Czech Technical University in Prague
Faculty of Electrical Engineering

DIPLOMA THESES

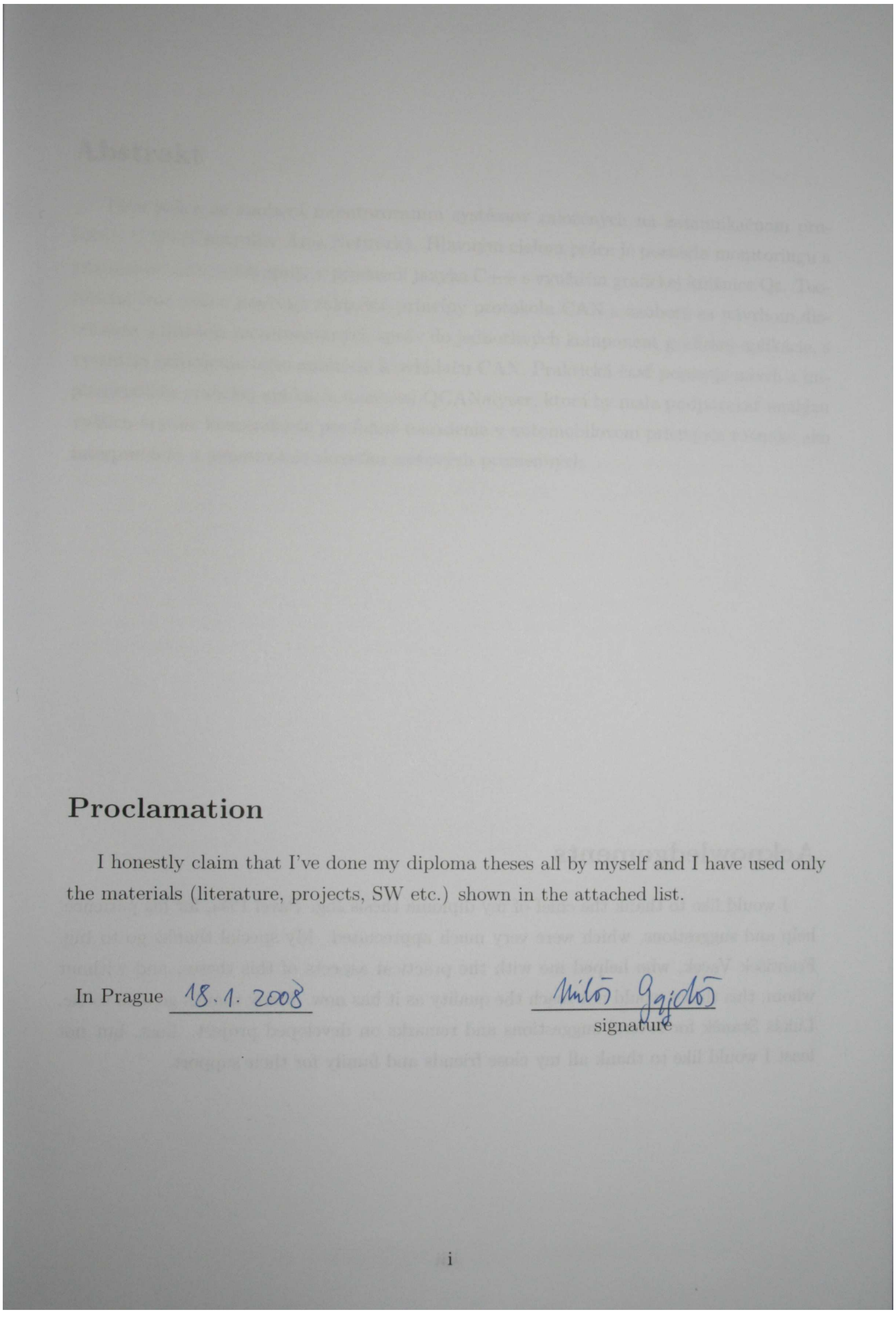
CAN bus communication protocol support and monitoring

Prague, 2008

Miloš Gajdoš

Acknowledgements

I would like to thank the chief of my diploma thesis Ing. Pavel Píša, for his patience, help and suggestions, which were very much appreciated. My special thanks go to Ing. František Vacek, who helped me with the practical aspects of this theses, and without whom, this theses would not reach the quality as it has now. Many thanks go also to Bc. Lukáš Staněk for several suggestions and remarks on developed project. Last, but not least I would like to thank all my close friends and family for their support.



Proclamation

I honestly claim that I've done my diploma theses all by myself and I have used only the materials (literature, projects, SW etc.) shown in the attached list.

In Prague 18. 1. 2008 [illegible]

signature

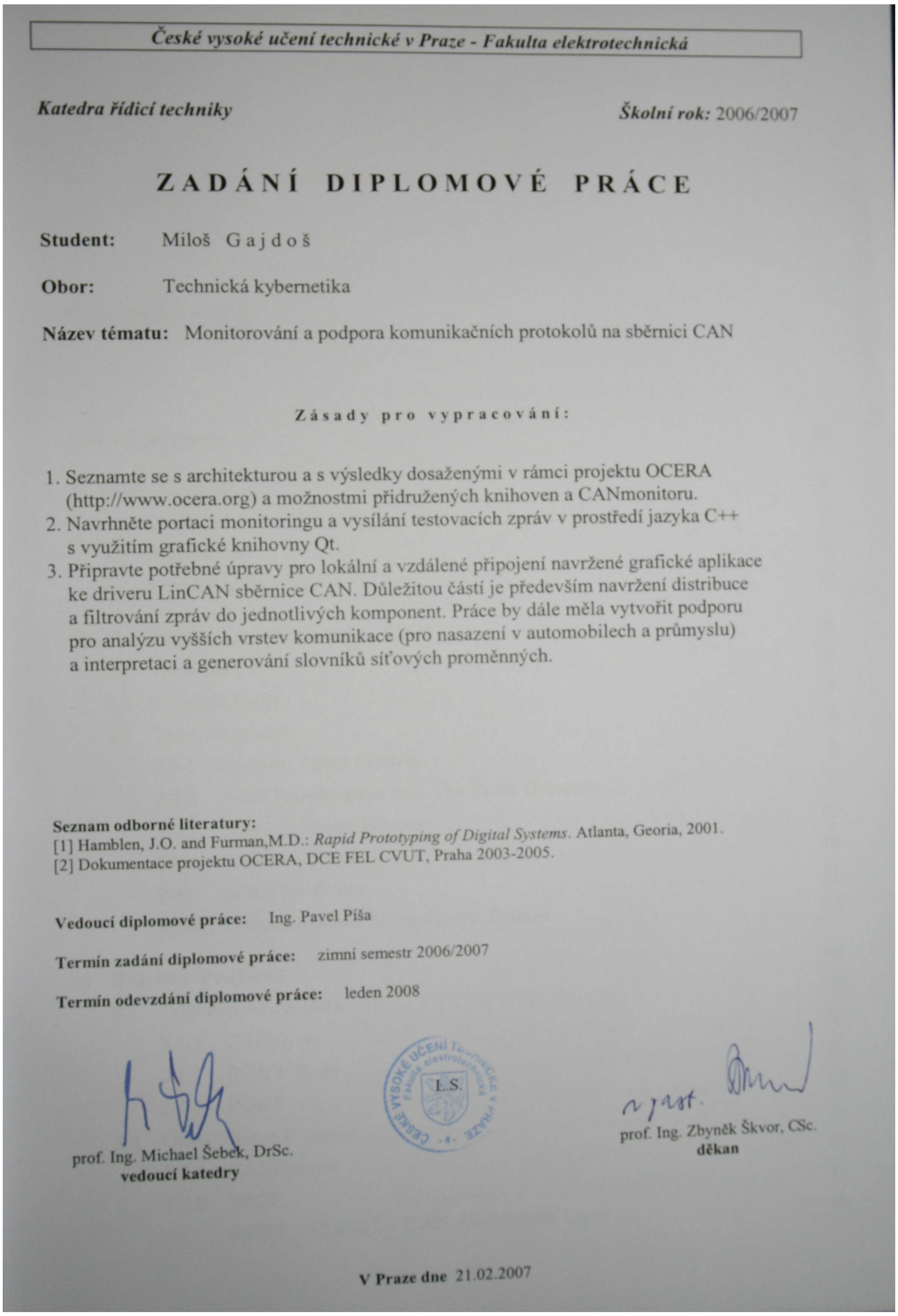
i

Abstrakt

Táto práca sa zaoberá monitorovaním systémov založených na komunikačnom protokole CAN (Controller Area Network). Hlavným cieľom práce je portácia monitoringu a posielanie testovacích správ v prostredí jazyka C++ s využitím grafickej knižnice Qt. Teoretická časť práce popisuje základné princípy protokolu CAN a zaoberá sa návrhom distribúcie a filtrácie monitorovaných správ do jednotlivých komponent grafickej aplikácie, s využitím pripojenia tejto aplikácie k ovládaču CAN. Praktická časť popisuje návrh a implementáciu grafickej aplikácie nazvanej QCANalyzer, ktorá by mala podporovať analýzu vyšších vrstiev komunikácie pre ľahké nasadenie v automobilovom priemysle rovnako ako interpretáciu a generovanie slovníku sieťových premenných.

Abstract

This diploma theses deals with the monitoring of CAN protocol based systems. The main goal of this work is implementation of monitoring application and sending of test messages in C++ language environment, with the use of Qt software development toolkit. Theoretical part of the work describes the basic principles of CAN protocol and discusses several possible suggestions of distribution and filtration of monitored messages into particular components of graphical application, with the use of connection of this application to CAN driver. Practical part describes the design and the implementation of the graphical application called QCANalyzer, which should support analysis of higher communication layers for easy stocking in the automotive industry, as well as the interpretation and generation of the network dictionary variables.



České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky **Školní rok:** 2006/2007

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Miloš G a j d o š

Obor: Technická kybernetika

Název tématu: Monitorování a podpora komunikačních protokolů na sběrnici CAN

Zásady pro vypracování:

1. Seznamte se s architekturou a s výsledky dosaženými v rámci projektu OCERA (http://www.ocera.org) a možnostmi přidružených knihoven a CANmonitoru.
2. Navrhněte portaci monitoringu a vysílání testovacích zpráv v prostředí jazyka C++ s využitím grafické knihovny Qt.
3. Připravte potřebné úpravy pro lokální a vzdálené připojení navržené grafické aplikace ke driveru LinCAN sběrnice CAN. Důležitou částí je především navržení distribuce a filtrování zpráv do jednotlivých komponent. Práce by dále měla vytvořit podporu pro analýzu vyšších vrstev komunikace (pro nasazení v automobilech a průmyslu) a interpretaci a generování slovníků síťových proměnných.

Seznam odborné literatury:
[1] Hamblen, J.O. and Furman,M.D.: *Rapid Prototyping of Digital Systems*. Atlanta, Georia, 2001.
[2] Dokumentace projektu OCERA, DCE FEL CVUT, Praha 2003-2005.

Vedoucí diplomové práce: Ing. Pavel Píša

Termín zadání diplomové práce: zimní semestr 2006/2007

Termín odevzdání diplomové práce: leden 2008

[illegible]
prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

v zast. [illegible]
prof. Ing. Zbyněk Škvor, CSc.
děkan

V Praze dne 21.02.2007

Contents

List of Figures

Chapter 1

Introduction

1.1 Prologue

Communication can be defined as a process that allows organisms to exchange information by several methods. In modern times communication is not only an issue of living organisms. It became a very important part of technical world. The world is now filled with the machines of different types that help people to communicate and replace them in many kinds of works due to several reasons. Machines are very often used for doing the jobs that a man is not able to perform. If you replace a human being with a lot of machines from which each of them is doing its particular part of job, you need to make them communicate with each other to acheive the goal they were designed for. Thus, when speaking about communication it is very important to be sure about what aspects of communication one is speaking about. Communication requires that all parties understand a language that is exchanged among them. To make the machines understand each other there had to be designed a way how they can exchange information with its surroundings by several methods. One of the possibilities to acheive this is to design a **communication protocol**.

Communication protocol is a set of standard rules for data representation, signalling, authentication and error detection required to send an information over a communication channel. It is basically following certain rules so that the system works properly. There is a lot of procedures that can help you to design your own protocol if you decide so. New communication protocol should comprise three main principles: *Effectiveness, Reliability, and Resiliency.* By *effectiveness* we mean, that it needs to be specified in such a way, that engineers, designers, and in some cases software developers can implement and/or use it. By *reliability* we mean error detection and correction and by *resiliency* we mean a readdressation of communication network failure, known as topological failure, in which a communications link is cut and a degradation below usable quality.

1.2 Motivation

There is a lot of already designed communication protocols assigned by several standardization organizations such as IETF, IEEE, ISO etc. Although sometimes we don't even know, we come to a contact with them every day. I am going to talk about the *industrial network communication protocols*. Such of them are for example: *Profibus, Interbus, Lon* etc. It depends on your application requirements whether you decide to design your own protocol or you will use some of the mentioned protocols. Each of them is suitable for different kind of application. These protocols are hugely used in several (not only industrial) applications where humans provides only an unnecessary *supervision*. The supervision is needed by many reasons. Sometimes the application/system is quite huge and manual inspection of an error would be very hard. Supervision covers the **monitoring** of the correct netowrk behaviour and takes care of unexpected events (such as corrupted communication eg. by mechanical problems, incorrect data exchange caused by unknown reasons), or *configure*s and provides a *diagnosis* of network devices. Each of these mentioned issues can be done and read (from monitored communication) thanks to the mechanism that is provided by a particular communication protocol. To conclude, by supervision we mean analysis of a communication over the communication channel among particular parties. To acheive this goal, there is a need to use or create a specialized tool that enables an easy analysis and high level inspection of the communication itself.

1.3 Problem statement

This work should support a monitoring of one of the above mentioned industrial protocols called **CAN** (Controller Area Network). It will provide a mechanism that will enable creating an application supposingly with an intuitive GUI (Graphical User Interface). This application will ease the analysis of the communication over the CAN bus. In order to create such an application there had to be designed a *driver* that would enable a communication of the particular hardware with this application program . Commercial vendors usually offer their own CAN hardware with their own driver and actual monitoring aplication, or an application with the drivers that support only very small group of CAN hardware, usually their own. This provides a comfort and a support of the product by its vendor, but on the other hand impossibility of adjusting it to your needs. Plus, there is also quite an important aspect for deciding to design your own application - the prize of commercial products.

Before I started to work on this diploma theses I had researched the internet, which is a normal practice when a new software is going to be developped either by company or by any programmer. This research gave me a picture about what commercial and open-source software dealing with CAN monitoring looks like (since the graphical user interface is very important part of this work) and what functionality it offers to its users. I had put the emphasis on portability to different operation systems and also on supported hardware. When researching the open source projects, I considered also a possibility of their further extension, while researching of the commercial projects was based on strictly practical reasons. List of researched projects with additional discussion can be found in the third chapter of this work.

Chapter 2

Controller Area Network - CAN

2.1 Overview

Before I start to talk about actual the aspects of this diploma theses, I will shortly present at least some of the basic principles of CAN that are necessary for understanding of the whole concept of this work.
CAN is a serial bus system, originally developed by the BOSCH company for automotive applications in the early 1980's. It is being used in a large and still growing extent especially in automotive industry (for which it was designed), but currently also in chemical industry and in many other branches of automation. CAN is currently supported by many leading manufacturers of integration circuits. It became a favourite alternative when designing an *intelligent* and *redundant* industrial network. The reasons for that are low cost, easy stocking, reliability, high transmission rate, high level of error detection and availability of maintanance parts. CAN was internationally standardized in 1993 as ISO 11898-1 and comprises two of seven ISO/OSI layer reference model. Those are *data link layer* and a part of *physical layer.* CAN provides two basic communication services between the nodes: data *frame transmission* (sending of a message) and remote *transmission request* (RTR - requesting of a message). The data integrity services such as error detection or automatic re-transmission of error frames are user-transparent, which means the CAN chip automatically performs these services. CAN offers a *multi-master hierarchy*, which allows every node on a bus to be a master and control other nodes. Thus, if one node is defect, the network is still able to work. CAN communication is based on a *broadcast* communicaton mechanism where a sender transmits the messages to all devices on the bus. All receiving devices read the message and then decide if they accept it. This

is called **Acceptance Filtering** and and it is shown for better understanding at figure 2.1.

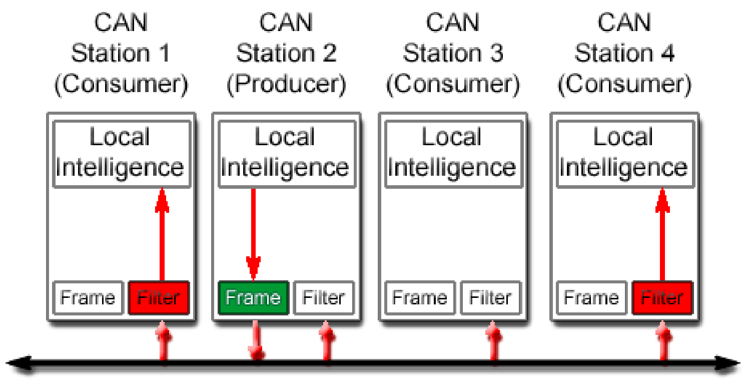


Figure 2.1: CAN Acceptance Filtering

Acceptance filtering guarantees the common view on transferred data, as all devices in the system use the same information. Every message starts with a *unique identifier* within the whole network, which defines a *content* and a *priority* of the message. The highest priority has the identifier of 0. CAN guarantees that the message with the higher priority is delivered preferably in case of a collision of several messages. By the identifier you can set the node to accept only for it significant messages. CAN offers a sophisticated error detecting mechanisms and re-transmission of faulty messages. The conformance test for the CAN protocol is defined in the ISO 16845 which guarantees the interchangeability of the CAN chips.

