TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Systems

Lev Avdejev 093455IASB

SOFTWARE FOR WORKING WITH AUDIO AMPLIFIER'S EEPROM

Bachelor's thesis

Supervisor: Viktor Leppikson
PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Lev Avdejev 093455IASB

TARKVARA TÖÖTAMISEKS HELIVÕIMENDI EEPROMIGA

Bakalaureusetöö

Juhendaja: Viktor Leppikson
PhD

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Lev Avdejev

22.05.2017

Abstract

The primary objective of the current thesis is to design and implement a special software application equipped with a graphical user interface for reading, changing the EEPROM content and also saving the initial memory state of car audio amplifiers.

Functionality of the tool is reliable independently of the audio amplifier type and software version running on it. Communication takes place through RS232 interface completed by operating system specific L2 and L3 protocols. Input data for requesting amplifier EEPROM memory records is acquired from database, which is in Excel format. In parallel, it is possible to read and modify the content and also save the initial state of the EEPROM memory. Additionally, application provides the possibility to monitor and pertain serial communication traffic. Graphical user interface is easily and uniquely understandable, robust. Additionally, it includes tool-tips and provides log history and support to avoid any possible incorrect usage.

The fully functional and tested application with GUI was developed using C# language. All problems, decisions and functional details are thoroughly described in the thesis. Functionality is demonstrated with picture material. In addition, the thesis also includes an user-guide.

The thesis is in English and contains 60 pages of text, 8 chapters, 14 figures, 5 tables.

Abstract

Tarkvara töötamiseks helivõimendi EEPROMiga

Käesoleva bakaleureusetöö põhieesmärgiks on disainida ning realiseerida spetsiaalne graafilise kasutajaliidesega tarkvara auto helivõimendi EEPROM mälu esialgse seisu salvestamiseks, andmete väljalugemiseks ning nende muutmiseks.

Disainitud ning arendatud tarkvara on kasutatav olenemata helivõimendi tüübist ning seal jooksvast tarkvara versioonist. Kommunikatsioon toimub läbi RS232 liidesse, ent sellele lisanduvad operatsioonisüsteemi spetsiifilised L2 ja L3 kihi protokollid. Sisendandmed väljaloetavate andmete küsimiseks tulenevad Excel formaadis olevast andmebaasist. Paralleelselt on võimalik mälus olevaid andmeid lugeda, muuta ning salvestada mälu esialgne olek selle hilisemaks taastamiseks. Loodud tarkvara on piisavalt robustne ning tagab ülejäänud EEPROM mälus olevate andmete muutumatus tööprotsessi jooksul. Lisafunktsionaalsusena on võimalik andmeliikluse monitoorimine ning salvestamine RS232 kanalil. Kasutamise lihtsustamiseks koostatud graafiline kasutajaliides on lihtsasti ning üheselt arusaadav, veakindel, sisaldab juhtnööre, on võimeline salvestama nii õnnestunud kui ebaõnnestunud kasutusjuhtusid. Sellele lisaks abistab tööriist kasutajat veasituatsioonides ning keelab selle väärkasutust.

Töö tulemusena valmis töötav C# programmeerimiskeeles arendatud graafilise kasutajaliidesega tarkvara, mis täidab kõik püstitatud eesmärgid. Kõikide probleemide lahendused, langetatud valikud ning funktsionaalsuse detailid on ammendavalt lahti seletatud. Tarkvara töötamine on tõestatud pildimaterjaliga. Koostatud on ka kasutusjuhend.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 60 leheküljel, 8 peatükki, 14 joonist, 5 tabelit.

List of abbreviations and terms

DUT	Device Under Test
EEPROM	Electrically Erasable Programmable Read-Only Memory
GUI	Graphical User Interface
WPF	Windows Presentation Foundation
XAML	Extensive Application Markup Language
AM	Amplitude Modulation
SW	Software
HW	Hardware
MOST	Media Oriented System Transport
L2	Layer 2
L3	Layer 3
MCU	Microcontroller
Rx	Receiver
Tx	Transmitter
Bps	Bytes per second
CRC	Cyclic Redundancy Check
KB	Kilobytes
CSV	Comma Separated Values
SWDL	Software Download
ANC	Active Noise Cancellation
EOL	End Of Line

Table of Contents

1 Introduction....................11
2 Development initiative....................13
2.1 General problem overview....................13
2.2 Functional requirements....................14
2.3 Communication with the amplifiers....................15
2.3.1 RS232 protocol....................15
2.3.2 L2 and L3 protocols....................16
2.3.3 Formats of the request and response messages....................17
2.4 Audio amplifier EEPROM memory....................18
2.5 Input database....................18
2.5.1 Layout....................18
2.5.2 Choosing the Excel parsing framework....................19
2.6 Working with the memory content possibilities....................20
2.7 Output files....................20
2.8 Solution proposal....................21
3 Design and implementation....................23
3.1 Serial port initialization....................23
3.2 Logging functionality....................24
3.2.1 Online....................25
3.2.2 To file....................26
3.3 Reading input from database....................26
3.3.1 Using the BackgroundWorker....................27
3.3.2 Searching specific records....................29
3.3.3 Result structure parameter description....................31
3.4 Saving initial EEPROM state....................33
3.4.1 Purpose....................33
3.4.2 Workflow....................34
3.4.3 Supporting tools....................37

3.4.4 Output....37
3.5 Reading and modifying the records....37
3.5.1 Building and sending the request commands....38
3.5.2 Parsing response and displaying result....38
3.5.3 Changing the values of records....39
3.6 History of tool usage....41
3.7 Data safety....41
4 Graphical user interface....43
4.1 Design....43
4.1.1 WinForms....43
4.1.2 WPF....44
4.1.3 Graphical solution....45
4.2 Content overview....47
4.3 Tooltips....49
4.4 Restrictions....51
5 User-manual....53
6 Testing....55
7 Future development....57
8 Summary....58
References....60

List of Figures

Figure 1: Simplified RS232 connection....15
Figure 2: L2 command example....18
Figure 3: System overview....22
Figure 4: Activated progress-bar....27
Figure 5: Parameters extraction functionality....30
Figure 6: CsvRecord class....32
Figure 7: Readout started notification....34
Figure 8: Readout finished notification....34
Figure 9: CsvRecord class....35
Figure 10: CSV generator loop....36
Figure 11: CSV content....36
Figure 12: Graphical user interface....46
Figure 13: Displayed tooltip....51
Figure 14: User-manual....54

List of Tables

Table 1: L2 telegram structure......15
Table 2: L3 telegram structure......16
Table 3: Advantages and disadvantages of WinForms......43
Table 4: Advantages and disadvantages of WPF......43
Table 5: Displayed tooltips......48

1 Introduction

Most of the systems or devices are equipped with an internal or external EEPROM memory, where different types of information is stored. The data formatted as certain memory records are read, written and modified during the normal operation of the device. Normally, due to security reasons it is not possible or allowed to access memory contents by an external tools. However, under some circumstances the need for accessing and either reading or changing the memory content can eventually become necessary.

For example, accessing and changing the EEPROM records might become required to clarify which values triggered an inaccurate behaviour of the device. Alternatively, it can be needed to achieve exact preconditions for testing specific software or hardware functionality. Normally it can be done manually by requesting the content from the amplifier with certain commands. However, nowadays such manual approach is not an option any more.

The gain of this thesis was to develop a special software application which provides such functional possibility with the external audio amplifier EEPROM memory of a car. In order to simplify the usage, the application is supplied with a self-explaining graphical user interface (further on GUI). The required input information for building the request commands is acquired from a relatively large database. This database is in Excel format and is given as the input parameter to the application. Communication with the target takes place through the RS232 serial interface and allows reading and modifying the contents of the EEPROM memory records, back-up initial state of the memory for recovering the start point and monitor the serial traffic. The software is written in C# programming language and Visual Studio 2013 Express development environment. The GUI is built by using the WPF and XAML frameworks.

The thesis is divided into 8 paragraphs. It starts with the general introduction. The second paragraph gives an overview of the problem and primary reasons why develop-

ment of the EEPROM configuration tool was initiated. The third section lists and describes the preconditions and input for the application. The next chapter provides detailed description of the implementation together with source code examples and explanations of different decisions. The fifth paragraph gives a short comparison of the WinForms and WPF, introduces the GUI and describes its elements. The further chapter focuses on giving an overview of general testing activities during the development of the tool and before releasing it towards the customer. The seventh chapter talks about further development ideas and possibilities. The final paragraph summarizes the tool, analyses whether the initial target was fulfilled and gives a final estimation for the thesis.

