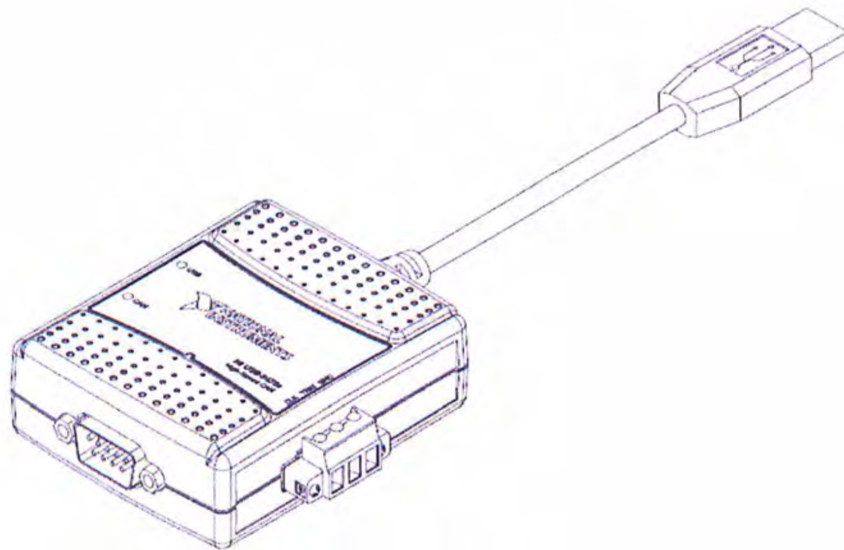
ARkivex RR08-10

CHALMERS



# CAN-drivers & Graphical User Interface in LabVIEW

*Master of Science Thesis*

KENNETH KARLSSON
JOHAN LILJEKRANTZ

Department of Radio and Space Science
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2008

# Preface

This thesis was written during the spring of 2008, as the final part of an education in electrical engineering at Chalmers University in Sweden. The level of the education was at a master's level and the scope of the thesis was 20 weeks. The work was done at Infotiv AB, a Swedish consultant company that has a "Preferred Quality Status" at Ford Motor Company. We would like to thank our supervisors at Infotiv AB, Ulf Jonsson and Peter Annlöv. Their knowledge of LabVIEW and CAN as well as their guidance was essential to the success of this thesis. We would also like to extend gratitude to the Infotiv crew at Volvo Torslanda.

# Abstract

LabVIEW® is a widely used programming tool for creating graphical user interfaces. CAN is the most common protocol for communication in cars. In this thesis the combination of the two is studied.

This thesis documents the testing of the USB-8473s hardware for CAN-communication using LabVIEW® as a programming tool. Drivers for the hardware were developed and standardized. Also a graphical user interface with capabilities of diagnostics using the D2-protocol, graphical monitoring of CAN-channels over time, logging CAN-traffic, basic CAN-communication as well as a demo-mode for demonstrating the connection between a computer and the car have been developed.

The graphical user interface was designed with a flexible structure that was event based, making it easy to implement additional features. The report also covers the structure of the program and some programming details.

# Table of Contents

*Preface* ..... *ii*

*Abstract* ..... *iii*

*Table of Contents* ..... *iv*

*1 Introduction* ..... *1*

**1.1 Background** ..... **1**

**1.2 Delimitations** ..... **1**

*2 Theoretical Background* ..... *2*

**2.1. LabVIEW** ..... **2**

**2.2 USB-8473s** ..... **4**

**2.3 CAN** ..... **6**

**2.4 D2 Protocol** ..... **7**

*3 Drivers for NI-CAN* ..... *9*

**3.1 Configure Interface** ..... **10**

**3.2 Start Communication** ..... **11**

**3.3 Send Frame** ..... **12**

**3.4 Read Frame** ..... **13**

**3.5 Change Baudrate** ..... **14**

**3.6 Stop Communication** ..... **15**

**3.7 Close Interface** ..... **16**

*4 The Graphical User Interface* ..... *17*

**4.1 The structure of the GUI** ..... **17**

**4.2 DHA** ..... **18**

4.2.1 The DDB database ..... 18

4.2.2 Front Panel ..... 20

4.2.3 Flowchart ..... 21

4.2.4 Vital Sequences ..... 22

**4.3 AnalyzeCAN** ..... **37**

4.3.1 Front Panel......37
4.3.2 Vital Sequences......39
**4.4 Demo......43**
4.4.1 Front Panel ......43
4.4.2Vital Sequences......45
**4.5 Log......49**
4.5.1 Front Panel......49
4.5.2 Vital Sequences......51
**4.6 Send Frame......55**
4.6.1 Front Panel......55
4.6.2 Vital Sequences......57
***5 Conclusions......58***
***References......59***

# 1 Introduction

## 1.1 Background

Today Infotiv is depending on the Vehicle Communication Tool, also known as the VCT, when they want to communicate on the CAN, for instance when they want to run some diagnostics. One of the disadvantages with the VCT is that it is quite clunky.

It connects through the serial port of the computer that the modern laptop often lack.

A flexible software that can be modified with ease and a hardware that is smaller and do not need a serialport is preferred. The thesis started with a test implementation to check if the desired goals where achievable.

## 1.2 Delimitations

The thesis is only compatible with Diagnostic II, also known as D2, and not compatible with GGD because there was not enough of time. D2 is an older standard and GGD is used more frequently in new cars.

# 2 Theoretical Background

## 2.1. LabVIEW

LabVIEW is short for **Lab**oratory **V**irtual **I**nstrument **E**ngineering **W**orkbench. It is an instrumentation and analysis software development application developed by National Instruments. LabVIEW uses a graphical programming language, known as the G programming language. This means that it is quite different to use LabVIEW compared to a traditional text-based programming language such as C. When using LabVIEW you create programs that rely on graphic symbols to describe programming actions. Since it uses graphical programming, it might be easier for a novice or a non-programmer to get started using LabVIEW rather than for example C.

LabVIEW is today a program widely used in the area of testing and measurements, industrial automation and data analysis. It is quite powerful and scientists at NASA's Jet Propulsion Laboratory used LabVIEW to analyze and display Mars Pathfinder Sojourner's engineering data [2].

### 2.1.1 Virtual instrumens

LabVIEW programs are called virtual instruments or VIs. The reason that they are called virtual instruments is that they have a lot of similarities with real life instruments. They always have an interactive user interface, known as the front panel (see Figure 1), where the user can input and control data and monitor the output data of the VI. This is the part of the program where you can change input data while running the program.

*Figure 1: An example of a LabVIEW front panel*

The block diagram aspect of the VI contains the logic of the VI, the source code. It is in this part that the logical behaviour is defined and modified. Different blocks are connected using wires. These wires contain values and are the equivalent to variables in regular text-based programming. These blocks that are connected in different ways are actually the executable code. To perform a simple addition of two values you connect them as shown in Figure 2.

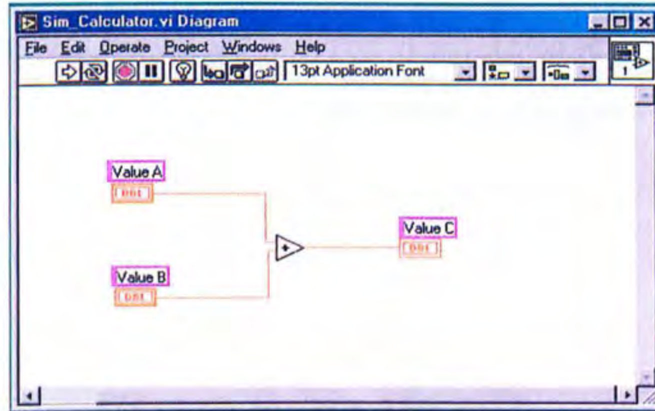



*Figure 2: An example of a LabVIEW block diagram*

The third aspect of a VI is the icon. In this part of the VI the different inputs and outputs are decided (see Figure 3). When you create the logic of your program in the block diagram you need to use either built-in-functions or your own created subVIs. A subVI is a VI that is intended to be used as a subroutine, as a part of a bigger program. These built-in-functions and subVIs are represented by icons in the block diagram and when creating a VI you can determine the which inputs and outputs the VI should have, and also their position by accessing the connector part of the icon.

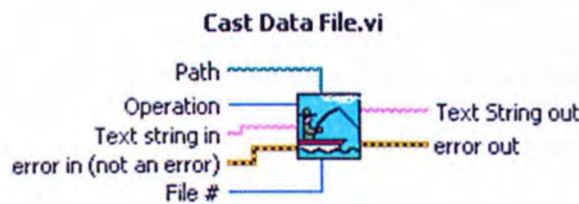
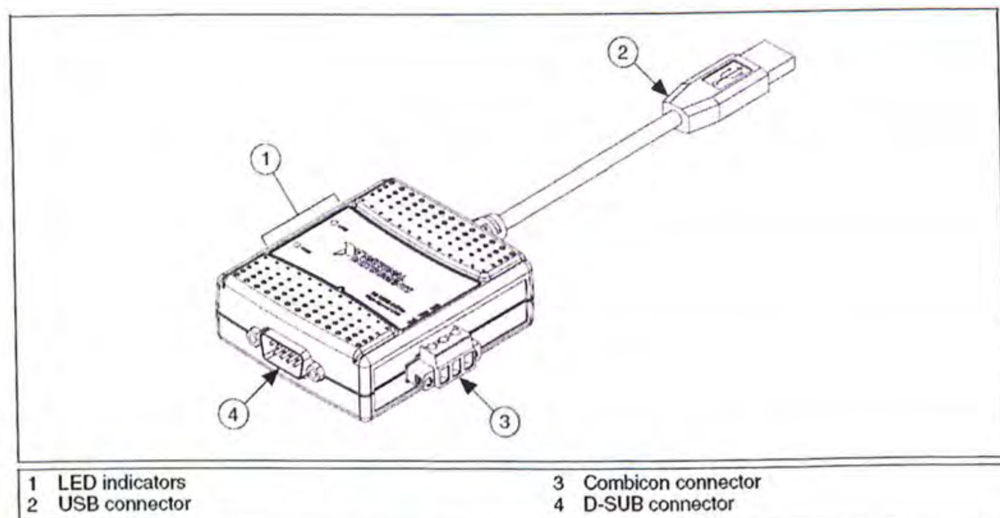



*Figure 3: An example of a LabVIEW icon*

## 2.2 USB-8473s

The USB-8473s from National Instruments is used to interface the physical bus wires of the CAN bus using the USB on your PC. It is a 1-port interface, using the Philips SJA1000 controller to implement the CAN protocol and the Philips TJA1041 for interfacing the physical bus.

The USB-8473s is designed to work at baud rates up to 1 Mbps and uses hardware timestamping, with a resolution of 1 µs. The physical layers are powered internally from the USB using a DC-DC converter. Therefor, there is no need to supply bus power. The physical layers are optically isolated up to 500 VDC (2 s max), which means that the PC and the NI-CAN hardware is protected from being damaged by high-voltage spikes on the CAN bus [1].




| | | | |
|---|---|---|---|
| 1 | LED indicators | 3 | Combicon connector |
| 2 | USB connector | 4 | D-SUB connector |

*Figure 4: Physical appearence of USB-8473s*

In Figure 4 the different physical parts of the hardware are labeled.

1. *LED indicators*: There are two LED indicators on the USB-8473s. One is labeled USB, this indicates connectivity to a USB Host signalling either a USB full speed connection (green) or a USB high speed connection (amber).
   The other one is labeled CAN. This flashes to indicate the prescence of traffic on the CAN bus.

2. *USB connector*: This connects the USB-8473s to the PC.

3. *Combicon connector*: This connector allows for synchronizing multiple NI-USB-CAN/LIN devices with each other and with a variety of DAQ, IMAQ and Motion Products. The three connections are a clock pin, a trigger pin and a ground pin. This enables the possibility of using a shared timestamp clock at rates of 20 MHz, 10 MHz or 1 MHz.

4. *D-SUB connector:* A 9-pin D-SUB connector.

The USB-8473s contains a controller for implementing the CAN protocol, as well as a transceiver for acessing the physical bus.

### 2.2.1 Philips SJA1000

The Philips SJA1000 is a stand-alone controller for the Controller Area Network. It is CAN 2.0B compatible, and supports both 11-bit and 29-bit identifiers. The circuit implements the CAN protocol and has features such as a 64-byte FIFO receive buffer [3]. Figure 5 shows the package and pinout for the circuit.

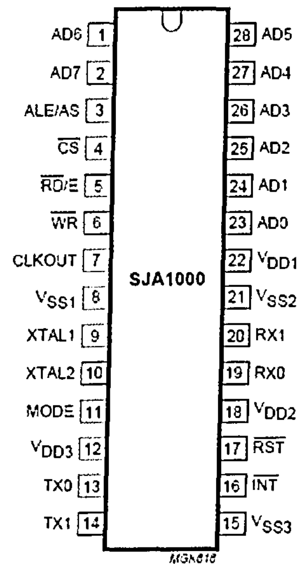



*Figure 5: Philips SJA1000 controller*

### 2.2.2 Philips TJA 1041 Transceiver

To access the physcial bus, a Philips TJA1041 is used (see Figure 6). It is a High speed CAN transceiver and is used as an interface between the protocol controller and the physical bus. When operating in normal mode it is used for normal bidirectional CAN communication. The differential analog bus signals on the pins CANH and CANL will be converted to digital data, which will be available for output on the pin RXD. The transmitter part of the circuit converts digital data on pin TXD into a differential analog signal for output on the bus pins [4].

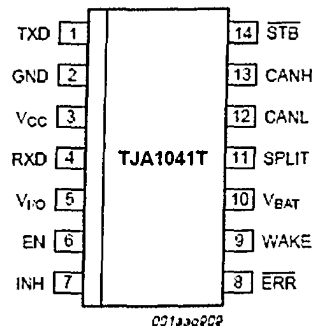



*Figure 6: Philips TJA1041 transceiver*

## 2.3 CAN

CAN is short for Controller Area Network and is a fieldbus developed by Bosch in 1988 for Mercedes. It is often used in automotive areas as the modern car is filled with electronics. The CAN protocol supports data rates up to 1Mbit/s. CAN makes the communication possible between different nodes without any host computer. Because of this there are fewer cables in the car as the nodes does not need to have one wire each and reduces the amount of cables needed. The common bus consists of two wires called CAN_H and CAN_L.

CAN also has a system for priority where for example time crucial nodes will have a higher priority on the bus. The messages or so called frames consists of several parts. The different elements are:

Start of frame, Identifier, Remote transmission request, Identifier extension bit, Reserved bit, Data length code, Data field, CRC, CRC delimiter, ACK slot, ACK delimiter and End of Frame.

All of these elements have a size of at least one bit and is shown in Figure 7.

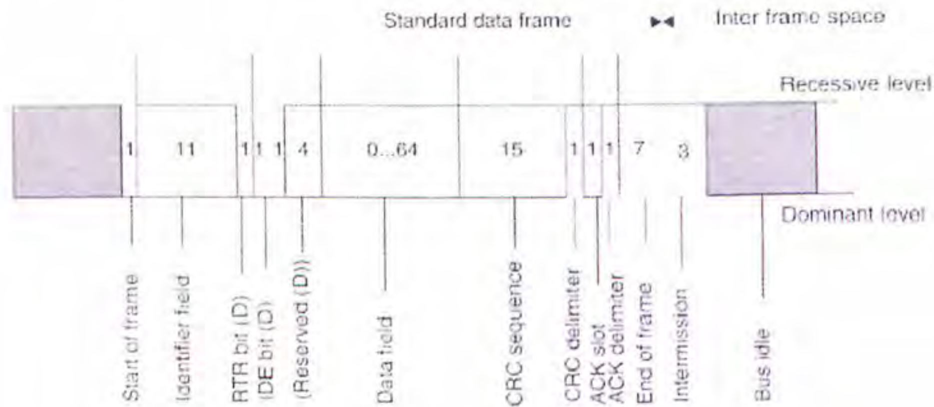



*Figure 7: CAN frame*

A message on the bus will be received by all nodes on the bus and the messages identifier will be compared with each and every node on the bus and if it is a match the node will receive it.
The identifier is also used to compare the priority. The node with lowest value on the identifier has the highest priority on the bus. For example if a device on the bus sends a message, the identifier will be compared with every nodes identifier and if device has the lowest value on the bus it will be able to send the message even if another device is trying to send a message at the same time. The device with lower priority will automatically retry to send its own message when the bus is in the idle state.

## 2.4 D2 Protocol

To properly interpret the data in the CAN messages there need to be some rules and structures. One of the methods available is the D2 protocol, or Diagnostic II. The protocol determines the purpose of the different bytes.

The first byte is called the format byte and it used to determine the type of message, size of the data, number of frames etc. There are a lot of information stored in the format byte which the program often uses to decide what to do with an incoming message.

| **Bit #7-6** | **Bit #5-3** | **Bit #2-0** |
|---|---|---|
| First/last frame | Type of message | Number of bytes or number of frames or frame seq. number |
| "11"=Single frame | "001" and "010"= diagnostic message | number of bytes |
| "10"=First frame | "001" and "010"= diagnostic message | number of frames is 111 = not specified |
| "00"=intermediate frame | "001" and "010"= diagnostic message | frame sequence number |
| "01"=last frame | "001" and "010"= diagnostic message | number of bytes |

For example, if the message is a single frame,diagnostic message and the number of data bytes is three the bit combination will be "1100 1011" which is "CB" in hexadecimal.

The second byte is called the ID byte, it is read from the database in the "ECU_ITEM" row. It is called the" COMM_ADDRESS" and is represented in decimal and it need to be converted into hexadecimal before sending the message.

"ECU_ITEM [NAME=DIM][SEND_ABLE=FALSE][NOTE=30682666 A Driver information module CAN] [FUNC_ADDRESS=209] [PHYS_ADDRESS=81] [COMM_ADDRESS=81]"

The third byte is called the Service byte and is located in the "SERVICE_ITEM" row in the database. It is called "VALUE" in the database and does not need any conversion.

"SERVICE_ITEM [NAME=Read Current Data By Offset][SEND_ABLE=FALSE][NOTE=] [FORMAT=SIGNED_BYTE] [BASE=HEX] [VALUE=A5]

The fourth byte is called the ID and is found in the DIAG_ITEM row. It is called "VALUE" in the database

DIAG_ITEM [NAME=Total Fuel level (not damped)][SEND_ABLE=FALSE][NOTE=] [FORMAT=OTHER:2] [BASE=HEX] [VALUE=01]

The fifth byte is called the mode byte and is located in the lower "DIAG_ITEM". It is only used by Read Current Data and Input Output Control.

DIAG_ITEM [NAME=Send the record once][SEND_ABLE=TRUE][NOTE=] [FORMAT=SIGNED_BYTE] [BASE=HEX] [VALUE=01]

The sixth byte is called Mask and is often "FF" in hexadecimal.

The rest of the bytes contains the data.

So with those values the message would have the following appearance:
CB 51 A5 01 01 FF XX XX

# 3 Drivers for NI-CAN

The first part of this work, consisted of testing the USB-8473s hardware, as well as the existing drivers for the USB-8473s that were provided by National Instruments. The purpose was to determine how they are used, and then enclose them in VIs that followed the standard that is used at Infotiv. By doing this we would get icons that correspond to the standard which they are using, and also to allow easy modification in regards to adding inputs or outputs, as well as logic.

All of the following VIs are used for basic CAN functions, and are based on the existing frame API provided by National Instruments. A boolean input called "Simulated" has been added to all of them. This is used to bypass the hardware specific logic, making it easy to test certain parts of the program, without using the actual hardware. There will be a short description of each VI, their input and output signals, as well as their functionality.

