TESI DI LAUREA

OBD-II access using a CAN to USB converter from Kvaser

Laureando: **ALBERTO GUIOTTO**

Relatore: **STEFANO VITTURI**

Dipartimento di Ingegneria dell’Informazione

Università degli Studi di Padova

Relatore: **JUAN MANUEL MORENO EGUILAZ**

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

Universitat Politècnica de Catalunya

Corso di laurea Magistrale in Ingegneria Elettronica

Anno Accademico: 2012 / 2013

..A chi ha permesso che raggiungessi questa meta.

..A chi mi ha dato tutto.

..Sempre gliene sarò grato.

RESUME

This project aims to develop an application for PC that allows to connect, analyze and show data present on the OBD (On Board Diagnostic) CAN bus based system of any vehicle built since 2008. The final goal is to run this application connected with a real vehicle, equipped with a concrete OBD-II system.

It has been used an emulator from OZEN ELEKTRONIK to simulate the OBD system, combined with an adapter cable from KVASER and a USB – CAN converter from the same brand. The programming language chosen to develop the application was Visual Basic .NET.

In this thesis there is an initial introduction, with the objective and the goal of the project. After that, the history of the OBD and the state of art are explained; this chapter in particular wants to show the actual market supply, listing some examples of products already existent with a short description for each one. The text continues with the characterization of the instrumentation utilized and the specifications followed. Furthermore, there is an illustration about the structure of the application and the forms that constitute the latter. The results capitol speaks about the CAN messages exchanged between the device and the PC, with the explanation of the meaning of each byte of these messages. Finally, there is a discussion about economic costs and the environmental impact of the project. In the conclusions, a set of no-conformity of the device is listed, with the solutions adopted to solve each problem.

INDEX

GLOSSARY

OBD	On Board Diagnostic
PID	Parameter ID
DTC	Diagnostic Trouble Code
SAE	Society of Automotive Engineers
CAN	Controller Area Network
SDK	Software Development Kit
ECU	Electronic Control Unit

1. INTRODUCTION

1.1. Objective

This project is aimed at building a software application which allows to connect a PC to a OBD-II system of a car, in order to access the diagnostic information. The software follows the standard SAE J1979, which specifies the rules of the communication between OBD-II system and the diagnostic device. The bus for which the application is developed is the CAN bus. This is the mandatory bus for the OBD-II system of a car built after 2008.

The connection between PC and the CAN bus of the OBD-II system is achieved by using a CAN to USB converter, such as Kvaser Leaf Light; the behavior of the OBD-II system is emulated through the OZEN ELEKTRONIK MobyDic 1610 board.

The computer language utilized to build the application is Visual Basic .NET, used with the help of the SDK Microsoft Visual Studio 2010 Professional. The driver that allows the communication between PC and Kvaser hardware is the CANLib, totally free and downloadable from the Kvaser site.

The project is compared with the already present offer of OBD applications for PC, more specifically with those compatible with the Kvaser converters. In chapter 2, there is a short list of these software, divided into Kvaser compatible and non-compatible.

1.2. Goal

To reach these objectives, the steps expected are:

- Study of the provisions over the OBD-II system
- Study of the functioning of the instrumentation
- Study of the chosen computer language
- Study of the Kvaser driver
- Development of the application
- Verification and testing of the application
- Writing of the thesis

The final step should be the use of this software with a real vehicle, equipped with an authentic OBD system.

2. STATE OF ART

OBD-II (On Board Diagnostic) is a term that refers to the capability of a technician or a vehicle owner to have access to the information about the state of health of the several sub systems in a car. The amount of information available has varied a lot since the introduction of this system, at the same time of the computer development.

The first OBD systems only indicated a malfunction through a MIL (Malfunction Indicator Light), without giving any other more specific information about it [1]. Nowadays a technician could know the nature of the problem, through the DTC information (Diagnostic Trouble Code), which is an encoded key that corresponds to a specific problem in a given sub-system.

In 1969 Volkswagen introduced the first on board system on its fuel-injected Type 3 models. Afterwards, due to the necessity of a real-time tuning on the sports car, the OBD seeped out, even if without any standardization.

In 1980 General Motor launched the proprietary ALDL (Assembly Line Diagnostic Link), which used the PWM (Pulse Width Modulation) and it was the first system that provided information about the nature of the problem.

In 1988 SAE (Society of Automotive Engineers) started to recommend a standardized connector and a set of diagnostic tests. The OBD began to become a widely used instrument for diagnostic purposes.

Between 1991 and 2004 the system became mandatory for all the cars built in those years, first of all in the California state (thanks to CARB, California Air Resources Board), later in all the United States and finally in Europe.

The final step was, in 2008, with the introduction of the CAN bus (Controller Area Network) as standard for communication between diagnostic instruments and cars.

Here it is explained the state of art, which is the present situation about the applications available in the market. There are many programs: the first distinction is made between programs compatible with Kvaser and programs not compatible. Among these, there are programs built with a free license, programs owned by car manufacturer and other written by specialized companies.

2.1. Kvaser compatible

PCMSCAN (Palmer Performance Engineering)

http://www.palmerperformance.com/products/pcmscan/index.php

PCMSCAN is a fully featured generic OBD-II scanner and diagnostic tool that supports a wide variety of OBD-II hardware interfaces. It allows viewing, charting, logging and playback of diagnostic data in real time via the vehicle's OBD-II diagnostic data port. It also allows viewing of vehicle Diagnostic Trouble Codes (DTC's), Freeze Frame data, and other vehicle information.

Compatible hardware:

- Autotap AT1, AT2, AT3, AT123 (v2.x)
- VIA LDV100, LDV200, LDV300, LDV123
- Multiplex Engineering T16
- ELM320, ELM322, ELM323 (v2.x)
- ELM327 (all versions)
- Any SAE-J2534 compliant OBD-II interface.

VEHICLE NETWORK TOOLBOX (MathWorks)

http://www.pr.com/press-release/226214

http://www.mathworks.it/products/vehicle-network/

Vehicle Network Toolbox™ provides connectivity to CAN devices from MATLAB® and Simulink®. The toolbox provides functions for encoding, decoding, and filtering CAN messages, enabling you to transfer messages between the MATLAB workspace and a CAN bus.

Compatible hardware:

- Kvaser
- National Instruments
- Vector

ScanMaster-ScanTool (WGSoft)

http://www.wgsoft.de/en/menu-scanmaster-passthru-links.html

ScanMaster-PassThru is an OBD-II/EOBD diagnostics for vehicle diagnostics in a legal extent required under OBD-II/EOBD standards that were developed specifically for the J2534 PassThru interface and supports all 10 defined in SAE J1979 OBD-II diagnostic modes $ 01 - $ 0A and communication protocols.

Compatible hardware:

- BOSCH
- Actia
- ETAS
- EASE Diagnostics
- Blue Streak Electronics Inc.
- Dearborn Group
- Kvaser
- Diagnostic-Associates

Module Analyzer (Influx Technologies)

http://www.influxtechnology.com/moduleanalyser.html

In the basic OBD mode this enables:

- EOBD/OBDII data retrieval and analysis and single click reports.

- Live monitoring of emissions parameters. This includes data such as engine speed, vehicle speed, engine temperature, manifold pressure, intake air temperature etc...

- Monitor and clear emissions trouble codes (DTC's)

- Read vehicle information

Compatible hardware:

- All Kvaser CAN or LIN hardware interface devices.

- CANdo CAN interface device.

DiagRA D (EAN Technologies)

http://www.eantechnologies.com/index.php?_m=mod_product&_a=view&p_id=158

DiagRA D is the diagnostic option from the DiagRA MCD Toolset for the selection of diagnostic data from vehicle control units. The functional range can be subdivided into three basic sections:

1. Workshop diagnostics

2. Scan-Tool for OBDII/EOBD/HD-OBD diagnostics

3. Advanced developer functions

The SAE J1979 scan tool function supports all 10 services (Service $01-Service $0A) defined by the authorities as well as all the sub-functions (PIDs).

Compatible hardware:

- I+ME Actia XS family

- Vector CANcard X/XL, and CANcase XL, ETAS CAN-link I/II

- Kvaser CAN adapter series

- PassThru devices according to SAE J2534 (v0202 and v0404)

- Devices according to RP1210 API for SAE J1939 protocol

- Support of interfaces with D-PDU-API after ISO 22900-2
- Siemens BlueVCI
- IXXAT CAN interface devices

Silver Scan-Tool (EAN Technologies)

http://www.eantechnologies.com/index.php?_m=mod_product&_a=view&p_id=161

Silver Scan-Tool™ software provides test functionality for onboard diagnostics according to SAE J1979, SAE J1939 and ISO 27145. Support for all 10 services (to A) defined as well as all subfunctions (PID´s).

Support of SAE J2534 and RP1210 API.

2.2. Generic

Fiat EcuScan (Fiat)

http://www.fiatecuscan.net/

FiatECUScan is a diagnostic software for Fiat, Alfa and Lancia vehicles. It allows you to perform various diagnostics tasks on the supported vehicles/modules.

Compatible hardware:

- KL (also known as VagCom 409)
- ELM327 (1.3 or newer)
- OBDKey 1.40
- OBDLink, ELM Scan 5
- CANtieCAR
- Bluetooth OBDKey
- ELM 327 and OBDLink interfaces are fully supported but not recommended for special functions (like PROXI Alignment, remote control programming, IMA coding, etc.)

ScanXL Standard (ScanTool.net)

http://www.scantool.net/software/scanxl-std.html

ScanXL is the most advanced diagnostic software package that works with ScanTool.net scan tools. Almost every part of the software is customizable and configurable: from the color, scale, and line width of the graphs, to the look of the gauges, to the built-in scripting support. Ford and GM add-ons (sold separately) provide access to thousands of enhanced parameters not available through generic OBD.

Compatible Hardware:

- OBDLink
- OBDLink CI
- ElmScan 5
- Ready-or-Not

Dash Command (ScanTool.net)

http://www.scantool.net/software/dashcommand.html

DashCommand harnesses the power of the patent-pending DashXL™ technology to deliver high quality dashboards on a screen of any size. The dashboards are fully customizable with both digital and analog gauges, and scale to fit any screen size from 320x240 to 1600x1200 and larger. Over 229 OBD-II PIDs are supported.

Compatible Hardware:

- OBDLink
- OBDLink CI
- ElmScan 5
- Ready-or-Not

obd2crazy.com

http://www.obd2crazy.com/software.html

Diagnostics

- View results of continuous/non-continuous tests
- Read and reset trouble codes/dash indicator Check Engine (MIL) lamp
- Flexible Diagnostic Trouble Code system, import, customize, and export trouble codes

EasyObdII Version

http://easyobdii.com/index.php?act=viewProd&productId=15

EasyObdII.com produces software for ScanTool.net OBD-II (OBD2) Elmscan Interfaces using the ELM 3xx Chipset. EasyObdII software is very easy to use making it ideal for vehicle workshops and individual vehicle repairers.

Compatible Hardware:

Elmscan Interfaces using the ELM 3xx Chipset

OBD 2007 (GLM Software)

http://www.glmsoftware.com/Features.aspx

OBD 2007 is a family of OBD II software applications designed to assist automotive professionals and enthusiasts alike in diagnosing and analyzing problems with modern vehicle engines. Presently there are two versions of OBD 2007, one for PCs, laptops and Windows Vista UMPCs and another for Pocket PCs.

OBD 2007 supports all services $01 through $09 as specified by SAE J1979 and ISO 15031-5.

ProScan (My Scan Tool)

http://www.myscantool.com/details.php

With the new Fuel Economy Analysis tool, is able to determine a vehicle's average fuel economy during any trip. It's real-time display allows you to monitor instantaneous fuel consumption rates which can help you learn to drive more efficiently, thus saving money on fuel. It is also possible specify the current price of fuel and learn exactly how much each trip costs the driver.

TouchScan (OCTech)

http://www.obdsoftware.net/TouchScanInfo.aspx#

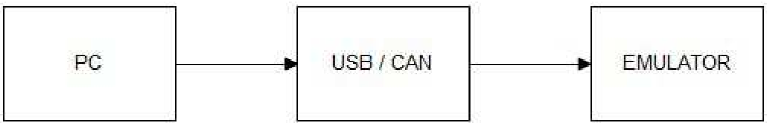
TouchScan is an easy-to-use yet powerful software package for monitoring vehicle data and diagnosing problems in modern vehicles. TouchScan works great with laptops, desktop PC's, touchscreens and car PC's.

Compatible hardware:

- OBDLink - ScanTool.net
- OBDLink SX - ScanTool.net
- OBDLink MX - ScanTool.net
- OBDLink S - ScanTool.net
- ElmScan 5 Compact - ScanTool.net
- Ready-or-Not - ScanTool.net
- All-In-One - OBD Diagnostics, Inc.

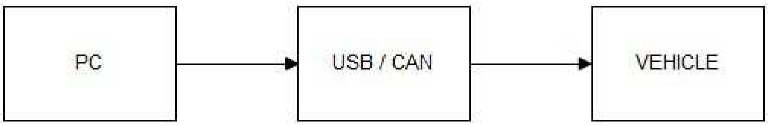
3. INSTRUMENTATION & SPECIFICATIONS

The basic idea of the project is resumed in this schema:



There is a PC, that runs the application. This application generates messages, which are provided at the USB port of the device. Subsequently, they are processed by the USB / CAN converter, to obtain standardized messages for the CAN bus. The latters are received from the simulator, which provides to process them and generates the correspondent responses.

The final aim is slightly different: the emulator should be replaced by a real vehicle, furnished with an actual OBD-II system:



In this case, the application, the instrumentation and the messages traffic are the same; what changes is only the final section, intended as processing unit. In the conclusions section, it will be explained how this change will be not so easy to do.

The software must follow the standard SAE J1979: it allows the communication through several protocols [1]:

- SAE 1850 PWM (Pulse Width Modulation), 41.6 kB/sec, standard of the Ford Motor Company
- SAE J1850 VPW (Variable Pulse Width), 10.4/41.6 kB/sec, standard of General Motors
- ISO 9141-2, 10.4 kBaud, similar to RS-232
- ISO 14230 KWP2000 (Keyword Protocol 2000)
- ISO 15765 CAN (Controller Area Network), 250 kBit/s or 500 kBit/s

The last, the CAN bus, has 2 principal characteristics when is utilized in this area of interest: the baud rate and the number of bits of the identifier of the message; the baud rate, in automotive applications, could be 250 kBit/s or 500 kBit/s, while the identifier could be made of 11 bit or 29 bit. Furthermore, a CAN message is made up of several parts:

- Start-of-frame
- Identifier
- Remote transmission request (RTR)
- Identifier extension bit (IDE)
- Reserved bit
- Data length code (DLC)
- Data field
- CRC
- CRC delimiter
- ACK slot
- ACK delimiter
- End-of-frame (EOF)

The PC / OBD – II connection is permitted by the Kvaser Leaf Light (Fig. 1.1): it is an instrument that turns a USB message in a CAN one, and vice versa. [2] The baud rate of the communication could be both 250 kBit/s and 500 kBit/s, as required from the standard. In the next chapters will be explained how the software selects the correct baud rate and identifier.

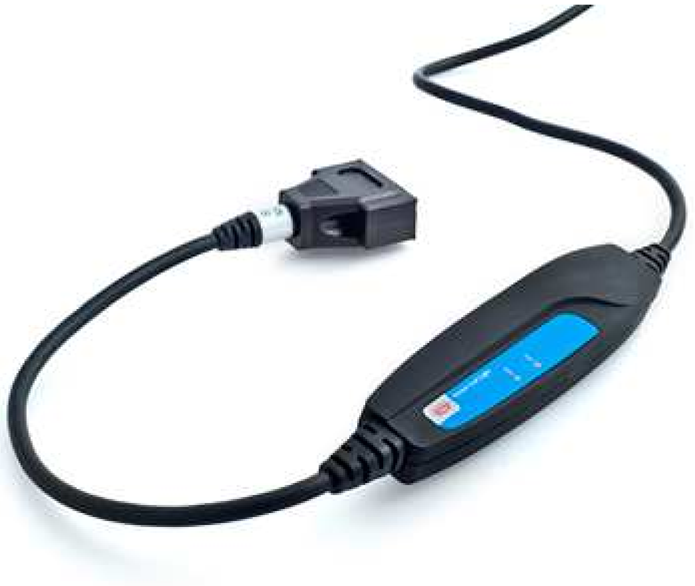


Fig 3.1: Kvaser Leaf Light

As we can see in the Fig. 3.1, the connectors cable are the standard CAN connector and the USB connector.

Kvaser provides a driver with which a user can send and receive CAN messages through the Kvaser Leaf Light. The CANlib library, free downloadable from the Kvaser site, allows to use several methods to initialize the hardware, put the bus on or off, set the bus parameters, write or read one or more messages or close the connection. The CANlib works with a SDK which permits to develop software for the Kvaser hardware. It supports the following compilers:

- Microsoft Visual C/C++
- Borland/CodeGear/Embarcadero C++ Builder
- Gcc, MinGW
- Borland/CodeGear/Embarcadero Delphi (all versions)
- Microsoft Visual Basic and VB.NET
- Microsoft C#
- Also various examples for managed C++ code, Python, etc.

It has been chosen to build the program with VB.NET: this programming language offers an efficient IDE (Integrated Development Environment), the Microsoft Visual Basic Studio 2010; furthermore it offers a rapid and easy to use graphic library, which permits to save time.

The OBD-II system is emulated by Mobydic 1610 provided of the OE91C1610 CAN bus ECU simulator from OZEN ELEKTRONIK; it reproduces the behavior of three ECUs (Electronic Central Units): they are ECM (Engine Control Module), TCM (Transmission Control Module) and ABS (ABS control module). Usually the maximum number of ECU in an OBD-II is 8 and each one supports the 9 operation modes of an OBD-II. In this device, there are only these 3 ECUs and each one supports only some modes. In any case, the standard followed is always the SAE J1979.

There are two switches on the emulator board, to set the baud rate and the number of bit of the identifier of the CAN message (11 or 29 bit). Each time the switch changes, the device must be turned off and re-start. There are 5 potentiometers, with which the user can regulate 5 different PIDs (Parameter ID), which are the “information” about the car, available and supported from the OBD-II system.

The OZEN ELEKTRONIK device communicates through a special standardized port, the female 16-pin (2x8) J1962 connector (showed in Fig. 3.2)

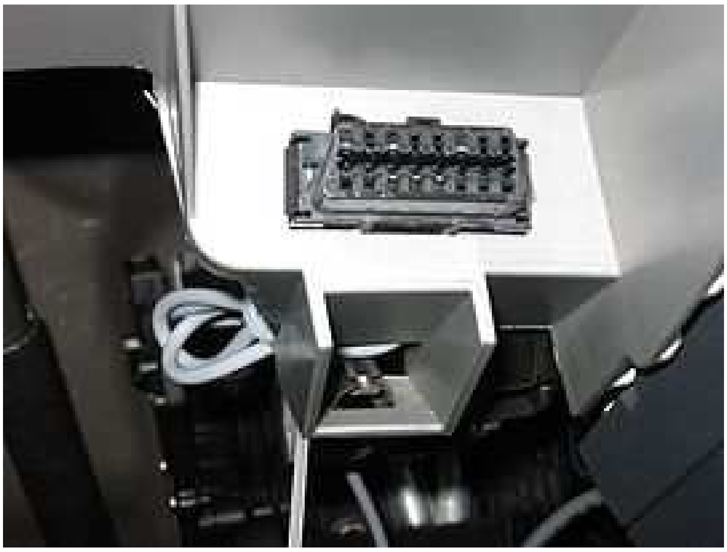


Fig. 3.2: Female 16-pin (2x8) J1962 connector

An adapter cable from Kvaser [3] enables to connect the J1962 port to the CAN port of the Kvaser Leaf Light: it has a J1962 male connector in one side, and a female CAN in the other (Fig. 3.3).

