CAN TOOL FOR ATLAS COPCO’S RIG CONTROL SYSTEM

Marcus Ljung and Per Turner

Computer Engineering Programme, 180 Credits

Örebro, Sweden, Spring 2013

Examiner: Lars Karlsson

CAN-VERKTYG FÖR ATLAS COPCOS RIGGKONTROLLSYSTEM
Abstract
The goal of this 15 hp thesis in computer engineering was to develop a tool for monitoring and analyzing the data flow on a Controller Area Network (CAN) called Rig Control System (RCS) that is used by Atlas Copco. Atlas Copco develops and manufactures machines for Mining and Rock Excavation. The Rocktec division is responsible for the Rig Control System platform used on machines in all division within the Mining and Rock Excavation Technique (MR) business area. The tool’s primary purpose is monitor and analyze data from the RCS network and present the analyzed data in an easy way to help with development and maintenance of RCS and machines that use RCS. The advantages and how data is sent over the CAN bus are presented as well as the CANopen protocol which is a higher layer protocol based on CAN. Two ways of data acquisition from RCS are presented, a simulated environment and real hardware. Different types of interprocess communication are presented as well as the pros and cons of each of these types. The creation of the tool required a Graphical User Interface (GUI) so different frameworks for this task are also presented and discussed. A version of the tool is presented and discussed in detail. The result of the project is a tool that with further development can be of great use to developers and service engineers working with RCS.

Sammanfattning
Preface

We would like to thank all the coworkers at Atlas Copco, especially Jan Knutsson, Lars Sandström, Fredrik Grahn and Hans Gustavsson for all the help they have given us during the project. We would also like to thank Jack Pencz for the support.
Contents

1 INTRODUCTION ................................................................................................................. 5
  1.1 BACKGROUND ............................................................................................................. 5
  1.2 PROJECT ..................................................................................................................... 5
  1.3 OBJECTIVE ............................................................................................................... 5
  1.4 REQUIREMENTS ......................................................................................................... 6
2 METHODS AND TOOLS ....................................................................................................... 7
  2.1 METHODS ................................................................................................................... 7
  2.2 TOOLS ....................................................................................................................... 7
    2.2.1 Kvaser USBcan Rugged HS/HS ........................................................................... 7
    2.2.2 Version control .................................................................................................... 8
    2.2.3 Documentation of code ....................................................................................... 8
  2.3 OTHER RESOURCES ................................................................................................. 8
3 IMPLEMENTATION ............................................................................................................. 9
  3.1 RETRIEVAL OF INFORMATION AND USE CASES .................................................. 9
  3.2 THE CAN BUS PROTOCOL ....................................................................................... 9
    3.2.1 Error control in CAN bus .................................................................................... 10
  3.3 CANOPEN .................................................................................................................. 11
  3.4 RIG CONTROL SYSTEM ............................................................................................ 12
  3.5 GRAPHICAL USER INTERFACE ................................................................................. 13
    3.5.1 Choice of framework for the Graphical User Interface (GUI)................................. 13
    3.5.1.1 Microsoft Foundation Class (MFC) ................................................................. 13
    3.5.1.2 wxWidgets .................................................................................................... 13
    3.5.1.3 Windows Presentation Foundation (WPF) ....................................................... 13
    3.5.1.4 Windows Forms ............................................................................................. 13
    3.5.1.5 Qt .................................................................................................................. 13
    3.5.1.6 Windows Template Library (WTL) ................................................................. 14
    3.5.1.7 Analysis ......................................................................................................... 14
    3.5.2 The bus view ........................................................................................................ 14
    3.5.2.1 Updating the bus view ..................................................................................... 15
    3.5.3 Analysis of data ................................................................................................... 15
    3.5.4 Filter data ............................................................................................................ 16
    3.5.5 Module view ........................................................................................................ 16
    3.5.6 Calculation of bus load ....................................................................................... 17
  3.6 INTER-PROCESS COMMUNICATION (IPC) .............................................................. 18
    3.6.1 D-Bus .................................................................................................................. 18
    3.6.2 Transmission Control Protocol .......................................................................... 18
    3.6.3 Relational Database Management System ....................................................... 18
    3.6.4 Shared memory .................................................................................................. 19
    3.6.5 Named pipe ....................................................................................................... 19
    3.6.6 Analysis ............................................................................................................. 19
  3.7 RETRIEVAL OF DATA FROM RCS SIMULATION .................................................... 20
  3.8 RETRIEVAL OF DATA FROM KVASER USBcan RUGGED HS/HS ......................... 20
  3.9 RETRIEVAL OF DATA FROM LOG FILE ................................................................ 21
  3.10 SAVING DATA TO A LOG FILE ................................................................................ 21
  3.11 I/O MODULE .......................................................................................................... 21
  3.12 I/O MODULE CLASS ............................................................................................. 22
  3.13 DECODER/RESOLVER CLASSES .......................................................................... 22
4 RESULT ............................................................................................................................ 23
  4.1 PERFORMANCE ......................................................................................................... 23
  4.2 TESTING ..................................................................................................................... 23
  4.3 USAGE ....................................................................................................................... 23
5 DISCUSSION .................................................................................................................. 26
5.1 SOCIAL ASPECTS ....................................................................................................... 26
  5.1.1 Economic aspects ................................................................................................. 26
  5.1.2 Safety aspects ...................................................................................................... 26
5.2 ACHIEVEMENT OF THE COURSE OBJECTIVES .............................................. 26
  5.2.1 Knowledge and comprehension ........................................................................... 26
  5.2.2 Proficiency and ability ........................................................................................ 27
  5.2.3 Values and attitude .............................................................................................. 27
5.3 COMPLIANCE WITH THE PROJECT REQUIREMENTS ........................................ 27
5.4 PROJECT DEVELOPMENT ....................................................................................... 28
5.5 FIGURE DESIGNATION .............................................................................................. 29
6 REFERENCES ................................................................................................................. 30

APPENDICES
  A: Use Case Scenarios
1 Introduction

In this chapter the background to the thesis is described in more detail, the project and the objectives of the project are discussed and the requirements for the project are listed.

1.1 Background

Atlas Copco develops and manufactures machines for Mining and Rock Excavation. Atlas Copco is divided in different divisions. The Rocktec division was responsible for a Controller Area Network (CAN) based control system platform used on machines in all division within the Mining and Rock Excavation Technique (MR) business area. Atlas Copco has 39800 employees in 86 different countries whereof 1900 are located in Örebro (Sweden) [1]. A need was identified to monitor data from the CAN bus in an easy way. Data would be presented in a signal view where all sensor and actuator signals easily would be identified. The expected use would be by service engineers on field and by Rig Control System (RCS) developers. It would help them to develop RCS further and help them when diagnosing machines that use RCS.