2 Development initiative

2.1 General problem overview

The automotive industry is nowadays one of the largest and most important industries in the world. First occurrences of in-car audio started appearing approximately in the 1930s, when the first in-car monophonic AM radio got introduced [1] . Since then, the audio has experienced a rapid development and become one of the most complicated and advanced systems installed into a vehicle. An increase in complexity has had a major impact on the quality of audio systems. Although all devices, including audio amplifiers, go through multi-level and thorough testing during the software and hardware development phases, integration and release phases, before and after manufacturing and also at the customer, it is very difficult to create a flawlessly functional device. Even after going through and passing such a difficult and multi-layered testing approach, some errors still tend to pop into reality when the system is being already used in the car.

The need for possibility to access the EEPROM memory content with an external application initially evolved from the production factory of the audio amplifier. Multiple samples of the product were returned by the customer with a claim indicating to incorrect functionality under certain conditions. In order to achieve such conditions and to be able to reproduce the error scenario it became inevitable to modify the content of specific data records in the EEPROM memory. At the beginning it was a question whether to allow such a possibility or not. The main reason was the fact that the amplifier had to go be delivered back to the customer unchanged. This means the content had to remain exactly the same as at the point of receiving. As a result, a new situation was introduced – a backup of the EEPROM image became unavoidable.

Eventually, it was decided to design such software application with a graphical user interface capable of fulfilling these needs without having any impact on the other data stored in the EEPROM memory.

2.2 Functional requirements

The functional requirements were defined in cooperation between the software development team of the audio amplifier and the production team after several discussions. The production requested following points:

- The software application allows reading and changing every EEPROM memory record of the audio amplifier.
- The software application provides possibility to generate a copy of the current EEPROM memory state.
- Generated EEPROM memory mask can be directly written back to the audio amplifier.

The software team of the amplifier decided to extend the list of functional requirements with:

- The functionality must not have any impact on the other EEPROM memory records.
- The software application acquires the input from the Excel database automatically.
- The software application is capable of displaying and saving the RS232 serial communication log with the amplifier.
- The software application is equipped with a well-organized and simple GUI.
- Each changeable option must be presented on the GUI with drop-down menus.
- The GUI must include tooltips providing instructions about the usage.

- Steps of the usage and work progress of the software application must be logged into a file.
- The developed tool must not require SW changes of the audio amplifier.

2.3 Communication with the amplifiers

The amplifier can be communicated to through two different external interfaces – MOST and RS232. MOST stands for Media Oriented Systems Transport and was developed by MOST Cooperative. It is widely used for communication with the other devices located on the MOST ring. This protocol is the primary interface in the Automotive industry. The serial communication is mainly used during the development and production phases of the amplifiers.

2.3.1 RS232 protocol

The communication through the RS232 serial port remained as the only communication possibility between the developed application and the audio amplifier. This fact made knowing this protocol inevitable.

RS232 is an asynchronous serial communication protocol, which is widely used as a linkage between the computer and its peripheral MCU device. The protocol consists of two data lines called Receiver (Rx) and Transmitter (Tx). Tx is a line which is used by one device to transmit the data to the counterpart device whereas Rx is used by the counterpart device to send its response. Certainly, it can also be the other way around.

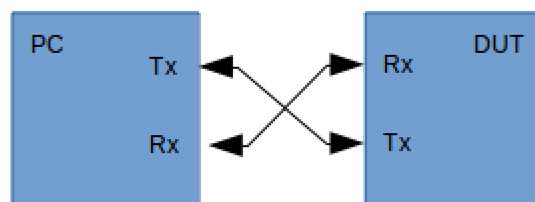

Figure 1: Simplified RS232 connection

The data itself is transmitted as a whole unit one bit at a time. This fact separates the RS232 communication from the parallel communication, where whole the data unit is transmitted at once.

Taking into account the fact of missing synchronization option, the serial communication requires to be carried out with specified standard speeds. This speed is defined as the baud rate and indicates to how many bits are transmitted per second. The standard baud rates are 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200 bps. The determined baud rate between the audio amplifier and the SW application is 115200 bps.

2.3.2 L2 and L3 protocols

The data bytes that are sent and received through the RS232 communication interface must correspond to the L2 and L3 protocols. These protocols are common for systems using an operating system. Current audio amplifier uses uCOS-II for it. It is a pre-emptive, multi-branched and well portable real-time operating system developed by company called Micrium. It is widely used in the error-critical real-time systems, because of the simple structure and independence of the hardware components. Apart from that, this operating system is resource-efficient and reliable [2] .

The L2 protocol is known as the outer layer which is wrapped around the L3 protocol. It interconnects the hardware components with physical addressing and consists of 3 bytes long header followed by the payload field. The structure is described in the following table.

Table 1: L2 telegram structure

L2 byte offset	Description
0x0	L2 protocol identifier with fixed value 0x1B.
0x1	L2 command type identifier. Can be either of control or data type.
0x2	L2 payload length
0x3 – N	L2 payload containing L3 telegram
(N - 1) – N	L2 message CRC

The L3 protocol connects the logical components of the amplifier software. Any module of the software can be counted as a logical component, which communicates with the other components by using the L3 messages. The structure of the L3 message is already more complex. It consists of 6 bytes long header describing the protocol and additionally the payload. The protocol bytes together with the descriptions are illustrated in the table below.

Table 2: L3 telegram structure

L3 byte offset	Description
0x0	Protocol version and flags
0x1	L3 telegram destination address
0x2	L3 telegram destination sub-address
0x3	L3 telegram source address
0x4	L3 telegram source sub-address
0x5	L3 payload length
0x6 – 0x7	L3 telegram identifier
0x8 – N	L3 payload

2.3.3 Formats of the request and response messages

Any data that must be acquired from the amplifier needs to be done through the RS232 interface with a telegram. Every telegram must follow the L2 and L3 protocols mentioned in the paragraph 2.3.2. They must start with a special L2 start identifier byte 0x1B and end with 2 bytes of CRC, which is calculated over the complete L2 message excluding the start-byte.

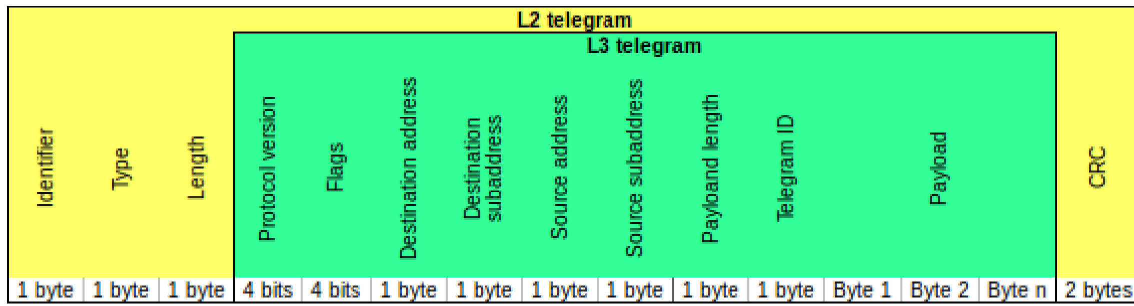

Figure 2: L2 command example

2.4 Audio amplifier EEPROM memory

The amplifier is equipped with a 4 KB large EEPROM memory. It consists of 128 pages of 32 bytes and each page is divided into two half-pages respectively. There is no size limitations for the data length of the memory records. Each of the entries can be located anywhere within the memory. The record can also be split onto two different pages or half-pages. The EEPROM memory can be accessed by a feature called Variant Handler. Every record is equipped with a special identification number called Variant Handler identifier (further on VH ID). This ID must be used for requesting the data from the memory by the rest of the system or an external device.

The Variant Handler is implemented in a way that any request made through this feature is responded with the full half-page, starting from the data bytes of the requested memory record. This means that the response does not contain only the requested data. Due to this, the correct content of the EEPROM memory records must be filtered out by any external device using the VH.

2.5 Input database

2.5.1 Layout

The database is built into the Excel spreadsheet document. Different types of information such as serial numbers, HW and SW versions, repair shop-codes and ANC switcher is described in it. It reflects the exact content of the EEPROM and Flash memory of the audio amplifier.

The database is divided into two sections. The first part contains the actual data records together with their specifications, parameters, description and the initial values. The second one serves a visualizing function. Its purpose is to illustrate how much space is already allocated and also show the exact location of all entries in the EEPROM and Flash memories. The data section is divided into platform and project specific parts. The platform section is common for every project. The customer specific part encloses values and records that belong to a certain project only. The following formula becomes obvious when taking the above-mentioned information into account: project data = platform data + customer data. A separate sheet is segregated for each customer.

2.5.2 Choosing the Excel parsing framework

Due to the fact that the database is in Excel and it is the only input parameter for the developed software application, parsing the Excel spreadsheet becomes imminent. Implementing such functionality from scratch is time-consuming and requires decent effort. Hence, an appropriate spreadsheet parsing framework had to be chosen to comply the fixed deadlines.

As modifying the Excel spreadsheet was not required for this application, a framework called Excel Data Reader library was chosen. The decision was done mainly because this framework is sufficiently lightweight, also developed in C# programming language and relatively fast. The second strong argument for choosing this framework is author's positive experience with using it during the development of previous SW applications. Additionally, the Excel Data Reader is quite simple to use and the code of the database parsing is well-structured, easily maintainable and readable.