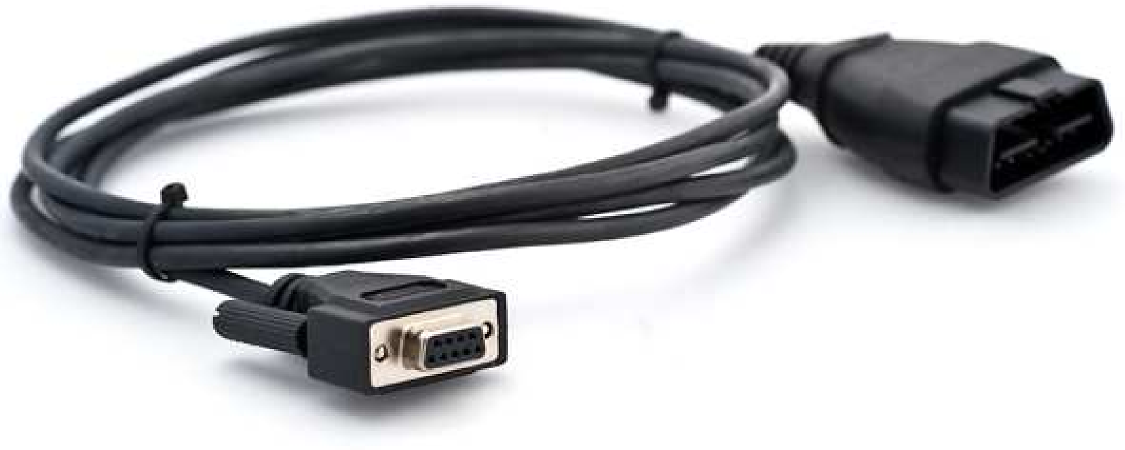


Fig. 3.3: Adapter cable from Kvaser

The developed application is multi – language; when the program is launched, a routine controls the standard language of the OS and depending on it, there are three possibilities: English (the default language), Italian or Spanish. The messages and the text are changed after this control.

Furthermore, the application is endowed with a Help: when the user presses the F1 button, or selects the Help from the menu, a guide appears and shows the principal command and the meaning of the supported modes.

4. APPLICATION STRUCTURE

In this chapter we are going to describe the structure of the program, with an overview of the different forms which compound the application, and the function that every form has.

There is a main screen, which contains the buttons for each available mode. There are 9 modes, for each one, a sub chapter will be dedicated, in which we will explain the purpose and the correspondent screen.

4.1. Main screen

First of all we are going to explain the different buttons that are shown on this screen; subsequently we will describe each method of which the form is composed.

On the right side there are all the different buttons: the first one is the *Initialize Button*. This method manages to create a bus handle for the communication and sets the correct baud rate and identifier of the CAN message. The other buttons are those for each mode: from these buttons the user can have access to each screen of each mode. For all these buttons, when the user puts the mouse over there, appears a description of what the mode serves. In the bottom of the screen there is a status bar, which indicates the information about the hardware: before the initialization it suggests to the user to initialize the hardware, and next of this action, it shows the number of the bus channel, the baud rate and the type of identifier. In the top of the screen, instead, there are three menus: the first allows to initialize the hardware, the second collects all the several modes, the third has two links, for the *About* screen and for the *Help* Guide. The latter is accessible by means of the keyboard, with the F1 key.

After introducing the composition of this screen, we will talk about the several methods that compound the form. Here there is a list of these methods, with each explanation.

4.1.1. Load Form

When the program is launched, it provides to check if the *Config.ini* file exists. An *.ini file is a text file that is used from an application to store the information about the configuration. Because the user can save the data received from the Mobydic device (through the hard disk of the pc or through an email), this file *.ini contains all the information about the email accounts of the sender and the receiver. So, the program needs to know:

- Sender Mail
- Sender Name
- Receiver Mail
- Server Name
- Server Port
- Username
- Password

At the first launch of the program, after the splash screen of presentation, a form to fill appears. When the user has filled it, the routine provides to create the Config.ini file. Obviously, this action is performed only once, if the Config.ini file doesn't exist.

4.1.2. Form Closing

In this form the only action performed is to ask a confirmation to exit the application.

4.1.3. Initialize Click

This method lets to initialize the hardware, creating a bus handle and setting the correct baud rate and identifier for the communication.

The first action performed is to use the *canlibCLSNET.Canlib.canInitializeLibrary()*. Like the name suggest, it will initialize the driver. Subsequently, the bus will be created through the *canlibCLSNET.Canlib.canOpenChannel(0, 0)*: this method accepts two parameters (number of channel and a flag) and returns a handle; the user can use this handle to refer at the correspondent bus. Normally this number is 0, which means that the operation was performed. In this case the next method to use is *canlibCLSNET.Canlib.canBusOn(hnd0)*, where hnd0 is the handle, which puts the bus on. Conversely, if the number is less than zero, it means that the hardware is not connected or there are other problems with the connection. In this case the program will launch a window with an error message.

After that, the program tries to set the correct baud rate and identifier. The standard imposes how to do that. It can be summarized in some outline (fig. 4.1 and 4.2) [4]:

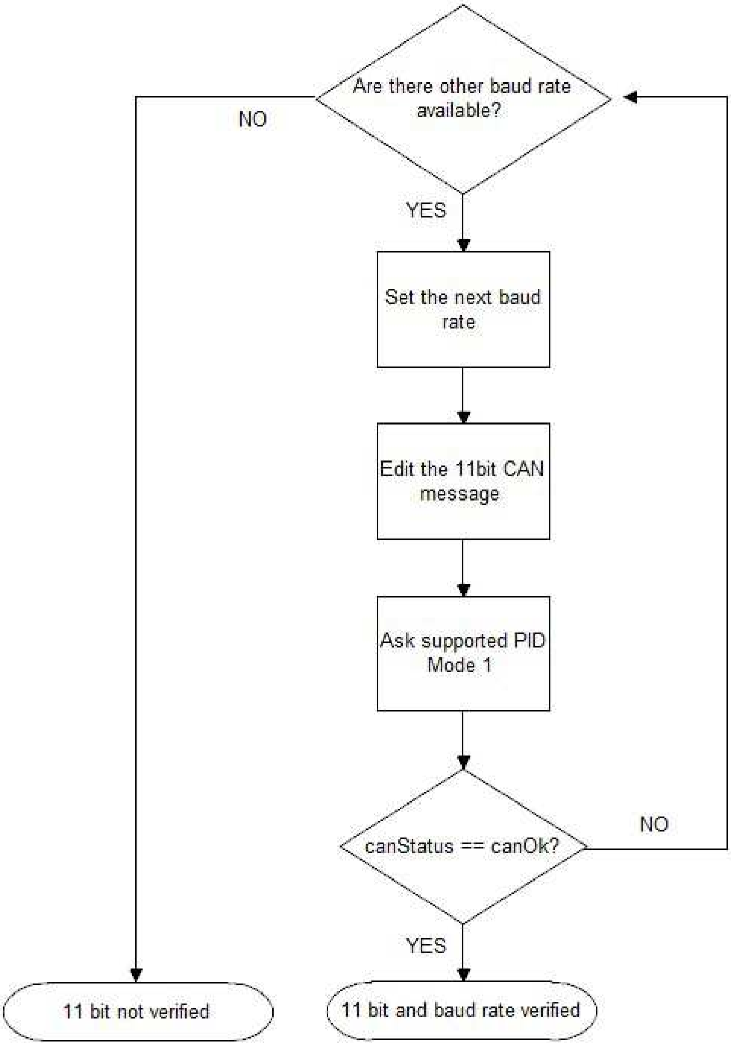


Fig. 4.1: Outline for 11 bit identifier verification

The application sets one of the available baud rates through the *canlibCLSNET.Canlib.canSetBusParams(hnd0, i, 0, 0, 0, 0, 0)* method, where hnd0 is the handle, i is the baud rate, and the subsequent are secondary parameters. After that, it builds the CAN message, with an 11 bit identifier. This message serves to ask to the different ECUs what PIDs they support and build a collection for remember it. The structure of the message will be explained in the next chapter, whit the Results. After the message has been edited, the application sends it to the OBD-II system.

The program verifies if the action was performed correctly evaluating the returned value, the *canlibCLSNET.Canlib.canStatus*: this is an Integer and can assume many values, depending on if an error occurs. If the operation goes right (*canStatus* = 0), it means that both of the baud rate and the identifier are correct. On the contrary, the application looks if there is any other available baud rate. If yes, repeats the same operation of before; if not, that means that the device doesn’t work with an 11 bit identifier. So the program passes to evaluate if the system works with a 29 bit identifier, like it is showed in the next Fig. 4.2 [4]:

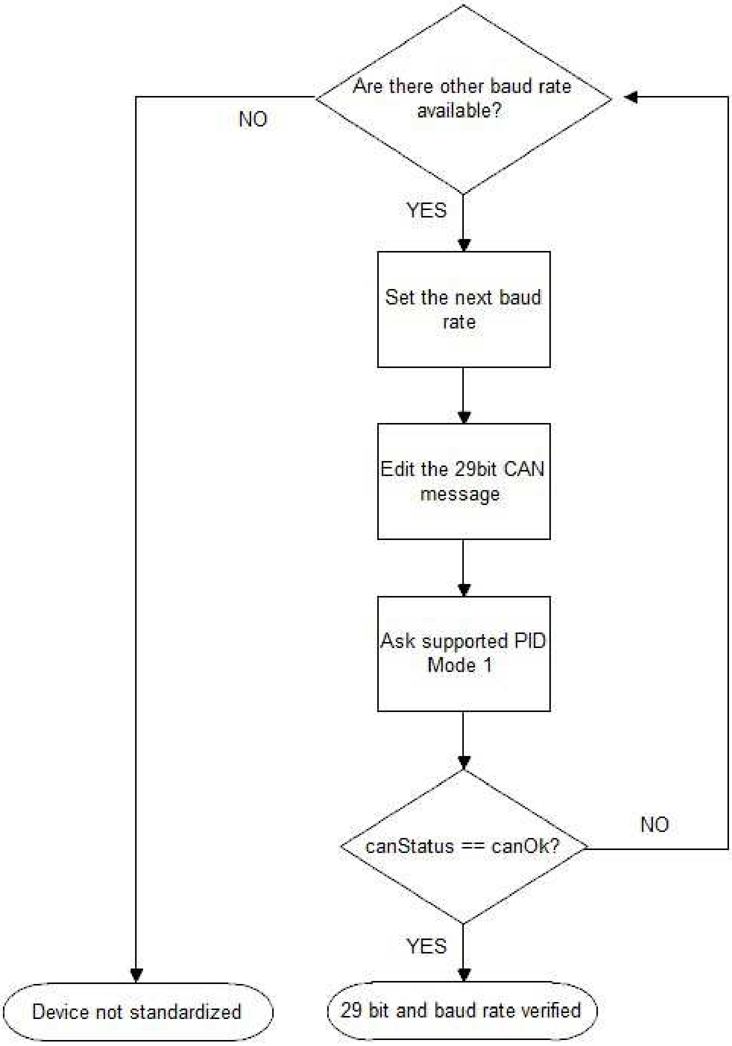


Fig. 4.2: Outline for the 29 bit identifier verification

The verification procedure is the same, exception for the number of bit of the identifier. If with one of the available baud rate the verification goes well, that means both the baud rate and the identifier are correct. If not, the program deduces that the device doesn’t follow the standardization. In this case, a message appears to let the user know that the device is not initialized. In every case, the status bar will inform the user about the status of the device.

4.2. Mode 1

The first mode [5] allows the user know a lot of information about the tested car. Like the description showed from the program “mode 1 is used to identify what powertrain information is available to the scan tool”; that means that each OBD-II system doesn’t support all the available PIDs. Furthermore, often many PIDs are customized from the manufacturer and they are hidden to the final user. With this form of the application the user can test which PIDs are supported, and then asks the data and show it to the user.

The screen is accessible clicking the “Mode 01” button, both from the main screen and the “Mode” menu. The mode 1 screen presents two textbox in which the user can know the baud rate and the identifier of the CAN connection. Under these, there are two radio buttons: with the first, putting the number of the desired PID in hexadecimal, the user can obtain the data about a single PID; with the second, the user can ask at the system all the supported PIDs. The results are showed in a textbox beneath. In the bottom of the screen, there are many buttons: with the “Glossary” the user can know the description of the several acronyms, showing a label. With “Save” and “Export” the user can file the results: with “Save” will be created a *.txt file saved in the hard disk, in the folder chosen by the user, while with “Export” will be sent an e-mail, from and to the addresses specified before. The last button clears the text box.

The two main methods used in this form are those which permit the test over PIDs. If the user chooses to request only one PID, the application check if there is a text like a number in the corresponding text box and after verifying if the PID is supported from the OBD-II system, seeks if it is stored in the collection built before. If yes, the result is showed, if not an error message will appear. A different error message will appear if the text in the textbox is not convertible to a number. If the user chooses to look all the available PIDs, the system asks and shows the results of all the PIDs stored in the collection (therefore supported from the system).

4.3. Mode 2

The mode 2 [5] allows the user know the Freeze Frame Data: in the same time when an error occurs, some PIDs are saved. When the user asks this data, can understand the situation at the moment in which the error appears.

The screen presents two buttons, one to ask the Freeze Frame Data, and one to clear the textbox. After there is a textbox to show the results, and when the user passes the pointer over the textbox, a label with the description of every single PID will appear. On the bottom, there are the buttons for file data.

When the form is loaded, the collection of the supported PIDs is built. This collection will be utilized after to request all the set of available data. Indeed, when the user asks the Freeze Frame Data, the program scans the collection and obtains the result from the Obd-II system. These results have sense only if there is a DTC (Diagnostic Trouble Code) that has caused them. In this case, the first data showed is the code of that DTC.

4.4. Mode 3 & Mode 7

Mode 3 [5] lists the emission-related confirmed diagnostic trouble codes stored. This means that when an error occurs, a Diagnostic Trouble Code is generated and stored in the pertinent ECU. This mode allows to obtain all the codes, for the different ECUs.

On the other hand, mode 7 is a request for emission-related diagnostic trouble codes detected during current or last completed driving cycle. The difference between the two modes is that mode 3 detects all the DTC occurred, while mode 7 is used to detect only the DTC occurred in the last driving cycle. This allows the technician to repair the car and then verify if the DTC will occur in the next driving cycle.

The screens of these two modes are identical, and are very similar to that of mode 2. There are two buttons, one for request the PIDs and one to clear the textbox, which stays under these buttons. In the bottom, the usual buttons to file the data.

When there aren’t any message, the textbox shows a message for each ECU, where is specified the name or the address of this ECU, and the number of DTC (zero, in this case). If there is any error, a method will “read” the correspondent code and shows it, after the name or addresses of the ECU and the number of DTC in that ECU.

4.5. Mode 4

Mode 4 [5] is used to clear emission-related diagnostic information, which means that if there is any DTC stored, this mode will clear it. The “cleaning” is made for each ECU.

If there is not any DTC stored, the program will show a message about it. If there is, the application try to clean it and if the operation goes right, a message appears and demonstrates for each ECU the cleaning of the DTC.

This mode has not a screen, because there is not any information available to show when the operation is executing.

4.6. Mode 9

Mode 9 [5] is used to retrieve vehicle information, i.e. data about the car like the VIN (Vehicle Identification Number).

The screen is identical to that of modes 2, 3 and 7. There is a label that explains the meaning of the available PIDs, when the user puts the mouse over the textbox.

As the similar mode explained before, when this mode is used, the form is loaded and a method provides to build the collection of available PIDs. When a request of information is made, the program analyzes this collection, requests the results for them, and finally provides to read the information sent by the OBD-II system, and shows it in the textbox.

4.7. Shared Components

There are shared methods which are used by several forms to develop their own tasks:

4.7.1. AskPID

This method is used to send to the OBD-II system the messages to ask the supported PIDs. To do it, there is a for loop, that allows to send the specific message and read the answer from the device. The method accepts 4 parameters: the number of bit of the identifier of the CAN message, the number of the mode (that of the OBD-II system), the maximum value (which will be explained in the next chapters) and the collection, which is an object that collect all the available PIDs for mode 1, 2 and 9.

4.7.2. PIDcollection

This method permits to build the collection of different PID available for each mode. With this collection the application can have a trail to remember which PIDs are supported. This method wants three parameters: the number of bits of the identifier of the CAN message, the data part of the CAN message, which contains the information about the supported PIDs, and the collection object, with the same meaning of before.

4.7.3. RequestPID

This is a basic method that allows to read and shows the different PIDs, requested by a mode. The necessary parameters are 3: the number of PID, the number of bits of the identifier and the mode of use. When a PID is requested, the application provides to build the correspondent CAN message to obtain that information. After that, the answer is read, and, with a select case structure, the information is decoded and showed to the user.

4.7.4. ReadDTC

With this method the program can read the DTC. All the DTC stored in the device are codes; each code corresponds to a specific problem. The several codes are codified in a standardized type and this method allows to decode them. The rules for this action will be explained subsequently. The two parameters are the two bytes that contain the data about each single code.

4.7.5. RecognizeID

This method, receiving an ECU ID like parameter, allows to know which ECU is sending a message and shows the correct name of this ECU, through a select case structure. If the address is not recognized, the hexadecimal number corresponding to the address will be showed.

4.7.6. Save

Often the user needs to save the information about a particular situation of his car. With this method, the program creates a *.txt file containing the information showed in the correspondent textbox and will save it in the hard disk, in the folder specified from the same user.

4.7.7. Export

When the user wants to send the information obtained from the system to a different user, this method permits to send an e-mail. The data about the account and the e-mail addresses are specified in a separated *.ini file, like explained before. When the e-mail is sent, a message will be showed to tell if the e-mail was sent in the correct way or if some errors occurred.

4.8. Other Modes

In an OBD-II system there are in general 9 modes [5]. Up to here, the modes supported by the OZEN ELEKTRONIK device were explained. But there are other modes, not supported and not implemented by the emulator, which are available in this context.

Mode 5 displays the oxygen sensor monitor screen and the test results gathered about the oxygen sensor.

Mode 6 is a request for on-board monitoring test results for continuously and non-continuously monitored system.

Mode 8 could enable the off-board test device to control the operation of an on-board system, test, or component.

5. Results

After the explanation about the purpose of each mode, in this chapter the results of this work will be explained, that is the most specific part of the thesis. For each screen, the program screen and the messages structures will be showed, with the help of a program, called CanKing, from Kvaser. This program monitors the message traffic on a CAN bus and shows it in a screen, from which the user can understand the meaning of each message. The texts and the messages are in Italian, because at the moments of the captures the application was launched in an operating system with Italian language.