## 3.1 Configure Interface

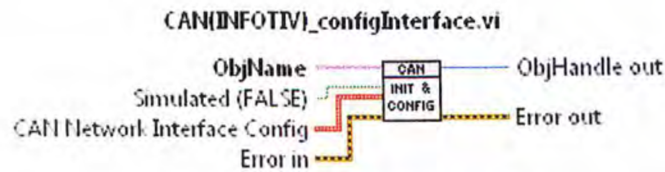



*Figure 8: Configure interface icon*

This VI is used to configure a CAN Network Interface Object. **ObjName** (see Figure 8) is a string that will determine the name for the object. It uses a syntax of "CANx", where x is a decimal from 0-63. There are two special values that can be used, x = 256 and x = 257. These are used for virtual interfaces [1]. **CAN Network Interface Config** is a cluster (see Figure 9) that contains a lot of options when configuring the CAN interface. When using the USB-8473s, only two attributes are valid. The rest of them are ignored. The two attributes available for control are "Start on open" and "Baud rate". Start on open is a boolean and describes whether the interface should start when opened, or wait for a specific start-command. The option of baud rate simply defines the speed of the CAN-bus it should be used with. For Low-speed CAN it is 125000 bps, and for High-speed CAN it is 500000 bps. When the interface has been configured, it will be opened for communication. The output **ObjHandle out** is the object handle for all subsequent CAN Vis for this object. The block diagram of the VI is shown in Figure 10.

*Figure 9: Control cluster for CAN Network Interface Config*

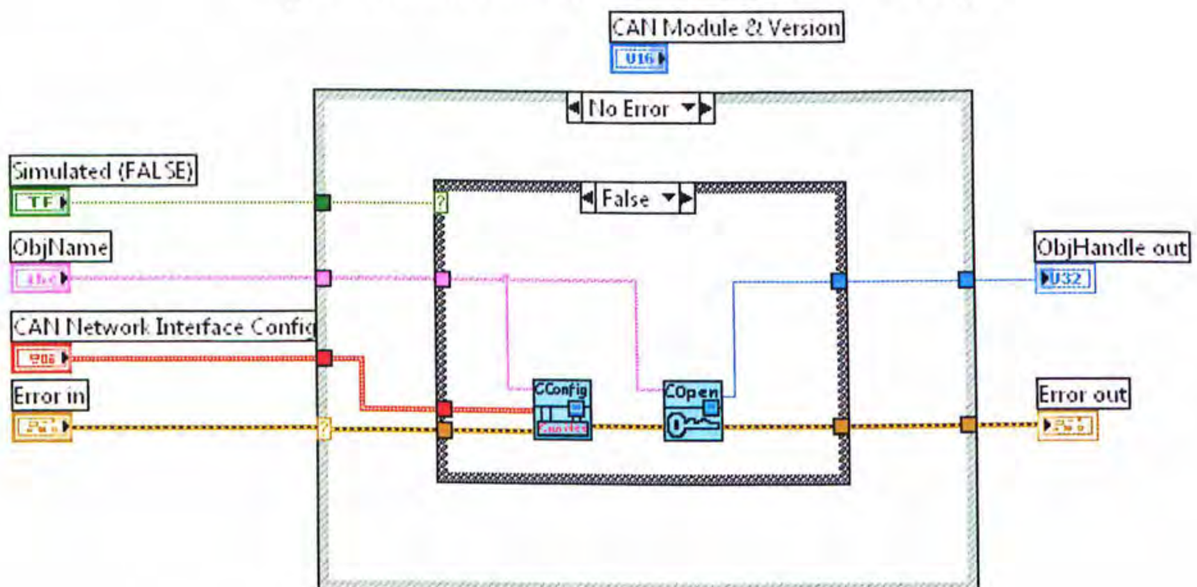



*Figure 10: Configure interface block diagram*

## 3.2 Start Communication

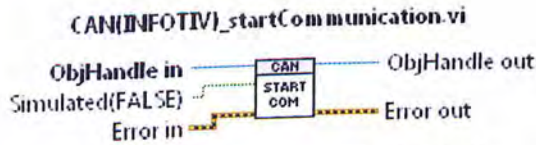



*Figure 11: Start communication icon*

This VI is used to start the communication for the interface chosen by the object handle **ObjHandle in** (see Figure 11). It will transition the network interface from stopped state (idle) to started state (running). This operation will have no effect if the interface is already started. When an interface is started it is communicating on the network. This means that the messages that are received on the CAN-bus, will be stored in a read buffer, and if not read and cleared, a buffer overflow might occur. Therefore it is important to continuously read the bus, or stopping the interface if no read will be performed.

A delay of 10 ms was added after starting the interface to make sure that there is no loss of data when trying to send a message directly after opening the interface. Figure 12 illustrates how this delay has been implemented.

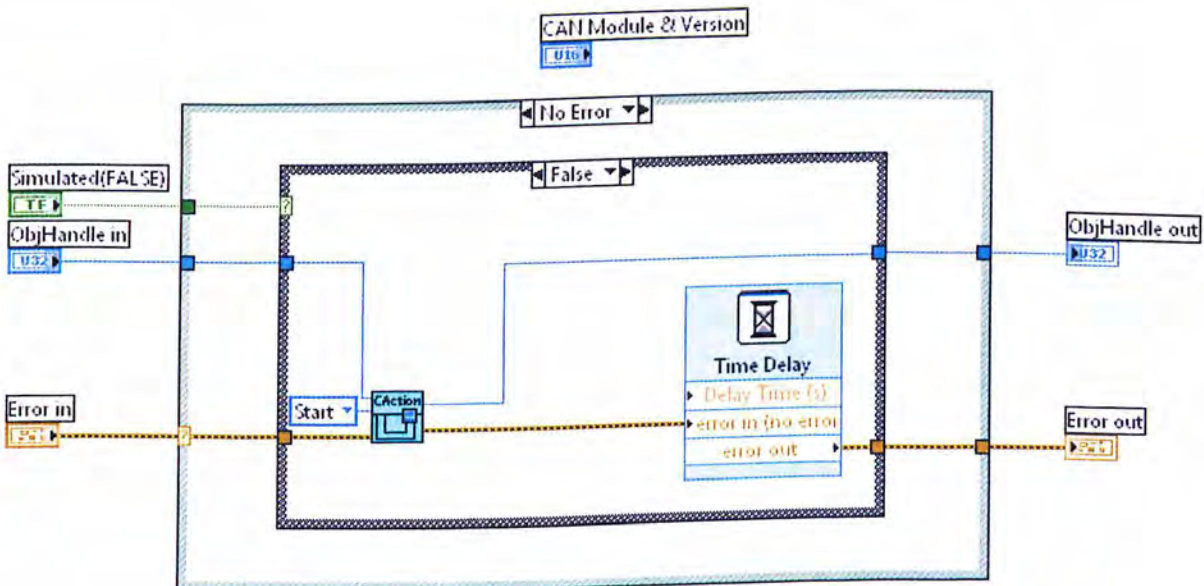



*Figure 12: Start communication block diagram*

## 3.3 Send Frame

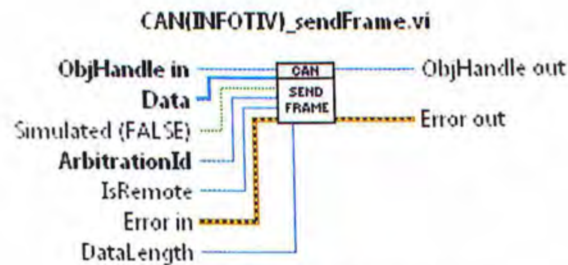



*Figure 13: Send frame icon*

The purpose of this VI is to send a single CAN frame on the CAN-bus. When this VI is called, a frame will be placed into the Network interface queue. This queue is handled in the background, and therefor this VI will not wait for the frame to be transmitted on the network. The icon is shown in Figure 13.

The **ObjHandle in** will signal which interface should be used. The **Data** will consist of maximum 8 bytes where the length is defined with **DataLength**. You can specify the arbitration ID for the frame that is to be transmitted with **ArbitrationId**. It can be either a standard 11-bit ID or an extended 29-bit ID. **IsRemote** indicates the frame type that you are sending. It will determine the interpretation of the remaining fields. For a CAN data frame this value should be 0. Figure 14 show the block diagram of this VI.

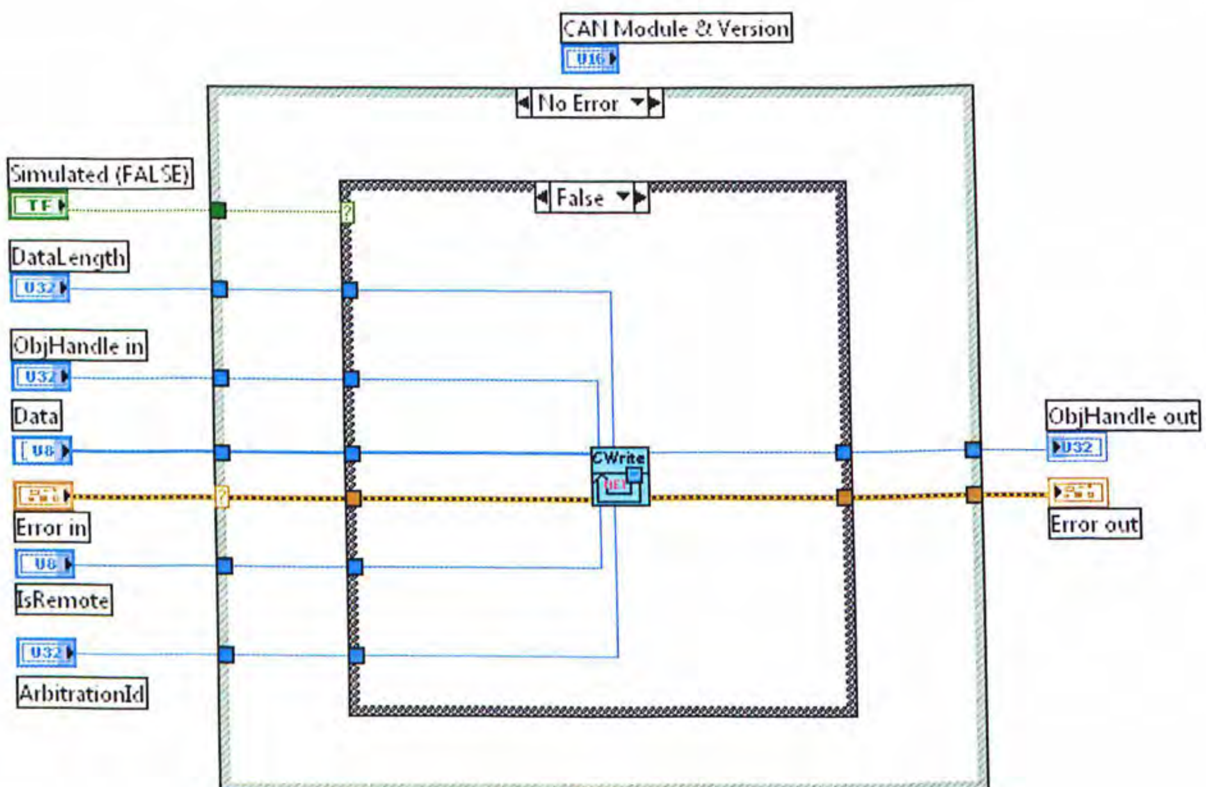



*Figure 14: Send frame block diagram*

## 3.4 Read Frame

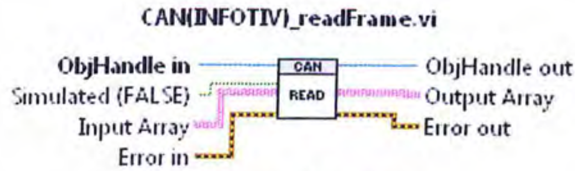



*Figure 15: Read frame icon*

The readFrame VI is used to read multiple frames from a CAN Network Interface Object. The new frames will be added to the input string array (see Figure 15).

First a check for new frames will be performed. If there are new frames waiting in the read buffer, these will be added to the input array. If there is no input array one will be created containing all the new frames. This is however not recommended since it might take a while for LabVIEW to create this array, resulting in a loss of frames. See Figure 16 for an illustration of the block diagram.

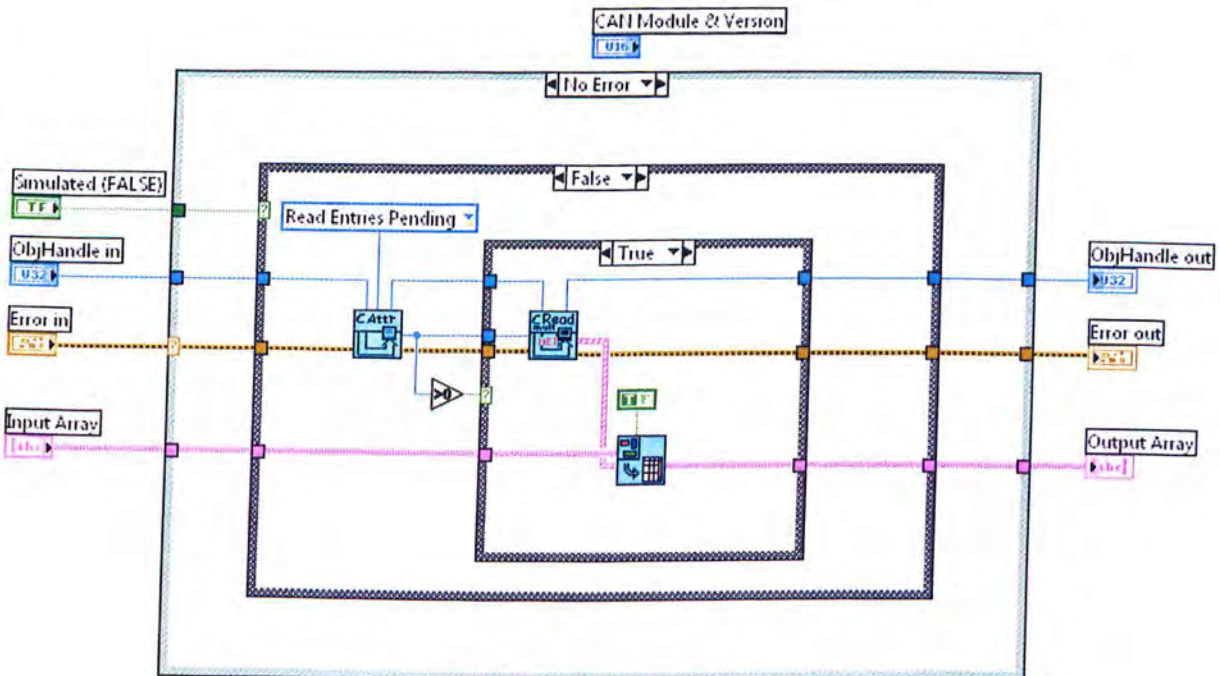



*Figure 16: Read frame block diagram*

## 3.5 Change Baudrate

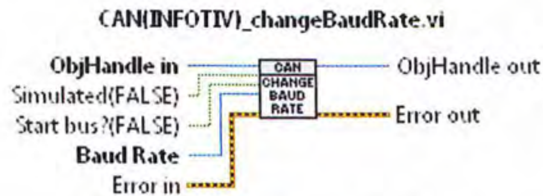



*Figure 17: Change baudrate icon*

The USB-8473s has support for different baud rates. Therefore you might want to change the baud rate of the interface. This VI will stop the interface, apply the new baud rate that is defined with the input, and optionally start the interface afterwards. If you are performing continuously readings on the bus, you might want to start the interface afterwards. However if you do not do this, it is a good idea to keep the interface in idle mode to prevent the read buffer from an overflow. The input and output signals are shown in Figure 17.

Figure 18 illustrates the logic of the VI. First of all, the interface is stopped. This action will have no effect if the interface is already in idle mode. Secondly the new baud rate defined with the input **Baud Rate** will be applied. Finally the interface will be started again, depending on the value of the boolean **Start bus?**.

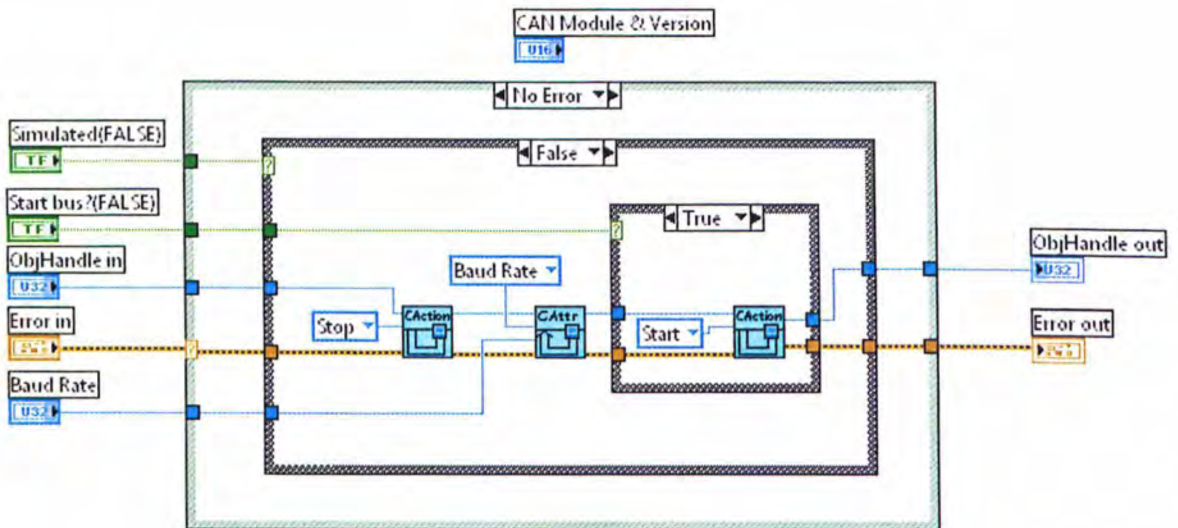



*Figure 18: Change baudrate block diagram*

## 3.6 Stop Communication

*Figure 19: Stop communication icon*

This VI will transition the network interface defined with **ObjHandle** in (see Figure 19) from started state to stopped state. This will have no effect if it is already in stopped mode. When a network interface is stopped, it will not communicate on the network and all entries in the read queue will be cleared.

It is efficient to stop the network interface if you want to avoid the read buffer from overflowing. If you are not continuously making readings, and thereby emptying the read buffer, it will eventually result in an overflow if the interface isn't stopped when no readings are performed. A delay of 10 ms was added previous to stopping the interface (see Figure 20). This was done to avoid any eventual data loss.

