r)  Get key $k_r$
7          UNLOCK ($r$, $k_r$)    Unlock resource $r$ with key $k_r$
8  SET_S_STATUS           Initialize the status bits
```

Figure 2.10 Pseudo code showing the CCP commands used for session log-in. R contains information of which resources that the user wants to use.

2.3.2.3 DAQ list initialization

The DAQ list initialisation process is done to register the variables in the ECU that shall be sent using data acquisition and also to start this process. This process, shown in Figure 2.11, assumes that the session log-in has been performed or at least the connect command. The figure does not show the answers from the ECU.

DAQ list initialisation starts with setting the DAQ status bit to 0 with the command SET_S_STATUS. Thereafter the GET_DAQ_SIZE command is used with the CRO message parameters *DAQ list number* and the *CAN identifier for the specified DAQ list number*. The command is used to clear the specified DAQ list, initialize it and stop DAQ for this list. The command returns the number of available ODTs and the first PID for that DAQ list. The command SET_DAQ_PTR is then used to initialize a DAQ list pointer by specifying the DAQ list, ODT number and an element number in the ODT. This information will later be used when interpreting the DAQ-DTOs sent to the master. The ODT number corresponds to the PID for a received message, the element number within an ODT corresponds to the element in a DAQ-DTO. After a DAQ list, ODT number and an element have been specified the command WRITE_DAQ is used to specify the address and address extension in ECU memory for the signal (variable) to be used in DAQ. The size of the signal is also specified. For each signal to be used for DAQ the SET_DAQ_PTR must first be performed and then the WRITE_DAQ command to set the ODT pointers and to define the format for the contents in the DAQ-DTOs. The session status bit for DAQ is then set to 1 with SET_S_STATUS and finally the data acquisition is started with the START_STOP command. START_STOP is used both for starting, stopping and to prepare a synchronous start of a specified DAQ list. A parameter *mode* is used to specify which of these actions to perform; 0x00 to stop, 0x01 to start and 0x02 to prepare for a synchronous start. A synchronous start is used to start all configured DAQ lists. An ECU might not support a synchronous start which might lead to start of data acquisition. START_STOP also takes a parameter *DAQ list number* that is used to specify which DAQ list should be used if there exist more than one DAQ list. A parameter *last ODT number* is used to know which ODTs shall be present in DAQ (from 0 to last ODT number). A parameter *event channel number* is used to specify the *sample period*, also called *cycle time*, that specifies how frequently the DAQ list shall be transmitted. A parameter *transmission rate prescaler* is used to reduce the desired transmission rate, where 1 is used for no reduction and higher values are used for higher reduction [17].

```
DAQListInitialization(S)
1  SET_S_STATUS          Set session status with DAQ bit = 0
2  GET_DAQ_SIZE          Stop, clear and initialize DAQ list. Get first PID and number of
                         ODTs
3  for signal in S
4      SET_DAQ_PTR       Set pointer
5      WRITE_DAQ         Set memory address for signal and signal size
6  SET_S_STATUS          Set session status with DAQ bit = 1
7  START_STOP            Start data acquisition
```

Figure 2.11 Pseudo code showing the CCP commands used for DAQ list initialisation that first register the variables used for DAQ and then start the DAQ. S contains information of which signals (variables) that the user wants to get transmitted.

2.3.2.4 Stop the data acquisition

Data acquisition is stopped by using the START_STOP command again but this time with other value for the parameter *mode*. If one wants to end the CCP session one shall also use the DISCONNECT command that is used to disconnect the slave device [17].

2.3.2.5 Upload data

There are two ways to retrieve data from an ECU with CCP, other than DAQ. The first is by using the CCP commands SET_MTA and UPLOAD and the second method is by using SHORT_UP. The first method uses the SET_MTA command to set the memory transfer address for a *base pointer* for all following memory transfers. UPLOAD is then used to read the data on the address that was specified with SET_MTA. The base pointer will for each UPLOAD command be post-incremented by the size of the data to be uploaded. The second method is to use the SHORT_UP command, that is used to retrieve data from the ECU by specifying the address of the data of interest and its size [17].

2.3.3 Universal Measurement and Calibration Protocol (XCP)

The *Universal Measurement and Calibration Protocol* (XCP) is based on CCP version 2.1 but does not limit users to use CAN as transport media. XCP support many transport media and the X in XCP is used to express this fact, that stands for the “various” transport layers that can be used by the users. Examples of transport media are CAN, FlexRay, Ethernet (TCP/IP and UDP/IP) etc. The main purpose of XCP is to acquire the current values of internal variables and adjust internal variables of an ECU [19, 20]. XCP is a software interface to interconnect a development tool with an ECU or a measuring instrument.

The XCP dialog is, like CCP, master/slave based but a XCP message allows 8 to 255 data bytes. A *Receive message* is sent from the master to the slave and a *Response message* is sent in the opposite direction. The terms *Command Receive Object* (CRO) and *Data Transmission Object* (DTO) are also used for XCP but the structure of these objects differ from the way they are defined for CCP [21].

2.3.3.1 The Command Receive Object (CRO) for XCP

The CRO is used for executing commands according to the *Command Message* (CMD). The CRO is also used for transmitting *synchronous data stimulation* (STIM) to the slave. The CRO messages are categorized according to their function into the groups *standard commands* (STD), *calibration commands* (CAL), *data acquisition and stimulation commands* (DAQ), *page switching commands* (PAQ) and non-volatile memory *programming commands* (PGM) [21].

2.3.3.2 The Data Transmission Object (DTO) for XCP

The DTO is used to send data back to the master. Like the CRO messages, the DTO messages are also categorized according to their function into groups. They are *command response message* (CRM), *data acquisition message* (DAQ), *error or event message* (ERR_EV) and the *service request message* (SRM) [21].

2.3.3.3 Synchronous Data Transmission

Like CCP, XCP also defines a method to transfer data synchronous from the slave to the master. One difference compared to CCP is that data acquisition (DAQ) with XCP allows dynamically configurations of DAQ lists. DAQ with XCP also allows the DAQ to start on power-up and DAQ messages can be sent without PID. Timestamps have also been defined. DAQ lists can also be prioritised over other DAQ lists. The features described here are benefits of using DAQ via XCP instead of DAQ via CCP. There are also other benefits of using XCP over CCP but these are not described in this report [21].

Chapter 3

Method

This chapter will describe the different choices made in this project, the experimental setup and the implementation.

3.1 Choice of protocol

CAN Calibration Protocol (CCP) was chosen to retrieve data from the ECUs. One reason for this is that CCP is especially intended for recording measurement data from control units and for calibration [22]. CCP is also standardized and has been used in the automotive industry for many years. There is a newer version of CCP standardized as the Universal Measurement and Calibration Protocol (XCP). The reason for choosing CCP instead of XCP is that Scania's ECUs already have support for CCP in the ECUs, but not for XCP. If XCP had been chosen, code had to be implemented for both the ECU- and the tool-side. XCP was thus excluded because of the limited time of a master's thesis.

More specifically data acquisition (DAQ) is the method chosen to retrieve data from an ECU. The method using SHORT_UP commands will be an additional method to be evaluated but the focus will be on DAQ. The reason for choosing SHORT_UP over the commands SET_MTA followed by UPLOAD is that the first method will be faster. This is due to the second method requires two commands and the first requires only one command. In the general case the addresses for the signals to be logged are assumed not to be in consecutive order. An advantage with DAQ, compared to a request method, is that only one message is needed for each data transfer while 2 messages are needed for the request method. The request method will therefore contribute to a higher busload. If assuming that both methods use the same one-way propagation time, DAQ will be a faster method for transmitting variables. The request method also needs some time to process the request while the DAQ method is preconfigured via the DAQ initialisation. DAQ also have the advantage that a DAQ message can contain more than 1 signal while the message for a request method only can contain 1 signal. Another advantage using DAQ is that all measurements are registered at the same time in the ECU, i.e. originate from the same ECU computing cycle, while this cannot be ensured for the request method.

3.2 Experimental setup

The experimental setup contains six parts; an ECU, a computer, a power supply, a Vehicle Communication Interface (VCI), the CAN network and cables and resistor used for termination. The VCI is used to allow communication between the ECU and the computer. The VCI is connected to the computer via a USB cord and to the ECU via CAN wires. The CAN wires are terminated via cables and the resistor. The ECU and the VCI are both connected to the power supply. The ECU has 1 additional cable to provide power for ignition. The experimental setup is shown in Figure 3.1.

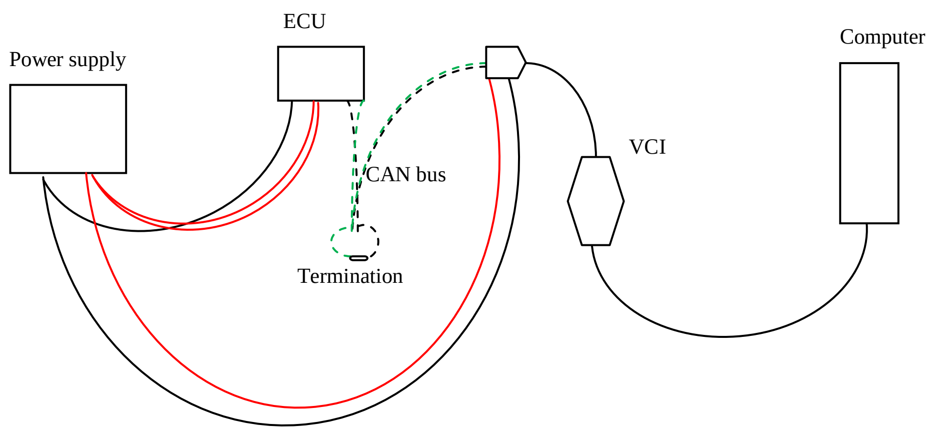

Figure 3.1 Experimental setup.

3.3 Choice of programming language

A CAN library was needed to communicate with the ECU via the VCI since CCP was chosen as protocol. The VCI contains hardware from the Swedish company Kvaser. They provide *CANLIB* which is a library to communicate over CAN. CANLIB have support for the programming languages C/C++, C#, Python, Visual Basic and Delphi [23]. An advantage with Python is that it is faster to write code in, since Python almost always requires less code than programming languages like C. Python code is interpreted at runtime and needs no compilation [24]. This means that errors are found at runtime which is a disadvantage. Since Python code is interpreted and not compiled, a program written in Python will in most cases be slower than programs written in other languages as C/C++ that compile the code before the code can be run. For these reasons Python was excluded. The CANLIB documentation code was written in C# and Delphi, but most examples were written in C. Therefore C was chosen as programming language. Later the language was changed to C++ since it contains functionality that would facilitate the code writing. Most of that functionality was provided by the C++ Standard Library, such as strings, vectors and functionality for input from file.

3.4 Implementation

This section describes the implementation of DAQ. The subsections will then describe some parts in closer detail.

As described in the previous section, Kvaser CANLIB was used to communicate over CAN with the ECU via the VCI. There are some CANLIB functions that must be called and some functions that are optional to communicate over CAN. The CANLIB functions used in the code implementation are described in Appendix A. After the necessary CAN commands have been called information about the internal variables in the ECU, necessary addresses, identifiers and parameters must be loaded from a file. There are two different file formats this information can be retrieved from: the ati-format and the a2l-format. The first file format is developed by the company ATI and the second is a standard developed by ASAM. The file format ati was used and the information loaded from these files is described in subsection 3.4.1. The signals that shall be logged are represented by a list (C++ vector) with text strings, e.g.

RTDB_VEHICLE_SPEED_E_val_S32 which is the signal name for the vehicle speed. A signal struct object is then created for each signal to be logged and added to another list. These struct objects contain information used to perform DAQ and their properties are described in subsection 3.4.1. The signals in the signal struct list (C++ vector) are sorted according to the algorithm that is described in subsection 3.4.2. The sorting function also creates a list of ODT (Object Descriptor Table) struct objects that also will be described in subsection 3.4.2. The next thing performed is the CCP session log-in described in Chapter 2. For each received CRM message error checking is done which is described in subsection 3.4.3. Thereafter a function is called that checks if DAQ is active and if that is the case the DAQ is stopped. DAQ is now started for the signals in the signal struct list by calling the DAQ list initialization method, which also is described in the theory section. Data from the ECU is transmitted to the tool in DAQ-DTO messages that are read by the tool using functionality for reading CAN data described in Appendix A. The received data needs to be interpreted according to information stored for each signal in the signal struct object list. The interpreted values are written to a file with the corresponding time when the values were registered in the ECU. The interpretation of the signals transmitted and the log file structure are described in subsection 3.4.4. During data acquisition, checks are performed to inform the user of high busloads (higher than 80 %), errors and overloads. Busload is a measure of how much the CAN bus is utilized, measured in %. DAQ is ended by pressing any key during the DAQ session. The CCP command for this is also described in Chapter 2. The organisation of the code into different files can be found in Appendix B.

3.4.1 The ati file and the signal objects

The ati file contains both information about each internal variable in the ECU but also ECU specific variables and addresses. Each ECU type has an ati file with its own signals, identifiers, addresses etc. A signal is described by comma separated values and/or text on one row in the ati file. Example on such information is: a number representing the data type of the signal, the minimum and maximum value for the signal, the corresponding ECU address, the signal name, the unit the signal is measured in, a formula for how the values from the ECU shall be interpreted, number of decimal places, a description of the signal etc. The information stored in my signal struct objects are: the signal name, the unit, the ECU address, an factor, number of decimal places, the signal size and a boolean that is true if negative values for the signal is allowed and otherwise false. The formulas for all signals for the ECUs used in this thesis have always been on the form ‘x’ or ‘x/some number’ and thus only a factor is needed. In the general case the formula can be a polynomial of second order divided by another polynomial of second order. A general formula parser can be implemented if it turns out necessary.

There are also some CCP specific addresses needed to perform DAQ: the CRO identifier, the DTO identifier and the station address and some DAQ related parameters. The DAQ related parameters are: the event channel number, the transmission rate prescaler and the CAN identifier of DTO dedicated to DAQ list number. The event channel number and the transmission rate prescaler are both used as parameters for the CCP command START_STOP and the CAN identifier of DTO dedicated to DAQ list number is used for the CCP command GET_DAQ_SIZE. These parameters are described in Chapter 2.

3.4.2 The sorting algorithm and the signal organisation in ODTs

The order of the signals that shall be logged have an effect on how many signals that can transmit via DAQ if using signals bigger than 1 byte. A CAN message has the size 8 bytes and for a DAQ-DTO message 1 byte is reserved for the packet id (PID) which means that there are 7 bytes for the payload. For DAQ with CCP only one-, two- and four-byte signals are allowed. The goal is to use as few ODTs as possible, in order to avoid using excessive resources. A more general case is the well known problem in computer science called the *bin packing problem*. The goal is to pack *n* objects $\{s_1, s_2, ..., s_n\}$ that satisfy $0 < s_i < 1$ into bins of size 1 so that the number of bins are minimal [25]. One idea is to first place the objects greater than the half the capacity of a bin into separate bins due to that there is no choice for these objects. This means that the objects placed in superfluous bins are a result of the packing of objects less than or equal to the half bin capacity. In the special case the four-byte signals are the only signals that require an ODT each and thus the four-byte signals are first packed into separate bins. Signals left have either size 1 or 2 bytes. Thereafter the two-byte signals are packed. The one-byte signals are then used when no other signals can be used or no other signals with other signal sizes than 1 are left. This is also the way to pack if more ODTs are needed for one-byte and/or two-byte signals. The one-byte signals will always fit into an ODT unless the ODT is full and this is the reason why the one-byte signals are packed after signals of other sizes. An optimal packing is, according to the reasoning above, achieved by first adding as many four-byte signals as possible (at most 1), then add as many two-byte signals as possible and then add as many one-byte signals as possible . When an ODT is full the procedure is started for a new ODT.

Four-byte signals and two-byte signals can be split into parts so that the parts are in different ODTs. More of the total ODT space may be utilized in this way. The sorting algorithm implemented does not split signals. The reason for not splitting signals is that the master will only receive part of the value if a message with a split signal is lost.

The function that sorts the signals also creates a list containing information about how the signals are organized in the ODTs for each signal and also calculates the last ODT number. Each description of a signal in an ODT is described by a `SignalInODT` struct object that contains ODT number (`ODTNumber`), an element number in the ODT (0, 1, ...) (`elementNumberInODT`) and a index of the first byte for the signal within the ODT (`indexInODT`). The last ODT number is later used in the CCP command START_STOP to know which ODTs are present in DAQ. The ODT number and the element number in the ODT are both used (later) in the CCP command SET_DAQ_PTR. The index of the first byte in the ODT for each signal is used when data shall be interpreted from the DAQ-DTO messages.

The pseudo code for the sorting and creating the `SignalInODT` list is shown in Figure 3.2. The function indexOfFirstSignalSpecifiedSize() takes the signal struct objects as the first parameter, an index as the second parameter and a desired size as last parameter. The function returns the index of the first signal in the signal struct list with size specified by the third parameter with the search started at the index specified by the second parameter. The function swap() is used to exchange the values of the two parameters.