5.1. Configuration

When the application is launched, after the splash screen presentation, it searches for the Config.ini file. In the case of it doesn't find that, a screen appears, where the user can put the data of the accounts of the sender and the receiver. The screen is as shown:

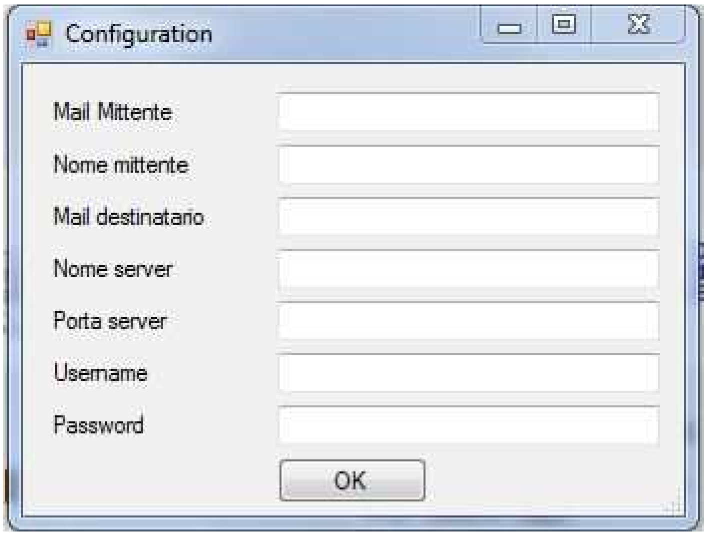
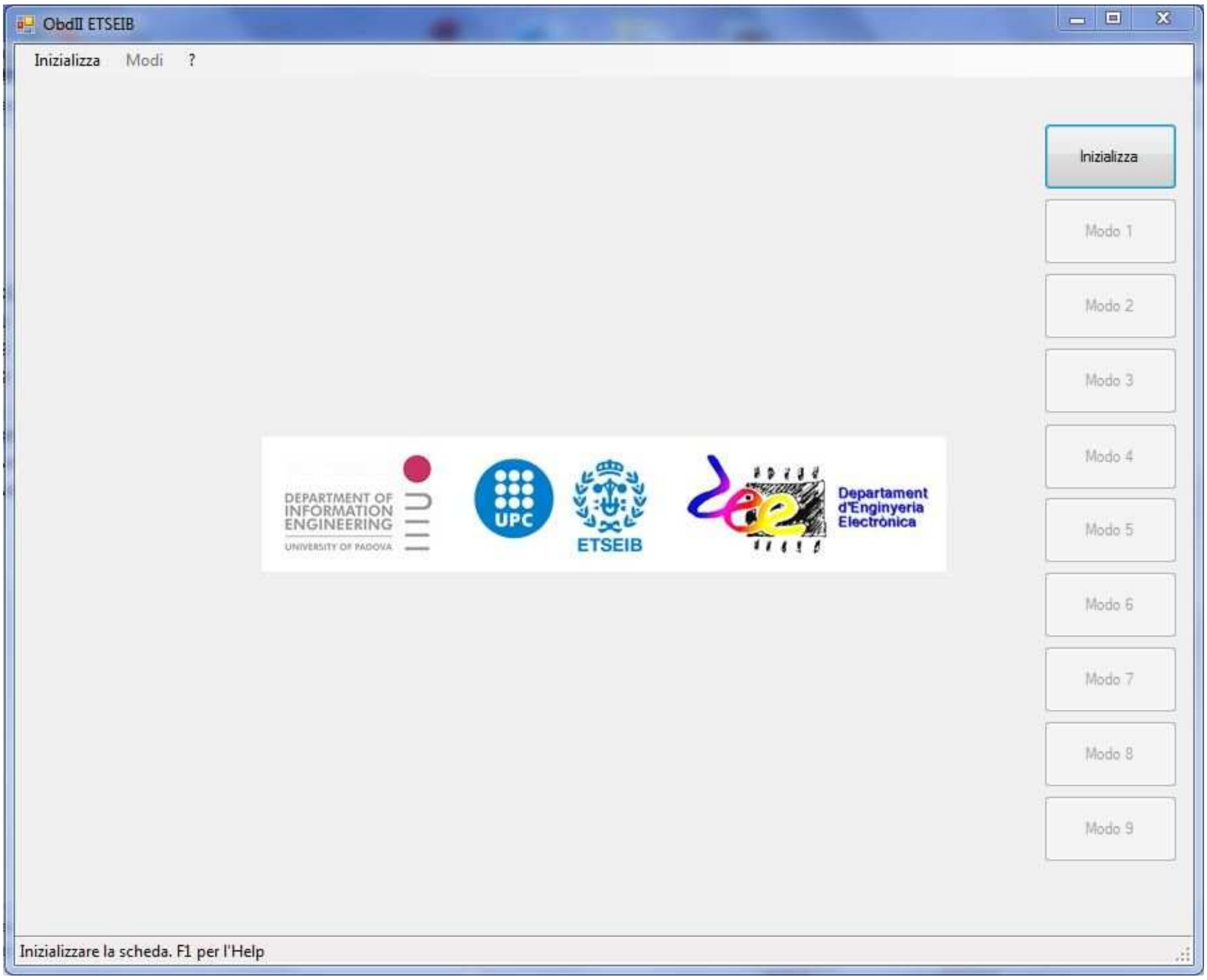


Fig. 5.1: Configuration form

5.2. Initialize

First of all we show the capture of the main form, comparing after the splash screen and eventually after the configuration form:

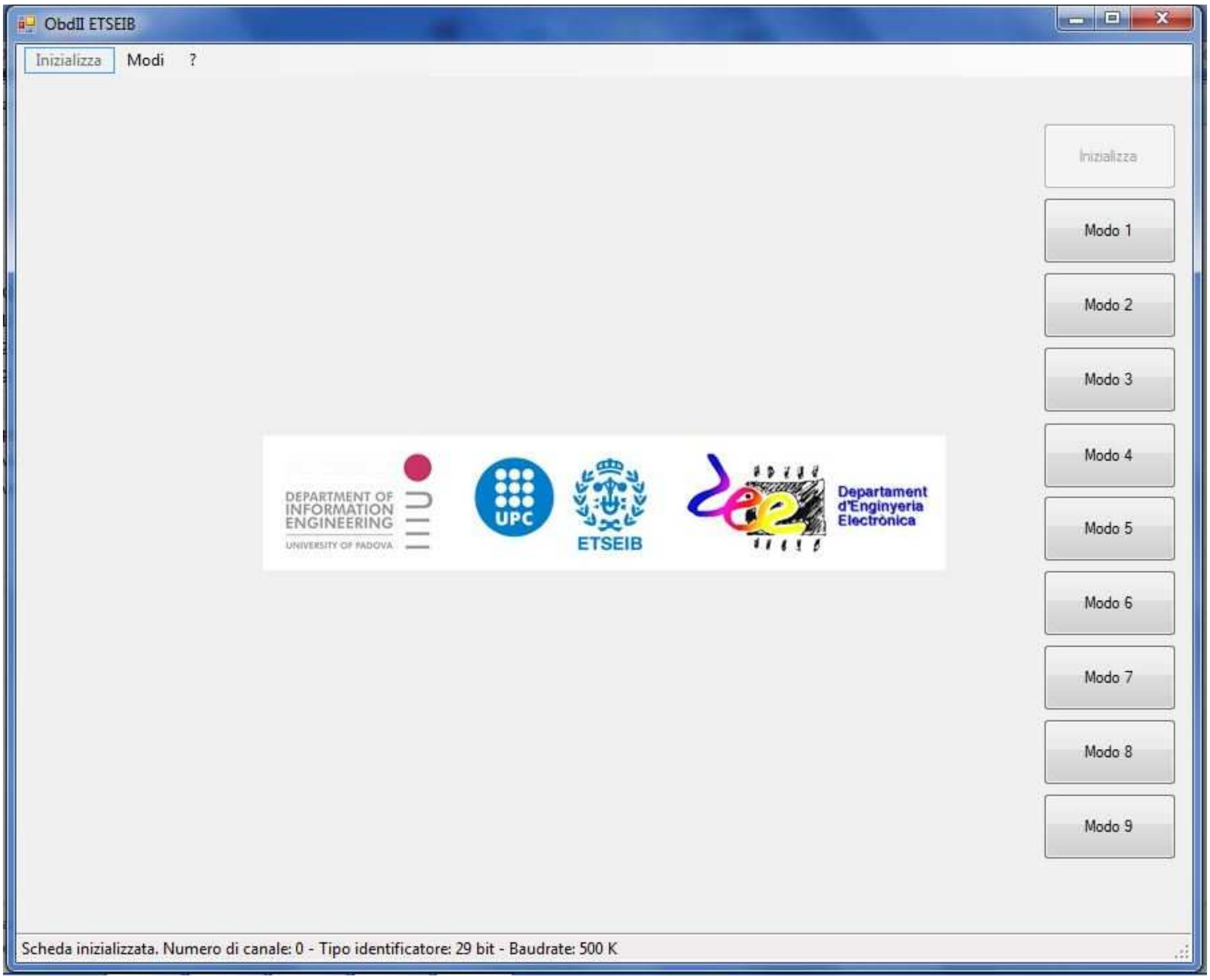


Inizializzare la scheda. F1 per l'Help

Fig. 5.2: Main form before the initialization

The form invites the user to initialize the device, both because is the unique available command on the mask, and because the status bar tells him to do that.

When this action is made, the form turns in this one:

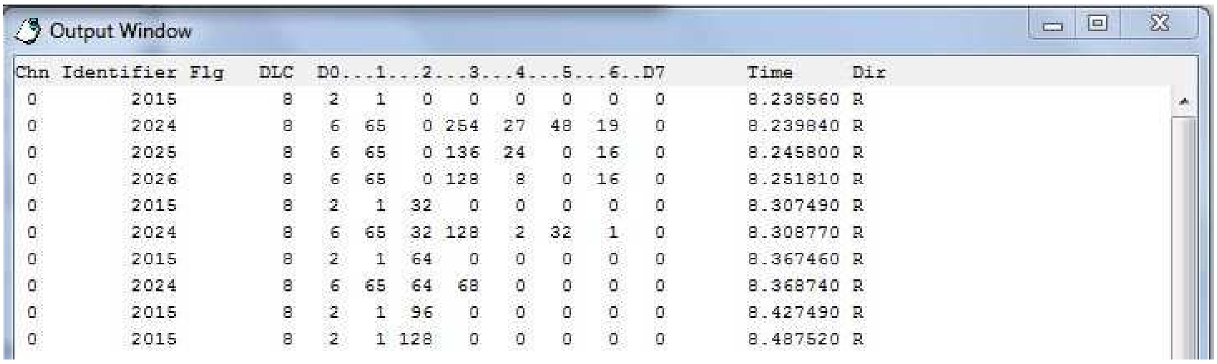


Scheda inizializzata. Numero di canale: 0 - Tipo identificatore: 29 bit - Baudrate: 500 K

Fig. 5.3: Main form after the initialization

Now the Initialize button is no more available, while all the modes buttons are available. The status bar on the bottom shows the configuration of the bus.

The configuration is made through a set of attempts of connection with the device, like explained before. After the bus is activated, in the same action, the application provides to ask at the device which PIDs of the mode 1 are available, through these messages:



```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7       Time      Dir
 0        2015         8   2   1   0   0   0   0   0   0       8.238560 R
 0        2024         8   6  65   0 254  27  48  19   0       8.239840 R
 0        2025         8   6  65   0 136  24   0  16   0       8.245800 R
 0        2026         8   6  65   0 128   8   0  16   0       8.251810 R
 0        2015         8   2   1  32   0   0   0   0   0       8.307490 R
 0        2024         8   6  65  32 128   2  32   1   0       8.308770 R
 0        2015         8   2   1  64   0   0   0   0   0       8.367460 R
 0        2024         8   6  65  64  68   0   0   0   0       8.368740 R
 0        2015         8   2   1  96   0   0   0   0   0       8.427490 R
 0        2015         8   2   1 128   0   0   0   0   0       8.487520 R
```

Fig. 5.4: CanKing Output Window for the Initialization

We use the occasion to explain the structure of the screen of the CanKing. The numbers are all decimal, but in this text the first citation often will be in hexadecimal: the Chn number is the number of the channel of the bus; Identifier is the identifier of the CAN message; flg is the flag of the CAN message, a codified number which can tell some information about the latter; DLC is the Data Length Code, that is the number of byte of the CAN message; D0…D7 are the 8 data byte; Time is the arrive time of the message passed since the starting of the session of the CanKing; Dir is a non-sense data in this contest.

The identifier is a fundamental part of a CAN message: it serves like an address, to understand where the message is directed to, or from where it was originated. It is known that there are two kinds of identifiers, with 11 bits and 29 bits. There is a broadcast address, with which all the ECUs receive the message, but only the ECUs that support the mode specified after in the same message can answer. But there are also two identifiers for each ECU: one it is used by the application to dialog with the ECU, and the other is used from the ECU to dialog with the application. With two tables it will be cleared what is the meaning of each identifier. The first one contains the 11 bit identifiers [4]:

IDENTIFIER	**MEANING**
0x7DF	For functionally addressed request messages sent by the application
0x7E0	Request CAN identifier from the application to ECU #1
0x7E8	Response CAN identifier from ECU #1 to the application
0x7E1	Request CAN identifier from the application to ECU #2
0x7E9	Response CAN identifier from ECU #2 to the application
0x7E2	Request CAN identifier from the application to ECU #3
0x7EA	Response CAN identifier from ECU #3 to the application

Table 5.1: 11 bit identifiers

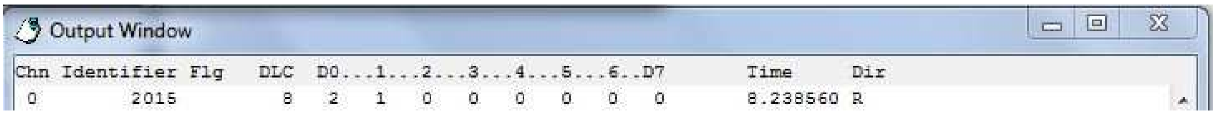
The other identifiers are those with 29 bit [4]:

IDENTIFIER	**MEANING**
0x18DB33F1	For functionally addressed request messages sent by the application
0x18DA**F1	Request CAN identifier from the application to ECU #**
0x18DAF1**	Response CAN identifier from ECU #** to the application

Table 5.2: 29 bit identifiers

All the screens which will be subsequently showed are arranged with 11 bits identifiers, but the application, once recognized the configuration, chooses the correct identifier and sends the message.

Now we will explain the structure and the meaning of each message, with the help of the following pictures:



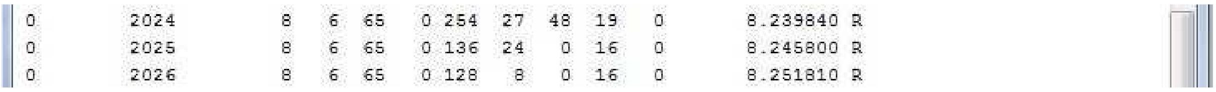
```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7       Time      Dir
 0        2015         8   2   1   0   0   0   0   0   0       8.238560 R
```

Fig. 5.5: Output window for the supported PIDs request

This is the first message of the set. It serves to ask to the device which PIDs are supported in the mode 1: this action, contrarily to these of mode 2 and 9, must be made in the initialization session, to follow the SAE 1979 standard.

The identifier is the 0x7DF (2015): that means the message is broadcast. In the D0 byte, like all the CAN message used in this application, there is a digit that tells the number of meaningful byte. In this case there are 2 bytes: therefore, when the device will receive the message, the byte from the number 3 will be discarded.

The D1 byte is the request for current powertrain diagnostic data request. When the OBD-II system receives this question, it must answer with the supported PIDs. The D2 byte indicates the group of supported PIDs: since the number of PIDs of which a CAN message can bring the information is 32, the PIDs are grouped in set of 32. This byte allow the user to obtain a determinate group: the first is the 0x00 group, the second is the 0x20, the third the 0x40, and so on until the 0xA0 group. After we will see how the next group are asked. The set of response from the device are:



```
 0        2024         8   6  65   0 254  27  48  19   0       8.239840 R
 0        2025         8   6  65   0 136  24   0  16   0       8.245800 R
 0        2026         8   6  65   0 128   8   0  16   0       8.251810 R
```

Fig. 5.6: Output window for the supported PIDs response

The identifiers are different: 0x7E8, 0x7E9 and 0x7EA (2024, 2025, 2026). The first is the ECM (Engine Control Module), the second is the TCM (Transmission Control Module) and the third is the ABS (ABS Module). Each ECU responses with each his own set of available PIDs. The structure of the data bytes is peculiar and identical for each of the 3 messages: the D0 byte (6), always indicate the meaningful bytes. 0x41 (65 in decimal) indicates the mode of response: this number is calculated from the number of the request (in this case 1) adding the number 0x40 (64 in decimal). The third byte (0), indicates the group of PIDs for which the next bytes are valid. From D3 (254) until the next 3 byte, the message represents the supported PIDs; each number represents a binary number, for example:

254 = 11111110

This number, the first of the series, represents the PIDs from 1 to 8. If a bit is set, that means the correspondent PID is available. In this byte, for example, the only PID not available is the number 8. The next 3 byte represent the availability respectively for 9-16, 17-24, 25-32 PIDs, with the same rules of before.

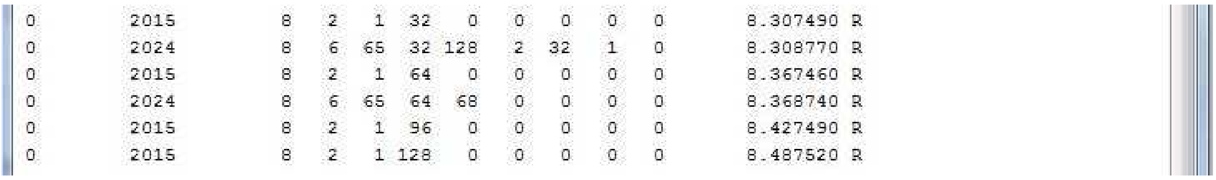
The last meaningful byte has a particularity: compare the two bytes, that of the first and that of the second message and analyze the last bit:

19 = 00010011

16 = 0010000

For the first ECU the bit is set, but not for the second. That means the first ECU supports at least 1 PID of the next groups, while the second ECU doesn’t support any other PIDs.