The workflow of the selected library is utterly self-explanatory. The library creates separate tables for each available sheet after reading the input Excel file. As the amplifier's memory database was exactly organized in a project-per-sheet format, it exactly fulfilled the needed objective. Such approach of converting the spreadsheets into the tables makes the parsing relatively straightforward as each table can be interpreted as a two-dimensional array. Finally, the Excel Data Reader library accesses the input Excel itself just once – meaning, it leaves the input file alone after the tables are generated. This brings a significant time advantage as the database is fairly large.

2.6 Working with the memory content possibilities

The interface between the outer world and the audio amplifier’s EEPROM memory supports two important functionalities. These are reading to and writing from the memory. Both operations are performed through the Variant Handler feature and strongly depend on the parameters specified in the database file.

For the user, the software application allowing a configuration possibility must provide exactly these operations. Every memory entry must be read out by using the unique VH ID and displayed for the user together with its explanation. The description must give information about the record length, name and type. The type can either be cached or not cached. In easy words, the cached type records are stored into the cache memory after applying the modifications instead of writing them immediately into the EEPROM memory. Such records will be actually written only after the amplifier has performed a full shutdown sequence. The same read-out needs to be used also for changing its value. The user has to obtain possibility to edit the data bytes directly through the user interface, without initializing another data request from the amplifier. Writing other number of bytes than specified in the database of a specific item must not be possible.

Due to the modification possibility, restoring the initial memory state after any change performed is inevitable. This is a strict requirement as any amplifier sent back to the production for analysis or repairing must be delivered back to the customer with the original memory content.

2.7 Output files

The software tool must be capable of creating several types of output files. They can be divided into three different sets according to their purpose.

The first set of files contains the content of the original EEPROM memory. It must be generated automatically and stored in different formats. Each of them serves a certain purpose and area of usage. The generated content must be directly usable and equipped with the correct flash area addresses. Without the addresses, the files can not be used by the other external tools in the software download process.

The second set of generated files are related to the serial communication. During the process all the traffic between the amplifier and the PC application must be logged online by displaying it on the screen and also saved into a file. This is mandatory in order to provide possibility to debug any misbehaviour of the amplifier while communicating with the tool.

The purpose of the third and final set of files is storing the history of the tool functionality. Any action performed with the application ought to be saved into a simple history file, which afterwards can be used for analysing different functionality errors and track the events that lead to such unexpected scenario.

2.8 Solution proposal

For a solution, the software team decided that developing such special software application for achieving the requested functionality is imminent. The tool will be used by the user only through the GUI to avoid any kind of unwanted usage. All necessary input parameters for requesting the EEPROM memory entries will be read from the given Excel database automatically. No selections other than providing the correct database file will be needed from the end-user. The communication between the application and the amplifier will take place through the RS232 serial communication interface. The tool will generate correct L2 commands for requesting the data of the record(s), calculate the CRC and send it to the amplifier. The response will be filtered only for the bytes which are relevant for the requested entry. Only the extracted content will be displayed to the user together with its mapped description. It will provide possibility to change the value if necessary. The possibility to save the initial state of the memory and generate usable EEPROM files will be solved as a one-click solution. The RS232 communication will be logged online in hexadecimal numeric system or ASCII format to the screen and also saved into a text file in parallel. Saving into the file must be activated by the user manually. The history of the application usage will be saved into a file automatically without asking any permission. All time-consuming processes will be implemented as a multi-thread functionality.

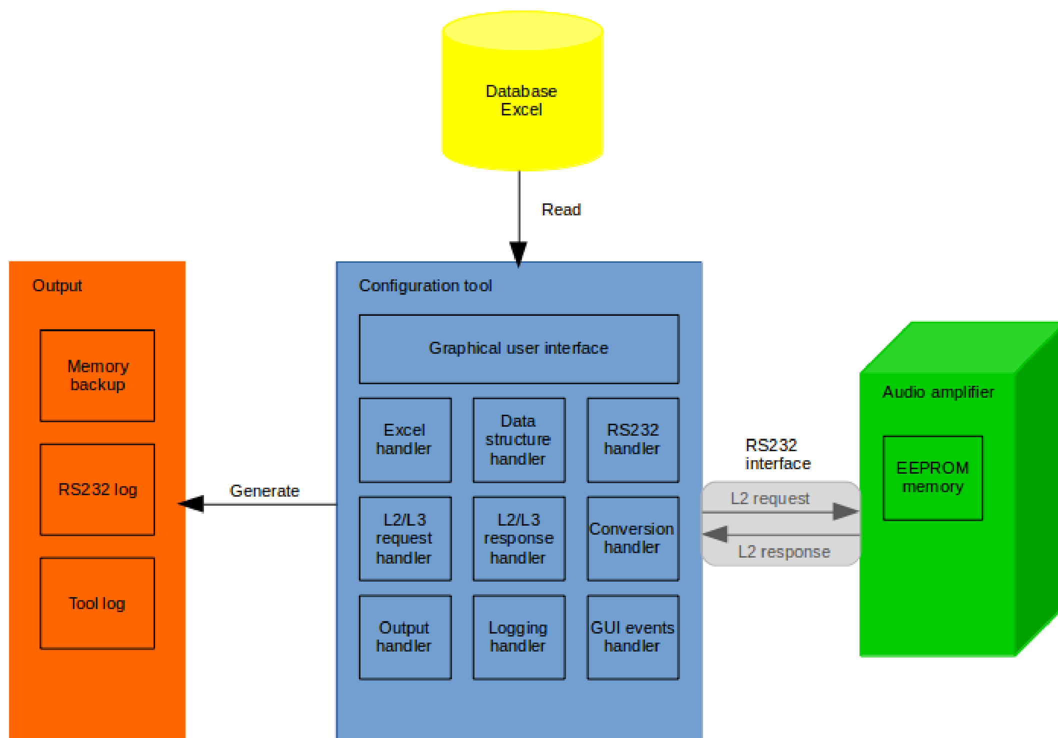

Figure 3: System overview

3 Design and implementation

The developed software application can be divided into larger modules and smaller functions. Some modules are independent, whereas the rest depend on the output of some other ones. This means a certain module can not perform assigned functionality before the other component has finished its job. The purpose of the following paragraphs is to give a decent overview of the main functionality of the application.

3.1 Serial port initialization

Handling of the serial communication is implemented by using the C# SerialPort class [3] . It provides a synchronous and event-driven input / output handling and access to all drivers properties. Also, using this class gives possibility to configure the RS232 specific parameters such as the baud rate, data bits, parity indicator etc.

During every start-up of the application, an instance of the *SerialPort* class gets created by using a constructor with the default parameters. In this case, the data bits property gets value 8, the parity identifier is defaulted to none and a single bit value is counted for the stop-bit. Only the baud rate and the COM port name are configurable by the user during the runtime. Both of these parameters can be changed through the GUI as it provides the lists containing possible options. The baud rate, which indicates to the number of symbol or signalling event changes across the transmission medium per second [4] , can be chosen from a list generated upon opening an appropriate combo box element. The correct selection for the communication with the amplifier is 115200MHz. This value is offered by the tool automatically as the first option. The connected COM port names are scanned by the tool by using the *GetPortNames()* method of the SerialPort class. This method filters out only those COM port names that are currently connected to the PC including the virtual ones. Only these will be displayed and prompted for the selection.

There is no possibility to distinguish which COM port the amplifier is connected to. The easiest solution is to select any of the available and check whether the RS232 communication log is being displayed to the log window.

Opening and closing the serial port interface is implemented with the events of the separate button clicks. Each time when the user presses the *Open* button, the application checks whether the baud rate is selected and the correct COM port name chosen. The size of the serial port input buffer is left to its default value, which is 4096 bytes. Furthermore, the program creates a new *DataReceivedEvent* handler. This event is triggered every time when at least one byte becomes available to read in the input buffer. Taking into account, that the minimum length of the L2 response is 17 bytes in case of the smallest possible memory record and the maximum length can expand until 242 bytes, then the input buffer is not checked before at least 256 bytes are available for reading. The serial port connection is finally established by using the *Open()* method and closing is done with the *Close()* method. Calling this method automatically sets the *IsOpen* property to false. Value of this flag is checked each time when the user wishes to either open or close the serial connection or perform any action requiring it. The final step during closing of the port is removing the *DataReceivedEvent* handler to avoid any processing events in case the serial connection is not established. Ignoring this fact will lead to a significant decrease of the application performance. The sequence which takes care of closing the port is event-driven. The event is triggered by pushing the *Close* button.