```
OrderSignals_CreateSignalInODTList(&S, &SignalInODTlist, &LastODTNumber)
1  num4byte = num2byte = num1byte = usedSpace = elementNumberWithinODT = 0
2  LastODTNumber = 0
3  ODTcapacity  = 7
4  for signal s in S
5      if s.size == 4
```

```
        num4byte += 1
    if s.size == 2
        num2byte += 1
    if s.size == 1
        num1byte += 1

for i = 0 to S.size() - 1
    if num4byte > 0 and ODTcapacity – usedSpace ≥ 4
        j = indexOfFirstSignalSpecifiedSize(S, i, 4)
        swap(S[i], S[j])
        Create SignalInODT struct object signalinodt
        signalinodt.ODTNumber = LastODTNumber
        signalinodt.elementNumberInODT = elementNumberWithinODT
        signalinodt.indexInODT = usedSpace
        SignalInODTlist.push_back(signalinodt)
        elementNumberWithinODT += 1
        usedSpace += 4
        num4byte -= 1
    else if num2byte > 0 and ODTcapacity – usedSpace ≥ 2
        j = indexOfFirstSignalSpecifiedSize(S, i, 2)
        swap(S[i], S[j])
        Create SignalInODT struct object signalinodt
        signalinodt.ODTNumber = LastODTNumber
        signalinodt.elementNumberInODT = elementNumberWithinODT
        signalinodt.indexInODT = usedSpace
        SignalInODTlist.push_back(signalinodt)
        elementNumberWithinODT += 1
        usedSpace += 2
        num2byte -= 1
    else if num1byte > 0 and ODTcapacity – usedSpace ≥ 1
        j = indexOfFirstSignalSpecifiedSize(S, i, 1)
        swap(S[i], S[j])
        Create SignalInODT struct object signalinodt
        signalinodt.ODTNumber = LastODTNumber
        signalinodt.elementNumberInODT = elementNumberWithinODT
        signalinodt.indexInODT = usedSpace
        SignalInODTlist.push_back(signalinodt)
        elementNumberWithinODT += 1
        usedSpace += 1
        num1byte -= 1
    else
        usedSpace = elementNumberWithinODT = 0
        LastODTNumber += 1
        i -= 1
```

Figure 3.2 Pseudo code showing how the signals are ordered before DAQ is started and how the list, containing information about how signals are organized in the ODTs, is created. S is passed by reference and contains information of which signals (variables) that the user wants to get transmitted via DAQ. This list will be sorted when the function is run. SignalInODTlist is a empty list that is passed by reference and created in the function. LastODTNumber is an integer representing the last ODT that shall be present in DAQ and is also calculated in the pseudo code. The character & is used to denote a pass by reference, += is used to increment a variable by the number to the right of the operator and -= is used in the same way but for decrementing the variable.

The algorithm shown in Figure 3.2 starts by initializing some of the variables on lines 1-3. On lines 4-10 the number of one-, two- and four-byte signals are counted and stored in the variables num1byte, num2byte and num4byte. The actual work is performed on the lines 11-48. On line 11 there is a loop, over the variable *i*, that loops as many times as elements in the signal struct object list S. Line 12 checks if there are any four-byte signals the user wants to get via DAQ and if there is space left in the current ODT for a four-byte signal. If this is the case the first four-byte signal is found in S and is exchanged with the value on index *i*. A `SignalInODT` struct object is created and its members are assigned on lines 15-18. The object is added to the `SignalInODT` list on line 19. The element number within the ODT (a counter) is added by 1, the used space variable is added by 4 and the variable num4byte is decreased by 1. If there were no four-byte signals or there was not enough space left in the current ODT when line 12 was evaluated, the condition on line 23 is evaluated. The condition is similar as on line 12 but now for 2 bytes. The code in lines 24-33 is also similar as the code in lines 13-22 but now for 2 bytes. If there are no two-byte signals or there is no space left in the current ODT similar code is run for one-byte on lines 34-44. If there are no one-byte signals left or there are no space left in the current ODT (the ODT is full) the code on lines 46-48 is run. It selects a new ODT by adding 1 to the variable LastODTNumber and sets the used space variable to 0 for the new ODT and also sets the element number within the ODT to 0. The loop index *i* is decremented to adjust for the increment of *i* when no signal was sorted.

The algorithm assumes that there only exists one-, two- and four-byte signals. If the CCP document is updated in the future and then allows other signal sizes, this algorithm will not work.

3.4.3 Error checking for CCP commands

In Appendix A error checks are described for the status code when performing a CANLIB CAN command. If the command was a CRO command, the DTO shall also be checked: the first byte shall have value 0xFF (DTO), the second byte shall have value 0x00 (acknowledgement) and the third parameter shall have the value of the command counter used for the CRO message. If this is not the case an error message is printed to the user and the program will exit.

3.4.4 The interpretation of the signals transmitted and the log file format

When a DAQ-DTO message is read, from the receive buffer, the packet id (PID) is used to know which ODT the message corresponds to. The first byte in the DAQ-DTO message is the PID and the rest of the bytes containing data are described by the sorted signal struct object list S and the signals in ODT list SignalInODTlist that are both described in subsection 3.4.2. S contains the size of the signals which is needed to interpret the data in the DAQ-DTOs and SignalInODTlist contains the ODT number and the index of the first byte within the ODT that also is used to interpret the data in the DAQ-DTOs. The first byte in the ODT corresponds to the second byte in the DAQ-DTO since the first byte in the DAQ-DTO is reserved for the PID. Therefore, the index in the ODT that contain the first byte for a signal have to be added by 1 to get the start byte in the DAQ-DTO, which is done after the correct ODT number is found in SignalInODTlist. When this is found all the values for the DAQ-DTO can be interpreted. This is done by knowing the first byte of the signal in the ODT, the size of the signal, the conversion formula and the value representing if the signal is signed or unsigned. If it turns out that a DAQ-DTO is missing, the program will then break. This happens if the PID is not one more than the PID of the previously sent DAQ-DTO (or 0 in the case of the previous PID was the last PID).

The values transferred from the ECU to the program are logged according to the format shown in Figure 3.3 where *n* signals are used. The format first starts with a heading containing Time (s), followed by the signal names for each signal used in the DAQ. After each signal name the signal unit is present in parenthesis and thereafter a comma. The lines thereafter contain the time when the *n* signals were registered in the ECU from the start of DAQ followed by the values for the signals in the heading. The time and the signal values are comma separated. The time values are the result of a counter that increments by the time specified by the event channel number in the ati file. The alternative would be to measure when the DAQ-DTO messages arrive to the tool but this is not the same as the time when the signals were registered in the ECU. This is why the counter is used; it increments by the DAQ sample period, i.e. when the signals are registered in the ECU. An example of a beginning of a logging file, that contains 2 signals, is shown in Figure 3.4.