Going on with the set of messages we find the next one:



```
0      2015      8  2   1  32   0  0   0  0  0     8.307490 R
0      2024      8  6  65  32 128  2  32  1  0     8.308770 R
0      2015      8  2   1  64   0  0   0  0  0     8.367460 R
0      2024      8  6  65  64  68  0   0  0  0     8.368740 R
0      2015      8  2   1  96   0  0   0  0  0     8.427490 R
0      2015      8  2   1 128   0  0   0  0  0     8.487520 R
```

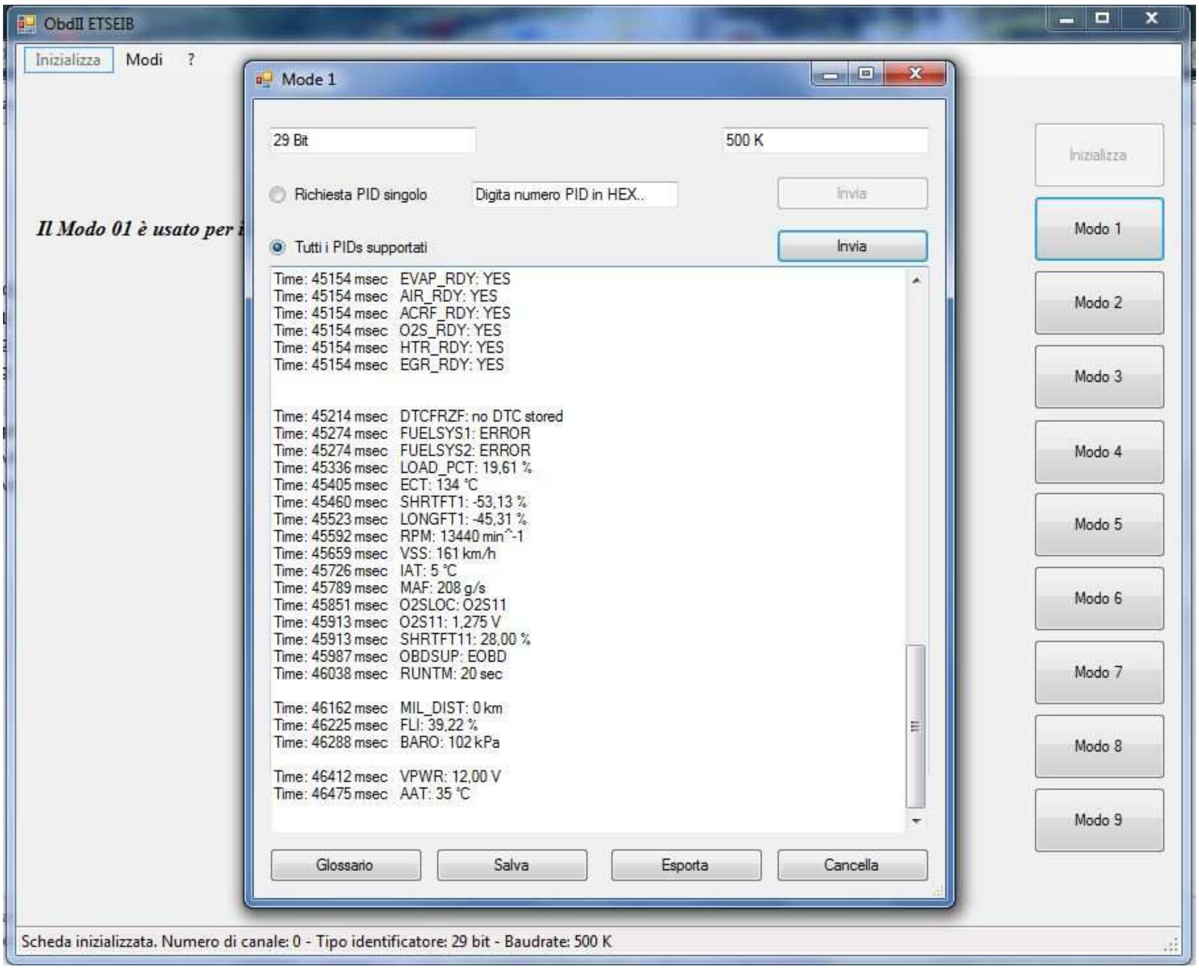
Fig. 5.7: CanKing Output Window for the next groups of supported PIDs

These are the messages for the next groups of PIDs: for the first, like the precedent, the identifier is the same, and the structure as well. The unique different byte is the D2 (0x20 or 32). This indicates to the ECUs to response with the available PIDs in the second group. The next response indeed arrives only from the first ECU and indicates as it should be the expected data. The third and the fourth messages are the same for the third group, while the fifth and the sixth are only request, and since any ECU supports any PID of those groups, no response arrives. The application knows how many groups request through the maximum value parameter, mentioned before.

These messages, as explained before, aim to build the collection of available PIDs, which permits to understand to the application which information is available and which not. The collection contains a pair of keys: the number of the PID, and the set of identifiers of the ECUs that support that PID. This action is performed before the use of the mode: this is a rule of the standard. For example, in mode 2 and 9, the same collections (of the respectively PIDs) are built at the moment of loading the form.

5.3. Mode 1

The mode 1, when a “All supported PIDs” request is made, appears like this:



29 Bit

500 K

○ Richiesta PID singolo   Digita numero PID in HEX..

● Tutti i PIDs supportati

```
Time: 45154 msec   EVAP_RDY: YES
Time: 45154 msec   AIR_RDY: YES
Time: 45154 msec   ACRF_RDY: YES
Time: 45154 msec   O2S_RDY: YES
Time: 45154 msec   HTR_RDY: YES
Time: 45154 msec   EGR_RDY: YES


Time: 45214 msec   DTCFRZF: no DTC stored
Time: 45274 msec   FUELSYS1: ERROR
Time: 45274 msec   FUELSYS2: ERROR
Time: 45336 msec   LOAD_PCT: 19,61 %
Time: 45405 msec   ECT: 134 °C
Time: 45460 msec   SHRTFT1: -53,13 %
Time: 45523 msec   LONGFT1: -45,31 %
Time: 45592 msec   RPM: 13440 min^-1
Time: 45659 msec   VSS: 161 km/h
Time: 45726 msec   IAT: 5 °C
Time: 45789 msec   MAF: 208 g/s
Time: 45851 msec   O2SLOC: O2S11
Time: 45913 msec   O2S11: 1,275 V
Time: 45913 msec   SHRTFT11: 28,00 %
Time: 45987 msec   OBDSUP: EOBD
Time: 46038 msec   RUNTM: 20 sec

Time: 46162 msec   MIL_DIST: 0 km
Time: 46225 msec   FLI: 39,22 %
Time: 46288 msec   BARO: 102 kPa

Time: 46412 msec   VPWR: 12,00 V
Time: 46475 msec   AAT: 35 °C
```

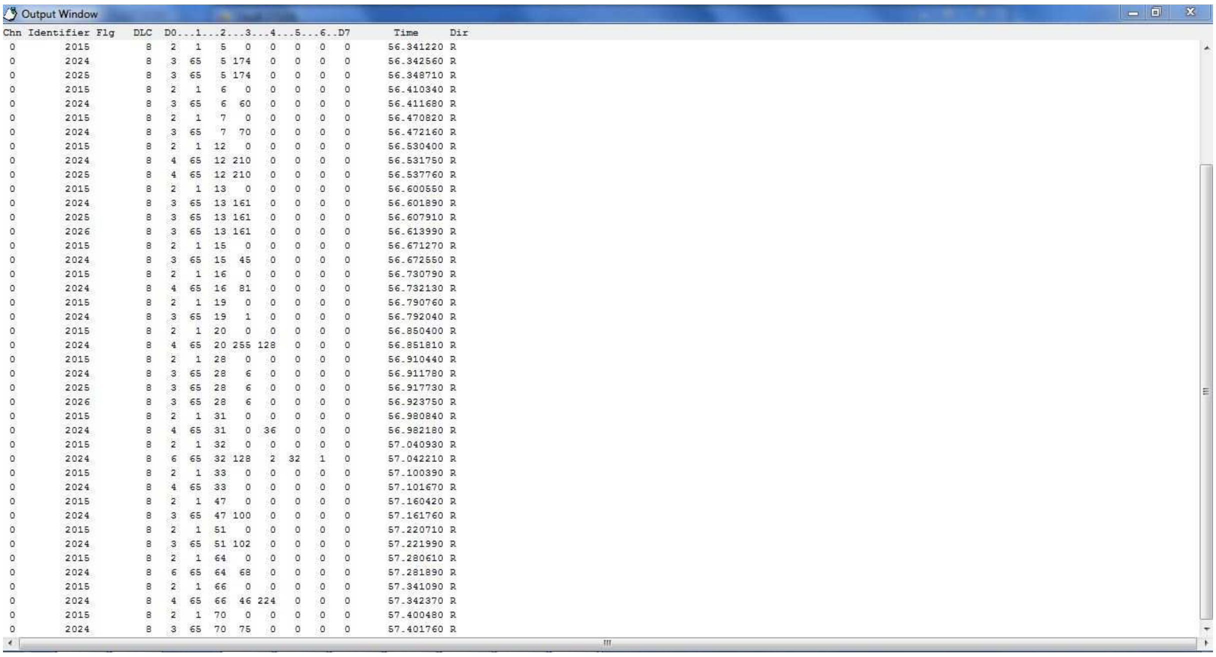
Scheda inizializzata. Numero di canale: 0 - Tipo identificatore: 29 bit - Baudrate: 500 K

Fig. 5.8: Mode 1 form with displayed results

As explained in the precedent chapter, onto the screen there is the information about the configuration of the bus. This information is redundant, because it is already shown in the status bar, but it is indicated for completeness of mode 1, which indicates all the information available of the ODB-II system.

As the picture shows, when an “All supported PID” request is made, the other option of a single request is not available, and vice versa. In the textbox there are all the available results, showed with the arrival time, the standard abbreviation and the data. All PIDs are displayed following the SAE J1979 standard, with their acronyms and their proper approximation and unit of measure.

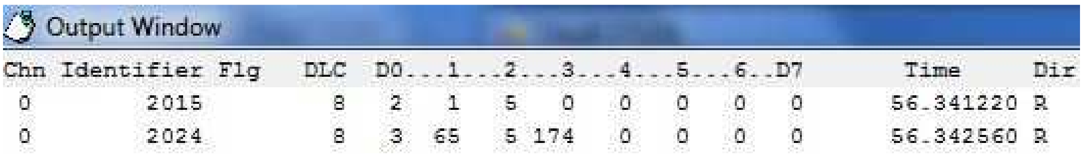
For this request, at the message – level, the idea is to consult the collection mentioned before, and look for the PIDs contained. For each one of them, in increasing order, send the correspondent request and wait the response. After that, the response is analyzed and displayed in the textbox with the correspondent result. At the end of the response, the output window of the CanKing appears like this:



```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7       Time     Dir
 0        2015          8   2   1   5   0   0   0   0   0      56.341220 R
 0        2024          8   3  65   5 174   0   0   0   0      56.342560 R
 0        2025          8   3  65   5 174   0   0   0   0      56.348710 R
 0        2015          8   2   1   6   0   0   0   0   0      56.410340 R
 0        2024          8   3  65   6  60   0   0   0   0      56.411680 R
 0        2015          8   2   1   7   0   0   0   0   0      56.470820 R
 0        2024          8   3  65   7  70   0   0   0   0      56.472160 R
 0        2015          8   2   1  12   0   0   0   0   0      56.530400 R
 0        2024          8   4  65  12 210   0   0   0   0      56.531750 R
 0        2025          8   4  65  12 210   0   0   0   0      56.537760 R
 0        2015          8   2   1  13   0   0   0   0   0      56.600550 R
 0        2024          8   3  65  13 161   0   0   0   0      56.601890 R
 0        2025          8   3  65  13 161   0   0   0   0      56.607910 R
 0        2026          8   3  65  13 161   0   0   0   0      56.613990 R
 0        2015          8   2   1  15   0   0   0   0   0      56.671270 R
 0        2024          8   3  65  15  45   0   0   0   0      56.672550 R
 0        2015          8   2   1  16   0   0   0   0   0      56.730790 R
 0        2024          8   4  65  16  81   0   0   0   0      56.732130 R
 0        2015          8   2   1  19   0   0   0   0   0      56.790760 R
 0        2024          8   3  65  19   1   0   0   0   0      56.792040 R
 0        2015          8   2   1  20   0   0   0   0   0      56.850400 R
 0        2024          8   4  65  20 255 128   0   0   0      56.851810 R
 0        2015          8   2   1  28   0   0   0   0   0      56.910440 R
 0        2024          8   3  65  28   6   0   0   0   0      56.911780 R
 0        2025          8   3  65  28   6   0   0   0   0      56.917730 R
 0        2026          8   3  65  28   6   0   0   0   0      56.923750 R
 0        2015          8   2   1  31   0   0   0   0   0      56.980840 R
 0        2024          8   4  65  31   0  36   0   0   0      56.982180 R
 0        2015          8   2   1  32   0   0   0   0   0      57.040930 R
 0        2024          8   6  65  32 128   2  32   1   0      57.042210 R
 0        2015          8   2   1  33   0   0   0   0   0      57.100390 R
 0        2024          8   4  65  33   0   0   0   0   0      57.101670 R
 0        2015          8   2   1  47   0   0   0   0   0      57.160420 R
 0        2024          8   3  65  47 100   0   0   0   0      57.161760 R
 0        2015          8   2   1  51   0   0   0   0   0      57.220710 R
 0        2024          8   3  65  51 102   0   0   0   0      57.221990 R
 0        2015          8   2   1  64   0   0   0   0   0      57.280610 R
 0        2024          8   6  65  64  68   0   0   0   0      57.281890 R
 0        2015          8   2   1  66   0   0   0   0   0      57.341090 R
 0        2024          8   4  65  66  46 224   0   0   0      57.342370 R
 0        2015          8   2   1  70   0   0   0   0   0      57.400480 R
 0        2024          8   3  65  70  75   0   0   0   0      57.401760 R
```

Fig. 5.9: CanKing Output Window for the mode 1 PIDs results

Obviously, that is a part of the longer screen. We will take as an example two significant messages:



```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7       Time     Dir
 0        2015          8   2   1   5   0   0   0   0   0      56.341220 R
 0        2024          8   3  65   5 174   0   0   0   0      56.342560 R
```

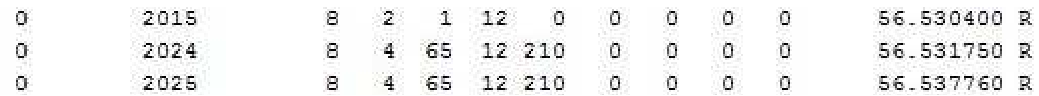
Fig. 5.10: CanKing Output Window for the PID n° 5 request and response

The first example message is the PID n° 5: Engine Coolant Temperature. The arrangement of the first message is the same for all the PID request, and is the same of which explained before, the unique difference is about the D2: in this byte the user puts the number of the requested PID. The particularity is the identifier: it is always the broadcast identifier (0x7DF or 2015); indeed, when a request is made, each ECU receives it. After that, the ECUs look if they support this PID, and if yes, they send a response. Therefore, it can happen that there are many messages, derived from different ECUs. But because the result is the same for each of them, the result will be displayed once.

The second message in this picture shows the response: the first interesting data is the identifier, which tells which ECU is answering. After that, the D0 byte indicates 3 meaningful bytes: the mode of response (0x41 or 65), the PID number (5) and the value. Talking about the value, each PID often has a minimum and a maximum value, and an offset. In this case, the SAE J1979 standard tell us that there is a minimum of -40° C, a maximum of 215° C and an offset of -40°. In this case, if we look at the fig. 5.8, at the ECT PID correspond a value of 134° C. Comparing this with the fig.5.10 we understand that the result is obtained with the operation:

$$174 – 40 = 134$$

It is a PID very easy to analyze. A more difficult structure and operation belongs to PID 13, or 0xC. This is the Engine Rotation Per Minute and has the following structure:



```
0        2015        8   2    1   12    0    0    0    0    0       56.530400 R
0        2024        8   4   65   12  210    0    0    0    0       56.531750 R
0        2025        8   4   65   12  210    0    0    0    0       56.537760 R
```

Fig. 5.11: CanKing Output Window for the PID 0xC request and response

The message request corresponding at that of PID 0xC, while the responses (arriving from ECU 0x7E8 and 0x7E9 that sent the same message), presents 4 meaningful data: 210 and 0. In formula, the operation to perform is:

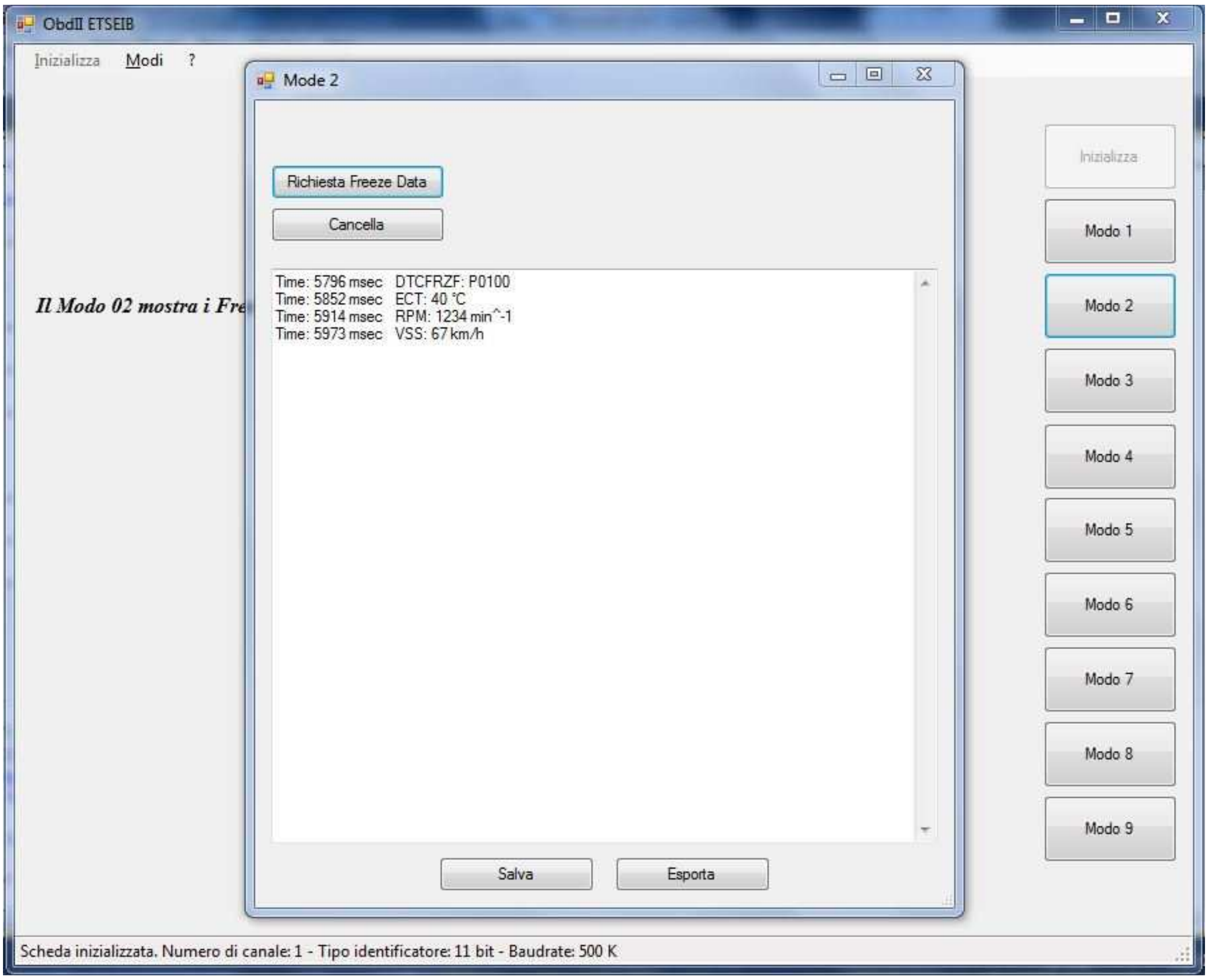
$$(A * 256) + B) / 4$$

Practically, the result is contained in two byte, and must be divided by 4 before showing.

Other PIDs need a select case structure to decode the message, which is codified setting bits inside a byte. For example, the n° 3 (Fuel system 1 status) or the 0x1C (OBD requirements to which vehicle is designed).

5.4. Mode 2

When the mode 2 is used and a DTC is stored, it appears with this screen:



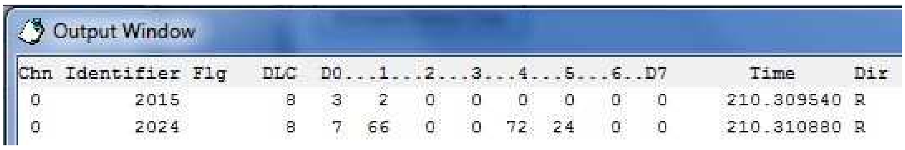
```
Time: 5796 msec   DTCFRZF: P0100
Time: 5852 msec   ECT: 40 °C
Time: 5914 msec   RPM: 1234 min^-1
Time: 5973 msec   VSS: 67 km/h
```

Scheda inizializzata. Numero di canale: 1 - Tipo identificatore: 11 bit - Baudrate: 500 K

Fig. 5.12: Mode 2 form with displayed results

The textbox and the composition of the screen are simpler than that of mode 1. There is the only possibility to ask all the supported PIDs, because the Freeze Frame Data have sense only altogether and, above all, if there is any error that has generated these data. The PIDs are listed as usual, with time, acronym, result and unit of measure. The first row indicates if there is a DTC that has generated the Freeze Frame Data and if yes, which is.