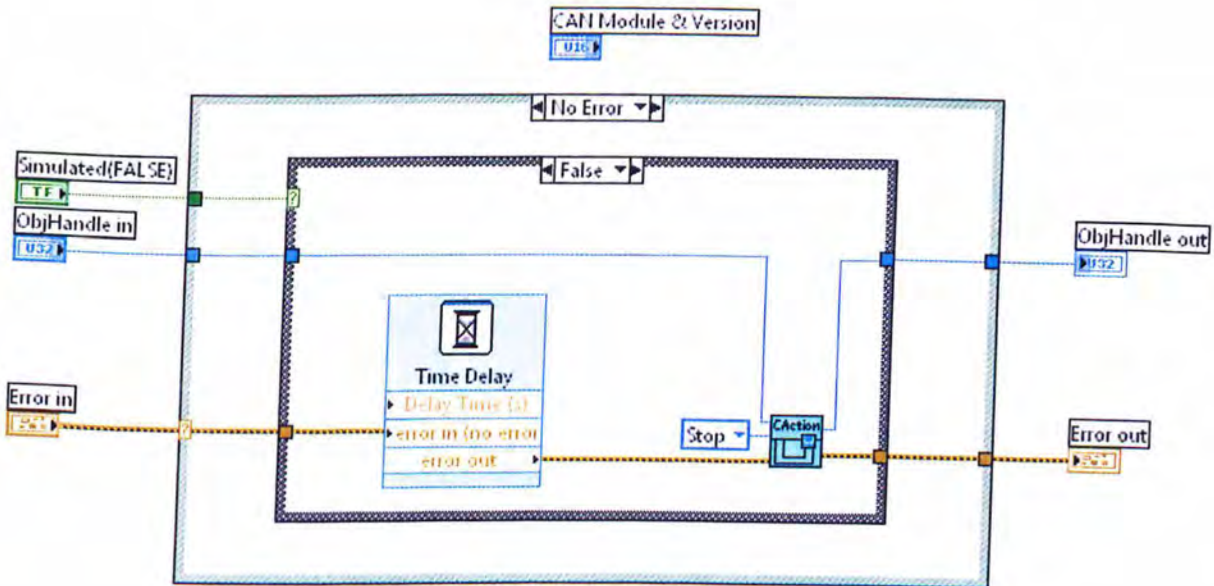



*Figure 20: Stop communication block diagram*

### 3.7 Close Interface

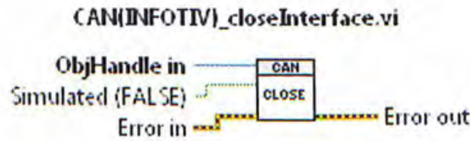



*Figure 21: Close interface icon*

This Vi is used to close the network interface specified by the object handle **ObjHandle in** (see Figure 21). It is to be used when the object no longer needs to be in use. For example when exiting an application. This action will stop all pending operations and the configuration for the object will be cleared. The object handle used will be terminated.

This VI will perform the action of closing the interface regardless if there is an error in the **Error in**. If a network interface is not closed when exiting an application, LabVIEW needs to be restarted to be able to use the hardware again. Figure 22 illustrates the logic of the VI.

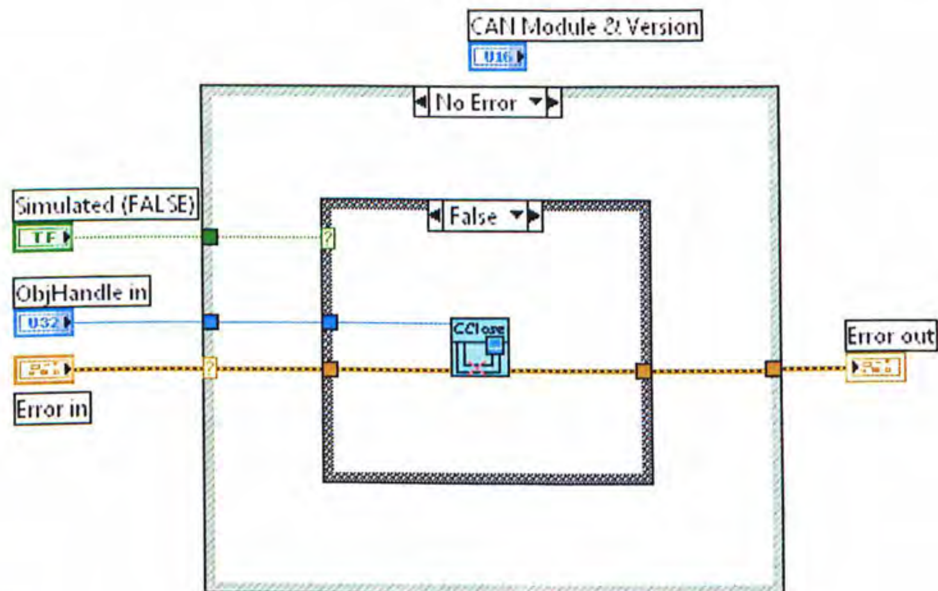



*Figure 22: Close interface block diagram*

# 4 The Graphical User Interface

The graphical user interface or GUI as it is also known as, is an intuitive way of controlling a system. This GUI is used for communication on the CAN bus in different ways. The goal was to create a broad variety of tools for the user. This includes different way to monitor the traffic on the bus, sending messages in a variety of ways, log the traffic on the bus, the possibility to read log files and a part that consists of a quick demonstration on the strengths with CAN.

The GUI is divided into several tabs, which contain different functionality. These different tabs will all be described, as well as the structure of the logic behind the GUI.

## 4.1 The structure of the GUI

There are different ways on constructing a program with the above criterias, and the method that was chosen where later proven to be a intuitive and graspable. It is based on structure of events, cases and a queue. An event can be triggered on a wide range of actions in the front panel. Most of the time an event triggers on button that changes value. When the value changes the program goes in to a event case where the program usually puts an element into the queue. The element in the queue consists of a string that will be sent to the case-structure. The case structure have a list of strings that it will compare the string from the queue with. If they match, the case will perform the assigned task/tasks. The structure is shown below in Figure 23.

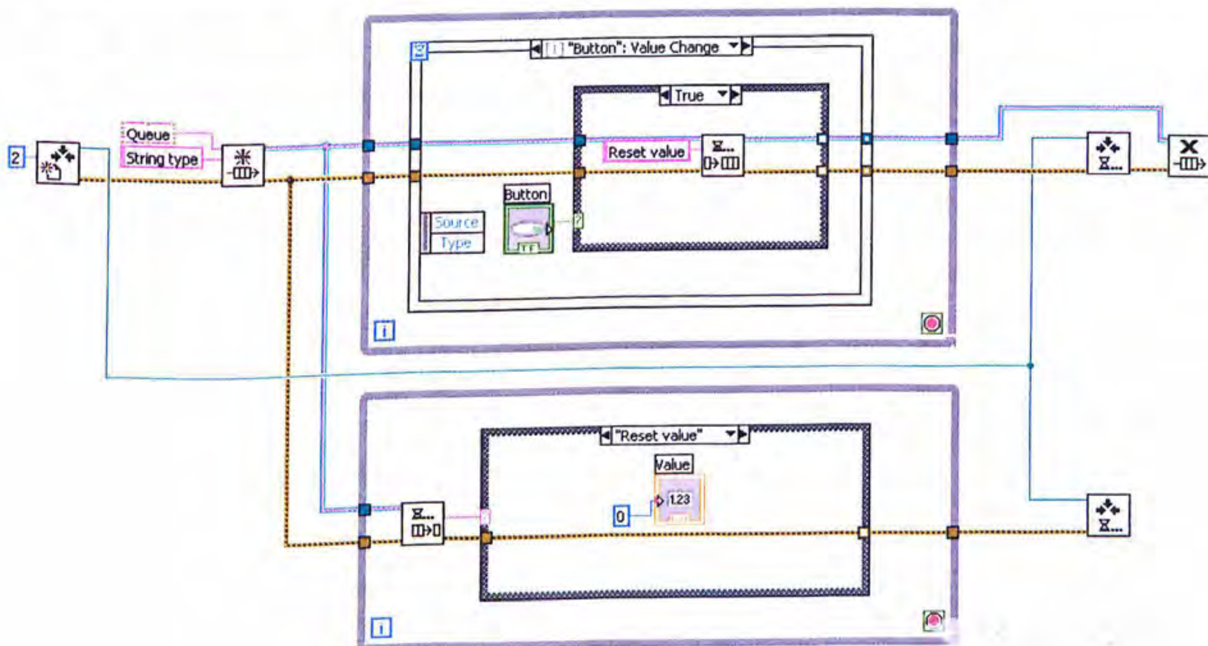



*Figure 23: GUI structure*

## 4.2 DHA

This tab is a remake of the program DHA, Diagnostic Host Application, and have a similar functionality. DHA is a program that is used to read the CAN bus. With DHA it is possible to interpret the data as it uses a database to correctly compute the correct units and values, the database is called the DDB database and will be used by the LabVIEW program. With the aid of different combo boxes the user can put together a message that will give a value to a node or request a response. Constructing the message and interpreting the response requires a database file to be accessed.

### *4.2.1 The DDB database*

The information about the messages and interpretation of the responses is stored in a DDB database. This section will try to explain the structure of the DDB database.
The database is indexed with blank spaces in a tree like manner. At the top there is the "SYSTEM_ITEM", and indexed under the "SYSTEM_ITEM" lies the different "ECU_ITEM":s. The "ECU_ITEM":s in their turn contains a "SERVICE_ITEM":s and then indexed under each "SERVICE_ITEM":s lies "DIAG_ITEM":s. The structure is illustrated below in Figure 24:

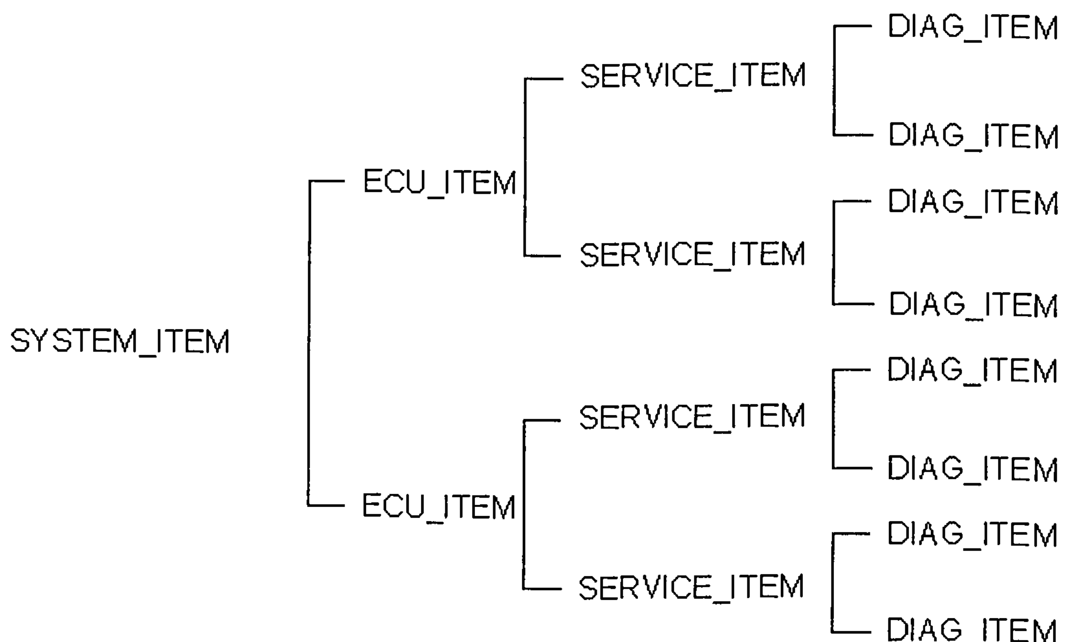



*Figure 24: Database structure*

The program constructs an array of the database that is indexed with columns, where two blank spaces is represented with a column. This way it is easy to find the same categories as they are indexed at the same column.
The "ECU_ITEM":s are the different nodes in the database. It can represent a hardware like the instrument panel for instance. Every node or "ECU_ITEM" has different "SERVICE_ITEM":s that has different tasks. Not all of the "SERVICE_ITEM":s are used, here are the ones that the program uses:

| Names in the database | Names in the front panel |
| --- | --- |
| Read Current Data By Offset | Read Record data |
| Write Data Block By Offset | Input Output Control |
| Input Output Control By Identifier | Input Output Control |
| Control Routine By Offset | Routine Offset |
| Read Data Block By Offset | Read Block Offset |
| Read DTC | Read DTC |
| Clear DTC | Clear DTC |

For instance the "SERVICE_ITEM" Read Current Data By Offset sends a requests message to a node, the "SERVICE_ITEM" Write Data Block By Offset sends a message that will change a value at the node. The "DIAG_ITEM":s are very hardware specific which means that they vary a lot depending of the "ECU_ITEM":s. They often represents the different hardware of the ECU. It can for example be an indicator for the vehicle speed or for the fuel level. The information about how to interpret a response and the information about the vital parts of message to be sent is stored in the "DIAG_ITEM":s. The information varies depending in which "SYSTEM_ITEM" the current "DIAG_ITEM" is located.
Here are some examples of "DIAG_ITEM":s from the DIM ECU:

Total fuel level
Engine coolant temperature value
Engine speed

### 4.2.2 Front Panel

In the DHA tab,illustrated in Figure 25, the user will need to start by loading a database file. Then it will be possible to select an ECU from the ECU-list. The list with the different “DIAG_ITEM”:s below the ECU-list will change when changing the radio buttons. When a “DIAG_ITEM” has been selected a message will be composed and visible in the “Message field”. The sent message will be shown in the “Response” window. If the sent message was requesting a response, the response will also show up in the “Response” window.
It is possible to compose a message manually in the “Custom message” field, the response will however not be interpreted.
The contents of the “Response” window can easily be saved to a text file with the “Save responses To File” button. The “Clear Responses” clears the contents of the “Response” window.

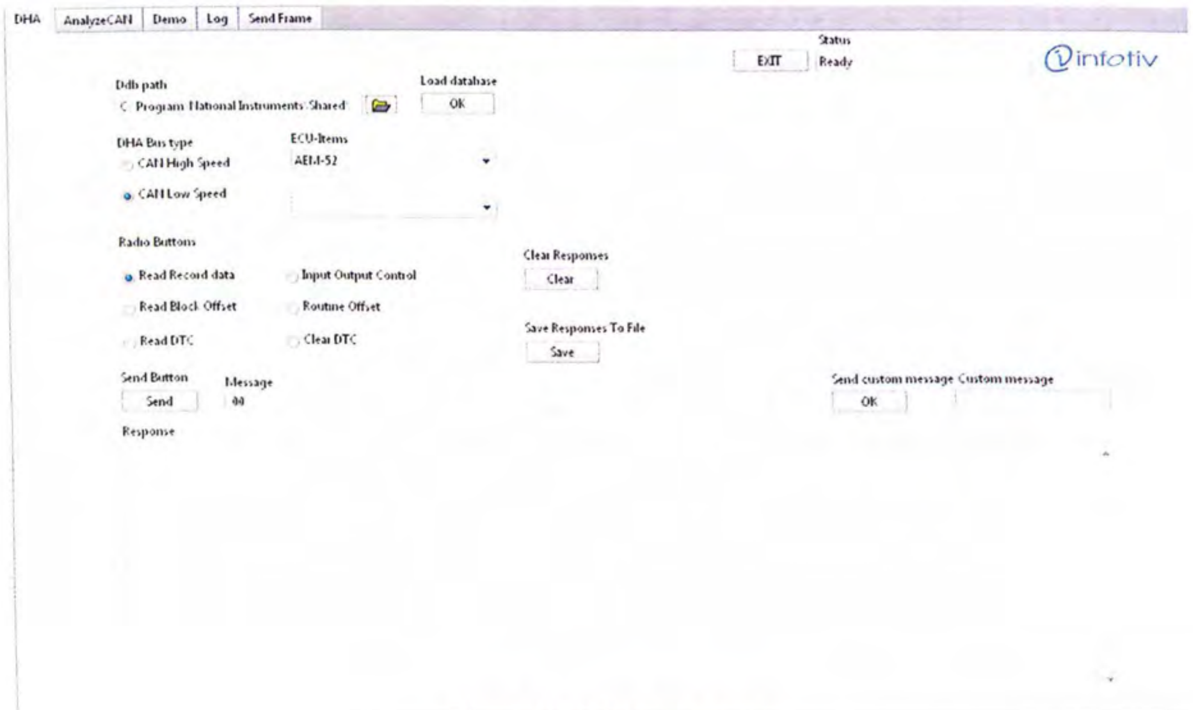



*Figure 25: DHA front panel*

#### 4.2.3 Flowchart

The DHA-tab needs a database to function. The content of the database will be stored in an array that will be indexed with an empty element for every two blank spaces in the database. This will make it easy to find the items as they are in different columns. The array will also function as a storage of additional information at column zero. At column zero the limits, message, current radio button and other information are stored. After the array is created, the row numbers that the current SERVICE_ITEM begins with is stored in the ECU-array, and so is the row number that the SERVICE_ITEM ends with, they are called the Service begin and Service end. The limits are important because between the limits lies the actual "DIAG_ITEM":s. The next step after the limits of the "SERVICE_ITEM" are found, the program need the limits of each of the "DIAG_ITEM":s that lies in the current "SERVICE_ITEM". The information in the "DIAG_ITEM":s are vital for interpreting responses and composing messages. When all the limits are found, the list in the front panel can be updated. When the user has selected a "DIAG_ITEM" in the front panel a message will be composed. Depending on what kind of message that was sent, the response can be interpreted. The flowchart is illustrated in Figure 26.

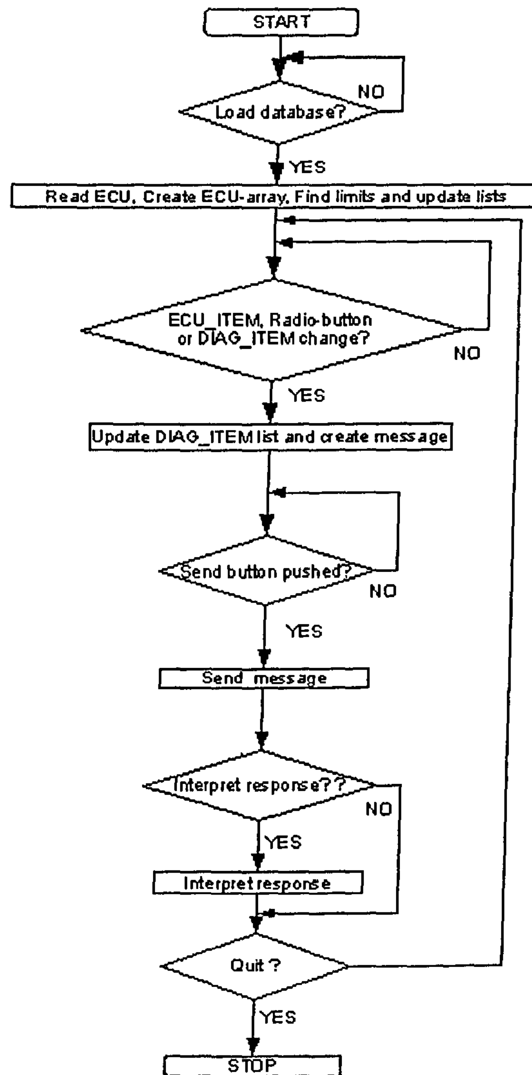



*Figure 26: DHA flowchart*

### 4.2.4 Vital Sequences

In the DHA-tab there are several parts that need further explanation. More details on the second part "Read ECU, Create ECU-array, Find limits and update lists" on the flowchart above will follow. It correlates with some subVI:as. How the database is read and array is created and how the limits of the "DIAG_ITEM":s and "SERVICE_ITEM":s are found will be explained in this section. The flowchart is illustrated in Figure 27.