2.2 Physical Layer

Basic requirement on physical media is the ability to represent a *dominant* and a *recessive* bit value. This is possible for electrical and optical media so far. Also powerline and wireless transmission is possible. The dominant and the recessive bits represent a generalized equivalence of logical signal levels. Their values are not strictly defined and actual representation depends on a concrete realization of physical media. For the electrical media the differential output bus voltages are defined in ISO 11898-2 and ISO 11898-3, in SAE J2411, and ISO 11992. With the optical media, the recessive level is represented by ”dark” and the dominant level by ”light”. The most commonly used physical media to implement CAN based networks is a differentially driven pair of wires (for vehicle body electronics single wire bus lines are also used) according to ISO-11898 standard. This standard defines electrical behaviour of transmitter and receiver as well as timing principles, bit encoding and decoding, synchronization and bit timing. The bus consists of two line wires (marked as CAN_H and CAN_L), where the dominant or the recessive level is determined by differential voltage of these two wires. The bus is adjusted on both ends by using terminating resistors to eliminate bounces on a power line . Fundamental structure of a CAN network according to ISO-11898 norm is shown at figure 2.2.

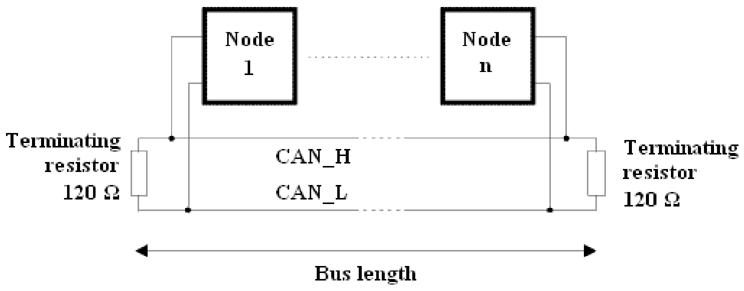


Figure 2.2: CAN network structure according to ISO-11898

There is no limit for number of the nodes connected to the bus caused by CAN protocol principles. Only maximal number of different message types is limited by size of the message ID fiels. But with regard to a bus and to ensure the correct static and dynamic parameters of the bus, the standard recommends maximum amount of 30 nodes. Depending on a number of connected nodes that causes a data propagation delay, maximum possible bus length at a specific data rate is determined. The signal propagation is determined by the two nodes within the network that are furthest apart from each other. It is the time that it takes to get from one node to the furthest one (considering the delay caused by transmitting and receiving), synchronization and the signal that returns back from the second one to the first one. Only then can the first node distinguish either its own signal level is the actual level on the bus or it has been replaced by the dominant level by another node. This fact is important for the *bus arbitration.*

In order to achieve the highest possible bit rate at a given length, a high signal speed is required. Maximum bus line length for the transmission rate of 1Mbit/s is 40 meters.

The standard doesn't specify the bus line length for other transmission rates. Other lengths for other transmission rates than 1Mbit/s depends on many parameters (eg. type of cable etc.) However it can be proven that for lower transmission rates the maximum bus line lengths will be bigger.

2.3 Data Link Layer

The data link layer consists out of two sublayers - MAC (*Medium Access Control*) and LLC (*Logical Link Contol*).

First of them takes care of medium acces, provides bit coding, bit stuffing and destuffing, controls the access of all connected nodes to the physical medium connsidering the message priority, detection of errors and their reporting and finally acknowledging of correctly received messages. The second one takes care of message filtering (*Acceptance Filtering*) and reporting of warnings about bus overload (*Overload Notification*).

2.3.1 Medium Acces Control

If a bus is in a *Bus Free state*, any of the connected nodes can start a transmission. First one that starts it, gets the bus for itself and the rest of connected nodes have to wait until it is finished transmitting a message. Then they can start transmitting if they get the bus for themselves. The only exception is the error frame transmission that can be transmitted by any node if it detects an error in actual broadcasting message. If several nodes start transmission at once, then the access to the bus is gained by the node that transmits a message with the higher priority (lower identifier). Every transceiver compares the actual transmitted bit value with the bit value on the bus and if it finds out that they differ (the only possibility is when it is transmitting a recessive bit and there is a dominant bit on the bus), it immediately interrupts the transmission. Described real-time concurrent transmission by several nodes is shown at figure 2.3.

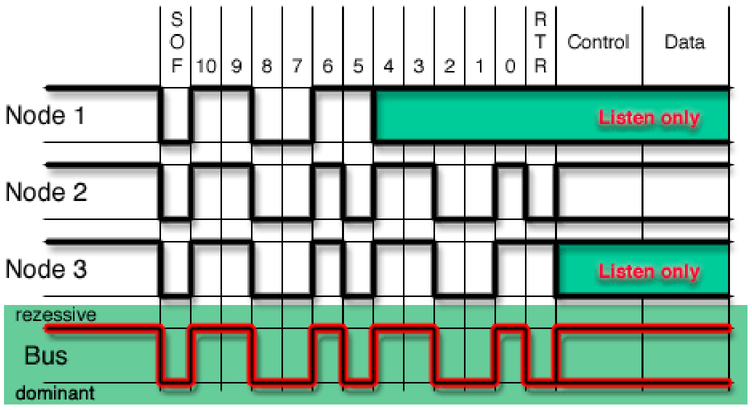


Figure 2.3: CAN data transmition

This guarantees that a message with a higher priority will be sent preferably and that it does not get corrupted. That could result in retransmitting of the message and an unnecessary time delay needed for transferring of the message. The node that didn't get the access to the bus must wait until the bus will be back in the Bus Free state. Then it can try to send the message again.

2.3.2 Data Transmission and The Error Detection

Messages transferred using CAN protocol are protected by several mechanism that work concurrently. When an error occurs, the node that detects it, generates an error frame. Basic error detection mechanisms implemented in CAN are:

- **Monitoring**
  A transmitter compares a value of an actual transmitted bit with the bit detected on the bus.
- **CRC (Cyclic Redundancy Check) coding**
  At the end of each message there is a 15 bit CRC code, generated from all previous bits of a message
- **Bit stuffing**
  After five consecutive equal bits the transmitter inserts a stuff bit into the message. This stuff bit has a complementary value, and is removed by the receivers. Apart from the error detection it is used for synchronization

- **Message frame check**
  A message is verified against the format and the size given by the specification
- **Acknowledge**
  If a message is received by any of the nodes it sends an ACK message back to transmitter (this is done by changing ACK bit). If a transmitter does not receive an ackowledgement an ACK error is indicated.

Every node has two internal error counters that determine number of errors when accepting or transmitting messages. According to the value of this counter a node can be in one of 3 states : *Error Active, Error passive, Bus-off*. If a node generates too many error messages it is automatically disconnected (switched to *Bus-off state*).

2.3.3 Message Frame Formats

The CAN protocol defines two basic message frame formats: *standard* and *extended*. There is only one difference between them - the length of the message identifier. The CAN standard frame has the identifier 11 bits long while CAN extended frame has 29-bit identifier. Apart from the data message frames, CAN defines 2 communication management frames.

2.3.4 Standard Frame

Structure of the standard frame is shown at figure 2.4. The frame starts with the **SOF** (start of frame) bit which is always dominant. It is followed by the *arbitration field* that contains 11-bit identifier and the **RTR** bit (Remote Transmission Request) that distinguishes the data frame from the data request frame. This bit has to be dominant in data frame and recessive in data request frame.

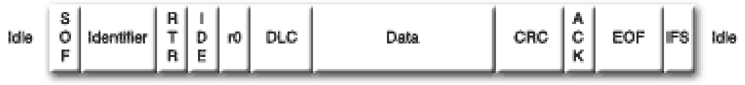


Figure 2.4: CAN message frame

After this bit the *control field* follows, containing 2 reserved bits and the 4-bit field that carries the information about the data length that are sent with the message. Then the actual *data field* comes. Only 8 bytes of data can be hold by one CAN message. The frame integrity is guaranteed by 15-bit CRC field.After **CRC** *field* there is a 1 bit delimiter that separates CRC field from **ACK** (acknowledgement) bit that is followed by another delimiter bit - **ACD** (acknowledgement delimiter). After the ACD bit, the **EOF** (end of frame) field follows. The space between two transmitted messages (interframe space) is 3 bits long.

2.3.5 Extended Frame

The message identifier of the extended frame is 29 bits long and is divided into two fields: 11 bit identifier (as the standard frame) and 18 bit identifier extension. The difference between standard and extended frames is the IDE bit. Accordng to this bit it is decided which message has the priority when accessing the bus in case of collision with the different formats that has the same base identifier. 11-bit identifier has always priority over 29-bit identifier, therefore it is transmitted as dominant in standard frame and as recessive in case of a 29-bit frame. When sending the extended frame, a bus latency time is longer (in minimum 20 bit-times), messages in extended format require more bandwidth (about 20%), and the error detection performance is lower.

2.3.6 Communication Management Frames

By the communication management frames we mean the *Overload Frame* and the *Error Frame*. They are used to manage the communication over the bus concretely for the error detection and signalling and for the requesting of a communication latency. The overload frame is usually transmitted by the nodes that are not able to receive and process any data due to their actual load. Error frame is used by any node that detects an error on the bus.

Further information about CAN protocol can be found on web pages maintained by the group of CAN manufacturers'[1].

Chapter 3

Related Projects

This chapter lists and shortly presents the most interesting projects dealing with the CAN monitoring subject that I have found on the internet. These projects can be divided by many criterias. I have chosen one of the most important when deciding to create or bring a new application on a software market - *economical criteria*. By that, we can divide these projects to Commercial (protected by copy rights) and Open Source (source codes are freely downloadable and distributable). It depends on your actual needs and economical status which one you decide to choose. I have chosen three projects I considered the best from each group.

3.1 Commercial projects

Commercial software offers a lot of advantages from which most important are maintanance and support of the company that develops and sells it. On the other hand it has many disadvantages - unavailability of application source codes, expensiveness, undesired functionalities and so on.

3.1.1 CANalyzer

It is developed by the Vector company. It provides advanced software tools not only for monitoring of CAN based systems. CANalyzer offers a graphic block diagram interface from which CAN based system is controlled or monitored. The basic functions offer multiple usage options (monitoring and listing of a bus traffic, graphical or textual display of traffic, sending of predefined message sequences etc.).

What distinguishes this tool from other commercial programs are several special functions that it provides. From many of them, the most interesting is programmability with *CAPL* (Communication Access Programming Language). By programmability I mean adjusting of user needs by inserting special graphical blocks at any point of data flow plan, so the user can program their function with the use of C-like language CAPL, for which it has an interactive development environment. Tool supports all bus interfaces offered by Vector. Complete list of supported HW can be found on company's web pages[2].

CANalyzer system requirements:

- Processor Pentium 4/2,6 GHz Pentium III/1GHz
- Memory (RAM) 1 GB 512 MB
- Hard disc space 200 - 600 MB (depending on the chosen option)
- Operating system: Windows Vista / XP SP2 / 2000 SP4 (Vista for CAN and LIN only, XP and Vista only 32 bit)

3.1.2 PCAN Tools

PCAN is a a package of software tools developed by the Peak company. It comes on dedicated CDs for free with the purchased HW modules. It covers the wide range of tools. From simple CAN monitoring ones, like PCAN-View, to more advanced tools like PCAN-Explorer. PCAN-Explorer provides all necessary functionality needed for advanced monitoring and analysis of CAN based systems (advanced logging mechanism, graphic views of monitored messages and data, support of CANopen etc.). PCAN-Light tool enables a selection of different drivers for Windows and Linux operating system.

PCAN-development tool allows a programmer to develop its own application using API libraries provided in form of precompiled libraries. Unfortunately, these APIs offer only a small range of functions (open, close, write, read, status, setfilter, resetfilter, resetclient etc.). On the other hand, they are available for different programming languages (C++, C#, Delphi, Borland Builder, VB.NET and VB header files) and come with the HTML documentation (as each tool in this package). Peak offers a freely downloadable driver for Linux operating systems, but it supports only a small group of all HW compoments developed by the Peak company. The whole list can be found on company's web pages[3].

PCAN System requirements:

- Windows 2000 / XP / Vista
- at least 256 MB RAM 700 MHz CPU
- Hard disc space 100 - 500MB (depending on the chosen options)

3.1.3 PORT

Port is another company offering CAN monitorubg software tools. Apart from developing of their own CAN HW modules, they offer a simple CANopen device monitor tool programmed in Tcl/Tk. Tcl/Tk is a GTK+ library binding for Tcl scripting language. This tool provides a basic CANopen features (editing of EDS, SDO/PDO transfers, NMT atc.). The usage of the CANopen Device Monitor requires a CAN interface - currently available interfeces are: USB, ISA, PCI, parallel port, serial port and PC-104 or an Ethernet interface like the standalone CANopen-Gateway-Server EtherCAN. Server can be started either in Windows OS or in Linux environment. The simplest application provided by Port, is a simple CAN-analyzer (also programmed in Tcl/tk) that can connect to the server and enable sending of messages (simple or periodic). Supplementary software modules provide extended functionality like service and protocol dependent interpretation of CAN messages.

3.2 Open Source Projects

Open source projects are usually maintained on a public place - usually the internet or some other publicly accesible source file repository where all codes are freely available to everyone for free. They are released under public license, depending on which, it is set what can be done with downloaded source files. Many of them are realeased under a license that allows anyone to change it and freely distribute it with their own actual application. Unfortunately, their documentation is often very poor.

3.2.1 CANFestival

CanFestival is a CANOpen framework, licensed with GPLv2[1] and LGPLv2[2]. It provides an ANSI-C platform independent CANOpen stack that can be implemented as master or slave nodes on PCs, Real-time IPCs, and Microcontrollers. With CanFestival you can :

- turn any microcontroller or PC into a CANOpen node
- edit Object Dictionary and EDS files
- use any CAN interface type
- link with a proprietary code

CanFestival focuses on providing an ANSI-C platform independent CANOpen stack that can be implemented as master or slave nodes on PCs, Real-time IPCs, and Microcontrollers.Supported CanOpen protocols are PDO, SDO, NMT, SYNC, Node Guarding, Life Guarding, Heartbeat and Bootup. CanFestival uses Arbracan CAN driver that supports only Linux 2.4 and Adlink Board, and is slowly deserted by users. Recently, there was added a new IXXAT VCI driver interface. Nevertheless, the best way to work with the CanFestival CanOpen stack is to bind it to an existing low level CAN driver. CanFestival provides GUI (CanFestivalGUI) and command line tool (CanFestival) that help in a process of creating a new CanOpen node and editing Object Dictionary.

CANFestival GUI (screenshot of the application is shown at figure 3.1) is programmed in Qt 2.x and supports many functions of CanFestival in an easy way:

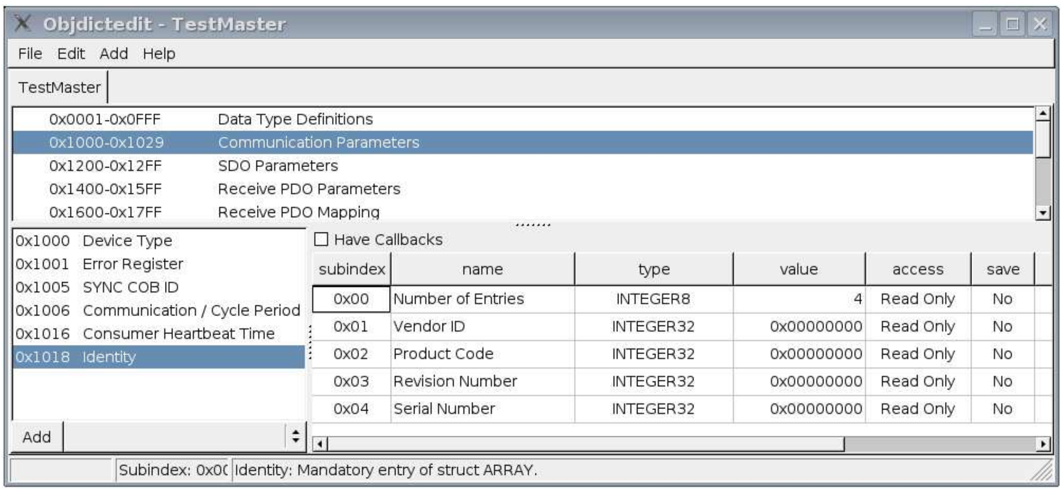
- Sending/Receiving raw CAN Messages
- Sending/Receiving of PDOs or SDOs
- Sending/Receiving of NMT
- Sending/Receiving of Sync Messages

Furthermore it provides some additional useful functions:

- Simulation Mode: in this mode you don't need an actual CAN interface
- Loading/unloading of the device driver (Arbracan, IXXAT) by pushing a button in the GUI. (module parameters are hardcoded at the moment. To modify them, you'll have to modify C++ code and recompile the software)

[1]General Public License
[2]Lesser General Public License

- CANFestival GUI acts as a CAN sniffer (receiving of CAN Messages is done in a separate thread)
- Values which should be sent over the CAN/CANopen network can be typed in as hex, decimal or binary values. Each of them can be converted to the other formats



0x0001-0x0FFF	Data Type Definitions
0x1000-0x1029	Communication Parameters
0x1200-0x12FF	SDO Parameters
0x1400-0x15FF	Receive PDO Parameters
0x1600-0x17FF	Receive PDO Mapping

0x1000	Device Type
0x1001	Error Register
0x1005	SYNC COB ID
0x1006	Communication / Cycle Period
0x1016	Consumer Heartbeat Time
0x1018	Identity

subindex	name	type	value	access	save
0x00	Number of Entries	INTEGER8	4	Read Only	No
0x01	Vendor ID	INTEGER32	0x00000000	Read Only	No
0x02	Product Code	INTEGER32	0x00000000	Read Only	No
0x03	Revision Number	INTEGER32	0x00000000	Read Only	No
0x04	Serial Number	INTEGER32	0x00000000	Read Only	No

Subindex: 0x0(Identity: Mandatory entry of struct ARRAY.

