FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

CIDESI®

Development of a monitoring system for processing the CAN bus data of public transport vehicles

A Thesis

Submitted to the Faculty of Fachhochschule Aachen and Centro de Ingeniería y Desarrollo Industrial

BY

Luis Adrián Lizama Servín

In Partial Fulfillment of the requirements
for the degree of Master of Science in Mechatronics

Santiago de Queretaro, Qro., Mexico, February 2018

Declaration

I hereby declare that this thesis work has been conducted entirely on my own accord.

The data, the software employed, and the information used have all been utilized in complete agreement to the copyright rules of concerned establishments.

Any reproduction of this report or the data and research results contained in it, either electronically or in publishing, may only be performed with prior sanction of the University of Applied Sciences of Aachen (FH Aachen), the Centro de Ingeniería y Desarrollo Industrial (CIDESI), and myself, the author.

Querétaro, February 2018

Luis Adrián Lizama Servín

Dedication

This work is dedicated to the memory of my best friend Matías.

"Commit to the Lord whatever you do, and He will establish your plans".

Proverbs 16:3

Acknowledgment

My biggest appreciation to CONACyT for funding my education in Mexico and in Germany, without it, this project wouldn’t have happened.

I would like to express my sincere gratitude to Ricardo Tejada and Andreas Hermanns, managers of the IVU Latin America Program. I highly appreciate the trust they have placed in me to participate in this project as well as the opportunity they have given me to be part of such a prestigious company as IVU. Moreover, I would like to remark the great support and the continuous assistance that Ricardo offered at each different stage of development.

I would also like to acknowledge my external advisor Dr. Michael Zimmer, senior software developer at IVU. He not only consistently supported my work to get results of better quality but also he provided expertise that greatly assisted the project. In addition, Michael always took the time to clarify my doubts by responding to each one of the issues with excellent solutions.

Furthermore, I would like to thank my thesis advisor Prof. Dr.-Ing. Günter Schmitz of the Mechanical engineering & mechatronics faculty at FH Aachen. The door to Prof. Schmitz office was always open, especially when I need it most. Without his significant participation and high availability, the project could not have been successfully conducted.

My sincere thanks also goes to Dipl.-Ing. Peter Börger who gave access to the IVU engineering department and hardware facilities. Without they precious support it would not be possible to conduct this thesis. I really appreciate his patience and orientation while I was learning about this matter.

Last but not the least, I would also like to acknowledge Dr. Hugo Jiménez, research professor at the Center for Engineering and Industrial Development (CIDESI) as the second advisor of this thesis, I am gratefully indebted to his for his very valuable comments on this thesis.

Abstract

The aim of this master’s thesis is the development of a system which can store relevant information from the stream of continuous data from the CAN Bus of public transport vehicles. The collected data must be transmitted through a telematics gateway to a back office server which will process the information received. Once these data have been stored and treated the next step consists of generating a human interface to provide access to the information in an easy and digestible form.

The first part of the project was achieved by developing an onboard computer software in order to retrieve the CAN Bus data. The computer software was designed to store data in a light and accessible way, therefore, the system includes a relational SQLite database. The information is organized and stored on the back office system through a windows service which aims to prepare the data for proper analysis. The user interface is based on dynamic WEB pages and server scripts to carry out the graphical visualization of data.

Kurzfassung

Das Ziel dieser Masterarbeit ist die Entwicklung eines Systems, welches die relevanten Informationen vom kontinuierlichen Datenfluss vom CAN Bus von öffentlichen Transport Fahrzeugen speichert. Die gesammelten Daten müssen durch einen Telematik Gateway an einen Back-office Server übermittelt werden. Dieser verarbeitet die empfangenen Informationen. Sobald diese Daten gespeichert und verarbeitet wurden, besteht der nächste Schritt darin eine menschliche Schnittstelle zu generieren, um in einfacher und verträglicher Form einen Zugang zu den Informationen herzustellen.

Das Projekt wurde in Zusammenarbeit mit der Firma IVU Traffic Technologies AG entwickelt und die dafür gebrauchte Hardware ist die IVU.box. Diese ist der Bordcomputer, welche von der Firma für Busse, Straßenbahnen und Züge angeboten werden. Der erste Teil dieses Projekts wurde dadurch realisiert, dass die Bordcomputer Software für die IVU.box entwickelt wurde, um die CAN Bus Daten abzurufen. Die Bordcomputer Software wurde entwickelt, um Daten auf einfache und leicht zugängliche Weise zu speichern. Um dies sicher zu stellen, beinhaltet das System eine integrierte SQLite Datenbank. Die Information wird auf dem Back-office Server durch einen Windows Service verarbeitet und gespeichert, was die Daten auf eine geeignete Analyse vorbereitet. Die Benutzerschnittstelle basiert auf dynamischen WEB Seiten und Server-Skripten, um die graphische Visualisierung der Daten auszuführen.

Contents

List of Figures

List of tables

1 Introduction

Vehicle monitoring systems are a fast growing product category due to the large number of companies that own a multitude of commercial vehicles such as buses, cars, trucks, ships, planes, rail cars and so on. Moreover, transport companies wish to have a central database to access vehicle information for product improvements and diagnostics. Therefore, there is a strong interest for a system that can store, monitor and analyze vehicle data regarding to the different needs of different users [1].

"*Because different companies and different industries have different vehicle data gathering and reporting needs, current solutions involve constructing specialized systems for each particular user application*" [1]. Accordingly, one method for monitoring, configuring, programming and diagnosing operation of at least one vehicle is that developed by Michael Kapolkaa.

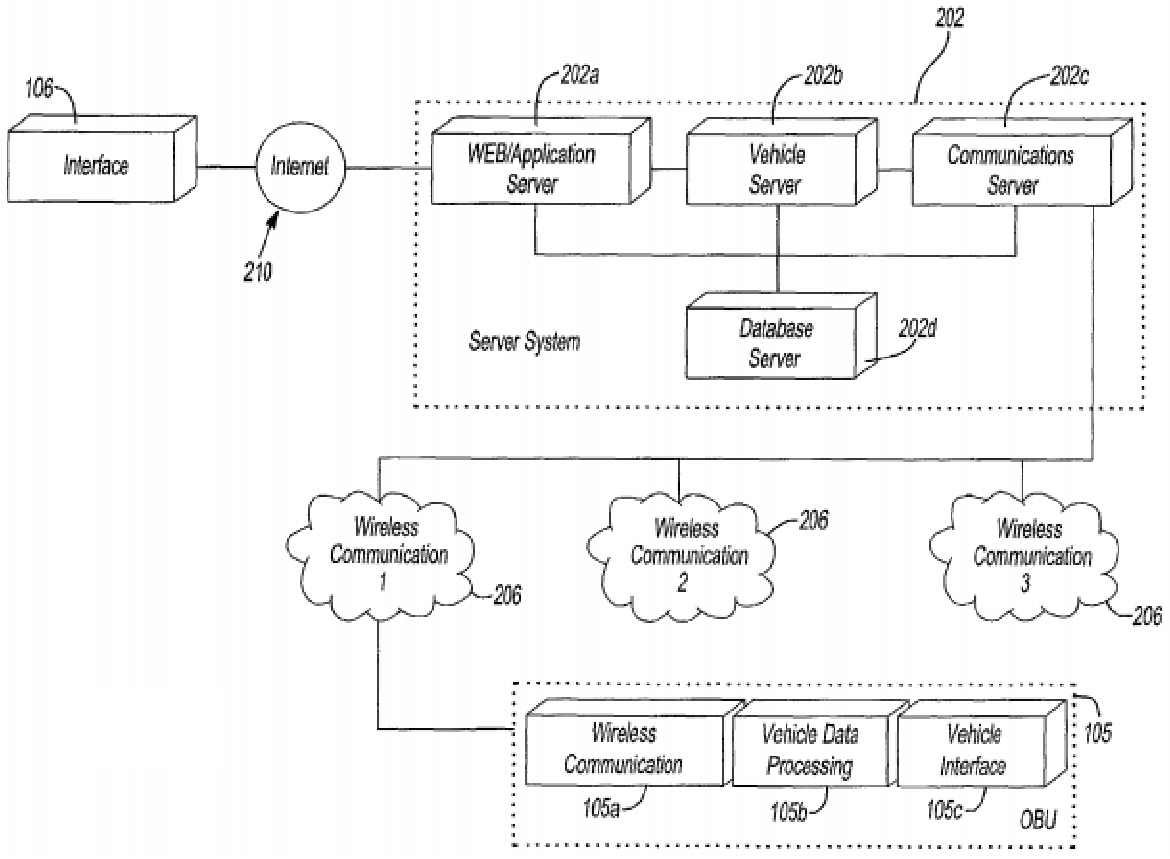


Figure 1. System architecture according to the patent of Michael Kapolkaa [1].

The system is designed with applications and services that interact with raw data from the vehicle bus so that vehicle information can be stored, analyzed and displayed in a format that is meaningful to a particular user. The system architecture shown in figure 1 represents the process for carrying out the functionalities of the system implemented [1].

Another method for monitoring a vehicle's engine control unit data is that implemented in the patent “Internet-based system for monitoring vehicles” developed by Larkin Hill. The invention provides a method for monitoring a vehicle that features the steps of [2]:

1) Generating a vehicle data packet.
2) Transmitting the data packet.
3) Processing the data packet.
4) Displaying the set of data on a web page.

Data from the above mentioned system are made available through a serial connector which retrieved the data from the standardized electronic vehicle bus.

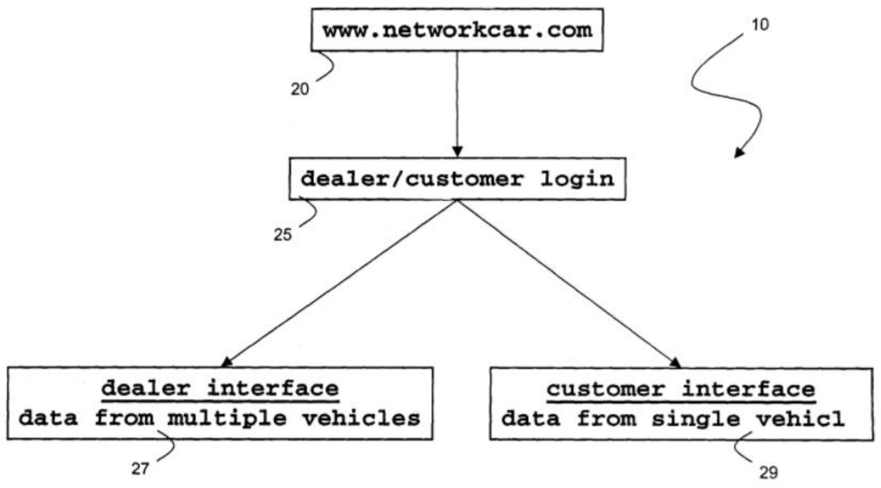


Figure 2. Schematic drawing of a website according to the patent of Larkin Hill [2].

The system proposed by Larkin Hill describes the development of a web page which represents the following benefits [2]:

1) The schematic drawing of a website shown in Figure 2 illustrates the capacity of the system to sort and analyze the retrieved data depending on vehicle and customer.
2) The web page also support a large number of tools that can be used to analyze data.
3) The web page can be accessed all over the world using wireless devices.

1.1 Motivation

As in the examples mentioned above the proper implementation of a vehicle monitoring system allows different companies to obtain benefits such as engine protection methods, remote maintenance assistance, vehicle diagnostics systems, determining driver negligence, vehicle inspection support and so on. Therefore, for any transport company, a data monitoring system represents an area of opportunity to increase its product offer. In the particular case of IVU, the project is of great interest to clients in Latin America, such as GEP (Grupo Express del Perú).

One of the main goals of the present work is the storage of significant CAN Bus data in order to generate historical data which can also be used in the future. The proper storage of the data as well as its adequate processing aims to generate the basis for a predictive maintenance system. Moreover, the implementation of a CAN Bus interface generates the possibility of real time data monitoring of the bus data.

Advantages and importance of a predictive maintenance system:

- Decreasing operational cost.
- It allows to monitor the health of individual components of the fleet avoiding costly manual inspections of the vehicle as a whole.
- Predictive maintenance aims to rarely have a bus break down in the middle of service and thus it decreases the cost of negative customer experience.

Advantage and importance of a real-time monitoring system:

- Protection against accidents.
- Improving employee driving habits.
- Improving emergency response: Reducing confusion, time saving, awareness to situation change, quickly identify problems.

The implementation of the system represents also a contribution to the management decision making process for, among other things:

- Operational cost
- Supervise vehicle's usage
- Monitor drive behavior

1.2 IVU Traffic technologies

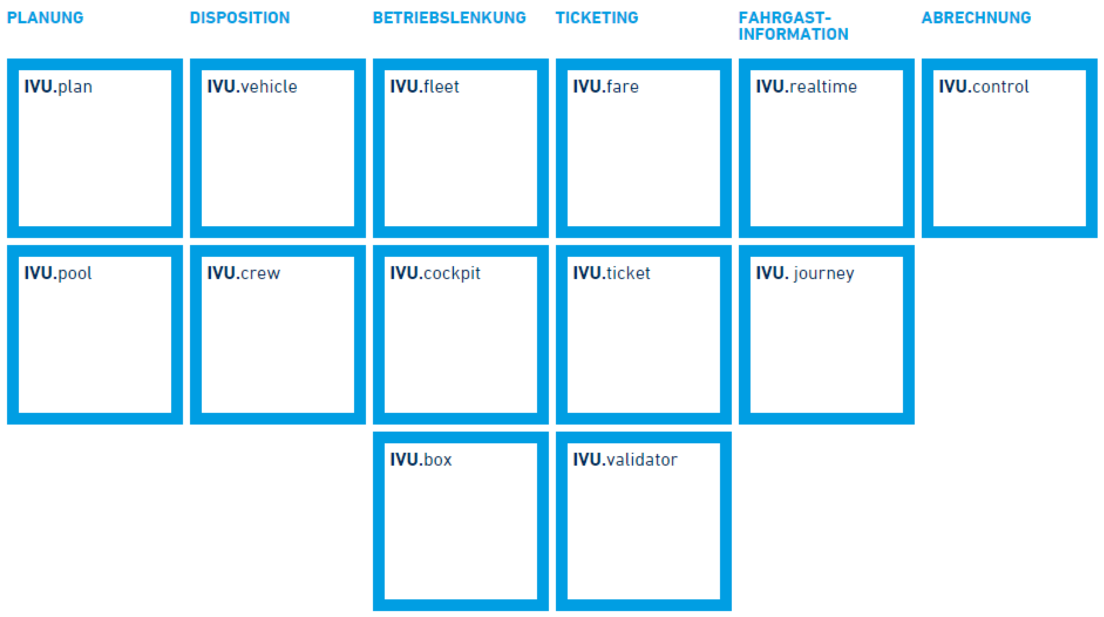


Figure 3. IVU.suite [3].

“**IVU Traffic Technologies AG** has been working for over 40 years with more than 400 engineers to ensure punctual and reliable transport in the world’s metropolises. In growing cities, people and vehicles are constantly on the move – a logistical challenge that calls for intelligent and secure software systems. Based on the IVU.suite, IVU develops high-performance solutions for public passenger and goods transport and transport logistics” [3].

IVU Traffic Technologies is a company which offers support for all fields of activity in transport companies and provides integrated solutions: from planning, dispatching, fleet management, and ticketing to passenger information [3]. In this project, the main objective is to develop a software and hardware solution which can retrieve and monitor on-board data from the vehicle, especially data from the CAN bus. The project therefore represents an addition to the current products offered by the company.

IVU.box is the onboard computer offered by the company for buses, trams, and trains. For that reason, the thesis focuses on developing software suitable for the hardware provided, in this case, the IVU.box. Moreover, IVU.fleet ensures a consistent supply of data for vehicles. Thus, IVU.fleet employs a telematics gateway to transmit to the back office system the collected data from the CAN bus.

1.3 Problem Statement

The vehicle on-board data consists of thousands of signals from the electronic control units such as engine control unit, cruise control, electric power steering, battery and recharging systems, auto start/stop, park brakes, and so on. The data is communicated through a CAN network. Since the frames are sent repeatedly with a specified frequency they form streams of continuous data which are used for vehicle control and status signaling.

The project focuses on developing a remote system for storing, monitoring and analyzing the vehicle stream’s data such as motor speed, fuel consumption, wheel speed, clutch pedal condition, accelerator pedal position, brake pedal condition, alternator status, vibrations, temperature, pressure, voltage, current, number of kilometers traveled and so on.

The system is developed with the following characteristics:

- The CAN bus interface reads CAN bus frames that adhere to SAE J1939 standard.
- The CAN bus interface is suitable for the onboard computer operating system Windows CE 7.
- The system stores (on the onboard computer) in a lightweight manner the data retrieved.
- The system transmits the data retrieved to the back office server for further analysis. Once the information has been backed up the system removes the data from the onboard computer.
- The back office system organizes and stores the data to generate historical information which can be used in the future.
- The system includes a user interface that allows information visualization according to customers and vehicles.

In addition the system presents the following benefits:

- Easy integration with the current products offered by the company.
- Reliable and precise data.
- General solution for the entire fleet.

The following chapter deals with some concepts required to understand and to implement the CAN Bus monitoring system.

2 Concepts & terminology

2.1 CAN Bus

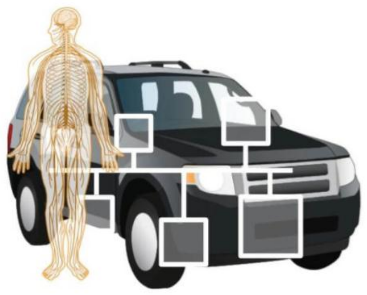


Figure 4. CAN bus [4].

Assuming that the car is like a human body, as show in Figure 4. "*The CAN bus is the nervous system, enabling communication between all parts of the body"* [4].

CAN stands for Controller Area Network, and it is a multi-master bus originally developed for the automotive industry to allow communication between electronic control units and sensors without a network host what results in the broadcasting of many messages to the entire network. The CAN Bus communication protocol was developed by the German BOSCH Corporation and specifies a communication speed up to 1MBPS [5].

CAN is an International Standardization Organization (ISO) standard implemented to replace the elaborate wiring harness with a two-wire bus [5]. On the contrary, CAN Bus offers benefits such as low cost of implementation since it avoids direct analogue signal lines, central error diagnosis, and it is robust towards failures. It is efficient because messages with the highest priority are not interrupted, and since the system allows easy modification or inclusion of additional nodes it is considered flexible. Nowadays, almost every European car comes with a CAN bus system which has increased the popularity of the protocol in other industries including ships, planes and industrial automation. Moreover, it is broadly used in more modern use cases which include drones, radar systems and submarines [4].

2.1.1 Architecture

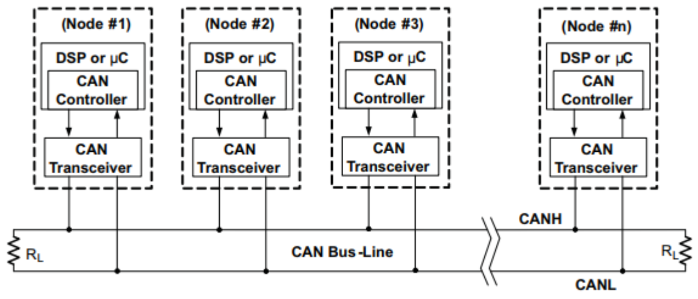


Figure 5. CAN Bus Architecture [5].

The CAN Bus is a transmission medium designed for Electronic Control Units (ECUs) communication. ECUs examples include (also known as nodes), engine control unit, cruise control, park brakes, audio systems, doors, or just a simple I/O device. The electrical characteristics of the CAN Bus incorporate a two wire bus terminated at each end with 120 Ω resistors, the first wire is called high-level transmission line CANH, and the second one is called low-level transmission line CANL [6].

Each node requires a digital signal processor (DSP), a CAN controller and a transceiver [5]. The DSP or microcontroller transmits messages to the network and executes indications to the sensors and actuators depending on the retrieved messages. The CAN arbitration process is handled by the CAN controller as well as ensuring that an entire message is available for fetching or sending. The transceiver is responsible for ensuring that the voltage levels are adequate for the transmission of messages between the network and the node. Figure 5 illustrates the electrical characteristics of the CAN Bus in conjunction with the components that integrate the nodes.

2.1.2 Frames

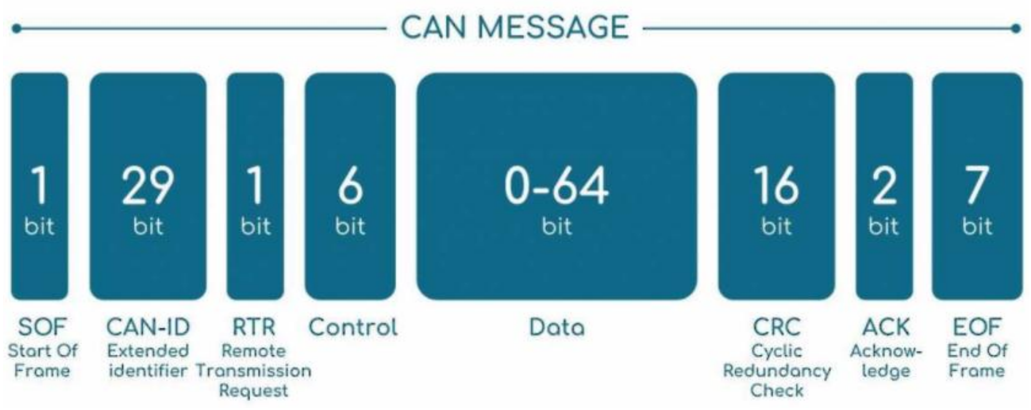


Figure 6. CAN message [4].

In the CAN bus protocol, messages are known as frames and are composed, among other values, by an identifier or ID, the frame type, and the data to be transmitted [5]. The ID is a unique value which also represents the message priority. Depending on the size of the ID two frame formats have been implemented. The base format contains an 11-bit ID while the extended format has a 29-bit format [6]. Figure 6 illustrates the structure of a frame with a 29-bit ID.

According to the CAN bus protocol, four possible messages can be transmitted [5]:

- The Data Frame: It contains data for transmission.
- The Remote Frame: It solicits the transmission of a specific node.
- The Error Frame: It violates the formatting rules. It is emitted by any node when an error is detected.
- The Overload Frame: It is used to generate a delay between messages.

2.1.3 Arbitration

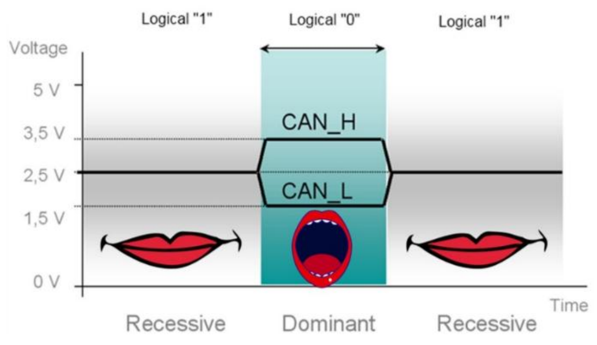


Figure 7. CAN signaling and logic levels [7].

"In the real world, if two people speak at the same time, how do you determine who should speak? Sometimes it´s the one who talks the loudest, and that´s essentially how a controller area network (CAN) bus work" [7].

As shown in Figure 7, one of the main characteristics of the CAN bus is a logic-high is associated with a zero, and a logic-low is associated with a one. This particularity is very important at the moment of the definition of the identifier since it must be unique for each node and it must represent the frame priority. The proper definition of the frame priority is essential to carry out the arbitration process correctly.

The arbitration process is a method used for defining the priority of the frame when two nodes try to occupy the bus simultaneously. The node winning arbitration continues with the message and the node that loses arbitration re-queues its message for later transmission. Accordingly, any node that transmits a logical one when another node transmits a logical zero loses the arbitration [5].