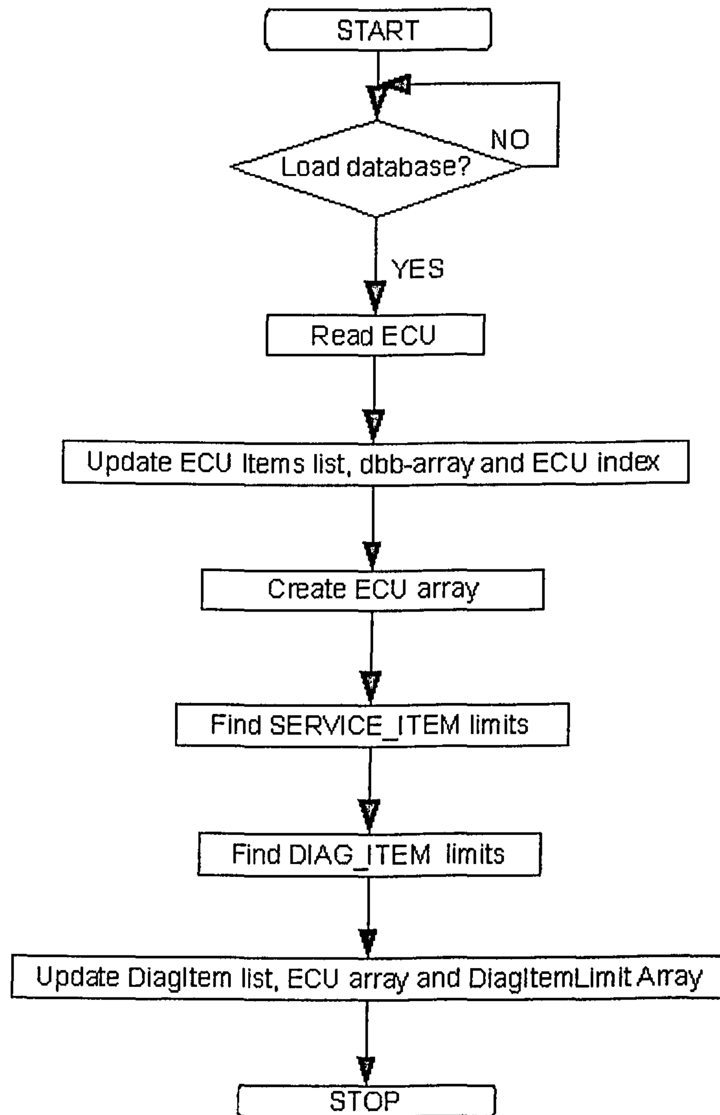



*Figure 27: DHA vital sequences*

### *Read ECU*

The inputs and outputs of the subVI is illustrated in Figure 28. The subVI reads the database row by row. The relevant row numbers are stored in the ECU-index and it creates a cluster to the ring menu in the front panel.

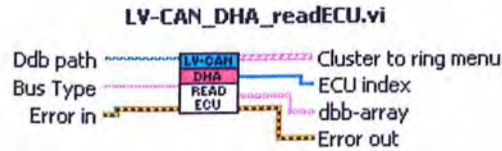



*Figure 28: Read ECU icon*

It checks each row for the string "ECU_ITEM" and if it finds it then it checks if the ECU uses the user defined bus speed. If the speed correlates with the user defined it sets a flag of a boolean type to true. That means that program has found the beginning of the ECU and now only need to search for a string that contains "END". This is because not all of the ECU:s should be displayed in the front panel, just those that corresponds with the bus speed set by the user.
The row numbers of begin and end of the ECU is stored in an array named "ECU index", the values of the array will change if the user chooses to change the bus speed. The names will be forwarded to the list of ECU:s in the front panel.
Figure 29 below explains how the Read ECU works:

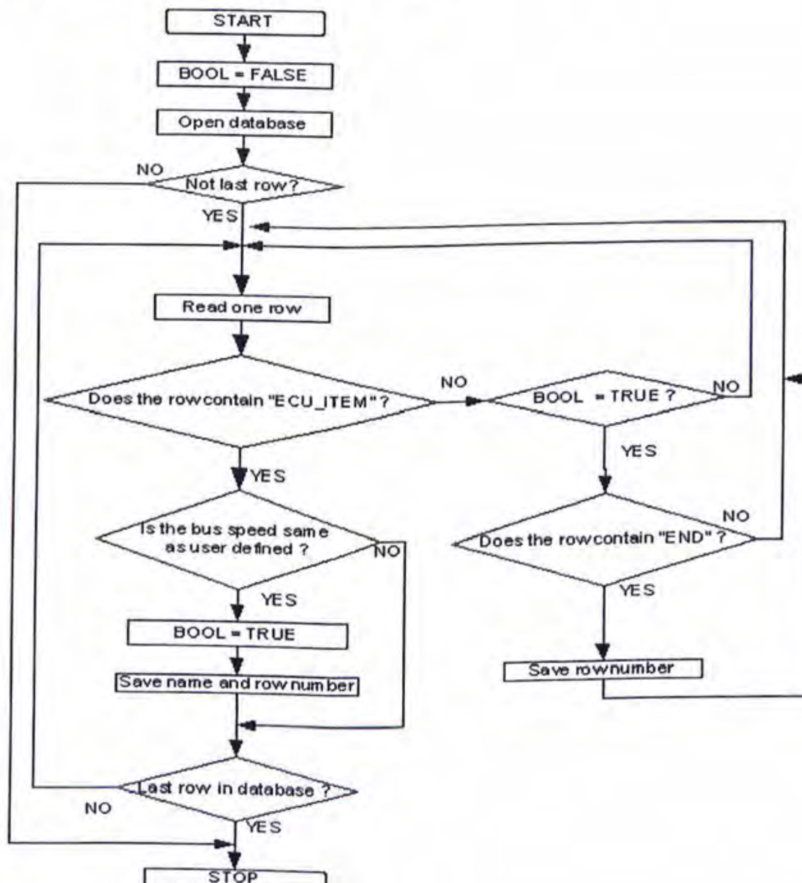



*Figure 29: Read ECU flowchart*

### *Create ECU array*

This part creates the "ECU array". The array contains the information of the current ECU. The VI uses the "ECU index" array to get the limits of the current ECU. It then start to index the array with empty columns instead of blank spaces. The inputs and outputs are illustrated in Figure 30.

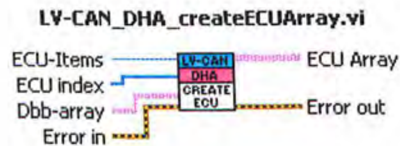



*Figure 30: Create ECU array icon*

The first column in the ECU array functions as variable storage. Instead of using local variables, the information is available from the ECU array. This is the additional information stored at the first column in the ECU array:

| | |
|---|---|
| Row 0: RadioButton | The actual radio button, tells which service that is chosen |
| Row 1: Service Begin | The row number at the beginning of the actual Service |
| Row 2: Service End | The row number at the end of the actual Service |
| Row 3: Single DiagItem or List | The DTC contains only one relevant DIAG_ITEM |
| Row 4: Number of Removal | Number of obsolete rows in the beginning of the Service |
| Row 5: Diag begin | The row number at the beginning of the actual DIAG_ITEM |
| Row 6: Diag End | The row number at the end of the actual DIAG_ITEM |
| Row 7: Message String | The composed message that will be sent |

### *Service Limit Search*

This VI works similarly as the Read ECU does. The purpose of this VI is to find the row numbers of the first and the last row of the Service. The inputs and outputs are illustrated in Figure 31.

*Figure 31: Service limit search icon*

The actual Radio button is stored in the ECU-Array at the column 0 and row 0. The VI starts with checking which Service that is the current one. Then it searches in the ECU array for a string that matches the service name. If they match, it saves the row number and start searching for the end of the SERVICE_ITEM. A flow chart describing the functionality is illustrated in Figure 32.

*Figure 32: Service limit search flow chart*

### *DiagLimitSearch*

The purpose of this subVI is to find the row number for BEGIN and END of every diag item in the actual service item. The VI starts with retrieving the row numbers for the beginning and end of the current service item. It then starts its search for the row numbers for begin and end for each diag item. The VI creates strings to the menu rings in the front panel. The inputs and outputs are illustrated in Figure 33 and a flowchart describing the functionality is illustrated in Figure 34.

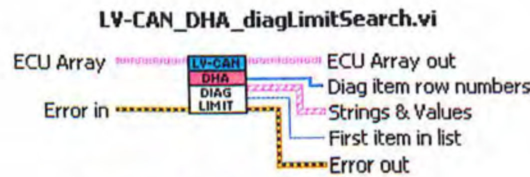



*Figure 33: Diag limit search icon*

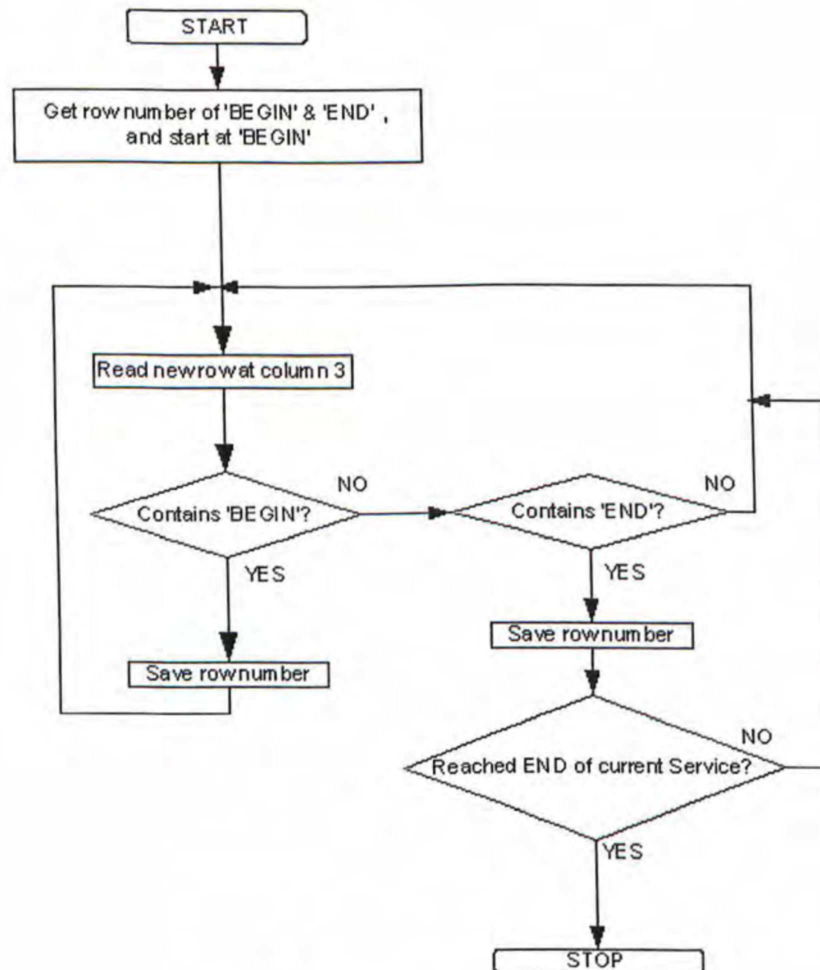



*Figure 34: Diag Limit Search Flow chart*

### *Create Message*

The message is composed of a number of bytes that are found in the database. The first byte in the message is size dependent. As the message is a single frame bit #7-6 is "11", the message is a diagnostic request or a diagnostic response so bit #5-3 is "001", last three bits varies with the number of bytes in the message. For instance if the message consists of three bytes the bits # 2-0 will be " 011". So the result will then be "CB" in hexadecimal. If the number of bytes is four the byte will be "CC" in hexadecimal and so on. This byte is called the "Format byte".
The next byte is found in the first row in the ECU item. It is called the COM_ADDRESS. The byte after the COM_ADDRESS is VALUE from the first row of the Service item. The byte after that is found in the first row of the DIAG_ITEM and is also called VALUE. If the Service is Read Current Data or Input Output Control there will be a additional byte to the message. It is called VALUE and resides two rows down in the DIAG_ITEM. The inputs and outputs are illustrated in Figure 35. Figure 36 illustrates the logic of Create Message.

*Figure 35: Create message icon*

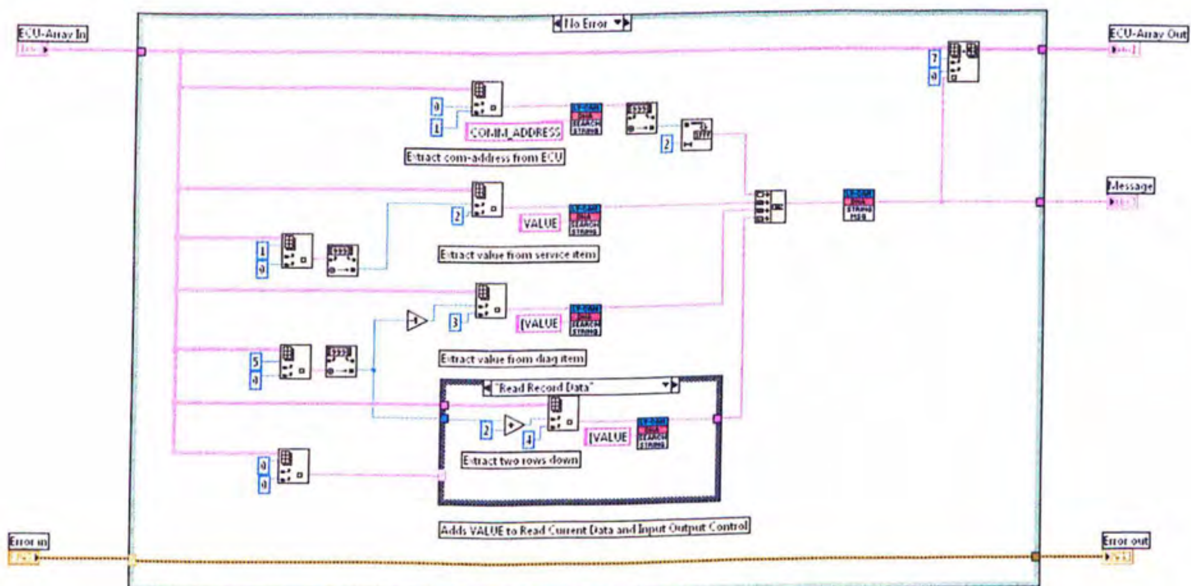



*Figure 36: Create message block diagram*

### *Send message*

When the message is composed and the message has been sent, the next step will be to read the bus to check if there is any response to the sent message. In the "read bus" VI, the composed message is read from the ECU-array so the VI:a knows what response message to expect. This way it has a string that it compares the messages on the bus with. If the string is equal to a message on the bus it start to check if it is a single frame or a last frame, and if the message is one of these it will get the relevant data by checking the format byte and then stop checking the bus for messages. If the message is not a single frame or a last frame, then it is an intermediate frame and the VI will store the bytes after an offset of two bytes. It will continue to do this until it encounters a last frame. All the data bytes is stored in a string that will be interpreted further on. Send message is illustrated with the flowchart in Figure 37.

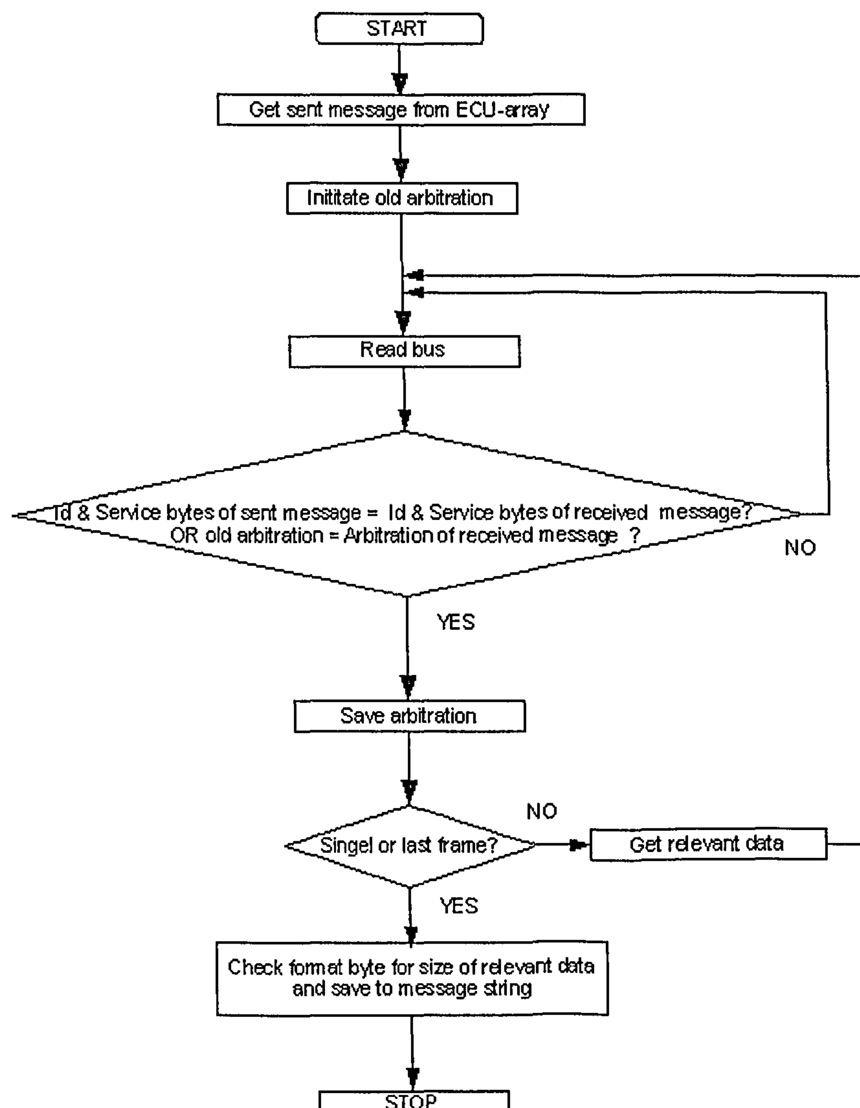



*Figure 37: Send message flowchart*

**Single or last frame**

To determine if the received message is a single message or a last frame the program uses the single OrLastFrame VI. It starts with checking if the message starts with 'C' or '4'. This determines if it is a single or a last frame. It the message is a single or last frame, the VI uses the getRelevantData VI to calculate the amount of relevantdata in the message. If it is neither a single frame or a last frame it will be treated as an intermediate frame and add the six last data bytes to the message string. The inputs and outputs are shown in Figure 38 and the logic is shown in Figure 39.

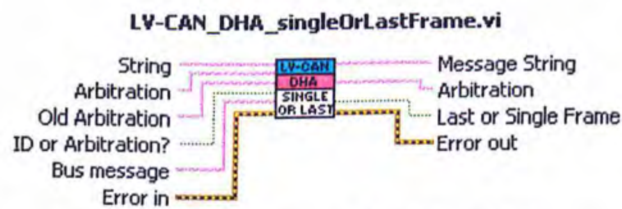



*Figure 38: Single or last frame icon*

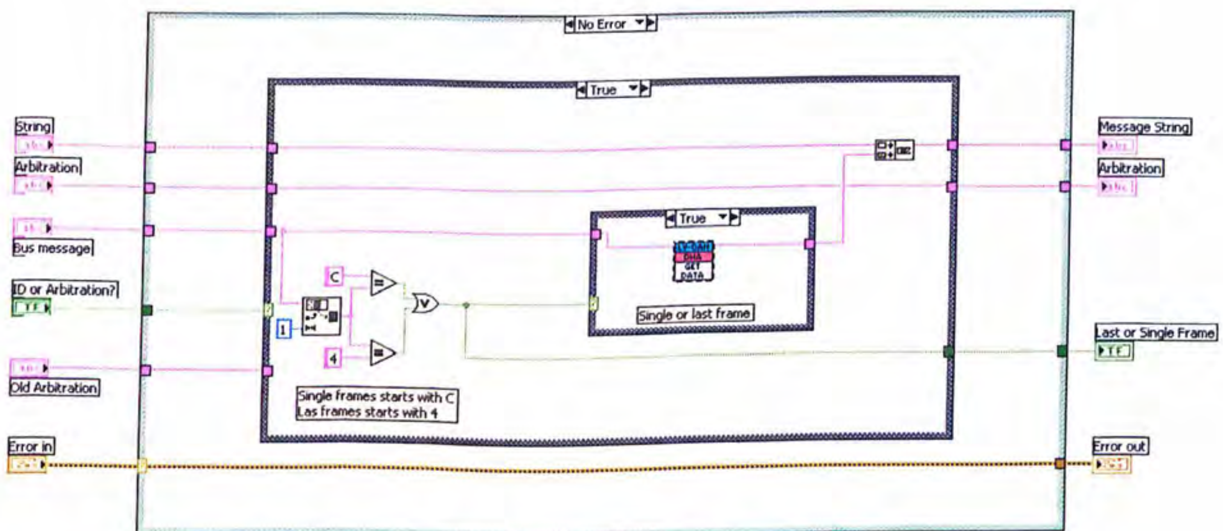



*Figure 39: Single or last frame block diagram*

### Get relevant data

The amount of relevant data is calculated through the getRelevantData VI. It uses the first byte of the message, also known as the format byte, and mask out the last three bits.
They represent the amount of the data bytes that is relevant data. The message will fill out the message with zeros if not all of the data space is used, so knowing how many bytes that is relevant is crucial. The inputs and outputs of getRelevantData is illustrated in Figure 40 and the logic is shown in Figure 41.