Earlier Atlas Copco used a program that was called CANLab, which was a general program aimed at monitoring and analyzing data on a CAN bus network. The problem with CANLab and similar programs was that they could not analyze or parse data from RCS. This made it hard for users to decipher the data into understandable texts. On top of that the system was very expensive, which meant that not every user had access to the software at their computer. Some tools already existed for helping developers with this problem: a program that logged all the data to file and a program that parsed the data into more readable form. These programs were only in the prototype stage and could not show real-time data so further development was needed to make them useful.

1.2 Project

This project was for the most part a development project but sometimes also an inquiry project. The task was to create a software program that read all the data from the CAN bus, and to create a Graphical user interface (GUI) to show the data in different views. The program would be able to read data in real time interacting with a USB CAN interface from Kvaser which is hardware connected through a CAN bus to a machine [2]. The program would also be able to read data from a log file and show the data in the GUI. The last data source the program had to be able to read from was a RCS simulator. The program would also have the ability to save all the data to a log file on the computer. It would also view bus load and error frames, this would be done by reading from the CAN bus directly or through the USB CAN interface. It would also have a view with analyzed data, meaning that the data was going to be deciphered to understandable texts.

1.3 Objective

The objective with this project was to create software that could read data from the CAN bus and show it in views that helped Atlas Copco’s developers and service engineers in their work. The most important part of the program would be to show analyzed data to the user. One more important key factor was that the earlier used software cost a lot of money, which led to the fact that not every developer could take part in this work. The vision was that every developer would have the software on their computer to help with the debugging of CAN bus specific problems.
1.4 Requirements

- Test application using Application Programming Interface (API) for USB CAN family of products from Kvaser
- Create classes for each of the following module type:
  - Input/Output Module (I/O)
  - Decoder
  - Resolver
- Translation of CAN messages into RCS I/O table and signal list for presentation
- Create a GUI for the following purposes:
  - Bus View
  - Module view
  - Bus statistics
- Select and view data from a node on the CAN bus, both inputs and outputs
- Select and view data from a specific message/sensor. Choose signals from the signal list to the monitor. Shall be presented as Analog and Digital (AD) values and physical units
- Evaluation of CAN data. Are all data delivered within specified time frames?
- Bus statistics such as BUS load and error frames shall be presented in real-time
- Select and view live CAN messages
- It shall be possible to monitor two CAN-busses in the same view
- Demonstration of a complete application
2 Methods and tools

In this chapter the methods, tools and other resources that have been used in the project are listed and explained.

2.1 Methods

The programming language was required to be C++, but the GUI implementation of the program was left for us to decide which framework to use. Specifications for CAN Open were provided. Agile development methods, containing a mix of extreme programming and scrum, were used, especially pair programming and rapid prototyping. It was an iterative development process where prototypes with certain goals were created every two weeks. These prototypes were demonstrated to a group of developers several times where important feedback was gathered. This helped with the development of the project since additions and changes could be made based on the feedback received on the prototype during the demonstration.

2.2 Tools

The development environment that was used for this project was Visual Studio 2005 that came preinstalled on the computers that Atlas Copco provided. The computers came with Windows 7 preinstalled as the operating system. Other tools used were Microsoft Office 2010, Lotus Notes and Acrobat Reader as help for planning, meetings and documentation. As for the more programming and simulation part IncredIBuild, Perl, Python 2.7, TortoiseHg, Mercurial, Visual Assist X, Qt Creator and FogBugz were used. Atlas Copco provided everything that was needed for successful completion of the project.

2.2.1 Kvaser USBcan Rugged HS/HS

The hardware used to connect the program with an RCS machine was the Kvaser USBcan Rugged HS/HS, see Fig. 1. This hardware is an interface between the CAN bus on a machine and the USB port on a computer. It has the ability to connect to two separate CAN networks at once and can collect data from them simultaneously. It also has the ability to calculate bus statistics such as bus load and the type of messages received for each connected CAN network. It is also built to be able to withstand water and dust which makes it ideal to be used on Atlas Copco machines which usually work in harsh environments. [2]
2.2.2 Version control
To make the coding process more efficient, a version control system was used. The candidates for representing the project were Git, Subversion and Mercurial. Mercurial was chosen mostly because Atlas Copco already used it in other projects. A version control system is used to be able to go back and start over from a previous point, often when a mistake has been done. More benefits are that more than one person can work on a project simultaneously without problems.

2.2.3 Documentation of code
To make sure that the code of the project would be easy to work with after the project was over a good documentation tool was needed. The choice fell on Doxygen. Doxygen is currently used by Atlas Copco to document classes and public functions. Doxygen is a widely used tool for creating documentation from source code.

2.3 Other resources
To search for academic texts Örebro University’s databases were used. Atlas Copco’s internal Wiki where specifications and documentation for RCS was located was used. A simulator of the RCS system to be able to test the program without access to real hardware was also used. To get a good start on the project, and not having to write everything from scratch, Atlas Copco’s development libraries were used. Access was given to Atlas Copco’s laboratory equipment for testing the program on real hardware. The laboratory results, the log files and the simulator were used to get the data that was needed. We had a lot of meetings, both scheduled and unscheduled, with people that were involved with the project. These meetings helped a lot in the development of the project. Communication was very quick and effective by using Lotus Notes.
3 Implementation
This chapter describes the implementation of the project. The CAN Bus protocol, CANopen and Rig Control System are described in detail. The choice of framework for the GUI is explained and the inter-process communication choices for the project are discussed and analyzed. The implementations of the different parts of the GUI are also described.

3.1 Retrieval of information and use cases
To be able to start the project a meeting took place where the requirements, tools, GUI and what was expected came up. To get a good start on the project use cases for the most common tasks were created. They gave a good starting point and something to build on. They were also a good reference when designing the program.

3.2 The CAN Bus protocol
The CAN bus was developed by BOSCH as a serial communication network bus. It is used in many industrial applications and has proven to be very reliable and has a high tolerance for interference. The low cost and high speed of the CAN bus has made it a popular choice for use in field bus applications [18]. The CAN bus can move data up to 1 Mbit per second. One more advantage is that the CAN bus can be implemented in real-time systems. The CAN bus is different from USB and Ethernet because it sends many short messages broadcast over the entire network instead of sending only big blocks of data. The CAN bus protocol is also able to handle faulty nodes without bringing down the entire network [19]. A CAN bus can only have two logical values. These values are ‘dominant’ and ‘recessive’. In most implementations of the CAN bus protocol the ‘dominant’ value is zero and the ‘recessive’ value is one.

There are four types of messages that can be sent over the CAN bus. These are:

- DATA FRAME
  This type of message contains data that is sent from a transmitter to one or more receivers. It is the most common type of message on a CAN bus network.