```
Time (s),Signal name 1 (unit 1),Signal name 2 (unit 2),...,Signal name n (unit n),
Time 1,Signal value 1 for signal 1,Signal value 1 for signal 2, ...,Signal value 1 for signal n,
Time 2,Signal value 2 for signal 1,Signal value 2 for signal 2, ...,Signal value 2 for signal n,
Time 3,Signal value 3 for signal 1,Signal value 3 for signal 2, ...,Signal value 3 for signal n,
Time 4,Signal value 4 for signal 1,Signal value 4 for signal 2, ...,Signal value 4 for signal n,
...
```

Figure 3.3 The log file format for n signals.

```
Time (s),RTDB_VEHICLE_SPEED_E_val_S32 (km/h),RTDB_CUR_CPU_LOAD_E_val_S32 (%),
0.000000,83.647,23.2,
0.010000,83.647,23.2,
0.020000,83.647,23.0,
```

Figure 3.4 The beginning of a log file for 2 signals with sample period 10 ms.

Chapter 4

Results

This chapter contains both results for only DAQ with CCP, comparison between DAQ with CCP and the KWP service readDataByCommonIdentifier and also a brief section for results of CCP SHORT_UP. The parameters of interest used in the result are time, busload, the CPU load for 100 Hz and the total CPU load. The *CPU load* or *CPU usage* is a measure of how much the ECU is utilized and is measured as a ratio of the CPU time to the CPU capacity. The *CPU time* is the amount of time the CPU is executing instructions. The reason for measuring the CPU load for 100 Hz is that all measurements for DAQ with CCP are performed with 10 ms sample period. The reason why the DAQ sample period was chosen to of 10 ms is that it is the only sample period for DAQ with CCP supported for the ECUs used at Scania and this is also the wanted logging frequency. All measurements are done on bench with the experimental setup described in Chapter 3 and with the same ECU for all tests. The busload on bench when no CCP or KWP measurements were performed was 0.03 %, i.e. nearly no busload at all. The reason for not performing the tests in a truck was to avoid interference from possible events. The bit rate is 500 kbit/s for all measurements.

Section 4.1 contains graphs for only DAQ with CCP, graphs comparing DAQ with CCP and the KWP service readDataByCommonIdentifier that is briefly described in the theory section. Section 4.2 contains measurements for how long time it takes to perform a KWP request. In section 4.3 the CPU load for 100 Hz and the total CPU load are presented for both the normal load when not affecting the ECU and for CCP with 30 ODTs. Section 4.4 presents measurements for the CCP SHORT_UP command. In section 4.5 theoretical busload calculations for DAQ with CCP are presented. The last section, section 4.6 contains measured busload for one DAQ-DTO.

4.1 CCP and KWP measurements

All DAQ measurements were carried out with sample period 10 ms during 12 seconds. The duration of 12 seconds was chosen to allow enough of busload measurements. The measurements for only CCP were done for 4 signals/DAQ-DTO or at most 4 signals/DAQ-DTO. The first signal in a DAQ-DTO when using 4 signals/DAQ-DTO or at most 4 signals/DAQ-DTO will always be a four-byte signal. The rest of the signals for 4 signals/DAQ-DTO are three one-byte signals and the rest of the signals with at most 4 signals/DAQ-DTO are between zero to three one-byte signals. The measurements for only CCP are presented in the subsections 4.1.1 and 4.1.2. The difference is that the measurements performed in subsection 4.1.1 are performed when adding 1 ODT per measurement while the measurements in subsection 4.1.2 are performed when adding 1 signal per measurement. When comparing CCP with KWP 1, 4 and 7 signals/DAQ-DTO are used for CCP. Only four-byte signals are used for 1 signal/DAQ-DTO. For 4 signal/DAQ-DTO one 4 byte signal and three one-byte signals are used. Only one-byte signals are used for 7 signals/DAQ-DTO. When using 4 signal/DAQ-DTO 4 signals are added for each measurement and not 1. Similarly 7 signals are added for each measurement when using 7 signals/DAQ-DTO. For each CCP measurement made, a corresponding KWP measurement was performed. When using CCP the number of values transmitted to the master was counted and the same number of requests was performed when using KWP. The measurements comparing CCP with KWP are presented in subsection 4.1.3.

The interesting variables to measure for CCP are the busload, the CPU load for 100 Hz (since the DAQ sample period is 10 ms) and the total CPU load. The interesting variables for CCP are also interesting for KWP but also the time between the first request to the last response. This is not needed for CCP since the DAQ time is predefined. Busload was measured, every second for both CCP and KWP, by using functionality in Kvaser's CANLIB described in Appendix A. The CPU load for 100 Hz and the total CPU load are two four-byte signals that can be retrieved from the ECU via CCP or KWP. This is possible only for measurements containing four-byte signals. If using 7 signals/DAQ-DTO, where only one-byte signals are present, CPU load for 100 Hz and the total CPU load cannot be measured. Since two four-byte signals are needed, to measure the CPU load for 100 Hz and the total CPU load, at least 2 ODTs are needed. This means that there will be at least 8 signals for 4 signals/DAQ-DTO, at least 2 signals for 1 signal/DAQ-DTO and at least 5 signals for at most 4 signals/DAQ-DTO. The CCP command GET_DAQ_SIZE can be used to retrieve the number of ODTs in a DAQ list. The ECUs used had 30 ODTs which means that if using 4 signals/DAQ-DTO there can be 30 four-byte signals and 90 one-byte signals and 120 signals in total. If only using one-byte signals 210 signals can be transmitted from the ECU. Both one- and four-byte signals are normally used when logging variables and thus 4 bytes/DAQ-DTO are used in the tests for only CCP. Two-byte signals are not used since they are unusual and do not exist at all in many ECUs. For both the busload, the CPU load for 100 Hz and the total CPU load many values are retrieved for each measurement; each point in the diagrams. The graphs are based on the mean values for these variables.

4.1.1 CCP measurements when adding 1 ODT per measurement

Figures 4.1 a-c show the busload, the CPU load for frequency 100 Hz and the total CPU load as a function of number of signals for CCP. 4 signals/DAQ-DTO (signal sizes 4, 1, 1 and 1 byte) are used and 4 signals are added for each measurement.

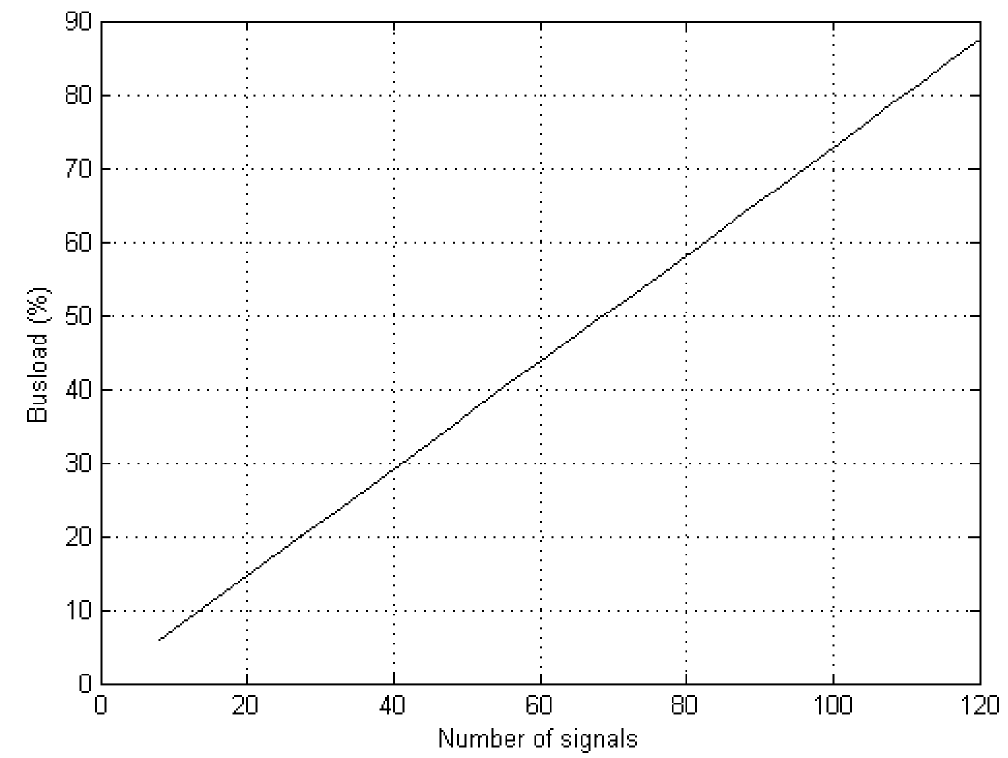

Figure 4.1 a) Busload as a function of number of signals for CCP. 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 4 signals are added for each measurement.