Figure 3.1: CANfestivalGUI in use

CanFestival runtime library is released under LGPL license, thus it can be linked with any code, proprietary or not. Bindings are very simple and can be theoretically adapted to any target or CAN interface. CANFestival necessary requirement is the only PC CAN Board supported - AdLink PCI-7841 Card. In order to be able to use the CANFestival GUI you need at least the following things:

- Qt 2.x installed on your computer
- An AdLink PCI-7841 Board is recommended, but if you do not own such a card, you can still use this program in simulation mode

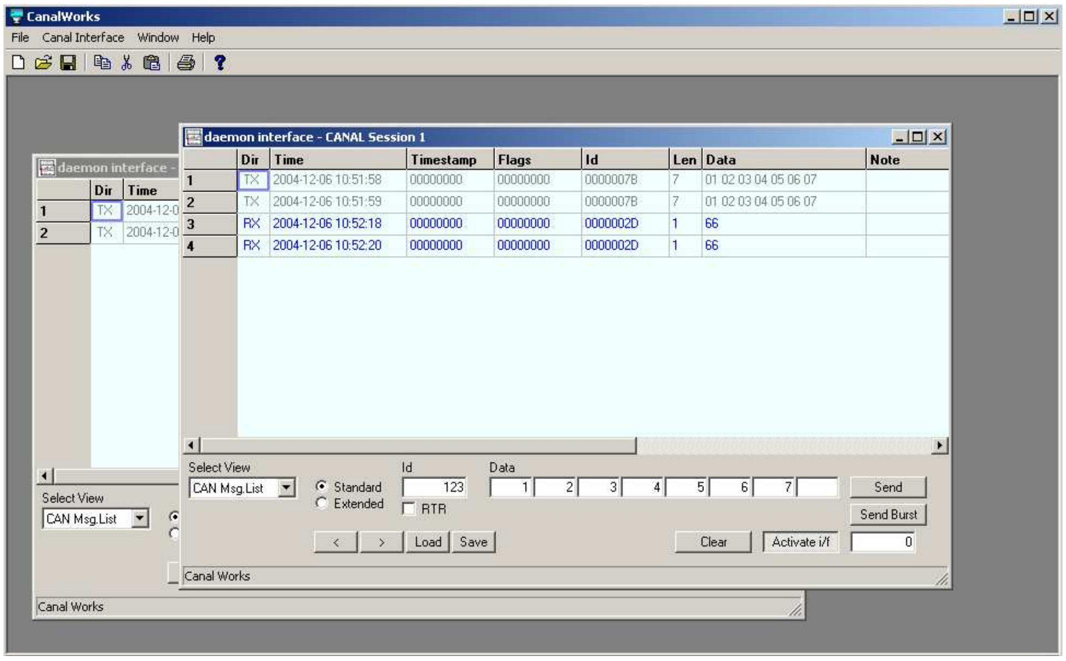
3.2.2 VSCP

VSCP stands for Very Simple Control Protocol. It has been developed for use on low end devices such as microcontrollers. It is more than just a protocol. It is a complete solution for configuration and control. It is very easy to use and very capable. It can be used in very demanding control situations. VSCP does not assume anything about the lower level system. It works with Ethernet TCP/IP, Wireless, Zigbee, Bluetooth, CAN, GPRS, RS-232, USB and everything else user wants. It is just a uniform way of describing the systems available. Every control situation can be described and implemented using VSCP. The VSCP Protocol was from the beginning designed to be used in CAN networks. It enables to understand the inner workings of the protocol with just minimal effort and it is supported throughout the industry. Even though the protocol is designed for CAN, there is no need for its using. It can be used equally well in other environments. It offers some of the following features:

- It is free and open for commercial and other use
- It has two levels. Level I and Level II where level I is designed with CAN as the least common denominator. It can be used for TCP/IP, UDP, RF, Mains, etc.
- It has globally unique IDs for each node
- It has a mechanism to automatically assign a unique ID to a newly installed node
- It has software and drivers for Windows and Linux operating systems
- It has a common specification language ”MDF” that describes a module in an uniform way that can be used by set up software

3.2.2.1 CANAL - CAN Abstraction Layer

CANAL is the lower layer for the CAN based nodes. It is built as two separate solutions just to make the CANAL useful also for other CAN tasks. It defines a message format and some basic function calls. On top of this, some CAN drivers have been built. The can232 driver for the Lawicel adapter is a typical example but there are being developed other for IXXAT, Zanthic Technologies Inc., Ferraris Elettronica, Vector, OMK (OCERA project) and all the others. The whole list can be found on the project web pages[4]. Application rogrammer can use one programming interface to all of them. Thus, one can build the application that works with the different types of hardware. A typical example of this kind of application is the *Canal* diagnostic tool - CanalWorks, that can work directly with any available device. Screenshot of the application is shown at figure 3.2.



	Dir	Time	Timestamp	Flags	Id	Len	Data	Note
1	TX	2004-12-06 10:51:58	00000000	00000000	0000007B	7	01 02 03 04 05 06 07	
2	TX	2004-12-06 10:51:59	00000000	00000000	0000007B	7	01 02 03 04 05 06 07	
3	RX	2004-12-06 10:52:18	00000000	00000000	0000002D	1	66	
4	RX	2004-12-06 10:52:20	00000000	00000000	0000002D	1	66	

Id: 123 Data: 1 2 3 4 5 6 7 Count: 0

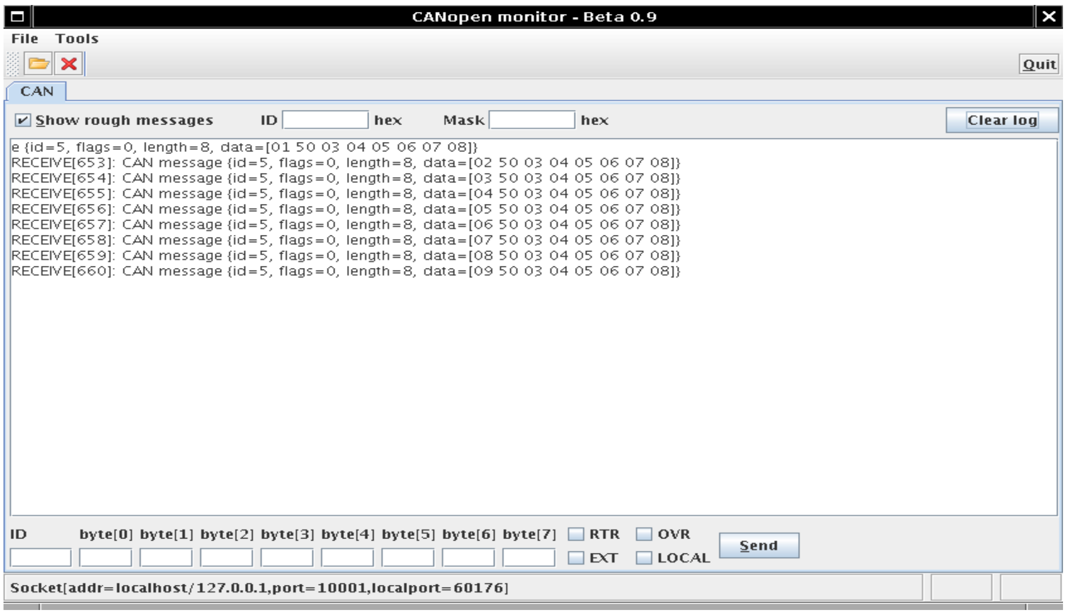
Figure 3.2: CanalWorks in use

CanalWorks can be used on WIN32 and Linux/Unix. The work on this application is still in progress. At the moment it allows you to open several channels to the VSCP Daemon (will be discussed later) and/or directly to the canal interfaces. You can send and receive messages, choose an interface or choose which driver should be loaded, set/remove filter masks and allows a user to have multiple session. The tool is working on both - the WIN32 and the Linux platform. On a top of CanalWorks, there was a daemon built (VSCP daemon). On one side there is a canal interface for clients and on the other side there is a canal interface for drivers. Programmer tells it which drivers to load at start-up. A client can now connect to the daemon (available for both Windows and Linux), send a message and it will be sent to all other clients and to all devices.

Project offers also a simple application called LoggerWnd that let you view the traffic on your net, then Canalocx tool that offers ActiveX control for interfacing the daemon and many other interesting tools. VSCP daemon and the applications are released under GNU GPL license, drivers, classes and interfaces are released under GNU LGPL. Unfortunately VSCP API is very simple and even the authors recommend to use more sphisticated API from different projects (OCERA or CANpie API)

3.2.3 OCERA

OCERA, that stands for Open Components for Embedded Real-time Applications is an European project, based on Open Source, which provides an integrated environment for embedded real-time applications. OCERA combines the use of two kernels, Linux and RTLinux-GPL to provide support for critical tasks (RTLinux-GPL executive) and soft real-time applications. Several components for both environments have been developed to bring an innovative development and deployment platform to the embedded system developer. The OCERA project offers LinCAN (LinuxCAN) driver, that supports many hardware interfaces. Appart from the driver, it offers a simple monitoring tool programmed in Java that provides some basic monitoring features like sending of messages to the bus, receiving, loading and editiing of EDS files and some other additional functionatilities.



```
e {id=5, flags=0, length=8, data=[01 50 03 04 05 06 07 08]}
RECEIVE[653]: CAN message {id=5, flags=0, length=8, data=[02 50 03 04 05 06 07 08]}
RECEIVE[654]: CAN message {id=5, flags=0, length=8, data=[03 50 03 04 05 06 07 08]}
RECEIVE[655]: CAN message {id=5, flags=0, length=8, data=[04 50 03 04 05 06 07 08]}
RECEIVE[656]: CAN message {id=5, flags=0, length=8, data=[05 50 03 04 05 06 07 08]}
RECEIVE[657]: CAN message {id=5, flags=0, length=8, data=[06 50 03 04 05 06 07 08]}
RECEIVE[658]: CAN message {id=5, flags=0, length=8, data=[07 50 03 04 05 06 07 08]}
RECEIVE[659]: CAN message {id=5, flags=0, length=8, data=[08 50 03 04 05 06 07 08]}
RECEIVE[660]: CAN message {id=5, flags=0, length=8, data=[09 50 03 04 05 06 07 08]}
```

```
Socket[addr=localhost/127.0.0.1,port=10001,localport=60176]
```

Figure 3.3: Canmonitor in use

Canmonitor (shown at figure 3.3) is a simple TCP/IP client that connects to a canmonitor daemon(called *canmond*), which enables sending and receiving of CAN messages via TCP/IP protocol. Its greatest advantage is that it is programmed in Java(multiplatform portability), but it lacks a lot of useful features in a view of user needs. The greatest advantage of the OCERA project is its high level virtual CAN API provided for system programmers to enable them communication with the LinCAN driver.

3.2.4 Conclusion

From the researched projects I deduced several conclusions. Commercial projects, like the Vector CANalyzer or the Peak PCAN-Explorer are very advanced tools, and are still being developed by many programmers. In this work I can only try to come close to these projects and at least get some inspiration for my own application. Open source projects offer also some advanced applications, but they have several issues. CANFestivalGUI offers a smart CANopen tool, but on the other hand this tool is programmed using Qt 2.x, which is a very old version of this library and not further supported by the vendors of Qt. So the possibility to extend this application was refused by this fact. VSCP project offers the CanalWorks application, but it uses only a very simplified API. Even by the authors of the VSCP recommend to use more advanced API (like the one offered by the OCERA project) in order to create a more advanced tool. I had decided to use in my work some of the components offered by the OCERA project, due to the following reasons:

- it is testted currently being used by some companies in their industrial applications
- it is released under GPL license
- it offers a *LinCAN driver*(to enable communication with several HW interfaces) and *canmond* daemon(for the remote monitoring)
- it offers a sophisticated VCA (virtual CAN api) system independent library
- some of the project contributors work at my university, so there is an easy way to contact them and consult any problem

LinCAN driver and the Virtual CAN API library will be shortly discussed in next chapter.

Chapter 4

LinCAN - Linux CAN driver

4.1 Overview

LinCAN is a Linux *kernel loadable* module that implements a CAN driver capable of working with multiple cards, even with different chips and IO methods. Each communication object can be accessed from multiple applications *concurrently*. Most important feature is that the driver supports multiple open of one communication object from more Linux and even RT-Linux (Real Time) applications and threads. The usage of the driver is tightly coupled to the virtual CAN API interface component which hides driver low level interface to the application programmers. LinCAN supports RT-Linux, 2.2, 2.4, and 2.6 with fully implemented `select`, `poll`, `fasync`, `O_NONBLOCK`, and `O_SYNC` semantics and multithreaded `read/write` capabilities. It works with the common Intel i82527, Philips 82c200, and Philips SJA1000 (in standard and PeliCAN mode) CAN controllers. The actual version has been tested at CTU by more OCERA developers, by Unicontrols and by BFAD GmbH, which use pre-OCERA and current version of the driver in their products.

Each chip/CAN interface is represented to the applications as one or more CAN *message objects* accessible as character devices. The application can open the character device and use `read/write` system calls for CAN messages transmission or reception through the connected message object. The parameters of the message object can be modified by the `IOCTL` system call. The closing of the character device releases resources allocated by the application. The intelligent CAN/CANopen cards should be supported by the near future. One of such cards is P-CAN series of cards produced by Unicontrols. The driver contains support for more than ten CAN card basic types with the different combinations of the above mentioned chips. Not all card types are held by the OCERA members, but Czech Technical University has and tested more SJA1000 type cards and will test some i82527 cards in near future.

4.2 Driver Architecture

The LinCAN provides simultaneous queued communication for more concurrent running applications. Driver's architecture is shown for better understanding at figure 4.1.

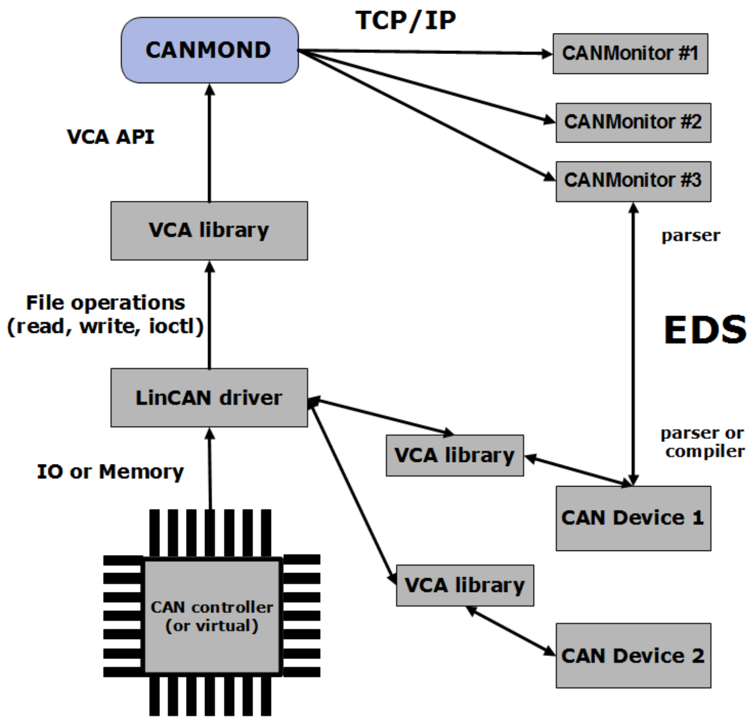


Figure 4.1: LinCAN architecture

Each communication object can be used by one or more applications that is connecting to the communication object internal representation by means of CAN FIFO queues. The driver can be configured to provide *virtual* CAN board (software emulated message object) to test CAN components on the Linux system without hardware required to connect to the real CAN bus. The example configuration of the CAN network components connected to one real or virtual communication object of LinCAN driver is shown at the figure above. The communication object is used by the CAN monitor daemon and two CANopen devices implemented by the OCERA CanDev component. The actual system dependent driver API is hidden to applications under **VCA** (Virtual CAN API) library. Each communication object is represented as a character device file. The devices can be opened and closed by applications in blocking or non-blocking mode. LinCAN client application state, chip and object configurations are controlled by IOCTL system call. One or more CAN messages can be sent or received through `write`/`read` system calls. The data read from or written to the driver are formed from sequence of fixed size structures representing CAN messages.

4.3 LinCAN system level API

4.3.1 Device Files and Message Structure

Each driver is a subsystem which has no direct application level API. The operating system is responsible for user space calls transformation into driver functions calls or dispatch routines invocations. The CAN driver is implemented as a character device with the standard device node names /dev/can0, /dev/can1, etc. The application program communicates with the driver through the standard system low level input/output primitives (open, close, read, write, select and ioctl). The CAN driver convention of usage of these functions is described in the next subsection. The `read` and `write` functions need to transfer one or more CAN messages. The structure `canmsg_t` is defined for this purpose and is defined in include file `can/can.h`. It has following fields:

```
struct canmsg_t {
    int flags;
    int cob;
    unsigned long id;
    canmsg_tstamp_t timestamp;
    unsigned short length;
    unsigned char data[CAN_MSG_LENGTH];
} PACKED;
```

`flags`
The flags field holds information about a message type. The bit `MSG_RTR` marks remote

transmission request messages. The bit `MSG_EXT` indicates that the message with extended (bit 29 set) ID will be send or was received. The bit `MSG_OVR` is intended for fast indication of the reception message queue overfill.

`cob`
The field reserved for a holding message communication object number. It could be used for serialization of received messages from more message object into one message queue in the future.

`id`
CAN message ID.

`timestamp`
The field intended for storing of the message reception time.

`length`
The number of the data bytes sent or received in the CAN message. The number of data load bytes is from 0 to 8.

`data`
The byte array holding message data.

A direct communication with the driver through system calls is not encouraged because this interface is partially system dependent and cannot be ported to all environments. The suggested alternative is to use OCERA provided VCA library which defines the portable and clean interface to the CAN driver implementation. The other issue is the addition of the support for new CAN interface boards and CAN controller chips.