- REMOTE FRAME
  This type of message is used whenever a node on a CAN bus network needs to request data from another node. It will then send a REMOTE FRAME to this node to initiate the transmission of the requested data.

- ERROR FRAME
  This type of message is sent whenever an error is detected by any node on a CAN bus network. The layout of an error frame is not in accordance with the CAN specification for the DATA FRAME layout. If a node detects an error in a received message it immediately sends an error frame which, when received by other nodes, triggers them to also send an error frame. When the transmitter of the faulty message receives the error frame it will try to send the message again. An error frame overrides any other message currently on the bus.

- OVERLOAD FRAME
  This type of message is used whenever a node receiving data is unable to keep up with the amount of incoming data. Whenever a node that is transmitting data receives an error frame it holds off on sending new data for a short period of time to let the receiver catch up.
The only message type that was used in the program was the DATA FRAME type. A CAN bus data frame consists of seven different bit fields. See Fig. 6. These fields are:

- **START OF FRAME (SOF)**
  This bit field marks the beginning of a data frame.

- **ARBITRATION FIELD**
  This bit field consists of the CAN identifier and the RTR (Remote Transmission Request) bit. The CAN identifier is 11 bits long (v. 1.x and 2.0A) or 31 bits long (v. 2.0B) and is used to identify which node the DATA FRAME originated from. (Atlas Copco’s Rig Control System uses 11 bits long CAN identifiers.) The RTR bit is used to identify if the message is a DATA FRAME or a REMOTE FRAME.

- **CONTROL FIELD**
  This bit field contains the DATA LENGTH CODE. It is six bits long and the DATA LENGTH CODE occupies 4 of these bits. The other two bits are reserved bits. The DATA LENGTH CODE contains the number of bytes in the DATA FIELD.

- **DATA FIELD**
  This bit field contains the actual data of the DATA FRAME. It can be 0 to 8 bytes long.

- **CRC FIELD**
  This bit field contains the CRC SEQUENCE (Cyclic Redundancy Code) which is used for checking that the frame has not been corrupted during transfer and a CRC DELIMITER which is one bit long and is ‘recessive’.

- **ACKNOWLEDGE FIELD**
  This bit field consists of the ACK SLOT and the ACK DELIMITER. The ACK SLOT is used for acknowledging to the transmitting node that at least one other node has received the sent frame. The ACK DELIMITER is one bit long and is ‘recessive’.

- **END OF FRAME (EOF)**
  This bit field consists of seven ‘recessive’ bits and marks the end of a frame.

The parts of the CAN data frame that this project used were the ARBITRATION FIELD, CONTROL FIELD and DATA FIELD.

### 3.2.1 Error control in CAN bus

CAN nodes check all messages for errors, even if the message is not meant for that particular node. If a CAN node detects an error it sends an ERROR FRAME to all other CAN nodes. CAN employs five different methods of error checking:

- **Bit error**
  When a node sends a bit on the bus it checks the bus to see that the checked bit value is the same as the sent bit value. If they are different a Bit error has occurred.

- **Bit stuffing error**
  When a message is received and 6 consecutive bits of equal value are detected a Bit stuffing error has occurred. Bit stuffing always inserts a bit of opposite value whenever 5 bits of equal value get sent so if 6 bits of equal value are encountered an error has occurred.
• CRC error
  If the CRC calculation of the receiver differs from the CRC value in the CAN message that was calculated by the transmitter a CRC error has occurred.

• Form error
  A Form error occurs whenever a predefined part of a CAN message has a different size than expected.

• Acknowledgment error
  An acknowledgment error occurs whenever a transmitter does not detect a dominant bit in the ACK SLOT after it has sent a message.

[19, 20, 23][23]

3.3 CANopen
The first step of analyzing data was to interpret what kind of message each data frame was. Since RCS is built upon the CANopen protocol the CANopen specification was consulted. CANopen is a higher layer protocol that is based on CAN. It provides abilities for devices from different manufacturers to communicate with each other. All devices connected to a CANopen network need to implement a device profile. This device profile guarantees that all devices connected to the same CANopen network can communicate and understand each other. The device profile contains an object dictionary. The object dictionary contains data, communication objects and commands. A lot of different profiles exist with different object dictionaries depending on the type of device the profile is intended for. 4 different types of communication are available for use in CANopen:

• Service Data Object (SDO) communication
  Used for configuration of nodes in the CANopen network. Has the ability to access device object dictionaries for both reading and writing.

• Process Data Object (PDO) communication
  PDO is the main option for transferring data between nodes. PDO communication can only occur once all devices on the network are up and running. There are three different modes of PDO communication:
    o Synchronous communication, which transmits and receives data on every sync message received
    o Event communication, which responds to a certain event, for instance a button being pressed, and sends an event message telling other nodes that the button was pressed
    o Polling, where the device responds to a CAN remote frame by sending a PDO containing the requested data

• Network Management (NMT) communication
  Used for network operation and management. A node on the CANopen network can be designated as the network manager. This node tells other nodes what state of operation they should be in. There are three modes of operation: initialization, preoperational and operational. Initialization occurs when the node is starting up, preoperational occurs when the node has completed the boot sequence or when the network manager tells the node to go into preoperational mode and operational which
occurs when the node receives a message from the network manager telling it to become operational. In the preoperational state only SDO messages can be received, these messages usually contain information on how the node should be configured. In the operational state both SDO and PDO messages can be sent and received.

- Predefined format message communication

These messages are used for error reporting, timing, and synchronization, for example the emergency telegram which is sent whenever an error occurs on a node.

CANopen splits the CAN identifier into two parts, CFC (Can Function Code) and CANopen node ID. The CFC is 4 bits long and the CANopen node ID is 7 bits long. A lot of information can be gained from just these two parts of the CAN message. Fig. 2 shows the information one is able to get from CAN ID. [25]

<table>
<thead>
<tr>
<th>service</th>
<th>default receive ID allocation</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>network management</td>
<td>0H (0)</td>
<td>1</td>
</tr>
<tr>
<td>node guarding</td>
<td>700H+node-ID (1792+node-ID)</td>
<td>2</td>
</tr>
<tr>
<td>SDO request</td>
<td>600H+node-ID (1536+node-ID)</td>
<td>3</td>
</tr>
<tr>
<td>receive PDO 1</td>
<td>200H+node-ID (512+node-ID)</td>
<td>4</td>
</tr>
<tr>
<td>receive PDO 2</td>
<td>300H+node-ID (768+node-ID)</td>
<td>5</td>
</tr>
<tr>
<td>sync telegram</td>
<td>80H (128)</td>
<td>6</td>
</tr>
<tr>
<td>DST services</td>
<td>7EBH (2024)</td>
<td>7</td>
</tr>
<tr>
<td>NMT services</td>
<td>7EAH (2028)</td>
<td>8</td>
</tr>
<tr>
<td>transmit PDO 1</td>
<td>180H+node-ID (384+node-ID)</td>
<td>9</td>
</tr>
<tr>
<td>transmit PDO 2</td>
<td>280H+node-ID (640+node-ID)</td>
<td>10</td>
</tr>
<tr>
<td>SDO response</td>
<td>500H+node-ID (1412+node-ID)</td>
<td>11</td>
</tr>
<tr>
<td>emergency telegram</td>
<td>80H+node-ID (1280+node-ID)</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 2: The information in a CAN ID

3.4 Rig Control System

Rig Control System (RCS) is the system in use by many of Atlas Copco’s machines. It is built upon CANopen and an RCS network consists of several hardware modules. The modules communicate with each other over CAN but can also use Ethernet for more data heavy tasks. An RCS network usually consists of a central network and one or more local networks. There are two main types of modules, application modules and peripheral modules. Application modules consist of:

- APP module
  The main part of the RCS network that handles all calculations.