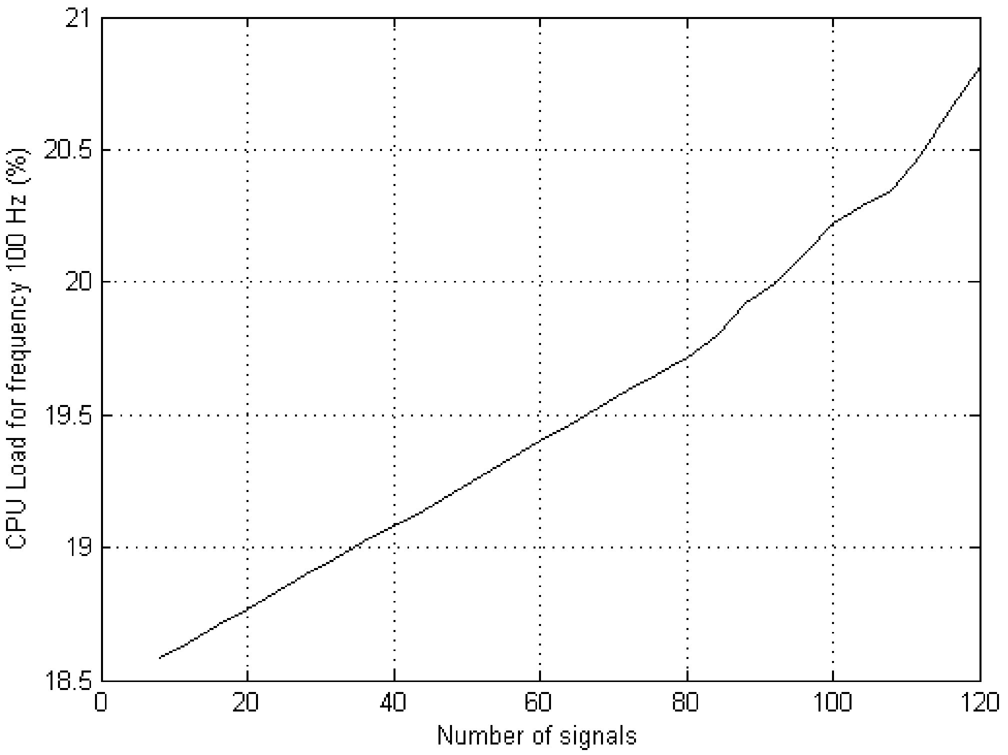

Figure 4.1 b) CPU load for frequency 100 Hz as a function of number of signals for CCP. 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 4 signals are added for each measurement.

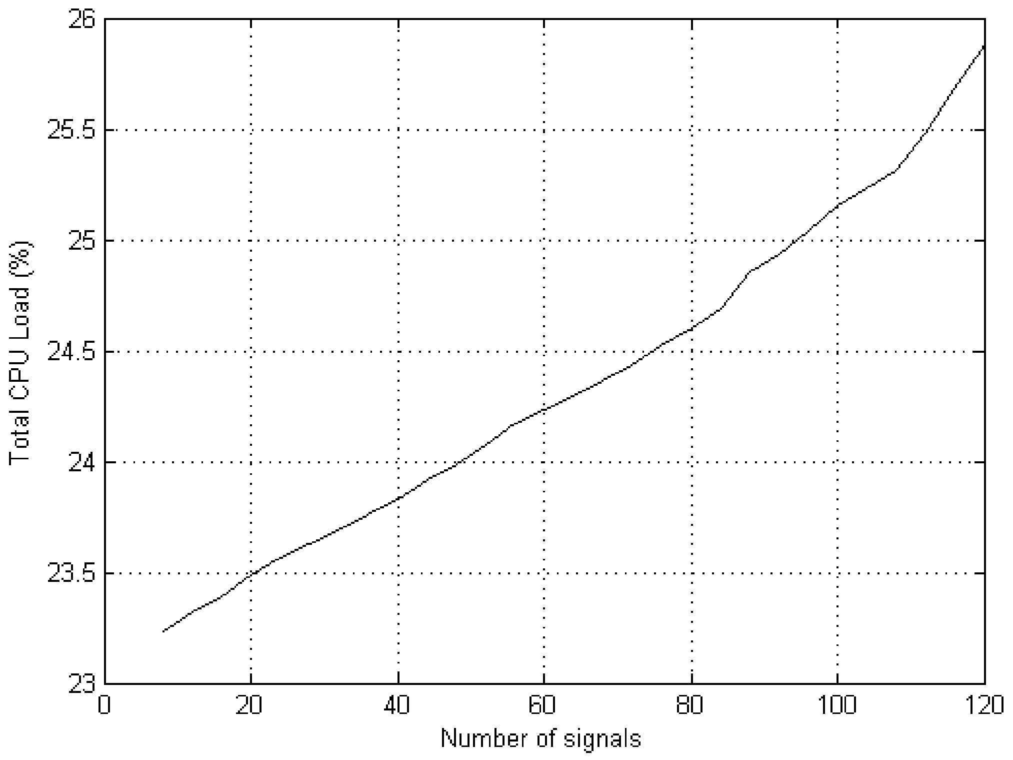

Figure 4.1 c) Total CPU load as a function of number of signals for CCP. 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 4 signals are added for each measurement.

4.1.2 CCP measurements when adding 1 signal per measurement

Figures 4.2 a-d show the busload, an enlarged part of the busload, the CPU load for frequency 100 Hz and the total CPU load as a function of number of signals for CCP. 4 signals/DAQ-DTO (signal sizes 4, 1, 1 and 1 byte) are used and 1 signal is added for each measurement.

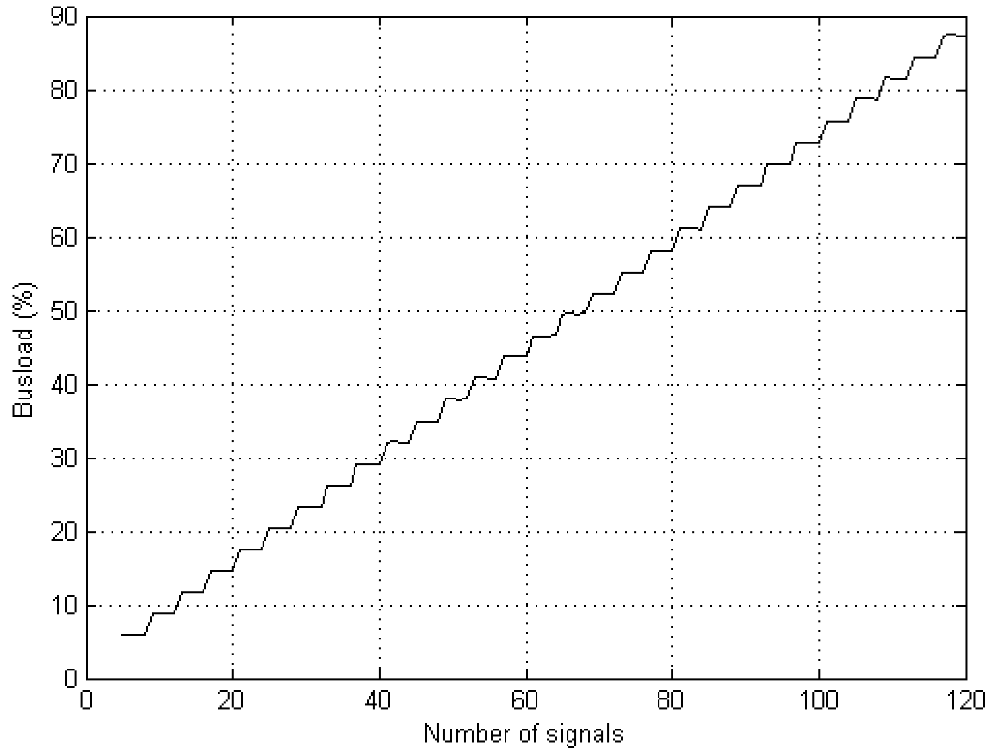

Figure 4.2 a) Busload as a function of number of signals for CCP. At most 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 1 signal is added for each measurement.

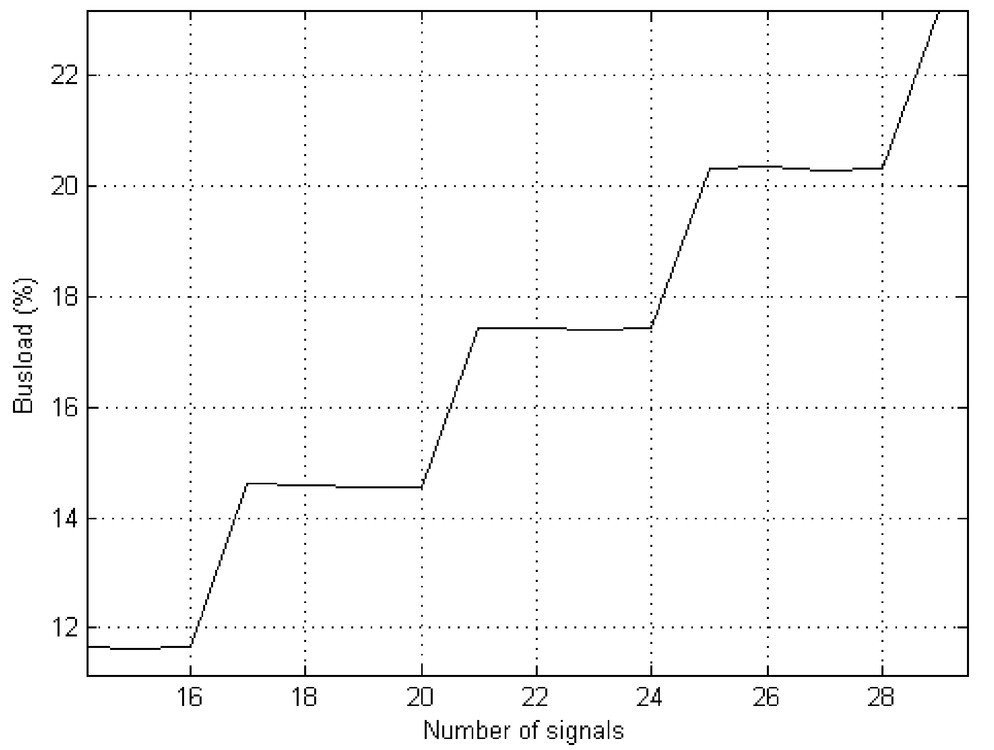

Figure 4.2 b) An enlarged part of Figure 4.2 a; busload as a function of number of signals for CCP. At most 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 1 signal is added for each measurement.

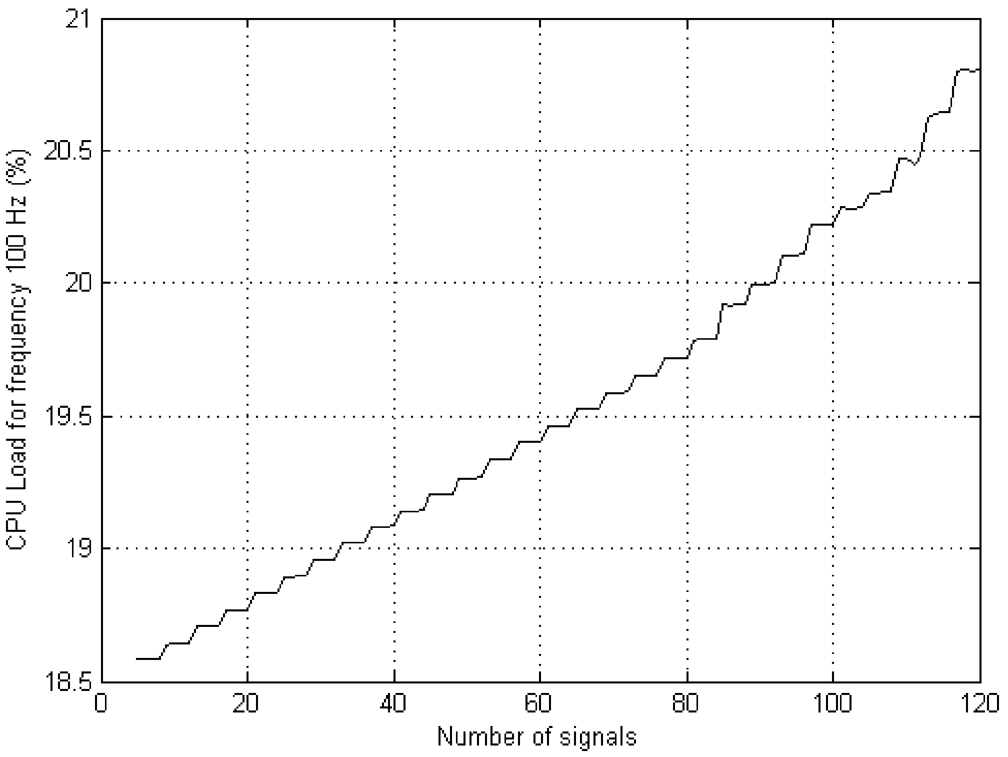

Figure 4.2 c) CPU load for frequency 100 Hz as a function of number of signals for CCP. At most 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 1 signal is added for each measurement.

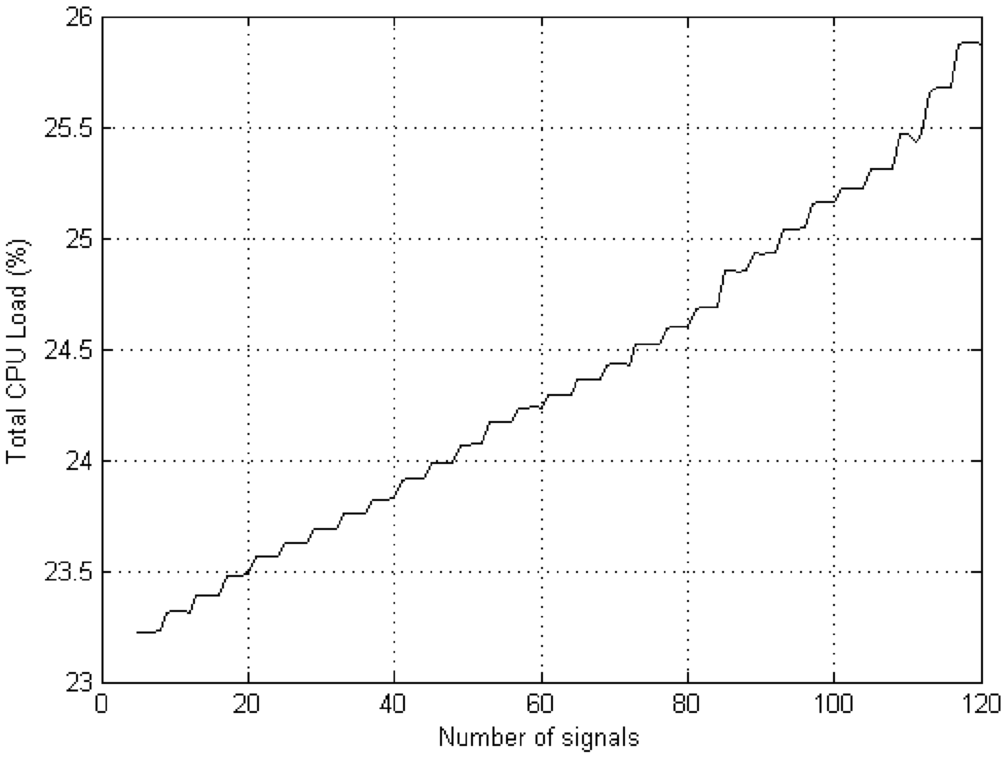

Figure 4.2 d) Total CPU load as a function of number of signals for CCP. At most 4 signals/DAQ-DTO with sizes 4, 1, 1, and 1 bytes are used and 1 signal is added for each measurement.

4.1.3 Comparison between CCP and KWP when adding 1 ODT per measurement

Figures 4.3 a-d show the comparison between CCP and KWP, first the time from the first request to the last response for KWP and from start to stop for DAQ, the busload for both CCP and KWP, the CPU load for frequency 100 Hz for both CCP and KWP and the total CPU load for both CCP and KWP as a function of number of signals. When 1 signal/DAQ-DTO (signal size 4 byte) is used 1 signal is added for each measurement. When 4 signals/DAQ-DTO (signal sizes 4, 1, 1 and 1 byte) are used 4 signal are added for each measurement and when 7 signals/DAQ-DTO (signal sizes 1, 1, 1, 1, 1, 1, and 1 byte) are used 7 signal are added for each measurement. The x-axis in the diagrams specifies the number of signals used with CCP and each CCP measurement is done during 12 ms. Each KWP measurement is performed until the number of KWP requests is equal to the number of signals transmitted via DAQ for the specified number of signals used with DAQ. The x-axis does thus not specify the number of the requests for KWP.

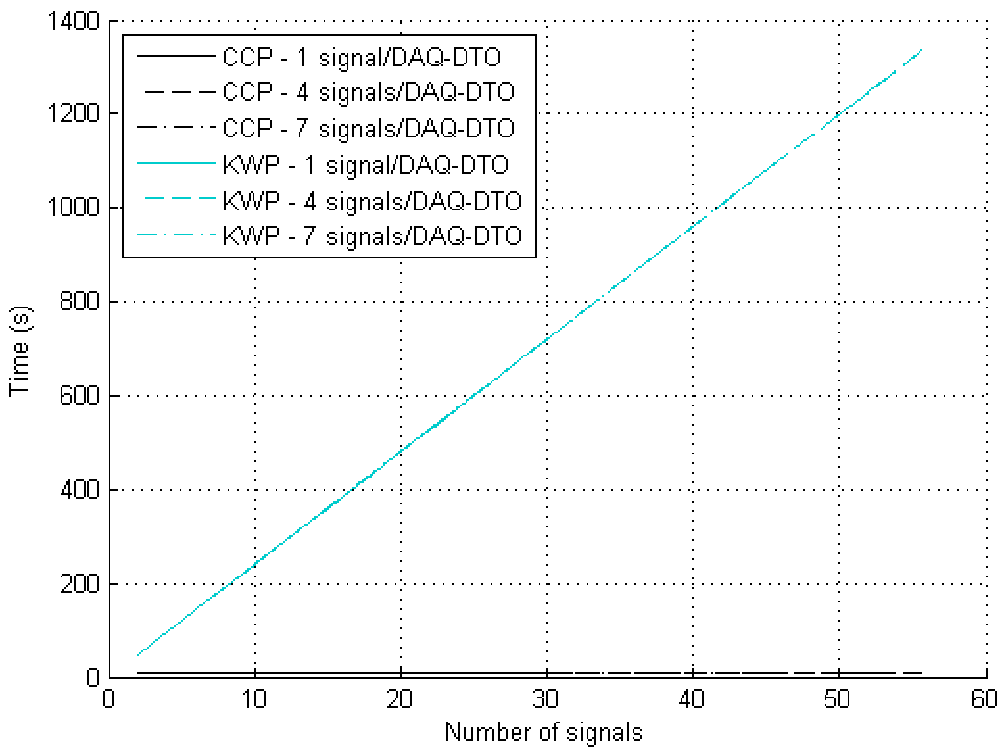

Figure 4.3 a) Time from the first request to the last response as a function of number of signals for KWP and the time from start of DAQ to stop of DAQ as a function of number of signals for CCP. All KWP graphs coincide. All CCP graphs coincide. The graphs corresponding 1 signal/DAQ-DTO[1] with sizes 4 bytes, starts with 2 signals and ends with 30 signals. 1 signal is added for each measurement. The graphs corresponding 4 signals/DAQ-DTO with sizes 4, 1, 1

[1] No DAQ-DTOs are present for KWP. The word DAQ-DTO is also used to refer to the KWP measurements that correspond to CCP measurements with a specific number of signals/DAQ-DTO.

and 1 bytes starts with 8 signals and ends with 56 signals. 4 signals are added for each measurement. The graph corresponding 7 signals/DAQ-DTO with sizes 1, 1, 1, 1, 1, 1 and 1 byte starts with 7 signals and ends with 56 signals. 7 signals are added for each measurement.

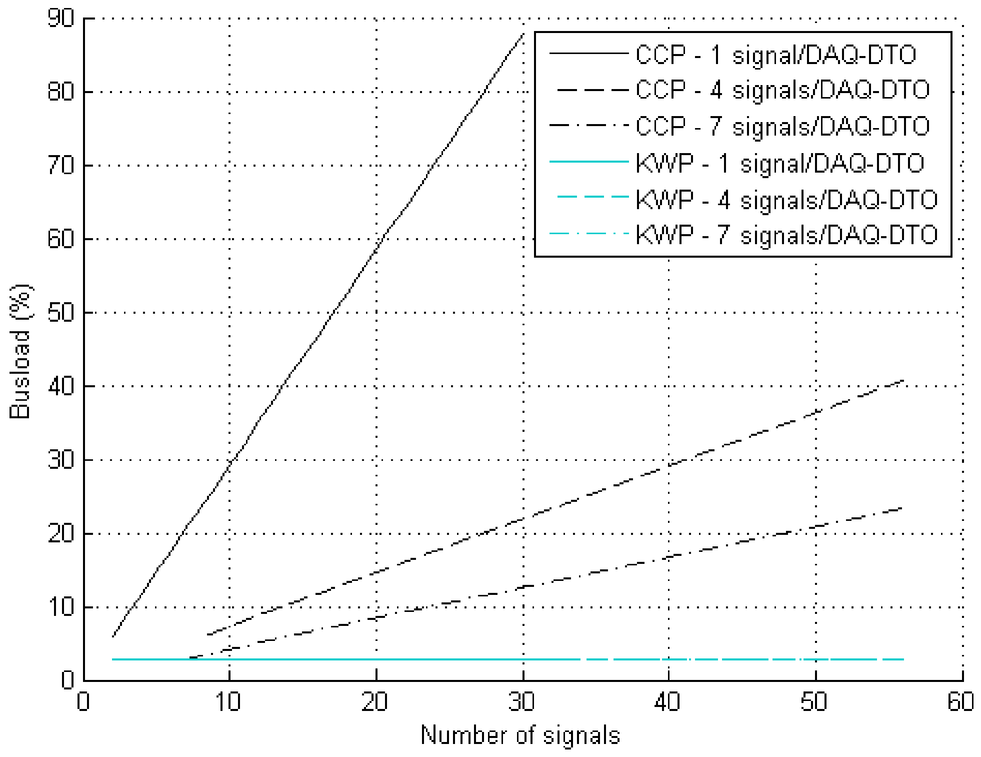

Figure 4.3 b) Busload as a function of number of signals for CCP and KWP. The graphs for KWP coincide. The graph corresponding 1 signal/DAQ-DTO[2] *with sizes 4 bytes starts with 2 signals and ends with 30 signals. 1 signal is added for each measurement. The graph corresponding 4 signals/DAQ-DTO with sizes 4, 1, 1 and 1 bytes starts with 8 signals and ends with 56 signals. 4 signals are added for each measurement. The graph corresponding 7 signals/DAQ-DTO with sizes 1, 1, 1, 1, 1, 1 and 1 byte starts with 7 signals and ends with 56 signals. 7 signals are added for each measurement.*

[2] No DAQ-DTOs are present for KWP. The word DAQ-DTO is also used to refer to the KWP measurements that correspond to CCP measurements with a specific number of signals/DAQ-DTO.

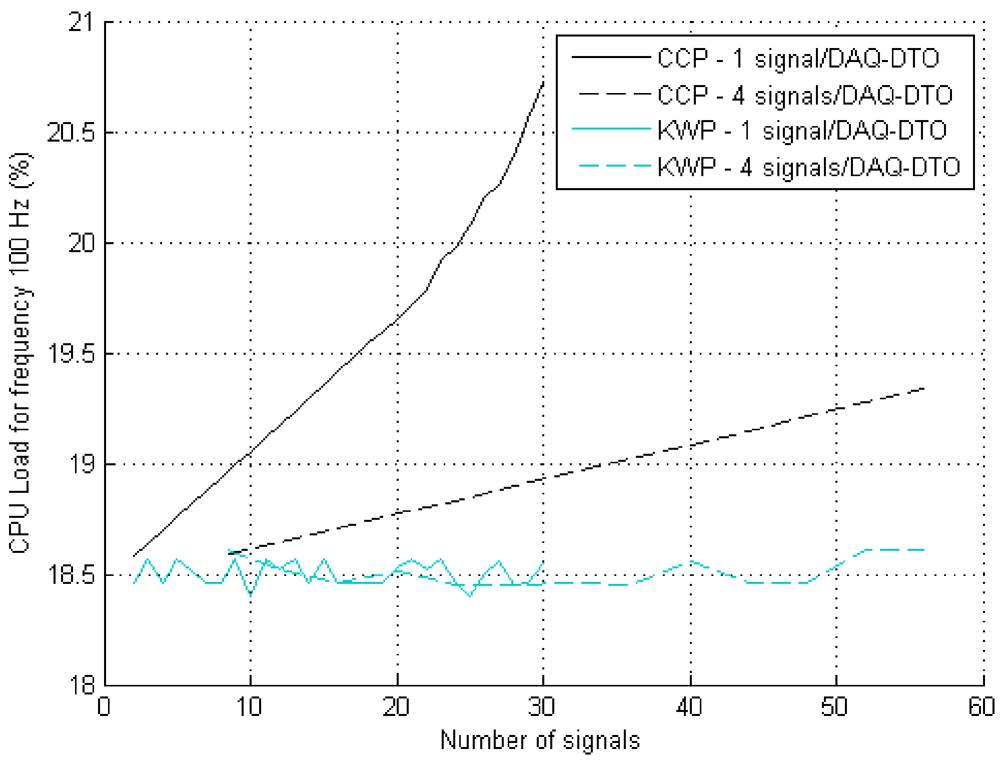

Figure 4.3 c) CPU load for frequency 100 Hz as a function of number of signals for CCP and KWP. The graph for 1 signal/DAQ-DTO[3] *with sizes 4 bytes starts with 2 signals and ends with 30 signals. 1 signal is added for each measurement. The graph with 4 signals/DAQ-DTO with sizes 4, 1, 1 and 1 bytes starts with 8 signals and ends with 56 signals. 4 signals are added for each measurement.*

[3] No DAQ-DTOs are present for KWP. The word DAQ-DTO is also used to refer to the KWP measurements that correspond to CCP measurements with a specific number of signals/DAQ-DTO.

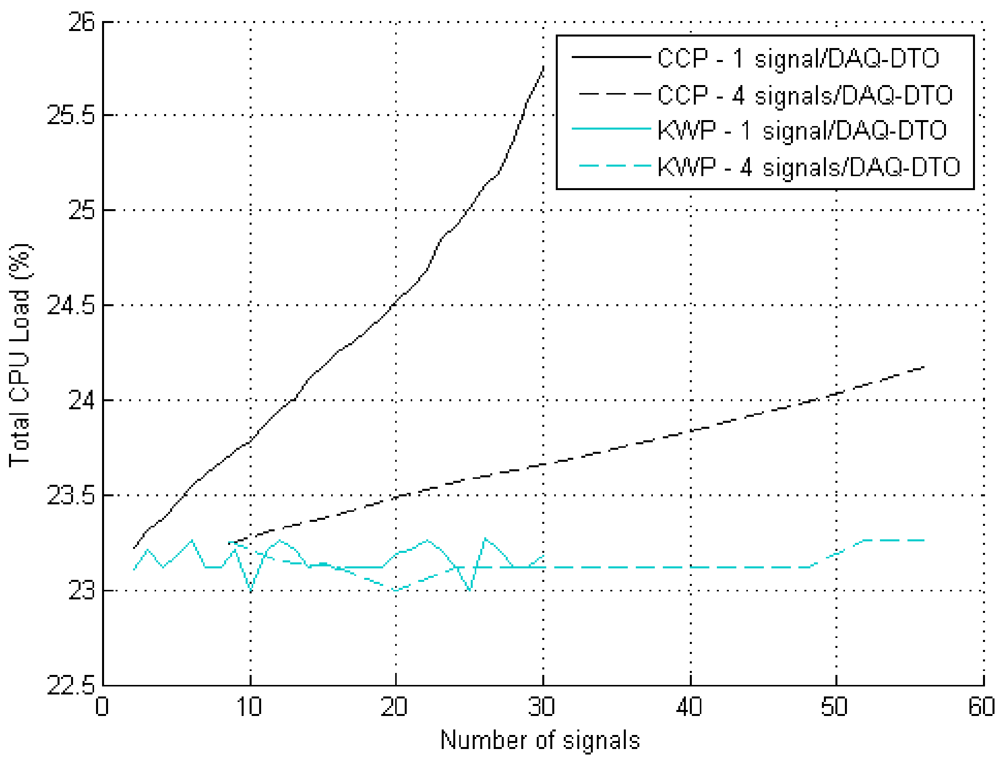

Figure 4.3 d) The total CPU load as a function of number of signals for CCP and KWP. The graph for 1 signal/DAQ-DTO[4] with sizes 4 bytes starts with 2 signals and ends with 30 signals. 1 signal is added for each measurement. The graph with 4 signals/DAQ-DTO with sizes 4, 1, 1 and 1 bytes starts with 8 signals and ends with 56 signals. 4 signals are added for each measurement.

4.2 Time to perform a KWP request

Some other measurements were also done besides measurements for producing the diagrams in previous section. To find out how long time 1 KWP request would take 67236 requests were performed. The time was measured to 1344688 ms and thus 1 KWP request takes 20 ms.

4.3 CPU load for CCP

In Table 4.1 results from other measurements are presented. The idea is to show the effect DAQ has on the CPU load for frequency 100 Hz and for the total CPU load. This is done by comparing the CPU loads for DAQ with 120 signals (maximum number of ODTs for the ECU that was used) with the CPU loads when no DAQ in not performed. The values for the CPU loads without DAQ were measured by performing KWP commands and calculating the average for those values.

[4] No DAQ-DTOs are present for KWP. The word DAQ-DTO is also used to refer to the KWP measurements that correspond to CCP measurements with a specific number of signals/DAQ-DTO.