2.1.4 Abstraction layers

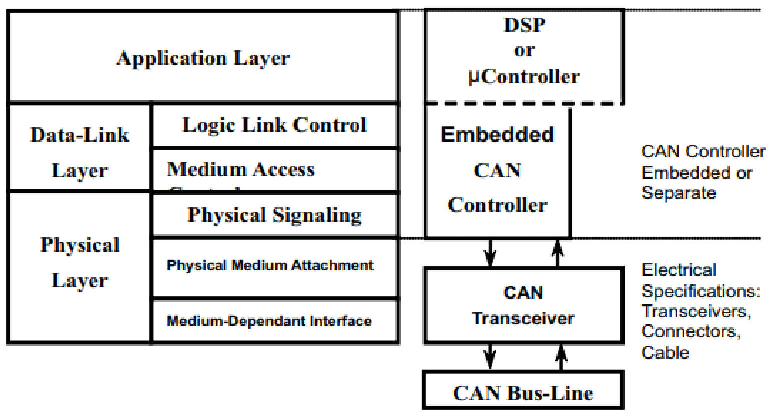


Figure 8. The Layered ISO 11898 Standard Architecture [5].

The different standards decompose the CAN protocol into different levels according to its functionalities and implementation characteristics. These levels are known as abstraction layers. For example, Figure 8 illustrates the layers defined in the ISO 11898 Standard [5]. The physical layer represents the transmission and reception of raw bit streams over a physical medium, and the Data-Link Layer is responsible for the reliable transmission of data frames. In the CAN protocol, these two levels are implemented with the transceiver and the CAN Controller respectively. Since the CAN standard does not include tasks such as device addressing and transport data blocks, the application layer establishes the communication link to a higher layer protocol such as CANopen, DeviceNet, SAE J1939, and so on [5].

2.1.5 SAE J1939

SAE J1939 is a standard developed by the Society of Automotive Engineers and focuses mainly on communication between electronic control units of heavy-duty trucks. It defines the physical layer, data link layer, network layer and application layer [6]. Therefore, SAE J1939 is considered a higher layer protocol with the following characteristics [8]:

- It enables the ECUs communication across manufacturers.
- It handles the transport of multi data blocks.
- It specifies the process to convert the raw data into readable data.

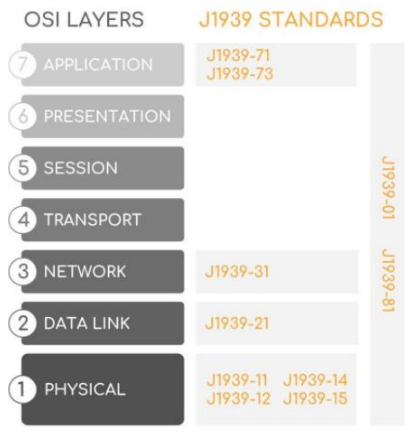


Figure 9. J1939 Standards [8].

The above characteristics are achieved by offering a family of SAE standards, which are illustrated in Figure 9. For example, key attributes according to the SAE J1939-71 standard, which is used by the European Automobile Manufacturers Association are [9]:

- Speed of 250kb/s
- Unused bytes are filled with FF
- Uses an extended 29 bit identifier
- Messages are identified by Parameter Group Numbers (PGN)
- A PGN contains Suspect Parameter Numbers (SPN)
- A SPN reflects parameters through 8 data bytes

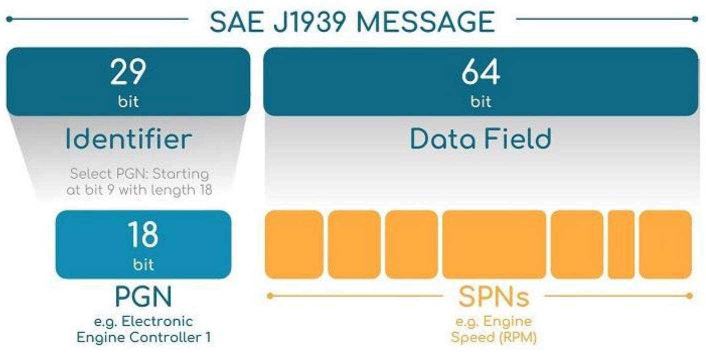


Figure 10. SAE J1939 Message [8].

The understanding of PGNs and SPNs is essential to monitoring the CAN bus data transmitted according to SAE J1939 standard. The parameter group number or PGN is a unique ID for each message. It is not possible to match the full 29-bit CAN identifier with the PGN. As Figure 10 shows, the 29-bit identifier must be interpreted to obtain the proper PGN. To establish the PGN according to the SAE J1939 standard, the PGN must have an 18-bit length starting at bit 9 of the 29-bit identifier. For instance, for identifier 0x0CF00401 the corresponding PGN is 0x0F004 [8].

0CF00401 FF FF FF 68 13 FF FF FF

Figure 11. SPN values [8].

Moreover, each PGN contains several SPNs which reflect the actual sensed values such as fuel consumption or engine hours. Figure 11 illustrates the 0x0CF00401 frame in which the only relevant data is in byte 4 and 5 (0x68, 0x13). According to SAE J1939, the mentioned frame represents an RPM value and, the proper scale for the specified frame is 0.125 RPM/bit with an offset 0. Therefore, taking the decimal value of 0x1368 (4968) and using the proper scale the resulting RMP value is 621 [8].

2.2 IVU.box

IVU.box is the on-board computer offered by the company for buses, trams and trains. In the current CAN Bus data monitoring project the fleet of vehicles is composed only of buses. IVU.box is based on standard components, has interfaces to all common positioning and communication systems (GPS, analog and digital radio communication, GSM, UMTS, Tetra) and controls the entire vehicle environment [10].

Figure 12. IVU.box [10].

IVU.box [10]:

- Operating system: Windows CE 7
- CAN driver system (Cortex M3 microcontroller and ISO1050 CAN transceiver)
- Touchscreen
- Analog and digital wireless communications (GSM, GPRS, UMTS, Tetra)
- Data upload and download with WLAN, GPRS, UMTS
- A variety of interfaces (serial, IBIS, CAN, Ethernet, DigIO)
- Arm-based i.MX6 processor
- 1GB of RAM
- Uses industrial grade SD card for mass storage (1 to 4GB)

2.1.2 Windows Embedded CE

Windows Embedded CE is an operating system developed by Microsoft. It is designed to create intelligent solutions for embedded devices that can be found in different engineering fields such as [11]:

- Automotive
- Consumer and entertainment
- Home and building automation
- Industrial automation, process control, and manufacturing
- Information kiosk and self-serve terminal
- Medical
- Mobile phone and communication
- Office equipment
- Robotics

Windows Embedded CE has more than 15 years in the market and each new version improves key aspects such as [11]:

- Small footprint
- Modular architecture
- Real-time support
- Support of broad range of hardware
- Efficient power management

- Efficient development tools
- Efficient debugging and testing tools

Windows Embedded CE is a robust development environment that allows implementing applications reliably using a wide variety of tools. Furthermore, it is compatible with different hardware components according to the needs of the market.

2.2.2 CAN Driver

As it is listed in section 2.2, the onboard computer has a CAN driver system which is integrated by the Cortex M3 microcontroller and the ISO1050 CAN transceiver. In addition, the IVU technology department implements a particular communication protocol called USB tunnel with which consistent communication is achieved with the CAN module integrated into the microcontroller. This communication protocol uses the GFCAN32.dll dynamic library developed by the company Gartz and Fricke.

2.2.2.1 Cortex M3 microcontroller & ISO 1050 CAN transceiver

The microcontroller LM3S5B91 (Cortex M3) is part of the family of Cortex microcontrollers developed by the Texas Instruments company. The Cortex M3 microcontroller offers benefits such as minimal memory implementation, low power consumption, outstanding computational performance and exceptional system response to interrupts. Some of its characteristics are [12]:

- High Performance: 80-MHz operation.
- 256 KB single-cycle Flash memory.
- 96 KB single-cycle SRAM.
- Advanced Communication Interfaces: UART, SSI, I2C, I2S, CAN, USB.
- Industrial (-40°C to 85°C) temperature range.

As mentioned above, one of the main features of the Cortex M3 microcontroller is its CAN interface. Thus, the Cortex M3 microcontroller includes a CAN module for receiving and sending the serial bits from/to the CAN bus accordingly to the algorithm implemented on the microcontroller. Essential features of the module are listed below [12]:

- CAN protocol version 2.0 part A/B.
- Bit rates up to 1 Mbps.
- 32 message objects with individual identifier masks.
- Maskable interrupt.
- Gluelessly attaches to an external CAN transceiver through the CANnTX and CANnRX signals.

The CAN module consists of three parts, CAN protocol controller, Message memory and CAN register interface, which must be programmed properly to obtain the correct CAN bus frames [12]. Figure 13 illustrates the frame structure which as explained in chapter 2.1.2 can be of the data, remote, error or overload frame type.

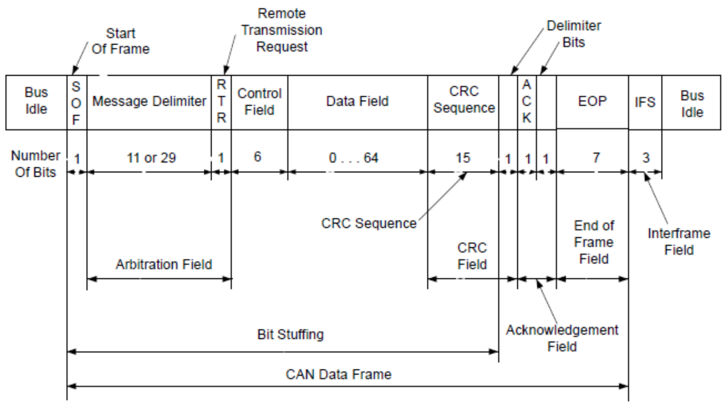
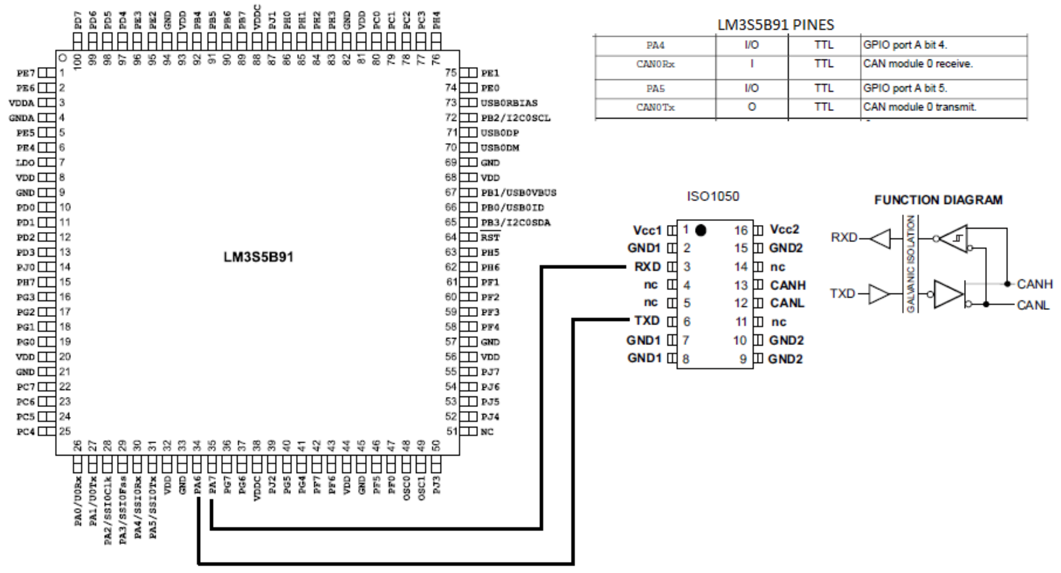


Figure 13. CAN Frame [12].

The connection between the CAN bus and the controller is made through the transceiver, which is responsible for handling the appropriate voltages between the controller and the bus. In the particular case of the IVU.box, the microcontroller is connected to the ISO1050 CAN transceiver as shown in Figure 14. The mentioned transceiver is also developed by the Texas instruments company and, it has the following features [13]:

- Meets the specifications of ISO 11898 standard.
- Bit rates up to 1 Mbps.
- Overvoltage protection from -27 V to 40 V
- 3.3 V Inputs are 5-V Tolerant
- Low loop delay: 150 ns

The connection between the ISO1050 CAN transceiver and the Cortex M3 microcontroller is basically the two-pin wiring, as shown in the following figure 14.



LM3S5B91 PINES

PA4	I/O	TTL	GPIO port A bit 4.
CAN0Rx	I	TTL	CAN module 0 receive.
PA5	I/O	TTL	GPIO port A bit 5.
CAN0Tx	O	TTL	CAN module 0 transmit.

Figure 14. Connection between LM3S5B91 and ISO150.

2.2.2.2 GFCAN32.dll dynamic library

As mentioned in the previous chapter, the CAN module consists of three parts, protocol controller, the Message memory, and CAN register interface. These three elements are considered the kernel-area and are responsible for obtaining the messages. The controller transfers and receives the data from the transceiver and passes the data on to the message's handler. Thus, the message handler is responsible for generating interrupts based on CAN bus events. Finally, the message object is a set of memory blocks which are accessed via the CAN registers [12]. However, all this process is carried out by the GFCAN32.dll dynamic library.

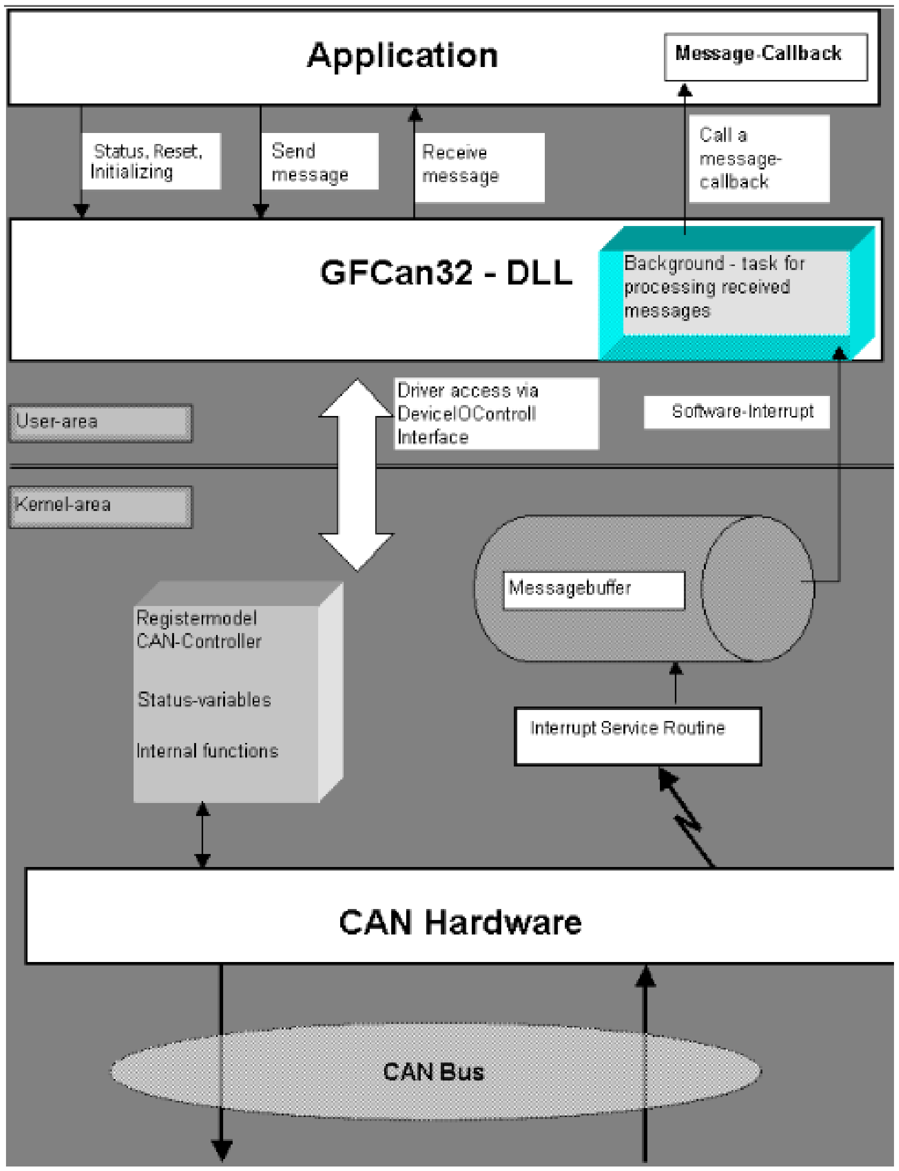
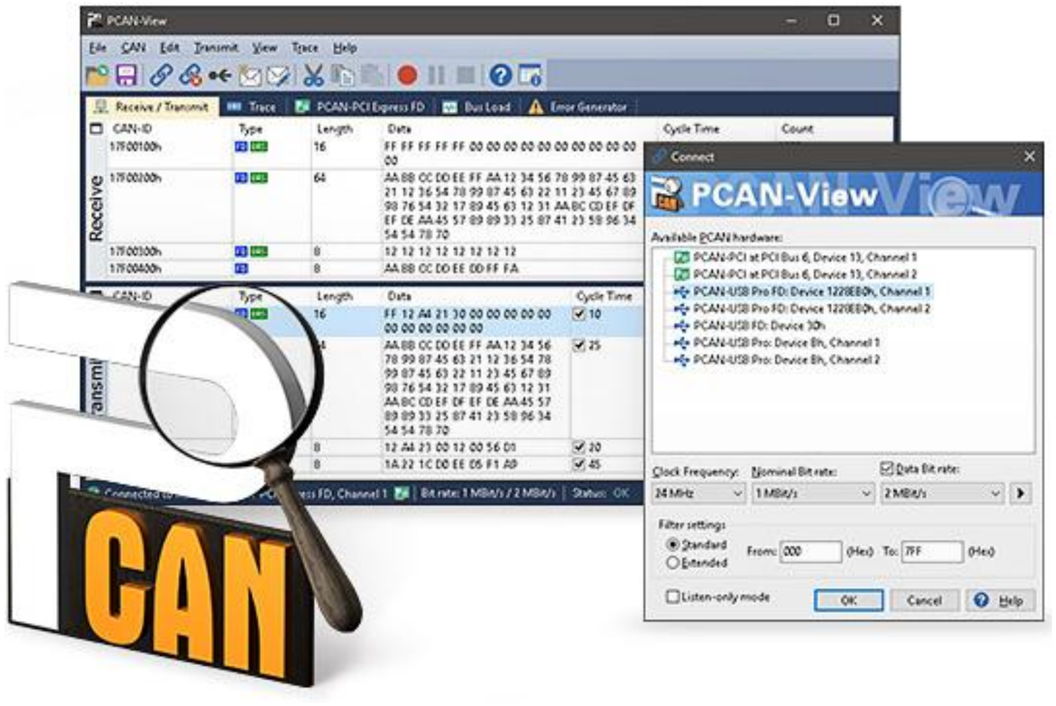


Figure 15. Schematic illustration of the GFCan32Api [14].

GFCan32 is a dynamic library developed by the company Ganz & Fricke and, it is designed primarily for Windows CE. In chapter 2.2.2, it was mentioned that through this library it is possible to access the CAN module of the microcontroller, thus, as shown in figure 15, by means of an interrupt service routine the library allows the transmission of information between the Kemel-area and the User-area [14]. The DLL library is loaded by the application as needed, and thus, the application gets the pointers to the proper functions in order to control the CAN bus module regarding the algorithm developed by the user [14].

2.3 PCAN-View & PCAN-USB

With the purpose of testing the on-board application, various software and hardware tools were used, among them PCAN-View and PCAN-USB. PCAN-View is a software tool for monitoring the data flow on a Controller Area Network (CAN). It is developed by the company PEAK-System Technik GmbH and, it has the following features [15]:



CAN-ID	Type	Length	Data	Cycle Time	Count
17F00100h		16	FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00		
17F00200h		64	AA BB CC DD EE FF AA 12 34 56 78 99 87 45 63 21 12 36 54 78 99 87 45 63 22 11 23 45 67 89 98 76 54 32 17 89 45 63 12 31 AA BC CD EF DF EF DE AA 45 57 89 89 33 25 87 41 23 58 96 34 54 54 78 70		
17F00300h		8	12 12 12 12 12 12 12 12		
17F00400h		8	AA BB CC DD EE DD FF FA		

CAN-ID	Type	Length	Data	Cycle Time
		16	FF 12 A4 21 30 00 00 00 00 00 00 00 00 00 00 00	✓ 10
		4	AA BB CC DD EE FF AA 12 34 56 78 99 87 45 63 21 12 36 54 78 99 87 45 63 22 11 23 45 67 89 98 76 54 32 17 89 45 63 12 31 AA BC CD EF DF EF DE AA 45 57 89 89 33 25 87 41 23 58 96 34 54 54 78 70	✓ 25
		8	12 A4 23 00 12 00 56 01	✓ 20
		8	1A 22 1C DD EE 05 F1 A0	✓ 45

Connected to … PC… ess FD, Channel 1 | Bit rate: 1 MBit/s / 2 MBit/s | Status: OK

Available PCAN hardware:
- PCAN-PCI at PCI Bus 6, Device 13, Channel 1
- PCAN-PCI at PCI Bus 6, Device 13, Channel 2
- PCAN-USB Pro FD: Device 1228EB0h, Channel 1
- PCAN-USB Pro FD: Device 1228EB0h, Channel 2
- PCAN-USB FD: Device 30h
- PCAN-USB Pro: Device 8h, Channel 1
- PCAN-USB Pro: Device 8h, Channel 2

Clock Frequency: 24 MHz
Nominal Bit rate: 1 MBit/s
☑ Data Bit rate: 2 MBit/s

Filter settings: [✓ Standard] Extended From: 000 (Hex) To: 7FF (Hex)

☐ Listen-only mode

Figure 16. PCAN-View [15].

- It can emit and receive CAN messages, including Remote request and Error frames.
- It displays CAN messages with its ID, Data length, and data bytes.
- Any number of frames can be emitted, either manually or automatically in fixed time.
- Support for CAN specifications 2.0 A/B and FD.
- Bit rates up to 1 Mbit/s.

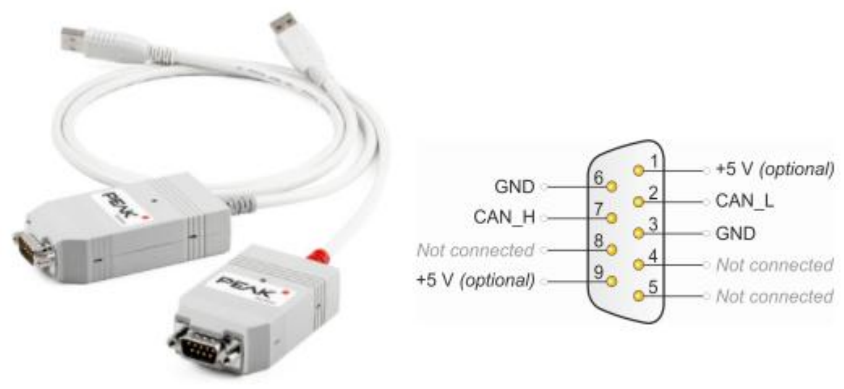


Figure 17. Pin assignment of PCAN-USB [15].

PCAN-View works in conjunction with PCAN-USB, which enables the connection to CAN network. It is an adapter with the following features [15]:

- Adapter for the USB connection.
- CAN connection according to ISO 11898-2
- Bit rates up to 1 Mbit/s
- It includes a NXP SJA1000 CAN controller and a NXP PCA82C251 CAN transceiver.
- CAN bus connection via D-Sub, 9-pin. Figure 17 illustrates the pin assignment.