3.2 Logging functionality

Nearly every IT (Information Technology) system or application generates a trace while operating. This is the only way to analyse any kind of unexpected issues such as software misbehaviour under certain circumstances or investigate the software bugs. The current application generates 2 different types of logfiles. The first set of files contains the trace of the communication with the amplifier through the RS232 interface. This trace can be displayed online to the GUI during the runtime and also saved into a simple text file in parallel. Additionally, it can be saved in two different ways – either in the hexadecimal numeric system or in the ASCII encoding standard. The second part is

a trace containing the operations performed by the application. This includes any kind of actions initiated by the end-user. Having a sequence of operations, which lead to erroneous behaviour of the application, will give a significant advantage in further investigation and also support solving the potential bugs.

3.2.1 Online

The logging functionality is built by using a *DataReceivedEvent* handler which was declared during the initialisation of the serial port. Normally, the *SerialPort* class reports with an event each time when at least one byte is available for reading. However, as one byte of data is definitely not enough, the application waits until at least 256 bytes are available in the input buffer. This is done by checking the number of available bytes in the input buffer. Having this condition met, an event is fired which starts the sequence of processing the input buffer of the serial port. During the first step, the data from the serial port buffer is copied into a local auto type buffer of the handler function. This approach allows to clear the input buffer soon as possible. Intention of it is to avoid collecting too high amount of data and thus avoid any possible buffer overflow. Processing of the input, which is acquired from the serial port, is running in a separate thread compared to the main application. After copying the input buffer into a local array of bytes, it is forwarded to the *HandleResponseFromSerial()* function. This function runs in the main thread and thus the processing takes place at the same time when new the content is already being saved into the serial input buffer. This operation is solved by using a C# special construction called *Dispatcher Invocation*. This topic will be covered in the paragraph 3.3.1. In case the user has not enabled the logging into the file, the log is displayed in the RichTextBox element. This element is located on the GUI. Writing to the output window is implemented by using a *RichTextBox* class method *AppendText()*. The encoding format is chosen by judging the input from the user. In case of ASCII encoding, the data bytes are converted into ASCII and then transformed into a string by using the *Encoding.ASCII.GetString()* function. Otherwise, each element of the byte array is converted into its equivalent hexadecimal string representation. Afterwards the result gets written to the GUI output window. Similar procedure is done also for writing the ASCII type data to the GUI output window. Otherwise, the byte array gets converted into the numeric value of each element of the

specified array of bytes into its equivalent hexadecimal string representation and then written to the GUI.

3.2.2 To file

Saving the RS232 log to the file has exactly the same core implementation as the online logging part. The same RS232 traffic, which is logged onto the GUI, is saved into a simple text file in parallel. This is done only in case the user has explicitly enabled this functionality. If so, the GUI prompts the user to select an appropriate file where to store the log. This is implemented by using a Microsoft static class named *OpenFileDialog*. The configuration is made in a way that only the files with *.txt and *.log extensions can be selected. The *.log files are exactly the same as *.txt files and can be opened by any text file editor. The *.log extension has rather an illustrative purpose. It just indicates to a log file. The other type of files are not displayed as an option. As soon as the selection is done, a global instance of *StreamWriter* class gets created. Similar as for the online logging, the correct encoding depends on the configuration. The handling of the correct encoding is solved in exactly the same way.

The log is appended single line per function call. The end of the line is detected by searching for the end-of-line ASCII identifier. Before saving the line into the file, it gets equipped with a timestamp indicating to the time when it was detected.

3.3 Reading input from database

The parsing of the Excel database for the required data fields is one of the most significant parts within the application. The complete input for generating the L2 commands, requesting the content from the amplifier and displaying the received content of the EEPROM memory record(s) are acquired from it. After the user has added and also selected an appropriate path to the database, the application begins working with it. The sequence is initiated by a change in the input file compared to the previously loaded one. The difference is detected by checking the last modification date of the database. This means as soon as the user introduces a change to the database and saves it, the tool will recognize it as the date is automatically updated during the save operation.

3.3.1 Using the BackgroundWorker

The database describing the memory records is relatively large. Its size is approximately 1MB. Acquiring the data from the file with that size is time-consuming. In order to avoid the GUI from getting frozen during the runtime, a *BackgroundWorker* class is used. The purpose of the Silverlight BackgroundWorker class is to provide a simple but convenient way to execute time-consuming operations, such as database transactions in a separate thread [6] . Using this class allows the user-interface to remain fully functional as long as the data is read from the database in the background. The user can do any other operation which does not depend on the database input, for example configuring the RS232 connection and logging the communication with the amplifier. This is possible as the data acquisition will run in a separate thread compared to the main body of the application.

Two instances of the *BackgroundWorker* class are created every time when a new database is detected. The first one is a thread in which the data is being transferred from the database and is called *bgwDataTransferer*. The second one handles the progress-bar on the GUI and is called *bgwProgressUpdater*. The purpose of the progress-bar is to give a glimpse to the user about the remaining time until the data has been transferred from the database into the internal buffers and can be used for further processing.

The progress-bar is synchronized with the time. The time for parsing the database was measured and the worst case working time was taken for the progress-bar interval time. In case the data gets acquired faster, the remaining gap of the progress-bar will be forced to the end.

The following image illustrates the progress-bar during the data acquisition from the database.

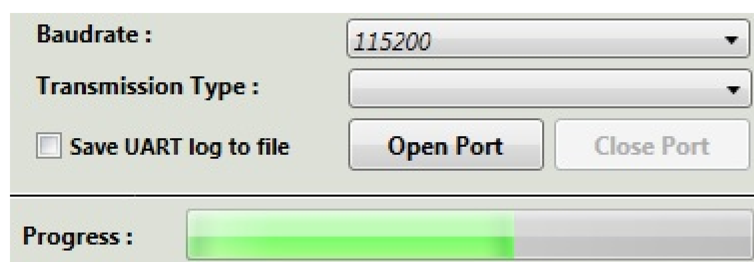

Figure 4: Activated progress-bar

The *BackgroundWorker* class contains several configurable events. The current application requires following events:

- *DoWork* – the event in which time-consuming operations are intended to be called.
- *RunWorkerCompleted* – the event happens when the background operation is finished, cancelled or run into an exception.
- *WorkerReportProgress* – enables or disables the progress update reporting of the BackgroundWorker
- *ProgressChanged* – the event happens when the progress is changed.

As a next step, both background threads are properly initialized and started. Aforementioned events are connected to the functionalities which get triggered when they occur. The events of the *bgwDataTransferer* instance are linked with the following functionality:

- *DoWork* – the event which starts the sequence of extracting the input from the Excel database and storing it into the internal buffers.
- *RunWorkerCompleted* – the event is fired as soon as the input acquisition is finished. It will be enabled depending on the functionality.
- *WorkerReportProgress* – the option is configured to false as this thread does not need to report the progress in live.

BgwProgressUpdater is configured in a very similar way with the following functionality:

- *DoWork* – the event which triggers the sequence for initializing and starting the progress-bar.

- *ProgressChanged* – the event updates the progress-bar value during the operation.

- *WorkerReportProgress* – configured to true as the ProgressChanged event is used.

After the configuration, both background threads are started by calling a *BackgroundWorker* class method *RunWorkerAsync()* with the correct input parameter. For the *bgwDataTransferer* it is the name of the selected database and for the *bgwProgressUpdate* the maximum value of the bar in percentage.

3.3.2 Searching specific records

Directly after starting the *bgwDataTransfer* thread, the application opens the database by using a Excel Data Reader *CreateOpenXmlReader()* library function. This method must be used as the database is created with the Excel framework newer than the 2007 edition. Alternatively, the application is also able to parse the older versions of Excel, but in this case it is done by using a binary reader. As a result of this function call, each spreadsheet located in the database gets stored in a separate table. All the tables together form a data unit called the dataset. Every spreadsheet within the database corresponds to a certain customer project. All found customers are selectable from a combo box displaying the projects on the GUI. Only finished *bgwDataTransferer* thread enables this element.

All further functionality requires an input from the user. First of all the user must select an appropriate project which matches with the SW running on the audio amplifier. This is required as the projects are different in the EEPROM memory layout and in the location of the records. Additionally, the parameters of the records are various between the projects. The amount of the projects is not determined as they can be easily added into the database. The application will recognize the new project and list it into the combo box as it creates a table for each sheet in the database.

Selecting any of the available projects triggers a similar functionality with the help of the *BackgroundWorkers* as was done for the database loading event. Also, the progress-bar is reinitialized and restarted as loading the project is quite time-consuming process.