For this mode, the correspondent collection of supported PID is built when the form is loaded. The mechanism is quit the same of that of mode 1; here it is showed the messages:



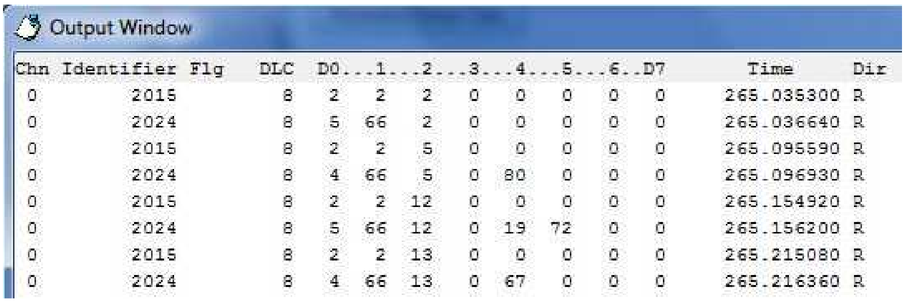
```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7      Time      Dir
 0        2015           8   3   2   0   0   0   0   0   0   210.309540 R
 0        2024           8   7  66   0   0  72  24   0   0   210.310880 R
```

Fig. 5.13: CanKing Output Window for the supported PIDs request

The first message is the request for the supported PIDs. The D0 byte tells to the user that there are 3 meaningful bytes. The D1 is the mode, the second is the group of PIDs requested (like in mode 1) and the D3 is the frame number. The frame number byte will indicate 0 for the freeze frame data. Manufacturers may optionally save additional freeze frames and use this service to obtain that data by specifying the freeze frame number in the request message.

The response has 7 important bytes, the D1 and D2 are respectively the response mode and the PID group, the D3 is the freeze frame number and the subsequently until the D7 are the supported PIDs, encoded in the same way of mode 1. Even in this mode there are many group of PID, but in the Conclusion chapter will be explained why it is asked only the first one.

After building the collection, the user can request the freeze frame data through the Request button. Once more, the algorithm is the same: look for all the available PIDs and for each one ask, analyze and show the answer. The set of messages is:



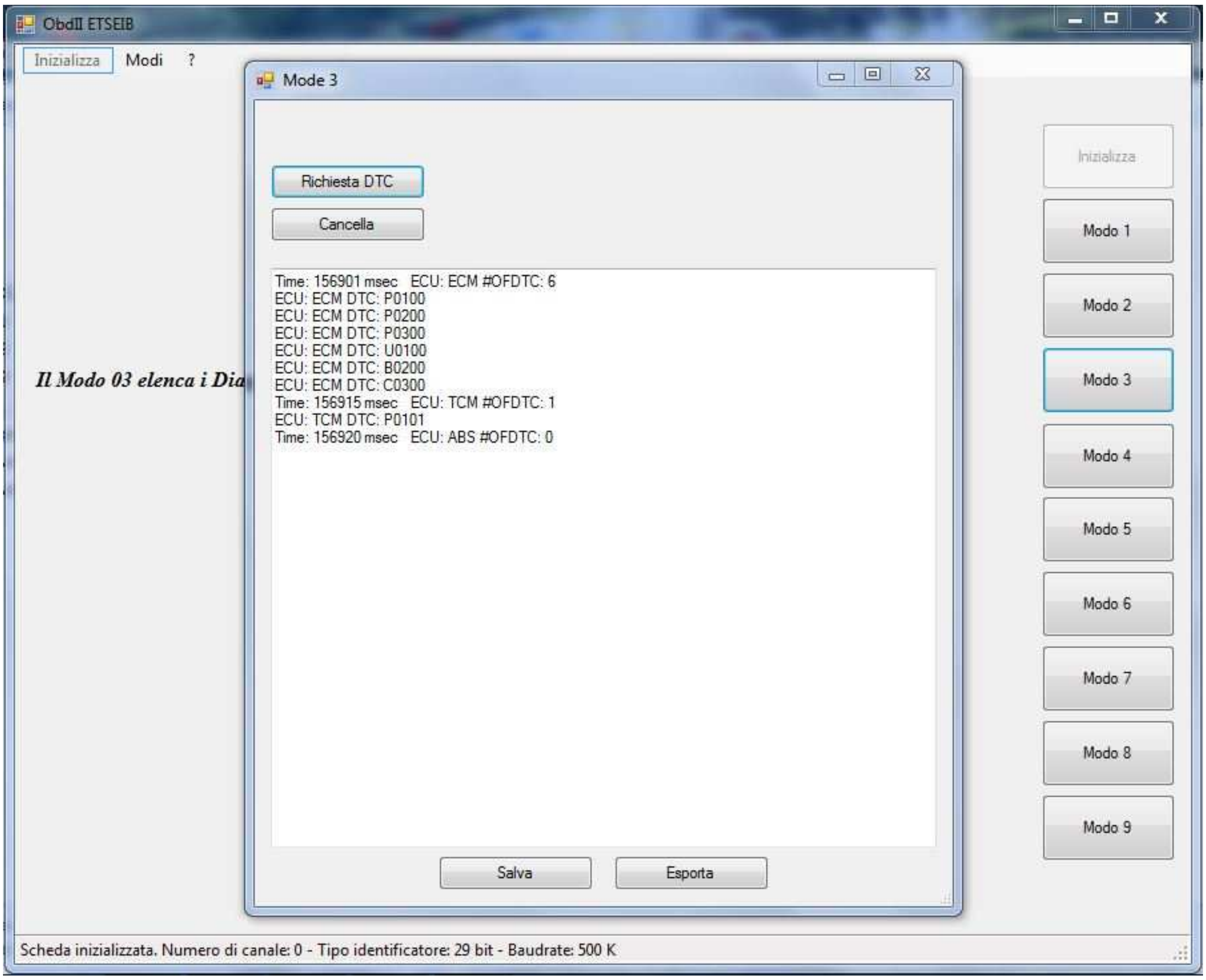
```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7        Time      Dir
 0         2015          8   2   2   2   0   0   0   0   0   265.035300 R
 0         2024          8   5  66   2   0   0   0   0   0   265.036640 R
 0         2015          8   2   2   5   0   0   0   0   0   265.095590 R
 0         2024          8   4  66   5   0  80   0   0   0   265.096930 R
 0         2015          8   2   2  12   0   0   0   0   0   265.154920 R
 0         2024          8   5  66  12   0  19  72   0   0   265.156200 R
 0         2015          8   2   2  13   0   0   0   0   0   265.215080 R
 0         2024          8   4  66  13   0  67   0   0   0   265.216360 R
```

Fig. 5.14: CanKing Output Window for the PIDs result request and response

Comparing with the messages in mode 1, there is a byte more in each one. It is the D3, and it is set to zero because is the freeze frame. For the remaining bytes, both for the request and the response, the organization is the same. That because the PIDs' number and structure are the same for mode 1 and mode 2. Indeed, from the Fig. 5.11 we can see that there are 3 PIDs, and they are supported even in the mode 1. Obviously, since the freeze frame data idea is to "take a photo" of the situation when an error occurs, the results are not the same.

5.5. Mode 3

The mode 3 provides to show the DTCs stored in the ECUs. The screen adopted is here:



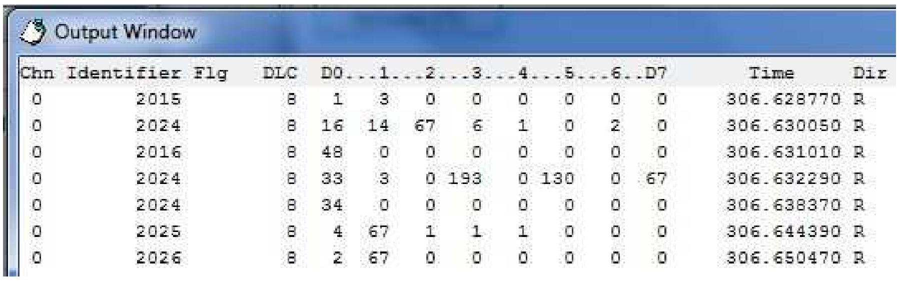
```
Time: 156901 msec   ECU: ECM #OFDTC: 6
ECU: ECM DTC: P0100
ECU: ECM DTC: P0200
ECU: ECM DTC: P0300
ECU: ECM DTC: U0100
ECU: ECM DTC: B0200
ECU: ECM DTC: C0300
Time: 156915 msec   ECU: TCM #OFDTC: 1
ECU: TCM DTC: P0101
Time: 156920 msec   ECU: ABS #OFDTC: 0
```

Scheda inizializzata. Numero di canale: 0 - Tipo identificatore: 29 bit - Baudrate: 500 K

Fig. 5.15: Mode 3 form with displayed DTCs

As explained before, for each ECU the number of DTCs stored are shown, and subsequently the codes of the latter. If there are not any DTC saved there will be only a row for each ECU, displaying the time and the number of DTC, that is 0.

With regard to the messages traffic, this is the figure:



```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7       Time     Dir
 0        2015        8    1   3   0   0   0   0   0   0   306.628770 R
 0        2024        8   16  14  67   6   1   0   2   0   306.630050 R
 0        2016        8   48   0   0   0   0   0   0   0   306.631010 R
 0        2024        8   33   3   0 193   0 130   0  67   306.632290 R
 0        2024        8   34   0   0   0   0   0   0   0   306.638370 R
 0        2025        8    4  67   1   1   1   0   0   0   306.644390 R
 0        2026        8    2  67   0   0   0   0   0   0   306.650470 R
```

Fig. 5.16: CanKing Output Window for the stored DTCs request and response

The first broadcast message provides to ask the stored DTCs at each ECU. There is only one important byte; it is the D1, which indicates the mode number.

This mode is capable to answer with multiple messages: because of the information could not be putted in a unique message, they are broken in multiple messages, in a specific way. Looking at the second message in Fig. 5.15, the D0 byte indicates a number of byte of 16. This byte, in a single message, should indicate the number of useful bytes.

In a multiple message, instead, it has a different meaning. It is divided in two nibbles: the first represents the type of multiple messages, while the second, together with the second byte, is the number of meaningful bytes. The types of multiple messages are: single frame (0), first frame (1), consecutive frame (2) and control frame (3). Applying to our case, the number 16 (00010000b) indicates a first frame message, and the next one (14) tells the number of useful bytes.

D2 is the mode of response, always calculated adding 0x40 to the request mode. The next byte represents the number of DTC stored in the ECU represented from the identifier. The next two pairs of bytes are the codes for the first two DTCs. For each error the OBD-II system needs of two bytes.

The first pair of bits of the first byte represents the letter of the DTC, which allows the user to localize the problem: 00 is Powertrain error, 01 is Chassis error, 10 is Body error and 11 is Network error.

The second pair of bits represents the first digit of the DTC, while the first and the second nibble of the second byte are the two other digits. For example, the first pair of bytes representing a DTC are 1 and 0, that corresponds to P0100.

For each type of multiple messages, the application reads the contained DTCs. After reading the information in a first frame message, always the program must advise the correspondent ECU that is ready to receive the other messages. To do that, it sends a control frame message like the third message in Fig. 5.15; it is clear how, to communicate with this unit, the identifier musts be 2016 (0x7E0), that is the response CAN identifier used from the application to interact with the first ECU. The only meaningful byte in this control frame message is the first, where the first nibble is a control frame. The second nibble could be 0 or 1: 0 means “clear to send” while 1 tells to the system to wait. With the subsequent two bytes the user can tell to the system how many and how frequently messages could receive. When the bytes are 0, it means to send how messages are available as frequently as possible.

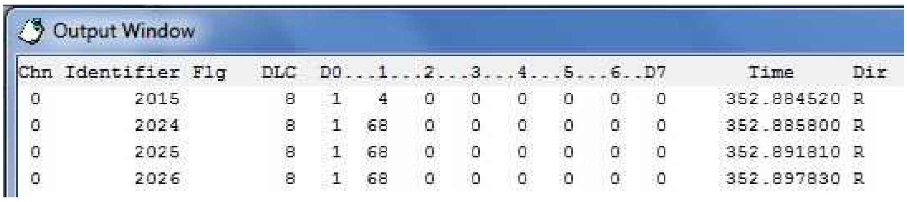
After receiving this information the system provides to send the remaining information. The first byte of each message is made of two nibbles: the first indicate that they are consecutive frame, while the second is the order number. The next bytes are all the data about the DTCs. Obviously, the number of DTCs present in the ECU tells to the application until where read bytes.

In the case of a single message, like those of 0x7E9 and 0x7EA, the first message is as always the important bytes, the second is the response mode, the third the number of DTCs and the next are the codes. In the case of no DTCs present, like the last message, the third byte is 0 and the subsequent bytes are useless.

5.6. Mode 4

Mode 4 doesn’t have a screen to develop its job. That mode only interacts with the user through messages, both if there are errors to delete and if there is not.

Analyzing the traffic message in the case that there are errors to clean, the CanKing screen results:



```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7       Time      Dir
 0        2015          8   1   4   0   0   0   0   0   0     352.884520 R
 0        2024          8   1  68   0   0   0   0   0   0     352.885800 R
 0        2025          8   1  68   0   0   0   0   0   0     352.891810 R
 0        2026          8   1  68   0   0   0   0   0   0     352.897830 R
```

Fig. 5.17: CanKing Output Window for clear DTCs request and response

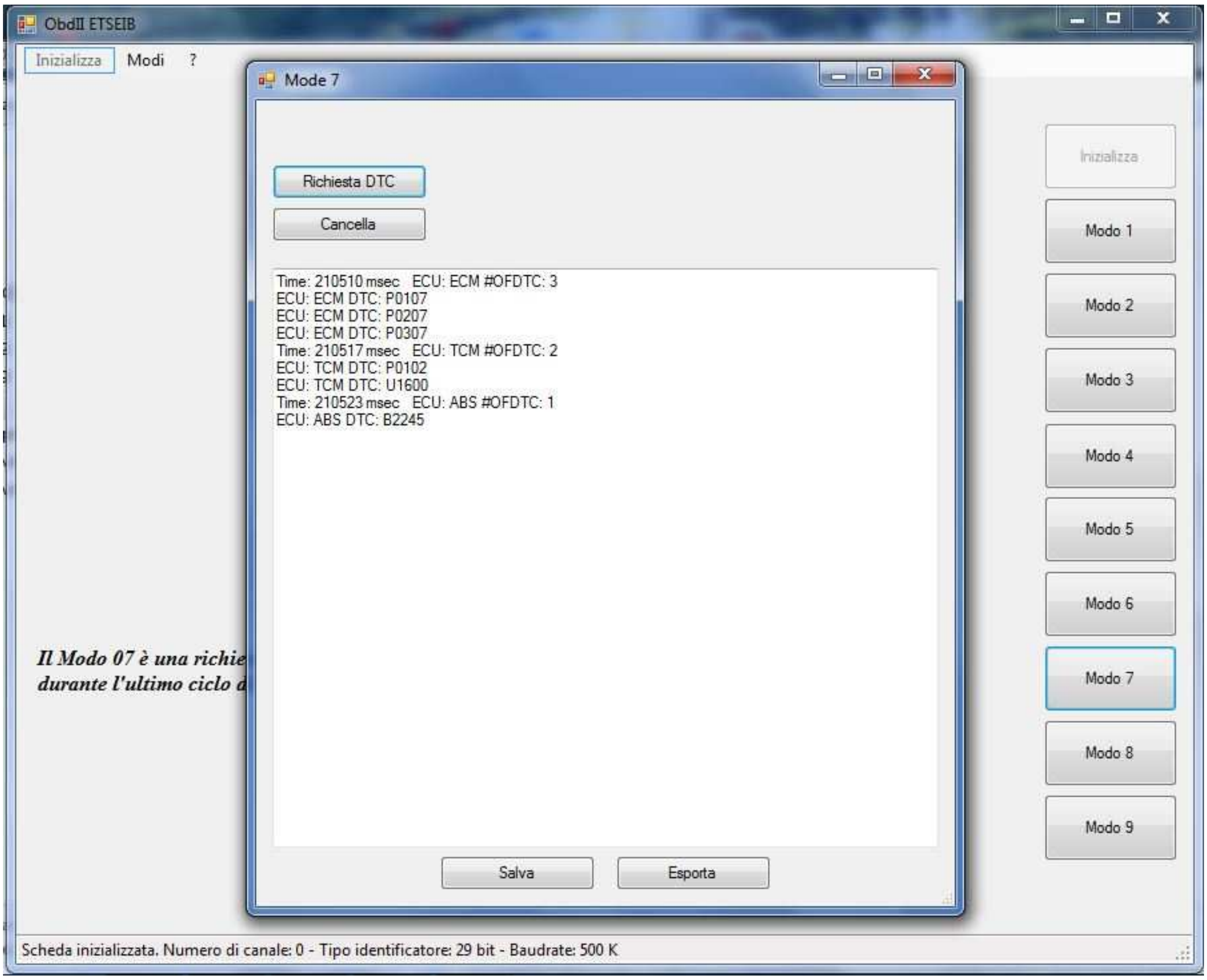
With a unique broadcast message the application tries to cancel all the DTCs present in all the ECUs. There is not any possibility to clear the errors in a unique ECU, because this action doesn’t make sense.

The D1 byte is set to 4, the number of the mode. In the response, this number turns in 68 (0x40 + 0x4). In the case that there is not error in an ECU, the latter doesn’t response to this message. In this screen, then, it is noted how even the third ECU (ABS) has errors, while in mode 3 the screen shows that ABS doesn’t have DTCs stored. This happens because DTCs belong even to mode 7; in the next subchapter, indeed, it will be clear that even ABS has DTCs stored.

There is even an error condition, which happens when there is a fault in the action of clean DTCs. In this case, byte D1 turns in 0x7F and the application, detecting this data, advises the user with an error message.