- DISP module
  The DISP module is a display that shows relevant data to the operator of the machine.

- CCI module
  The CCI module is used for external communication.

Peripheral modules consist of:

- I/O module
  An I/O module consists of many digital and analog inputs and outputs that can drive relays or read signals from actuators.
• Decoder module
  Decoder modules are usually used in operator panels for reading buttons and joysticks.

• Resolver module
  A resolver module is used for sensors that can measure angles, for example the angle of a boom on a boomer.

3.5 Graphical User Interface

3.5.1 Choice of framework for the Graphical User Interface (GUI)
When it was time to choose what framework to use for the construction of the user interface, a list was made containing some of all the possible candidates. To choose impartially, pros and cons of each possible framework were weighed against each other. The requirements for the frameworks were the following:

  • Must work with Visual Studio
  • Useful documentation
  • A stable release, meaning no beta or newly released version which might contain bugs and other errors

The wishes were the following:
  • Cross-platform compatibility
  • Easy to work with
  • A good community
  • Able to program in C++

3.5.1.1 Microsoft Foundation Class (MFC)
The Microsoft Foundation Class Library (MFC) is a library and a wrapper for the Win32 API. The preferred language is C++ when developing with MFC. Visual Studio has built-in support for MFC. [4]

3.5.1.2 wxWidgets
wxWidgets is a cross-platform C++ library that supports Windows, Linux, UNIX and Mac OS X. It also has support for mobile platforms. It is free and open-source. C++, Python, Perl and C# are all supported languages for use with wxWidgets. There exists some support for Visual Studio but other Integrated Development Environments (IDE) has better built-in support for wxWidgets. [5]

3.5.1.3 Windows Presentation Foundation (WPF)
Windows Presentation Foundation is a part of the .NET framework since version 3.0. The only supported operating system is Windows. It is used for creating user interfaces for applications that run on Windows. WPF has support for the C# and Visual Basic programming languages. Visual Studio has built-in support for WPF. [6]

3.5.1.4 Windows Forms
Windows Forms is a graphical API. It is a part of the .NET framework. It is a wrapper for the Win32 API. It has support for the C#, Visual Basic and C++/CLI programming languages. Visual Studio has built-in support for Windows Forms. [7]

3.5.1.5 Qt
Qt is a cross-platform C++ library that supports Windows, Linux, UNIX and Mac OS X. It also has support for some mobile platforms such as Symbian. It has a commercial version and
an open-source version. Qt is primarily made for developing with C++. An add-in exists for Visual Studio support as well as Qt’s own IDE Qt Creator. [8]

3.5.1.6 Windows Template Library (WTL)
Windows Template Library is a C++ library that only supports Windows. Its sole focus is on the graphical user interface. An add-in exists for Visual Studio to help with development in WTL.[9, 10]

3.5.1.7 Analysis

<table>
<thead>
<tr>
<th>Requirements / Wishes</th>
<th>MFC</th>
<th>wxWidgets</th>
<th>WPF</th>
<th>Windows Forms</th>
<th>Qt</th>
<th>WTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Useful documentation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Stable release</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cross-platform compatibility</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Easy to work with</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Good community</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>C++ support</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The support for Visual Studio was good for all the chosen frameworks, either directly or through an add-in. The documentation for WTL was, at the time of this writing, incomplete and was missing features added in the recent releases of WTL. The other choices had excellent documentation that was easy to understand and work with. All the chosen frameworks had a stable release to work with. MFC had no form of cross-platform compatibility, it was only for Windows, this was the same for WPF, Windows Forms and WTL, only wxWidgets and Qt fulfilled the wish of cross-platform compatibility. The wish of the framework being easy to work with was not rated since it was hard to know if something was easy to work with without at least trying it out. The same could be said about the wish for a good community around the framework but a quick analysis showed that all choices seemed to have a good community around them. C++ was supported by all frameworks except for WPF and Windows Forms (C++/CLI is supported by WPF and Windows Forms but it is not native C++).

After this analysis the decision was made that Qt 4.8.4 would be used. The reasons for this choice were the following:
- Good documentation
- A big community around Qt which can provide support if needed
- Atlas Copco already uses Qt in their RCS system
- Tools for aiding the implementation of the GUI
- Cross-platform support

The GUI was implemented by using Qt 4.8.4 and Qt Creator 2.7.0.

3.5.2 The bus view
One of the requirements of the project was to create a view where all the CAN messages sent on the RCS network could be seen so a bus view was created for this purpose. This view would hold all the messages that had been sent on the RCS network. It consisted of a table
with 9 different columns. The columns showed relevant data for each CAN message, for example, the time when the message was received, what CAN ID the message had and the data that the message contained. To be able to see and understand the data better, a toggle between hexadecimal and decimal were implemented on the columns CAN ID and Data in the table view. A toggle between absolute and relative time was also added. The color checkbox indicates if sync messages should be represented with a yellow background for clarity since the sync messages can be important to check if all nodes respond to sync within a certain time frame. To always see the newest message an auto scroll feature was added. This function made sure that the bus view scrolled to the last received message whenever an update to the bus view occurred.

3.5.2.1 Updating the bus view
The large amounts of data that the program received put a strain on the computer’s CPU, especially when filtering was involved, to the point that the program became unresponsive and locked up. After some research it was found that the filtering plus constant drawing of new data on the GUI was too much for the program to handle, it could not keep up. Not so surprising when the program receives 800-900 messages per second with a bus load of only 20%. And the program had to update the bus view every time a new message was received. To alleviate this problem a timer was implemented with a default value of 250 milliseconds that updated the bus view whenever 250 milliseconds had passed and new data had been received. This stopped the program from becoming unresponsive and eased the strain on the CPU of the computer. A textbox was added where a user could change the update frequency from 1-1000 milliseconds, all depending on how fast the user needed to see the data and how powerful the computer was.