The estimated time needed for the loading of any project is approximately 6 seconds and it is quite same for all the projects as their size is similar. The occurrence of the changed selection event, a Xlsx handler function *GenerateStructures()* is called. At first, the selected project is mapped to the correct table of the dataset. Each project consists of the platform and customer specific records as mentioned in the paragraph 2.5.1. This is followed by looping through the platform and customer tables accordingly. The loop goes through every row and column of the table and searches for the “VH_EEPROM” string. This type indicates that the current entry is located in the EEPROM memory and not in the Flash memory. As soon as such entry is found, a structure of type *EepomRecord* (described in the paragraph 3.3.3) is created and filled with the corresponding data found on this particular row. The correct indexes of the appropriate columns are chosen from the custom class *Constants* based on the selected project.

```
for (int i = 0; i < sheet.Columns.Count; i++)
{
   if ((row[i].ToString() == "VH_EEPROM") && row[0].ToString() != "free")
   {
      EepromRecord rec = new EepromRecord();
      string temp;
      rec.size = Convert.ToInt16(row[consts.colSize]);
      temp = row[consts.colOrderIndex].ToString();
      rec.orderIndex = Convert.ToInt16(temp.Replace("100", ""));
      rec.cacheType = row[consts.colCacheType].ToString().Contains("VH_CACHE") ?
true :  false;
      rec.strSection = row[consts.colAddress].ToString();
      rec.uintSection = uint.Parse(rec.strSection.Replace("Sec",
string.Empty).Split('.')[0]);
      rec.wordNr = rec.strSection.Split('.').Length > 1 ?
uint.Parse(rec.strSection.Split('.')[1]) :       uint.MinValue;
      rec.length = Convert.ToUInt16(rec.size - 1);
      rec.strID = row[consts.colID].ToString();

      records.Add(rec);
      }
}
```

Figure 5: Parameters extraction functionality

Finally, the generated structure is stored in a list of structures of *EepromRecord* type. The list is used instead of a typical array due to following advantages:

- There is no need to specify the length of a list as it is dynamic.
- It is simple to add, remove and parse the elements.
- Easy to access any member by using the VH ID number

3.3.3 Result structure parameter description

Each EEPROM memory record is stored as a custom data structure containing different type of elements. Each of the parameters has its own purpose and is used by the application for further activities. The following picture illustrates the *EepromRecord* class and describes each element.

```
public partial class EepromRecord
{
   /* ID of a memory item in ASCII representation */
   public string strID { get; set; }
   /* Special index for ordering the content from the memory. */
   public int orderIndex { get; set; }
   /* Describes whether the record is stored in cache before writing to EEPROM. */
   public bool cacheType { get; set; }
   /* The size of a record in bytes */
   public int size { get; set; }
   /* ASCII representation of the section where memory the memory record resides */
   public string strSection { get; set; }
   /* Numeric representation of the section where the memory record resides
      used when creating a CSV image of the EEPROM
   public uint uintSection { get; set; }
   /* Used for calculating the startOffset value */
   public uint wordNr { get; set; }
   /* Used for calculating the endOffset value */
   public uint length { get; set; }
   /* Defines the start offset of the record for the initial EEPROM state image */
   public uint startOffset
   {
      get { return 16 * uintSection + wordNr; }
   }
   /* Defines the end offset of the record for generating the initial EEPROM state image */
   public uint endOffset
   {
      get { return startOffset + length; }
   }
}
```

Figure 6: CsvRecord class

All entries present in the EEPROM memory of the audio amplifier are saved into a structure described in the previous paragraph. The structures are placed into a list of *EepromRecord* type. This is done just once, directly after acquiring all the records from the Excel database as generating the structures is relatively time-consuming process.

Directly after all the structures are created and stored in the list, it gets sorted according to the *orderIndex* parameter. As a next step, the application creates another simple list, which contains the sorted order indexes. They will be displayed on the GUI once the

user has opened the corresponding drop-down menu for choosing which memory record must be either read or modified. After an appropriate selection, the list containing the *EepromRecord* type elements is searched for the correct member by using the *ElementAt()* method. As a result, it returns the element from the list that has the matching access identifier index. As soon as there is a hit, the parameters of the searched record are collected and used for the further procedures.

3.4 Saving initial EEPROM state

This paragraph describes the purpose of saving the initial state of the EEPROM memory, how does the workflow look like in the application and introduce the final output.

3.4.1 Purpose

Each amplifier has exactly the same entries stored in the EEPROM memory, but their values are different within each device. Part of the content is dynamic which means it is updated during the runtime of the amplifier. Such data is for instance the system runtime, statuses of the diagnostic trouble codes, different measurement values etc. The other part of the memory is static, which is opposite to the dynamic ones. Such values remain unchanged starting from the time they are written during production until the end of the amplifier’s life-time. Sometimes it is necessary to change a certain content of the EEPROM memory for achieving a special state of the amplifier or for simulating certain preconditions for some further activities. For instance, testing certain input signals for the ANC EOL test must be written before misbehaviour during the ANC measurements can be reproduced and analysed further. Achieving such preconditions requires modification of this record in the EEPROM memory.

At some point the dynamic memory records can introduce a scenario, where some values will be updated during the testing although they should not. The second problem is that by a mistake, a modification of an unwanted data record(s) can happen. These points are the main arguments for having such functionality that gives the opportunity to create a copy of the memory initial state. The result of this functionality is a memory image, which reflects the exact state before any further changes are applied. It can also

be called the primary state. The outcome must be converted into a valid file format, which can be directly used by the tools that are responsible for the SWDL process.

3.4.2 Workflow

The process gets started by an event which is fired as soon as the user presses the *Read EEPROM* button on the GUI. The list of the order indexes plays an important role in this functionality as it is taken as an input parameter. At first, the application disables any other functionality which can interrupt the reading of the EEPROM. The start point of the readout is saved and displayed to the user.

Started reading EEPROM [7.03.2016 12:51:05].

Figure 7: Readout started notification

The content of each record is then requested from the amplifier one-by-one based on the *orderIndex* parameter. The functionality is created by using the same *RequestEepromEntry()* function as when requesting a single entry. The difference is that first time this function is called with the first orderIndex value and after receiving the response from the amplifier the index is incremented and the next record is requested. This loop continues until the values of every EEPROM memory record has been acquired. The end of the readout process is reported to the user.

EEPROM readout finished [7.03.2016 12:51:12].

Figure 8: Readout finished notification

Each time the response is received, the content is stored into a structure containing the following elements: *dataID*, *startOffset*, *endOffset* and the array of bytes indicating to the received value of the entry. Each structure is added to a list by using the *Add()* function.

```
public class CsvRecord
{
   public string DataId { get; set; }
   public uint StartOffset { get; set; }
   public uint EndOffset { get; set; }
   public byte[] Data { get; set; }
}
```

Figure 9: CsvRecord class

After all the needed information is received from the amplifier, mapped to their logical meaning and equipped with the *startOffset* and *endOffset*, the content is stored into a CSV file. The application creates the CSV file automatically and configures the path where the output is stored by building up the pathname relative to the location of the application. All the files remained from the previous runs will be removed.

Further on, the tool creates a buffer of 4KB and pre-fills it with the values of 0xFF. This is made due to the reason that the default content of the EEPROM is 0xFFs. Such approach requires less effort. Finally, each record is written to its appropriate location with the help of the *startOffset* and *endOffset.* Each line in the CSV file has 4 data bytes per line and starts with an identifier 0x indicating a hexadecimal value. The writing of the records is solved in a simple for loop.

```
StreamWriter csvWriter = new StreamWriter(csvFile);
byte[] init = Enumerable.Repeat<byte>(0xFF, 4096).ToArray();
for (uint offsetIndex = 0; offsetIndex < init.Length; offsetIndex++)
{
   if (csvrecords.Any(x => x.StartOffset == offsetIndex))
      {
      CsvRecord csvr = csvrecords.First(x => x.StartOffset == offsetIndex);
      Array.Copy(csvr.Data, 0, init, offsetIndex, csvr.Data.LongLength);
      }
}

for (uint i = 0; i < init.Length; i++)
{
   if (i != 0 && i % 4 == 0)
      {
      csvWriter.WriteLine(",");
      }
   csvWriter.Write(init[i].ToString("X2"));
}
csvWriter.Close();
```

Figure 10: CSV generator loop

A cut of the result CSV file is illustrated on the following figure.

```
892  FFFFFFFF,
893  FFFFFFFF,
894  34AC23AB,
895  FFFFFFFF,
896  ABBAFFFF,
897  FFF43FFF,
898  FFFFFFFF,
899  FFFFFFFF,
900  FFFFFFFF,
901  FFFFFFFF,
```

Figure 11: CSV content

3.4.3 Supporting tools

The final output of the EEPROM copy generated by the tool itself ends with the CSV file. However, this file is not yet ready to be used in the SWDL sequence as it is not converted into a valid file format and is still missing the correct addresses.

This is achieved by the support of the external tool called FlashWrapper. The purpose of the FlashWrapper is converting the raw content from the CSV file into Motorola S-Record [5] (MOT, extension *.mot). It sorts the entries into the EEPROM pages having 32 bytes per line and each line starts with the start address of the EEPROM memory and ends with a CRC.

3.4.4 Output

As a result, the initial state of the EEPROM is stored in 2 different formats:

- CSV format – generated by the configuration tool.
- MOT format – generated based on the CSV format by the FlashWrapper.

These files contain the exact state of the EEPROM content at the time when it was completely read out. The second one can be flashed directly to the amplifier by using a special debugger device.