Table 4.1 CPU load for frequency 100 Hz and the total CPU load for the cases DAQ with 120 signals (maximum number of ODTs for the ECU that was used) and when no DAQ is performed.

	CPU load for frequency 100 Hz (%)	Total CPU load (%)
DAQ 120 signals (30 ODTs)	20.81	25.89
Without DAQ	18.42	22.97

The difference in CPU load for frequency 100 Hz for DAQ when using 120 signals (30 ODTs) compared to when not using DAQ is 2.4 percentage points. The corresponding difference for the total CPU load is 2.9 percentage points.

4.4 The CCP SHORT_UP command

As described in Chapter 2 the CCP command SHORT_UP can be used to read data from the ECU. Tests were also performed for SHORT_UP and in the same way as for the KWP requests. The time for a SHORT_UP command varied between different executions from 1.39 ms to 2.03 ms. The busload was higher for shorter SHORT_UP time and lower for higher SHORT_UP time with mean busload values between 28 and 45 %. The values for the CPU load and the CPU load for 100 Hz for SHORT_UP does only differ in decimals compared to the values for the background CPU loads.

4.5 Theoretical calculation of the busload for DAQ with CCP

This section presents how many ODTs can be used with DAQ for a CAN bus with bit rate 500 kbit/s and CCP for a DAQ sample period of 10 ms. To do this the normal busload that is present in the truck without DAQ with CCP must be taken in account for. This is not necessary if a sub bus is used where no background busload is present. The theoretical worst case busload is given by

$$Busload\ (\%) = number\ of\ bits/frame * \frac{1}{bit\ rate} * 100 \sum_{i=1}^{n} \frac{1}{t_i} \qquad (4.1)$$

where the number of bits per frame is the number of bits in a CAN message, t_i is the sample period for message *i* that is periodically sent out on the CAN bus and *n* is the number of messages that are periodically sent out on the CAN bus. The formula does not take into account for event messages and thus the formula does not really give the worst case busload.

4.5.1 Example of the busload on a bus

In this subsection the busload is calculated for the bus that was connected to the ECU used in this thesis. The busload was calculated using (4.1) for messages that are periodically sent out on the CAN bus. The number of bits used for a CAN data message on the CAN bus varies between 131 to 154 bits depending on the number of stuffing bits (see Appendix C) that are needed. There were 100 messages for the bus connected to the ECU that was used. The sample periods for these messages are not presented here but resulted in 28.16 – 33.10 % busload depending on the number of stuffing bits. This means that there are 66.90 – 71.84 % busload that are not being used.

4.5.2 Busload for 1 DAQ-DTO

Formula (4.1) can also be used to calculate the theoretical busload used to transmit 1 DAQ-DTO message. The sample period is 10 ms which gives a busload of 2.62 – 3.08 % depending on how many stuffing bits are present.

4.5.3 Number of DAQ-DTO messages that can be used for DAQ

If assuming that all the remaining busload, calculated in subsection 4.5.1, can be used for DAQ 21 – 27 (66.90/3.08 – 71.84/2.62) DAQ-DTOs can be used at most. The DTO messages for DAQ must have very low priority if all remaining busload is used. This is due to that if the busload is high no essential messages shall be missed because of a DAQ-DTO message. It does not matter if DAQ-DTO messages are missed. Carefulness is necessary if the DTO identifier does not have the highest value compared to the other identifiers so that no important messages are lost. DAQ should not be used for busloads near 100 % in this case. If using DAQ on a sub bus with almost 100 % busload 32 – 38 DAQ-DTO messages can be used at most.

There are cases where 100 % busload cannot be used. One case is if the CAN circuit's send buffer does not sort the messages according to CAN identifier with highest priority first. There might also be a good idea to have some margin for event and error messages.

4.6 Measured busload for 1 DAQ-DTO

The busload for 1 DAQ-DTO was also calculated based on measurements. The background busload was first measured during a minute with 1 second intervals on bench. The mean value for the background busload on bench was 0.03 %. The reason why this value differs from the theoretical value is that there are only a few messages sent out on the bus for the ECU on the test bench. For an ECU in a truck many more signals are sent out on the bus. Thereafter the busload was measured for DAQ with CCP for 30 ODTs (30 four-byte signals and 90 one-byte signals) to 87.46 %. The busload for 1 DAQ-DTO is given by (87.46 – 0.03)/30 ≈ 2.91 %.

Chapter 5

Discussion

The discussion will evaluate the chosen method data acquisition (DAQ) with CCP and also compare it with the method using KWP-requests. The protocol XCP is also discussed.

The diagrams in subsection 4.1.1 only for DAQ with CCP are first commented. The busload increases linearly with the number of ODTs added. DAQ with CCP have significant influence on the busload. The CPU load for 100 Hz and the total CPU load are strictly increasing functions. DAQ with CCP has no major impact neither on the CPU load for 100 Hz nor on the total CPU load as seen in the diagrams. This is also shown by the measurements in section 4.3 where the background CPU loads are compared to the CPU loads with the maximum number of ODTs for the ECU used. In the diagrams in subsection 4.1.1, four signals (corresponding 1 ODT) are added per measurement. In the diagrams in subsection 4.1.2 one signal is added per measurement and this result in functions that increase only when a new ODT is needed. The added signals, that do not affect the number of ODTs, do not increase the bus traffic and thus there is no increase in the busload. The CPU load for 100 Hz and the total CPU load are not affected when no ODTs are added. The diagrams in subsection 4.1.2 are more correct than the diagrams in subsection 4.1.1 since the diagrams in subsection 4.1.2 are produced by measurements with a higher resolution for every increase in number of signals. The reason for also having the diagrams in subsection 4.1.1 is to show the difference between the two ways the measurements are performed. The method with lower resolution is used for the comparison of DAQ with CCP and KWP request due to that the KWP measurements take a lot of time.