5.7. Mode 7

This mode is very similar to mode 3. The function is the same; the DTCs pointed out are different, as explained before. For this, the screen and the message traffic are quite identical:

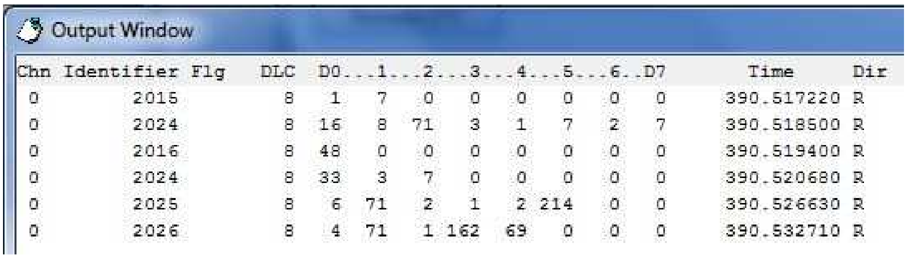


```
Time: 210510 msec   ECU: ECM #OFDTC: 3
ECU: ECM DTC: P0107
ECU: ECM DTC: P0207
ECU: ECM DTC: P0307
Time: 210517 msec   ECU: TCM #OFDTC: 2
ECU: TCM DTC: P0102
ECU: TCM DTC: U1600
Time: 210523 msec   ECU: ABS #OFDTC: 1
ECU: ABS DTC: B2245
```

Il Modo 07 è una richie
durante l'ultimo ciclo d

Scheda inizializzata. Numero di canale: 0 - Tipo identificatore: 29 bit - Baudrate: 500 K

Fig. 5.18: Mode 7 form with displayed DTCs



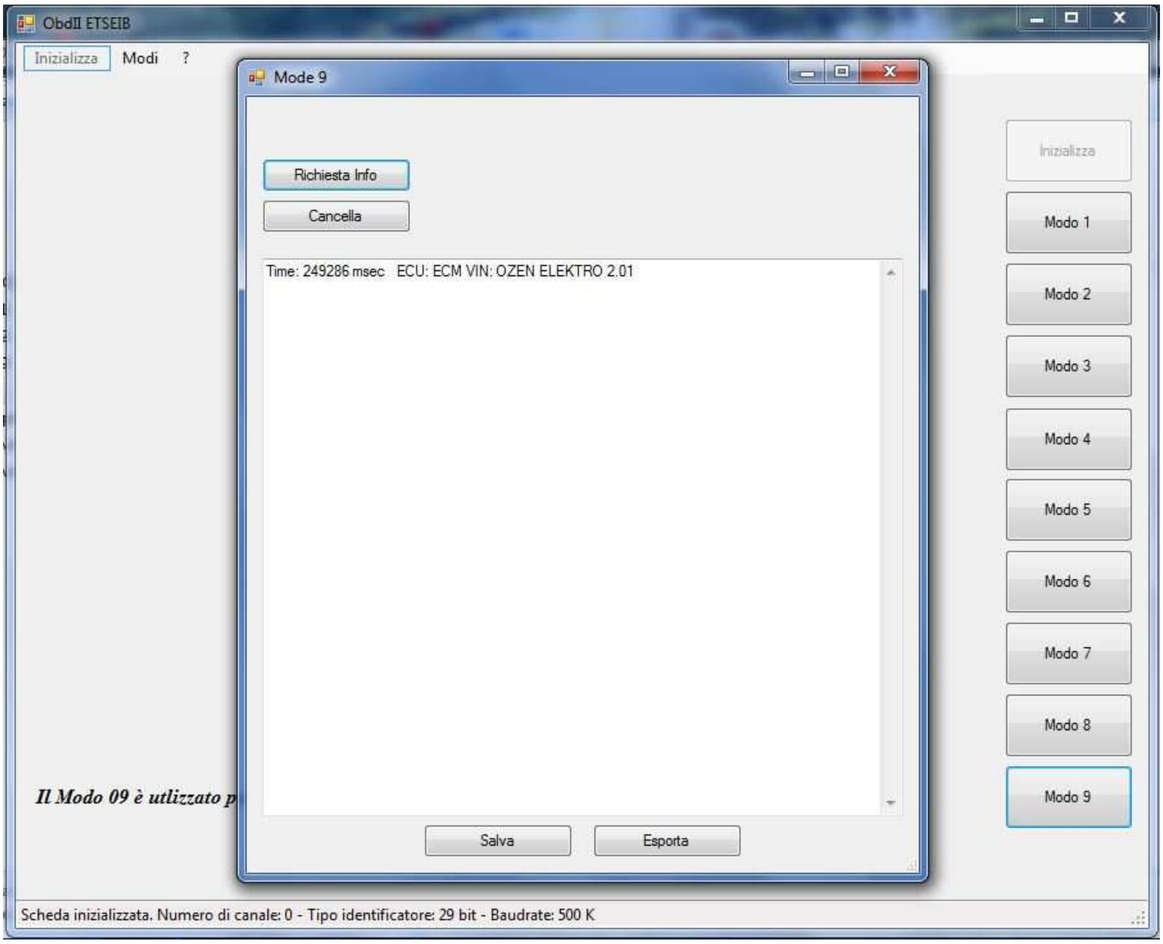
```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7        Time     Dir
 0        2015          8   1   7   0   0   0   0   0   0    390.517220 R
 0        2024          8  16   8  71   3   1   7   2   7    390.518500 R
 0        2016          8  48   0   0   0   0   0   0   0    390.519400 R
 0        2024          8  33   3   7   0   0   0   0   0    390.520680 R
 0        2025          8   6  71   2   1   2 214   0   0    390.526630 R
 0        2026          8   4  71   1 162  69   0   0   0    390.532710 R
```

Fig. 5.19: CanKing Output Window for the stored DTCs request and response

This time the D1 byte turns to 7, and analyzing the message traffic we discover the first ECU with 3 DTCs (D3 in second message) and therefore multiple message, the second with 2 (D2 of second last message) and the third with 1 error (D2 of the last message).

5.8. Mode 9

This mode looks like mode 2: there is some information to obtain about the vehicle, to keep track of these there is a collection, that is built when the form is loaded and there is a textbox which collects these data. The way to construct the collection is always the same. Indeed, the screen is:



```
Time: 249286 msec   ECU: ECM VIN: OZEN ELEKTRO 2.01
```

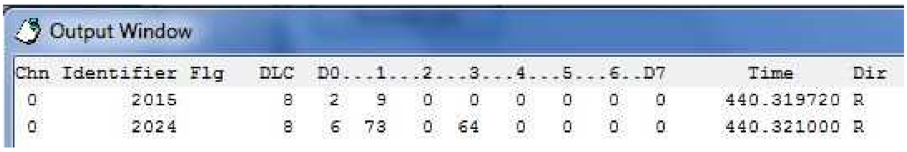
Il Modo 09 è utlizzato p

Scheda inizializzata. Numero di canale: 0 - Tipo identificatore: 29 bit - Baudrate: 500 K

Fig. 5.20: Mode 9 form with displayed information

The unique PID available in this mode, for this OZEN ELEKTRONIK device, is VIN (Vehicle Identification Number).

As already said, the collection is built loading the form, with these messages:

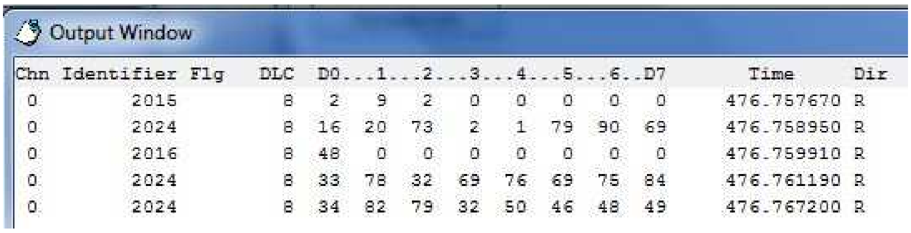


```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7        Time     Dir
 0         2015          8   2   9   0   0   0   0   0   0   440.319720 R
 0         2024          8   6  73   0  64   0   0   0   0   440.321000 R
```

Fig. 5.21: CanKing Output Window for supported PIDs request and response

In the broadcast message the D1 byte is set to 9, the used mode, while in the unique response the same byte is 73 (0x40 + 0x9). Other ECUs don’t answer because they don’t support this mode.

When a request of information is made, these are the messages:



```
Chn Identifier Flg   DLC  D0...1...2...3...4...5...6..D7        Time     Dir
 0         2015          8   2   9   2   0   0   0   0   0   476.757670 R
 0         2024          8  16  20  73   2   1  79  90  69   476.758950 R
 0         2016          8  48   0   0   0   0   0   0   0   476.759910 R
 0         2024          8  33  78  32  69  76  69  75  84   476.761190 R
 0         2024          8  34  82  79  32  50  46  48  49   476.767200 R
```

Fig. 5.22: CanKing Output Window for information request and response

The first message is a request of PID 2; indeed the D2 byte is set to 2. The response is a multiple message, and the division and the identification of the type of the message are the same already mentioned. Indeed, the D0 and D1 bytes concern with the length of the message, the D2 is the response mode, while the D4 is the number of data items. From D5 to D7 there are the first three ASCII characters of the data requested. Following the rules for the multiple messages, the application sends to the tested ECU a frame control message; to do this, the identifier must be the request CAN identifier, used by the application to communicate with that ECU (2016 or 0x7E0). In response to the latter message, the ECU sends all the other messages, where the first byte is the consecutive frame, and all the other are ASCII characters. Obviously, the application stops to read characters when all the information specified in the D1 byte of the first response message are examined.

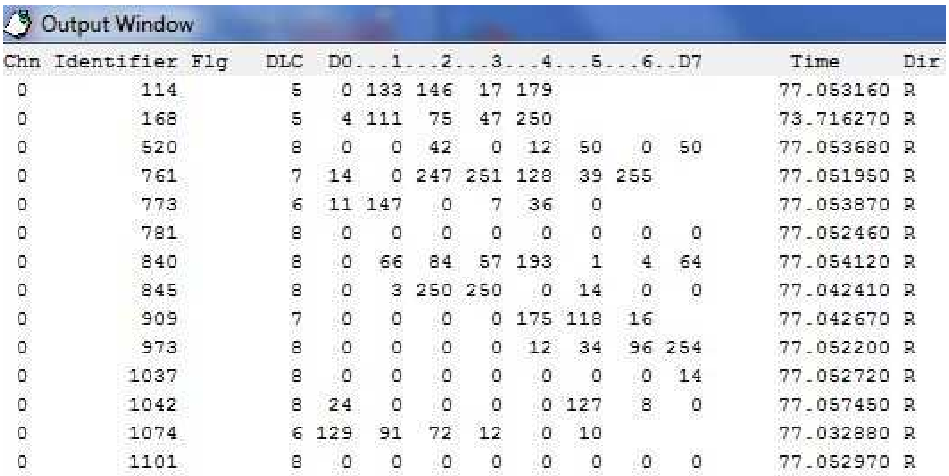
The string resulting from these three encoded messages is indeed “OZEN ELEKTRO 2.01”.

6. REAL TESTS

June 2012, the application was tested at the Mavisa - Peugeot dealer in Sabadell. The intent was to discover if the program worked in the same way both with the emulator and the real vehicle.

The first test was made in a Peugeot 206, a 2002 car. Once connected the PC at the J1962, the application was launched. The system did not respond, this means that the PC sent the messages, but the OBD system could not recognize and elaborate them. This happened because of the age of the car, indeed in 2002 there was not the duty of implement the CAN bus in the OBD system.

For this reason another car was used to test the application: the Peugeot 207 SW. This car, built in 2008, had to support the CAN bus for its OBD system. Indeed, once connected the J1962 connector and turned the key, the Kvaser CanKing started to sniff several messages, detecting a 11 bit, 500 kbit/s CAN bus. The situation is showed in figure 6.1:

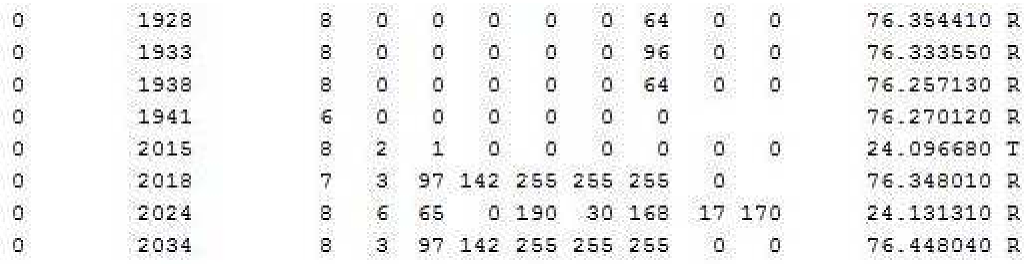


```
Chn Identifier Flg    DLC  D0...1...2...3...4...5...6..D7         Time       Dir
 0         114          5    0 133 146  17 179                   77.053160 R
 0         168          5    4 111  75  47 250                   73.716270 R
 0         520          8    0   0  42   0  12  50   0  50       77.053680 R
 0         761          7   14   0 247 251 128  39 255           77.051950 R
 0         773          6   11 147   0   7  36   0               77.053870 R
 0         781          8    0   0   0   0   0   0   0   0       77.052460 R
 0         840          8    0  66  84  57 193   1   4  64       77.054120 R
 0         845          8    0   3 250 250   0  14   0   0       77.042410 R
 0         909          7    0   0   0   0 175 118  16           77.042670 R
 0         973          8    0   0   0   0  12  34  96 254       77.052200 R
 0        1037          8    0   0   0   0   0   0   0  14       77.052720 R
 0        1042          8   24   0   0   0   0 127   8   0       77.057450 R
 0        1074          6  129  91  72  12   0  10               77.032880 R
 0        1101          8    0   0   0   0   0   0   0   0       77.052970 R
```

Fig. 6.1: CanKing Output Window for real test messages

First of all we considered the identifiers: normally, for the diagnostic messages, they must start from 2024 or 0x7E8. In this screen, the identifiers were all lower than this number. Another unexpected fact is that the messages were continuously: since when the system was started, until when it was turned off, the Kvaser CanKing sniffed constantly messages.

These facts indicated that these messages were not for diagnostic purpose. In fact, when a diagnostic message was sent from the application, the system responded with a diagnostic message, plus several non – diagnostic messages:



```
0      1928      8  0   0   0   0   0  64   0   0     76.354410 R
0      1933      8  0   0   0   0   0  96   0   0     76.333550 R
0      1938      8  0   0   0   0   0  64   0   0     76.257130 R
0      1941      6  0   0   0   0   0   0             76.270120 R
0      2015      8  2   1   0   0   0   0   0   0     24.096680 T
0      2018      7  3  97 142 255 255 255   0         76.348010 R
0      2024      8  6  65   0 190  30 168  17 170     24.131310 R
0      2034      8  3  97 142 255 255 255   0   0     76.448040 R
```

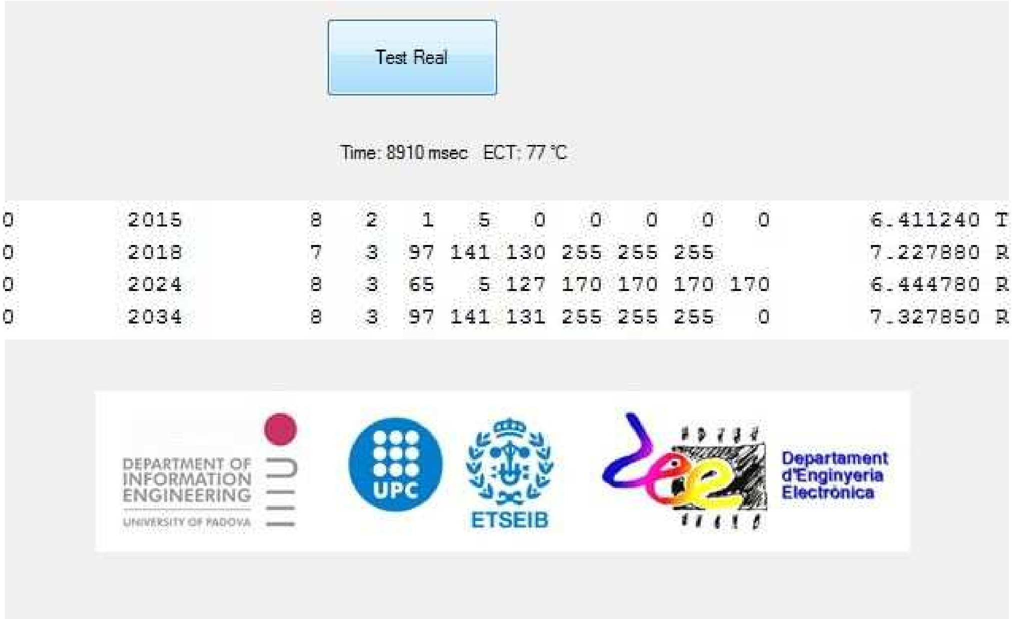
Fig. 6.2: CanKing Output Window for diagnostic real test messages

The request message, defined by a 2015 identifier (0x7DF), was followed by a 2018 message, then by the response message, with the real requested data, and finally by a 2034 message. The meaning of the two non – diagnostic message was not clear: every constructor personalizes the system, adding messages or other details at the standard ones.

These important differences between the behavior of the emulator and the real car one, forced to modify the application: it had to recognize the messages that had the wished identifier and analyze it. Practically, some exception were introduced, based on the identifiers of the messages, with the purpose to ignore the non – diagnostic messages.

To test the functioning of the modified application, was verified the behavior of two PIDs: the 0x05 (Engine Coolant Temperature) and the 0x0C (Engine RPM). The expected behavior was that, once clicked a Test Button, a label showed a real time value of the two PID, together with the time value of the CAN message, both refreshed every few millisecond.

The screen capture, testing the first PID was:



Test Real
DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA
UPC
ETSEIB
Departament d'Enginyeria Electrònica

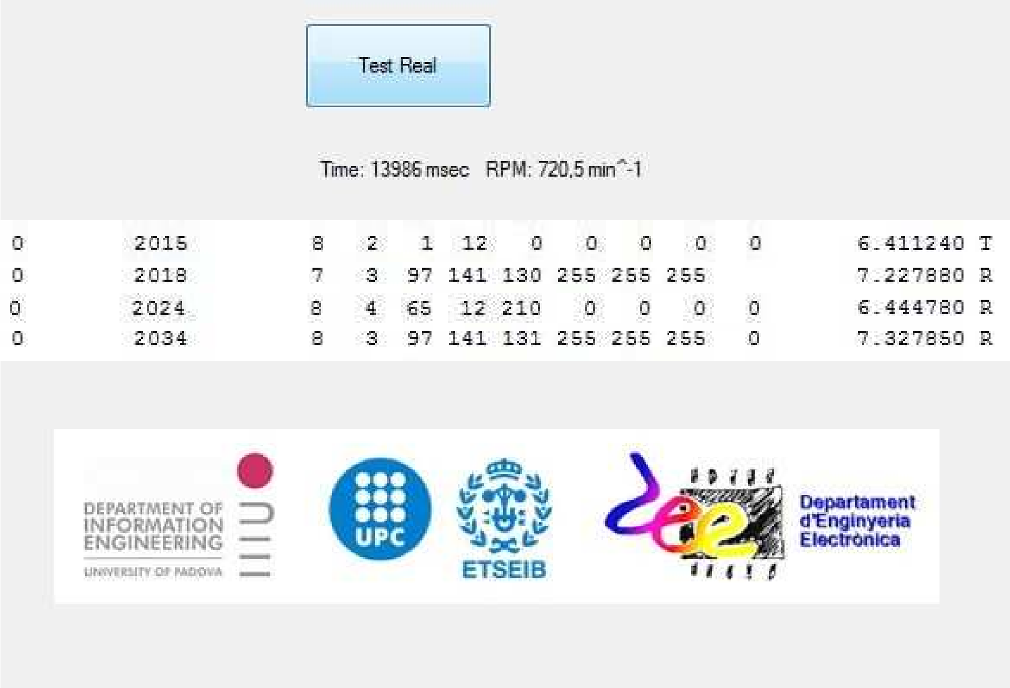
Time: 8910 msec   ECT: 77 °C

```
0      2015      8  2   1   5   0   0   0   0   0     6.411240 T
0      2018      7  3  97 141 130 255 255 255         7.227880 R
0      2024      8  3  65   5 127 170 170 170 170     6.444780 R
0      2034      8  3  97 141 131 255 255 255   0     7.327850 R
```

Fig. 6.3: CanKing Output Window and application screen for PID 0x05

In figure 6.3 is showed the result of the real test with the PID 0x05: once clicked the Real Test, the application entered in a loop, that allowed to request and obtain the real time data regarding the Engine Coolant Temperature. The structure of the request and response were the same mentioned in the previous chapter, even if in this case they were broken up by other “personalized” messages. Even the methods used to elaborate the CAN message and obtain the real data were the same.