3.5.3 Analysis of data
When a row was selected, information of the specific message was shown in an information box to the right of the table view. See Fig. 3. When a new message was selected the information box was cleared from the previous information and the information of the new message was shown. More information was displayed if a message from an I/O module was chosen. This was accomplished with the use of the I/O module class which analyzed the message more thoroughly.
3.5.4 **Filter data**

To filter the data a proxy model was used that was placed between the data model and the GUI. To get it working with the massive amount of data sent via the CAN bus a timer had to be implemented. The proxy model gave the possibility to filter the data on a specific column value. A combo box was added to choose how the program should filter, the choices were regular expressions, wildcard or fixed string. It was possible to filter on Can Id, Cfc and Module Id. The filter function is shown in Fig. 4.

![Filter function](image)

Figure 4: The filter function

3.5.5 **Module view**

To be able to see the modules connected to a CAN bus network a module view was added. This view showed the modules and the ability to expand a specific module to show more information about it. This view was located to the right on the screen, next to the bus view. This is shown in Fig. 5. When a specific module has sent a message that module was added in the module view if it did not already exist. The modules were sorted by module number, that way the modules always came in the correct numerical order. The module view got updated as soon as a new message that either got sent or was received was parsed by the client. The data for this view was fetched from the I/O module class.
3.5.6 Calculation of bus load

A calculation of the current bus load of the can bus was needed for the simulation. When connecting through the Kvaser hardware the bus load and other information was received from it through a simple command. This information did not exist when connecting to the simulation. It was therefore needed to calculate this in the client. The formula used for calculating the bus load was:

\[
\text{bus load} = \frac{\text{bits received}}{\text{bus speed}}
\]

Bits received were calculated by the following formula:

\[
\text{bits received} = \text{messages received} \times (\text{size of CAN message excluding data field}) + \text{data bytes received} \times 8 + \text{bit stuffing}
\]

The size and structure of a CAN message can be seen in Fig. 6.

![Size and structure of a CAN message](image)

Figure 6: Size and structure of a CAN message [22]

Bit stuffing had to be estimated since access to the complete CAN message as sent on the CAN bus was not possible in the client software. The maximum number of stuff bits a CAN message can contain is [21]
Since this was the worst case estimation the used formula in the client divided this by 2 to get a more average estimation. The estimation was calculated using the following formula:

\[
\text{bit stuffing} = \frac{8 + 2 \times \text{data bytes received}}{2}
\]

A timer that fired every second was used to trigger the calculation of the bus load. When comparing the calculated bus load to the bus load received from the Kvaser hardware the difference was usually below 1%, in rare cases above 2%. This was deemed as good enough.

3.6 Inter-Process Communication (IPC)

When it was time to choose how the communication between the simulation and our application should be created, a list was made containing some of all the possible candidates. To choose impartially, pros and cons of each possible communication were weighed against each other. The requirements for the inter-process communication were:

- Cross-platform compatibility
- Good scalability
  It should not matter how many messages are received within a certain time frame, the messages must still be delivered.
- Latency
  The messages need to be received by the client software within the order of milliseconds from when the server sent the message in order to make sure that the data presented to the user is up-to-date.
- Existing implementation for C++

3.6.1 D-Bus

D-Bus is a protocol developed for enabling communication between processes. It is built as a message bus system with a publish-subscribe model. This means that a process can publish a message on a message bus and several other processes can subscribe to it and receive the message. It also has one-to-one direct communication through Remote Procedure Calls (RPC). This mode is simpler and faster since it skips the whole message bus system. D-Bus is mostly used on Linux but implementations exist for almost all other platforms including Windows. Bindings exist for many frameworks and languages, including Qt and C++. [11, 12]

3.6.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) is a network protocol in wide use today. Almost all operating systems on the market today support TCP. TCP is mainly used for network communication but can also be used for inter-process communication. TCP is a reliable protocol that makes sure that every message sent is received, that the message hasn’t been corrupted in transit and that messages arrive in the order they were sent. [13]

3.6.3 Relational Database Management System

A Relational Database Management System (RDBMS) is a database that stores data in tables that are made up of columns and rows. A RDBMS can be used for inter-process
communication if it has some way of notifying clients when new data has arrived. Most RDBMS implementations have built-in support for this. [14]

3.6.4 **Shared memory**

Shared memory is a method of inter-process communication that is supported by Windows, Linux and other operating systems. The shared memory is a physical or virtual segment of memory that one process has allocated. Other processes can map to the same segment of memory thereby enabling inter-process communication by writing and reading to the same memory segment. [15]

3.6.5 **Named pipe**

A named pipe, or FIFO (First In First Out) is a method to communicate between processes that exists in all the major operating systems on the market today. A named pipe is an extension of a normal pipe which is a redirection of data from for example a file to the console. A named pipe can handle both one-way and two-way communication. A named pipe-server creates a named pipe and a named pipe-client connects to it. Named pipes are mostly used for inter-process communication on the same computer but can also be used for inter-process communication between computers connected through a network. [16, 17]

3.6.6 **Analysis**

All of the listed options were viable, some more than others.

**D-Bus**
- Good support on many operating systems
- Qt has built-in support for it
- Light-weight
- Easy to implement on RCS

**TCP**
- Good support on many operating systems
- Qt has built-in support for it
- RCS has built-in support for it
- Ability to easily connect over network
- Good scalability

**RDBMS**
- Good scalability
- Provides easy way to save the data
- Depending on which implementation, the support for different platforms differ
- Time-consuming to implement in RCS

**Shared memory**
- Many platforms support it but the implementations differ
- Poor scalability [15]

**Named pipe**
- Many platforms support it but the implementations differ
- Easy to implement on RCS
In the end TCP was chosen as the means for facilitating the inter-process communications. The reasons for choosing TCP were the following:

- Cross-platform compatibility
  A lot of other choices are dependent on the operating system whereas TCP is the same across all platforms
- Ability to connect to the simulator from another computer
  This is possible with TCP as long as a network connection exists between the two computers
- RCS already has support for TCP
- Proven technology

A TCP server was implemented on the data collector and a TCP client was implemented on the client side.

### 3.7 Retrieval of data from RCS Simulation

To connect to the simulated CAN bus and get the data needed existing code from Atlas Copco’s base system implementation was used. A new program was created which used an existing example from Atlas Copco as a reference. The new program was written using Atlas Copco’s development libraries which meant that the program would be able to be executed directly on the RCS system. Once the connection was established to the CAN bus the next step was to retrieve data from it. This was accomplished by using a threaded process that read the data from the CAN bus and put it into a queue where it could easily be used by other parts of the program. To start with, the data was collected and written to a log file which was compared to the log file created by the example program to make sure that the received data was complete and no messages were getting lost. This part of the program was then extended with a TCP server that waited for a connection from the client which was part of the GUI implementation. When a connection was established the server began retrieving the messages from the queue and started sending them to the client. The client received the messages and added them to the data model. The path of a CAN message from the RCS simulation to the GUI is illustrated in Fig. 7.