4.3.2 CAN Driver File Operations

Following list of call operation will be described very shortly. Their detailed description can be found in LinCAN documentation at the OCERA project web pages[5].

- `open` - message communication object open system call

- `close` - message communication object close system call
- `read` - reads received CAN messages from message object
- `write` - writes CAN messages to message object for transmission
- `struct canfilt_t` - structure for acceptance filter setup
- `IOCTL CANQUE_FILTER` - sets acceptance filter for CAN queue connected to client state
- `IOCTL CANQUE_FLUSH` - flushes messages from receiption CAN queue

4.4 Summary

Authors of the LinCAN driver are: Pavel Píša, Arnaud Westenberg and Tomasz Motylewski. Last released version of LinCAN driver is 0.3. The driver was released under GPL license and can be found on several internet resources. For further information about LinCAN or about any other component of the OCERA project which it is part of, you can visit OCERA project home page[6]

Chapter 5

VCA - Virtual CAN API

API stands for Application Programming Interface. It is a source code interface that a library provides to a programmer to support his needs for services to be made of it by his application. The OCERA project offers quite an advanced API for the communicaction with the LinCAN driver. It is called Virtual CAN API - VCA. The main idea of VCA was to have only one application interface between the LinCAN driver and a application that uses it. Figure 5.1 shows the position of VCA library in the application program that uses LinCAN driver:

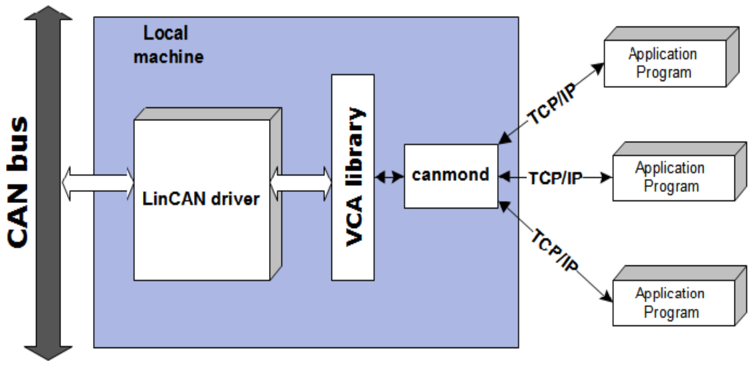


Figure 5.1: Position of VCA in application program

VCA can be divided into three parts:

- Base CAN protocol support
- CANopen protocol support
- Utility functions

The part of VCA, that provides base CAN protocol support will be described in more detail, because it is closely related to this work. Other components of VCA will be mentioned very shortly, since they will not be used in prepared CAN monitoring application.

5.1 VCA Base CAN Protocol Support

This part of VCA provides a basic set of primitive functions used for `open/close` of LinCAN driver and `send/receive` of CAN messages to and from the CAN bus. Most of this part is implemented in *vca_base.c* and *vca_pollfdnodecan.c* source files. These two components of VCA are very closely related to a part that allows an application programmer to create and monitor several CAN networks. Part that provides this possibility is implemented in *vca_net.c* source files. Using it, one can set-up several acceptance-filtering masks for the particular CAN nodes, present in simulated CAN netowrk, represented by their IDs. Finally, VCA provides a logging support used by the whole CAN/CANopen component. It is implemented in *vca_log.c*.

The most important component of this part, and probably most important component of the whole VCA library, is the mechanism that enables sending and receiving of CAN messages, as well as the mechanism that provides the acceptance-filtering. By these two mechanisms I mean redistributing of received CAN messages to the nodes that can accept them according to their acceptance-filtering masks. Sending and receiving of CAN messages is implemented unsing `write/read` function calls on a particular file descriptor that represents actual CAN bus interface, through which the driver communicates with actual CAN hardware (measurement card, pc, etc.). CAN message distribution in VCA is implemented using the **callback function mechanism**. This mechanism is based on a registration of particular callback function to an event. When the event occurs (for example a CAN message arrival), its registered callback function is called. This is quite fast and hugely used in many (not only C-based) applications and libraries. Every message that arrives on CAN interface is received and then dsitributed to all nodes in monitored network using this mechanism.

In order to distribute received CAN messages to particular CAN nodes, there had to be designed some specialized data structures that would keep all netowrk IDs and their acceptance-filtering masks. VCA stores these IDs in *generic sorted array* (GSA) list, to which the ID of new arriving message is added when a new node is connected to the network. IDs can have several acceptance-filtering masks set. These masks have to be kept in a structure that would provide fast lookup, adding and removing of an item. The speed of these operations is very important considering a situation of a huge CAN network traffic, when every new arriving message ID must be looked-up in a data structure that keeps the acceptance-filtering masks. After this lookup, the message is broadcasted to all CAN nodes that can accept it. The decision either to refuse or accept new message, must be taken in the shortest time. AVL/GAVL trees satisfy all mentioned requirements. VCA implements these structures using the uLan utilities library.

For better understanding of VCA architecture and message distribution look at the figure 5.2:

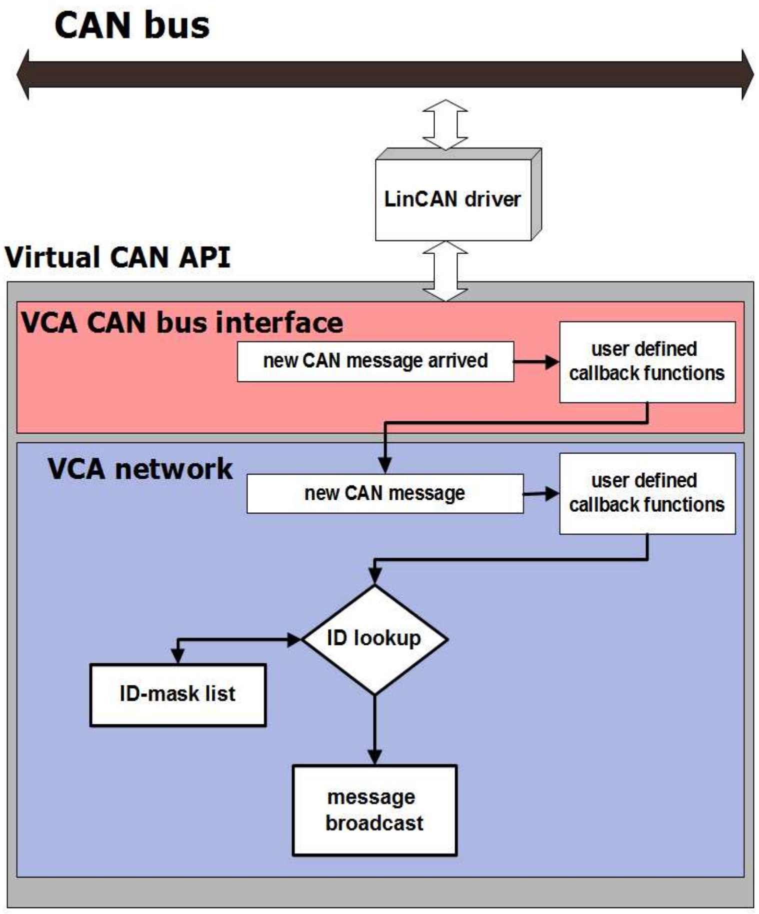


Figure 5.2: VCA architecture and message distribution

5.2 VCA CANopen Protocol Support

This part of VCA covers the *Object Dictionary* (OD) access as well as *Processs Data Object* or *Service Data Object* (PDO, SDO) processing of PDO/SDO requests. OD is implemented as a GAVL tree of OD objects. The functions that enable access to these objects (`vcaod_find_object`, `vcaod_get_value`, `vcaod_set_value`) are implemented in `vca_od.c` source file. The core structure for the SDO processing is in `vcasdo_fsm.c` file. It implements a CANopen final state machine. Wrappper for this state machine is implemented in `vca_net.c` source file. PDO processing is implemented using a structure called `vcaPDOProcessor_t`. PDO processor knows which OD objects are PDO mapped (because it is written in EDS) and it can store/retrieve them to/from OD automatically. Further information, as well as the documentation of implemented functions, can be found on OCERA web pages[4].

5.3 VCA Utility Functions

This group provides a set of help functions to parse text, convert it to number or serialize CAN messages to human readable form. Again, Further information as well as documentation of implemented functions can be found on OCERA web pages[4].

Chapter 6

Graphical library

The core of this work concludes creating a graphical application. In order to do so, I had to decide what graphical library will be used, to acheive this goal. I was considering only free available or Open-Source libraries and their portation to different operating systems. After small research I have narrowed my choice to the following possibilities:

- **GTK+** Toolkit
- **Qt** software developement toolkit
- **SWING** - Java widget toolkit

Each of them will be shortly discussed with regard to development of graphical user interface of CAN monitoring application.

6.1 GTK+

GTK+ is a multi-platform toolkit for creating graphical user interfaces. It offers a large set of different widgets. It is a free software (released under LGPL license) and a part of (the GNU Project). GTK+ has been designed to support not only C/C++ programming language. It also provides the support for the scripting languages such as Perl, Python and many others (the whole list of supported bindings can be found on GTK+ web pages[7]). Today, GTK+ is used by a large number of applications. From all of them, Wireshark or the GNU project's GNOME desktop environment are the most famous.

6.2 Qt

Qt offers the whole framework for high performance, cross-platform (not only graphical) application development. It is produced by the Trolltech company. It includes:

- An intuitive, easy to use class library
- Integrated development tools (qmake, Qt Linguist, Qt Designer, Qt Assistant)
- Support for C++ and Java development

Qt uses C++ with several non-standard extensions. It can be used in other programming languages too. Current supported bindings exist for Python (PyQt), Ruby (RubyQt), PHP (PHP-Qt), Pascal, C#, Perl, Java, and Ada. Qt based programs can be run on all platforms (Win32, Linux OS, MAC OS-X). Appart from the commercial edition, Trolltech offers also an Open-Source freely downloadable edition. The use of Qt in application development is increasing every year and the community of Qt developers is hugely growing. From many projects developped in Qt it is enough to mention KDE desktop environmment for Linux and Unix workstations, Mathematica, Opera web browser, Google Earth or Skype. Further information about Qt development framework can be found on Qt web pages[8].

6.3 Java - SWING

Swing is a widget toolkit for Java. It is a part of SUN Microsystem' Java Foundation Classes (JFC) - an API providing a graphical user interface for Java based programs. As Java, it is a cross-platform toolkit that guarantees uniform look and behaviour on all platforms. Further information about SWIN can befound on projects web pages[9].

6.4 Conclusion

The **GTK+** basic API is is based on C programming language. Since the VCA is also programmed in C, using GTK+ for GUI development of this application would speed up the whole development process. Structures and mechanisms provided by VCA could easily be connected to GTK+ graphical objects without any major change of their implementation. Programming in C offers also a speed advantage of a programmed application and eventually better portability. On the other hand, writing a graphical application is a process completely different from developing of a typical console program or library. It would require to design a lot of graphical object which would be easier implemented using an object oriented approach. Another big disadvantage of developing in GTK+ is its poor documentation. My experience with the GTK+ toolkit couple of years ago, when I was trying to develop CAN monitoring application, warned me before using it again. Development of CAN monitoring tool went quite slowly, mainly because of the documentation resources. GTK+ web pages offer short tutorial and API reference, but for a person who has never used it before, it is very difficult.

Java is a cross platform object oriented programming language. It has a lot of advantages - advanced documentation resources, portability to different operating systems and many other. Even the OCERA project offers simple monitoring application that is written in Java-SWING. I could easily continue in its developement, but sevral reasons made me reject this option. I will mention the one that I found the most important - portabiliy of Java (even the execution of any Java program) is **strongly** dependant on its version although its developers would disagree.

According to the problems that would come out if GTK+ or Java-SWING was used for development of CAN monitoring application, I turned my attention to Qt. First, I had to accept some handicaps that this solution would bring. Programming in Qt will cause a little slow-down of the application comparing to the GTK+ (because of the overhead that comes with the object oriented programming), but it would still be faster than any Java-based program. Plus, connecting a graphical interface to already prepared VCA API would require some additional changes that would not be needed if I decided to use GTK+. Despite all these facts, there is a lot of advantages that speaks for Qt:

- Documentation - very advanced with many practical examples
- Class library - optimized and tested class data structures not only for GUI programming
- Developement tools - qmake (for auto-generating of Qt Makefiles), Qt Designer (powerful tool for designing of user interfaces) and many other
- Signal/Slot mechanism - thread-safe feature of Qt, which is very suitable for the

event driven programming as programming of GUI is

The decision of using the Qt library for CAN monitoring tool development brought some issues that had to be reconsidered. VCA is programmed in C, which would not mean any complications generally, but in order to ease the creating of the graphical interface it would be better to reimplement some of the VCA components using the structures provided by Qt. It would simplify further connecting of VCA to the graphical objects and ensure better cooperation between graphical and non-graphical parts of the application.

Chapter 7

QVCA - VCA portation to Qt

In order to ease the programming of the graphical application, there was decided to reimplement some parts of VCA library. This reimplementation meant portation of VCA library to Qt environment. More options for porting of VCA to Qt were considered. From the small modifictaions, concerning the change of some data structures and mechanisms, to the large ones, that would require complete reprogramming of VCA, using the mechanisms and structures provided by Qt. At last, I had decided for the compromise between these two options - complete reprogramming of those parts of VCA, that will closely cooperate with the graphical part of our CAN monitoring application. According to the Trolltech recommendation to use the letter 'Q' at the beginning of the objects' and applications' names developed in Qt, I decided to change the name of VCA library to QVCA - *Qt Virtual CAN API.* All suggested changes can be divided into three closely related groups that regard:

- C++ integration of chosen VCA components
- Reimplementation and optimization of VCA's specialized data structures
- Extensions and enhancements

7.1 C++ integration of chosen VCA components

C++ integration of VCA meant the transformation of old *data-oriented* (or process-oriented) C-based structures to the *object-oriented* ones. The goal of object-oriented approach is to make the system elements more reusable. In addition, this approach improves system quality and the productivity of its analysis and design. All objects in

C++ are represented as instances of *classes*. **Class** is the base C++ data structure. It defines particular object abstract characteristics and behaviour. QVCA defines several different types of classes. Each of them was implemented as **QObject**. It means, that they inherit QObject class, which is the base class of all Qt classes. QObject is the heart of the whole Qt object model. Very advanced and a probably the most powerful QObject feature is a mechanism for the object communication called **signals and slots**.

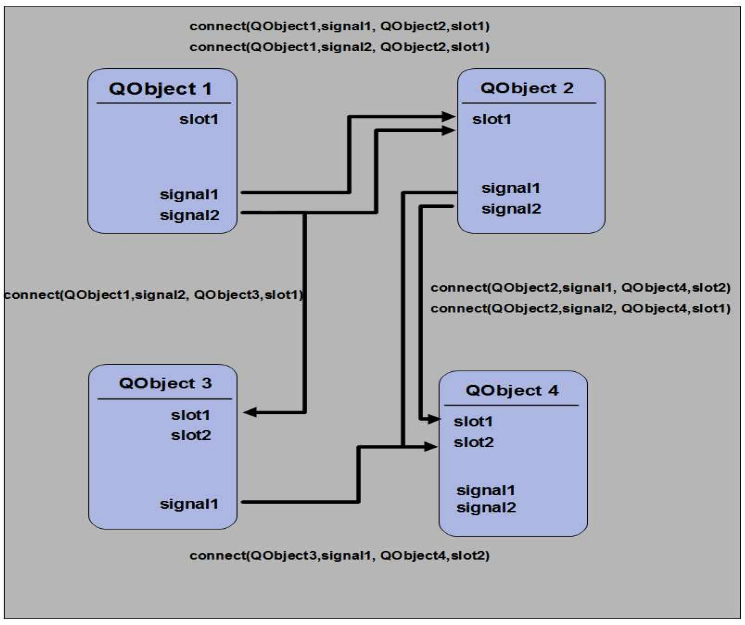

connect(QObject1,signal1, QObject2,slot1)
connect(QObject1,signal2, QObject2,slot1)
QObject 1
slot1
signal1
signal2
QObject 2
slot1
signal1
signal2
connect(QObject1,signal2, QObject3,slot1)
connect(QObject2,signal1, QObject4,slot2)
connect(QObject2,signal2, QObject4,slot1)
QObject 3
slot1
slot2
signal1
QObject 4
slot1
slot2
signal1
signal2
connect(QObject3,signal1, QObject4,slot2)

Figure 7.1: Signals and slots mechanism

The signal/slot mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. *Signal* is emitted when a particular event occurs. Qt's objects have a lot of predefined signals, but a programmer can simply subclass any QObject and add his own signals to it. *Slot* is a function that is called in response to the particular signal. There is a lot of predefined slots for particular Qt objects, but a programmer can create its own ones and connect them to the particular signals. Several slots can be connected to several signals as far as the signal signature matches the signature of the receiving slot. In fact a slot may have a shorter signature than the signal it receives, because it can ignore extra arguments. For better understanding of signal/slot mechanism look at figure 7.1.

The signal/slot mechanism is, in comparison to the callback mechanism used in VCA, **type and thread safe**. When using the callback functions, one can never be sure that the processing function will call the callback function with the correct arguments. Other drawback is that, the callback function is strongly coupled to the processing function, since it must know which callback function to call. In Qt, a QObject class which emits a signal neither knows, nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. More information on QObject and Signal-Slot mechanism can be found in Trolltech online documentation[10].