The diagrams in subsection 4.1.3 that compares DAQ with CCP and KWP requests, will now be discussed. The first variable is time. For DAQ the time is not dependent on the number of signals used while the time increases for each KWP request. Figure 4.3 a) shows only the time for KWP and all of these values are greater than the time period of 12 seconds for DAQ. As more signals are added for DAQ, more ODTs will be needed and the frequency of the number of packets needed increase on the CAN bus. Therefore the busload will increase. For the KWP requests on the other hand the frequency of CAN messages remains the same independent on number of signals and the busload is thus more or less constant. The busload for DAQ is larger than for KWP. In the same way the CPU load for 100 Hz and the total CPU load also increase with the number of signals for DAQ when more ODTs are needed. The CPU load for 100 Hz and the total CPU load are more or less constant for KWP, independent of the number of the signals used. The CPU load for 100 Hz and the total CPU load are generally greater for DAQ via CCP compared to KWP for CCP with 10 ms sample period. The diagrams in subsection 4.1.3 also show differences in slope for e.g. busload between DAQ with CCP depending on the number of signals/DAQ-DTO. The 1 signal/DAQ-DTO curve (only four-byte signals) has the highest slope which can be explained by the ODT not having utilized as much as for the other cases since no more four-byte signals fit into one ODT. The 4 signal/DAQ-DTO curve (one four-byte signal and three one-byte signals per ODT) has the next highest slope which is explained with the same reasoning.

KWP is not an alternative for logging many variables from the ECU. Not even 1 signal can be requested with KWP within 10 ms. A KWP request takes 20 ms according to measurements described in section 4.2. CCP SHORT_UP is a better request method from a time perspective.

One SHORT_UP request takes 1.39 ms to 2.03 ms which is significantly faster than for KWP requests. This means that within 10 ms 4 to 7 signals can be transmitted with SHORT_UP. On the other hand the busload is significantly higher for CCP SHORT_UP (28 to 45 %) than for KWP (23 to 23.25 %). However, like KWP, CCP SHORT_UP is no alternative to DAQ with CCP. The main reason for this is that too few signals can be transferred with CCP SHORT_UP during 10 ms. The transmission times for KWP requests or CCP SHORT_UP requests can be compared with DAQ with CCP where 210 one-byte signals or 30 four-byte signals and 90 one-byte signals can be fetched during 10 ms. If 120 signals (30 four-byte signals and 90 one-byte signals) are transmitted during 10 ms the corresponding time for 1 signal would be 10/120 ms. A KWP request takes 240 times more time than DAQ with CCP, in the case with 30 four-byte signals and 90 one-byte signals with a sample period 10 ms for DAQ. This is a huge improvement in time for transferring ECU variables from the ECU compared to using KWP requests. The limitation of using DAQ with CCP is that the busload becomes high when using DAQ for many signals.

If using a CAN bus that is intended for not only DAQ with CCP, the number of ODTs that can be used will depend on the background busload. Assuming that the background busload (28.16 – 33.10 %) calculated in subsection 4.5.1 is included, 21 – 27 ODTs can be used in DAQ with CCP. If the number of ODTs is not enough, a way to increase the number of ODTs is to use a sub bus with no other bus traffic than DAQ-DTOs. If a sub bus is used for only DAQ 32 - 38 ODTs can be used, depending on the number of stuffing bits. In section 4.6 a practical measurement showed that a DAQ-DTO with sample period 10 ms yielded a busload of 2.91 %. This corresponds to $\lfloor 100/2.91 \rfloor = 34$ ODTs if using all busload capacity for DAQ with CCP. The number of ODTs that can be used is enough for today’s need. DAQ can be used with a longer sample period to reduce the busload if huge amount of data shall be logged. As stated in the results there are cases where 100 % busload cannot be used and there might also be a good idea to have some margin for event and error messages.

The busload becomes significantly higher for DAQ with CCP compared with using KWP requests. The busload might become an issue for DAQ with CCP. It is important that the DTO identifier has a higher value than identifiers of more important messages to decrease the risk that important messages are lost if a bus with other CAN traffic than DAQ is used. A sub bus with no background busload would allow a shorter sample time or more signals to be logged or both. No important CAN messages can be lost if a sub bus with only DAQ traffic is used.

Chapter 6

Conclusion and future work

6.1 Conclusion

DAQ with CCP is an excellent method for logging variables from ECUs since it may allow 240 times faster transfer rate of the ECU internal variables compared to the use of KWP requests. Consequently, DAQ with CCP increase the logging frequency of the ECU internal variables compared to the method using KWP requests.

6.2 Future work

In this thesis, the transmissions are done via a CAN bus. CCP is based on CAN and CCP does not support other bus systems. In the future another bus system might be used and in that case another communication protocol has to be used, e.g. the Universal Measurement and Calibration Protocol (XCP) that support many different transport layers.

DAQ with XCP also have some benefits compared to DAQ with CCP as described in Chapter 2. The benefits are as follows:

- DAQ with XCP allows use of 1 byte extra per ODT.
- Time stamping ability is available.
- The slave can configure DAQ lists dynamically.
- DAQ lists can be prioritized over other DAQ lists.
- Functionality that allows DAQ to start at power-up is also provided for DAQ with XCP.

The fact that DAQ with XCP allows use of 1 byte extra per ODT is a big benefit if many four-byte signals are logged since this allows two four-byte signals to fit in one DAQ-DTO instead of one.

In the future, more data than the data that can be transmitted today might be desirable. One solution is to implement XCP that allows 1 extra byte per ODT as already described. Another solution is to use a longer sample period. In the future, the bit rate might be higher which will allow faster transmissions and thus less busload. This will allow more data to be transmitted.

References

[1] TechTarget, bus, [Online]. Available: http://searchstorage.techtarget.com/definition/bus. [Accessed 27 September 2014].

[2] W. Voss, *A Comprehensible Guide To Controller Area Network*, 2nd ed., Greenfield: Copperhill Technologies Corporation, 2008.

[3] B. Ranjan and M. Gupta, Automobile Control System using Controller Area Network, *International Journal of Computer Applications*, vol. 67, pp. 34-38, 2013.