About the second PID test, the screens were these:


Test Real
DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA
UPC
ETSEIB
Departament d'Enginyeria Electrònica

Time: 13986 msec   RPM: 720,5 min^-1

```
0     2015     8   2   1  12   0   0   0   0   0     6.411240 T
0     2018     7   3  97 141 130 255 255 255         7.227880 R
0     2024     8   4  65  12 210   0   0   0   0     6.444780 R
0     2034     8   3  97 141 131 255 255 255   0     7.327850 R
```

Fig. 6.4: CanKing Output Window and application screen for PID 0x0C

As the previous case, once clicked the Test Button, the data was refreshed instantly and showed in the central label.

In conclusion, the application works correctly both with the emulator and the real car. In the last case, it needs some small modification, but the structure and the methods used are the same.

7. PROJECT COST

To build an application a designer musts spend time and money. Here there is a summary of the costs, both for human and material resources.

7.1. Instrumentation cost

Talking about the instrumentation, we used the OBD-II simulator from OZEN ELEKTRONIK, the CAN – USB converter from Kvaser, the J1962 – CAN adapter and the notebook, with the license for the Microsoft Visual Studio 2010 Professional:

DESCRITPION	QUANTITY	PRICE [€]	TOTAL [€]
mOByDic 1610 CAN BUS ECU Simulator [7]	1	99 €	99 €
Kvaser Leaf Light HS [6]	1	180 €	180 €
Kvaser OBD II Adapter Cable [6]	1	60 €	60 €
Asus X53S Series Notebook	4 months	800 €	52 €
Microsoft Visual Studio 2010 Prof. Lic. [8]	1	534 €	534 €
TOTAL			**925 €**

Table 7.1: Instrumentation costs

All the entries are clear, except the cushioning of the notebook: it is calculated taking the entire cost of the device, dividing for the medium life time and multiplying for the moths spent for the project.

7.2. Engineering cost

The engineering costs are divided in four parts: the hours spent for reading the learning material about the OBD-II system and other topics; the ones spent for programming the application; the ones spent for writing this thesis and finally for test it.

DESCRITPION	QUANTITY [hs]	PRICE [€]	TOTAL [€]
Reading	60 hs	20 €	1.200 €
Programming	259 hs	40 €	10.360 €
Writing	120 hs	40 €	4.800 €
Testing	5 hs	40 €	200 €
TOTAL			**16.560 €**

Table 7.2: Operator costs

Totally, the cost results 17.485 €.

8. ENVIRONMENTAL IMPACT

The first prototype of OBD system, which was born in the seventies, was thought to control the emission – related data in a vehicle. The principal idea was to know, as soon as possible, when a fault of this system occurred and to signal it, turning on a led (MIL). The objective was to alert the user, which had to return hastily to the machine shop to repair the problem.

With the power evolution of the microcontrollers, joint with the dimensions shrinking, OBD started to develop and concern to other systems of the vehicle; this fact allows to obtain the nowadays systems, which for example, through the controlling of several parameters, estimate the tiredness of the driver, and advises that it is worthwhile do a halt.

Furthermore, above all in the utility cars, it is spread the *Start and Stop* system: when a car does a stop, for any reason (like a semaphore or for leave get down a passenger), the engine switches off; just when the user pushes on the accelerator the engine starts again. This practice is always connected with the OBD system, that processes the data present on the CAN bus.

The reader can conclude how, in the perspective of air pollution, OBD has introduced a revealing enhancement: since the dawning the objective was the lowering of the emissions and with the evolving the problem has become even much important.

Another important improvement derives from the time saving: now, with the help of OBD, when a vehicle arrives in a machine shop, the mechanic knows the exact problems of the latter, through the DTCs. Some time ago, in the same situation, the mechanical made trust only on his experience.

CONCLUSIONS

A PC application has been developed for communication with an automotive OBD-II system based on CAN bus. This application is made with the purpose of being modular. Indeed, in the main form there are also the buttons for the modes that are not implemented in the OZEN ELEKTRONIK device. This means that when the user will want to realize modes 5, 6 and 8, he musts simply add a new form to the application and develop the code that executes the correct algorithm. It is not necessary to operate in the other parts of the application, just because it is modular.

The same regarding the PIDs: only a few PIDs are supported by the different ECUs. If a user is interested to extend the code for all the available PIDs just has to add the specific code in the *select case* structure of the *requestPID* method.

This “modularity” is allowed because, for the entire project, the SAE J1979 standard was followed. However, the compatibility with all the OBD-II system is not ensured: there are some modes that have details that are not completely in compliance with the standard.

For example, in modes 2 and 9, if the application tries to ask the supported PIDs for the other groups as well as the first (even if it is a useless operation, because there is not any supported PIDs in these groups), the emulator responses with several error messages, and after that, it does not work anymore. To make this imperfection insignificant, there is a parameter that permits to specify “how many groups” ask to the device (the maximum value parameter), therefore it is enough to change a parameter to solve the problem.

Another unconformity regards the identifier for the request messages. The application cannot obtain a data from a specific ECU: if it wants a PID, it musts to request it with a broadcast message to all the units; in consequence, when a PID is supported by all the ECUs, the messages that arrive are as many as the number of the latter. The standard, on the other hand, tells to use the specific identifier for the request message at the ECU, and doing this, only that ECU will response. Even for this problem, it is enough change a parameter (the identifier) of the CAN message, in the case of the program is used with another OBD-II system.

In conclusion, if this application were connected to a vehicle (built after 2008), the results should be quite similar at those obtained with the OZEN ELEKTRONIK simulator device. However, this unconformity makes this task more elaborate.

ANNEXES

Annex A – OE91C1610 Data Sheet

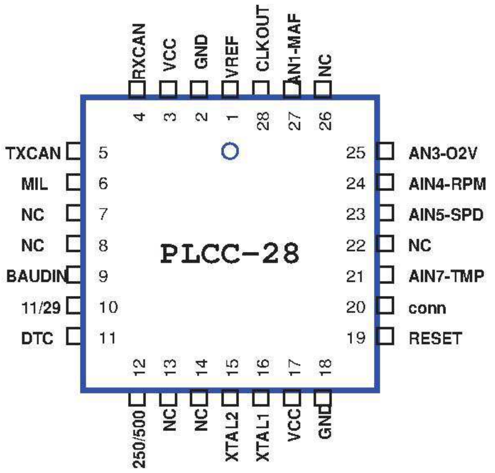
Below there are the data sheets about the OBD-II simulator, the MobyDic 1610 provided of the OE91C1610 controller by OZEN ELEKTRONIK. These sheets are useful because contain information about the supported PIDs for each modes, with other technical data.

Features

- Compatible with ISO 15765-4
- 2.7 to 6V operating range
- MIL LED output
- 11/29 bit ident selectable
- variable PIDs
- ECM,TCM and ABS ECU Address range
- 250/500 Kbaud selectable

Description

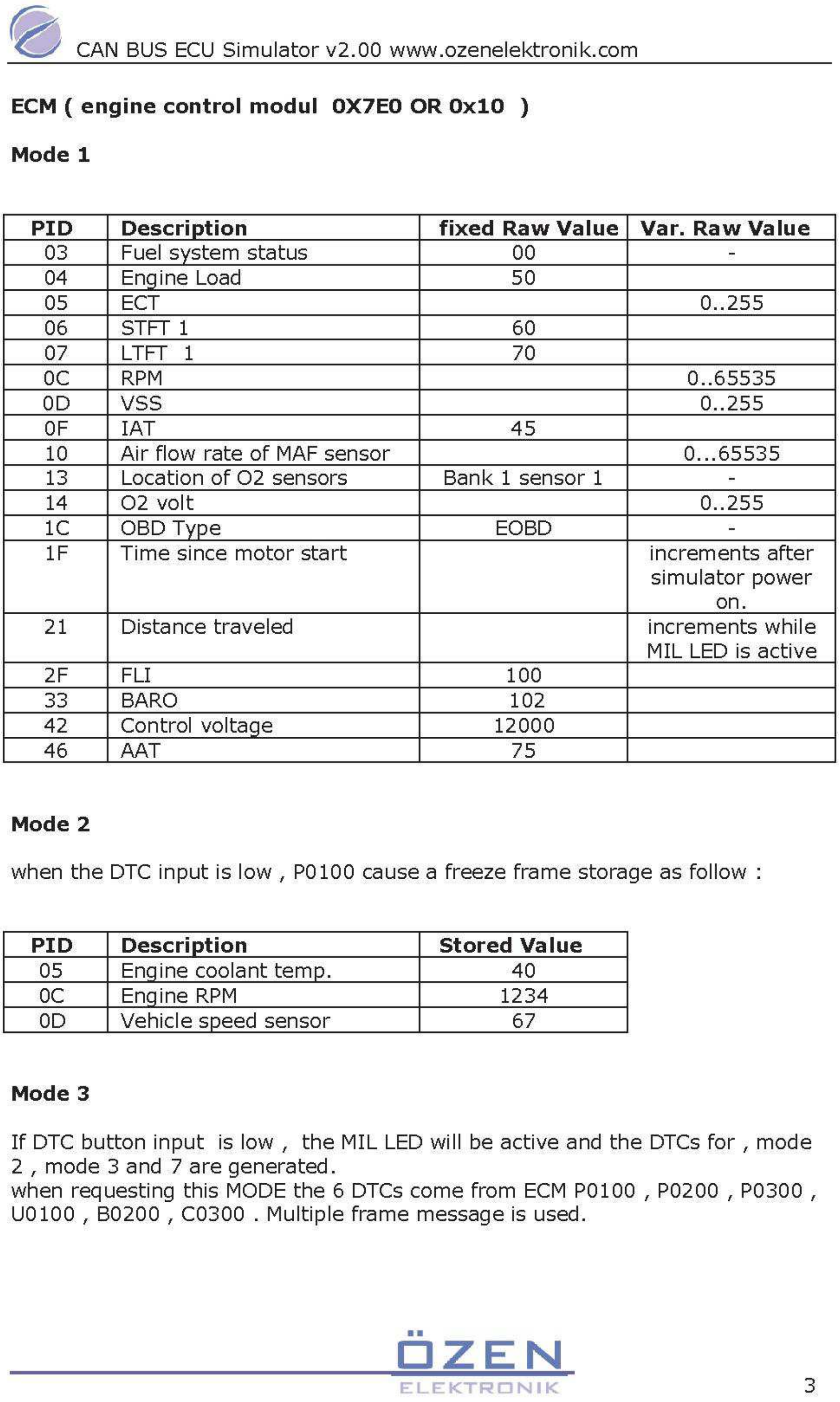
OE91C1610 can simulate ECM (engine ECU address 0x7E0) , TCM (transmission control ECU address 0x7E1) and ABS ECU address 0x7E2 simultaneously.
It is compatible with OE90C1600. EOBD modes (1,2,3,4,7,9) are implemented. Each ECU has own PID table and variable PID can be changed via potentiometers. The ECM can generate more than 3 DTCs.
The OE91C1610 communicate at 250/500 Kbaud with 11/29 bit ident



ÖZEN
ELEKTRONIK

CAN BUS
MULTIPLE ECU
simulator according
to ISO15765-4

OE91C1610



CAN BUS ECU Simulator v2.00 www.ozenelektronik.com

ECM (engine control modul 0X7E0 OR 0x10)

Mode 1

PID	Description	fixed Raw Value	Var. Raw Value
03	Fuel system status	00	-
04	Engine Load	50	
05	ECT		0..255
06	STFT 1	60	
07	LTFT 1	70	
0C	RPM		0..65535
0D	VSS		0..255
0F	IAT	45	
10	Air flow rate of MAF sensor		0...65535
13	Location of O2 sensors	Bank 1 sensor 1	-
14	O2 volt		0..255
1C	OBD Type	EOBD	-
1F	Time since motor start		increments after simulator power on.
21	Distance traveled		increments while MIL LED is active
2F	FLI	100	
33	BARO	102	
42	Control voltage	12000	
46	AAT	75	

Mode 2

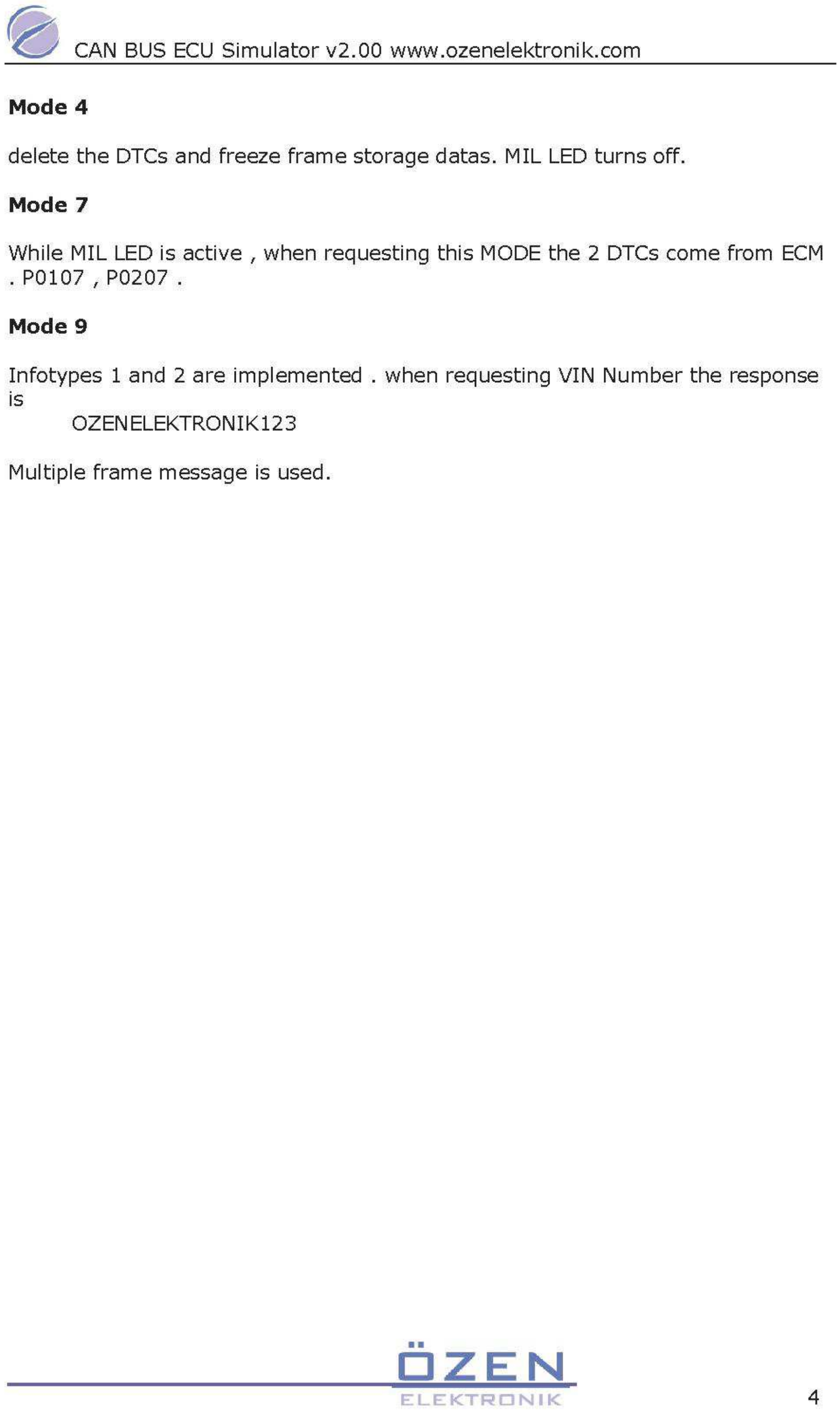
when the DTC input is low , P0100 cause a freeze frame storage as follow :

PID	Description	Stored Value
05	Engine coolant temp.	40
0C	Engine RPM	1234
0D	Vehicle speed sensor	67

Mode 3

If DTC button input is low , the MIL LED will be active and the DTCs for , mode 2 , mode 3 and 7 are generated.
when requesting this MODE the 6 DTCs come from ECM P0100 , P0200 , P0300 , U0100 , B0200 , C0300 . Multiple frame message is used.

ÖZEN ELEKTRONIK 3



Mode 4

delete the DTCs and freeze frame storage datas. MIL LED turns off.

Mode 7

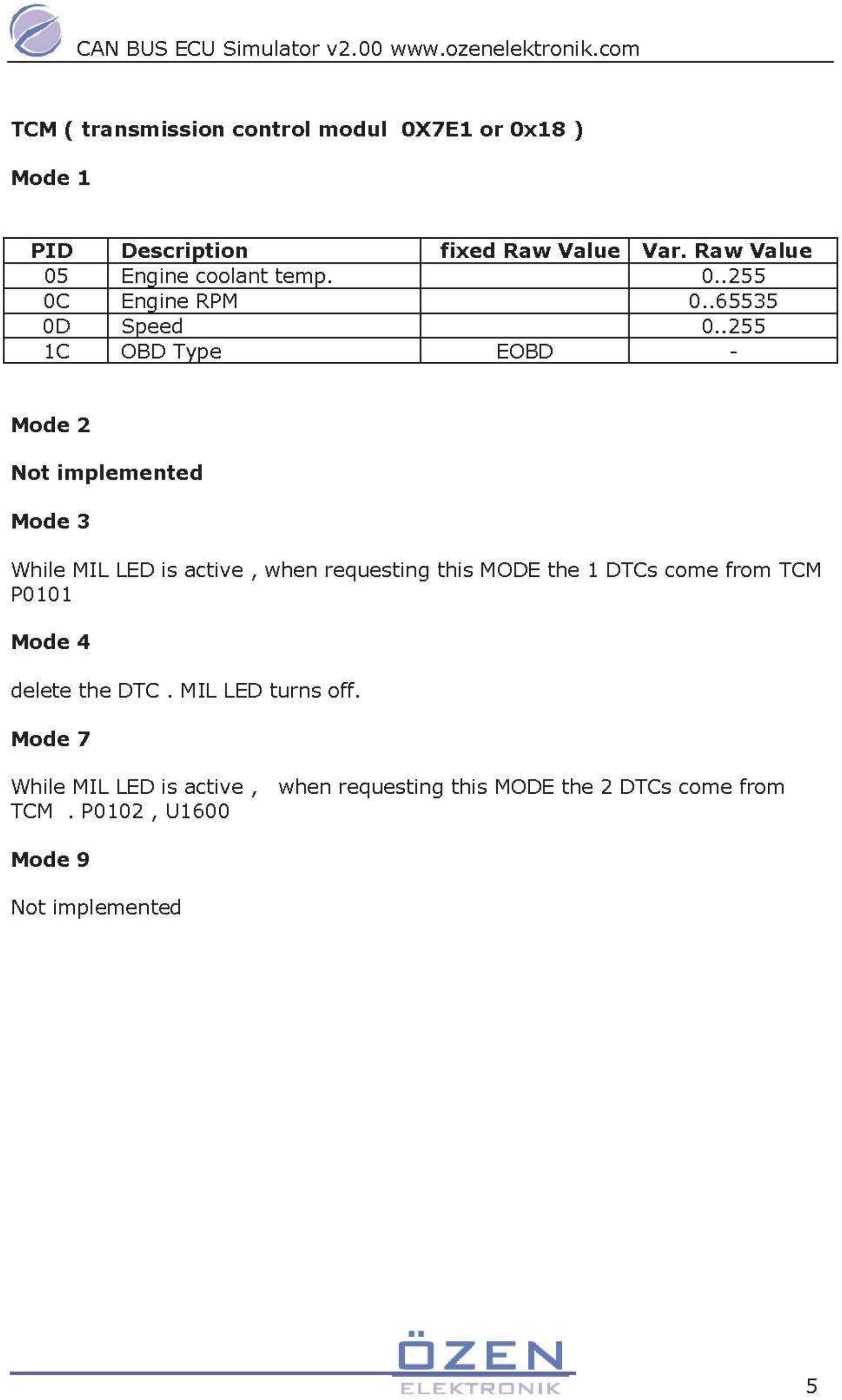
While MIL LED is active , when requesting this MODE the 2 DTCs come from ECM . P0107 , P0207 .