QVCA implementation provides several classes that were developed from the structures offered by VCA library. We can divide these newly designed classes to three groups:

- **CAN interface classes** - provide the connection to the LinCAN driver and enable sending and receiving of CAN messages
- **CAN network classes** - provide CAN message distribution similar to the one implemented in VCA
- **CANopen classes** - provide CANopen protocol support

7.1.1 CAN interface classes

Basic requirement for the implementation of CAN interface classes was, that they should provide a unified connection to the LinCAN driver. Apart from the possibility of the local connection, QVCA offers a possiblity of the remote connection. In order to unify both ways, there had to be designed an *abstract class*, from which the particular interface classes would be derived. I had decided to call the abstract class `QVcaBusIfcAbstract` and the derived classes `QVcaBusIfcLinCAN` resp. `QVcaBusIfcSocketPickle`. All mentioned classes were designed as *QObject*s to enable their communication using of *signal/slot mechanism.*

7.1.1.1 QVcaBusIfcAbstract

`QVcaBusIfcAbstract` class provides only the definitions of basic CAN interface programming features. It defines the methods that enable *sending* and *receiving* of CAN messages, as well as the signals and slots for CAN message *distribution* to different parts of QVCA library.

7.1.1.2 QVcaBusIfcLinCAN

`QVcaBusIfcLinCAN` class is the interface class that is used for the local connection to the LinCAN driver. By the local connection I mean binding the instance of this class to a device file (usually /dev/can#), present on a local machine, through which the actual monitoring is performed. Binding the QVcaBusIfcLinCAN to the device file is implemented using Qt **QSocketNotifier** class, that performs the monitoring of the activity on particular device file descriptor. This solution replaced the original, quite complicated solution of VCA, based on using the `select` function.

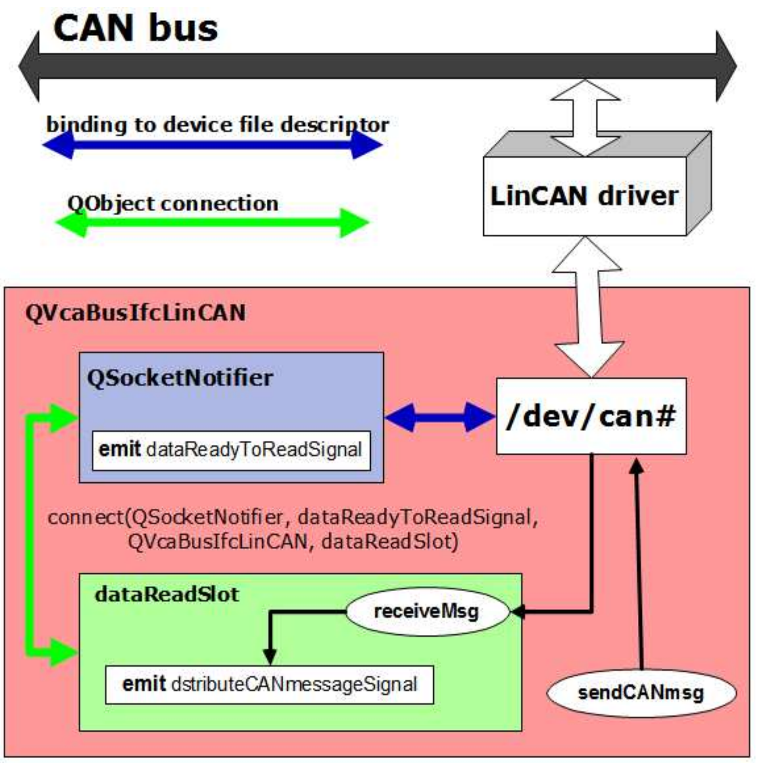

CAN bus
binding to device file descriptor
QObject connection
LinCAN driver
QVcaBusIfcLinCAN
QSocketNotifier
emit dataReadyToReadSignal
/dev/can#
connect(QSocketNotifier, dataReadyToReadSignal, QVcaBusIfcLinCAN, dataReadSlot)
dataReadSlot
receiveMsg
emit dstributeCANmessageSignal
sendCANmsg

Figure 7.2: CAN bus monitoring using the local connection to LinCAN driver

Figure 7.2 shows the actual monitoring of CAN bus using QVcaBusIfcLinCAN class for the local connection to the LinCAN driver. When there is some data available for reading on device file descriptor used for monitoring, *QSocketNotifier* emits a signal which is then caught by *QVcaBusIfcLinCAN* slot connected to it. In this slot, new CAN message is received using the `read` function called on the file decriptor to which QSocketNotifier is bound, and then immediately emitted with *QVcaBusIfcLinCAN* CAN message dsitribution signal. If a programmer decides to do some other processing with the received message, he can just simply connect some other QObject's slot(s) to this signal. Sending of CAN message is performed the very same way that is implemented in VCA - using the `write` function, called on particular CAN device file descriptor.

7.1.1.3 QVcaBusIfcSocketPickle

`QVcaBusIfcSocketPickle` is the interface class that provides the mechanisms for the remote connection to the LinCAN driver. It enables to connect to the **canmond** daemon through the **QTcpSocket** (Qt class that provides TCP socket). It is assumed that canmond is running on some remote machine on the internet and performs the actual monitoring of CAN bus. Canmond is actually a simple TCP/IP server that resends all received CAN messages to all clients connected to it.

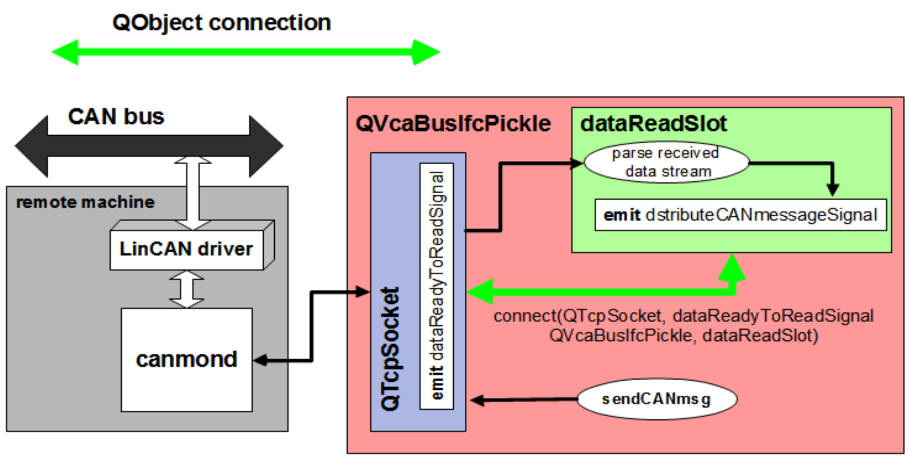

QObject connection
CAN bus
remote machine
LinCAN driver
canmond
QVcaBusIfcPickle
dataReadSlot
parse received data stream
emit dstributeCANmessageSignal
QTcpSocket
emit dataReadyToReadSignal
connect(QTcpSocket, dataReadyToReadSignal QVcaBusIfcPickle, dataReadSlot)
sendCANmsg

Figure 7.3: CAN bus monitoring using remote connection to LinCAN driver

Figure 7.3 shows the actual monitoring of CAN bus using the QVcaBusIfcSocketPickle class for the remote connection to the LinCAN driver. When a new data is available on the socket for reading, QTcpSocket emits the signal, which is being caught by corresponding *QVcaBusIfcSocketPickle* slot connected to it. This slot provides the mechanism that parses CAN message from received data stream and then emits it with the signal. Again, if a programmer decides to do some other processing with the received message he can just simply connect some other QObject's slot(s) to this signal. Sending the CAN message is implemented using *QTcpSocket* `write` data methods.

7.1.2 CAN network classes

This group contains only two classes: `QVcaNet` and `QVcaNetId`. *QVcaNet* class simulates monitored CAN network. It can contain several *QVcaNetId* objects, that represent actual CAN nodes connected to this network and offer a possiibility of setting-up acceptance-filtering masks. All CAN network nodes and acceptance-filtering masks are kept in `QVcaNet` specialized data structures satisfying several requirements regarding the speed of the manipulation with the items stored in them. *QVcaNet* represents a parent object in the whole QVCA parent-children hierarchy. When it is destroyed, all its children objects are destroyed in its destructor, too. QVCA architecture is shown at figure 7.4.

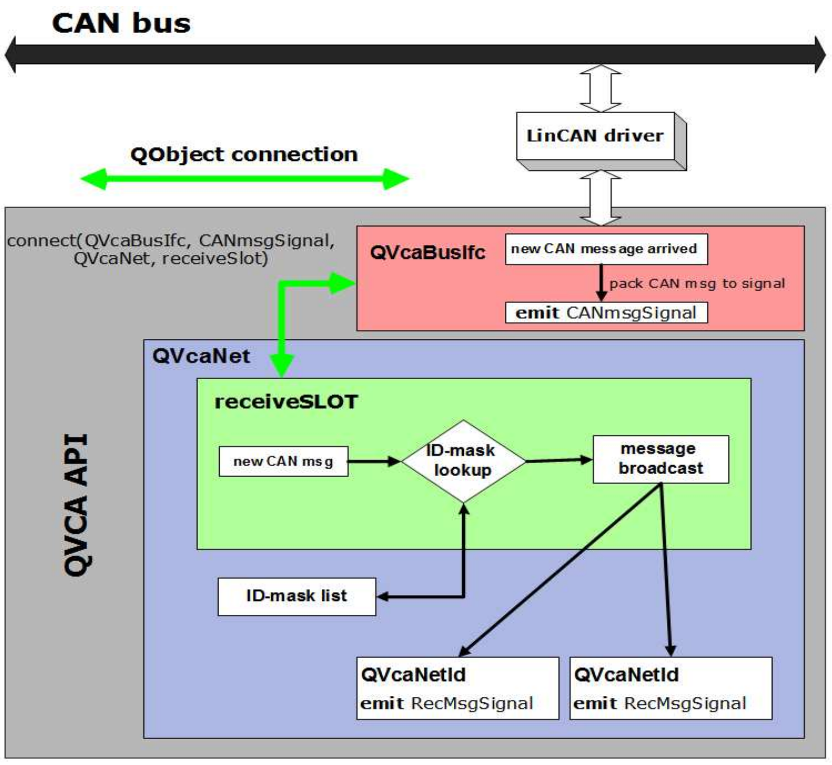


Figure 7.4: QVCA architecture and message distribution

Original VCA architecture of CAN message distribution has not been changed. Only the mechanism that was implemented using callback functions was replaced by the signal-slot mechanism. This way we get many advantages, plus the source code that implements these mechanisms gets more readable. *QVcaNet* object broadcasts all arriving messages to all present *QVcaNetId* objects using its particular signals. *QVcaNetId* objects can either accept broadcasted messages or refuse them according to their set-up masks.

7.1.3 CANopen classes

This group will be discussed very shortly, since the programmed application does not provide the support for CANopen protocol, yet. VCA provides advanced CANopen API so I decided to reimplement at least some of its parts that will ease future CANopen support. The base class of this group is `QVcaCanOpenNode` that represents an actual CANopen node in particular *QVcaNet* CAN network. It contains `QVcaSDOFsm` object that simulates CANopen final state machine. *QVcaSDOFsm* is just a C++ wrapper for the actual final state machine that is implemented in different source file and left unchanged. Apart from the reference to `vcasdo_fsm_t` (original VCA implementation of final state machine) object, `QVcaSDOFsm` class keeps the list of its SDO requests. They are implemented as `QVcaSDORequest` objects.

7.2 Specialized data structures

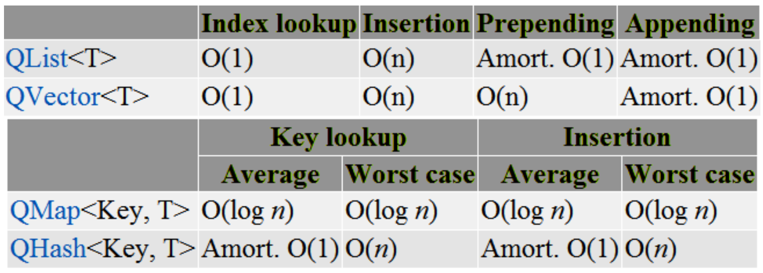
Very important issue I had to deal with when I was reimplementing VCA library, was the storage of CAN network IDs and masks, as well as the storage of CANopen SDO requests. Data structures, that would keep the mentioned items should provide fast lookup, adding and removing of an item. I had considered many possible options that would satisfy these requirements:

- Preserve the original VCA container solution
- Design specialized template structures
- Use the templates from STL (Standard Template Library)
- Use Qt generic containers

Each of the considered options would bring its advantages and disadvantages. Preserving the original solution would mean the use of optimized and already tested structures. It would eliminate one dereference in each access and simplify many operations including type conversion hiding. In addition, the library would remain independent from the application graphical interface. This choice would not be suitable for suggested C++ integration, so it was rejected right after I decided for the portation of VCA to Qt. Design of new specialized templates would mean creating the structures that would easily use the nesting of node structures to the objects. This solution would lead to the faster code writing, but slower optimization.
Using the templates from STL would lead to the previous problem. Plus, STL does not contain environment good enough for our needs, so there would still be need for creating of new templates.
Using the Qt generic containers would mean a dependance on graphical library, but I had accepted this constraint when I decided for the portation VCA to Qt.

7.2.1 Qt generic containers

The Qt framework provides a set of template-based container classes. They can be used to store items of a specified type. Container classes are designed to be lighter, safer, and easier to use than the STL containers. They are implicitly shared (method that maximizes resource usage and minimizes copying of data), fully reentrant, and they are optimized for speed, low memory consumption, and minimal inline code expansion, resulting in smaller executables. In addition, they are thread-safe in situations where they are used as read-only containers by all threads used to access them.



	Index lookup	Insertion	Prepending	Appending
QList<T>	O(1)	O(n)	Amort. O(1)	Amort. O(1)
QVector<T>	O(1)	O(n)	O(n)	Amort. O(1)

	Key lookup		Insertion	
	Average	Worst case	Average	Worst case
QMap<Key, T>	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)
QHash<Key, T>	Amort. O(1)	O(*n*)	Amort. O(1)	O(*n*)

Figure 7.5: Algorithmic complexity of Qt generic container classes

From all the container classes offered by Qt I was considering four of them as a substitution of original VCA constructs. As for the GAVL/AVL trees I was considering

`QHash` or `QMap`, and for the GSA lists, `QVector` or `QList`. At last, I had decided to use `QMap`s and `QList`s. `QHash`, in comparison to `QMap`, offers faster lookups, since it stores its data in an arbitrary order, but `QMap` stores the data in key order, what can be sometimes very useful. `QVector` stores an array of values at adjacent positions in memory. Inserting at the front or in the middle of a vector can be quite slow, because it can lead to a large number of items having to be moved by one position in memory. That is why I decided to use `QList`.

Summarizing table 7.5 shows the algorithmic complexity of Qt generic container classes. It can be assumed that chosen solution will sufficently fulfil all requirements for the new storage structures and won't cause any noticeable inconvenience.

7.3 Extensions and enhancements

Apart from the changes that enabled portation of VCA to QVCA, I had suggested and implemented some useful features that simplify connecting of QVCA to the application graphical objects. These features represent an imaginary bridge between the graphical and non-graphical part of the application. I have decided to keep them as a part of QVCA - independently of the graphical part as it is required when one wants to use Qt's **model-view programming** technics that simplify the presenting of the data to he user.

7.3.1 Qt model-view programming

Qt introduces new architecture to manage the relationship between the data and the way it is presented (displayed) to the user. This architecture is called *model-view programming*. This approach enables to keep the data independent from the graphical part, which gives developers greater flexibility to customize the presentation of item, and provides a standard model interface to allow a wide range of data sources to be used with existing item views (classes that enable display of model's data to the user).

The model communicates with a source of data that provides an interface for other components in the architecture. The view obtains model *indexes* (references to data items) from the model. By supplying model indexes to the model, the view can retrieve items of data from the data source. A delegate renders the items of data. When an item is edited, the delegate communicates with the model directly using the model indexes. Further information on model-view programming can be found in online documentation[10]. For better understanding of this approach look at figure 7.6.

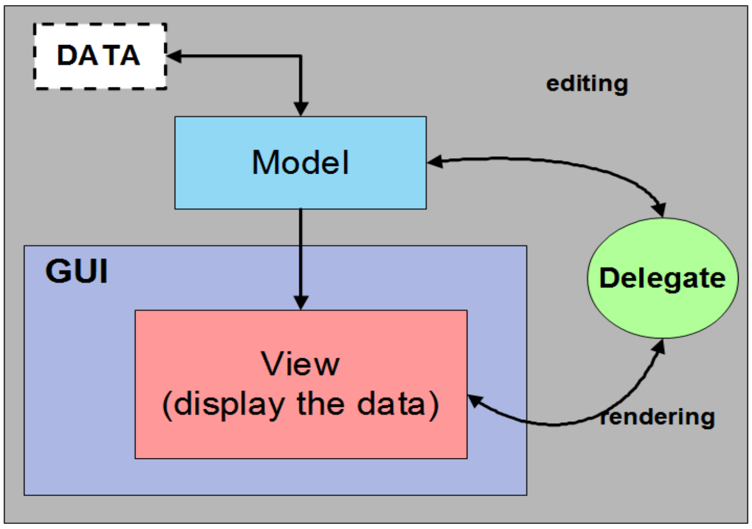


Figure 7.6: Model-view programming

To use *model-view programming* technics, there had to be designed a model that would store the monitored data and provide it to the view components of the application graphical interface. More important than the actual implementation of the model is what data it stores (monitored CAN messages) and how. In addition, possible multithreaded architecture of developed application had to be taken in mind. To deal with the minimizing of copying of data, maximizing of resource usage and multithreading, there was designed a new object that fulfilled all mentioned requirements. Qt offers the resources to enable creating such an object - **Implicitly Shared Classes**.

An implicitly shared class consists of a pointer to a shared data block that contains a reference count and the data. When a shared object is created, it sets its reference count to 1. The reference count is incremented whenever a new object references the shared data, and decremented when the object dereferences the shared data. The shared data is deleted when the reference count becomes zero. The benefit of sharing is, that a program does not need to duplicate data unnecessarily, which results in lower memory use and less copying of data. Implicitly shared classes can be safely copied across threads, like any other value classes. They are fully reentrant, which is implemented using atomic reference counting operations. Atomic reference counting is very fast, much faster than using a mutex. Even in multithreaded applications, you can safely use them as if they were plain, non-shared, reentrant classes. Further information can be found in Qt implicitly shared classes documentation[10].