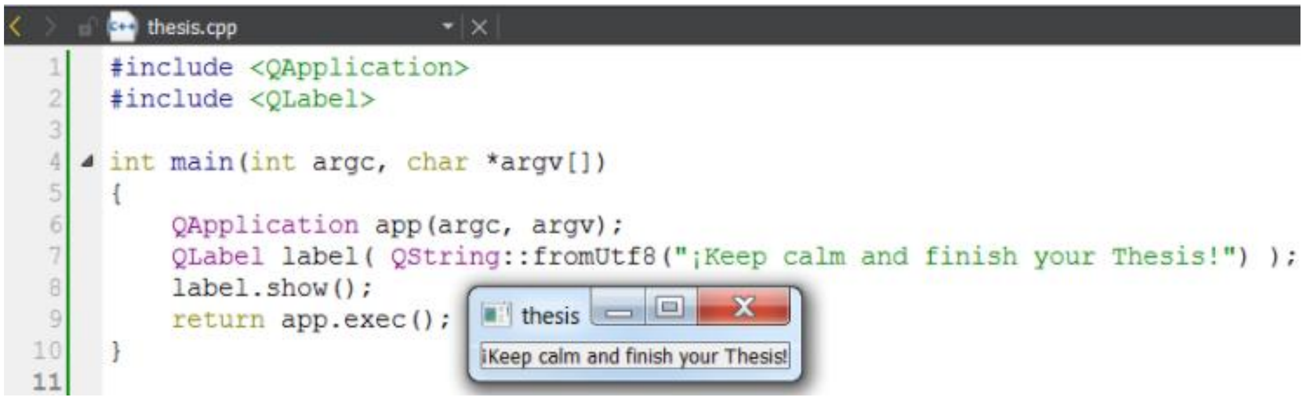
2.4 Qt

Qt is a complete cross-platform software framework with a wide range of integrated tools for software development. It is used to extend the C++ language with intuitive and comprehensive libraries and thus, applications using it can be compiled by any C++ compiler [16]. Qt is a complete software development framework with a large number of resources such as libraries for data storage, multimedia, network and connectivity, location and positioning, graphics, web content, and so on [16].

The Qt Integrated Development Environment (IDE) is named Qt Creator. Qt Creator IDE includes an intuitive interface, a code editor with syntax highlighting and auto completion, a UI creation, a visual debugger, and other integrated tools. Furthermore, Qt supported platforms include [16]:

- **Desktop:** Linux, macOS, Windows.
- **Embedded and RTOS:** Linux, QNX, VvWorks, Windows CE
- **Mobile:** Android, iOS, Windows

In this thesis Qt version 5.6.1 is used. Figure 18 is an example using Qt, where a window with a label (QLabel) is created showing the text ¡Keep calm and finish your Thesis!.



```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel label( QString::fromUtf8("¡Keep calm and finish your Thesis!") );
    label.show();
    return app.exec();
}
```

Figure 18. Qt example.

2.5 SQLite

"SQLite is the most used relational database engine in the world" due to the following features [17]:

- **Self-contained**. It uses no external libraries and the entire library is encapsulated in a single source code file.
- **High-reliability**. For more than a decade it has been used in billions of devices.
- **Embedded**. It is stored on a memory disk, and thus, any program that can access the disk can read and write directly from the database.
- **Full-featured**. SQLite has all the functions and capabilities as a SQL database, such as tables, views, index, triggers, and so on.
- **Public-domain.** SQLite does not require a license, and all the documentation is available in public domain.

2.6 Web Technologies

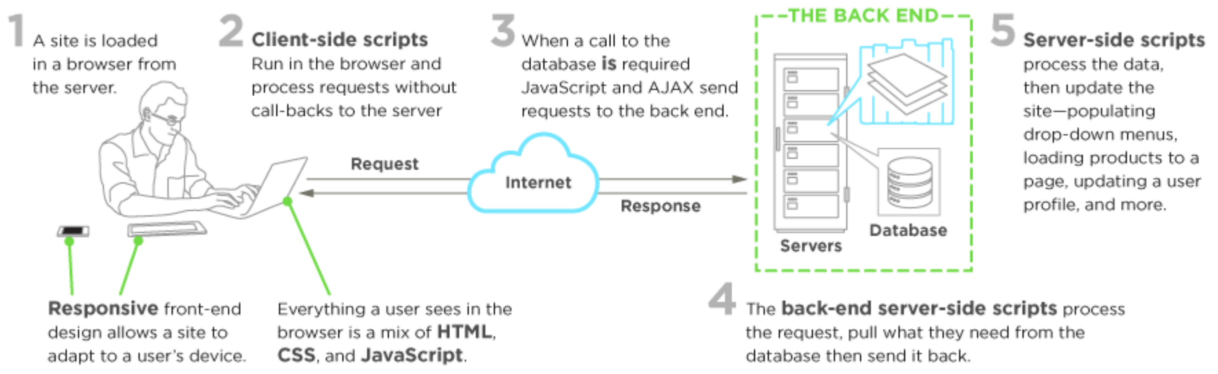

1 A site is loaded in a browser from the server.
2 Client-side scripts Run in the browser and process requests without call-backs to the server
3 When a call to the database is required JavaScript and AJAX send requests to the back end.
THE BACK END
5 Server-side scripts process the data, then update the site—populating drop-down menus, loading products to a page, updating a user profile, and more.
Request
Internet
Response
Servers
Database
Responsive front-end design allows a site to adapt to a user's device.
Everything a user sees in the browser is a mix of HTML, CSS, and JavaScript.
4 The back-end server-side scripts process the request, pull what they need from the database then send it back.

Figure 19. Web Technologies [18].

Web technologies refers to the technologies used to develop applications, solutions or tools used in the World Wide Web. Applications designed with web technology can be separated into two groups depending on their use. The first group is the front-end applications (Client-side), which involve interaction with the user, that is, everything the user observes, selects or manipulates, and the clearest example is a web page. Some examples of technology that focuses on developing front-end applications are HTML, CSS, and JavaScript.

The second group of web applications is those called back-end applications (Server-side), and its function is to interact with the server. It is all that the user cannot observe, for example, instructions that the server performs such as the page update process or storing information in a database. For this reason and since they allow interaction with the database, programming languages used to work in back-end applications are PHP, .NET, ASP, Perl, among others. Figure 19 describes the interaction between the types of web technologies.

As mentioned in chapter 1.3, the present project involves accessing to CAN bus information through an online interface. It was developed using HTML, CSS and JavaScript for describing the web page structure and its presentation. Moreover, the web page displays the CAN Bus data through charts and graphs which are deployed by means of the CanvasJS library. It is a charting library for HTML5 which rendering across devices such as iPhone, iPad, Android, Mac, and windows [19]. In addition, CanvasJS presents benefits such as simple API, 30 charts types, supports Chrome, Firefox, Safari, IE8+ and, it can render 100,000 Data-Points in milliseconds [19].

The online interface used PHP to create the server instructions. It is a scripting language designed mainly for web development [20]. It is a server-side language which employs scripts to generate responses for each web application request [20]. These responses can include queries to the database to load products, update the user's profile, as well as the data processing to update the site accordingly, etc.

3 Requirements Specification & Methodology

3.1 Objective

Design and implementation of a vehicle monitoring system to retrieve, store, and analyze the CAN Bus data.

3.2 System requirements

1. **Developing an on-board computer software:**

- The main purpose of the on-board computer is to retrieve the CAN bus data.
- This application will run on the IVU.box, taking into account that it must not affect the applications that are currently running on the device.
- The application should be configurable to define the relevant data to be stored**.**
- The on-board storage of the retrieved data must take into account the available disk space and it must not obstruct the storage of the generated data by other applications.
- The retrieved data has to be transferred to the back-office.

2. **Developing a database on the back-office system:**

- The database must present a proper organization of the retrieved data.
- Predictive analytics uses historical data to predict future events. Typically, historical data is used to build a mathematical model that captures important trends. Therefore, the proper storage of the retrieved data will contribute to the later implementation of a predictive maintenance system.

3. **Developing a service on the back-office system:**

- The service must be able to process the file containing the CAN bus information generated by the application running on the on-board computer.
- The service must be executed automatically once a day.
- The main goal of the service is to process the file which has been generated by the on-board computer software and store its data (CAN Bus data) in the back-office database.

4. **Developing an online interface:**

- Provide a backend or other viewing tool for a website that can be used to browse the data.

- The final step is to generate a frontend viewer such as web page to provide access to the information.

The following Figure 20 gives an overview of the project.

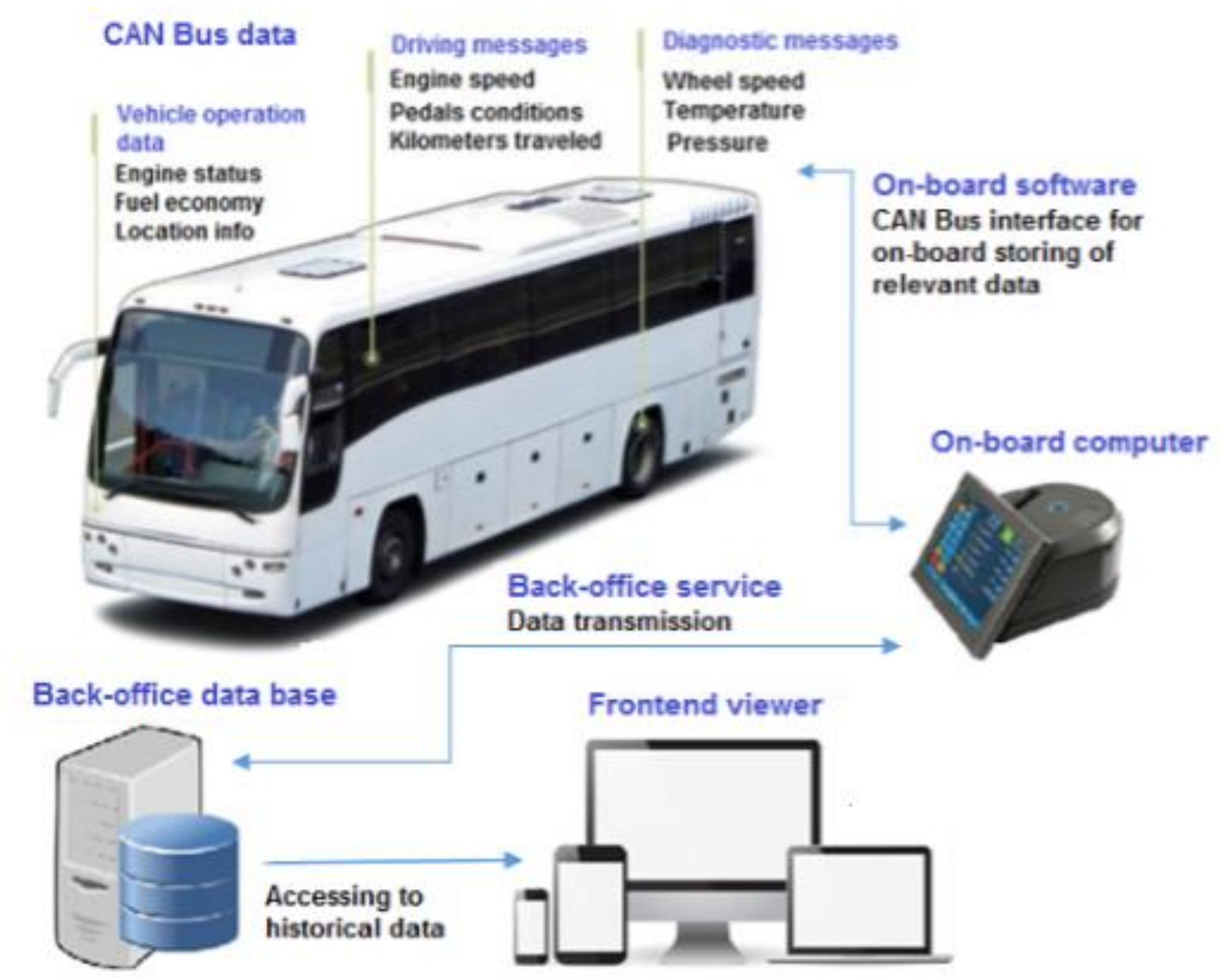


Figure 20. System overview.

3.3 Methodology Adopted

1. Identify functios and capabilities of IVU.box.

a) Operating system (OS).

b) Amount of available memory space and disk space.

c) Type of CAN bus hardware used.

d) Type of communication with back-office server.

2. Select the tool to store the CAN bus data on board.

The system stores on the onboard computer in a lightweight manner the data retrieved according to the amount of available disk space. Once the information has been transmitted to the back-office system, the software removes the data from the onboard computer.

3. Create the on-board computer software with the following features:

a) It runs on WinCE 7.

b) The software main goal is retrieving the CAN bus data.

c) It does not affect the applications that are currently running on the device.

d) The application defines the relevant data to be stored.

e) The retrieved data is transferred to the back-office.

4. Analyze the current back-office server.

a) Operating system (OS).

b) Current storage process for customer and vehicle information according to the different products offered by IVU.

5. Develop a data base in the back-office server.

The database represents a proper organization of the retrieved data. In the future, the information can be used for predictive analysis methods.

6. Develop a back-office service.

The main goal of the service is to process the CAN bus data generated by the on-board computer software and properly store it in the back-office database. The service is executed automatically once a day.

7. Create an online interface.

It consists of a web page to provide access to the information. It takes into account a log in procedure to visualize the information according to clients, vehicles and desired information.

3.4 Use cases

According to the mentioned requirements and the adopted methodology, the following Figures 21, 22, and 23 illustrate use cases to describe the different system interactions.

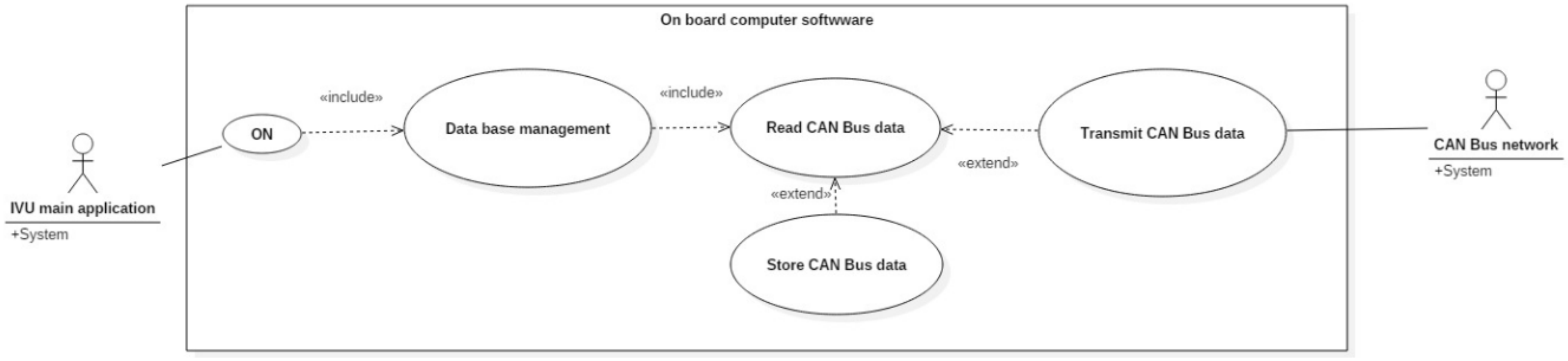


Figure 21. On board computer software diagram.

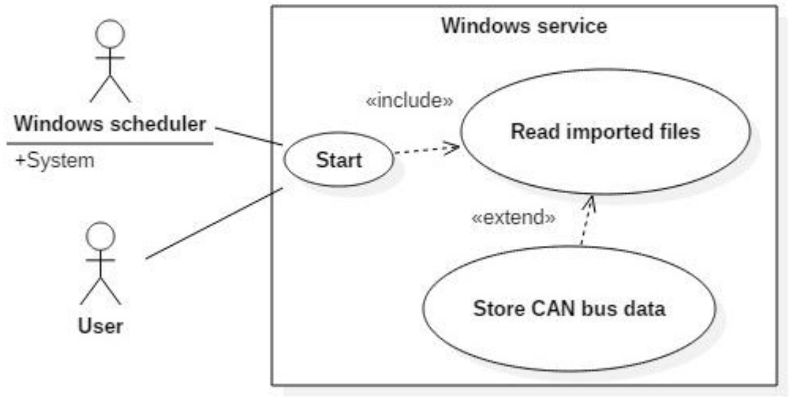


Figure 22. Windows service diagram.

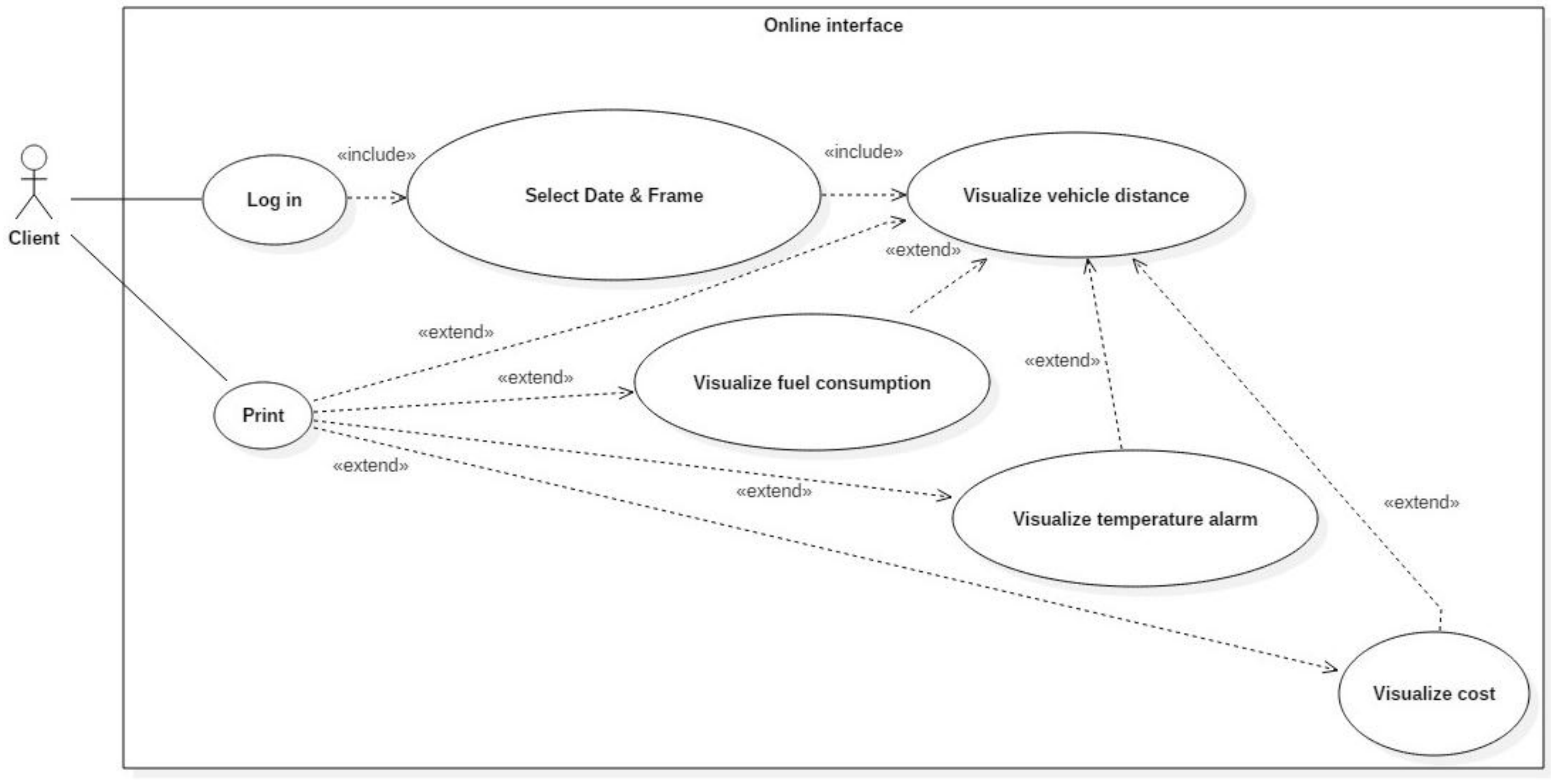


Figure 23. Online interface diagram.

3.5 Test cases

The following test cases were established in order to verify the expected results according to the proposed use cases.

1. Test the On board computer system. (System in Figure 21).
2. Test the Windows service system. (System in Figure 22).
3. Test the Online interface system. (System in Figure 23).

The chapter 5.1 presents the description of the tests as well as its execution process according to established pre-requisites in order to compare the expected results with the actual results.

Hardware & Software requirements for development

Project area	**Software requirements**	**Hardware requirements**
On-board computer software	• Windows embedded CE 7 • Qt creator version 5.6.1 • PCAN-View version 4.1.3.505 • GFCAN32 dynamic library. Copyright © 2000-2004, Garz & Fricke GmbH, Hamburg.	• IVU.box • Cortex M3 microcontroller • ISO 1050 CAN transceiver • PCAN-USB
Back-office service	• Windows 7 • Qt creator version 5.6.1 • Windows task scheduler	• Desktop PC
Online interface	• Apache web server module of XAMPP version 7.1.10 • Web browser	• Any device that has access to the IVU network.

Table 1. Hardware & Software requirements for development.

4 Implementation

4.1 On-board computer software

The objective of the on-board computer software is to allow reading and storage of CAN Bus frames for further analysis. It is executed on Windows CE 7, which is the operating system of the IVU.box. The IDE used to develop the application was QtCreator, and therefore, the programming language adopted is C++. Moreover, it has the following features:

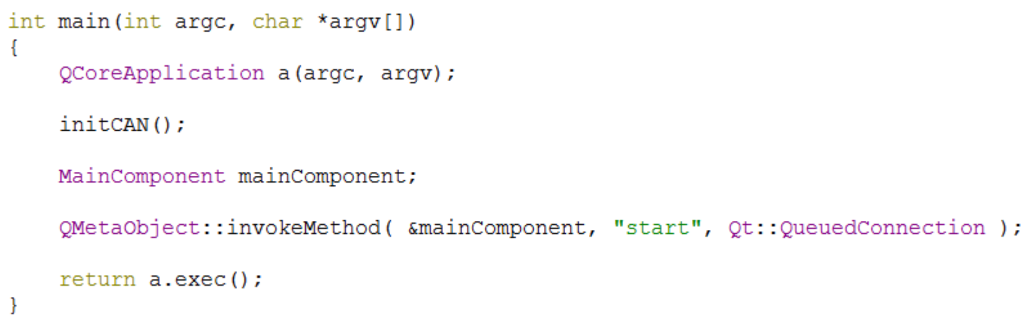
- It reads CAN bus frames that adhere to SAE J1939 standard.
- It is able to filter the CAN Bus frames.
- It utilizes an SQLite database to store information locally.