[4] K. Etschberger, *Controller area network : basics, protocols, chips and applications*, Weingarten: IXXAT Press, 2001.

[5] National Instruments, Controller Area Network (CAN) Overview, 1 August 2014. [Online]. Available: http://www.ni.com/white-paper/2732/en/. [Accessed 14 September 2014].

[6] CAN in Automation (CiA), CAN protocol, [Online]. Available: http://www.can-cia.org/index.php?id=systemdesign-can-protocol. [Accessed 14 September 2014].

[7] Robert Bosch GmbH, *CAN Specification Version 2.0,* Stuttgart: BOSCH, 1991.

[8] B. DeMuth and D. Eisenreich, *Designing Embedded Internet Devices*, Elsevier Inc., 2002.

[9] P. Zhang, *Advanced Industrial Control Technology*, Elsevier Inc., 2010.

[10] mysecurecyberspace, Communications Protocol, [Online]. Available: http://www.mysecurecyberspace.com/encyclopedia/index/communications-protocol.html. [Accessed 27 September 2014].

[11] techopedia, Communication Protocol, [Online]. Available: http://www.techopedia.com/definition/25705/communication-protocol. [Accessed 27 September 2014].

[12] ISO 14230 Road vehicles - Diagnostic systems - Keyword Protocol 2000 - Part 3, 1996.

[13] SSF 14230 Road Vehicles - Diagnostic Systems - Keyword Protocol 2000 - Part 3 - Application Layer, 2000.

[14] *Introduction to the CAN Calibration Protocol Version 3.0,* Vector CANtech, Inc., 2004.

[15] Kvaser, Introduction to the CCP/XCP protocols, [Online]. Available: http://www.kvaser.com/zh/about-can/related-protocols-and-standards/53-introduction-to-the-ccp-xcp-protocols.html. [Accessed 20 January 2014].

[16] K. Lemon, T. Dmuchowski and B. Emaus, *Introduction to CAN Calibration Protocol,* SAE International, 2000.

[17] H. Kleinknecht, *CCP CAN Calibration Protocol Version 2.1,* ASAP Arbeitskreis zur Standardisierung von Applikationssystemen, 1999.

[18] R. Zaiser, *CCP A CAN Protocol for Calibration and Measurement Data Acquisition*, Stuttgart, Germany: Vector Informatik GmbH.

[19] R. Schuermans et al., XCP Version 1.1 Part 1 - Overview, ASAM Association for

Standardisation of Automation and Measuring Systems, 2008.

[20] ASAM, ASAM MCD-1 XCP, 15 July 2013. [Online]. Available: https://wiki.asam.net/display/standards/asam+mcd-1+xcp. [Accessed 27 September 2014].

[21] K. Lemon, Introduction to the Universal Measurement and Calibration Protocol XCP, Vector CANtech, Inc, 2003.

[22] M. Di Natale, H. Zeng, P. Giusto and A. Ghosal, *Understanding and Using the Controller Area Network Communication Protocol Theory and Practice*, Springer, 2012.

[23] Kvaser, Kvaser CANlib SDK, [Online]. Available: http://www.kvaser.com/support/developer/canlib-sdk/. [Accessed 15 July 2014].

[24] Tutorialspoint, Python Overview, [Online]. Available: http://www.tutorialspoint.com/python/python_overview.htm. [Accessed 15 July 2014].

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd ed., London: The MIT Press, 2009.

[26] Kvaser, Kvaser CANLIB - CAN, [Online]. Available: http://www.kvaser.com/canlib-webhelp/group___c_a_n.html. [Accessed 16 July 2014].

[27] Kvaser, Kvaser CANLIB - canlib.h File Reference, [Online]. Available: http://www.kvaser.com/canlib-webhelp/canlib_8h.html. [Accessed 16 July 2014].

[28] Kvaser, Kvaser CANLIB - Programmer's Overview, [Online]. Available: http://www.kvaser.com/canlib-webhelp/page_user_guide_intro_programmers.html. [Accessed 17 July 2014].

[29] Kvaser, CAN Dictionary - Master the CAN terminology, [Online]. Available: http://www.kvaser.com/about-can/can-dictionary/. [Accessed 16 July 2014].

[30] Kvaser, Kvaser CANLIB - Code Examples, [Online]. Available: http://www.kvaser.com/canlib-webhelp/page_code_snippets_examples.html. [Accessed 17 July 2014].

[31] Kvaser, Kvaser CANLIB - Sending Messages, [Online]. Available: www.kvaser.com/canlib-webhelp/page_user_guide_send_recv_sending.html. [Accessed 17 July 2014].

[32] Kvaser, Kvaser CANLIB - Overruns, [Online]. Available: http://www.kvaser.com/canlib-webhelp/page_user_guide_send_recv_overruns.html. [Accessed 18 July 2014].

[33] Kvaser, Kvaser CANLIB - Handling Bus Errors, [Online]. Available: http://www.kvaser.com/canlib-webhelp/page_user_guide_bus_errors_error_frames.html. [Accessed 18 July 2014].

[34] Kvaser, Kvaser CANLIB - canBusStatistics_s Struct Reference, [Online]. Available: http://www.kvaser.com/canlib-webhelp/structcan_bus_statistics__s.html. [Accessed 18 July 2014].

Appendix A

Kvaser CANLIB functionality used in the code implementation

The CANLIB functions used in the code implementation will be described here. First the essential CANLIB code and functions are described which are shown in Figure A.1. To use the library, `canlib.h` must be included which is done on line 1. The CAN library is then initialized on line 4 with the function `canInitializeLibrary()` which is the first CANLIB function that shall be used. On line 5 CAN channel 0 is opened to a CAN circuit with the `canOpenChannel()` function. The channel number is hardware dependent and can be found in the Kvaser Hardware Configuration application. The first channel number is always 0 [26]. The second argument `canOPEN_ACCEPT_VIRTUAL` is used to allow virtual channels [27]. This feature is not needed but `canOPEN_ACCEPT_VIRTUAL` also allows the circuit to be shared with other applications, such as XCOM, that can be used simultaneously. `canOpenChannel()` returns a handle to the CAN circuit which will be used for subsequent calls to CANLIB functions [28]. Lines 6-12 handle a `canOpenChannel()` failure (`hnd < 0`) and will be described later in this subsection. Next function call is `canSetBusParams()` where the bus parameters are set such as the bit rate, in this case to 500 kbit/s by using the constant `canBITRATE_500K` defined in CANLIB. On line 14 a check is done for the status of the previous CAN command which will be described later. The `canSetBusOutputControl()` function is performed on line 15 to set the bus driver type and on line 16 the check function is called again. The next function call is to `canBusOn()` on line 17 and when this is performed the *CAN controller*, the chip that for instance handles the CAN protocol and message buffers, can participate in the bus traffic and is said to be *bus on* or *on the bus* [29]. CAN functions can now be used to e.g. send or receive messages. When done with the CAN communication some circuit cleanup is done by the function call to `canClose()` on line 21. The call to `canBusOff()` on line 19 is used to temporarily get off the bus and is actually not needed, but is there for completeness [28].