Apart from implementing CAN message as implicitly shared object, I decided to use the very same way for SDO requests implementation, too. Although SDO request objects will not be used in this work, it will ease further extension of the application to support CANopen protocol. QVCA CAN message model was designed as an abstract class with one virtual method - CAN message receiving slot. This was done to enable the use of this model for different kinds of message processing tasks in different graphical widgets that will eventually use it.

7.4 Conclusion

Portation from VCA to QVCA turned out to be the right solution after all. It brought many useful features. Signal-slot mechanism enabled simple communication between the QVCA objects. Model-view programming separated graphical and non-graphical part of the application and concurrently created a bridge between them. Qt template structures replaced original VCA constructs without any noticeable inconvenience and made the source code more readeble. At last, implicit sharing improved application memory and resource usage and simplified possible multitheading issues.

Chapter 8

QCANalyzer design and implementation issues

QCANalyzer is a graphical application that allows a user to interact with the CAN bus. It is a simple CAN monitoring tool, that enables to analyze monitored CAN traffic either from a live CAN network, or from a previously saved traffic files. Instead of offering typed commands, it gives a user possibility to use its graphical elements, generally called *widgets*, to reach the functionality that the application provides. Application architecture, as well as its implementation, depends on functionality requirements that had been determined before the beginning of the implementation process:

- CAN network traffic display
- Sending of CAN messages
- Filtering of received and displayed CAN messages
- Plot display of monitored data values
- Miscellaneous

This chapter will discuss QCANalyzer architecture and the implementation of its particular parts with regard to the specified requirements.

8.1 QCANalyzer architecture

Proposed QCANalyzer architecture keeps the implementation of its graphical objects independent from the QVCA library that the application will use for monitoring. Therefore, it can be divided to two basic parts:

- non-graphical part - it takes care of CAN bus monitoring
- graphical part - it takes care of display of monitored traffic

Each of these parts is running in a different **thread**. *Thread* is a way how a program forks (splits) itself into two or more simultaneously running tasks. Threads inside one process share some memory while two different processes don't. Thread programming is very capable way how to implement a graphical application. It preserves the freezing of the user interface and thus improves user interaction with the application, while some other computation is running in the background. Moreover, multithreading increases the program performance and lowers memory consumption. Qt provides quite an advanced multithreading support. Apart from platform-independent classes, it offers a thread-safe way of communication across the threads - the *signal/slot mechanism.*

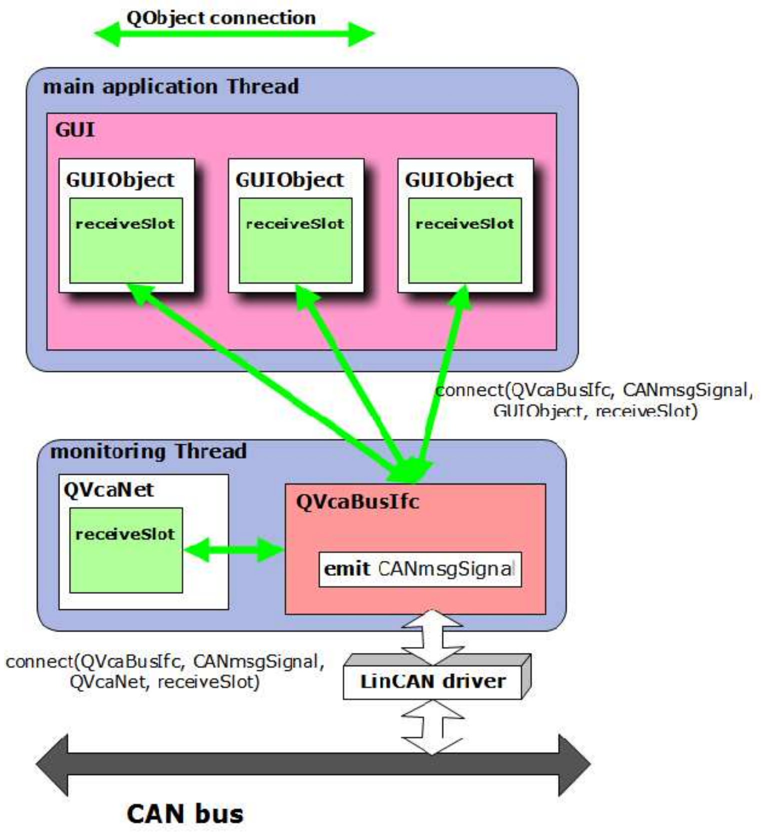


Figure 8.1: QCANalyzer architecture

In a following text, non-graphical thread, that takes care of actual CAN traffic monitoring, will be refered to as *monitoring thread*, and the graphical thread, in which the graphical part of the application is running and which allows a user to watch the monitored traffic, *main application thread.* Monitoring thread is created and then immediately started from the main application thread using particular QCANalyzer widget. Thread creates QVCA network as well as QVCA interface objects that perform actual CAN bus monitoring and CAN message distribution to different parts of the application in a way that was described in previous chapter. QVCA interface object signals, that distribute received CAN messages, are caught by corresponding slots in main application thread, which performs their further processing. Described mechanism is shown at figure 8.1.

8.2 Implementation

Base class of the whole QCANalyzer application is the class that implements the *application main window.* It is called **MainWindow** (MW) and consists of several other objects. Instance of MW is the parent object of all objects in the whole application parent-children hierarchy. It has to be underlined, that the terms *application main window* and *MainWindow* have different meanings in this text: application main window represents *graphical object* that is displayed to the user when the application is started, while MainWindow represents the class that defines and implements application main window functionality. Design and implementation of MW, as well as all components it consists of, will be discussed on following lines in more detail with regard to particular application functionality requirements.

8.2.1 CAN network traffic display

Monitored CAN message traffic is displayed in QCANalyzer main window. It was designed in *Qt-designer* - tool for designing and building GUIs from Qt components. Output of Qt-designer is a customized XML file called *user interface form.* With Qt's integrated **qmake** tool, the code for user interface created with Qt Designer is generated automatically when the rest of the application is built. Forms can be included and used directly from the application, or they can be used to extend standard or user customized widgets. MW inherits Qt-designer generated form and implements the functionality of each graphical object that it contains. This way the implementation of MW graphical objects is hidden in specialized file.

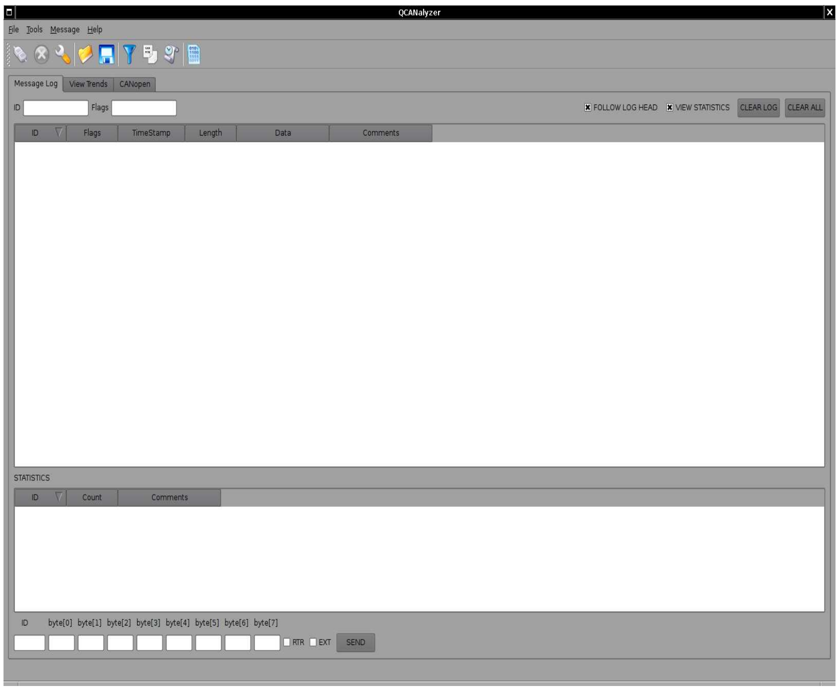


Figure 8.2: QCANalyzer main window

Application main window is shown at figure 8.2. It is organized as a tab widget with two tab pages. First tab page provides a traffic view, second one provides QCAN plotter tool. Monitored traffic is shown in a CAN message *table view*. It displays one particular message on one row. Every row is is divided to several columns. Each column displays different CAN message field (ID, flags, etc.). In addition, application main window provides another table view, that shows the statistics for each received CAN message ID.
The mechanism that enables to display CAN bus monitored traffic consists of three steps:

- CAN message reception and distribution
- CAN message filtration
- CAN message display

CAN message reception and distribution is ensured using the mechanisms implemented in QVCA library, which QCANalyzer uses to capture the monitored data (CAN messages). QVCA CAN interface object signals that distribute received CAN messages from the CAN bus to different components of the application, are caught by the slot of *MW filtering object*, which performs the filtering of captured before its actual display to a user. In this slot, user unwanted messages are rejected (will not get displayed in a traffic view). The ones that do not get filtered are immediately emitted by another signal, that is caught by *MW CAN message model* slot. In this stage of processing, new CAN message is appended to the model's QList, so all messages are stored in the order they were received from the CAN bus. MW CAN message model then provides stored messages to the application main window's table view for display. For better understanding of this mechanism look at figure 8.3

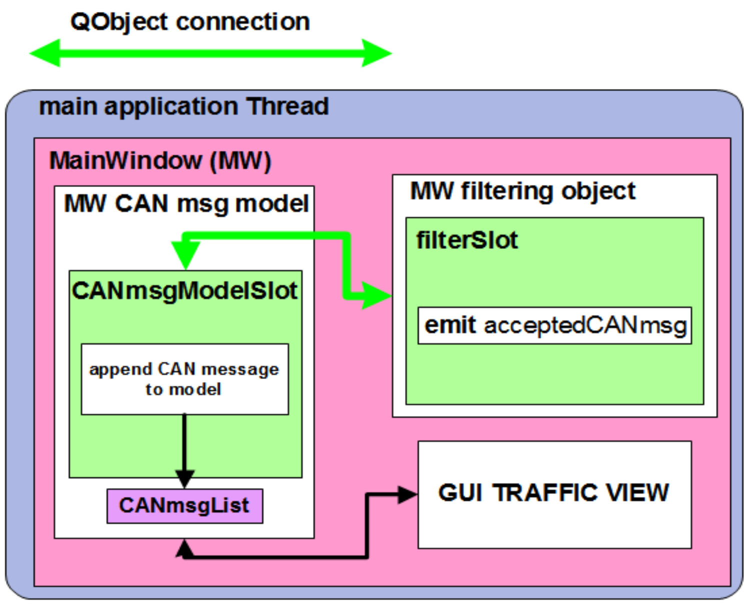


Figure 8.3: QCANalyzer CAN message flow

Apart from the MW CAN message model slot, signal emitted by the MW filtering object is also caught by a slot of another MW object, which takes care of createing message ID statistics. CAN message statistics are displayed in a different table on the same tab page as the monitored traffic . This require design of a new model, which unlike the CAN message model, stores its data in QHash (Qt generic container similar to QMap). ID of new arriving CAN message is checked if the statistics model does not already contain it. If not, it is added to the model's data QHash and its count is set to one. If the model already contains it, it increments its count by one.

8.2.2 Sending of CAN messages

QCAnalyzer offers two possibilities for sending CAN messages to the CAN bus:

- Sending of a single CAN message
- Sending of CAN message sequences

CAN message(s) can be sent only when the monitoring thread is running - since the QVCA interface objects, that allow the user to do so, are created with the start and destroyed with the stop of monitoring thread. Each message sent to the CAN bus is immediately enqueued to *sent message history model* used by QCANalyzer *Sent Message History Tool*, that allows a user to resend or to keep the track of already sent messages. In addition, this tool provides some other functionality that will be described later in this chapter.

8.2.2.1 Sending of a single CAN message

Sending of a single CAN message can be done in two different ways. User can send an instant CAN message from the application main window using the widgets placed at the bottom of it. Another possibility is to use the application *Message Sequence Tool*.

8.2.2.2 Sending of CAN message sequences

QCANalyzer allows a user to send CAN message sequences to the CAN bus using the specialized tools:

- CAN message sequence tool
- Sent message history tool

Both tools were designed in Qt-designer and both offer same set of widgets with slightly different functionality. Both offer a possibility to save the messages to a file and open previously saved files. After opening they can be used the very same way as it was described on previous lines.

8.2.2.3 CAN message sequence tool

CAN message sequence tool is a primary tool used for sending CAN message sequences. It allows a user to prepare CAN messages that can be later sent to the CAN bus. Messages are displayed on the rows and divided to particular columns as it is done in QCANalyzer main window traffic view. Unlike the MW CAN message model that uses CAN message model implemented by QVCA library, tool implements its own model. In comparison to the QVCA message model, it offers the possibility to edit model's items and drag and drop of item data to different widgets that allow it. In order to enable these item properties, there had to be set-up some special features to this model. Each model item has a number of data elements associated with it, and each of these can be retrieved by specifying a role to the model's `data()` function. Items can be queried with `flags()` to see if they can be selected, dragged, or manipulated in any other way. To enable editing and drag/drop of the items, particular item flags must return editable flags and drag/drop flags for the valid item indexes. Plus, the model must have `setData()` method implemented.

8.2.2.4 Sent message history tool

Sent message history tool is a secondary option how a user can send a message sequence to the CAN bus. By secondary, I mean that this tool was not designed for this purpose and adds this functioniality because of additional user requirements. User can send to the CAN bus only the messages that this tool displays. So it offers only the possibility of **resending** of messages that had already been sent to the CAN bus by either of mentioned options - application main window or message sequence tool. Model, that this tool implement, does not allow a user to edit particular items, but it enables dragging its items data to different application widgets that allow it (i.e. CAN message sequence tool).

8.3 Filtering of received and displayed CAN messages

The notion of filtering covered implementation of two different filtering objects in this application. Although they may seem the same, they cover two different tasks:

- Traffic filtering
- View filtering

Traffic filtering gives a user option to filter the captured data that can be displayed to the user, while view filtering allows a user to filter only the view of actually displayed data.

8.3.1 Traffic filtering

To allow a user to filter monitored traffic, there was a specialized *traffic filtering tool (TFT)* designed. Again, it was designed with the Qt-designer and consists of the same group of widgets as the CAN message sequence or CAN message history tool. Main difference between these tools is the model that they implement. TFT model stores the set-up masks in its QList. Actual mask is defined as an implicitly shared object. This tool takes a very important position in the whole application architecture. It allows a user to define several filtering masks with different roles in the whole application. It is this tool's model slot that performs first processing of received CAN messages.

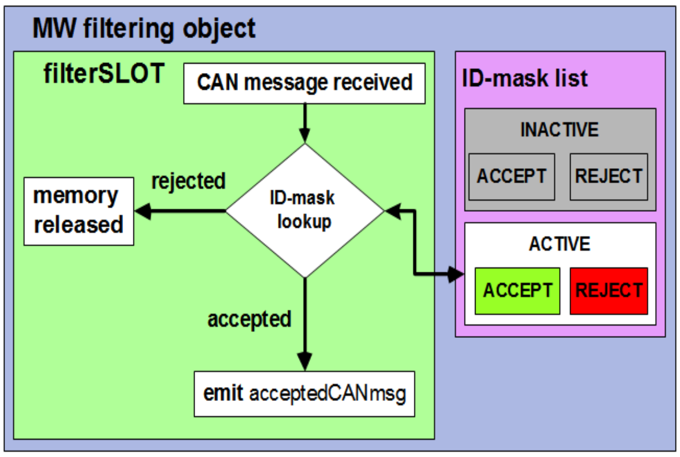


Figure 8.4: QCANalyzer traffic filtering

The pinciple of CAN traffic filtering implemented in TFT can be understood from the figure 8.4. When a new message is received by QVCA interface object, it is emitted by the signal that is connected to TFT model slot. ID of received message is checked against all set-up masks in TFT model. If it passes any of the accept mask compare, it is emitted by another signal that is connected to MW CAN message model slot. Then it is enqueued to its model and displayed to user. If ID of new arriving message doesn't get through any of the *accept masks* or if it passes any of the *reject masks*, it is immediately rejected and not displayed in the main window. Memory allocated for the message is released automatically, since the reference for the implicitly shared object (in this case CAN message) will no more exist.

8.3.2 View filtering

View filtering covers displaying only those items in the view, that match some user filtering conditions or expressions. View filtering is implemented using specialized model offered by Qt - QSortFilterProxyModel. It provides the support for sorting and filtering of data passed between another model and a view. QSortFilterProxy model is applied on already existing model, which is refered to as a source model. QSortFilterProxyModel works as a wrapper for the original source model. It transforms (maps) source model indexes it supplies to new indexes for different views to use. This enables source model to be reorganized as far as the views are concerned, without any transformation of model's data. Plus, the data stored in the original model is not duplicated. Actual view filter is imeplemented using QRegExp object. It can be used as a wildcard pattern, a regular expression or a fixed string. Whenever it changes, a signal that notifies the QSortFilterProxyModel model is emitted . QRegExp is applied to the filter role (display role by default) of each item. Items that don't match filtering expression are hidden in the view. View filtering is implemented in every QCANalyzer widget that provides a table view - main window message log, message (sequence and history) tools and message filtering tool.

User can specify filtering expression by typing it into tool's line edits placed at the top of the view, which reflects applied changes immediately. It had to be reminded again that this mechanism **filters only already displayed data - it won't filter message traffic or any other data**. Data that does not match the expression specified byt the user remains stored in original model during the filtering action.