4.1.1 Initialization

As mentioned in chapter 2.2.2.2, the GFCan32.dll dynamic library allows access to the CAN module installed in the IVU.box. Furthermore and according to the library documentation, the following steps have to be taken when using GFCan32 within an application [14]:

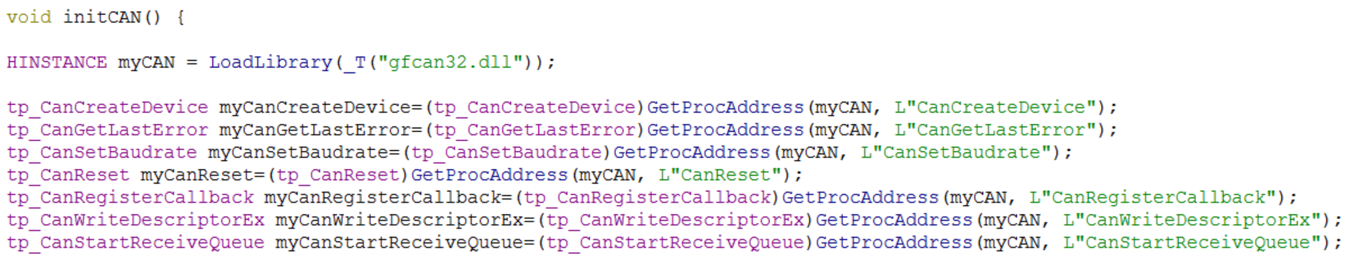
- Reset of the CAN-hardware
- Setting the transfer speed (BAUD-rate)
- Registering the CAN-callback
- Initializing of the CAN-controller in a valid descriptor for the message-filtering
- Activation of the background-thread
- Processing messages

To this end, a function called initCAN () was created, which aims to load the library and define the functions necessary to use it. It is the second instruction executed in the main function of the program. With this, access and adequate manipulation of the CAN module is achieved.



```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    initCAN();

    MainComponent mainComponent;

    QMetaObject::invokeMethod( &mainComponent, "start", Qt::QueuedConnection );

    return a.exec();
}
```

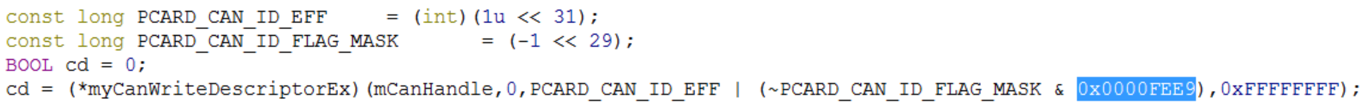
Figure 24. Main function.



```
void initCAN() {

HINSTANCE myCAN = LoadLibrary(_T("gfcan32.dll"));

tp_CanCreateDevice myCanCreateDevice=(tp_CanCreateDevice)GetProcAddress(myCAN, L"CanCreateDevice");
tp_CanGetLastError myCanGetLastError=(tp_CanGetLastError)GetProcAddress(myCAN, L"CanGetLastError");
tp_CanSetBaudrate myCanSetBaudrate=(tp_CanSetBaudrate)GetProcAddress(myCAN, L"CanSetBaudrate");
tp_CanReset myCanReset=(tp_CanReset)GetProcAddress(myCAN, L"CanReset");
tp_CanRegisterCallback myCanRegisterCallback=(tp_CanRegisterCallback)GetProcAddress(myCAN, L"CanRegisterCallback");
tp_CanWriteDescriptorEx myCanWriteDescriptorEx=(tp_CanWriteDescriptorEx)GetProcAddress(myCAN, L"CanWriteDescriptorEx");
tp_CanStartReceiveQueue myCanStartReceiveQueue=(tp_CanStartReceiveQueue)GetProcAddress(myCAN, L"CanStartReceiveQueue");
```

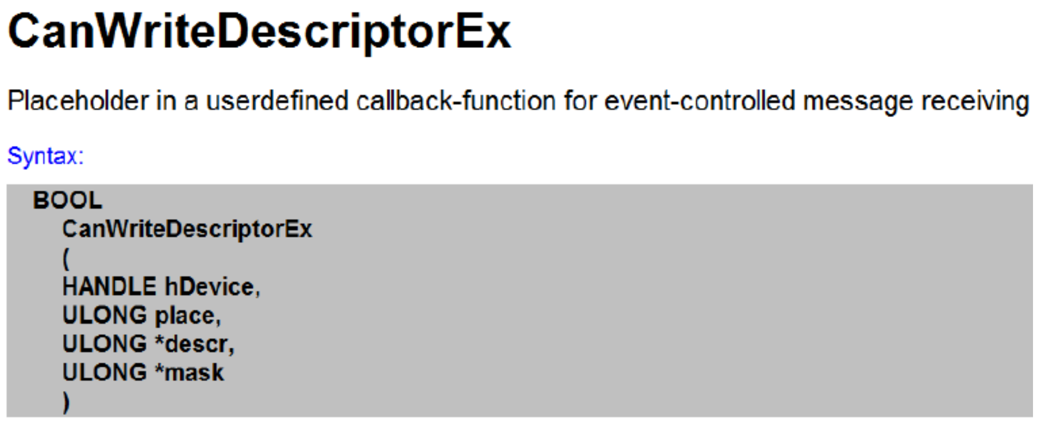
Figure 25. initCAN() function.

As shown in Figure 24, the dynamic library is loaded by the application and thus, an HINSTANCE is created. The functions to manipulate the CAN module are exported by the library, however, the initCAN() function must get the pointers so that the functions can later be called by the code. After this initialization process, it is possible to access the CAN module and define parameters such as the baud rate, the filter mask and the filter descriptor, among others.



```
const long PCARD_CAN_ID_EFF      = (int)(1u << 31);
const long PCARD_CAN_ID_FLAG_MASK        = (-1 << 29);
BOOL cd = 0;
cd = (*myCanWriteDescriptorEx)(mCanHandle,0,PCARD_CAN_ID_EFF | (~PCARD_CAN_ID_FLAG_MASK & 0x0000FEE9),0xFFFFFFFF);
```

Figure 26. CanWriteDescriptorEx function.



CanWriteDescriptorEx

Placeholder in a userdefined callback-function for event-controlled message receiving

Syntax:

```
BOOL
    CanWriteDescriptorEx
    (
    HANDLE hDevice,
    ULONG place,
    ULONG *descr,
    ULONG *mask
    )
```

Figure 27. CanWriteDescriptorEx function structure [14].

Figure 26 illustrates the use of one of the exported functions, in this particular case CanWriteDescriptorEx function. In addition, as shown in Figure 27, the library documentation provides the structure and the necessary parameters to execute the mentioned function. Moreover, the CanWriteDescriptorEx function is used to create the CAN Bus filter, which is an essential feature of the software. Thus, it requires the following parameters [14]:

- **HANDLE hDevice.** Handle of the CAN device, it is created via CanCreateDevice function.
- **ULONG place**. Number of the messageplace that should be read.

- **ULONG *descr**. Address of a 32-bit variable, where the desired descriptor is stored into
- **ULONG *mask**. Address of a 32-bit variable, where the desired mask is stored into.

According to the above mentioned, Figure 26 shows the implementation of a function that allows filtering the frames with the 0xFEE9 identifier. The const long variables (PCARD_CAN_ID_EFF & PCARD_CAN_ID_FLAG_MASK) together with the identifier (0xFEE9) are used to create a 32-bit descriptor variable, moreover, a 0xFFFFFFFF mask value is defined. In this example, it is obtained the following binary values for the descriptor and the mask:

Frame ID = 1111111011101001 (16 bits)

Descriptor = 10000000000000001111111011101001 (32 bit)

Mask = 11111111111111111111111111111111 (32 bits)

The filter mask and the filter descriptor are used to define which Frames will be accepted [14]. If a bit mask is set to a zero, the corresponding ID bit will be accepted, regardless of the value of the bit descriptor. If a bit mask is set to one, the corresponding ID bit will be compared with the value of the bit descriptor. If they match it is accepted otherwise the frame is rejected. Since the mask only includes numbers 1, only frames with 0xFEE9 ID will be accepted.

4.1.2 Classes & Objects

The Qt Object Model provides very efficient runtime support and a high level of flexibility for the object paradigm. Therefore, "*Qt adds the following features to C++"* [16]:

- Object communication called signals and slots.
- Queryable and designable object properties.
- Events and event filters.
- Contextual string translation for internationalization.
- Even-driven GUI.
- Hierarchical and queryable object tress.
- Guarded pointers that are automatically set to 0 when the referenced object is destroyed
- A dynamic cast that works across library boundaries

Furthermore, Qt also provides modules with many useful functions implemented in an operating system independent way. In the present work, the QDatabase module was used for SQLite integration as well as QFile module in order to provide an interface for reading from and writing to files.

For the above, the on-board computer software has been developed with the following three classes:

- **MainComponent class**. It prepares the database for its transmission to the back-office system and, it creates an instances of DataHandler class and CANReader class.
- **CANReader class**. It retrieves the CAN Bus data.
- **DataHandler class**. It stores the retrieved data in the SQLite database.

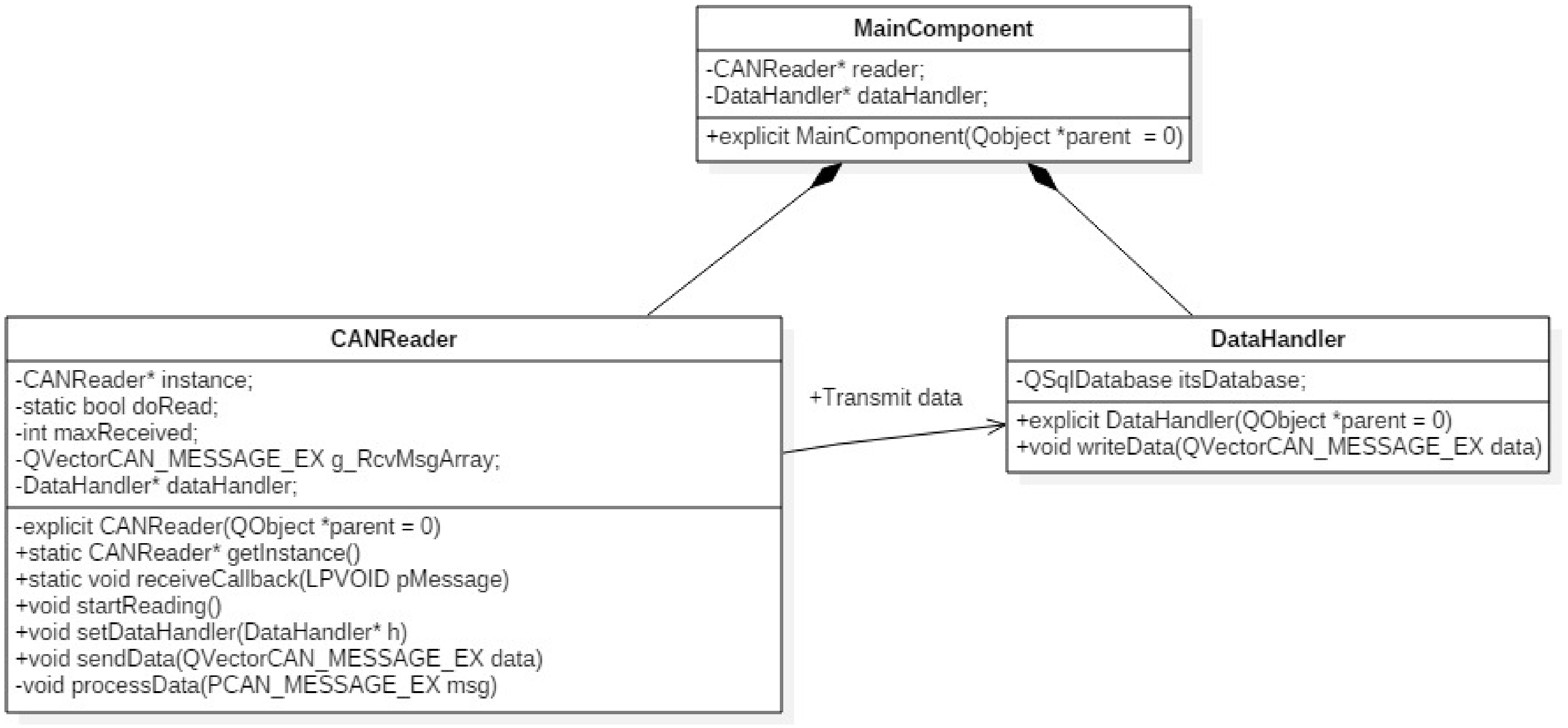

MainComponent
-CANReader* reader;
-DataHandler* dataHandler;
+explicit MainComponent(Qobject *parent = 0)
CANReader
-CANReader* instance;
-static bool doRead;
-int maxReceived;
-QVectorCAN_MESSAGE_EX g_RcvMsgArray;
-DataHandler* dataHandler;
-explicit CANReader(QObject *parent = 0)
+static CANReader* getInstance()
+static void receiveCallback(LPVOID pMessage)
+void startReading()
+void setDataHandler(DataHandler* h)
+void sendData(QVectorCAN_MESSAGE_EX data)
-void processData(PCAN_MESSAGE_EX msg)
+Transmit data
DataHandler
-QSqlDatabase itsDatabase;
+explicit DataHandler(QObject *parent = 0)
+void writeData(QVectorCAN_MESSAGE_EX data)

Figure 28. Class diagram of the on-board computer software.

Figure 28 shows the relationship among the different classes used by the application. From the diagram, it can be deduced that CANReader class and DataHandler class are part of the MainComponent class, and therefore, objects of these classes cannot exist without a MainComponent object. In addition, it can be inferred that there is also an association between classes CANReader and DataHandler and, it can be interpreted as the data emission from CANReader class to DataHandler class.

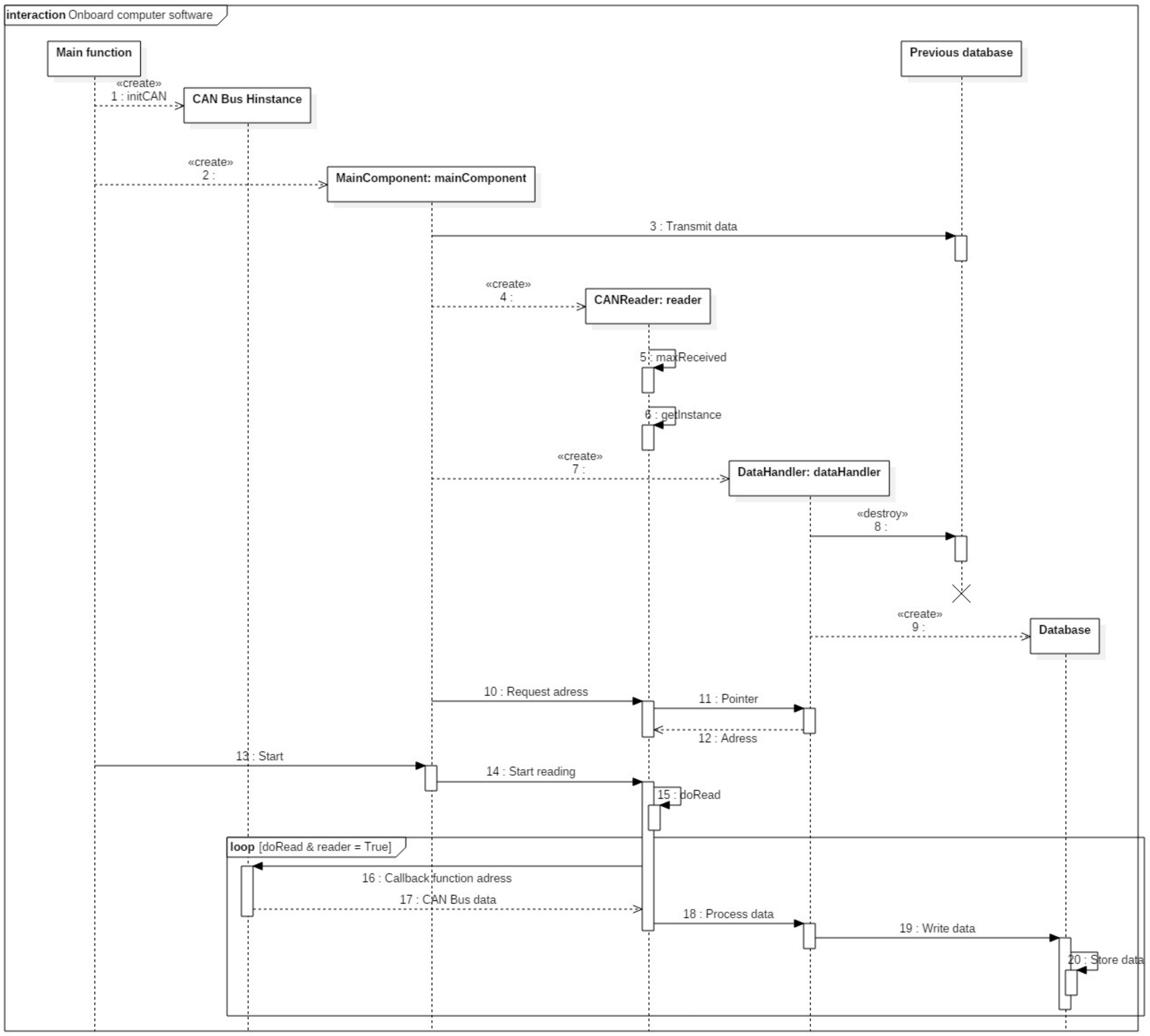


Figure 29. Onboard computer software sequence diagram.

Figure 29 illustrates the interaction among the different functions and objects which have been implemented in the software. Furthermore, it can be deduced that the process of reading and storing the CAN bus data is achieved in 20 steps, which are described below (Functions and objects used in the following steps are organized in the class diagram of Figure 28).

- **Step 1**. CAN Bus Hinstance is created via the initCAN() function. It allows the proper handling of the CAN Bus module. As mentioned above, Figure 25 shows the function structure.

- **Step 2.** mainComponent object is created from MainComponent class. As can be seen in Figure 24, the main function executes the instantiation of this object.

- **Step 3.** explicit MainComponent (QObject *parent = 0) function is executed. It is the constructor function of the mainComponent object and its first action is to prepare the previous data base to be exported to the back-office server. It is possible only if actually a previous data base exists (a previous data base is a database created by the on-board computer software in the previous session). Moreover, in order to transmit information to the back-office server through the already established functions of the IVU.box, it is necessary to copy the desired database to a special folder called ***logger***. It is achieved via the mainComponent constructor.

- **Step 4.** reader object is created from CANReader class. It is also done via the explicit MainComponent (QObject *parent = 0) function.

- **Step 5.** explicit CANReader(QObject *parent = 0) function is executed. It is the constructor function of the reader object and it is used to define the value of int maxReceived. It is an integer value which determines the number of frames that the application will temporarily store. This is a kind of buffer which allows to temporarily store a specific amount of frames. Once the buffer has reached its maximum value (int maxReceived), then the frames are actually stored in the database in **one single transaction**.

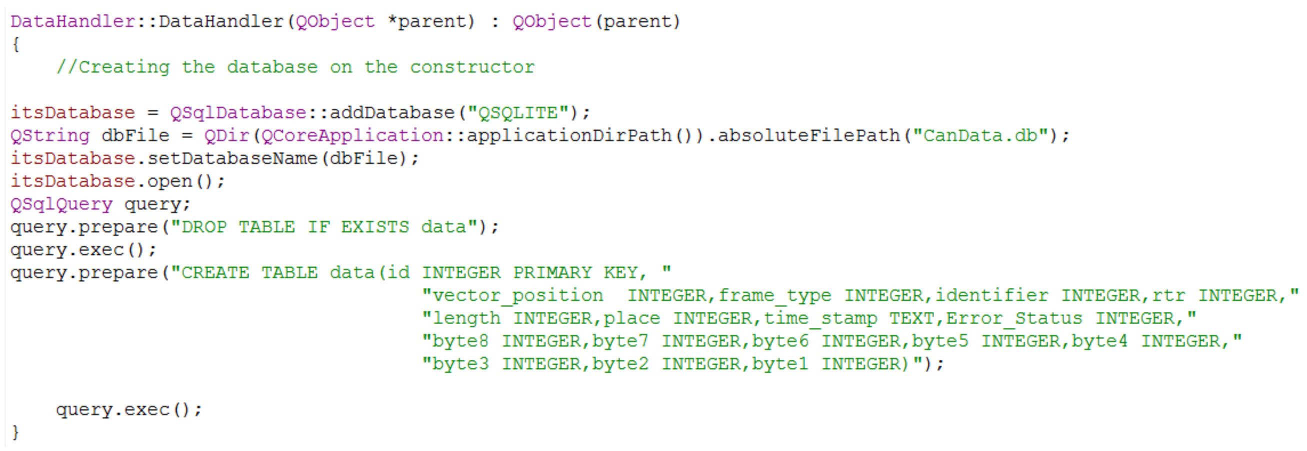
  The hard disk of the IVU.box is an SD memory and since writing many times on the card can lead to memory corruption, it is necessary to use the buffer. The buffer implementation prevent the memory from being damaged because it avoids writing considerable times on the card.

- **Step 6.** getInstance() function is executed**.** It is designed to verify if an instance of the CANReader class already exists. At this point, an instance of the CANReader class exists, it is the reader object. Therefore this function should return a TRUE value. Thus, it can be considered an implementation of the singleton pattern (CANReader is a singleton).

- **Step 7**. dataHandler object is created from DataHandler class. It is also done via the explicit MainComponent(QObject *parent = 0) function.

- **Step 8.** explicit DataHandler(QObject *parent = 0) function is executed. It is the constructor function of the dataHandler object. It is used to **destroy** the previous data base.

- **Step 9.** It is used to create the new data base and it is also done via the explicit DataHandler(QObject *parent = 0) function. As mentioned in chapter 4.1, the application utilizes an SQLite database, which is created by means of the QSqlDatabase class. In addition, the following code in Figure 30 exemplifies the process to create a new database.



```
DataHandler::DataHandler(QObject *parent) : QObject(parent)
{
    //Creating the database on the constructor

itsDatabase = QSqlDatabase::addDatabase("QSQLITE");
QString dbFile = QDir(QCoreApplication::applicationDirPath()).absoluteFilePath("CanData.db");
itsDatabase.setDatabaseName(dbFile);
itsDatabase.open();
QSqlQuery query;
query.prepare("DROP TABLE IF EXISTS data");
query.exec();
query.prepare("CREATE TABLE data(id INTEGER PRIMARY KEY, "
                                    "vector_position  INTEGER,frame_type INTEGER,identifier INTEGER,rtr INTEGER,"
                                    "length INTEGER,place INTEGER,time_stamp TEXT,Error_Status INTEGER,"
                                    "byte8 INTEGER,byte7 INTEGER,byte6 INTEGER,byte5 INTEGER,byte4 INTEGER,"
                                    "byte3 INTEGER,byte2 INTEGER,byte1 INTEGER)");

    query.exec();
}
```

Figure 30. SQLite data base code.

In Figure 30 the following is shown:

a) itsDatabase object is created from DataHandler class. It allows the access to the database functionalities offered by Qt. In particular, the itsDatabase object represents an SQLite database.
b) Since SQLite stores the whole database in one file, this is also the filename of the database ("CanData.db").
c) The database is created in the same folder where the application is located.
d) The data generated in the last session is deleted. However, it has been copied to the logger directory before, so the data is not lost.
e) The database only includes one table with the following columns:
   id INTEGER PRIMARY KEY,
   vector_position INTEGER,
   frame_type INTEGER,
   identifier INTEGER,
   rtr INTEGER,
   length INTEGER,
   place INTEGER,
   time_stamp TEXT,
   Error_Status INTEGER,
   byte1-8 INTEGER (8 columns, one column for each byte).

- **Step 10.** mainComponent object indicates to reader object that it must request the memory address of dataHandler object.

- **Step 11**. reader object creates a pointer to dataHandler object.
- **Step 12**. reader object actually retrieves the memory address of dataHandler object. Therefore, through this process it is possible to establish the communication channel between reader object and dataHandler object.

- **Step 13.** Main function executes start function which is a method of mainComponent object, it is done through QMetaObject::invokeMethod (Figure 24).

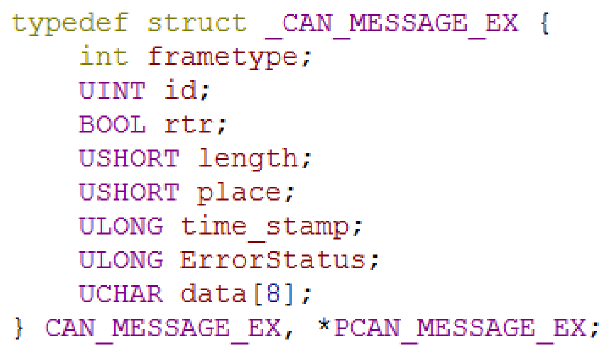
- **Step 14**. The start function indicates to reader object that it must define the value of static bool doRead. It is a bool variable which is used to control the looping of the CAN Bus data reading process, as long as static bool doRead and getInstance() are true, the loop shown in figure 29 will be executed.

- **Step 15**. Reader object defines a TRUE value for the static bool doRead.

- **Step 16**. Reader object provides the memory address of the static void receiveCallback(LPVOID pMessage) function to the CAN Bus Hinstance. It is a special function used by the GFCAN32.dll library to store the data retrieved from the CAN Bus. This function is designed as placeholder only.

- **Step 17**. Once the CAN Bus Hinstance get a memory address to send the retrieved data, in this case the receiveCallback function address, it is possible to start reading the CAN Bus. Therefore, at this moment the CAN Bus data emission begins.