3.5 Reading and modifying the records

The application provides two possibilities to work with the memory records. They can be just read out and observed as a result on the GUI or they can be modified with the necessary values and written back to the amplifier memory. Both functionalities have a common part of the implementation, but modifications of the records is expanded accordingly.

3.5.1 Building and sending the request commands

Reading the records is by far the most important functionality of the tool as it is the base for many other functions such as modifying the values and saving the initial content of the EEPROM.

The process begins with an event which is triggered after the user has chosen an appropriate logical name of the EEPROM record from the drop-down menu. The application finds the correct order index matching the selection and calls the *RequestEepromEntry()* function. Then it searches the correct matching element from the list of *EepromRecords* and generates the L2 command (described in the chapter 2.3.3). The process continues with the CRC calculation over the prepared command. As a final step, the first byte of the L2 command – 0x1B is inserted to the first position of the array as it is ignored during the CRC calculation.

Due to the fact that the L2 commands differ only in the *orderIndex* parameter value and the CRC, there is no recalculation of the L2 and L3 message lengths. The generated L2 command is then sent towards the amplifier by using the *SerialPort* class *Write()* method.

3.5.2 Parsing response and displaying result

The *SerialPort* class is running in a separate thread, which makes the usage of the *Dispatcher.BeginInvoke()* method inevitable. It allows to forward the event to the main thread which processes the information independently from the serial port thread.

At least 256 bytes must be available in the serial port input buffer to initiate the parsing mechanism. This value is chosen to avoid a scenario where just some bytes are present in the input buffer which do not form a valid L2 response yet. Firstly, the software searches for the special L2 start byte (0x1B) from the buffer. Detecting such value means that the beginning of the L2 response is located. The response is then saved into a created local buffer that will contain only the data bytes of the requested EEPROM record. This is achieved by ignoring the L2 and L3 header bytes and taking only the number of bytes defined in the record size parameter.

The process ends with displaying the content of the requested record together with its logical name on the GUI. Each byte is separated with a dash when displayed. Additionally, the application notifies about the type of the readout. It can be either cached or not cached.

3.5.3 Changing the values of records

The readout procedure of the memory record ends with its value displayed for the user on the GUI. Sometimes it can be necessary to change it to any other value and then write it back to the audio amplifier. Hence, the tool gives this opportunity.

After the data field of the memory record has been modified as needed, the user must trigger an event to initiate the writing procedure. It is achieved by pressing a corresponding button, which generates an event. As soon as the event occurs, the SW takes the VH ID of the currently selected record. This ID is then used to find the matching *EepromRecord* element from the list of all records. It is necessary to acquire the needed parameters of selected record, such as length for instance.

During the next step of the sequence, the application copies the data bytes from the result box where the content of the record was displayed. This content now is already modified by the user. The type of the content is string and thus the dashes are easily removed and then the remaining string containing only the data bytes is converted into an array of the same type. The converted array already represents the payload of the L3 command. Further on it is passed into a L2/L3 command generator, which creates the valid requests. The input for the generator are the payload array, the VH ID and the size of the record.

The generator functionality is implemented by using the list data type and its manipulation functions I*nsert()*, *Add()* and *AddRange().*

- *Insert()* - inserts an element into the list at the specified index.
- *Add()* - adds an object to the end of the list.
- AddRange() - adds an element of the specified collection to the end of the list.

Initially, 3 lists are created already containing certain constant values.

- *EepromWriteCmd* – the L2 type identifier byte
- *L3ConstantBytes* – MsgId and first 3 bytes of the record identifier.
- *L3StartBytes* – ProtocolVersion and Flags, L3 destination address, L3 destination sub-address, L3 source address, L3 source sub-address

The first step is appending the VH ID to the end of the *L3ConstantBytes* list. Then the generator proceeds with the calculation of the L3 telegram payload length by taking the size of the memory record and adding the length of the *L3ConstantBytes* and subtracting 2 bytes indicating the *MsgId*. The calculated length is then inserted to the first position of the *L3ConstantBytes* list using the *Insert()* method. The next step is the calculation of the L2 payload length. This value is got by taking the number of bytes currently present in the *L3StartBytes* list and adding the number of bytes present in the *L3ConstantBytes* list and adding the size of the memory record. During the next step, the *EepromWriteCmd* is filled with the contents of the remaining of 3 lists. The L2 length is added to the beginning of it and the *L3StartBytes*, *L3ConstantBytes* and the EEPROM data is appended to the end by using the *AddRange()* method. The process continues with calculating the CRC over the generated L2 command currently stored in the *EepromWriteCmd* list. As this list does not include the L2 start identifier yet, the CRC is calculated over the all bytes and then appended to the end of the list. Finally, the L2 start byte is added to the first position of the list using the Insert() method. The process ends with returning the complete and valid L2 command from the generator function.

Having the L2 command available, it is forwarded to the serial port in the same way as any previously mentioned requests. Directly after sending the command, the application checks the type of it. If it is cache type, then the user gets notified about required rest of the amplifier. The cache type record means that it is stored in the cache memory and is not immediately written to the volatile memory. Such records are written into the volatile memory during the shutdown sequence of the amplifier. For this purpose the GUI is equipped with a button which fires an event that generates and sends the reset command. Only after executing an amplifier reset the value is persistently written into the EEPROM memory.

3.6 History of tool usage

The tool is equipped with a special log window which tells the user what is currently in progress or what is intended to be done. Basically the tool remembers all the steps made by the user during the runtime. Apart from that, it is able to point out what must be performed in order to enable certain functionality. For example, one might want to open the serial port without selecting the COM port and the baud rate or send a L2 command without having the serial port open. In this case the tool informs what preconditions are missing and must be fulfilled to get it working. Another example is informing about the finished data transfer period or whether the L3 command generation was successful or not.

Having stored all the steps made by the tool gives an opportunity to simplify reproducing the different possible error scenarios. Most of the times any misbehaviour is reported in a style that something failed when modifying the record. However, it might not be enough to know only this information. It is possible that some previously executed steps opened the path for this error to occur. If so, the sequence will be nicely present in the logfile.

On the GUI, the tool shows the last 50 lines of the log lines. A queue is used for implementation which is cleared as soon as more than 50 lines are saved. This is done to avoid the scrollbar getting too small and uncomfortable for the user to use it afterwards. However, the data does not get lost in that case. Exactly the same information messages are also saved into a simple text-file in parallel. It is created as soon as the tool is started and everything is written to it line-by-line until the end of the run cycle. Every logfile contains the date and creation time in its name and are not deleted before creating the new ones.

3.7 Data safety

The database contains also records which are not located in the EEPROM but in the Flash memory. Every entry has a strictly defined length and some values must not be read or changed at all. These facts actualise several points regarding the data safety, which must be taken into account.

The implementation of the tool assures that only these records that located in the EEPROM memory can get modified. Hence, only database items which are marked as EEPROM device type are acquired from the Excel by the tool. The rest of the content is ignored. This ensures that data in the Flash remains out of operation.

Due to the strictly defined length of the items it must not be possible to write other number of bytes than defined. After displaying the requested content on the GUI, the application does not allow to resize the data field – meaning making it shorter or longer is prohibited. Writing more bytes than defined by the record size would corrupt the data located in the following cells of the memory. Additionally, only the values 0x0 up to 0xF are allowed to be entered. Any other keyboard character is forbidden.

Finally, some certain memory items must not be at all viewed or changed at all. Example of such value is the decryption public key and some other classified information. The identifiers of such items are not displayed at all and thus it is not possible to either read or change them.

4 Graphical user interface

4.1 Design

One of the requirements proposed by the amplifier software development team was a graphical user interface. Two different frameworks were considered for developing it. Both of them were briefly analysed before making the final decision which one to use. The next sub-paragraphs will give an overview of the considered frameworks.

4.1.1 WinForms

WinForms has been used as a platform for developing Desktop applications for years. It’s a graphical class library that provides native Microsoft Windows elements to the developer [7] . The WinForms is basically a blank paper which needs to be filled with elements such as buttons, textboxes, combo boxes etc. Such approach gives an opportunity to create the user interface and complement element’s functionality according to the needs. It is simply a layer on top of the standard Windows controls. This platform is an excellent for developing the client applications for desktop, laptop and tablet computers [8] .

Table 3: Advantages and disadvantages of WinForms

Advantages	**Disadvantages**
Excellent documentation provided by Microsoft	Designing own look requires much effort
Large number of the examples	
Has been used for years and thus thoroughly tested	
Works on Windows versions older than 2000	
Supports the WPF	

4.1.2 WPF

WPF stands for Windows Presentation Foundation. It is a system created by Microsoft for developing client applications. WPF allows creating a wide range of stand-alone applications and implement functionality which responds to the user interactions. It includes vector-based and resolution-independent engine and gives the possibility to create interfaces and common media elements by using the vector and raster images, audio, videos etc. [7]. On the contrary to the WinForms, the WPF does not rely on the standard Windows controls.