Mode 9

Infotypes 1 and 2 are implemented . when requesting VIN Number the response is

OZENELEKTRONIK123

Multiple frame message is used.



TCM (transmission control modul 0X7E1 or 0x18)

Mode 1

PID	Description	fixed Raw Value	Var. Raw Value
05	Engine coolant temp.		0..255
0C	Engine RPM		0..65535
0D	Speed		0..255
1C	OBD Type	EOBD	-

Mode 2

Not implemented

Mode 3

While MIL LED is active , when requesting this MODE the 1 DTCs come from TCM P0101

Mode 4

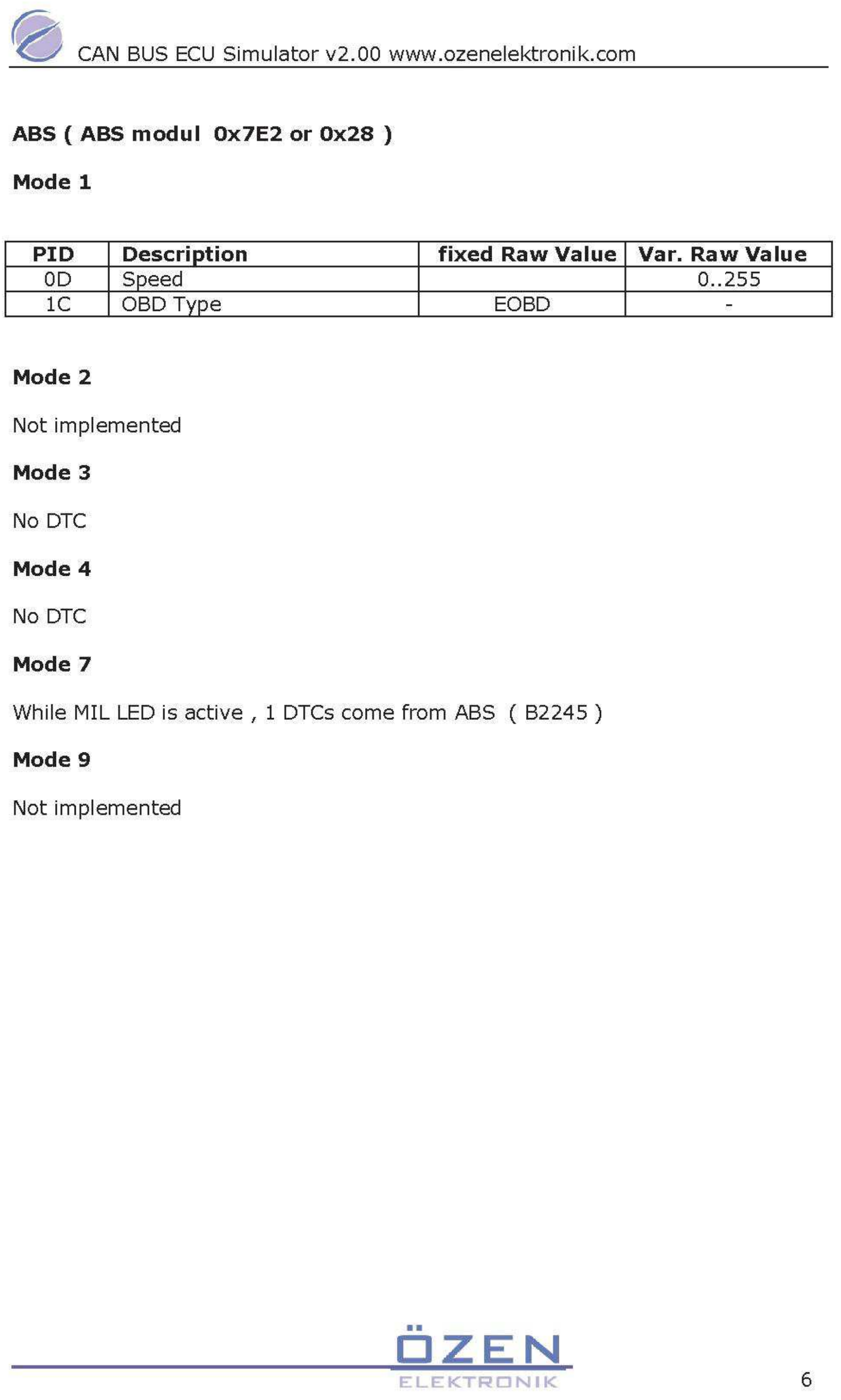
delete the DTC . MIL LED turns off.

Mode 7

While MIL LED is active ,   when requesting this MODE the 2 DTCs come from TCM . P0102 , U1600

Mode 9

Not implemented



ABS (ABS modul 0x7E2 or 0x28)

Mode 1

PID	Description	fixed Raw Value	Var. Raw Value
0D	Speed		0..255
1C	OBD Type	EOBD	-

Mode 2

Not implemented

Mode 3

No DTC

Mode 4

No DTC

Mode 7

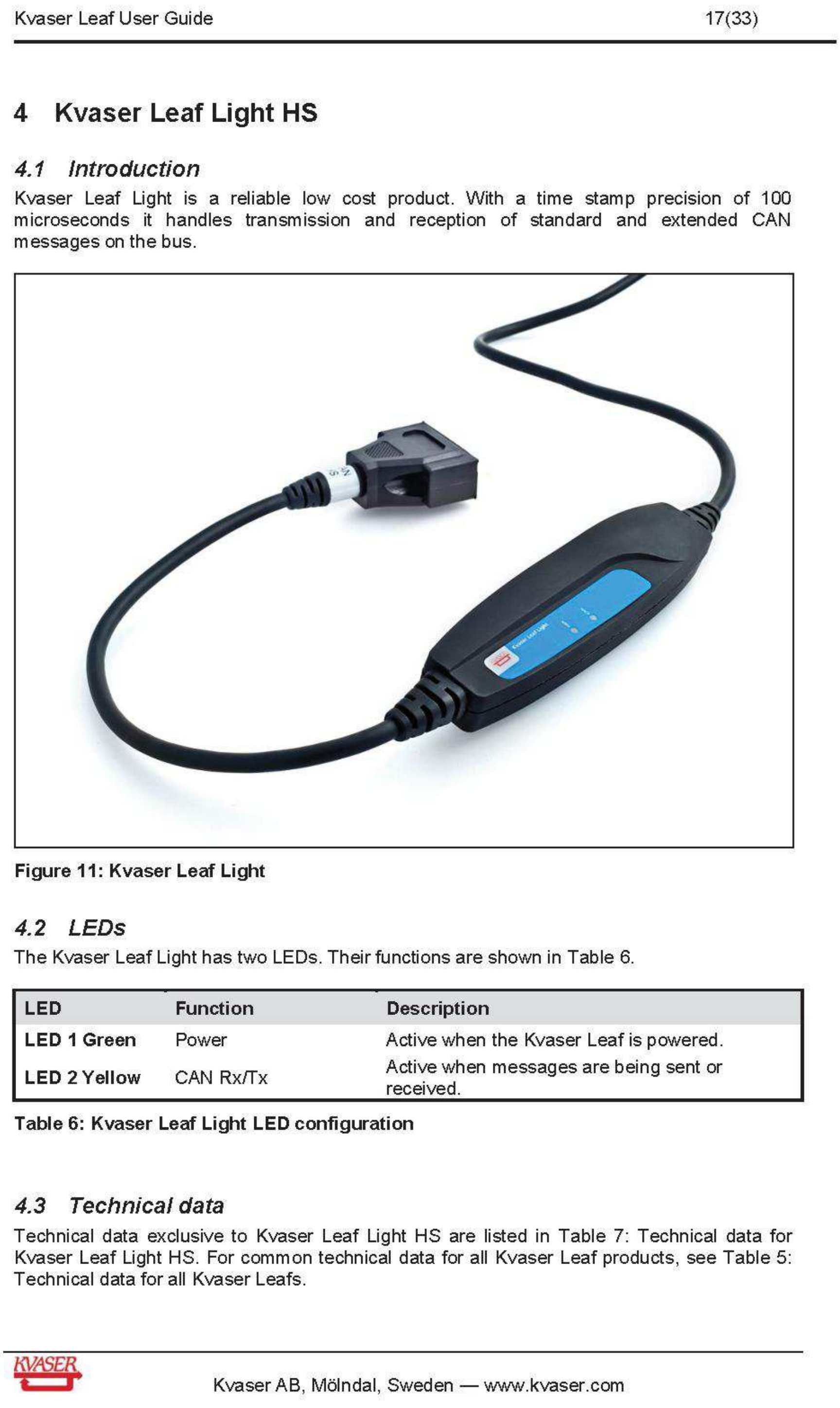
While MIL LED is active , 1 DTCs come from ABS (B2245)

Mode 9

Not implemented

Annex B – Kvaser Leaf Light HS Data Sheet

Here there are the data sheets about the Kvaser CAN – USB converter. There is information about the technical data of the product.



Kvaser Leaf User Guide 17(33)

4 Kvaser Leaf Light HS

4.1 Introduction

Kvaser Leaf Light is a reliable low cost product. With a time stamp precision of 100 microseconds it handles transmission and reception of standard and extended CAN messages on the bus.

Figure 11: Kvaser Leaf Light

4.2 LEDs

The Kvaser Leaf Light has two LEDs. Their functions are shown in Table 6.

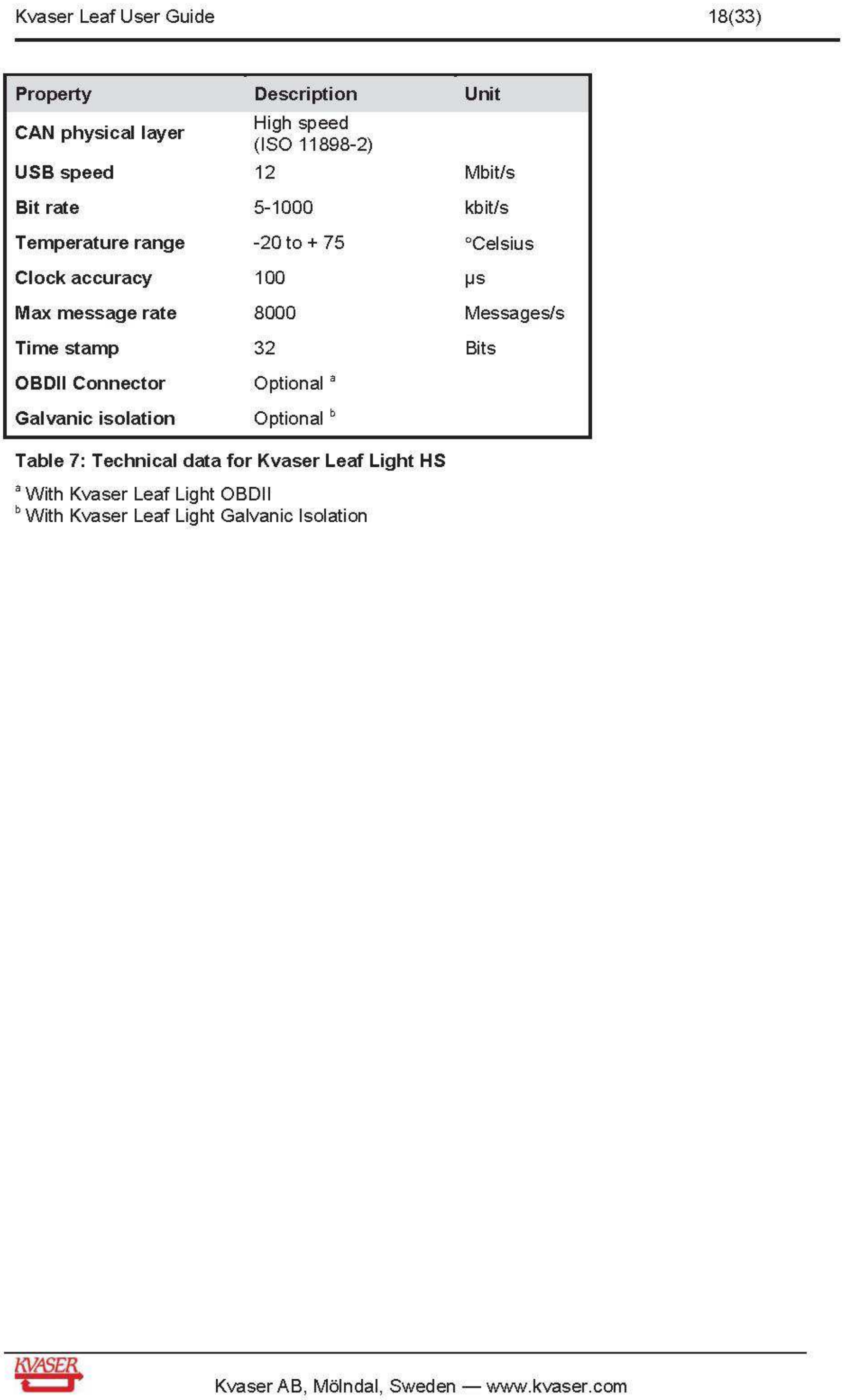
LED	Function	Description
LED 1 Green	Power	Active when the Kvaser Leaf is powered.
LED 2 Yellow	CAN Rx/Tx	Active when messages are being sent or received.

Table 6: Kvaser Leaf Light LED configuration

4.3 Technical data

Technical data exclusive to Kvaser Leaf Light HS are listed in Table 7: Technical data for Kvaser Leaf Light HS. For common technical data for all Kvaser Leaf products, see Table 5: Technical data for all Kvaser Leafs.

Kvaser AB, Mölndal, Sweden — www.kvaser.com



Kvaser Leaf User Guide 18(33)

Property	Description	Unit
CAN physical layer	High speed (ISO 11898-2)	
USB speed	12	Mbit/s
Bit rate	5-1000	kbit/s
Temperature range	-20 to + 75	°Celsius
Clock accuracy	100	μs
Max message rate	8000	Messages/s
Time stamp	32	Bits
OBDII Connector	Optional [a]	
Galvanic isolation	Optional [b]	

Table 7: Technical data for Kvaser Leaf Light HS

[a] With Kvaser Leaf Light OBDII

[b] With Kvaser Leaf Light Galvanic Isolation

Kvaser AB, Mölndal, Sweden — www.kvaser.com

RINGRAZIAMENTI

A conclusione di sei anni di università, credo che alcuni ringraziamenti siano doverosi.

Mamma e Papà: siete le persone che hanno permesso tutto questo: questa Laurea, questa tesi, l'Erasmus nel quale è stato sviluppato questo progetto, e mille altre cose. C'è un detto in cui credo fermamente, e che spesso ripeto: "La mela non cade mai lontana dall'albero". Spero tanto di esserne un chiaro esempio sempre, sarebbe per me la più grande soddisfazione, nonché la maniera migliore per ricordare quello che avete fatto per me. Grazie.

Maura, Marina, Lorenzo, Gianluca e Davide: due sorelle, due cognati e un nipote. Non è una barzelletta, è parte di una famiglia senza la quale tutto sarebbe più difficile. Chi con un sorriso e qualche parolina, chi con la presenza, chi con la vicinanza, tutti avete contribuito a questo successo. Anche se non ve lo dimostro mai, sappiate che vi voglio bene.

Prof. Eguilaz e Prof. Vitturi: due professionisti nel loro campo. Competenza e supporto non sono mai venuti meno, tanto quanto la disponibilità. Grazie.

Anna e Irene: a persone come voi non è abbastanza augurare ogni bene. Perché avete avuto in ogni momento un sorriso, una battuta, una carezza. Anche solo per avermi ascoltato tante volte, non sapete quanto io vi sia grato. Figuratevi per tutto il resto che avete fatto.

Michael e Adriano: siete due Amici. O quattro, dipende da quanti gin tonic abbiamo bevuto! È chiaro che i ritmi che abbiamo mantenuto durante la vita universitaria non saranno sostenibili ancora a lungo, ma perlomeno, non perdiamoci di vista. Mi raccomando, ci conto.

I coinquilini: in generale, a tutti: grazie per aver sopportato il mio carattere difficile. Sono ben consapevole di non essere un "tipo facile", avete avuto tutti indubbiamente più pazienza di me! A Katiuscia e Laura: le mie due psicologhe. Mi avete fatto ridere un sacco, via Fortin la ricorderò perché c'eravate voi. Alberto: non mi ricordo un tuo "no" alla domanda "stasera festa?". I no casomai li dicevo io, purtroppo. Sappi però che a far festa con te non ci si annoia mai. Ramona e Chiara: avere due ragazze così a casa, con cui parlare in ogni momento, è una fortuna per pochi. Io l'ho avuta, non c'è dubbio. So che davanti a mille difficoltà andrete sempre avanti col sorriso, e lo farete perché il vostro è uno di quei sorrisi talmente spontanei da non potersi spegnere mai.

I ragazzi dell’erasmus: Michela, Luca, Stefano, Sabrina e Marco. Voi sapete quanto un Erasmus sia indescrivibile. Voi fate parte di quel ricordo straordinario. Spero che concordiate con me nel dire che un’esperienza così ti segna la vita. Tornerei a quei mesi all’istante, questo è anche merito vostro.

Ilaria: ti ho sempre detto che invidio il tuo ottimismo e la tua spensieratezza. Mi hai dato delle intere lezioni a proposito, dimostrandoti una persona incantevole. Avrò sempre un ricordo meraviglioso di te.

Max Erre e Fabio: due Amici da una vita. Due Amici che posso anche vedere poco, ma che sono sicuro che ci saranno, sempre.

Ho scritto di getto queste righe. Senza voler dare un ordine di importanza, sicuramente dimenticando qualcuno, sicuramente omettendo qualcuno. In ogni caso, a tutti voi che avete partecipato a questa avventura, un sincero grazie.

BIBLIOGRAPHY

[1] WIKIPEDIA – THE FREE ENCYCLOPEDIA. *On-board diagnostics*. July 2009

[http://en.wikipedia.org/wiki/On-board_diagnostics] [URL, 08 – 05 - 2012]

[2] KVASER – ADVANCED CAN SOLUTIONS. *Kvaser Leaf Light HS.*

[www.kvaser.com/prod/hardware/leaf_light.htm] [URL, 10 – 05 - 2012]

[3] KVASER – ADVANCED CAN SOLUTIONS. *Kvaser OBD II Adapter Cable*.

[http://www.kvaser.com/index.php?option=com_php&Itemid=303&eaninput=7330130003026&lang=en&product=Kvaser%20OBD%20II%20Adapter%20Cable]

[URL, 11 – 05 - 2012]

[4] DRAFT INTERNATIONAL STANDARD ISO/DIS 15765-4.2. *Road vehicles - Diagnostics on Controller Area Networks (CAN) - Part 4: Requirements for emissions-related Systems*. February 2003.

[5] SAE SURFACE VEHICLE STANDARD J1979. *Diagnostic Test Modes*. April 2002.

[6] KVASER – ADVANCED CAN SOLUTIONS. *Price list of Kvaser product.*

[https://secure.assurebuy.com/98057/98057.htm] [URL, 23 – 05 - 2012]

[7] OZEN ELEKTRONIK. *Price list of Ozen Elektronik device*.

[http://www.ozenelektronik.com/?s=products2&group=eobd-obdii-ecu-simulators]

[URL, 23 – 05 - 2012]

[8] MICROSOFT STORE. *Price of Visual Studio 2010 Professional license*.

[http://microsoftstore.it/shop/it-IT/Microsoft/Visual-Studio-2010-Professional]

[URL, 23 – 05 - 2012]