- **Step 18**. Reader object executes processData(PCAN_MESSAGE_EX msg) function.



```
typedef struct _CAN_MESSAGE_EX {
    int frametype;
    UINT id;
    BOOL rtr;
    USHORT length;
    USHORT place;
    ULONG time_stamp;
    ULONG ErrorStatus;
    UCHAR data[8];
} CAN_MESSAGE_EX, *PCAN_MESSAGE_EX;
```

Figure 31. Retrieved data structure.

processData(PCAN_MESSAGE_EX msg) function uses as a parameter a pointer to the structure shown in Figure 31. This structure is used to store the retrieved data from the CAN Bus. Actually, the function creates a vector of this structure to store the data (QVector<CAN_MESSAGE_EX> g_RcvMsgArray).

- **Step 19.** Once g_RcvMsgArray has the same number of elements as int maxReceived, then processData(PCAN_MESSAGE_EX msg) function transmits the vector to dataHandler object.
- **Step 20.** dataHandler object executes writeData(g_RcvMsgArray) function. This function stores the vector in the CanData.db. As can be assumed from the previous steps, g_RcvMsgArray contains the CAN Bus data.

- **Loop.** According to the diagram, steps 16, 17, 18, 19 and, 20 are within an infinite loop, as long as static bool doRead and getInstance() are true. Therefore, the process of retrieving and storing the CAN Bus data is continuous.

4.2 CAN Bus data transmission

4.2.1 Back-office server

As mentioned in the methodology, in order to collect the CAN Bus data from the IVU.box, it is necessary to analyze the back-office server. Therefore and following the products offered by IVU:

A) It uses Windows 7 as the operating system.
B) It has a remote data transmission process. IVU.box employs a telematics gateway to transmit the collected data from the CAN bus to the back-office system. The data is collected in the logger folder on the IVU.box, and then the folder is compressed and transmitted to the server. Once the logger folder is received on the server, it is saved according to the corresponding tenant (IVU client) and the corresponding vehicle in the system files. Finally, the server indicates to the IVU.box that the logger folder has been received and it can be deleted from its storage. The data transmission process can be seen in the following Figure 32.

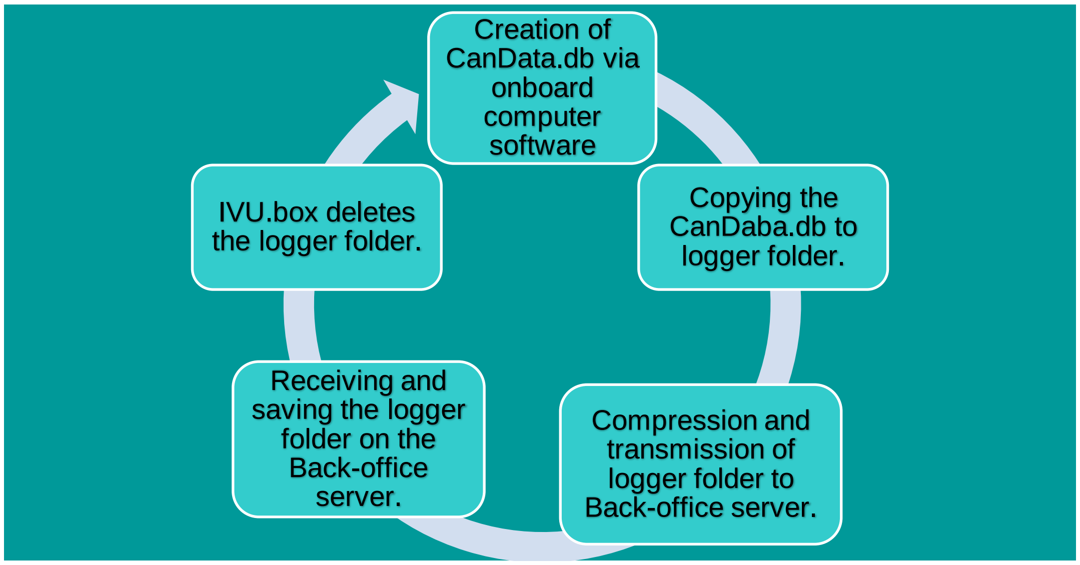
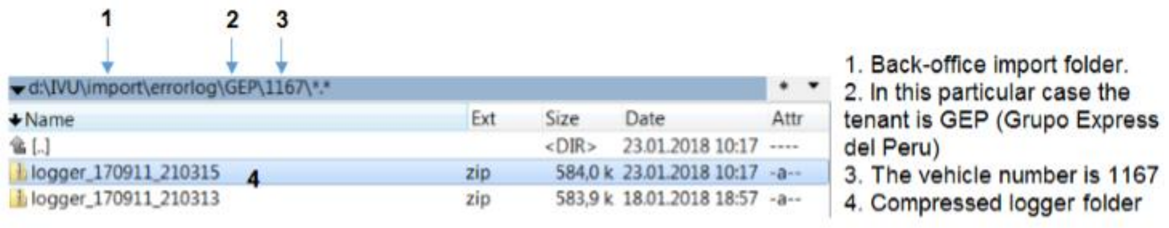

Creation of CanData.db via onboard computer software
Copying the CanDaba.db to logger folder.
Compression and transmission of logger folder to Back-office server.
Receiving and saving the logger folder on the Back-office server.
IVU.box deletes the logger folder.

Figure 32. Data transmission process

C) The back-office server receives the logger folder, and it is saved according to the tenant and vehicle number. The following Figure 33 illustrates the location of the logger folder on the back-office server (d:\IVU\import\errorlog\GEP\1167\).



```
d:\IVU\import\errorlog\GEP\1167\*.*
Name                      Ext   Size     Date              Attr
[..]                            <DIR>    23.01.2018 10:17  ----
logger_170911_210315      zip   584,0 k  23.01.2018 10:17  -a--
logger_170911_210313      zip   583,9 k  18.01.2018 18:57  -a--
```

1. Back-office import folder.
2. In this particular case the tenant is GEP (Grupo Express del Peru)
3. The vehicle number is 1167
4. Compressed logger folder

Figure 33. Logger folder location.

4.2.2 Back-office database

According to the features mentioned in the previous chapter, a database was created on the back-office server with the following characteristics:

- It has been developed to store all the CAN Bus data according to the different tenants and vehicles.
- It is a relational SQLite database which is used only as a placeholder, a full scale data base management system such as Oracle will be used in the future.
- Figure 34 illustrates the tables that compose the server database as well as the relationships between them.

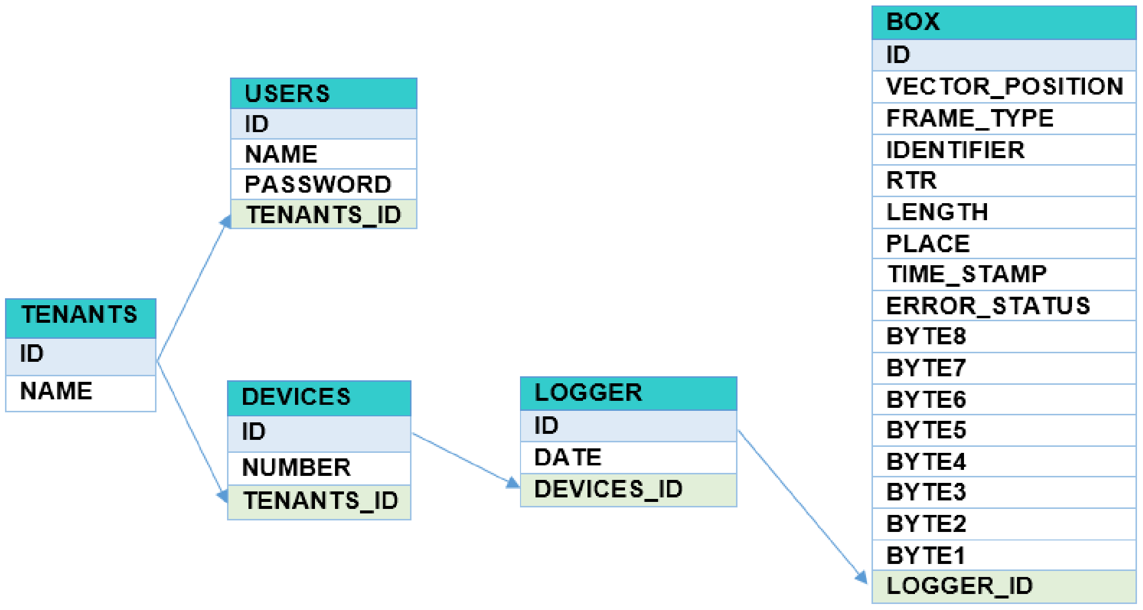


Figure 34. Tables in the server database.

Values stored in table tenants contain the name of the Tenants (IVU Clients). For example, GEP Grupo Express del Peru.

COLUMN	TYPE	CHILDREN	PARENTS	COMMENTS
ID	INTEGER	USERS DEVICES		PRIMARY KEY
NAME	TEXT			TENANT NAME

Table 2. Tenants.

Values stored in users table contain mainly the user credentials to access the online interface.

COLUMN	TYPE	CHILDREN	PARENTS	COMMENTS
ID	INTEGER			PRIMARY KEY
NAME	TEXT			USER NAME
PASSWORD	TEXT			USER PASSWORD
TENANT_ID	INTEGER		TENANTS	FOREIGN KEY

Table 3. Users.

Values stored in devices table contain the vehicle number (device number) according to the proper MANDANT.

COLUMN	TYPE	CHILDREN	PARENTS	COMMENTS
ID	INTEGER	LOGGER		PRIMARY KEY
NUMBER	INTEGER			VEHICLE NUMBER
TENANT_ID	INTEGER		TENANTS	FOREIGN KEY

Table 4. Devices.

Values stored in logger table contain the CAN Bus data properties such as tenant name, vehicle number and emission date to the back-office server.

COLUMN	TYPE	CHILDREN	PARENTS	COMMENTS
ID	INTEGER	BOX		PRIMARY KEY
DATE	TEXT			EMISSION DATE
DEVICES_ID	INTEGER		DEVICES	FOREIGN KEY

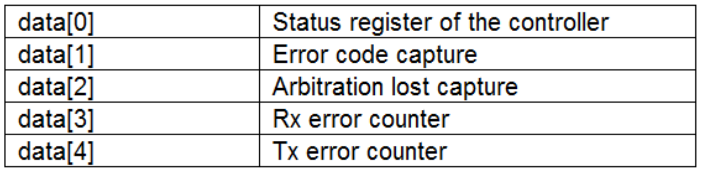
Table 5. Logger.

Values stored in box table contain mainly the retrieved CAN Bus data

COLUMN	TYPE	CHILDREN	PARENTS	COMMENTS
ID	INTEGER			PRIMARY KEY
VECTOR_POSITION	INTEGER			Frame position in vector g_RcvMsgArray (Chapter 4.1.2)
FRAME_TYPE	INTEGER			*"This structure supports messages with 11- and 29-bit identifier and can take the value STANDARD or EXTENDED* [14]*"*
IDENTIFIER	INTEGER			*"Identifier of the message. Can be 11 or 29 bits long* [14]*"*

RTR	INTEGER			*"Remote frame flag. If true, this value, it is a remote frame* [14]*"*
LENGTH	INTEGER			*"Message-lenght in bytes* [14]*"*
PLACE	INTEGER			*"Number of the message place, a message has been sent or received through* [14]*"*
TIME_STAMP	TEXT			*"Time stamp of the received message* [14]*"*
ERROR_STATUS	INTEGER			*"The data length is 5 bytes (Figure 35)* [14]*"*
BYTE8-1	INTEGER			*"Databytes of the message* [14]*"*
LOGGER_ID	INTEGER		LOGGER	FOREIGN KEY

Table 6. Box.



data[0]	Status register of the controller
data[1]	Error code capture
data[2]	Arbitration lost capture
data[3]	Rx error counter
data[4]	Tx error counter

Figure 35. Figure 35. ERROR_STATUS bytes [14].

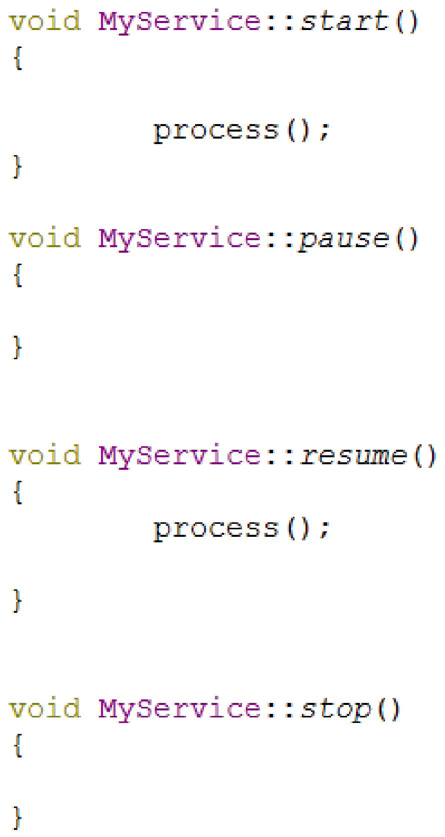
4.2.3 Windows service

Microsoft Windows services enable the creation of long-running executable applications that run in the background of Windows sessions. They can be automatically started when the operating system starts and they run as long as Windows is running. In addition, they can be paused and restarted either manually or according to scheduled events via Windows scheduler. "*Services are ideal for use on a server or whenever it is needed a long-running functionality"* [21].

According to the server characteristics and its data transmission process, a Windows service was implemented on the back-office server. It is designed to extract the CAN Bus data from the logger folder and store it in the server database. Moreover, it automatically runs once a day.

4.2.3.1 Service development

The Windows service was developed with QtCreator, and it uses QtService project (hosted on GitHub) as the basis for its implementation [22]. The QtService project provides a base class for a Windows service that will exist as part of a service application. Furthermore, when an instance of this class is created, it is possible to use three functions which correspond to the three possible states of the service (started, paused, resumed, and stopped). Thus, every time the service changes its state, the corresponding function is executed.



```
void MyService::start()
{

        process();
}

void MyService::pause()
{

}


void MyService::resume()
{
        process();

}


void MyService::stop()
{

}
```

Figure 36. Service functions [22].

Figure 36 illustrates the instructions for each service state. In this particular case, only the start and resume functions are used to execute the process() function. This function aims to extract the CAN Bus data from the logger folder and store it in the server database. Therefore, it can be concluded that each time that the service is started or resumed the CanBus data is extracted and stored. However, the pause functionality is currently not supported. The service needs to process everything and cannot be paused while it is processing.

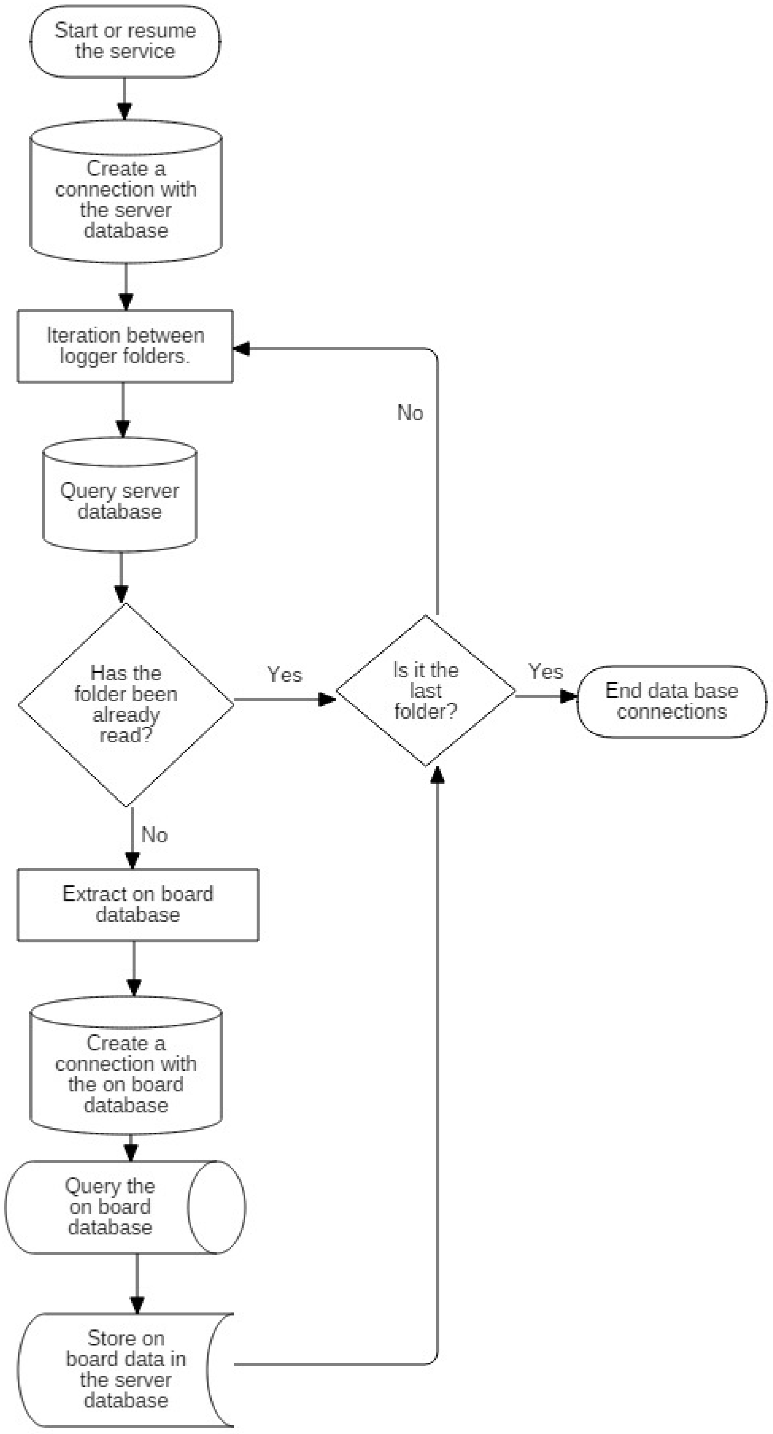

Start or resume the service
Create a connection with the server database
Iteration between logger folders.
Query server database
Has the folder been already read?
Yes
Is it the last folder?
Yes
End data base connections
No
No
Extract on board database
Create a connection with the on board database
Query the on board database
Store on board data in the server database

Figure 37. Service flow chart.

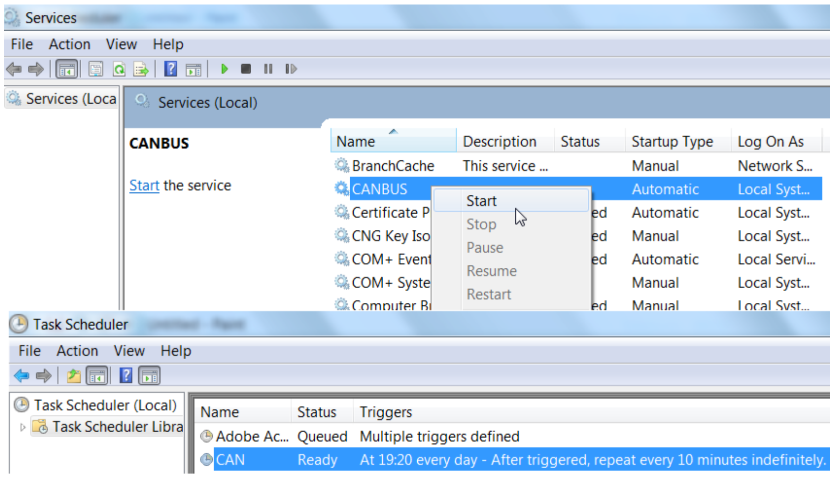
Figure 38 illustrates the service flow chart, from which can be deduced the following steps to extract and save the CAN Bus data:

1. First of all, the service must be started or resumed. As shown in figure 36, when the service is started or resumed, the process() function is executed.
2. It creates a connection with the server database.
3. It performs an iteration (one by one) through the several logger folders located on the server (Figure 33).
4. It queries the server database using the name of the folder as a parameter. Moreover it checks if the logger file is already imported, if so, the process continues with the iteration process. In addition, if the file is already imported and the current folder is the last one in the iteration process then the process concludes.
5. In case that the folder data has not yet been stored, then the service unzips the respective logger archive and extracts the on board database.
6. After that, the program creates a connection with the on board database.
7. It queries the on board database in order to obtain all its information which actually is all the retrieved data from the CAN Bus.
8. Finally, the program stores the CAN Bus data in the server data base and if the current logger folder is the last one in the iteration, then the process concludes. Otherwise the process continues with the iteration.

As explained above, the service was implemented with Qt, and as a result of this an exe file is obtained. This file is the basis for generating the long-running executable application (Windows service). In order to create the service in Windows, it is necessary to do the following 2 steps [22]:

- Run "cmd.exe" as administrator
- Execute the following command *sc create "CANBUS" binpath = "C: \ CAN.exe" start = auto*. Thus, CANBUS is the desired name for the service and binpath is the exe file address or another directory were the service application is installed.

Furthermore, the service can be started manually via “services” or it can be scheduled via “task scheduler” (Figure 38). In the present work the service is scheduled to be started once per day.



Name	Description	Status	Startup Type	Log On As
BranchCache	This service ...		Manual	Network S...
CANBUS			Automatic	Local Syst...
Certificate P		ed	Automatic	Local Syst...
CNG Key Iso		ed	Manual	Local Syst...
COM+ Event		ed	Automatic	Local Servi...
COM+ Syste			Manual	Local Syst...
Computer Bi		ed	Manual	Local Syst...

Name	Status	Triggers
Adobe Ac...	Queued	Multiple triggers defined
CAN	Ready	At 19:20 every day - After triggered, repeat every 10 minutes indefinitely.

Figure 38. Running the service.

4.3 Online interface

Finally, the present thesis includes the development of a web page (Online interface). As mentioned before, the primary objective of the online interface is to provide access to the retrieved CAN Bus data according to the different Tenants and their respective vehicles. Moreover, the online interface is a client-server computer program with three main participants:

1. The user interface (client):

    - It is used to perform requests to the server and display the information returned.
    - It is developed using HTML and JavaScript programming languages.
    - It displays a special window of graphics and results. This window is created via CanvasJS.
    - It runs in a web browser.

2. The server:

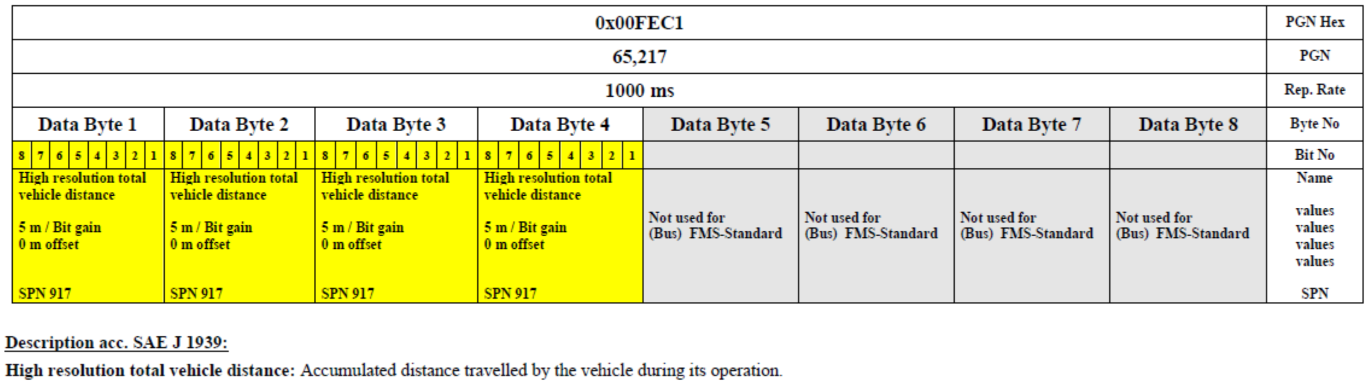
    - It is used to query the server database and process the resulting CAN Bus data.
    - The server instructions are created through scripts written with the PHP programming language.
    - It runs in an Apache HTTP Server.