Table 4: Advantages and disadvantages of WPF

Advantages	**Disadvantages**
Easy to create own design for any component – independent of the 3rd party controls.	It is still in development.
The structure is easy.	Lack of examples compared to WinForms.
Supports XAML (Extensive Application Markup Language).	Less tested compared to the WinForms.
Newer and more innovative compared to the WinForms.	Requires the .NET Framework 3.0.
Good performance – uses hardware acceleration.	
Flexible and allows reusing the existing code.	

4.1.3 Graphical solution

In order to gain experience with a new technology, the WPF was chosen as a framework for developing the graphical user interface. It was designed to be easy and as self-explaining as possible. It contains several types of elements with different functional purpose. Complete interface is designed by using the XAML. It is XML based markup language developed by Microsoft. It’s a language behind the graphical representation of the developed application. The power of the XAML is possibility to manipulate the visual representation by using the code, which makes the WPF far more flexible than the WinForms [9] . The GUI of the current application was created and built by using the design and development views (code) in parallel.

The following page illustrates the GUI.

Figure 12: Graphical user interface

4.2 Content overview

The graphical user interface contains buttons, combo boxes, text boxes, labels, radio buttons and a check box. The interface can be divided into 4 different group boxes.

The upper sections contains a simple menu bar with two sections – File and Help. Under the *File* there is just one option to close the application, but the *Help* contains commands for displaying the user-manual, contact information of the responsible developer and also general information about the tool.

The upper group called *Databases* is for adding and selecting the Excel databases (input) and choosing the required project. In case of a completely new database, a button that opens a window allowing the user to navigate to the location of the correct file must be clicked. The implementation is done by using the *OpenFileDialog* functionality. Only the files ending with .xlsx or .xls extensions are accepted and displayed. This is configured in the *OpenFileDialog* filter. Every occurrence of a new database will be also saved into a file, which contains the history of recently used databases. Additionally, the selected database will be displayed into a combo box right next to it. Recently used database files will not be forgotten by the tool after terminating or restarting it. The paths are acquired by the tool from the history file. The group ends with a checkbox that provides possibility to choose the needed project.

The second group is called *RS232 configuration*. It is meant for configuring and opening the serial port communication. There are two combo boxes for choosing the correct baud rate and COM port name. A list containing the COM port names shows only the available ones as the ports are scanned each time before the menu is opened by the user. Opening and closing the COM port is handled by two separate buttons. They are located below and can not be active at the same time, meaning the button for closing the port is disabled when the port is actually closed and vice versa. Additionally, it is possible to choose the format of the RS232 log. Based on the selection it will be either displayed in ASCII and hexadecimal format. The group ends with a checkbox which activates the functionality to save the log into a text file. As soon as the user marks the checkbox, an *OpenFileDialog* window opens allowing to choose any text file where the log shall be written to.

Right before the next group there is a progress-bar, which shows the approximate status of time-consuming operations.

The third group is called *Custom commands*. This section is meant for sending any valid L2 commands to the amplifier. The valid command must be entered into the field and sent towards the amplifier by clicking the *Send data* button. The command must be added without the CRC as the software calculates these bytes automatically. In case of an invalid command, the application still sends it towards the amplifier, but it will be discarded by the target.

The next section is named *EEPROM read* and it contains two radio buttons, one usual button and a combo box. The purpose of the radio buttons is to activate the reading of a single EEPROM record or the complete EEPROM. They are exclusive as these operations can not be performed at the same time. Choosing the radio button with a label *Complete EEPROM* activates the button which allows reading out and saving the memory image. The list containing VH IDs remain disabled. The readout process starts after clicking the *Read EEPROM* button. During the operation, it is not possible to open or close the serial port and thus the RS232 group is disabled. In case the user selects the radio button labelled as *One entry*, the button for reading complete memory turns disabled. The activated list contains the list of available memory records for requesting. As soon as any of the records is chosen, the tool automatically generates the correct L2 request command (details in the paragraph 3.5.1) and sends it towards the amplifier.

The group right below the reading section is for changing the content of the memory. This area is called *EEPROM write*. In case the readout of a one entry was initiated, the result will be displayed into the text box called *Result*. The modification of the EEPROM data must be made directly into the text box by the user as it is not read-only. Only the values from 0x0 until 0xF are accepted and any other character is discarded. Additionally, the application restricts writing more or fewer bytes than specified with the size parameter of the EEPROM record. The content will be sent to the amplifier after clicking the *Modify* button. The button called *Reset target* is for resetting the amplifier if the modified record is of cache type. If so, the application will inform and the colour of the *Reset target* button turns into red.

The largest elements on the GUI are the logging windows. One of them is for displaying the RS232 communication in live mode. The second one at the bottom of the GUI is for showing the actions performed by the software. Apart from that it provides hints regarding what must be done in order to achieve some certain functionality or to get the preconditions correct. For example, the baud-rate and the COM port name must be selected before the port can be opened or the port must be open before any command can be delivered to the amplifier.

At the bottom of the panel there are two buttons. The first of them closes the application and the second one clears the RS232 online log window.

4.3 Tooltips

Every element on the GUI is equipped with a tooltip. *Tooltip* is a standard class which activates a small pop-up window for displaying a brief description of a control's purpose [10] . It gets triggered when the user moves the cursor to any of the elements. The idea of the tooltips is to provide information about the purpose of the highlighted element even without the user manual. They are activated by using the *Show()* method, which sets up a text linked to the highlighted tooltip and then displays it.

The following table illustrates all available tooltips together with their linked elements.

Table 5: Displayed tooltips

Element	**Displayed tooltip**
btnOpenPort	Opens the serial port if the baud rate and port name are chosen.
btnClosePort	Closes the serial port.
btnSendData	Sends the L2 command via the serial port to the amplifier.
btnReadEEPROM	Triggers a sequence to read all the EEPROM records for the specified project.
btnClear	Clears the logging window.
btnAddDatabase	Adds a new database which can be selected from the drop down menu.

btnExit	Exits the application.
btnModify	Sends the modified EEPROM record data to the amplifier.
btnReset	Sends the reset command to the amplifier.
tboSendSerial	Field to enter the custom L2 command which is intended to be sent to the amplifier. The white space characters are allowed.
tboVhID	Displays a list of the EEPROM Ids that can be requested.
tboRespHex	Displays the EEPROM record response in the hexadecimal format.
tboInfo	Displays helpful information about the process status.
tboType	Displays the type of the EEPROM record.
checkBoxSaveLog	Option to choose whether to save the RS232 log into a file in parallel.
RbtnReadEntry	Allows the user to read the EEPROM entries one-by-one.
rbtnReadEEPROM	Allows the user to read out the complete EEPROM.
cboxPorts	Contains a list of available ports.
cboxBaudrates	Shows a list of the available baud rates.
cboxVhID	Shows list of EEPROM Ids which can be requested.
cboxProjects	Shows a list of the projects which can be chosen.
cboxTransmissionType	Shows a list of the transmission types. The ASCII type is used by default.
CboxDatabases	Displays a list of the recently used databases.
pbarWorking	Shows the status of the time-consuming processes.

An example of a displayed tooltip looks as on the following capture.

Figure 13: Displayed tooltip

4.4 Restrictions

Majority of the PC applications has certain rules of usage. Not all the functionality can be performed simultaneously. Therefore, while using the tool some restrictions must be taken into account.

The first restriction is that the amplifier's project can not be selected if the database is not loaded. Thus, the combo box containing the projects remain inactive until the database is provided as an input. The next restriction blocks opening the serial port as long as the baud rate and the COM port name has not been selected. Once the serial port is open, these parameters become disabled.

As a precondition for sending a custom L2 command towards the audio amplifier, there must be something to send. This means the tool will discard the attempts to send anything without having a valid data available in text box meant for it. An open serial port serves as another precondition for this functionality.

Viewing and changing the EEPROM memory records is possible only when the serial port is properly configured, the database is loaded and the desired project selected. Only in this case the radio buttons allowing selecting between these functionalities become active. Another major restriction is that it is not possible to read a single EEPROM entry while reading the complete memory is in progress. Thus, these functionalities are oppositely exclusive.

Changing the data of the memory records can only be done in case the wanted record has been read out in advance. If so, the content is displayed in the result text box. There are several restrictions which apply when modifying the data bytes. First, it is not possible to shorten or extend the data field of the records. It means every time only the

exact number of bytes will be sent that is defined with the size field of the EEPROM record. In case of using backspace to remove the bytes, they will be replaced with the 0x00 value instead to maintain the visualisation of the record size for the user. Additionally, only the valid hexadecimal values are allowed to be entered. This means the correct range of values is from 0x0 up to 0xF. The attempts to type any other character will be discarded and also notified.

Finally, in case a cached type record is changed during modification, the *Reset target* turns red as the restart of the amplifier becomes necessary. Otherwise, the changes will not be saved and thus lost in case a power off condition occurs.