![Figure 7: Path of a CAN message from the simulation](image)

### 3.8 Retrieval of data from Kvaser USBcan Rugged HS/HS

Kvaser had documentation on how collecting data from a Kvaser product should be done in the form of an API. With that API, a simple test program that collected data from the RCS CAN network was created. The program connected to the CAN network through the Kvaser USBcan Rugged HS/HS which in turn was connected to a laboratory RCS machine. The data that the program collected was written to a console. Parts of the code from the test program were implemented into the main program which instead of just writing the data to the console retrieved the data and added it to the data model. The path of a CAN message from the machine to the GUI is illustrated in Fig. 8. [3]
3.9 Retrieval of data from log file

A specific parser was written per log file type. The parser could handle both Kvaser and CanSniffer log file formats. This parser went through the file and parsed it into the data model. The data from the data model was sent to the GUI, which wrote the data in descending order on the screen, shown in Fig. 9. By reading the first line of the file, the program could establish if the file was valid or not. If a file of the wrong type was chosen, an error message was shown and the parsing of the file was stopped.

3.10 Saving data to a log file

When saving all the data on a log file the format was decided to be the same as the log files from Kvaser. This way the parser for the Kvaser log file type could be used for parsing the RCS CAN Tool log file type. The log files started with a header with information of what the file contained and information of when the file was saved. The data was saved in a comma-separated value format. Every message had a separate row; the time was saved in number of seconds and the data in decimal format.

3.11 I/O Module

An I/O module is the physical interface to actuators and sensors on an Atlas Copco machine. An I/O module consists of digital inputs, analog inputs, current controlled PWM (Pulse Width Modulation) outputs, digital outputs and an encoder interface. The I/O module was one of
three modules that the program should be able to analyze more thoroughly than the other data being sent on the CAN bus network.

### 3.12 I/O Module Class

To be able to parse and analyze data that belonged to I/O modules on the CAN bus network an I/O module class was created. This class contained functions for parsing and analyzing CAN data frames that were either sent or received by an I/O module. It also stored all the messages that were sent or received by a certain I/O module. This class was developed with the use of the I/O module specification from Atlas Copco which contained the necessary information needed for being able to create an I/O module class that could be seen as a software representation of the hardware. If the program was connected to the machine when it started and was able to pick up the initialization messages from RCS to the I/O module the class also contained information about how a specific I/O module was configured. This information was only able to be retrieved at the start of a machine so if the program was connected to an already running machine the module configuration could not be read. The difference this made in the program was that if the initialization was read then the program had the information to know what ports were enabled on the I/O module and in what mode they were in. Otherwise the program would just have to show all ports and could not know in what mode the ports were in. There was no way around this problem so if the configuration data was needed a restart of the machine was required.

### 3.13 Decoder/Resolver classes

When a couple of weeks of the project time remained a meeting was held and after a discussion with the supervisor it was decided that the decoder and resolver classes should be excluded. The main reason for this decision was that the I/O module should be prioritized and be finished before working on decoder and resolver. The time was not enough to create all classes and the I/O module class was the most needed one, so focus was shifted to creating just one class and creating it in such a way that it could be used as a template for the other classes. The I/O module class was the most complicated one of the three which would make it an ideal template for the other less complicated modules.
4 Result

The results of the project are discussed in this chapter, starting with performance of the tool that was developed for the project. The usage of the tool is also described in detail.

4.1 Performance

In the beginning one major concern was that the program had to be able to collect a large amount of data in a short amount of time and display this data to the user in a timely manner. The machine the program was connected to was sending about 800-900 messages per second at 20% bus load and other machines at Atlas Copco had a much higher utilization of the CAN bus which meant a lot more data to handle per second. The program received all the messages without any problem at all when connected to real hardware. There was no problem receiving messages through TCP from the simulation either and the time frame from when the message was sent from the server to when the client received it was so small that it was not noticeable so the scalability and latency when using TCP was more than adequate. The more difficult task for the program was to write out all the data on the GUI, this had to be done by a timer to make sure that the program had the time needed to update the GUI and show the data to the user. When filtering was added on top of this the program could become quite slow when a lot of data had been gathered and the filter parameters were changed. In the end though, the performance was adequate for normal usage of the program.

4.2 Testing

To make sure that the program functioned correctly several tests were carried out. The first test was with the simulation and a program that wrote all CAN messages to a file was used at the same time as the server sent CAN messages to the client. This test involved several tests with different elapsed time from 1 minute to 1 hour. The client’s data was then compared to the file output and these matched completely on all tests from the 1 minute test to the 1 hour test. The same kind of test was carried out with the Kvaser hardware connected to a machine but instead of having a log file to compare with a program from Kvaser called CANKing was used [29]. This program had the ability to connect to the Kvaser hardware. Both programs were connected at the same time to the Kvaser hardware and the output from both programs was compared. The output was exactly the same for both the project software and the CANKing software. The next test that was carried out was to make sure that the analyzed data was correct, for example if port X20 has a value of ten then the program should show a value of ten. This test used both the simulated environment and a real machine. A port was set to a predetermined value in the simulation and the client was then checked to see if the analyzed data showed the same value. All ports on the I/O module were tested and all the values matched. The same test was carried out with real hardware and the result was the same. All the tests that were carried out showed that the project software showed all data that was sent over the CAN network and that it analyzed this data correctly.

4.3 Usage

In the end the program looks like in Fig. 10, we moved the information box to the bottom left corner to make room for our module view. Instead as having a single window for both views we decided that we would have both views in the same window. This made the program more informative which is the most important thing on a program of this type.
To connect to the RCS Simulator the user presses the “Simulation” button. Then a new dialog box is shown to the user in which the user enters the address and port to connect to. By pressing the “OK” button with the right values in address and port the program tries to connect to the simulator, as shown in Fig. 11.

To connect to Kvaser hardware the user presses the “Kvaser” button. Then a new dialog box is shown to the user, in which the user chooses the Kvaser hardware and channel to connect to, as shown in Fig. 12. When this is done the program tries to connect to the chosen Kvaser hardware. If no hardware is connected to the computer only the virtual Kvaser channels are shown in the dialog box.
To open a log file the user presses “File”, “Open Log File” and what type of log file to open, as shown in Fig. 13. Then a dialog box is shown to the user where the user chooses what file to open.
5 Discussion
In this chapter the achievements of the project are discussed, if all requirements have been met and if all goals of the course objectives have been fulfilled. The future of the project and what can be improved upon is also discussed.