3. The server database (Chapter 4.2.2).

4.3.1 Data processing

The online interface developed in the present thesis allows only the analysis of the following three CAN Bus frames (other frames are currently not supported):

- **Vehicle distance**



0x00FEC1								PGN Hex
65,217								PGN
1000 ms								Rep. Rate
Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8	Byte No
8 7 6 5 4 3 2 1	8 7 6 5 4 3 2 1	8 7 6 5 4 3 2 1	8 7 6 5 4 3 2 1					Bit No
High resolution total vehicle distance 5 m / Bit gain 0 m offset SPN 917	High resolution total vehicle distance 5 m / Bit gain 0 m offset SPN 917	High resolution total vehicle distance 5 m / Bit gain 0 m offset SPN 917	High resolution total vehicle distance 5 m / Bit gain 0 m offset SPN 917	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Name values values values values SPN

Description acc. SAE J 1939:

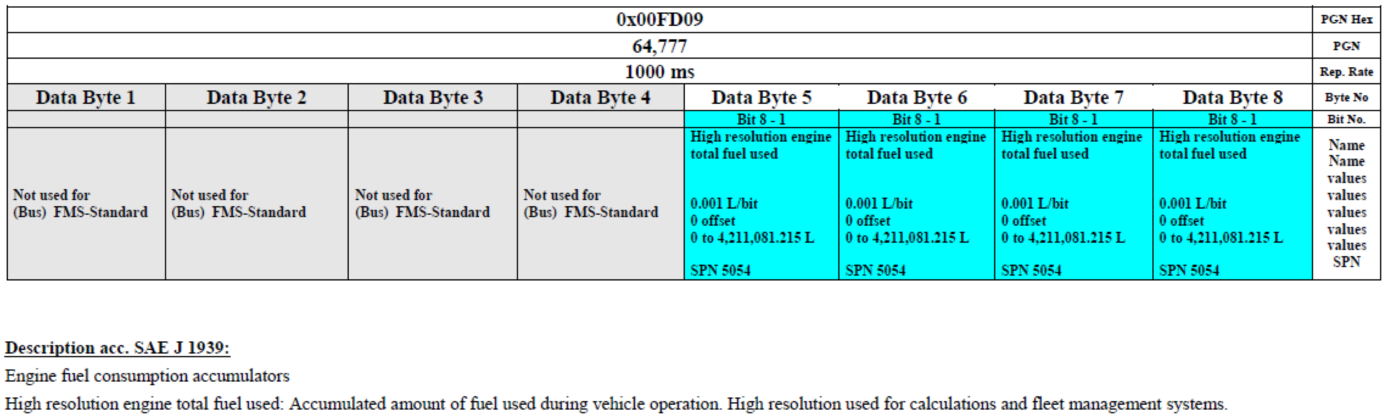
High resolution total vehicle distance: Accumulated distance travelled by the vehicle during its operation.

Figure 39. Vehicle distance frame [9].

Figure 39 illustrates the characteristics of the vehicle distance can bus frame according to the SAEJ1939 standard. From the Figure, the following can be deduced [9]:

a) It is used to indicate the accumulated distance travelled by the vehicle during its operation.
b) Its corresponding PGN number is 0x00FEC1. In the present work and for simulation purposes it is assumed that the PGN number is equal to the ID number. For future implementations it is necessary to provide the ID number, which depends on each vehicle.
c) Only the data in bytes1-4 must be considered. Therefore, the sensor resolution is 4 bytes (0xFFFFFFFF)
d) It has an offset of 0m and for each bit increment the sensed value increases by 5m. For instance, if the sensed value is 0x0000000F, it represents a distance traveled of 75 m.
e) It has a rate of 1000 milliseconds.

- **Fuel consumption**



0x00FD09								PGN Hex
64,777								PGN
1000 ms								Rep. Rate
Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8	Byte No
				Bit 8 - 1	Bit 8 - 1	Bit 8 - 1	Bit 8 - 1	Bit No.
Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	High resolution engine total fuel used 0.001 L/bit 0 offset 0 to 4,211,081.215 L SPN 5054	High resolution engine total fuel used 0.001 L/bit 0 offset 0 to 4,211,081.215 L SPN 5054	High resolution engine total fuel used 0.001 L/bit 0 offset 0 to 4,211,081.215 L SPN 5054	High resolution engine total fuel used 0.001 L/bit 0 offset 0 to 4,211,081.215 L SPN 5054	Name Name values values values values values SPN

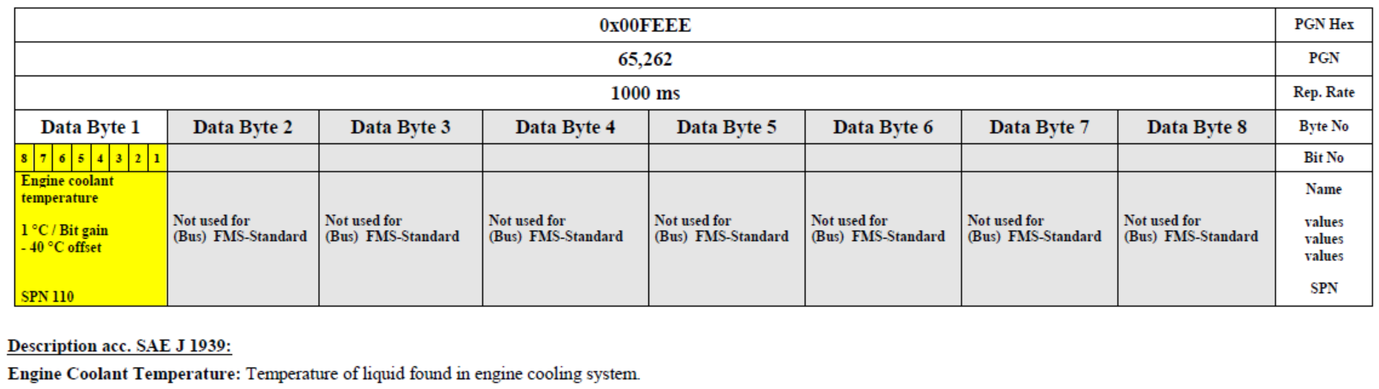
Description acc. SAE J 1939:
Engine fuel consumption accumulators
High resolution engine total fuel used: Accumulated amount of fuel used during vehicle operation. High resolution used for calculations and fleet management systems.

Figure 40. Fuel consumption frame [9].

Figure 40 illustrates the characteristics of the fuel consumption can bus frame according to the SAEJ1939 standard. From the Figure, the following can be deduced [9]:

a) It is used to indicate the accumulated amount of fuel used during vehicle operation.
b) Its corresponding PGN number is 0x00FD09. In the present work and for simulation purposes it is assumed that the PGN number is equal to the ID number. For future implementations it is necessary to provide the ID number, which depends on each vehicle.
c) Only the data in bytes 5-8 must be considered. Therefore, the sensor resolution is 4 bytes (0xFFFFFFFF)
d) It has an offset of 0 liters and for each bit increment the sensed value increases by 0.001 liters. For instance, if the sensed value is 0x0000000F, it represents a fuel consumption of 0.015 liters.
e) It has a rate of 1000 milliseconds.

- **Engine Temperature**



0x00FEEE								PGN Hex
65,262								PGN
1000 ms								Rep. Rate
Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8	Byte No
8 7 6 5 4 3 2 1								Bit No
Engine coolant temperature 1 °C / Bit gain - 40 °C offset SPN 110	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Not used for (Bus) FMS-Standard	Name values values values SPN

Description acc. SAE J 1939:

Engine Coolant Temperature: Temperature of liquid found in engine cooling system.

Figure 41. Engine temperature frame [9].

Figure 41 illustrates the characteristics of the engine temperature can bus frame according to the SAEJ1939 standard. From the Figure, the following can be deduced [9]:

a) It is used to indicate the temperature of liquid found in the engine cooling system.
b) Its corresponding PGN number is 0x00FEEE. In the present work and for simulation purposes it is assumed that the PGN number is equal to the ID number. For future implementations it is necessary to provide the ID number, which depends on each vehicle.
c) Only the data in byte 1 must be considered. Therefore, the sensor resolution is 1 byte (0xFF)
d) It has an offset of -40°C and for each bit increment the sensed value increases by 1°C. For instance, if the sensed value is 0x0000000F, it represents an engine coolant liquid temperature of -25°C.
e) It has a rate of 1000 milliseconds.

Remark: The system only processes frames with values equals to 0xFE (Maximum value). This means that the temperature is very high and an alarm must be generated.

4.3.2 Operation

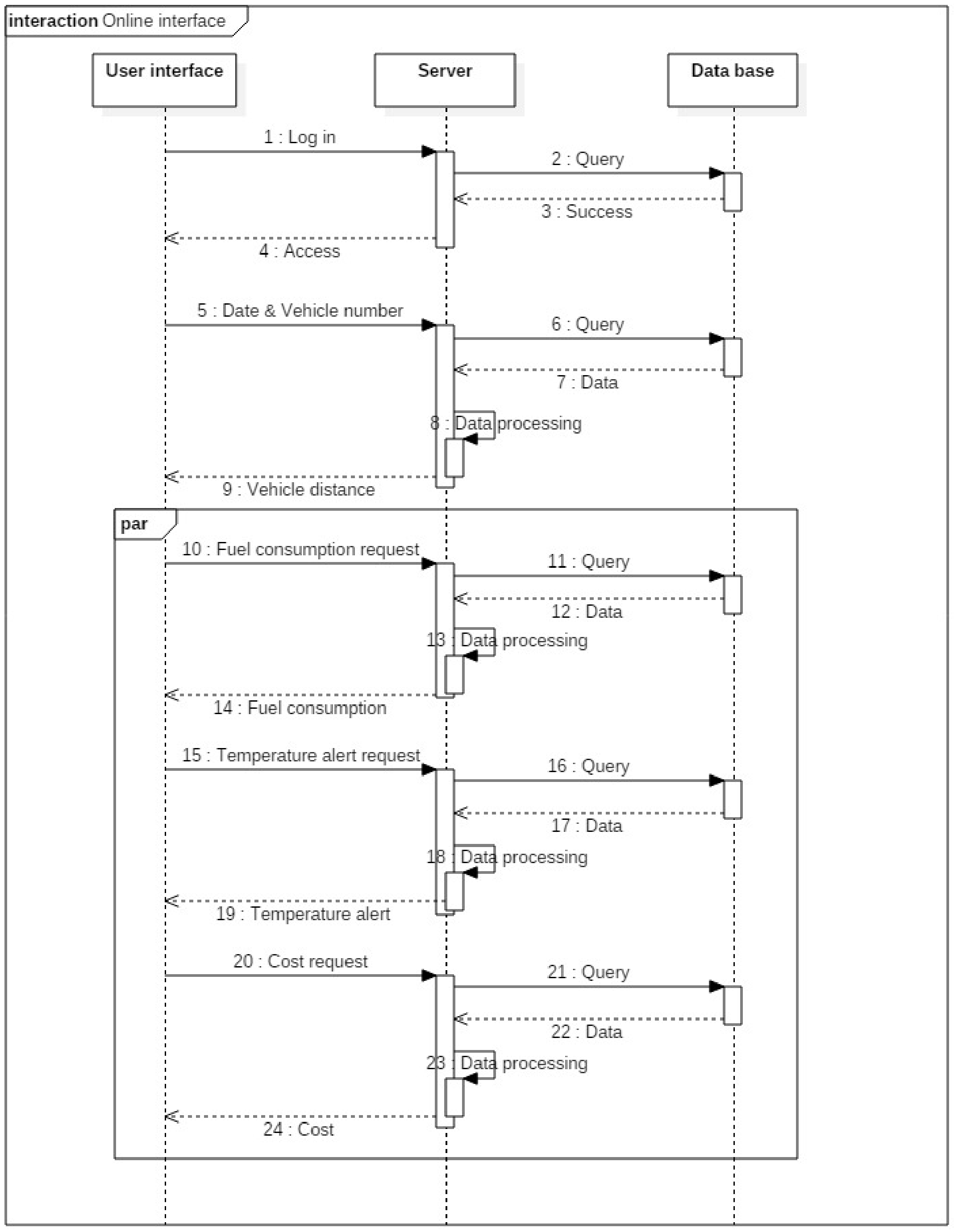
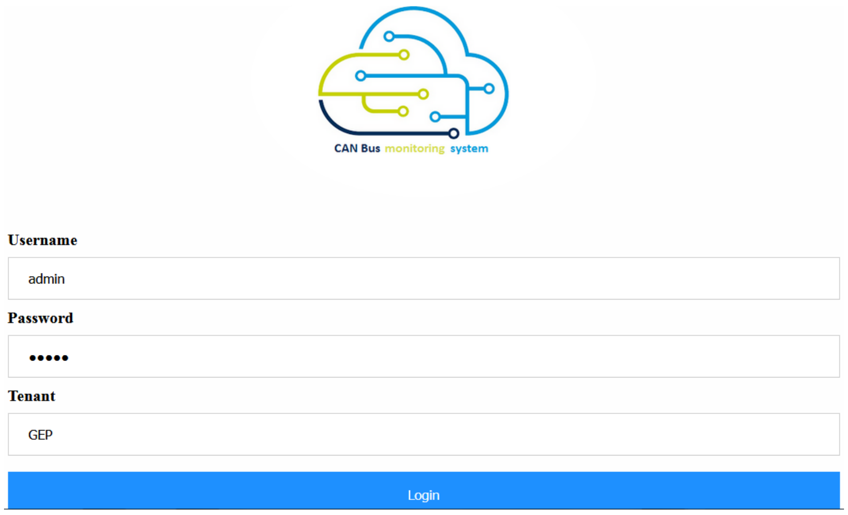


Figure 42. Sequence diagram of the online interface.

As mentioned above, the online interface has three participants. The Figure 42 illustrates the interaction among them as well as the online interface operation process which can be described in the following 24 steps:

Step 1. The process starts with the user's log in. In order to access it is necessary to present the appropriate credentials. The user interface (UI) captures the user's credentials and sends them to the server for validation. Figure 43 shows the log in screen.

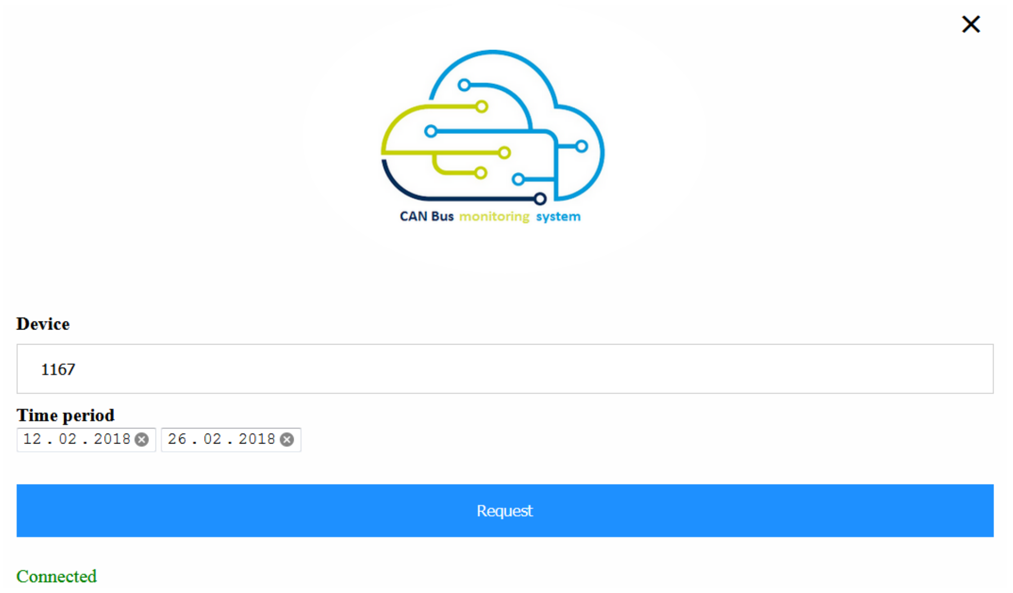


Username: admin
Password: •••••
Tenant: GEP

Figure 43. Log in screen.

Step 2-4. The server uses the user's credentials as parameters to create a query in the server database. If the query is successful (the user exists in the database) then the online interface allows access to the user.

Step 4. Once access has been authorized, the UI allows the user to enter the desired vehicle number (device) and the desired time period to consult. Figure 44 shows the screen to define the vehicle number and the time period.



Device

1167

Time period

12 . 02 . 2018 26 . 02 . 2018

Connected

Figure 44. Vehicle number & time period screen.

Step 5-6. The server uses the vehicle number and the time period as parameters to create a query in the server database. The query automatically defines the desired frame as Vehicle distance.

Step 7-8. After consulting the database, the server processes the retrieved information. Mainly, the server interprets and converts the data to the corresponding value in Kilometers according to the vehicle and time period consulted. It returns the accumulated vehicle distance in kilometers per day.

Step 9. The UI uses CanvasJS to display the processed data. Figure 45 shows the vehicle distance screen.

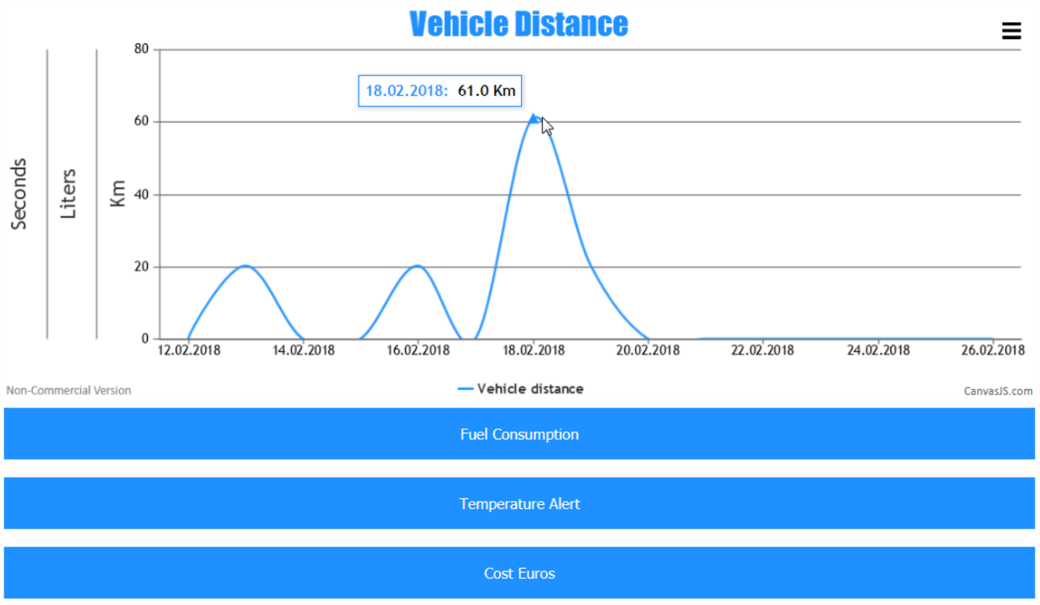


Figure 45. Vehicle distance screen.

Step 10-24. As the diagram in Figure 42 shows, steps from 10 to 24 can be executed in parallel. Thus, as shown in Figure 45, the UI allows consulting the following three parameters:

- Fuel Consumption: It returns the Vehicle fuel consumption in liters per day.
- Temperature Alert: It returns the time in seconds in which the engine temperature exceeded the maximum value per day.
- Cost Euros: It returns the cost in euros according to the vehicle fuel consumption per day.

Depending on the user's request, the server query the database according to the vehicle number, time period and desired frame. Once the server queries the requested information, it processes the data for its proper visualization. Figure 46 shows a possible visualization if all the frames have been requested.

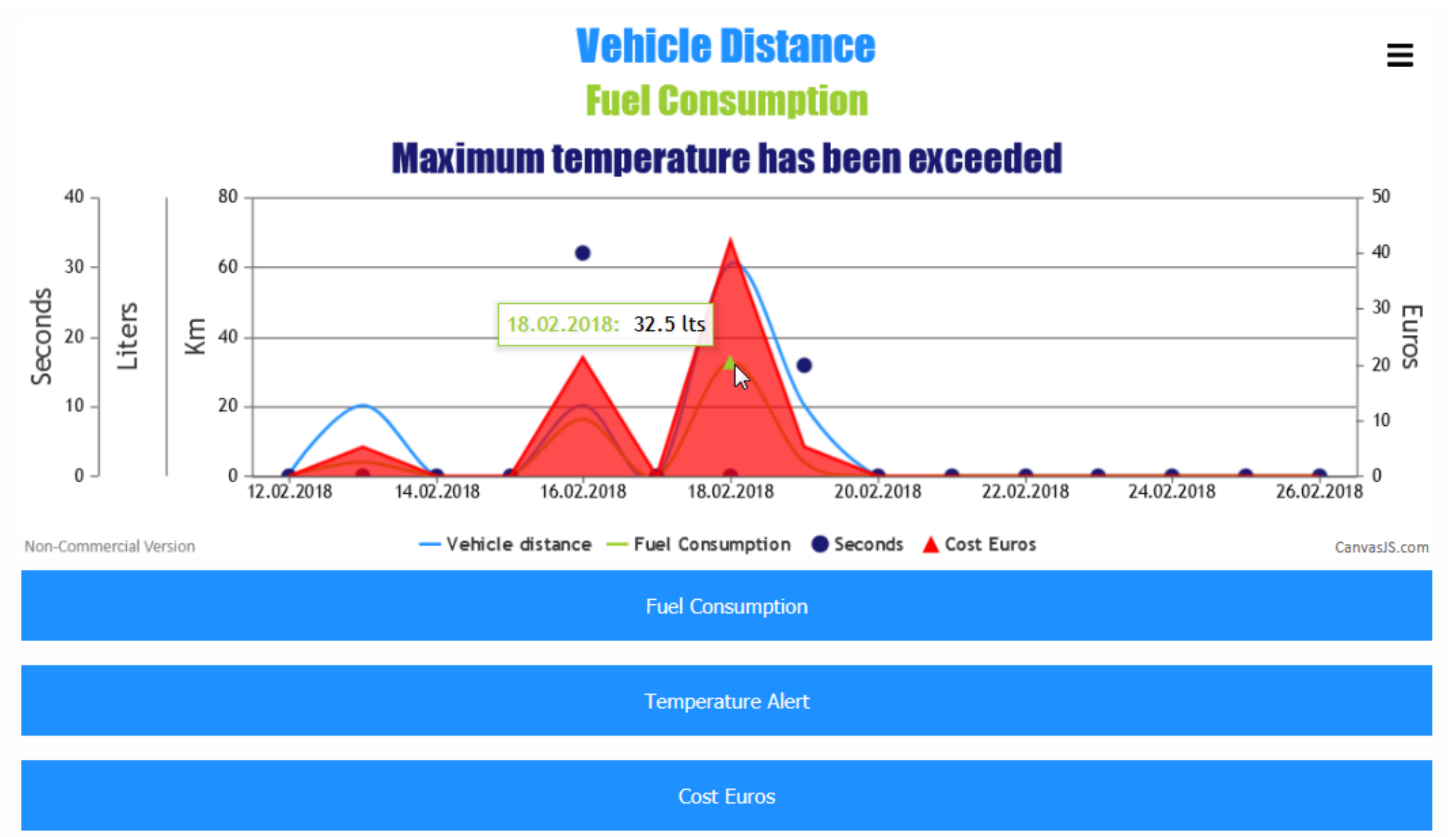


Figure 46. CAN Bus data visualization.