8.4 Plot display of monitored data

Qt offers a large set of objects that allows a user to create sophisticated scientific widgets. This method of programming the scientific widgets "from the scratch" would require advanced knowledge of Qt toolkit, too much code writing and additional optimization before the widgets could be used in CAN monitoring application. Therefore, I had decided to use the solutions offered by open-source project called **Qwt**. Although this decision will cause additional dependancy on another library, I will get a benefit of already implemented and optimized objects, which will lead to faster code writing.

8.4.1 Qwt library

Qwt is a graphical library built upon Qt. It offers a big set of classes, containers and graphical widgets that can be used in the programs with technical background. Besides a simple 2D plotting widgets it offers scales, sliders, compasses, wheels and many other scientific widgets to control or display the values of type double. Although it was not integrated with the whole framework, Qt-designer provides a support for basic Qwt graphical widgets. Library is distributed under LGPL license and can be used in all environments where Qt can be used. Only inconvenience that it brings is, that Qwt does not provide binary packages. Instead, all source files have to be downloaded and compiled the way any other Qt based program is compiled. More information on Qwt can be found at project's http pages[11].

8.4.2 Plotting of monitored data

Data transferred by CAN messages in monitored CAN network can be displayed in application's main window *QCAN plotter* tab page. This tab page provides drawing of 2D plots. User can plot several curves at once using this tool.Curve that gets displayed using this tool is implemented as a customized Qwt object, which is created for every new added message ID. Message ID and the byte range from which the actual displayed value of the curve is counted, must be specified in order to read the data transferred by particular CAN message.

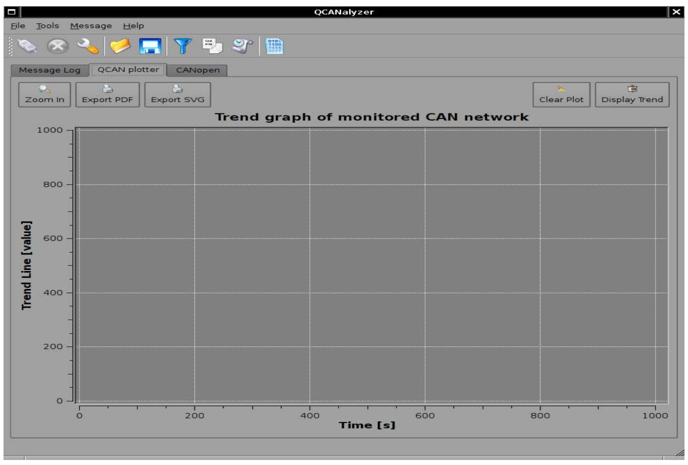


Figure 8.5: QCAN plotter

Data used for drawing the trend plots are gained in the QCAN plotter slot that is connected to the TFT model signal, thus only the data of the messages accepted by TFT can be displayed using this tool shown at figure 8.5. Data transferred by CAN messages are in little endian order. They are transformed by this tool to the correct byte order, transformed to a number and stored in the curve object's data storage. This data storage reimplements another Qwt object that keeps the monitored data in QwtArrays. QwtArray is a structure similar to QLIst. It offers many useful features for the use in the technical programs (for example adding arrays of values apart from adding a single values).

8.5 Miscellaneous

This group consists of the objects that simplify the user interaction with other application widgets or with the application itself. It covers the *configuration dialogs* and *application settings*. By configuration dialogs I mean the dialog that allows a user to config monitoring options and the dialog that alows a user to configure QCAN plotter drawing options. Both dialogs were designed by Qt-designer. Their sginals are connected to particular slots implemented as the methods of MW. Their use will be described in user manual.
Application settings are stored in a special object that inherits one of the most useful Qt object - **QSettings**. QSettings provides platform-independent application settings. QSettings is a wrapper around different OS technologies: Windows stores the application settings in the system registry, MAC OS X in XML preferences files and UNIX systems in INI files. The use of QSettings class is very simple and intuitive - application programmer specifies only a couple *(setting, key)* that are added by QSettings object to the application settings file. All other processing is automatically taken care of behind the scene by QSettings object itself. Application settings are loaded when the programm is executed. Apart from graphical settings of QCANalyzer widgets (to restore position and size of the application main window), they store monitoring session options, too.

Chapter 9

QCANalyzer User manual

QCANalyzer does not contain any installation script yet. It has to be compiled and built directly from the given source codes.

9.1 Compilation an building the QCANalyzer

There are some precompilation prerequisities that need to be satisfied in order to compile the program:

- Qt toolkit must be installed - at least version 4.3
- Qwt library must be downloaded and compiled from its source codes

On some Linux OS distributions (debian, ubuntu), Qt and Qwt is distributed in binary packages, so there is no need for their additional compilation. Qt and Qwt documentation describes how to compile these two libraries using Qt tools on different OS. You can find more information about Qt application deployment in dedicated manual[10]. We assume that user has installed and deployed Qt and Qwt libraries as it is described in their documentation. Now the source files of the QCANalyzer application distributed in one tar file have to be downloaded and compiled. In order to compile the application read the following installation steps:

1. Generate a Makefile using qmake utility this way: **qmake -o Makefile src.pro**
2. Run **make** utility

Compilation should pass without any warnings. Built application binary file called **qc-analyzer** can be found in **bin** project directory. Now the user can copy the application binary to his directory or it can be executed from the directory where it was built.

9.2 Basic functionality

QCANalyzer can be started either by double click on the program executable or from the command line. When a user starts the application from the command line he can specify configuration file, from which the application reads its settings. Configuration file is specified as a program argument. If the specified file does not exist, application will load last used settings. If the configuration file with last settings does not exist, default application settings are loaded.

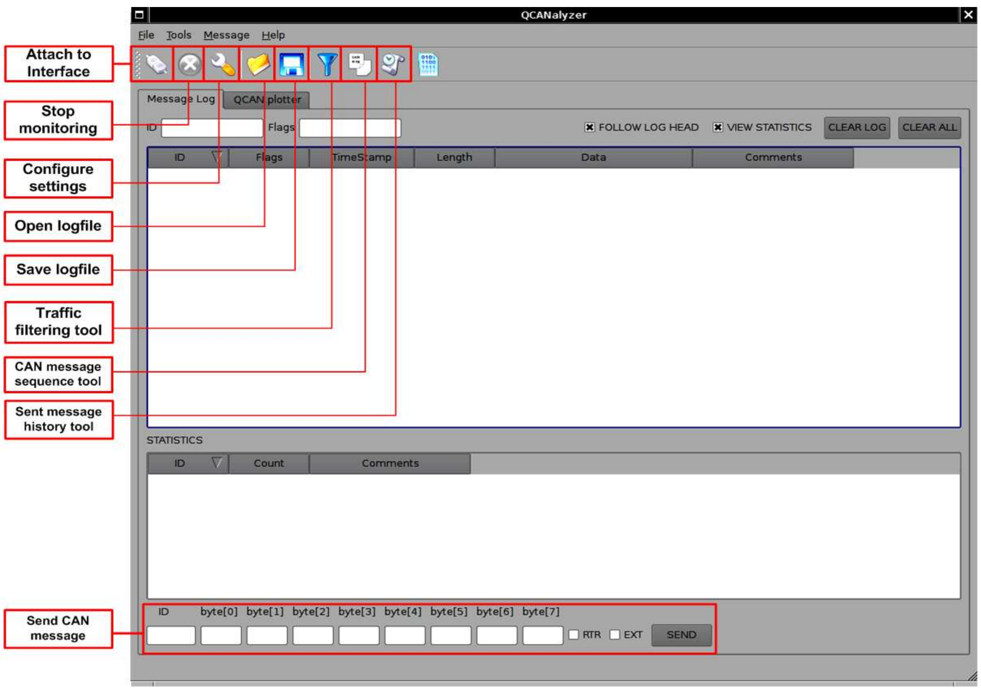


Figure 9.1: QCANalyzer main window

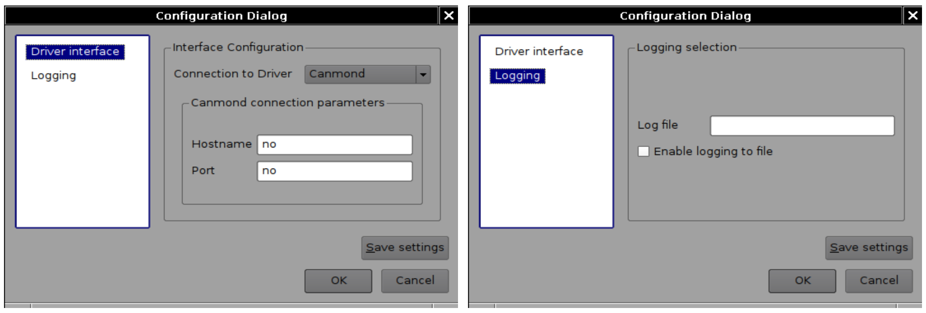
9.2.1 Connection to the LinCAN driver

Before the user starts to monitor the CAN bus, he should check or configure the settings for actual monitoring session. This can be done using the QCANalyzer configuration dialog. Dialog is opened either by clicking particular icon in toolbar or by clicking *"Con-*

figure Settings" in *"Tools"* menu from the application main window menu bar, or using the configuration shortcut - **Ctrl+D**. Configuration dialog allows a user to edit following monitoring session settings:

- Driver interface connection
- Logging

User can configure the connection of the application to the LinCAN driver through manipulating the dialog's widgets. Application connects by default to *canmond* daemon, which supposingly runs on remote machine to which the application is connecting over TCP/IP. If a user specifies CAN device file in configuration dialog, local connection will be used no matter what settings are typed in *canmond* connection configuration. If the name of specified CAN device file does not start with the string */dev/can* or if it can device configuration is set to **no**, application automatically connects to *canmond* daemon with the connection parameters specified in configuration dialog shown at figure 9.2.


Configuration Dialog

Driver interface
Logging

Interface Configuration
Connection to Driver: Canmond
Canmond connection parameters
Hostname: no
Port: no

Save settings
OK
Cancel

Configuration Dialog

Driver interface
Logging

Logging selection
Log file:
☐ Enable logging to file

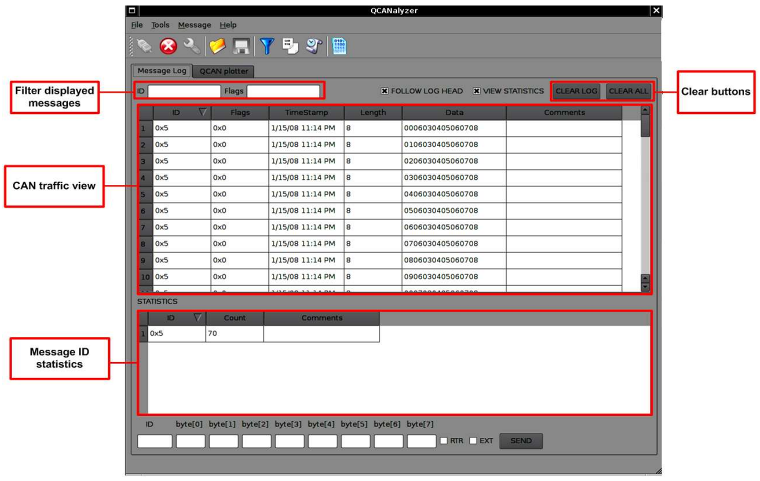
Save settings
OK
Cancel

Figure 9.2: Configure connection to LinCAN driver

Apart from specifying of the connection to the LinCAN driver, user can choose if he wants to save automatically all monitored traffic to, by him specified, log file. This option is disabled by default due to possible accumulation of this file. This option is useful for short time monitoring. Saved file can be later opened and manipulated in the application. User can specify the name of the logging file by checking the particular configuration dialog checkbox and typing the name of the file to the *"Log file"* line edit. Specified settings can be saved by clicking the *"Save settings"* dialog button. If a monitoring of the CAN bus is in progress user can't change any of the described settings.

9.2.2 Monitoring

When the connection to the LinCAN driver is configured correctly, user can start monitoring either by clicking the *"Attach to interface"* button int application mainwindow toolbar or by clicking the *"Connect"* in *"Tools"* menu, or using the predefined keybord shortcut **F5**. If there is some traffic on monitored CAN network it is immediately displayed in application main window message table view if it passes set-up traffic filters. Statistics table view right under the message log shows the number of received messages of particular message ID.



	ID	Flags	TimeStamp	Length	Data	Comments
1	0x5	0x0	1/15/08 11:14 PM	8	0006030405060708	
2	0x5	0x0	1/15/08 11:14 PM	8	0106030405060708	
3	0x5	0x0	1/15/08 11:14 PM	8	0206030405060708	
4	0x5	0x0	1/15/08 11:14 PM	8	0306030405060708	
5	0x5	0x0	1/15/08 11:14 PM	8	0406030405060708	
6	0x5	0x0	1/15/08 11:14 PM	8	0506030405060708	
7	0x5	0x0	1/15/08 11:14 PM	8	0606030405060708	
8	0x5	0x0	1/15/08 11:14 PM	8	0706030405060708	
9	0x5	0x0	1/15/08 11:14 PM	8	0806030405060708	
10	0x5	0x0	1/15/08 11:14 PM	8	0906030405060708	

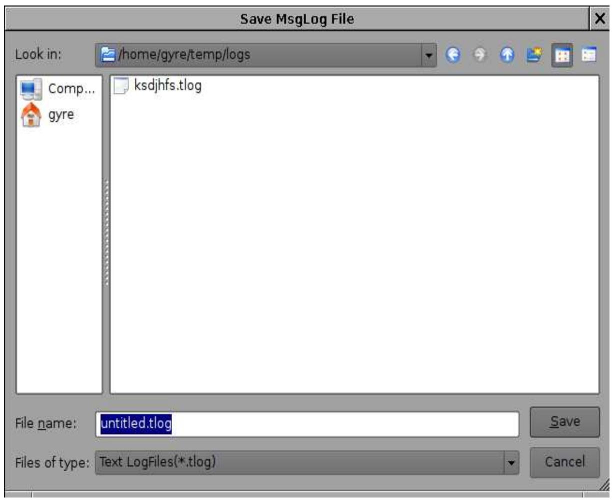
	ID	Count	Comments
1	0x5	70	

Figure 9.3: QCANalyzer in use

Screenshot of running application is shown at figure 9.3. CAN message fields shown in particular column of the application main window traffic view are displayed in hexadecimal form. Displayed traffic view can be cleared using the *"CLEAR LOG"* button placed above the main traffic table view. After this action no messages will be displayed in a traffic view. Message ID statistics remain unchanged. If a user decides to clear both views - traffic and statistics views - he must click the *"CLEAR ALL"* button that is next to the previously mentioned button. User can even hide monitored statics unchecking by defaeult checked *"VIEW STATISTICS"* checkbox. User comments can be added to every monitored message and to every message ID statistics. User can filter the displayed messages by typing the message ID or message flags filtering condition to prepared filtering line edits placed at the top of main window. While the application is connected to the driver, user can send single message using the main window widgets placed at the bottom of traffic tab page. He can specify the message ID either in decimal or in hexadecimal form. When typing the hexadecimal number, user must add **"0x"** prefix before the actual hex value, so the application could recognize value form. Message is sent using the *"SEND"* button. Monitoring can be stopped by clicking the *"Detach interface"* toolbar button or from the *"Tools"* menu, or using the keyboard shortcut **F6**. This button is by default disabled when the monitoring is not in progress. Clicking it will exit the monitoring thread, so no other message will be displayed in message log.

9.2.3 Saving and Opening monitored traffic files

When the monitoring is stopped, user can save the displayed messages to the file - either to text or to binary file. Saved *text files* have **".tlog"** suffix added while the *binary files* have **".blog"**, so they can be recognized when a user wants to open any of the log files in some text or hexa editor. User can save the message log in a similar way as previous actions were described: clicking the *"Save"* tool bar button, using the *"File"* menu or using the usual save keyboard shortcut "**Ctrl+S"**. Saving the log files is disabled when the monitoring of the CAN bus is in progress.



Look in: /home/gyre/temp/logs

ksdjhfs.tlog

File name: untitled.tlog

Files of type: Text LogFiles(*.tlog)

Figure 9.4: Save message log dialog

Apart from saving the monitored log to the file, application provides a possibility to open already saved log files. Open files is enabled through the similar dialog window as the save dialog one shown at figure 9.4. It can be called similar way as the save action is. Both dialog offer the filtering option of displayed files - user can display either text-saved or binary-saved messages. Files are opened in new tab page inserted after the QCAN plotted tab page. Opened message log is displayed in a table view. User can't edit any of the items in it, but he can drag the item data to different widgets that allow dropping of dragged data. Even the application logging files specified in the application settings can be opened and manipulated this way. Closing of opened message log is done by clicking the *"CLOSE"* button placed at the top of open log tab page.

9.2.4 QCANalyzer specialized tools

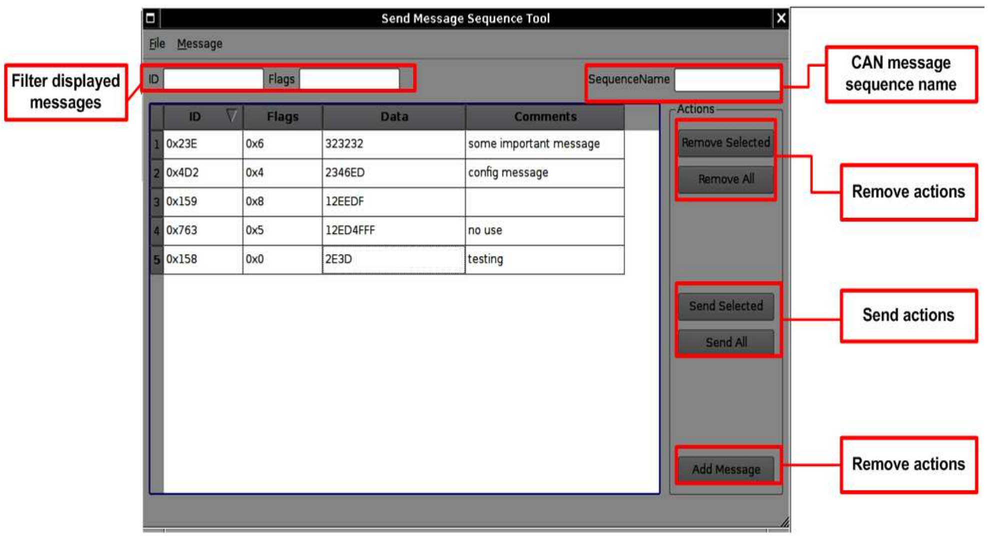
In addition to previously described functionality, application offers some specialized tools:

- CAN Message sequence tool
- Sent message history tool
- Traffic filtering tool

Implementation of each tool was described in previous chapter. This section will show how to use these tools in the application.