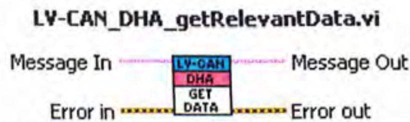



*Figure 40: Get relevant data icon*

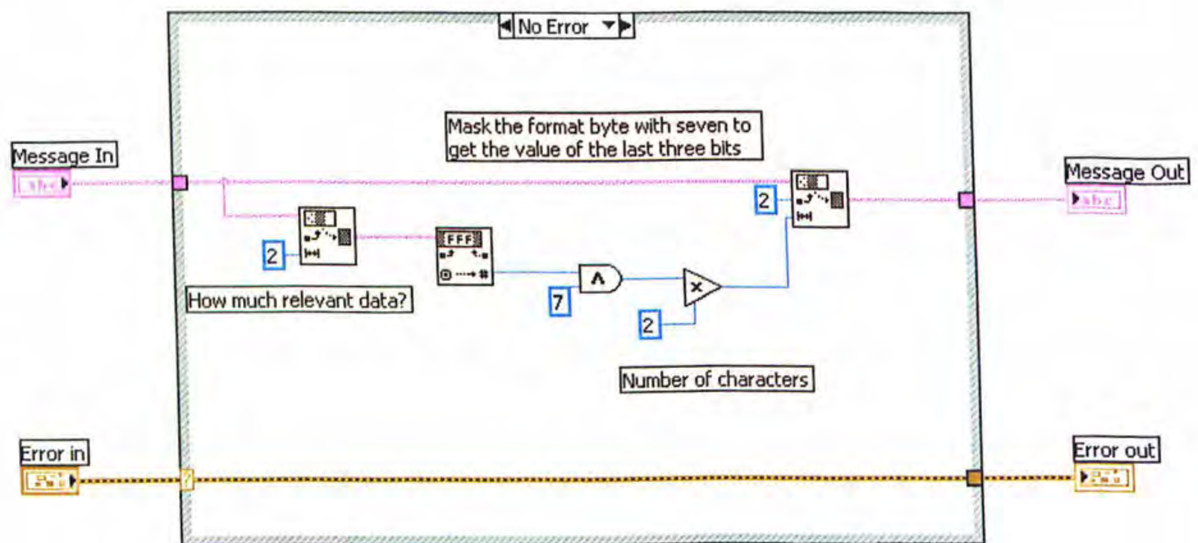



*Figure 41: Get relevant data block diagram*

The format byte:

| bit #7-6 | bit #5-3 | bit #2-0 |
|---|---|---|
| 11=Single frame | 001 and 010= diagnostic message | number of bytes |
| 10=First frame | 001 and 010= diagnostic message | number of frames is 111 not spec. |
| 00=intermediate frame | 001 and 010= diagnostic message | frame sequence number |
| 01=last frame | 001 and 010= diagnostic message | number of bytes |

### *Interpreting responses*

When a diagnostic request has been sent, a response from the ECU is to expect. This has to be decoded and presented in a format that is easily understandable. The information used for decoding these messages is located in the database file.

In the database-file, it is possible to find information for each diagnostic request response item, such as a compare-value, the base of the response item as well as the formula for decoding the response. A diagnostic request response can contain several items, and therefore it is also possible to see the offset and length for each response item. The offset indicates where in the response string that the information should be located and to avoid making unnecessary comparisons between the response string and the response items, there is no comparison being made if the offset in the database file exceeds the length of the response string.

In some cases there is a formula located in the database file, which describes how to decode and calculate the desired value, from the hexadecimal response string. These formulas can contain regular arithmetical operations such as addition, subtraction, multiplication and division. There can also be expressions of boolean algebra such as AND for example, located in the formula string. The formula strings also contain constants, that are expressed in different bases. They can be either decimal, hexadecimal or binary. To be able to interpret the formula strings correctly, and acquire the correct value from the response string, several VIs have been developed.

**Interpreting numbers expressed in different bases and formatting the formula string**
Since the constants in the formula strings are expressed in different bases, there is a need for converting them into decimal values, expressed as strings, and replacing the ones expressed in either binary or hexadecimal bases. There is also a need for a VI that can perform AND-operations expressed in a string.

**Formatting the formula**

*Figure 42: Format formula string Icon*

This VI basically just removes all blankspaces in the formula string, as well as replacing all "." with ",". This is necessary to make sure that LabVIEW interprets floating point values correctly. As can be seen in Figure 42, there is just a string input as well as a string output.

### Interpreting binary

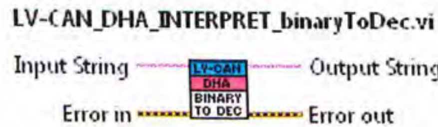



*Figure 43: Binary to decimal Icon*

The VI (see Figure 43) for interpreting the strings with binary numbers utilizes a function that exists in the mathscript node called "bin2dec". It can convert a binary string to a decimal integer. To determine which numbers in the formula string that are actually binary numbers, a simple scan has to be performed. This VI will search for the combination "0b". This is used to signal that the following digits represent a binary number. After it has been interpreted as a decimal value, the binary string is replaced with a decimal string.

### Interpreting hexadecimal

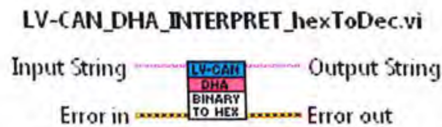



*Figure 44: Hexadecimal to decimal Icon*

To determine the constants that are expressed in hexadecimal format, the VI will search for the combination "0x". This signals that the following characters represent a hexadecimal value. This is converted to a decimal value and then replaced by a decimal string. The input and output signals are shown in Figure 44.

### Interpreting AND

*Figure 45: Interpret AND-operator Icon*

A VI (see Figure 45) for handling AND operations expressed in a string has been created. It will find the & sign, convert the two decimal strings that surrounds the &-sign into numerical values, and perform the &-operations, and finally return a numerical value.

**Calculate Formula**

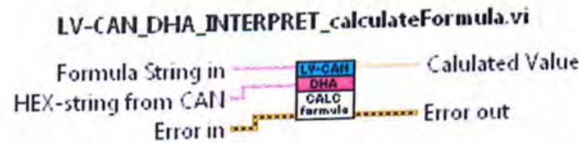



*Figure 46: Calculate formula Icon*

The VI (See Figure 46) that is used to calculate a value based on the formula from the database file and the response from the ECU uses the VIs previously described. Connected in a sequence (see Figure 47) they will result in a VI that converts both binary and hexadecimal values into decimal values. It will also perform calculations for any eventual AND-operators that are located in the formula. Finally, it will perform the arithmetics that the formula string describes, and return a numerical value.

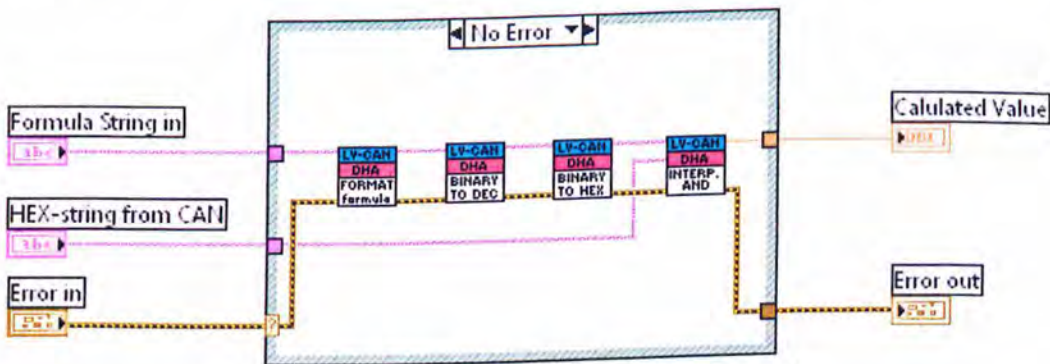



*Figure 47: Block diagram for Calculate formula*

**Comparing values**

When a message has been decoded and numerical values have been acquired, you might have to do a comparison. This information is located in the database file, and there will be an indication if a comparison should be done. A typical comparison can be to see if the numerical value that you have decoded is for example greater than a defined constant. This would be indicated in the database-file as a compare value of ">XX", where "XX" is the value of the constant. A VI for handling these operations has been created. See Figure 48.

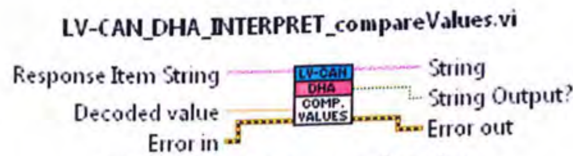



*Figure 48: Compare values Icon*

There are several different types of comparisons that can be done. The VI will search for different types of comparisons and depending on which sign that is present, that operation will be performed. In the case that there is no compare value, the calculated value will be presented. A flow chart illustrating this function is shown in Figure 49.

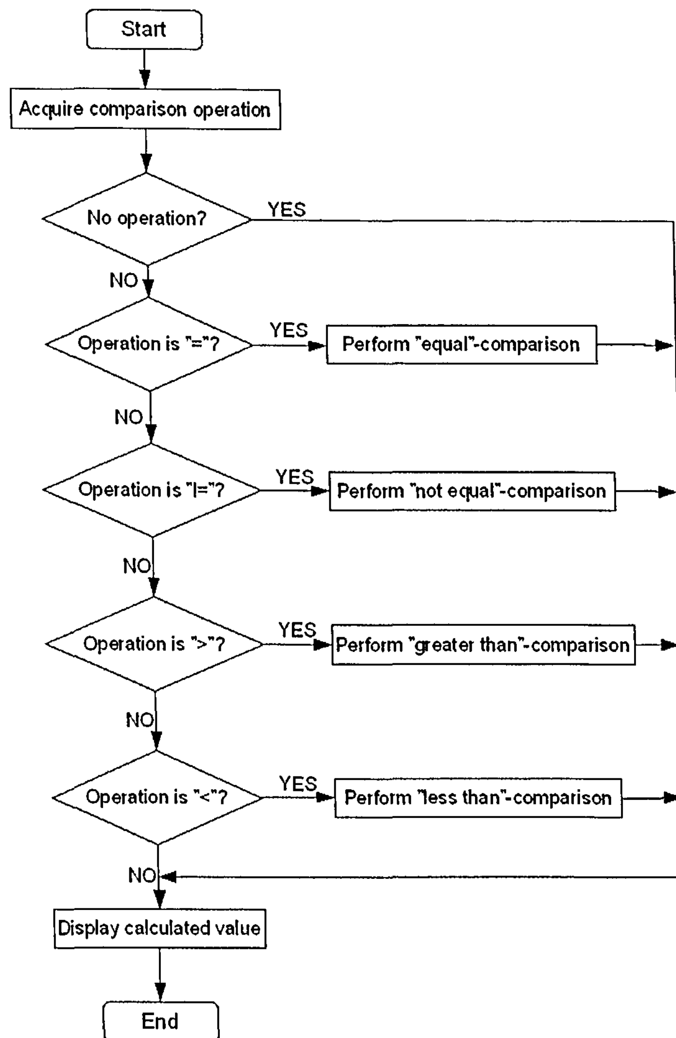



*Figure 49: Flow chart of functionality of comparisons*

**Handling different bases**

As previously mentioned, there is information regarding the base of the values that are decoded. Some may be interpreted as hexadecimal values, some as decimal. Some may even be expressed as ASCII or BCD. To handle these different bases a VI was developed. There is not much to mention about how the bases of decimal, hexadecimal and binary values are handled, since this is basically handled by the VI that calculates the formula. However, when the received message is to be interpreted as ASCII or BCD, an explanation might be helpful.

**Decoding ASCII**

ASCII is short for American Standard Code for Information Interchange[5] and is a way to express different symbols. Each symbol is represented with a decimal value. For example the letter 'A' is represented by the decimal value of 65.

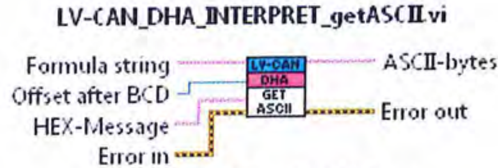



*Figure 50: Get ASCII Icon*

This VI (See Figure 50) will convert the hexadecimal string to a decimal value, then it will output a string containing the ASCII characters that these decimal values represent.

**Decoding BCD**

BCD is short for Binary-Coded Decimal. It is a way of presenting numbers digit by digit. For example the number 4 equals $0100_2$ in binary, and the number 7 equals $0111_2$. With BCD the number 47 would be expressed as $0100\ 0111_{BCD}$ [6].

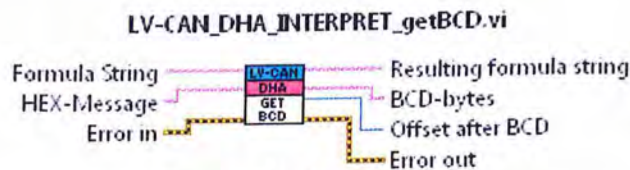



*Figure 51: Get BCD Icon*

This VI will read how many bytes that are expressed in BCD and return a substring containing these bytes. The Icon can be seen in Figure 51.

**Decoding both ASCII and BCD**

For some diagnostic items, the response string will contain both BCD and ASCII. A typical base string could for example be "4 BCD 3 ASCII". This means that the first four bytes should be interpreted as BCD, and the next three as ASCII. Figure 52 shows the icon for the VI that handles this.

*Figure 52: Interpret ASCII and BCD Icon*

By using the VI for ASCII and BCD in a sequence, it is possible to decode a message with a mixed base. The block diagram is illustrated in Figure 53.

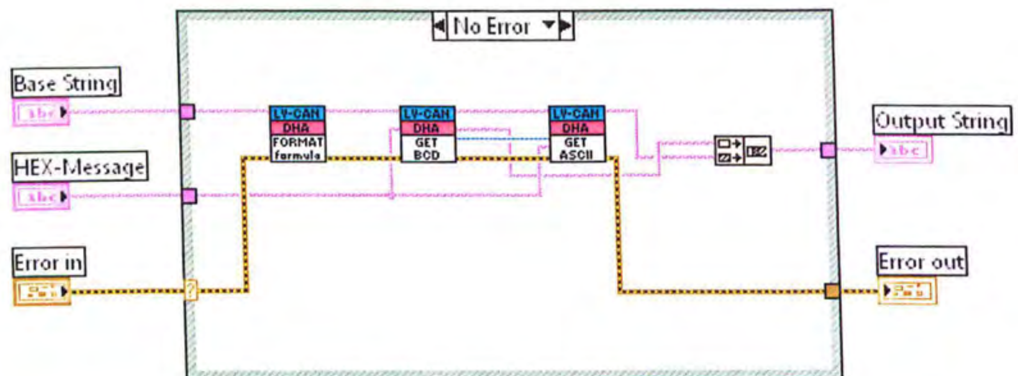



*Figure 53: Block diagram of ASCII and BCD interpretation*

## 4.3 AnalyzeCAN

This part of the program has adapted some features from a widely used CAN-tool. It enables the user to graphically monitor how the different signals or channels varies in time. The information about the channels are located in a .dbc-file. For instance, max and min values and also units for each channel are stored in this database file. The .dbc-format is the quasi-standard for describing CAN communication data.

### *4.3.1 Front Panel*

It is possible to load a .dbc file by entering the path of the file and pressing "Load dbc file", see Figure 54. By doing this, the different messages in the dbc file are available for browsing. When a specific message is chosen, the different channels in the message appears. Using the two buttons "Add item" and "Remove item" it is possible to add and remove desired channels to a list called "Analyze list". This list contains the different channels that will be available for graphical monitoring. When pressing "Run analysis" the upper part of the front panel will be disabled, and curves will appear in the waveform chart. These curves are updated in real-time, and they are being sampled at a rate of 50 ms. Each channel will be represented by an individual curve with an individual color.

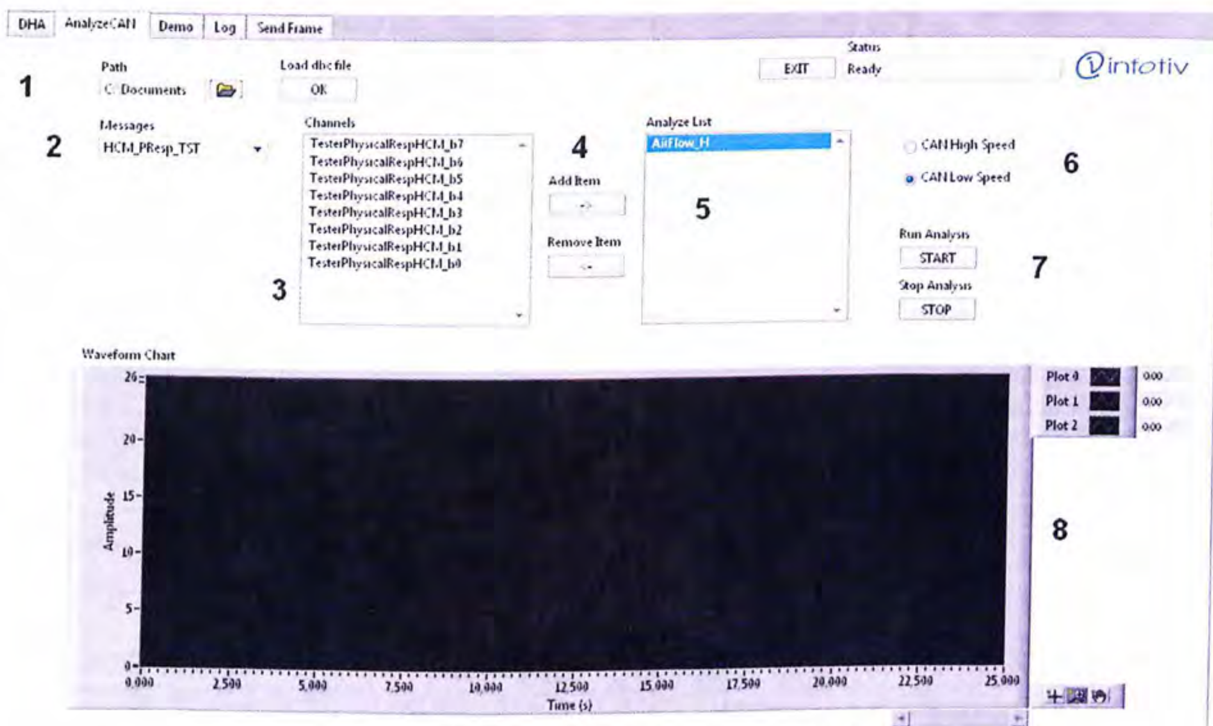



*Figure 54: Front Panel of AnalyzeCAN tab*

1. File path for .dbc file
2. List of messages in loaded .dbc file
3. List of channels in the chosen message
4. Buttons for adding and removing channels from "Analyze list"
5. List of channels to be graphically monitored
6. Radio buttons to select High or Low speed CAN
7. Buttons for starting and stopping the analysis
8. Waveform chart displaying chosen channels over time

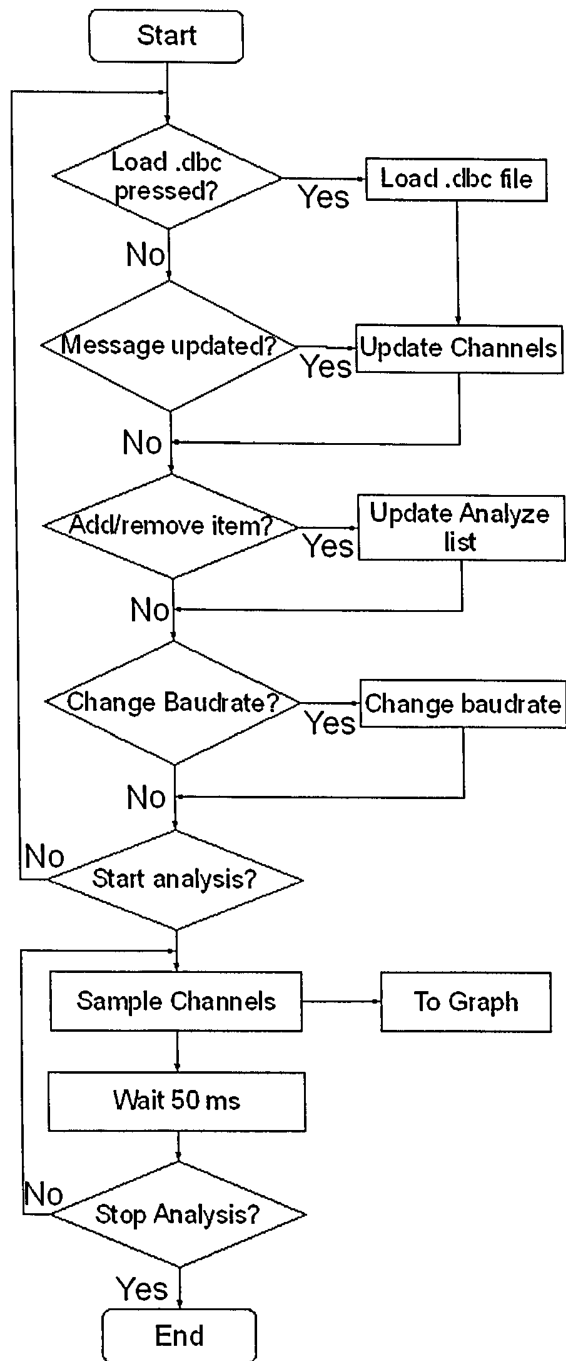



*Figure 55: Flow Chart of program in tab AnalyzeCAN*

The flowchart of the AnalyzeCAN tab is shown in Figure 55.