5.1 Social aspects
The social aspects that this project touches upon are mostly related to mining and rock excavation since that is the main focus of the Rig Control System that Rocktec, the department in Atlas Copco where the project was located, develops.

5.1.1 Economic aspects
The Swedish mining industry generates 1% of the Swedish GDP (Gross Domestic Product), 4% if associated industries, such as Atlas Copco, are taken into the calculation [27]. The mining industry in Sweden employs around 8000 persons directly and around 30000 indirectly [28]. Atlas Copco has around 1900 employees in Sweden [1]. All in all, the mining industry is an important part of the Swedish economy and this project will hopefully in some small part contribute to this part of the economy.

5.1.2 Safety aspects
Many dangers are present when working in a mine, including, but not limited to, explosions, loose rocks falling, fires and equipment malfunction. By developing autonomous systems that can perform the dangerous tasks in a mine without having a human nearby the dangers to human lives are minimized. Many companies have developed systems for automating tasks in mines and Atlas Copco has developed a system for localization and routing in mines. This system can guide the Load-Haul-Dump (LHD) vehicle around the mine. The LHD vehicle is first driven along the pre-determined route where data about the route is collected. It can then automatically drive this route without the need for human control. There is also a tele-remote system that lets the LHD be remote-controlled by an operator with the help of cameras mounted on the vehicle. The operator does not need to be in close proximity to the vehicle which minimizes the dangers to the operator and since the operator usually does not need to drive the vehicle and only assist it when it is loading or dumping the operator can be in charge of more than one LHD vehicle at once, raising productivity. Both these types of automation leads to the fact that fewer people need to be in the mines performing dangerous tasks and this will hopefully lead to fewer fatalities and injuries and to increased productivity. [30, 31]

5.2 Achievement of the course objectives
We believe we have achieved most of the goals of the course. The goals were broken up into three main parts and each one is presented with a reasoning of why we believe that specific part of the course objectives has been achieved.

5.2.1 Knowledge and comprehension
We learned a lot about CAN and CANopen over the course of the project. To be able to use Qt as a framework for the creation of the GUI we had to study the framework and test a lot of different ideas to see what would work for this project. It was time consuming but in the end it was worth it. To use tools you have never used before pushes yourself to learn new things and develop as a person.
5.2.2 Proficiency and ability
We spent quite a lot of time searching for academic sources to use as a foundation for our work. We used academic sources for most of our decisions when such sources were available. It was hard to find good academic sources for some parts of the project; in these cases we used sources that felt credible without being peer-reviewed, for example the documentation for the Qt project. We planned the project in the beginning and we did achieve all the milestones as the time went by. Only when the decision was made that we would prioritize differently as mentioned before the previous plan fell apart. We had this in mind but according to us the most important thing was to create the program in a way that benefited Atlas Copco the most. We weighed that higher than holding our planning in the smallest detail. This also meant that some of the requirements of the project were missed.

5.2.3 Values and attitude
We have got a new and clearer insight of how a company works, how to work in groups and how to plan our work. We wrote a blog where we listed everything we did and the time it took, for us this was a good way to get an overview of our project. We have had weekly meetings with our supervisor Jan Knutsson and other employees at Atlas Copco where we discussed how far the project had come and in what direction it should continue. In some of the meetings we had a prototype to show to get feedback, and in others we just had an oral discussion. The source code for the tool has been well documented to make it easier to develop further and the GUI has been made easy to understand and work with to make sure that developers and service engineers can use it as intended.

5.3 Compliance with the project requirements
We have managed to fulfill the prioritized requirements for the project. We have created a test application using the Application Programming Interface (API) for the USB CAN family of products from Kvaser.

We have created a class for the Input/Output (I/O) module used in RCS. This class has been created in such a way that it is relatively easy to implement other RCS modules. This class is used in the translation of CAN messages into RCS I/O tables and signal lists. This translation can be seen in the module view of the program.

We have created a GUI with the following parts:

- Bus View
- Module view
- Bus statistics

These three parts of the GUI are all visible as soon as the program is started which gives the user a good representation of the data available without having to use buttons or menus to switch between different views. CAN messages can be viewed in real-time in the bus view. If the message is sent from or received by an I/O module, more information can be gained by selecting the message in the bus view. This extra information is then shown to the user in the information box below the bus view. We have created a way to select and view data from a node on the CAN bus, both inputs and outputs, by filtering the bus view on the module number for the particular node. It is also possible to filter on a specific message or specific CAN Function Code.

Signals to and from the I/O module can be monitored in the module view and are updated in real-time. They are shown as both analog/digital and physical values.
It is possible to monitor two CAN-busses in the same view by connecting two channels on a Kvaser USBcan interface to different CAN-busses. There is no limit in the software to the amount of CAN-busses that can be monitored.

The non-prioritized requirements that did not get complete were:

- Create classes for each of the following module type:
  - Decoder
  - Resolver
- Are all requested data delivered within time?

As mentioned earlier we did not create these missing classes because it was decided that we would prioritize the I/O module class to make it work perfectly and to be able to use it as a template for the other module types instead of just rushing it and ending up with all three classes that were of subpar quality.

We think that we could have planned the project better, that way we might have had the time to achieve all of the projects requirements. Another thing we could have done better was to set the requirements in a more realistic way, for example have four hard requirements and have the rest as soft requirements. By setting all of the requirements as hard ones, we had no room for error.

The choice of framework for the GUI may have contributed to the fact that we did not fulfill all the requirements. We had not worked with Qt before so a lot of time went into learning how it worked. If we had chosen a framework that we were more familiar with, for example .NET, we would have had more time to focus on the project at hand. On the other hand, Qt is a great framework, it is easy to work with once you get past the first steps of learning it and it still feels like the right choice for this project.

5.4 Project development

In the future the RCS CAN Tool program can be modified to be even better. Some future hopes that have been discussed with our supervisor are:

- To be able to see the initiation in a new window, this to rule out wrong initiation in case of error.
- To see if all data is received in time. To be able to choose time yourself.
- To be able to save configuration on file, this because you can have different configurations on different machines.
- Better filtering function, ability to filter on more than one CAN ID at a time.
- To add functionality for CAN bus according to standard SAE J1939
- To be able to send messages. To be able to choose order of the messages to send.