9.2.4.1 Message sequence tool

Message sequence tool is used for sending CAN message sequences. It allows a user to prepare messages and then send them to the CAN bus. User can send unlimited count of messages. Each newly created message is appended at the bottom row of tool's table view. Tool allows a user to specify message ID and message flags either in decimal or hexadecimal form as it is done in main window. Again, if a user wants to specify any of mentioned fields in hexadecimal mode he has to put "0x" prefix before the actual number. Data field is automatically converted to hex form. User can decide either to send all messages, or only the ones selected by the mouse. Messages are being sent in the order they are displayed in tools's view (as it is shown at figure 9.5) or in a way they were selected by the mouse.



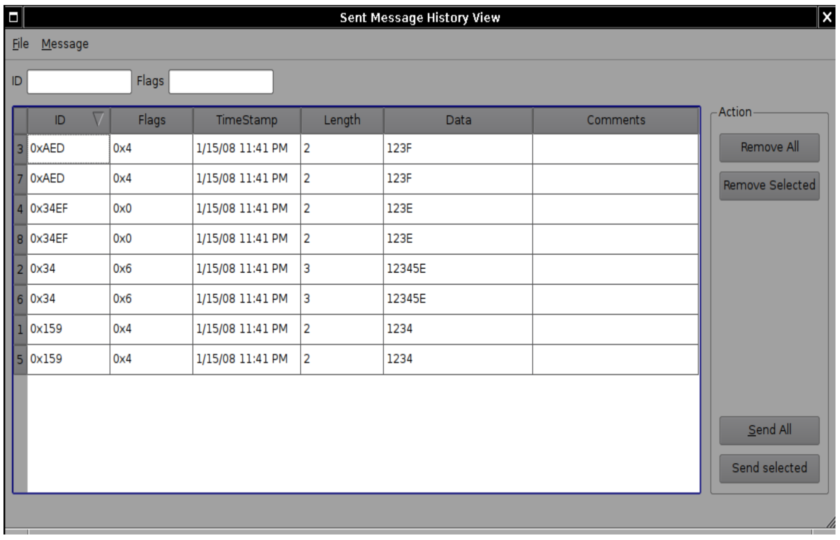
	ID	Flags	Data	Comments
1	0x23E	0x6	323232	some important message
2	0x4D2	0x4	2346ED	config message
3	0x159	0x8	12EEDF	
4	0x763	0x5	12ED4FFF	no use
5	0x158	0x0	2E3D	testing

Figure 9.5: Message sequence tool in use

User can remove alreadey prepared or sent messages. He can remove either all or only the selected ones. Sending or removing of selected messages can be also done by clicking the item in the context menu called by the mouse right button click, or clicking coreesponding item in tool's *"Message"* menu bar Tool also offers a possibility to save prepared messages either in binary or text form and open already saved messages. Several can message tools can be opened from the application main window. Tool also provides the option of dropping the messages from the mainwindow traffic view, from the log file opened in it or even from sent message history tool that will be describe in next section. User can specify the name of prepared sequence in a *"SequenceName"* line edit placed at the top of the message table view. Name typed in this line edit is immediately reflected in tool's window title.

9.2.4.2 Sent message history tool

Message history tool gives a user possibility to keep the track of messages sent either by the message sequence tool, or from the application main window. Apart from reviewing of sent message history, tool enables to resend either all, or only selected messages that it stores and displays as it is shown at figure 9.6.



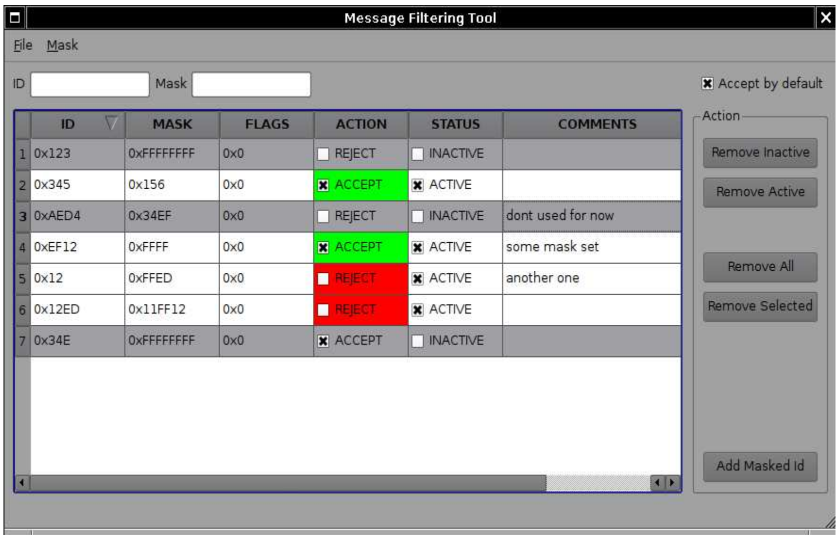
	ID	Flags	TimeStamp	Length	Data	Comments
3	0xAED	0x4	1/15/08 11:41 PM	2	123F	
7	0xAED	0x4	1/15/08 11:41 PM	2	123F	
4	0x34EF	0x0	1/15/08 11:41 PM	2	123E	
8	0x34EF	0x0	1/15/08 11:41 PM	2	123E	
2	0x34	0x6	1/15/08 11:41 PM	3	12345E	
6	0x34	0x6	1/15/08 11:41 PM	3	12345E	
1	0x159	0x4	1/15/08 11:41 PM	2	1234	
5	0x159	0x4	1/15/08 11:41 PM	2	1234	

Figure 9.6: Sent message history tool in use

Message history can be saved either in text or in binary form. Saved files can be reopened and used again: messages can be dragged from this tool and dropped to any opened message sequence tool, in which they can be edited and send to the CAN bus. Sent message history tool enables to remove either all of the displayed messages or only the ones selected by the mouse. As well as message sequence tool, this tool provides a possibility of sending and removing of displayed messages by clicking corresponding item in context menu called by the mouse right button click or by clicking the item in *"Message"* menu bar. Messages are being sent in the order they are displayed in tool's view or in order they were selected by the mouse.

9.2.4.3 Traffic filtering tool

Message filtering tool gives a user possibility to filter the monitored traffic by the masks created using this tool. Tool can be opened from the application main window clicking particular toolbar button, item in menu bar or using the shortcut **Ctrl+T**. Each newly added mask is appended at the bottom row of tool's table view. User can specify the mask and message ID either in decimal or in hexadecimal form. Again, if the specified values are in hexadecimal form user has to add "0x" prefix so they can be correctly interpreted by the tool. Screenshot of TFT in use is shown at figure 9.7.



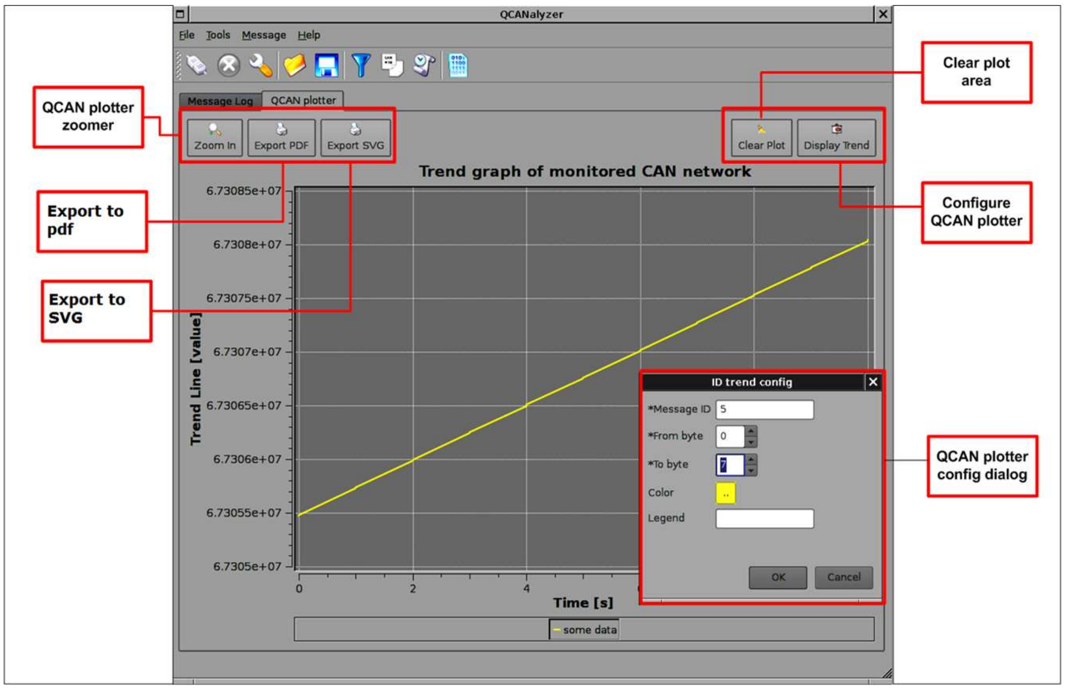
	ID	MASK	FLAGS	ACTION	STATUS	COMMENTS
1	0x123	0xFFFFFFFF	0x0	☐ REJECT	☐ INACTIVE	
2	0x345	0x156	0x0	☒ ACCEPT	☒ ACTIVE	
3	0xAED4	0x34EF	0x0	☐ REJECT	☐ INACTIVE	dont used for now
4	0xEF12	0xFFFF	0x0	☒ ACCEPT	☒ ACTIVE	some mask set
5	0x12	0xFFED	0x0	☐ REJECT	☒ ACTIVE	another one
6	0x12ED	0x11FF12	0x0	☐ REJECT	☒ ACTIVE	
7	0x34E	0xFFFFFFFF	0x0	☒ ACCEPT	☐ INACTIVE	

Figure 9.7: Traffic filtering tool in use

User can define several filtering masks or rules. Each new added masks is automatically created with *"REJECT action" "INACTIVE status"*. Reject means that monitored messages that pass set-up mask rule will be reject when the status of set-up rules is set to *"ACTIVE"*. User can change the mask status or action by simple checking or unchecking the checkbox in particular status/action column. As well as the previous tools, this tool gives a possibility to remove all or only by the user selected masks (rules). It also allows a user to remove all *"ACTIVE"* or all *"INACTIVE"* masks. User can save defined masks to a file that can be later opened by this tool and used in the very same way that was described here. When there are no masks specified in the tool, or all mask rules status are set to *"INACTIVE"*, user can define default filtering behaviour by checking/unchecking the *"Accept by default"* checkbox placed at the bottom of the tool's table view.

9.2.5 QCAN plotter

QCAN plotter is a plotting tool that allows a user to draw the trend curves of data transferred by the CAN messages with the particular message ID. In order to display these data, user must configure the plottiing options using its tool's configuration dialog. Obligatory fields that must be specified are **message ID** and the **byte range** of the data carried by the CAN messages. They are marked with the asterisk in tool's configuration dialog. user can specify the legend of displayed trend curve and even choose the color of the curve. QCANplotter as well as its configuration dialog is shown at figure 9.8.



*Message ID: 5
*From byte: 0
*To byte: 7
Color
Legend

Figure 9.8: QCAN plotter in use

Trend curves are displayed in a plot grid that automatically adjust its scales according to the displaying data. Apart from displaying the curves, tool offers other functionality. To enable a closer look on displayed curve(s) values, tool provides a graph zoomer. If a user wants to use the zoomer, he has to toggle the *"Zoom In"* button and mark required plot area. Right click on a plot canvas area will return the view to the previous displayed state. If a user performed several zooming actions he can return to the original displayed view either by toggling the *"Zoom In"* button or by using the combination of **Ctrl+mouse right click** shortcut. The tool allows to display several trend curves with different colors and legends specified by the configuration dialog. Added curves can be hidden from the plot area and redisplayed back using the legend's toggle buttons placed below the displayed graph. Plot area can be cleared by clicking the *"Clear Plot"* tool button. User can export displayed graphs either to pdf or to SVG format by clicking corresponding tool buttons placed at the top of the plot area.

Chapter 10

Conclusion

In this diploma theses I managed to create a graphical application called QCANalyzer that allows a user to monitor CAN network. Using it, one can perform a higher analysis of captured data either from the live CAN network, or from previously saved files. In order to enable the communication of the application with actual CAN hardware, some of the results reached byt the OCERA project were used. QCANalyzer uses OCERA LinCAN driver, as well as the VCA library, which provides an advanced API for the communication with this driver. Application was programmed using Qt development toolkit. The use of Qt for developing, not only graphical applications, is hugely growing in the world of IT. Thus, the decision of choosing the Qt gave me a chance to learn new advanced technology, plus it eased the whole development process of the application, since Qt provides a complete framework for cross-platform application development.

Getting familiarized with unknown development framework caused a little slow-down of development at the beginning, but at last it showed up to be the right solution. First, some components of VCA library were reprogrammed using this toolkit in order to ease the distribution of captured data to particular application's graphical objects, then the actual implementation of QCANalyzer graphical interface followed. It required learning of several Qt programming technics that were applied to acheive the functionality requirements claimed at the the beginning. Apart from standard functionality of sending and receiving of CAN messages offered by many CAN monitoring tools, QCANalyzer offers many other useful solutions. It provides an advanced tool for sending CAN message sequences as well as the tool that enables to filter captured traffic. Traffic filtering tool gives a user possibility set-up several traffic filtering masks with different roles (accept, reject) and states (active, inactive), which slightly resembles *iptables* firewalls implemented on

UNIX machines. QCANalyzer provides mechanisms for local and remote monitoring over TCP/IP. User can connect to any machine on the internet, on which *canmond* daemon is running, and perform the monitoring remotely. Feature that distringuishes this tool the most from other free available CAN monitoring tools is advanced tool for plotting transferred data over CAN network. User can start monitoring and after simple configuration of *QCAN plotter* tool he can watch the transferred data on the real-timingly drawn graph.

Application has been tested on virtually loaded LinCAN driver, using the utilities provided by the OCERA project. Program has not been tested on a real CAN hardware yet, but testing results showed, that QCANalyzer should appropriately fulfil all requirements that are posed on CAN monitoring tool. Testing of the application on real CAN hardware should be performed by the user himself, before he decides to use it in a commercial or in other project.

Future extensions of the application should provide implementation of CANopen protocol support. QCANalyzer already provides the API that can be used for this porpose. The only thing that has to be implemented to enable this support is graphical interface, that would provide a user an easy way of manipulation with CANopen devices. CANopen support will enable generation of network variables, the only assignment requirement that has not been fulfilled in this work. Other possible extension can be implementation of more advanced traffic filtering or sending of periodic CAN messages what can be very useful in automotive applications. Another possible extension is implementation of advanced *project files* that would enable an advanced project control and that would allow a user to store all monitoring session configuration and settings in partricular project file. In order to speed-up the development process of further application extensions, official QCANalyzer SourceForge[1] release is currently being prepared. QCANalyzer will be released under GPL or LGPL license. Anyone will be able to freely download the application source codes and change them for its own possible purposes.

In the end I would like thank again my tutor Ing. Pavel Píša for his patience, help and for many useful suggestions. Without him, this theses would not reach the quality it has now.

[1]source code repository on http://sourceforge.net/index.php web site for software developers to control and manage open source software development

Bibliography

[1] CAN international users and manufacturers group web site
, http://www.can-cia.org/.

[2] The Vector company web site
, http://www.vector-worldwide.com/.

[3] The Peak company web site
, http://www.peak-system.com/.

[4] VSCP project web site
, http://www.vscp.org/.

[5] OCERA project web pages
, http://www.ocera.org/.

[6] OCERA project development web pages
, https://sourceforge.net/projects/ocera/.

[7] Gtk+ project documentation
, http://gtk.org/.

[8] Qt framework web pages
, http://trolltech.com/products/qt.

[9] Java-SWING tutorial web pages
, http://java.sun.com/docs/books/tutorial/uiswing/.

[10] Qt framework online documentation
, http://doc.trolltech.com/4.3/.

[11] Qwt Project Online Documentation
, http://qwt.sourceforge.net/.

Attachement A

List of abbreviations

ISO	The International Organization for Standardization
IEEE	The Institute of Electrical and Electronics Engineers
IETF	The Internet Engineering Task Force
CAN	Controller Area Network
GUI	Graphical User Interface
ISOOSI	ISO Open Systems Interconnection Model
EDS	Electronic Data Sheet
PDO/SDO	Process/Service Data Object
NMT	Network Management Technology
SYNC	Synchronization Object
OCERA	Open Components for Embedded Real-time Applications
API	Application Programming Interface
GPL	General Public License
LGPL	Lesser General Public License
ANSI-C	American National Standards Institute standard for the C programming language

API Application Programming Interface

VSCP Very Simple Control Protocol

GAVL/AVL Generic/Generalized Adelson-Velski Landis trees

KDE K Desktop Environment

GNOME Desktop environment for computers running Unix based operating systems

GNU Computer operating system composed entirely of free software

JFC Java Foundation Classes

STL Standard Template Library

TFT Traffic Filtering Tool