4.4 System architecture

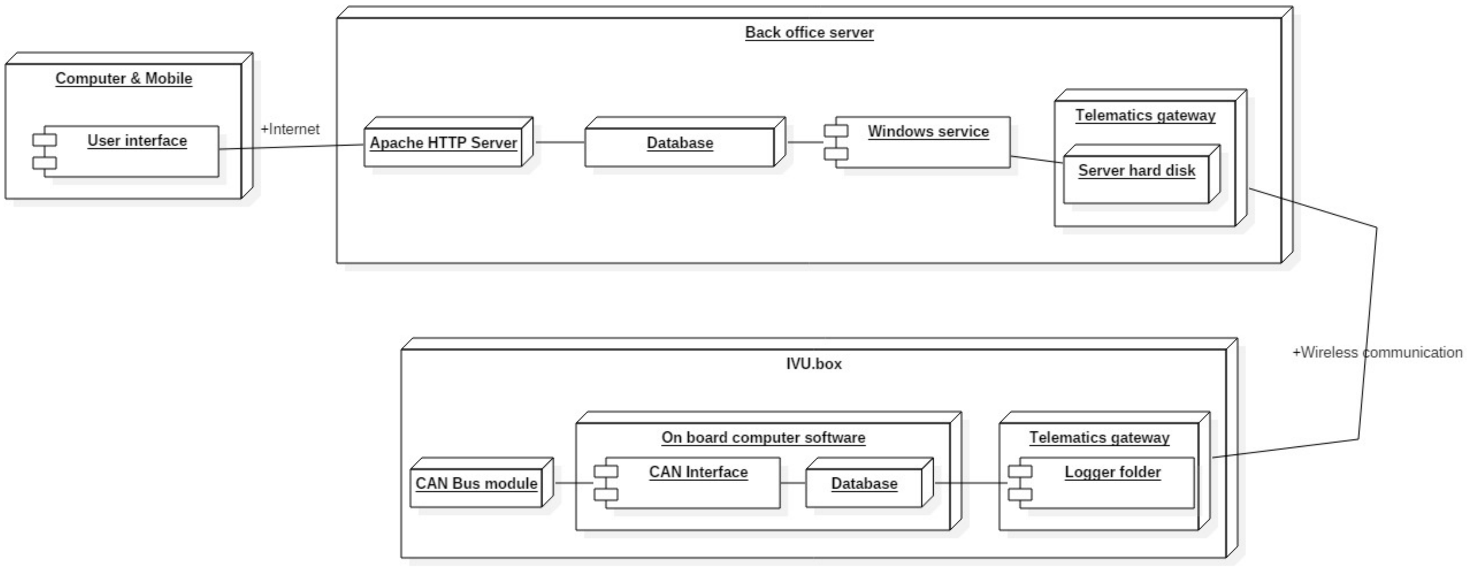


Figure 47. System Architecture.

According to the implementation, Figure 47 illustrates the system architecture. As can be seen, the IVU.box includes the CAN module described in chapter 2.2.2.1, which is controlled by the on board computer software, specifically by the CAN interface section. In addition, it can be observed that as a result of the CAN Bus reading, the program generates a database which contains the information retrieved. Moreover, the last component of IVU.box consists of a telematics gateway, which uses the logger folder to transmit information to the server.

Once the information has been received by the server, it is stored on the hard drive according to the tenant and its corresponding vehicle number. After that, it is possible to run the Windows service, which extracts the CAN Bus data from the hard disk and stores it in the server database. Finally, the user can access the information through an online interface supported by an Apache server.

5 Results & Discussions

5.1 Test cases execution

Testing is an integral part of any development process. Therefore and as mentioned in chapter 3.5, three test cases were defined with the aim of providing the system stability. In addition, the test cases execution allows the proper analysis of the expected results and the actual results. The test cases are described below along with their results.

5.1.1 Test the On board computer system. (System in Figure 21)

Summary:

Testing the on board computer software to ensure adequate reading, filtering and on board storing of the CAN Bus data.

Prerequisites:

- The IVU.box must be connected to a CAN Bus network (Figure 48). In this particular case, **PCAN-View & PCAN-USB** were used to simulate the CAN Bus Network. Figure 1 illustrates the connection between the IVU.box and the CAN Bus system. Moreover, since the software reads CAN bus frames that adhere to SAEJ1939 standard, PCAN-View must be set up with a **bit rate of 250kBit/s** as well as with an **extended frame format**.

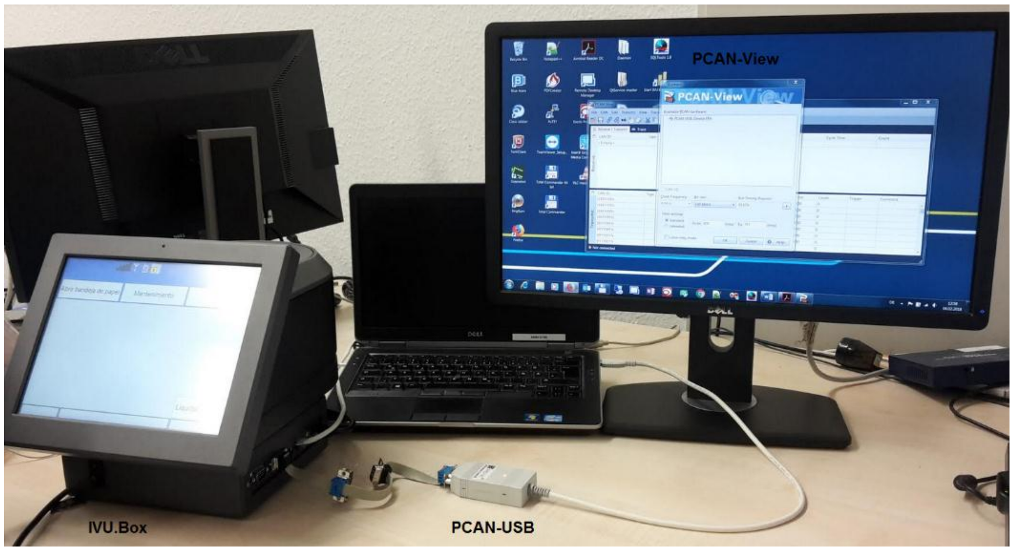


Figure 48. CAN Bus system.

- int maxReceived must be 16 (Chapter 4.1.2 Step 9).
- The application filter must be programmed to read only frames with the following identifiers:

  a) 0xFEC1
  b) 0xFD09
  c) 0xFEEE

Test procedure:

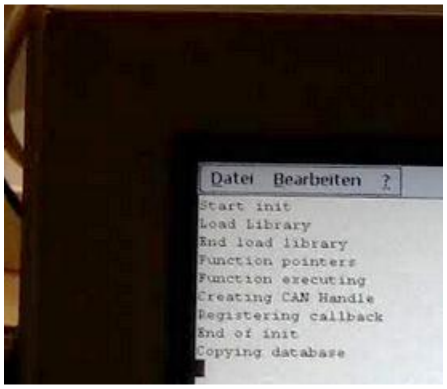
1. Start the on board computer software.
2. Create a new CAN Bus connection with PCAN-View.
3. Transmit 16 frames with an identifier of 0xFEC1 (Cycle time = 1000 milliseconds).
4. Transmit 16 frames with an identifier of 0xFD09 (Cycle time = 1000 milliseconds).
5. Transmit 16 frames with an identifier of 0xFEEE (Cycle time = 1000 milliseconds).
6. Transmit 16 frames with an identifier of 0x1F0F0F0F (Cycle time = 1000 milliseconds).
7. Transmit 16 frames with an identifier of 0x1000000F (Cycle time = 1000 milliseconds).
8. Transmit 16 frames with an identifier of 0x00000000 (Cycle time = 1000 milliseconds).
9. Stop on board computer software
10. Start on board computer software

Expected results:

1. Proper software initialization (Chapter 4.1.1)
2. Proper creation of the CAN Bus connection.
3. Reading and storage of frames with the following IDs (all other frames must be ignored): 0xFEC1, 0xFD09, 0xFEEE.
4. The CAN Bus data must be stored in the database every time g_RcvMsgArray reaches int maxReceived (Chapter 4.1.2).
5. The database must content the retrieved CAN Bus data.
6. Starting the software again triggers that the database is copied to the logger folder for its correct transmission.

Actual results:

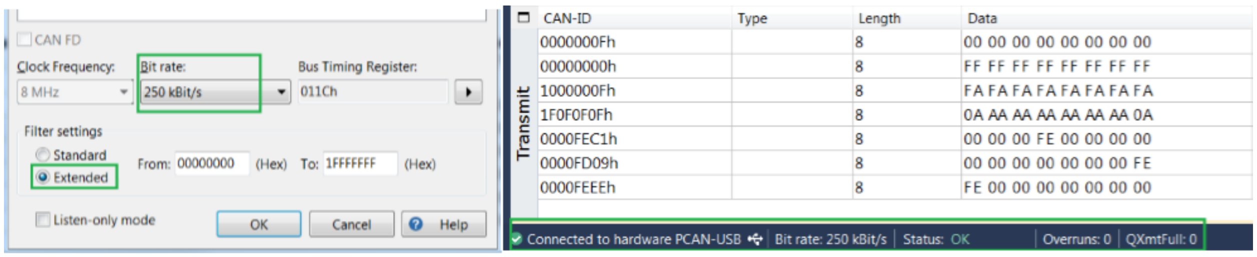
1. Software initialization achieved. Figure 49 illustrates the IVU.box messages generated by the program once the initialization has been carried out (Chapter 4.1.1).


Datei
Bearbeiten
?

```
Start init
Load Library
End load library
Function pointers
Function executing
Creating CAN Handle
Registering callback
End of init
Copying database
```

Figure 49. On board computer software initialization.

2. CAN Bus connection achieved. Figure 50 illustrates the proper set up of PCAN-View as well as the resulting connection.



CAN FD

Clock Frequency: 8 MHz

Bit rate: 250 kBit/s

Bus Timing Register: 011Ch

Filter settings: Standard [✓ Extended]

From: 00000000 (Hex) To: 1FFFFFFF (Hex)

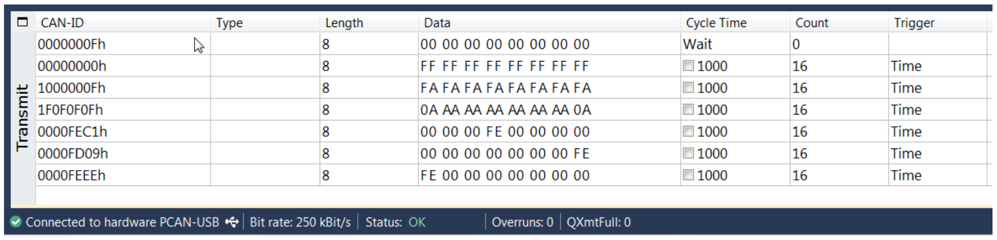
Listen-only mode

CAN-ID	Type	Length	Data
0000000Fh		8	00 00 00 00 00 00 00 00
00000000h		8	FF FF FF FF FF FF FF FF
1000000Fh		8	FA FA FA FA FA FA FA FA
1F0F0F0Fh		8	0A AA AA AA AA AA AA 0A
0000FEC1h		8	00 00 00 FE 00 00 00 00
0000FD09h		8	00 00 00 00 00 00 00 FE
0000FEEEh		8	FE 00 00 00 00 00 00 00

Connected to hardware PCAN-USB | Bit rate: 250 kBit/s | Status: OK | Overruns: 0 | QXmtFull: 0

Figure 50. CAN Bus connection.

3. The following Figure 51 illustrates the frames that were transmitted to the CAN Bus network. However, only the frames with the proper IDs (0xFEC1, 0xFD09, 0xFEEE) were read and stored. Therefore, it is concluded that the filter was working correctly.

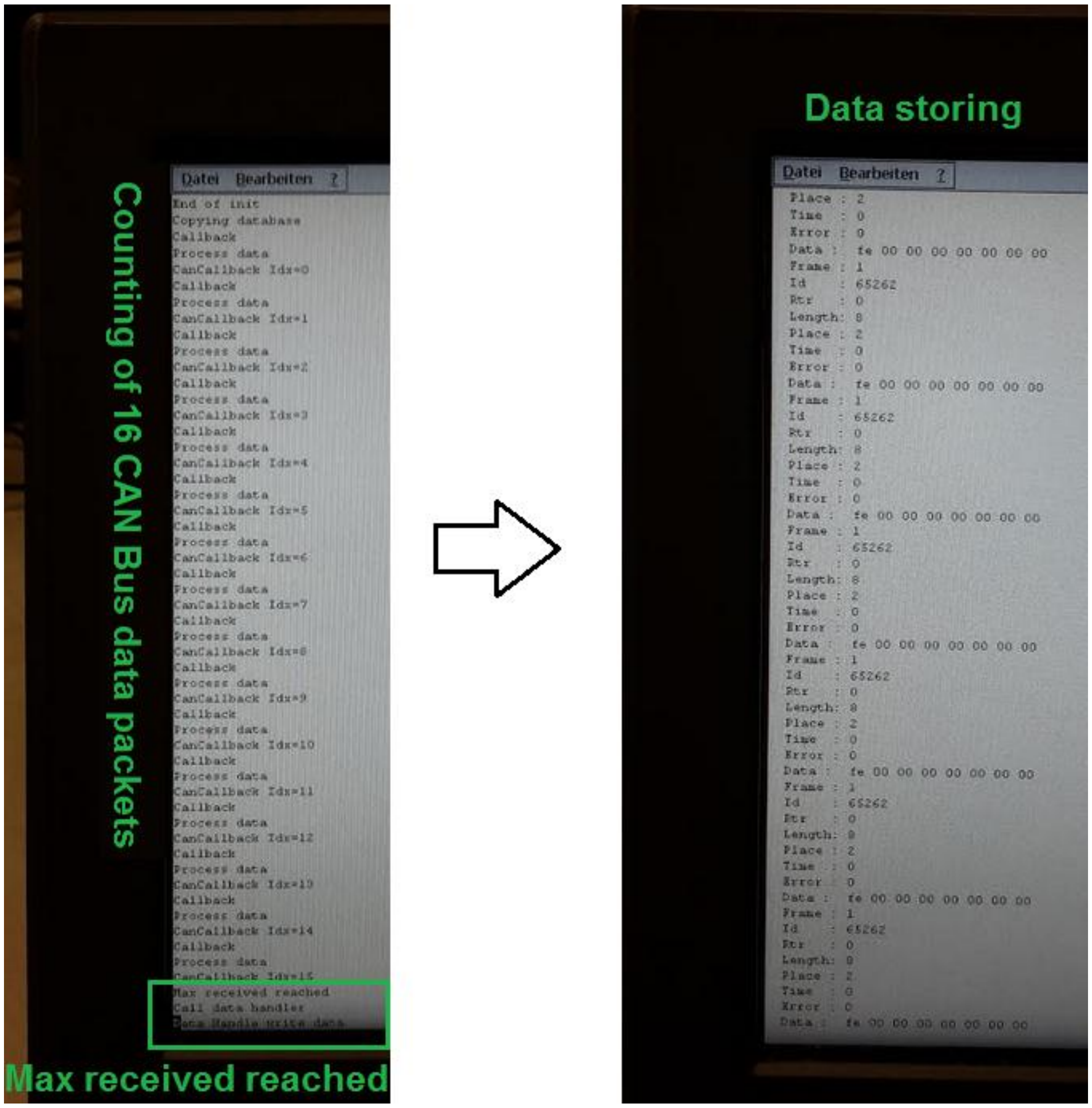


CAN-ID	Type	Length	Data	Cycle Time	Count	Trigger
0000000Fh		8	00 00 00 00 00 00 00 00	Wait	0	
00000000h		8	FF FF FF FF FF FF FF FF	1000	16	Time
1000000Fh		8	FA FA FA FA FA FA FA FA	1000	16	Time
1F0F0F0Fh		8	0A AA AA AA AA AA AA 0A	1000	16	Time
0000FEC1h		8	00 00 00 FE 00 00 00 00	1000	16	Time
0000FD09h		8	00 00 00 00 00 00 00 FE	1000	16	Time
0000FEEEh		8	FE 00 00 00 00 00 00 00	1000	16	Time

Connected to hardware PCAN-USB | Bit rate: 250 kBit/s | Status: OK | Overruns: 0 | QXmtFull: 0

Figure 51. Frames transmitted.

4. The CAN Bus data is successfully stored in the database every time g_RcvMsgArray reaches int maxReceived. Since int maxReceived is equal to 16, the CAN Bus data is stored in one single transaction through vectors of 16 elements, as shown in figure 52.

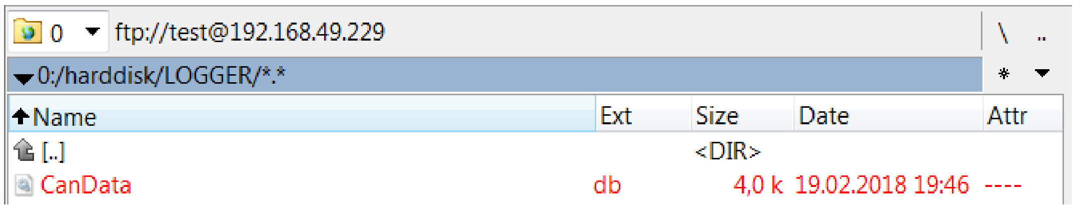


```
End of init
Copying database
Callback
Process data
CanCallback Idx=0
Callback
Process data
CanCallback Idx=1
Callback
Process data
CanCallback Idx=2
Callback
Process data
CanCallback Idx=3
Callback
Process data
CanCallback Idx=4
Callback
Process data
CanCallback Idx=5
Callback
Process data
CanCallback Idx=6
Callback
Process data
CanCallback Idx=7
Callback
Process data
CanCallback Idx=8
Callback
Process data
CanCallback Idx=9
Callback
Process data
CanCallback Idx=10
Callback
Process data
CanCallback Idx=11
Callback
Process data
CanCallback Idx=12
Callback
Process data
CanCallback Idx=13
Callback
Process data
CanCallback Idx=14
Callback
Process data
CanCallback Idx=15
Max received reached
Call data handler
Data Handle write data
```

```
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
Frame : 1
Id    : 65262
Rtr   : 0
Length: 8
Place : 2
Time  : 0
Error : 0
Data :  fe 00 00 00 00 00 00 00
```

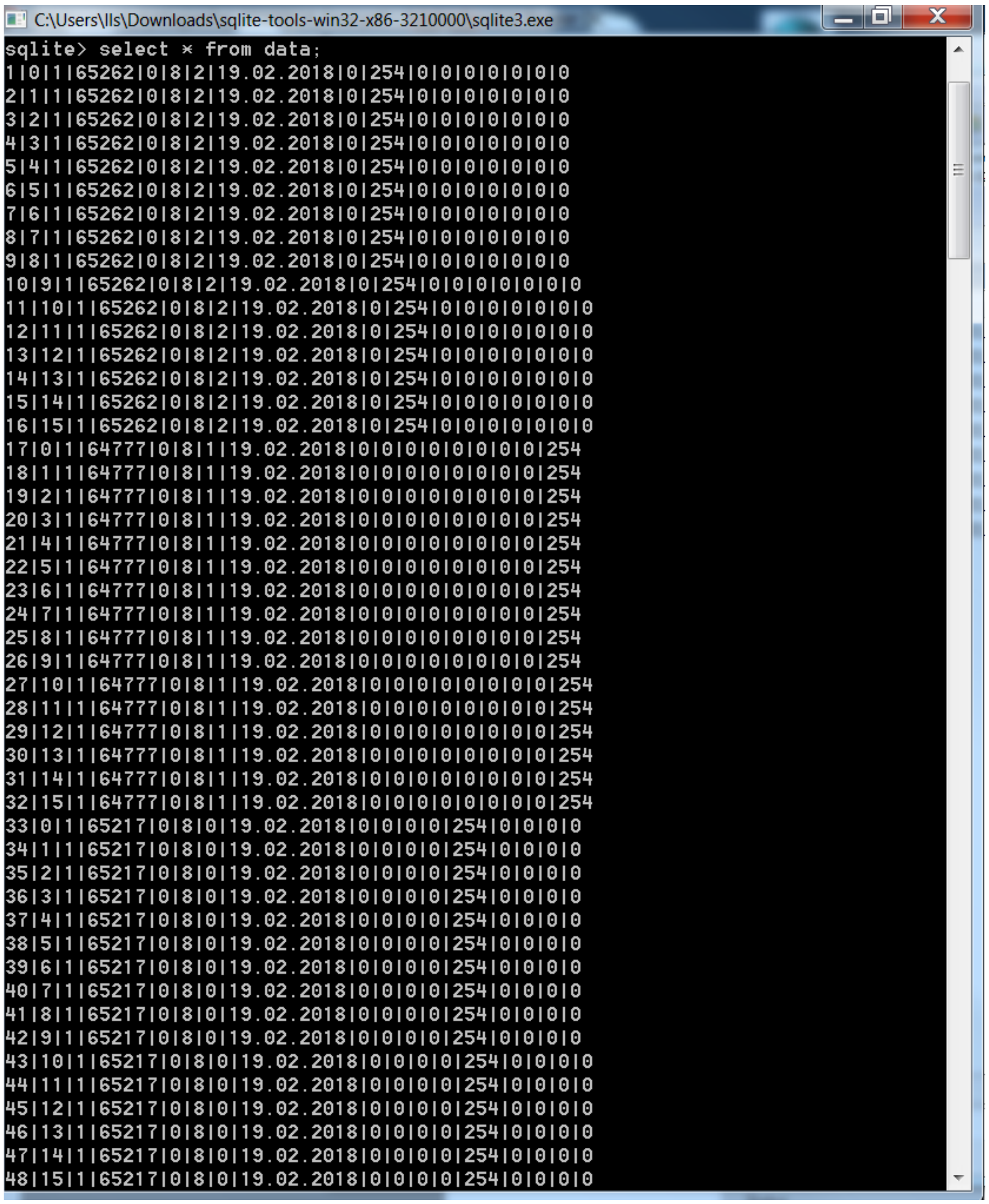
Figure 52. . g_RcvMsgArray reaches int maxReceived.

5. Once the application was restarted, the database was successfully copied to the logger folder (Figure 53). In addition, Figure 54 shows the CAN Bus data stored in the database, which successfully corresponds to the data emitted (Figure 51).



Name	Ext	Size	Date	Attr
[..]		<DIR>		
CanData	db	4,0 k	19.02.2018 19:46	----

Figure 53. Logger folder.


C:\Users\lls\Downloads\sqlite-tools-win32-x86-3210000\sqlite3.exe

```
sqlite> select * from data;
1|0|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
2|1|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
3|2|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
4|3|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
5|4|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
6|5|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
7|6|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
8|7|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
9|8|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
10|9|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
11|10|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
12|11|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
13|12|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
14|13|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
15|14|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
16|15|1|65262|0|8|2|19.02.2018|0|254|0|0|0|0|0|0|0
17|0|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
18|1|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
19|2|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
20|3|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
21|4|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
22|5|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
23|6|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
24|7|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
25|8|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
26|9|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
27|10|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
28|11|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
29|12|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
30|13|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
31|14|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
32|15|1|64777|0|8|1|19.02.2018|0|0|0|0|0|0|0|0|254
33|0|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
34|1|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
35|2|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
36|3|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
37|4|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
38|5|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
39|6|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
40|7|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
41|8|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
42|9|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
43|10|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
44|11|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
45|12|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
46|13|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
47|14|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
48|15|1|65217|0|8|0|19.02.2018|0|0|0|0|254|0|0|0|0
```

Figure 54. . Retrieved CAN Bus data.

5.1.2 Test the Windows service system. (System in Figure 21)

Summary:

Testing the Windows service system to ensure adequate extraction and storing of the CAN Bus data in the Back-office server.

Pre requisites:

- A new data logger folder must be transmitted from the IVU.box. Therefore, the logger folder generated in the previous test case will be used to carry out the present test case.
- The mentioned logger folder belongs to the GEP tenant and corresponds to vehicle number 1167.
- The service must be scheduled to operate each day at 20:00 hrs.