5 User-manual

Each released software application must be equipped with an user guide. Although the tooltips give plenty of information, there are still some details that must be described in a more specific way.

Current SW application has a built-in user-manual. It can be activated from the Help menu present on the toolbar. A new window will appear containing a combo box and a text window. The combo box contains different scenarios that can be performed with the tool. Choosing any of them will display step-by-step instructions describing how to achieve the goal.

The functionality is implemented based on the selection index change event. The event happens each time when the user chooses a scenario from the list. Based on the index the application searches for the correct content from a simple text file and displays it into the text box. The content in the file is separated with the same indexes as in the list. This means the SW reads all the lines from the file which follow the identifier until the next identifier. An advantage of such implementation is the possibility to update the contents of the user-manual without compiling a new version of the application.

The following picture illustrates an activated user-manual.

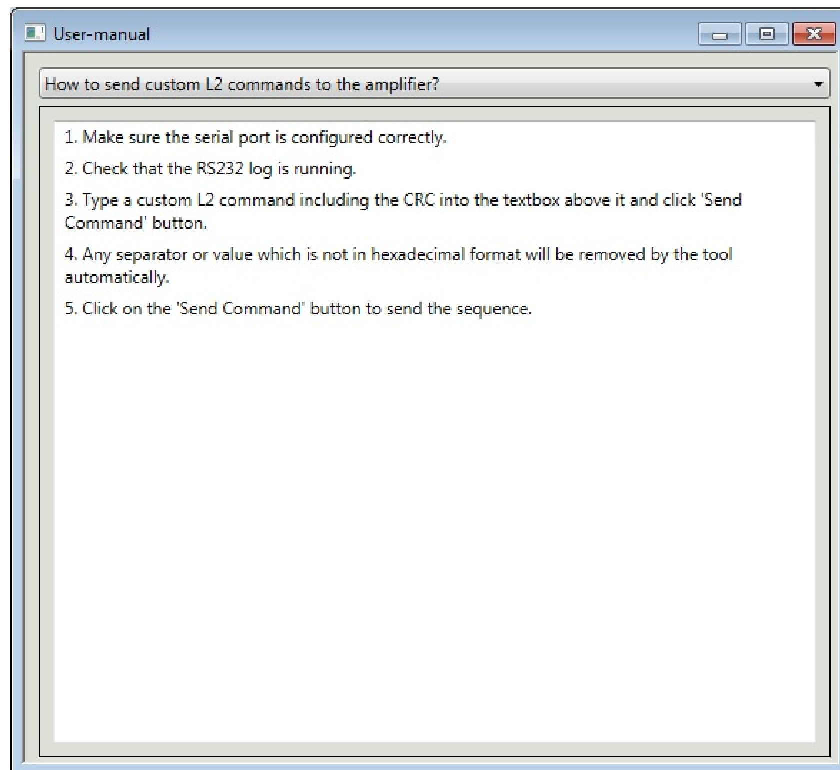

Figure 14: User-manual

6 Testing

Testing of the application mostly took place during the development process. The most common technique was functionality verification of every module directly after the implementation. In general, testing can be separated into 4 bigger parts.

The first par was unit testing, in which every module was tested separately. For example, data acquisition from the Excel was verified by checking whether each parameter in the result structure of the memory record was stored and whether the array elements were sorted according to the needed parameter. Such verification was mainly done with the help of the GUI and allowed to test two major parts concurrently. Logging the intermediate variable values was added directly to the GUI debug window to track every checkpoint of the functionality.

The unit testing was followed by integration testing. During this process composition of the different modules and their intercommunication was checked. All previously generated data has to be valid and thus it was checked by the module who had it as an input. Similar to the unit tests, the GUI debug windows were used to track the steps, checkpoints, values of the variables and so on.

The third point was verification of the functionality in online mode – meaning connected with the audio amplifier. The RS232 serial port log was mainly used to check whether the generated L2 requests reached the destination and whether they were responded by the amplifier. Successful communication guaranteed that the RS232 interface was correctly configured and fully functional. No response or CRC check failure reported by the amplifier meant that the L2 command generator was not working correctly. This fact was taken as a main proving point. In case of correctly received L2 response, the parsed data was checked against the EEPROM value that was initially pre-flashed into the amplifier EEPROM memory. This way it was assured that the data was filtered out from the RS232 traffic properly and saved into the result structure. The parsed values were displayed to the GUI to simultaneously also verify the functionality

of the GUI. The same way was used to prove that saving the initial memory state was working. In more detail, a generated back-up file was compared against exactly the same file which was initially flashed to the amplifier. Furthermore, some values of the memory were changed and the complete readout repeated. The outcome was compared once again against the initial one to see whether only the modified values were different.

The final step was release testing. It takes place for every new release version by responsible developer and also with the support from the SW testing team. Here, all basic functionalities of the tool are checked. This test covers verification of the RS232 configuration, project selection, reading and modifying the EEPROM values, saving the initial memory state and comparing it against the initial image. Furthermore, correct enabling and disabling of the GUI components is verified. For example, it must not be possible to send memory read request before the COM port is open.

7 Future development

The application for accessing and changing the EEPROM memory content of the audio amplifier is still under development. Several suggestions have been brought out by the users already in order to improve the tool.

The biggest functional addition is introducing a command line interface as currently it can be used only through the GUI. The main goal of this change is allowing automation, for instance using in different scripts or in build server. An example of such usage can be automatic verification of the complete EEPROM memory content.

The second functional change request is detecting an invalid or corrupt database file. Currently the application does not have a functionality to detect any errors or even an invalid Excel file if such happens to be supplied. In order to avoid it from happening, the tool is being released together with a verified database file.

Additionally, it has been considered about building the SWDL of the audio amplifier into the tool. Ideally, it could be possible to flash any component of the audio amplifier including also the EEPROM memory. Having this functionality available, the user would be able to restore the initial memory content without using the other SWDL tools.

Another request concerns the appearance. Some users have claimed that the colour of the GUI is not the best. Thus, in the future it will become possible to change the background colour of the GUI according to the wish of the tool user.

8 Summary

The purpose of this thesis was to provide access and modification possibility of the audio amplifier's EEPROM memory content. For achieving the set goal, a special software application with a graphical user interface was designed, developed, tested and released towards the customer.

The requirements for the tool were defined in co-operation between the production team of the amplifier and the SW development team. The primary target was providing the read/write possibility, but additionally the tool had to be equipped with a self-explanatory, simple and robust graphical user interface.

The input parameters for generating the L2 and L3 request commands are acquired automatically by the tool from the Excel file by using the Excel Data Reader library. All the slow processes are solved by running them in the separate threads. The parameters of each EEPROM record are stored in a custom structure and all the structures are saved into a list. The request commands are created by the L2 and L3 generators and sent towards the amplifier through the RS232 serial port. The response is parsed for the valid content and displayed for the user on the GUI for modifications.

Due to changing functionality the tool is capable of creating a backup image of the current EEPROM memory state. The image is generated by using the same functionality as for the single entry requests. The copy is stored in several formats which are directly usable by the SWDL tools of the amplifier.

The graphical user interface is developed by using the WPF framework and XAML language. It is divided into 4 sections of different functional purpose. It is equipped with the tooltips providing hints about how to use the tool correctly.

The outcome of this thesis is a working, thoroughly tested and released software application with the GUI providing requested functionality and fulfilling all customer requirements. However, the tool can still be extended with several proposed

suggestions. During the development the author gained a lot of practical experience with the C# and XAML languages, WPF framework, multi-threading, Excel parsing library and general rules of the PC application development process.

References

[1] History of Automotive Audio [WWW] http://www.pcmag.com/article2/0,2817,2399878,00.asp [website]

[2] uCOS-II Operating System [WWW] https://www.micrium.com/rtos/kernels [website]

[3] SerialPort Class description [WWW] https://msdn.microsoft.com/en-us/library/system.io.ports.serialport(v=vs.110).aspx [website]

[4] How RS232 works [WWW] http://www.best-microcontroller-projects.com/how-rs232-works.html [website]

[5] Motorola S-Format [WWW] http://www.lucidtechnologies.info/moto.htm [website]

[6] Using BackgroundWorkers [WWW] https://msdn.microsoft.com/fr-fr/library/cc221403(v=vs.95).aspx [website]

[7] WinForms and WPF description [WWW] http://blackthorn-vision.com/let-the-battle-begin-winforms-vs-wpf-2/ [website]

[8] WinForms and WPF description [WWW] https://www.infragistics.com/community/blogs/devtoolsguy/archive/2015/04/17/windows-presentation-foundation-vs-winforms.aspx [website]

[9] XAML Overview [WWW] https://msdn.microsoft.com/en-us/library/ms752059(v=vs.110).aspx [website]

[10] ToolTip class [WWW] https://msdn.microsoft.com/en-us/library/system.windows.forms.tooltip(v=vs.110).aspx [website]