If these things are implemented the program will be more complete and therefore a better product. If this is done the program will start to become more and more useful for the employees at Atlas Copco and finally every developer will be able to have RCS CAN Tool on its computer if wanted.
## 5.5 Figure designation

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Kvaser USBcan Rugged HS/HS</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>The information in a CAN ID</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Early prototype analyzing data in information box</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>The filter function</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>The different views in an early prototype</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Size and structure of a CAN message</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Path of a CAN message from the simulation</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Path of a CAN message from the machine</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Log File represented in the GUI</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>The program in its final stage</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>The connection to the simulator</td>
<td>24</td>
</tr>
<tr>
<td>12</td>
<td>The connection to the Kvaser hardware</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>Opening a log file</td>
<td>25</td>
</tr>
</tbody>
</table>
6 References

Fetched 2013-06-10
URL: http://www.atlascopco.se

[2] USB CAN family products of Kvaser
Fetched 2013-04-24
URL: http://www.kvaser.com/sv/products/can/usb.html

Fetched 2013-04-30
URL: http://www.kvaser.com/canlib-webhelp/index.html

[4] MFC Desktop Applications
Fetched 2013-04-27
URL: http://msdn.microsoft.com/library/d06h2x6e.aspx

Fetched 2013-04-27
URL: http://www.wxwidgets.org/

[6] Introducing Windows Presentation Foundation
Fetched 2013-04-27

Fetched 2013-04-27

[8] Qt Home Page
Fetched 2013-04-27
URL: http://qt-project.org

Fetched 2013-04-27
URL: http://sourceforge.net/projects/wtl/

URL: http://ieeexplore.ieee.org.db.ub.oru.se/stamp/stamp.jsp?tp=&arnumber=6164633&isnumber=6164486

Fetched 2013-05-02
URL: http://www.freedesktop.org/wiki/Software/dbus/
URL: http://ieeexplore.ieee.org.db.ub.oru.se/stamp/stamp.jsp?tp=&arnumber=5680396&isnumber=5679517

Fetched 2013-05-02
URL: http://www.ietf.org/rfc/rfc793.txt

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1593204&isnumber=33506

URL: http://www.aicit.org/JDCTA/ppl/JDCTA1841PPL.pdf

[16] Pipes: A Brief Introduction
Fetched 2013-05-02
URL: http://www.linfo.org/pipe.html

[17] Named Pipes
Fetched 2013-05-02
URL: http://msdn.microsoft.com/library/windows/desktop/aa365590%28v=vs.85%29.aspx

URL: http://maxwellsci.com/print/rjaset/v5-4279-4283.pdf

URL: http://ieeexplore.ieee.org.db.ub.oru.se/stamp/stamp.jsp?tp=&arnumber=5116731

URL: http://ieeexplore.ieee.org.db.ub.oru.se/stamp/stamp.jsp?tp=&arnumber=788104&isnumber=17068
URL: http://ieeexplore.ieee.org.db.ub.oru.se/stamp/stamp.jsp?tp=&arnumber=6489559&isnumber=6489522

Controller Area Network (CAN) Implementation Guide
Fetched 2013-05-15

Fetched 2013-04-15
URL: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf


CANopen Application Layer and Communication Profile DS-301, v4.01, CAN in Automation e.V., June 2000.
Fetched 2013-04-15
URL: http://www.can-cia.org/index.php?id=specifications

Qt Reference Documentation
Fetched 2013-04-15
URL: http://qt-project.org/doc/qt-4.8/

Inspel till en svensk mineralstrategi 2012
Fetched 2013-06-17
URL: http://www.regeringen.se/content/1/c6/19/82/82/bc6605df.pdf

Så ser Gruvsverige ut i dag
Fetched 2013-06-17

Kvaser home page
Fetched 2013-06-24
URL: http://www.kvaser.com/

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:oru:diva-22264

Scooptram Automation: Underground loader
Fetched 2013-06-27
URL: http://www.atlascopco.us/usus/products/product.aspx?id=1515262
Use case scenarios

Start RCS CAN Tool
Prerequisites: RCS CAN Tool is installed and working on a computer
Exit: RCS CAN Tool has been started
Actors: User, RCS CAN Tool, PC

1. Connect to the machine or simulator you want to run the program with
2. User starts the computer
3. User starts the program by double click the RCS CAN Tool symbol
4. The main view is shown

Choose data source
Prerequisites: RCS CAN Tool has been started
Exit: A data source has been chosen and is shown to the user
Actors: User, RCS CAN Tool, PC

1. User clicks the data source button
2. User is presented with a selection of data sources, example: Kvaser, Log file or Simulation
3. User chooses a data source
4. User presses the OK button
5. Data from the chosen data source is shown to the user

Select and view data from a node
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User gets a view with data from a specific node
Actors: User, RCS CAN Tool, PC

1. User clicks on filters
2. Choose node in the dropbox to the left
3. Choose to the right which node you want to monitoring
4. Click OK
5. The program is an passive node that is just listening to all the messages
6. Data from the specific node is shown

Select and view data from specific message/sensor
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User gets a view with data from a specific message/sensor
Actors: User, RCS CAN Tool, PC

1. User clicks on filters
2. Choose message or sensor in the dropbox to the left
3. Choose to the right which message/sensor you want to monitoring
4. Click OK
5. Data from the specific message/sensor is shown
Select and view database signals
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User gets a view with data from a specific database signal
Actors: User, RCS CAN Tool, PC

1. User clicks on filters
2. Choose database signals in the dropbox to the left
3. Choose to the right which database signal you want to monitoring
4. Click OK
5. The program becomes an active node and sends messages to the database that answers
6. A list of CAN-messages is shown

Evaluation of CAN data
Prerequisites: RCS CAN Tool is monitoring a data source
Exit: User sees a closer look at the CAN data
Actors: User, RCS CAN Tool, PC

1. User marks a row on the data view
2. User sees evaluation of the chosen row under the data view

Bus load %
Prerequisites: RCS CAN Tool is monitoring a data source
Exit: User can see how much the bus load in percent (%) is
Actors: User, RCS CAN Tool, PC

- The bus load in percent is shown to the user on the left side of the screen

Error frames
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User gets a list of Error frames
Actors: User, RCS CAN Tool, PC

1. User clicks on filters
2. Choose Error frames in the dropbox to the left
3. Choose to the right which Error frame you want to monitoring
4. Click OK
5. A list of Error frames is shown

Select and view CAN messages
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User gets a list of CAN messages
Actors: User, RCS CAN Tool, PC

1. User clicks on filters
2. Choose CAN-message in the dropbox to the left
3. Choose to the right which CAN-message you want to monitoring
4. Click OK
5. A list of CAN-messages is shown
**Log data to file**
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: Data has been logged to file
Actors: User, RCS CAN Tool, PC

1. User clicks on log to file
2. The data that is shown on the screen is logged to a file. This is done by reading the vectors that are shown to the user and writes them down to a brand new file

**Timing/sync**
Prerequisites: RCS CAN Tool is monitoring a data source and there is data on the screen
Exit: User sees if the message was sent and received in time
Actors: User, RCS CAN Tool, PC

1. User clicks to change view
2. User choose the timing/sync view
3. The timing and synchronizing is shown to the screen