```
1  #include <canlib.h>
   ...
2  canHandle hnd;    // typedef int canHandle
3  int stat, channel = 0, bitrate = canBITRATE_500K;

4  canInitializeLibrary();
5  hnd = canOpenChannel(channel, canOPEN_ACCEPT_VIRTUAL);
6  if (hnd < 0) {
7      char errText[100];
8      errText[0] = '\0';
9      canGetErrorText((canStatus)hnd, errText, sizeof(errText));
10     fprintf(stderr, "canOpenChannel failed (%s)\n", errText);
11     exit(1);
12 }

13 stat = canSetBusParams(hnd, bitrate, 0, 0, 0, 0, 0);
14 Check("canSetBusParams", (canStatus)stat, EXIT);

15 stat = canSetBusOutputControl(hnd, canDRIVER_NORMAL);
```

```
16 Check("canSetBusOutputControl", (canStatus)stat, EXIT);

17 stat = canBusOn(hnd);
18 Check("canBusOn", (canStatus)stat, EXIT);

   // Send and receive CAN-messages here

19 stat = canBusOff(hnd);
20 Check("canBusOff", (canStatus)stat, EXIT);

21 stat = canClose(hnd);
22 Check("canClose", (canStatus)stat, EXIT);
```

Figure A.1 Essential code to be able to participate in the bus traffic and code to get off the bus.

The check function will now be described in more detail and the code can be seen in Figure A.2. It is a modification of the check function in the Kvaser CANLIB documentation [30]. `Check()` takes 3 arguments. The first argument contains the name of the CANLIB function that was called. The second argument is the status code from the called CANLIB function. The last argument shall contain a nonzero value if the program shall exit in case there was an error and zero to not exit. On line 3 the check is done to see if the status from the called CANLIB function was successful (`canOK`) or not. If the call was not successful the function `canGetErrorText()` is called on line 6 to retrieve a error text which will be printed on line 8 if `canGetErrorText()` was successful and otherwise on line 10. The function finally ends by exiting the program if `exitNow` is set to a nonzero value. The code on line 6-12 in Figure A.1 is similar to `Check()` with the differences that no check is done for the `canGetErrorText()` function, the error condition differs and the program will always exit if there is a failure.

```
/* Check
 *
 * Checks if the CAN status code stat is not OK and if that is the case a error
 * message, starting with msg failed, is printed and the program might exit
 * depending on the value of exitNow
 *
 * [IN] msg: Function name of the function that you want to see if it failed
 * [IN] stat: The status code returned by the function used that you want to see
 *            if it failed
 * [IN] exitNow: Set to 1 if you want to exit the program if failure, otherwise
 *               set it to 0
 */
1  void Check(char * msg, canStatus stat, int exitNow) {
2      char errText[100];
3      if (stat != canOK) {
4          int stat2;

5          errText[0] = '\0';
6          stat2 = canGetErrorText(stat, errText, sizeof(errText));
7          if (stat2 == 0)
8              fprintf(stderr, "%s failed: error code=%d (%s)\n", msg, (int)stat,
                       errText);
9          else
10             fprintf(stderr, "%s failed: error code=%d (not a valid error
                       code)\n", msg, (int)stat);
11         if (exitNow) {
12             exit(1);
13         }
14     }
15 }
```

Figure A.2 Error checking for the status code from CANLIB call.

The function `canWrite()` is used to send CAN messages and a function call is shown in Figure A.3. `canWrite()` takes first a handle to a open CAN circuit as the first parameter, a CAN identifier as the second parameter, the data in a CAN message as the third parameter, the length of the data field as the forth parameter and finally the parameter flags [26]. The CAN identifier used for sending data to the ECU with CCP is the CRO identifier. `msg` will contain data corresponding to a CCP command. The flag parameter will have the value `canMSG_EXT`, defined in CANLIB, which is the identifier for the extended 29 bit CAN format. `canWrite()` queues the CAN message in a transmit queue that is emptied according to the principle first-in, first-out (FIFO), i.e. the messages are sent in the same order they arrived to the queue. `canWrite()` returns immediately after the message is queued while the similar function `canWriteSync()` is used to wait until the message has been sent [31].

```
canHandle hnd;     // typedef int canHandle
unsigned long CROIdentifier;
BYTE msg[8];       // typedef unsigned char BYTE
unsigned int flags;
  ...
stat = canWrite(hnd, CROIdentifier, msg, 8, flags);
Check("canWrite", (canStatus)stat, EXIT);
```

Figure A.3 A call to `canWrite()` that is used to send can messages.

There are numerous variants of functions to use for reading CAN messages. The function used in my implementation is `canReadWait()` which is called in Figure A.4. This function retrieves the first message in a receive buffer. If the receive buffer is empty `canReadWait()` will wait until a CAN message appears in the buffer or until timeout occurs. If timeout occurs the error code `canERR_NOMSG` is returned. `canReadWait()` takes a handle to a open CAN circuit as the first parameter, a CAN identifier as the second parameter, a pointer to a buffer for the data to be received as the third parameter, a pointer to buffer for the message length as the forth parameter, a pointer to a buffer for the message flags as the sixth parameter, a pointer for the message time stamp as the seventh parameter and the timeout (in milliseconds) as the last parameter [26]. When calling `canReadWait()` after a CCP command is performed the timeout parameter will have value specified in [17] for that command. The CCP specification contains a table that specifies timeout to acknowledgement for each CCP command [17]. The identifier used when reading messages from the receive buffer is the DTO identifier for DAQ with CCP. Thus `canReadWait()` is put in a loop that loops until a message with the DTO identifier is found. Another way of reading CAN messages with a specific identifier is to use `canReadSyncSpecific()` and `canReadSpecificSkip()`. `canReadSyncSpecific()` waits until a message with a certain identifier is available in the receive buffer. `canReadSpecificSkip()` reads the first message in the receive buffer with a certain identifier and throws away the messages before that message. The reason for not using `canReadSyncSpecific()` and `canReadSpecificSkip()` is that errors and overloads are not caught with `canReadSpecificSkip()` if an identifier is used that is not the identifier for a errors and overload frames. The advantage of using `canReadWait()` is that the identifier of each CAN message can be compared to different identifiers which means that errors and overloads can be found. For that reason `canReadWait()` was chosen.

```
canHandle hnd;     // typedef int canHandle
long id;
BYTE msg[8];       // typedef unsigned char BYTE
unsigned int dlc, flags;
unsigned long time, timeout;
  ...
stat = canReadWait(hnd, &id, msg, &dlc, &flags, &time, timeout);
Check("canReadWait", (canStatus)stat, exitNow);
```

Figure A.4 Code retrieves the first message in the receive buffer. If no message is available, `canReadWait()` *waits for a CAN message to appear in the receive buffer or returns an error code if timeout occurs.*

The flag parameter in `canReadWait()` would normally contain flags corresponding a data frame which is either `canMSG_STD` or `canMSG_EXT`. The flag parameter might contain overrun flags (`canMSGERR_OVERRUN`) or an error flag (`canMSG_ERROR_FRAME`) and if that is the case special actions can be performed. An overload condition occurs if the driver or the CAN interface runs out of buffer space or the busload is so high that the controller cannot keep up with the traffic [32]. An error frame containing an error flag is transmitted when the CAN controller detects an error [33]. These frames should occur rarely and indicates often a fatal error such as some node transmitting at wrong bit rate, bad cable etc. [33]. The error frames and overruns can be found by using *bitwise and* between the flag parameter for a message fetched from the receive buffer and the e.g. overrun flags (`canMSGERR_OVERRUN`) and compare this to 0. If the result is 0 then there were no overrun but if the result was a none zero value then there was a overrun and appropriate actions can be performed. This is shown in Figure A.5. In the example code an error message is printed to the user, the data acquisition is stopped and finally the program exits. The function `stopDAQ_DTOTransmission()` is not a CANLIB function and will not be described in more detail.

```
if ((flags & canMSGERR_OVERRUN) != 0) { // Overrun
    fprintf(stderr, "Driver ran out of buffer space or the busload is so high
            that the CAN controller can't keep up with the traffic.\n");
    stopDAQ_DTOTransmission(CTR, CROId, DTOId, DAQListNum, EventChNo);
    exit(1);
}
```

Figure A.5 Code that checks if any of the two possible overrun errors have occurred, by comparing the overrun flags (`canMSGERR_OVERRUN`*) with the flags for the received message from the receive buffer, and if overrun(s) have occurred actions are performed.*

Kvaser's CANLIB also provides functionality for retrieving statistics about busload, number of error frames, number of received extended data frames, standard data frames, number of overruns etc. [34]. This statistics must first be requested from hardware by calling `canRequestBusStatistics()` and can thereafter be retrieved in a `canBusStatistics_s` object using `canGetBusStatistics()` [26]. The first function `canRequestBusStatistics()` takes only a handle as parameter and the second function `canGetBusStatistics()` takes a handle as first parameter, a pointer to a `canBusStatistics_s` struct as second parameter and the size of the struct buffer in bytes as third parameter [26]. Since CCP can have big influence on the busload, depending on how many ODTs that are needed, busload checks are perform every second in the DAQ application developed. This is achieved by calling `canRequestBusStatistics()` and `canGetBusStatistics()` periodically with 500 ms intervals. The time between the call to `canRequestBusStatistics()` and the time when the statistics actually is retrieved via `canGetBusStatistics()` is not defined [26]. Therefore busload checks are not done more frequently. Figure A.6 shows how this is achieved where that code part is run every 500 ms. The busload variable is number between 0 and 10000 which represents 0.00 - 100.00 % busload. The statistics are cleared when the corresponding channel goes on the bus [34].

```
if (!getBusload) {     // Requests bus statistics from the hardware
    stat = canRequestBusStatistics(hnd);
    Check("canRequestBusStatistics", (canStatus)stat, DONT_EXIT);
    getBusload = true;
}
else {                 // Retrieve statistics
    canBusStatistics_s canBusStat;
    stat = canGetBusStatistics(hnd, &canBusStat, sizeof(canBusStatistics_s));
    Check("canGetBusStatistics", (canStatus)stat, DONT_EXIT);
    if (canBusStat.busLoad > 8000)
        fprintf(stderr, "Note that the busload (%f %%) exceeds the recommended
                busload limit (80 %%).\n", canBusStat.busLoad/100.0);
    getBusload = false;
}
```

Figure A.6 Code that shows how CAN statistics are requested from hardware and then retrieved the next time this code is visited. A busload check is done and a warning text is printed if the busload is high.

Appendix B

Organisation of the code into different files

The implementation is divided into several files to facilitate the reading of the code. Two files (a .cpp and a .h file) are used for the main program that performs the CANLIB and CCP commands to accomplish DAQ. The pure CCP functions are implemented in a separate (.cpp) file (with a corresponding .h file). The functionality for reading data from the ati file is also separated to an own (.cpp) file (with a corresponding .h file). A separate (.h) file is used for the signal struct and this is also the case for the signal in ODT struct.

Appendix C

Number of bits for an extended CAN data frame on the CAN bus

Here the number of bits in an extended (29-bit identifier) CAN data frame on the CAN bus, including interframe space, is calculated. A CAN extended data frame consists of a start of frame (SOF) bit (1 bit), an arbitration field (32 bits), a control field (6 bits), a data field (0 to 64 bits with 8 bits increment), a CRC field (16 bits), an ACK field (2 bits) and an end of frame (7 bits). In the calculations used for the busload the CAN messages used have a 64 bit data field which yields in total 128 bits. If a CAN data frame has been sent on the CAN bus there must be 3 bits interframe space before the next data frame can be sent. When sending the CAN data frame on the CAN bus stuffing bits might be added to the data frame. The worst theoretical case would be if every bit is the same which would yield a stuffing bit for every 5:th bit for the bits start of frame to the CRC delimiter. This results in $\left\lfloor \frac{(1+32+6+64+16)}{5} \right\rfloor = 23$ stuffing bits. Thus there are 131 to 154 bits needed for each CAN data frame sent on the bus.