Test procedure:

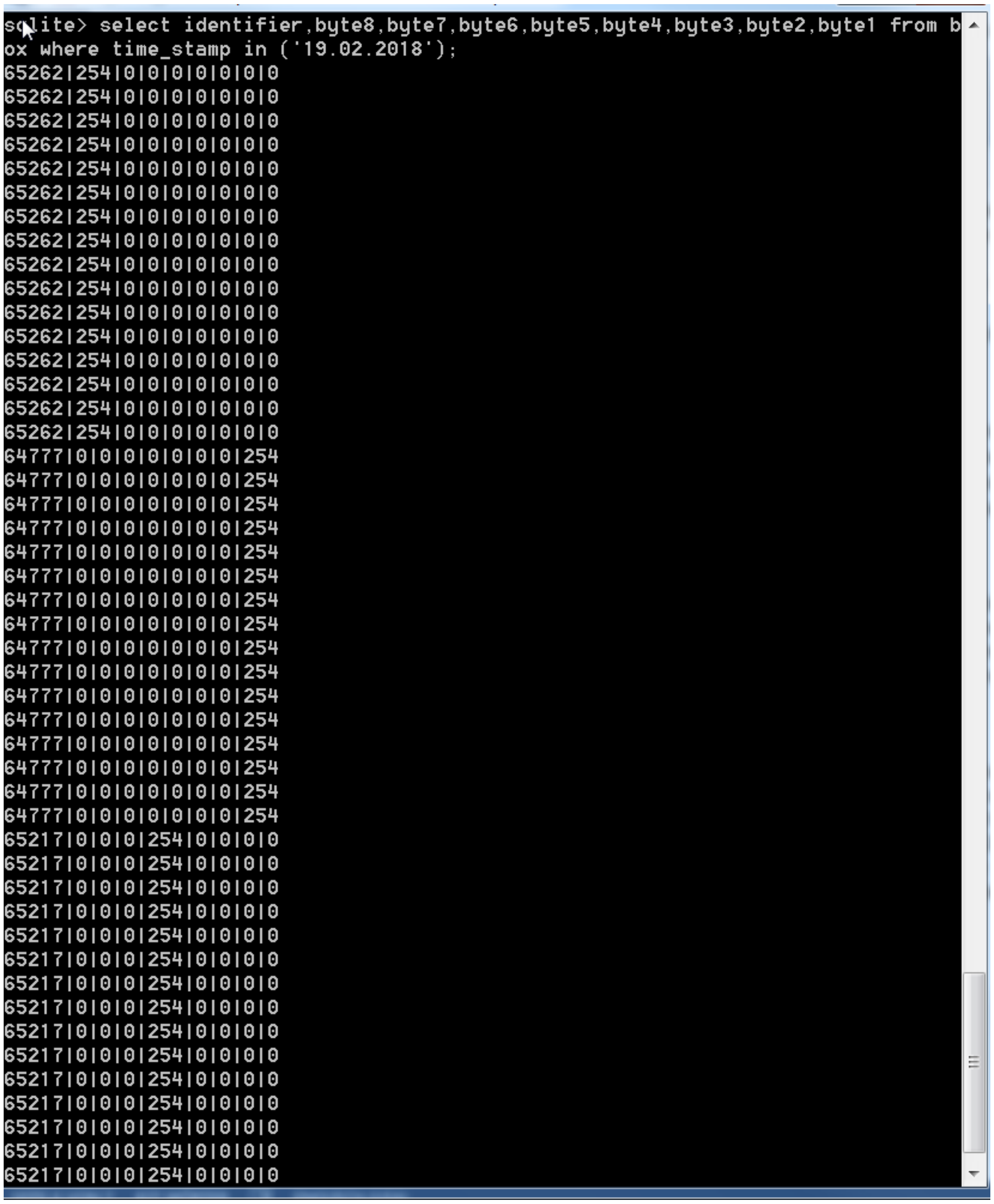
- Wait for it to be 20:00 hrs.
- Query the server database.
- Start manually the Windows service.
- Query the server database.

Expected results:

1. At 20.00 hrs. The Windows service must be automatically executed (Figure 38).
2. The server database must contain the data shown in figure 54.
3. The Windows service must be manually executed (Figure 38).
4. The server database must contain only one time the data shown in figure 54.

Actual results:

1. The Windows service was successfully executed at 20:00 hrs.
2. The server database contains the data shown in figure 54. Figure 54 illustrates that the box table content actually corresponds to data emitted by the IVU.box.
3. The Windows service was manually executed.
4. The server database contains only the data shown in figure 54. Thus, it is deduced that the logger file was read only once.



```
sqlite> select identifier,byte8,byte7,byte6,byte5,byte4,byte3,byte2,byte1 from b
ox where time_stamp in ('19.02.2018');
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
65262|254|0|0|0|0|0|0|0
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
64777|0|0|0|0|0|0|0|254
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
65217|0|0|0|254|0|0|0|0
```

Figure 55. Content of Table box.

5.1.3 Test the Online interface system. (System in Figure 23)

Summary:

Testing the Online interface system to ensure adequate access to the CAN Bus data according to different Tenants and their corresponding vehicles.

Pre requisites:

- New CAN Bus data must be stored in the server database. Therefore, The CAN Bus data which have been stored in the previous test case will be used to carry out the present test case (Figure 55).
- The server database must include at least one user with the proper credentials to access the online interface. Therefore, a general user was created with the following credentials:
  Username: admin & Password: admin
  Tenant: GEP (It is assumed from point two of the pre-requisites of the previous test case)

Test procedure:

- Log in with the wrong credentials.
- Log in with the proper credentials.
- Select date and vehicle number according to the stored data in the previous chapter (19.02.2018 & 1167).
- Consult Fuel consumption.
- Consult Temperature alert.
- Consult Cost.
- Print image

Expected results:

1. Access denied.
2. Access allowed.
3. Visualize Vehicle distance. Moreover, according to chapter 4.3.1 as well as the data stored in the previous chapter (Figure 55), it is expected to visualize a vehicle distance of **20.3 km**.
4. Visualize Fuel consumption. Moreover, according to chapter 4.3.1 as well as the data stored in the previous chapter (Figure 55), it is expected to visualize a fuel consumption of **4.1 liters**.
5. Visualize Temperature alert. Moreover, according to chapter 4.3.1 as well as the data stored in the previous chapter (Figure 55), it is expected to visualize a temperature alert of **16 seconds**.
6. Visualize Cost. Moreover, according to chapter 4.3.1 as well as the data stored in the previous chapter (Figure 55), it is expected to visualize a cost of **5.3 Euros**.
7. Printing of generated image

Actual results:

1. Access denied (Figure 56).

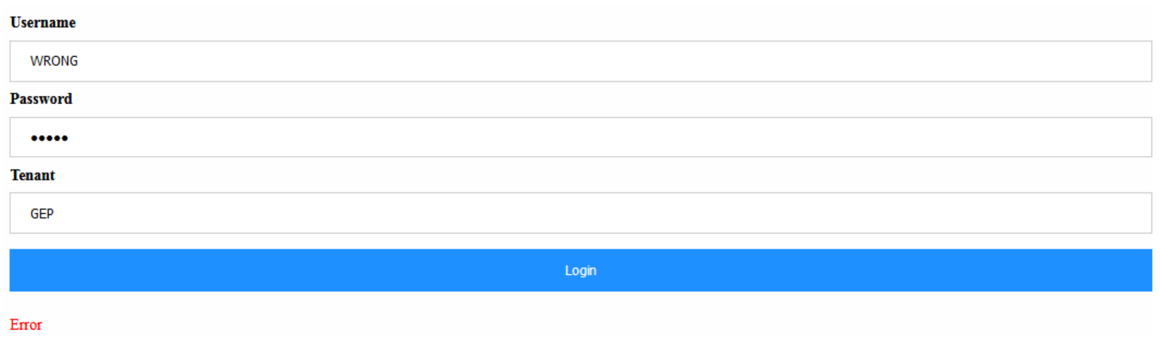
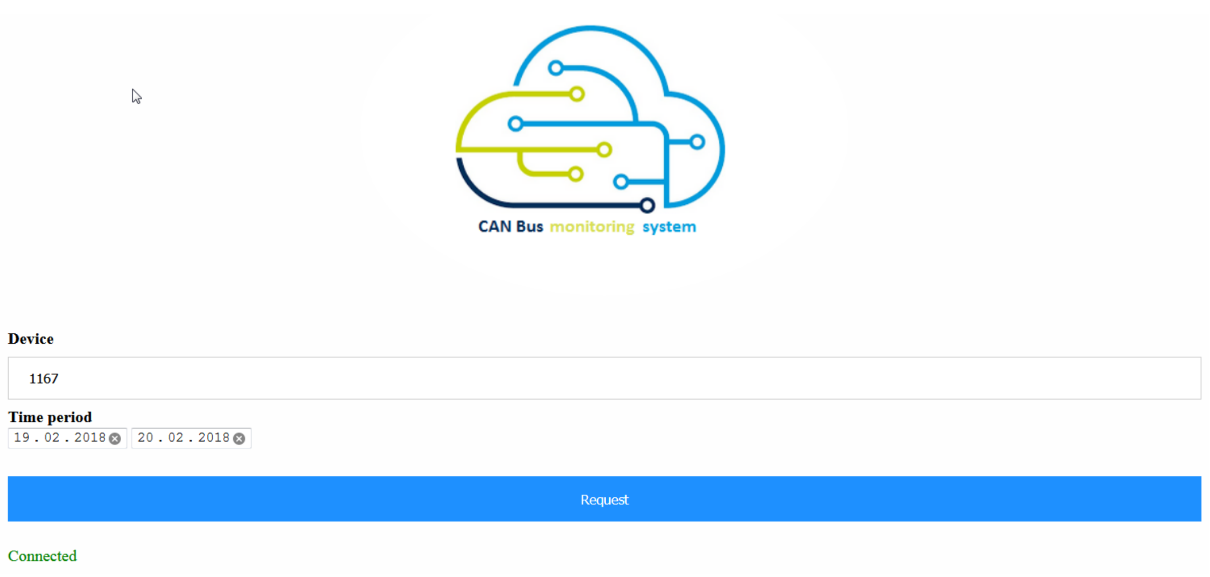

Username
WRONG
Password
•••••
Tenant
GEP
Login
Error

Figure 56 . Wrong log in.

2. Via the proper credentials the access is allowed and thus it is possible defining the device (vehicle number) as well as the time period (Figure 57).



Device
1167
Time period
19 . 02 . 2018 20 . 02 . 2018
Request
Connected

Figure 57. Device and time period selection.

3. It can be visualized a vehicle distance of **20.3 km**.

4. It can be visualized a fuel consumption of **4.1 liters**

5. It can be visualized a temperature alert of **16 seconds**

6. It can be visualized a cost of **5.3 Euros**

7. It is possible printing the generated image (Figure 58).

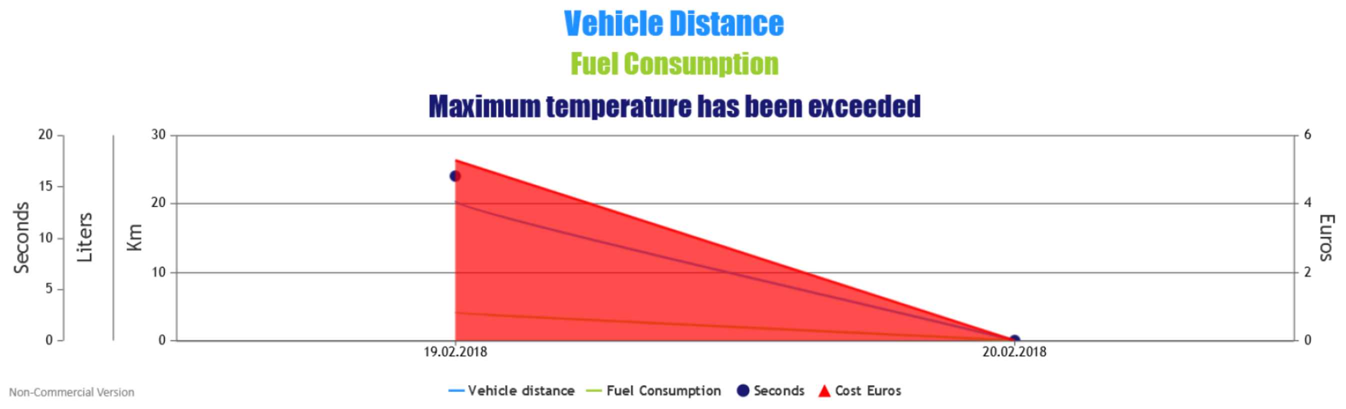


Figure 58. CAN Bus data print.

5.2 Observations

The system developed in the present Thesis allows the reading of CAN Bus frames according to the specifications established in the SAE J1939 standard. In addition, it enables the on-board storage of data in a lightweight manner. Moreover, once the data has been transmitted to the back-office server, the present monitoring system extracts the transmitted data and store it in the server database. Finally, the online interface allows access to information according to the different Tenants and their respective vehicles. Therefore, it can be concluded that the CAN Bus data monitoring system has been successfully implemented. However, it is important to mention the following aspects according to each implementation section:

- **Onboard computer software**.

  a) It only allows the reading of frames mentioned in chapter 4.3.1. Nevertheless, it is programmable (only in form of C++ code) according to the customer needs.
  b) The application was tested through a simulation of the CAN Bus network (Figure 43). The application has not been tested in a real CAN Bus environment (vehicle engine).
  c) Data volume of the stored file.

Frame	Value range	Storage type	Rate
Vehicle distance	long	4 byte	1000 milliseconds
Fuel consumption	long	4 byte	1000 milliseconds
Temperature alert	long	1 byte	1000 milliseconds

Table 7. Data volume.

The following conditions apply:

1. The few bytes at the drive start are neglected.
2. 9-byte per second.
3. The operating day is 12 hours long.
4. There will also be some (small) overhead by SQLite.

12*3600*9 ≈ 400 000 Byte. According to the IVU.box capabilities, it is acceptable.

- **Windows service.**

  a) It is designed for the current folder structure of the Back-office system. If the structure changes, the service is no longer useful.
  b) Pause functionality is currently not supported. That means, the service cannot be paused while it is processing the data.
  c) The server database is only a placeholder and a full scale data base system will be used in the future.

- **Online interface.**

  a) It was tested using only one user, one vehicle number and one tenant.
  b) It is able to process only the frames mentioned in chapter 4.3.1
  c) CanvasJS was used with a student license. If the product becomes commercial it is necessary to acquire the appropriate license.

5.3 Summary

The test cases were successfully run according to the pre requisites. Therefore, the CAN Bus monitoring system was found to be quite stable. In addition, the below table represents the summary of all the results obtained.

Section	Implementation	Drawbacks	Remarks
On board software	**Successful**. It allows the reading as well as the on board storage of the CAN Bus data.	• Is not configurable once it has been deployed.	• Limited to IVU.box storage capacity. • It was tested through a simulation of the CAN Bus network
Windows service	**Successful**. It extracts and stores the emitted data from the IVU.box.	• If the folder structure changes, the service is no longer useful. • Pause functionality is currently not supported	• The server database is only a placeholder
Online interface	**Successful.** It allows the access and visualization of the CAN Bus data	• It is able to process only three frames: vehicle distance, fuel consumption, and engine temperature.	• Developed with a CanvasJS student license

Table 8. Summary.

6 Scope for future work

According to the results obtained and the observations made, it can be deduced that the CAN BUS data monitoring system can be improved with the following proposals:

- The on board computer software should be customizable (without changing the source code) in aspects such as the filter, the value of int maxReceived, the sampling time and so on. It could be possible through a configuration file which enables saving and restoring settings in portable manner.
- The Windows service should support changes in the directory structure.
- The Windows service should include the pause functionality. It should be able to be interrupted while it is processing.
- The server database should be replaced for a full scale database such as Oracle.
- The Online interface should be able to process a broad range of CAN Bus frames.

Moreover, the implementation of the system enables the possibility to generate new development areas such as (Chapter 1.1):

- Predictive maintenance.

The aim of predictive maintenance is the continuous monitoring and diagnosis of in-service equipment in order to predict when equipment failure might occur as well as prevent unexpected equipment failures by performing maintenance. Since Predictive maintenance is highly dependent on the collection of historical data for the component being inspected, the CAN Bus interface implementation, as well as the proper storage of the data, aims to generate the basis for a predictive maintenance system. Therefore, the development of the present work combined with analytical software such as Matlab, Python or R could produce a mathematical model to predict future failures.

- Real time data monitoring.

The implementation of the on-board computer software with a CAN Bus interface coupled with a proper web service technology could allow the implementation of a real time monitoring system. Making remote access to "Live data" is a modern approach with huge benefits such as protection against accidents, maintenance handling, evaluating driver efficiency, and so on. Currently, IVU offers real time data monitoring for public transport vehicles through IVU.fleet (Figure 3). Therefore, the proprietary data telegrams technology (IVU.fleet) offered by the company combined with the present monitoring system could produce a new system which allows the remote access to the CAN Bus data in real time.

The following Figure 59 gives an overview of the project. The blue lines represent the developed system and the green lines are the possibilities and the future work produced with the implementation of the system.

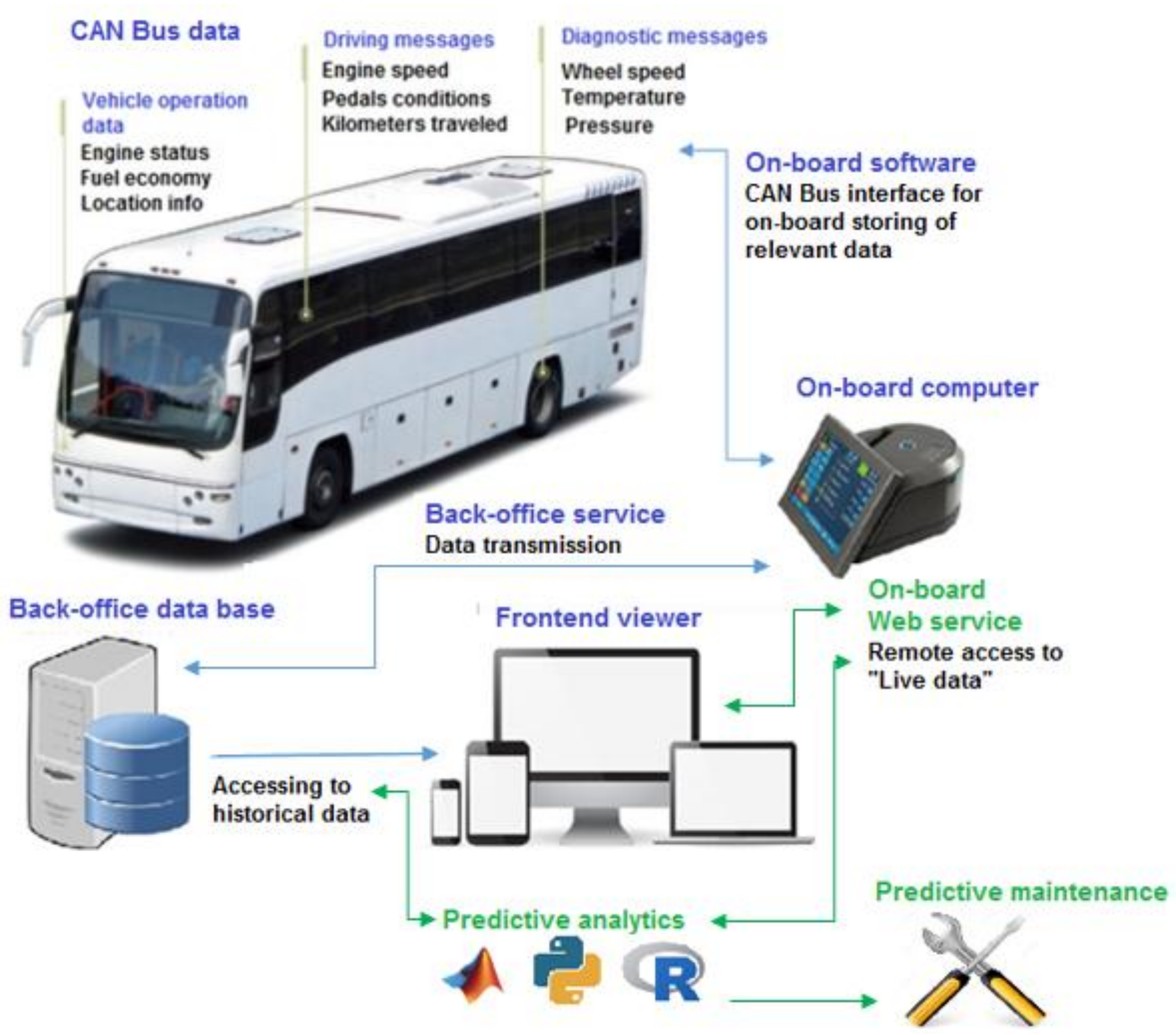


Figure 59. Future work.

As a general conclusion it can be mentioned that the present Thesis represents a product concept for monitoring the CAN Bus data of public transport vehicles. Moreover, the present work presents benefits such as a CAN Bus interface which is able to filter the frames according to the SAE J1939 standard. It also stores the retrieved data in a light way manner and it offers an appropriate interface for online access to information.

7 Bibliography

[1] I. M. Kapolka, S. Heights, S. Chang, W. Bloomfield, M. I. Us, and B. Crull, “so :,” vol. 2, no. 12, 2006.

[2] I. Larkin *et al.*, “(12) United States Patent (10) Patent No .:,” vol. 1, no. 12, 2010.

[3] “IVU.suite IVU Traffic Technologies.” [Online]. Available: https://www.ivu.com/ivusuite.html. [Accessed: 08-Feb-2018].

[4] “CAN Bus Explained - A Simple Intro (2018).” [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/ [Accessed: 08-Feb-2018].

[5] S. Corrigan, “Introduction to the Controller Area Network (CAN),” *Texas Instruments*, no. August 2002, pp. 1–17, 2016.

[6] Li Ran, Wu Junfeng, Wang Haiying, and Li Gechen, “Design method of CAN BUS network communication structure for electric vehicle,” *Int. Forum Strateg. Technol. 2010*, pp. 326–329, 2010.

[7] “CAN bus arbitration: To yell and back - Analog Wire - Blogs - TI E2E Community.” [Online]. Available: https://e2e.ti.com/blogs_/b/analogwire/archive/2015/01/30/can-bus-arbitration-to-yell-and-back. [Accessed: 08-Feb-2018].

[8] “SAE J1939 Explained - A Simple Intro (2018).” [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-j1939-explained. [Accessed: 08-Feb-2018].

[9] M. A. N. Truck and B. Ag, “Daimler Buses - EvoBus GmbH FMS-Standard description Version 03,” 2012.

[10] “IVU.box IVU Traffic Technologies.” [Online]. Available: https://www.ivu.com/ivusuite/fleet-management/ivubox.html. [Accessed: 08-Feb-2018].

[11] S. Phung, D. (David W. Jones, and T. Joubert, *Professional Windows Embedded Compact 7*. John Wiley & Sons, 2011.

[12] T. I. Incorporated, “Stellaris LM3S5B91 Microcontroller,” *History*, 2011.

[13] F. Outputs *et al.*, “ISO1050L Dominant Time-Out Function ISO1050L,” no. June 2009, 2011.

[14] Garz & Fricke Industrieautomation GmbH “GFCan32 function library” 2000-2004.

[15] PEAK-System Technik GmbH, “PCAN-USB CAN Interface for USB,” vol. 1, pp. 1–33, 2014.

[16] “Complete software development framework | Qt.” [Online]. Available: https://www.qt.io/what-is-qt/. [Accessed: 08-Feb-2018].

[17] “SQLite Home Page.” [Online]. Available: http://sqlite.org/. [Accessed: 08-Feb-2018].

[18] “A Beginner’s Guide to Back-end Development.” [Online]. Available: https://www.upwork.com/hiring/development/a-beginners-guide-to-back-end-development/. [Accessed: 08-Feb-2018].

[19] “Beautiful HTML5 JavaScript Charts | CanvasJS.” [Online]. Available: https://canvasjs.com/. [Accessed: 08-Feb-2018].

[20] “PHP: Hypertext Preprocessor.” [Online]. Available: http://www.php.net/. [Accessed: 08-Feb-2018].

[21] “Introduction to Windows Service Applications | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications. [Accessed: 08-Feb-2018].

[22] “GitHub - cedoduarte/QtService.” [Online]. Available: https://github.com/cedoduarte/QtService. [Accessed: 08-Feb-2018].