### 4.3.2 Vital Sequences

There are two ways of handling data on the CAN-bus when using hardware from National Instruments. The first one is the CAN frame. This is a raw frame consisting of an ID, type, data bytes and timestamp. The second one is the CAN channel. This is at a higher level than the frame. It represents a field in the data of a specific ID, which is scaled to a floating point value in physical units. If you want to use a .dbc-file for monitoring signals on the CAN bus, you have to use the channel API. Since the hardware that is used, doesn't have support for using LabVIEW's channel API, there is a need to set up a virtual CAN-bus. This virtual bus will make it possible to send all messages that were received on the actual CAN-bus, onto a virtual bus that has support for channel API [1]. The VIs for handling this conversion will be shortly described.

#### Init VT-bus

The first part of the virtual bus is the initilazation. This is done in a single subVI, which initilazes and sets up the different parameters that are needed for the virtual bus to work. Figure 56 shows the icon for the VI that handles the initialization.

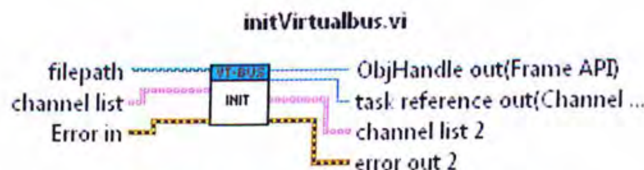



*Figure 56: The VI initVirtualBus Icon*

The input required is a filepath, indicating the location of the .dbc-file that is used. A list of the channels that are to be monitored as well as an error line that is used to disable the functionality of this VI, in case of an error. What this VI does is that it creates two virtual CAN interfaces. One using the Frame API, and another one using the Channel API. The idea is that the messages that are received on the actual CAN-bus, can be sent on the virtual bus with the Frame API, and then be read on the virtual bus by the Channel API. As can be seen in Figure 57, the block diagram is divided into two layers. The top layer contains the interface using the Frame API, and the bottom layer contains the interface using the Channel API. Two types of object handles are avaliable for output, as well as the channel list that was used as an input.

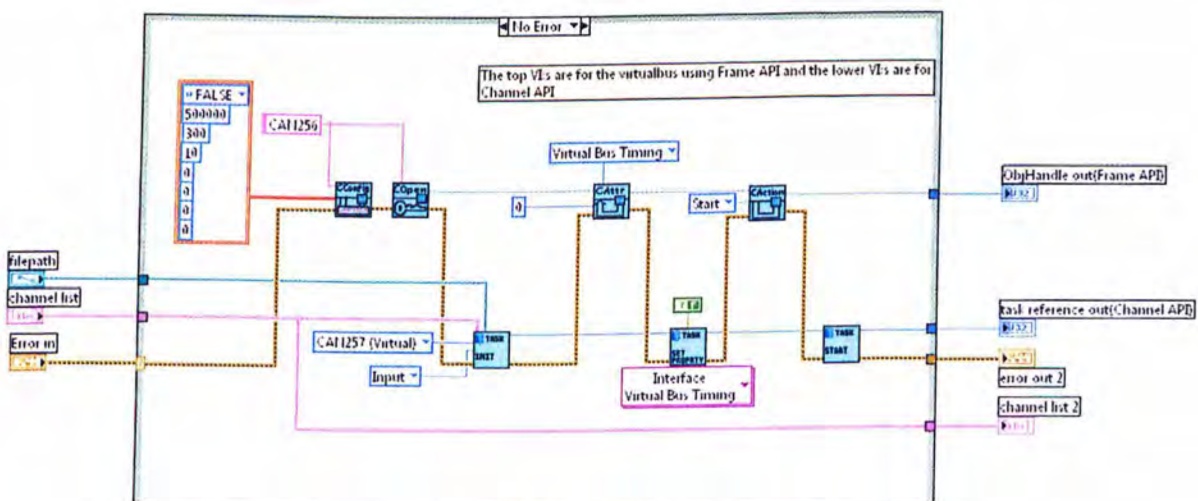



*Figure 57: Block diagram view of Init VT-bus*

### *Get max & min values*

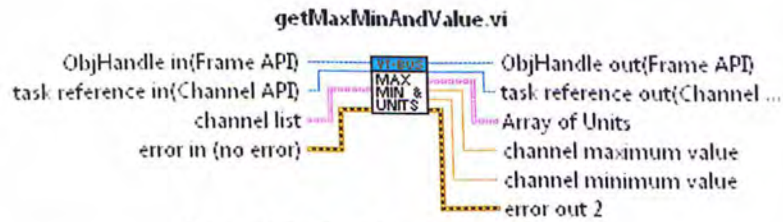



*Figure 58: Get Max & Min values Icon*

To get a nice presentation of the signals in the waveform window, the maximum and minimum possible value for each channel is determined. The largest of them all will be the upper limit on the Y-axis, and the the smallest will be the lower limit on the Y-axis, these are the channel maximum value and the channel minimum value in Figure 58. The information about the range in which each channel varies, can be found in the CAN database file.

The principle of dynamically determing the maximum and minimum value is shown in Figure 59.

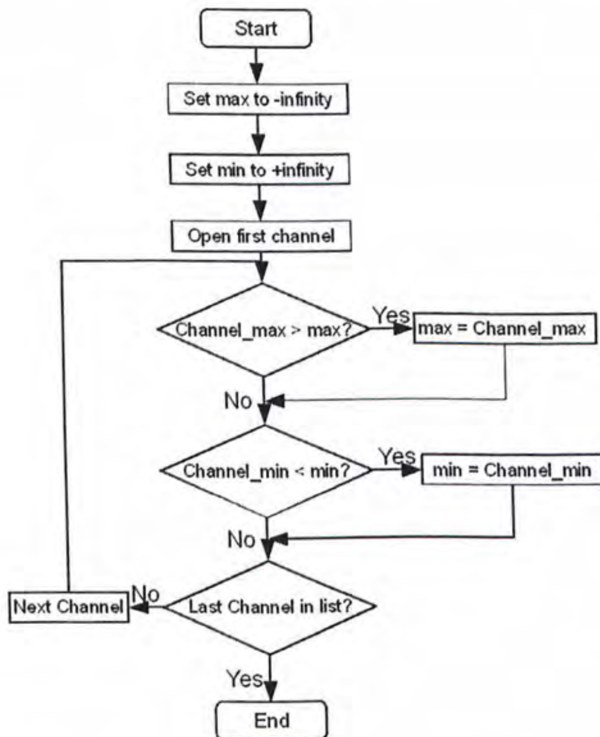



*Figure 59: Flow chart of determining the maximum and minimum value*

***Read, convert to channel data***

There are several inputs to the VI that handles the conversion between frame data and channel data. There are three object handles, one for each CAN-interface (CAN0, CAN256, CAN257). There is also an initialized array that is used for storing the new CAN-frames that are received on the bus. The ouputs are the three object handles, and an array containing samples of the channels, chosen to be monitored. These signals are shown in Figure 60.

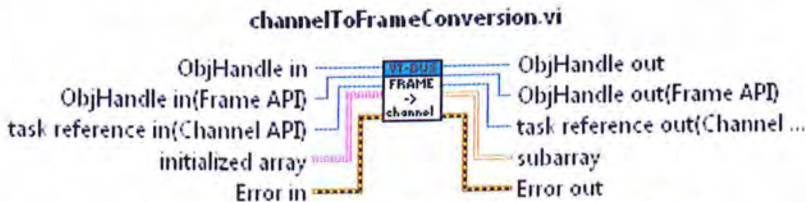



*Figure 60: Channel to frame conversion Icon*

This VI is placed within a while-loop that is executed every 50 ms (see Figure 61). Each execution, a read on the actual CAN-bus (Interface CAN0) is performed. An array for storing the new frames that are read, has already been initialized to improve performance. The data and arbitration are extracted from each frame, making it possible to send the received data on the virtual bus with the virtual interface using Frame API, and reading it with the virtual interface using the Channel API. The information is then ready for output as a double, with no scaling or formatting needed. If the virtual channel interface has been initialized to monitor several channels, The data will be available in the form of an array.

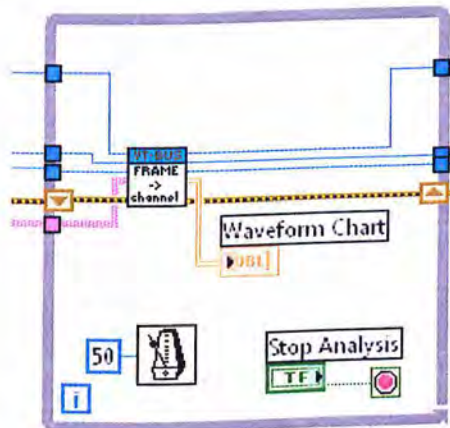



*Figure 61: Block diagram of while loop*

Figure 62 and 63 illustrates the functionality of the frame to channel conversion.

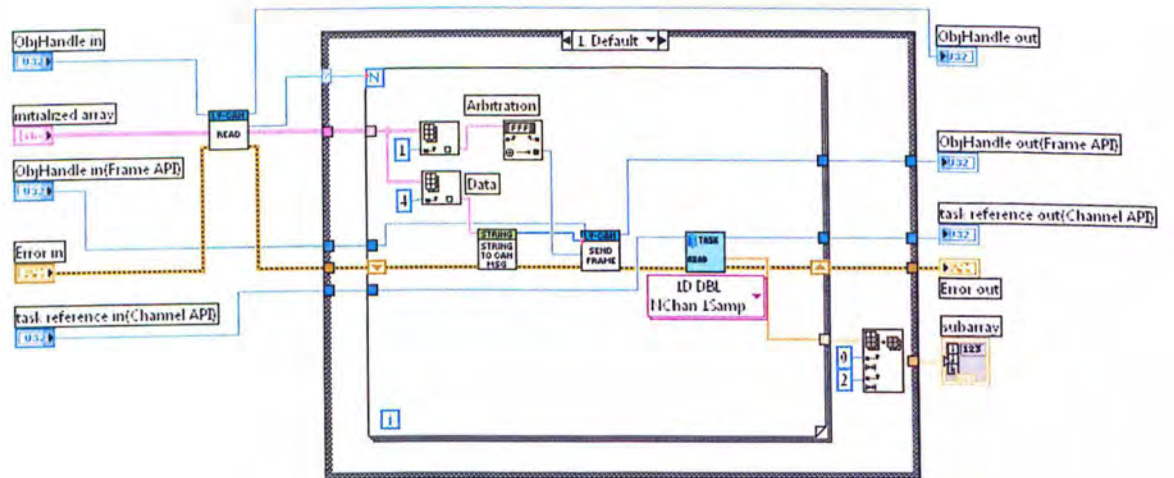



*Figure 62: Block digram view of frame to channel VI*

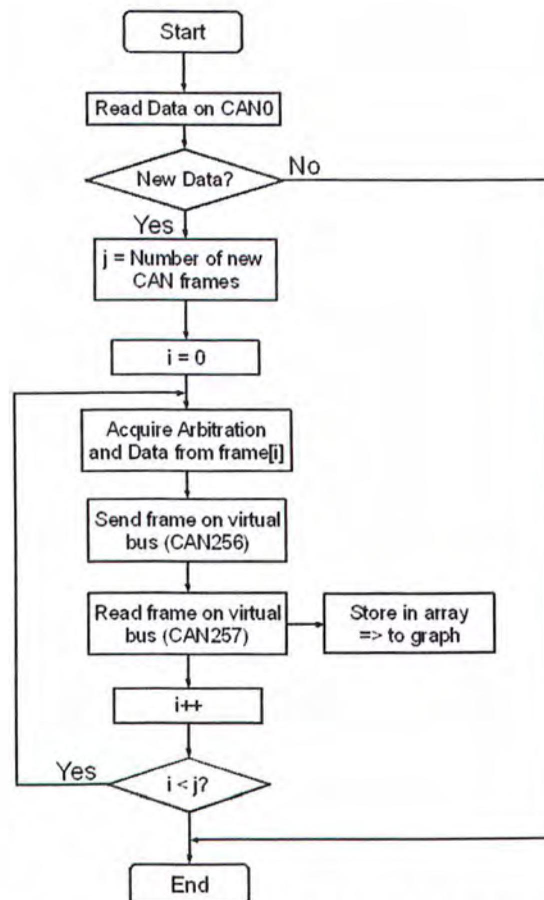



*Figure 63: Flow chart of frame to channel conversion*

Each iteration shown in Figure 63 is executed each 50 ms, until the user presses the "stop analysis" button.

## 4.4 Demo

This part of the program is used for demo purposes. It is intended to be used at Infotiv's test center.

### 4.4.1 Front Panel

The front panel contains a photo of a Driver Information Module or a DIM. This is located at the driver's seat in most modern cars. It presents the driver with important information such as the speed at which the vehicle is traveling, the engine speed, fuel level and engine coolant temperature. It is also used to signal warning lights if for example a seat belt is not fastened or if there is another type of problem with the car.

It is intended to be used while having your CAN-hardware connected to a car or simply just a DIM. There are two modes of operation.

The first is a Demo-mode. It is a sequence of events that activates when pressing the button "Run Demo". When activated, a sound of an engine starting will play from the speakers of the computer. While playing this sound, the indicators presenting the vehicle speed and the engine speed will move to fit the sound of the running car. While they are moving in the front panel, these same values will be sent on the CAN-bus, giving a physical representation on the actual DIM that correlates with the movements of the indicators in the front panel.

The second part is used to manually set the indicators on the actual DIM, by moving the red needles in the front panel. It is also possible to set different indicator lights located at the bottom left of the DIM, by clicking them. When a needle has been dragged to a new position, or a LED has been clicked, the appropriate command for setting the new value on the real DIM is being sent on the CAN-bus. The Front panel of this tab is shown in Figure 64.

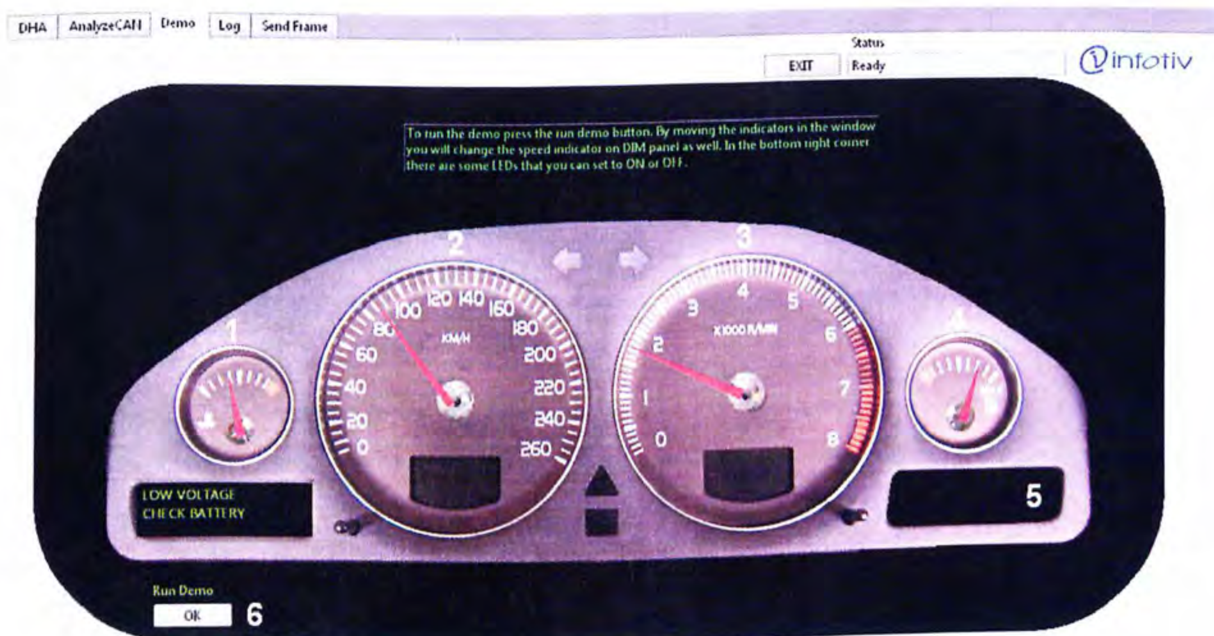



*Figure 64: Front Panel of tab Demo*

1. Needle for setting ”Engine coolant temp value”
2. Needle for setting ”Vehicle speed value”
3. Needle for setting ”Engine speed value”
4. Needle for setting ”Total fuel level value”
5. Pusbuttons for activating/deactivating ”LED indicators”.
6. Button for starting demo sequence.

Figure 65 shows the the principal of how the Demo tab works.

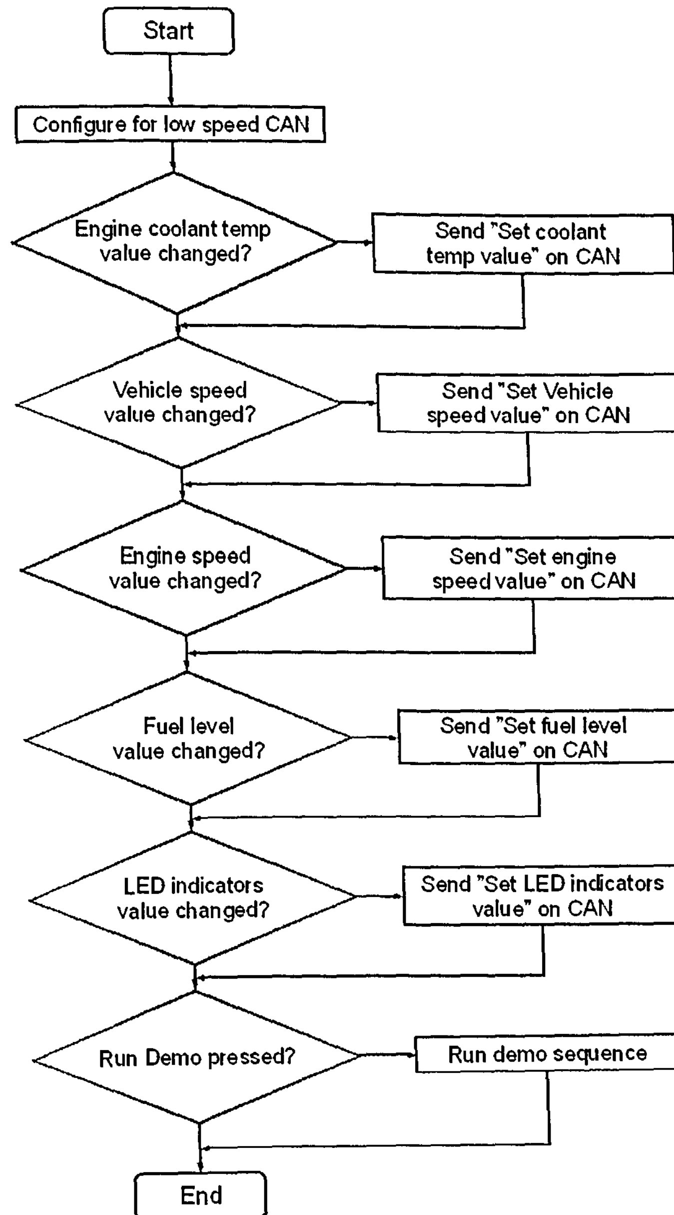



*Figure 65: Flow chart of Demo tab*

### *4.4.2 Vital Sequences*

Several sequences are important to get the desired functionality. A short description of how the demo sequence is constructed with sound playing and indicators moving, as well as how messages are created for setting the indicators manually.

#### *Start Demo*

The demo sequence is activated when the "Run demo" button is pressed. The sound of an engine roaring will be played and the needles indicating vehicle speed and engine speed will move, matching the sound playing. The values that the two indicators will assume, are extracted from two series of numbers from a sine function. The two needles vary with different speed and between different values, and therefore, the two sine functions are used with different periods and different amplitudes. Figure 66 shows how the two arrays of values are created.

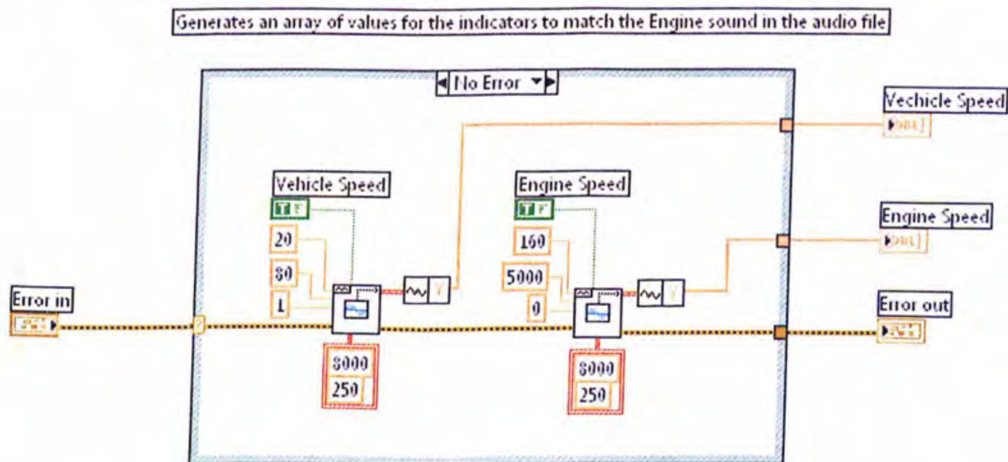



*Figure 66: Creating Arrays of values for demo sequence*

These values will be used to set the needels in the front panel of the program, as well as for creating messages that can be sent on the CAN-bus for setting the needles in the actual DIM.

The VI that creates the messages that are to be sent on the CAN-bus works as illustrated in Figure 67. It will have an input of three values, one containing the vehicle speed, one with the engine speed, and a final variable that is used to select which message that will be created. It will switch between creating a message for the engine speed and the vehicle speed, every other time.

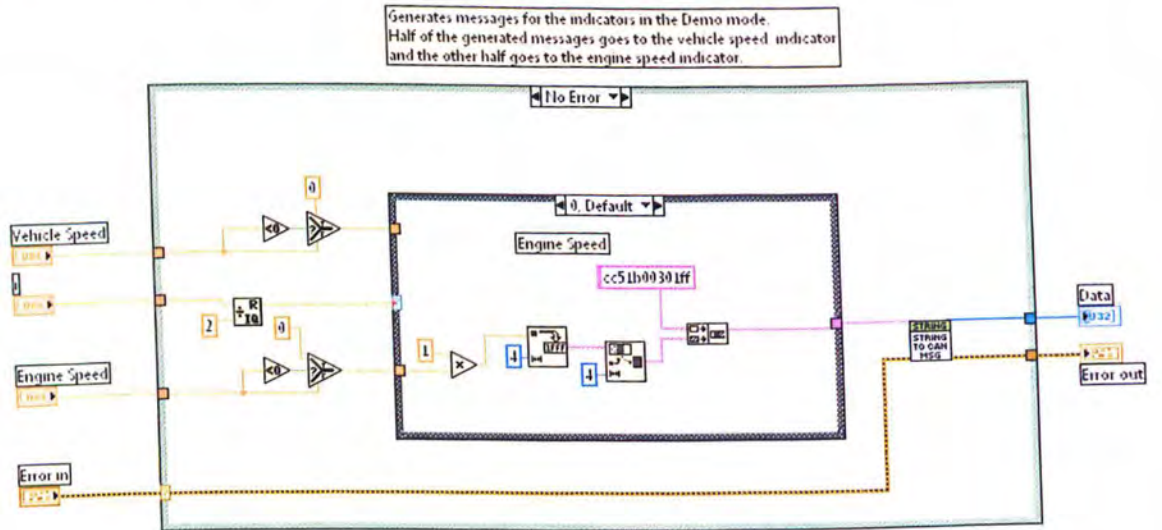



*Figure 67: Creating messages to send on the CAN-bus*

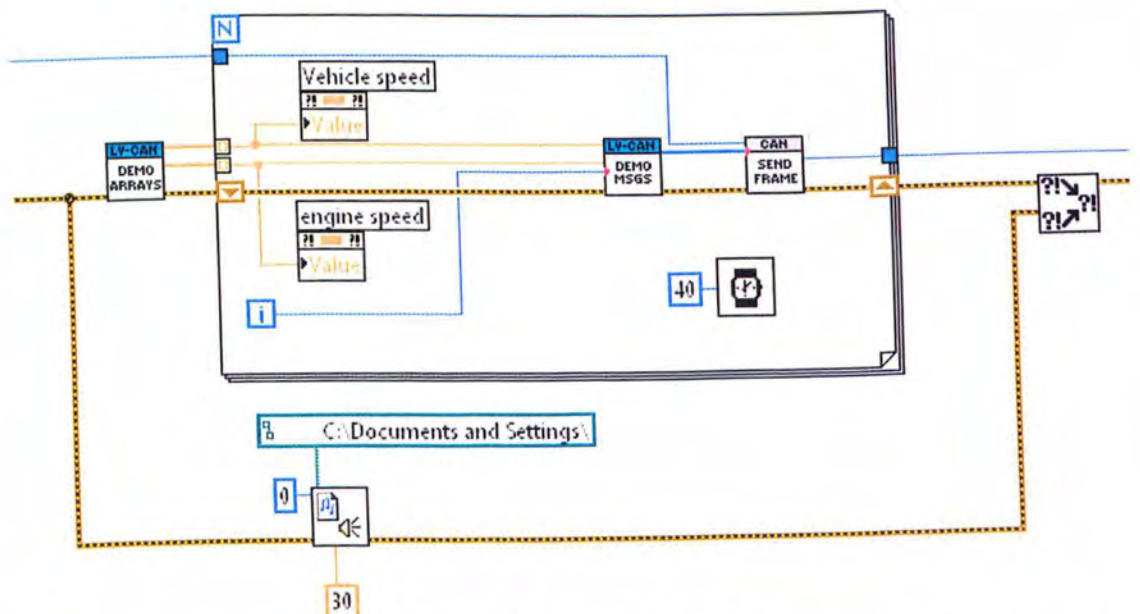



*Figure 68: Run demo sequence*

Finally the messages are sent on the CAN-bus. As can be seen in Figure 68, the sound file will be played at the same time as the for-loop is being executed.

### *Set indicators*

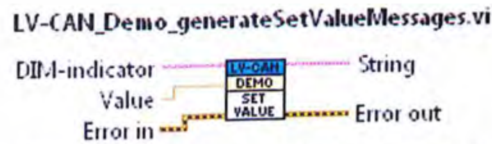



*Figure 69: Generate set value messages Icon*

When dragging and releasing one of the needles in the front panel, an action will be performed. A message for setting the new value is generated, depending on which needle that has been moved. This message is then being sent on the CAN-bus, resulting in an update on the actual DIM.

There are two inputs for the VI (see Figure 69). First there is a string input, which signals which indicator the message is for. The second input is a double, containing the new value of the moved needle. This value is then being scaled and converted to a string with the correct format for sending on the CAN-bus. Figure 70 shows the block diagram.

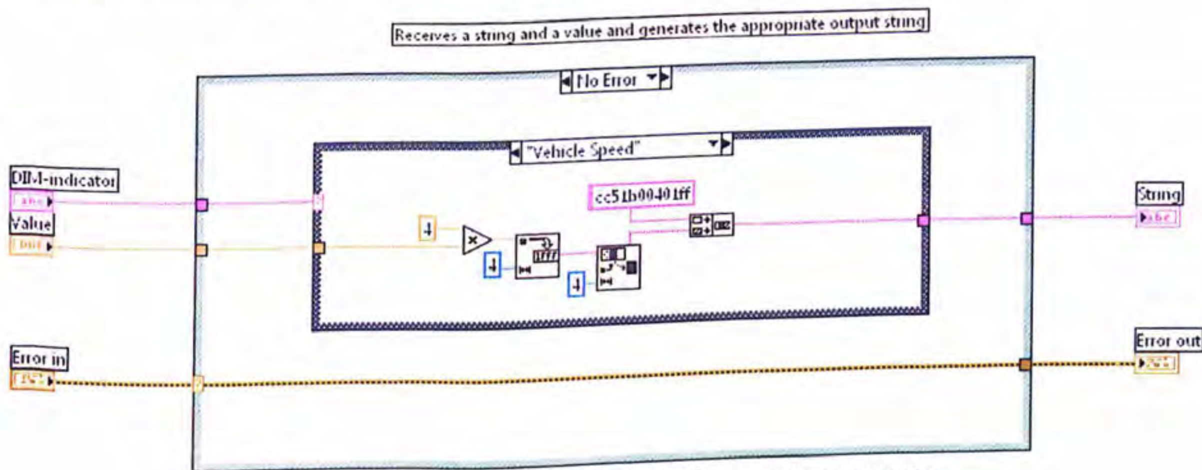



*Figure 70: Block digram of VI used to create messages*

### *Set LED-indicators*

The LED-indicators in the bottom right of the DIM, are actually buttons that have been created. Available controls has been customized and their appearance has been changed from their standard view by choosing different images for their individual on/off-state thus creating new types of controls. The result is buttons that are enabled or disabled by clicking them. The appearance is illustrated in Figure 71.

*Figure 71: Customized appearence of buttons*

The messages for setting the LED-indicators are created differs slightly from the messages for the indicators with needles. In the front panel of the program these are to be considered as booleans that are either true or false. To create the proper messages for setting these the following method has been used.

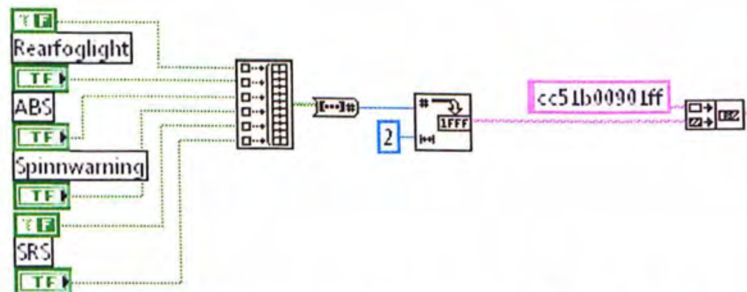



*Figure 72: Method for creating messages for LED-indicators*

The different booleans are used to build an array (see Figure 72). This array of booleans is then interpreted and converted into a number. This number is then converted into a hexadecimal string, and concatenated with the necessary string for setting a new value to the group of LED-indicators on the DIM.

## 4.5 Log

The log tab contains the logger part of the program. It makes it possible for the user to save the traffic to a file and open a log file to display the traffic. The log file is compressed using the ZIP format of compression in order to increase the amount of time that the program can log the traffic. The degree of compression was up to 20 times at best. It shows how good ZIP is on compressing text files.
To compress files, the ZIP program searches for patterns in the text file that repeat themselves. It replaces the repeated sequences with a shorter variant and builds a kind of dictionary. The dictionary contains the words that has been replaced and with what it has been replaced with, so it knows how it should rebuild the text file again. For the log program this means that it can log the traffic for about 20 times longer than without the compression.

### *4.5.1 Front Panel*

The user has the choice to save the traffic or not, if not then it will only display the traffic on the bus in the CAN log window. However, should the user want to save the traffic to a log file, then a valid path must be given or else it will not start. The log is saved in ZIP-archive that will be given the name of the current date. The log file/files inside the ZIP-archive will be named after the time stamp of the first CAN-frame of each file. When a log files reaches 10 MB in size it will start on a new log file in the archive that will be named after the first CAN-frames time stamp in that log file. This will continue until either the time is up, the user cancels the process or if it is only 10% left of free disk space on the medium where the archive is saved.
The user can set the duration for the logging by setting the Hours and Minutes with an appropriate time. The front panel is illustrated in Figure 73.

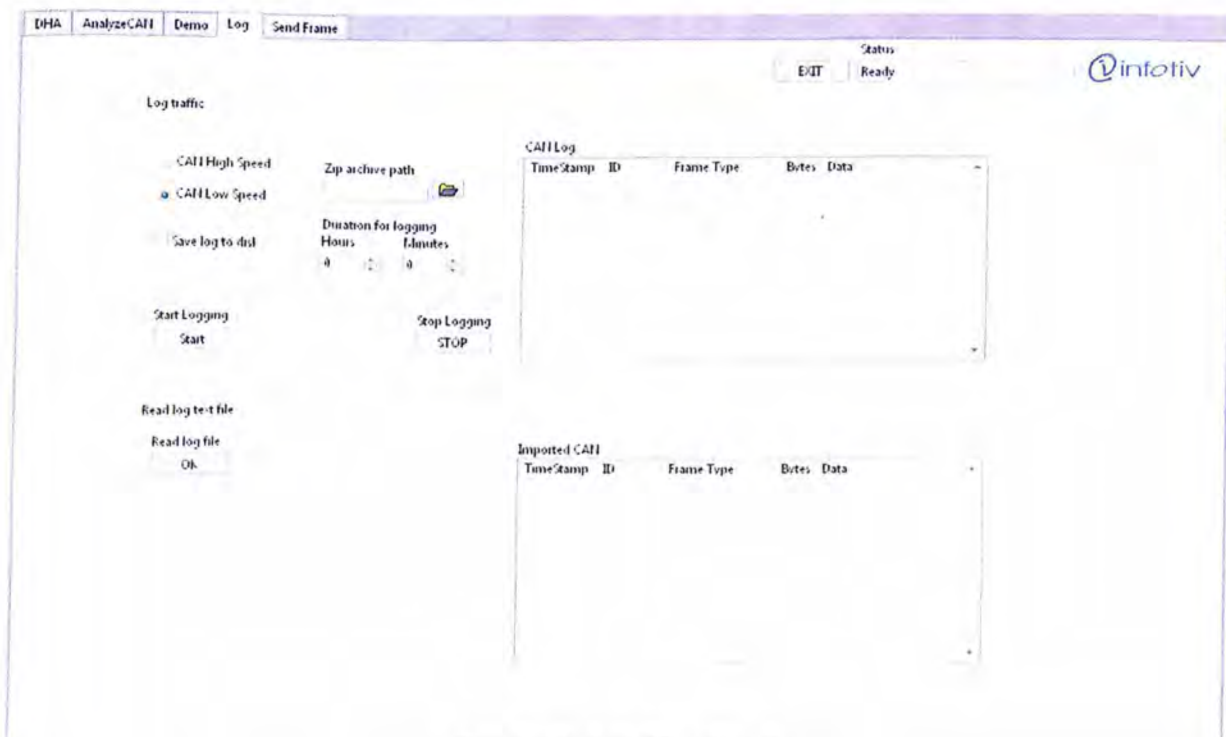



*Figure 73: Log front panel*

The flowchart below shows how the log operates when it saves the traffic to text files. It starts by waiting for the user to press the Start logging button. Then it uses the time values that were given in the Duration for logging as a run time. If the user has given a valid path for the text file it will proceed to "Create log file" or else it will abort. After the creation of the log file the actual logging starts. There are three conditions that will break the logging, if the medium only has 10 % left of free space, the user cancels or if the the time runs out. The function of is shown in the flow chart below in Figure 74.

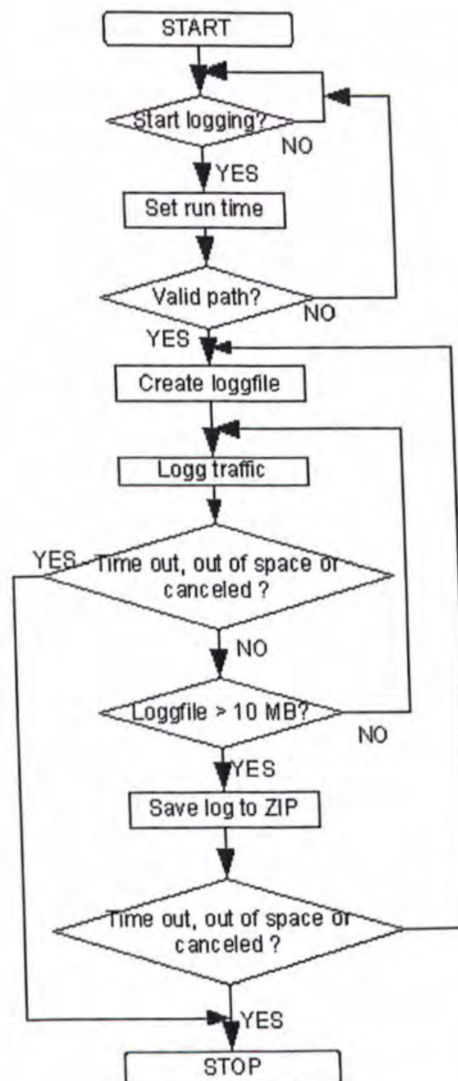



*Figure 74: Log flow chart*

### 4.5.2 Vital Sequences

Some steps in the flowchart will be explained here. The different steps correlates with some SubVIs and will therefore be explained more in detail. The sequences that will be explained further is Set runtime, Create logfile, Log traffic and Save log to ZIP.

#### Set runtime

The "setRunTime" subVI receives two values that correlates with minutes and hours that is given from the user. They are then converted to milliseconds. The time in milliseconds is then added to the milliseconds timer. The result is used to check if the runtime is out. The inputs and outputs is shown in Figure 75 and the corresponding logic in Figure 76.

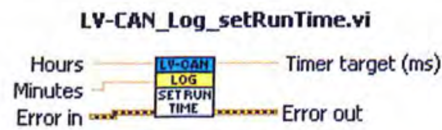



*Figure 75: Set runtime icon*

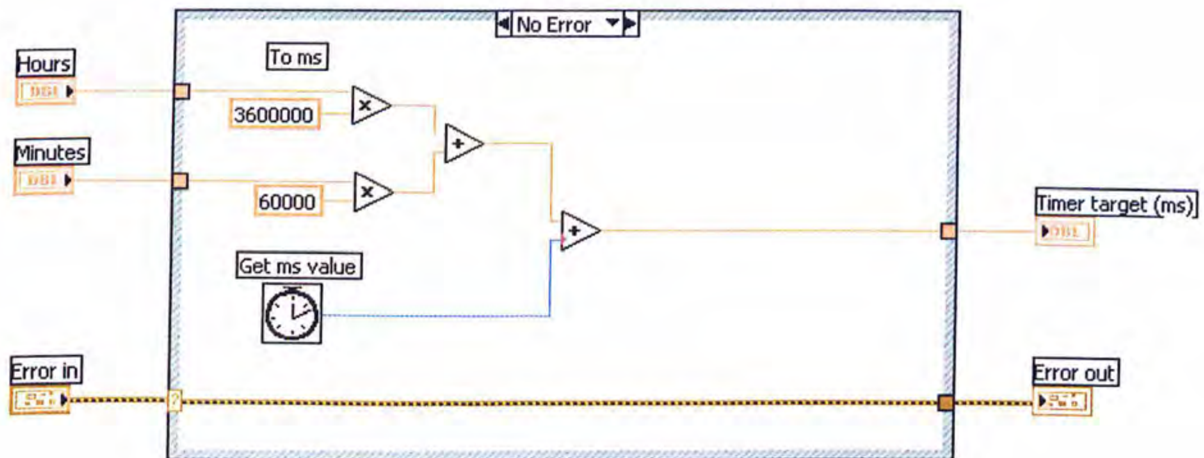



*Figure 76: Set runtime block diagram*

### *Create logfile*

The "createTemporyLogfile" subVI uses a tempory text file to store the traffic in. It is later stored in the ZIP archive with the time given in the Create logfile as name. The inputs and outputs is shown in Figure 77 and the corresponding logic in Figure 78.

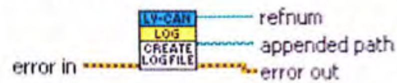



*Figure 77: Create logfile icon*

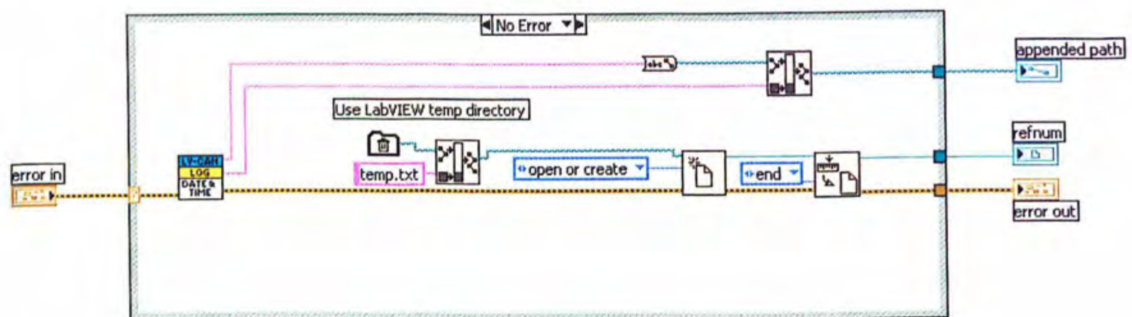



*Figure 78: Create logfile block diagram*

### *Log traffic*

The log traffic sequence consists several parts. It checks if the time has run out and if the user has canceled the process. The "Time Left" subVI uses the millisecond timer to check if the time that the user has run out. It does so by comparing the value of the millisecond timer with the value that the "setRuntime" gave. The boolean Stop logging and ZIP is a button that the user can use to stop the logging manually. The logic is shown in Figure 79.

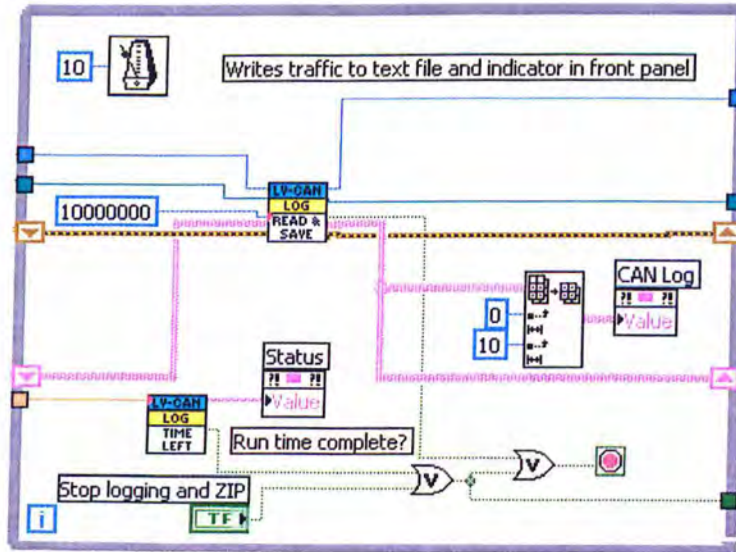



*Figure 79: Log traffic block diagram*

The "save and read" subVI, shown in Figure 80, reads the bus and saves the information in a text file. Before saving it to a text file Log → String formats the CAN messages to strings, by removing some EOL and tabs. The subVI sends the CAN messages to a table to show the traffic to the user.

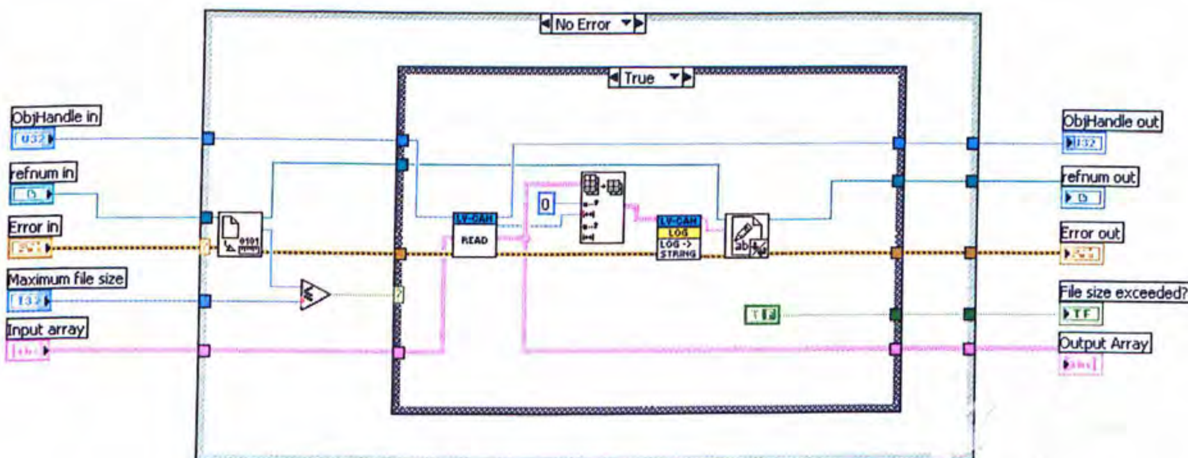



*Figure 80: Save and read block diagram*

### *Save log to ZIP*

When a log file exceeds 10 MB it is saved in a ZIP archive. This is done with the "Save log to Zip" VI shown in Figure 81. It closes the file and saves it to the ZIP. It then removes the temporary text file.

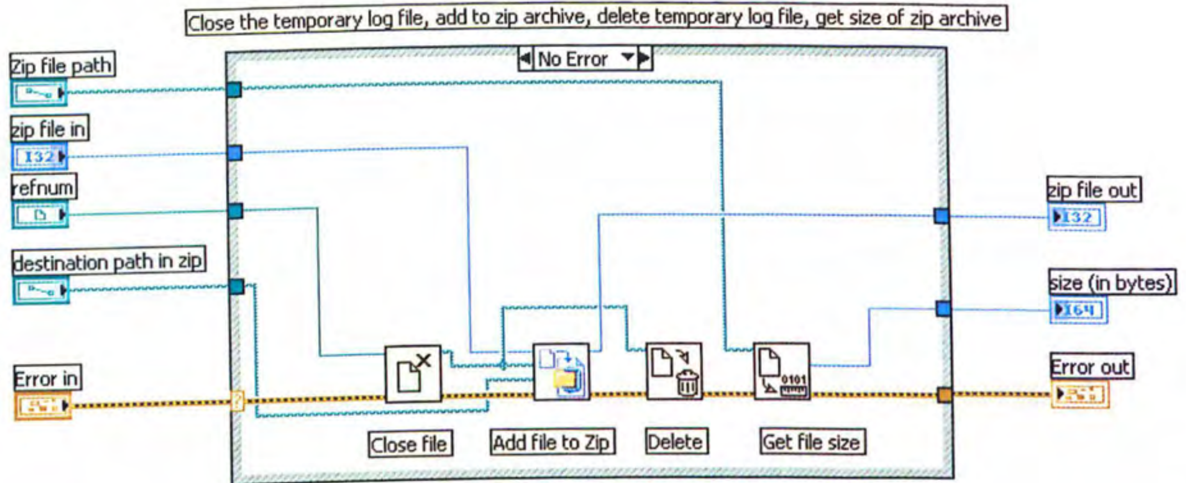



*Figure 81: Save log to ZIP block diagram*

## 4.6 Send Frame

The tab Send frame, shown in Figure 82, makes it possible to send a manually composed message and at the same time observe the traffic on the bus through a table. The programming to get it to work is quite straightforward. It is based on a program from National Instruments that came with the drivers for NI-CAN USB-8473s. It can be found through NI Exampel Finder, Hardware Input and Output → CAN → Frame API → Basic → Transmit Receive same Port.

### 4.6.1 Front Panel

When pressing the OK button for Start Transceiver Mode the traffic will be shown in the table and the user will be able to send a manual message. The user will need to specify the Arbitration Id and the data to write and then push the write button.

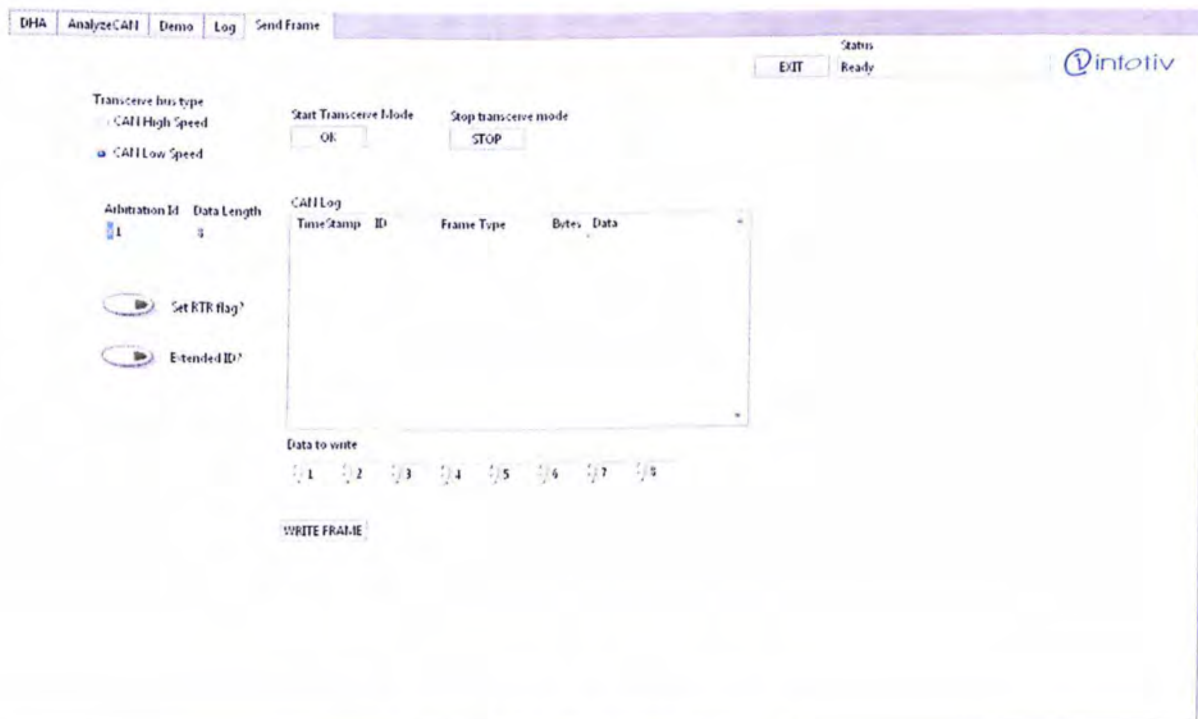



*Figure 82: Send frame front panel*

The flowchart for the Send Frame is trivial, but the part that will be further explained is Write message. A flow chart over the functionality in Write message is illustrated in Figure 83.

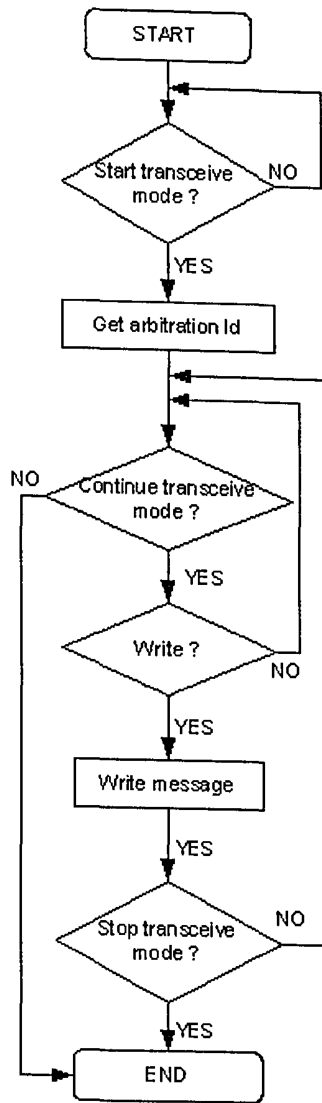



*Figure 83: Send frame flowchart*

### *4.6.2 Vital Sequences*

In this tab the write message will explained further, it is a basic function that is used in a similar way when a message need to be sent on the bus.

#### *Write message*

The write message is made mainly of two subVI, "Read" and "Send Frame", see Figure 84. The button "Write frame" boolean enables the "Send frame" to get the information it needs to send a frame. After the frame has been sent the traffic is then read with "Read" which sends the traffic to the table "CAN log".
The "Write message" runs until an error occurs or the user presses the "STOP" button.

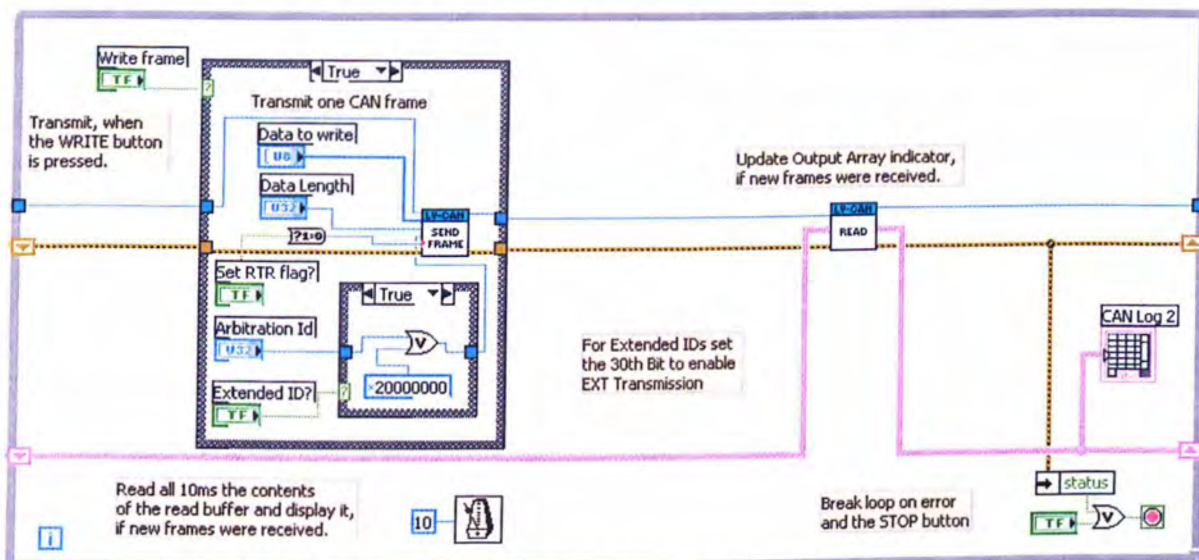



*Figure 84: Write message flowchart*

# 5 Conclusions

LabVIEW has shown to have both strengths and weaknesses as a programming tool for our work. It is very easy to create a satisfying graphical user interface but it is not as intuitive when handling information and more complicated calculations compared to other programming languages. For this thesis it has clearly more advantages than disadvantages. As with other programming languages it is important to follow the programming rules and conventions. Otherwise it will be almost impossible for other programmers to use the code for other projects. The conventions are in this case somewhat different because of its graphical structure of the language.

The NI 8473-USB has worked flawlessly and overall impressive with its good programming examples and easy to use SubVI:s and is highly recommended by the authors of this thesis. It has shown that a good CAN-interface can be made with LabVIEW and we foresee that it is possible to replace the VCT with a NI 8473-USB in many cases.

The GUI that has been created clearly has capabilities to replace and complement some of the tools that Infotiv are using at the moment. It has shown some of the capabilities that LabVIEW combined with CAN can achieve.

A continuation on the thesis would benefit with a implementation of the diagnostic protocol GGD, it would increase the life time of the program as the D2 protocol is not implemented in newer cars. To be able to interpret the response of node when sending a custom made message would also increase the usability.

# References

[1] National Instruments. (2006). *NI-CAN Hardware and Software Manual* [www]
http://www.ni.com/pdf/manuals/370289k.pdf
Received 2008-07-28

[2] Bishop, R.H., (2001). *Learning with labVIEW 6i.*New Jersey: Prentice Hall.

[3] Philips. (2000). *Datasheet SJA1000 Stand-alone CAN controller* [www]
http://www.nxp.com/acrobat_download/datasheets/SJA1000_3.pdf
Received 2008-07-28

[4] Philips. (2007). *Product data sheet TJA1041 High speed CAN controller* [www]
http://www.nxp.com/acrobat_download/datasheets/TJA1041_6.pdf)
Received 2008-07-28

[5] Encyclopædia Britannica. (2008). *"ASCII." Encyclopædia Britannica Online* [www]
http://search.eb.com.proxy.lib.chalmers.se/eb/article-9001602
Received 2008-09-08

[6] Nordlund, Wiklund, (2003). *Digitalteknik.* Partille: Natura Läromedel

[7] Volvo Car Corporation. Document: Diagnostic II, Part 1.2

[8] Givargis, Vahid, (2002). *Embedded System Design.*John Wiley & Sons, Inc