Faculdade de Engenharia da Universidade do Porto

Development of a Flexible Communication Framework to support Testing and Production of Automotive Products

Manuel Pedro Rebelo

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Paulo Portugal (PhD)

Co-Supervisor: Eng. Nuno Silva

24th June 2019

© Manuel Pedro Rebelo, 2019

Abstract

In the Automotive Industry, the various components are carefully tested to ensure the specified quality levels and consequently the expected performance and reliability. Among these components are the Vehicle Infotainment Systems, which use their own interfaces such as RS-232, Ethernet and Controller Area Network (CAN) to communicate with test systems. Therefore, during the development and production processes of Vehicle Infotainment Systems, there is a need to use different tools and work sets due to the various types of involved communication protocols. These protocols should be integrated to establish connections and exchange data between systems on different communication interfaces and consequently optimize all involved test processes, and minimize potential failures.

In this context, the present Dissertation aimed to present the work developed during an internship at APTIVPort - Services, S.S., in the Department of Manufacturing Test Engineering with the main objective to standardize and to method the practices actually used to establish communications between Infotainment Systems and test systems, through the creation of a framework to characterize, establish, control and monitor RS-232, Ethernet and CAN communications. The framework should consist of several standardized and integrated modules with different objectives that exchange relevant data between their several functions.

In this scope, this Dissertation intends to address the need of a framework that integrates those tools and facilitates the monitoring of different connections. Thus, Product Communication Framework main objective is to permit the use of the same functions to establish connections of different types, integrating tools that have similar purposes. The framework can be used to test equipment and control it, sending commands and receiving messages through the established connections. Analyzing a certain connection is also guaranteed by the framework, which permits to consult all the exchanged messages and signal messages with defined patterns.

Additionally, the framework permits to remotely consult the content of a established connection between equipment and systems that use the framework's functions to communicate over RS-232, Ethernet or CAN.

The integration of the different tools was done incrementally, concretely adding first functions to communicate over RS-232, then functions to communicate over Ethernet and finally functions to communicate over CAN to a Dynamic Link Library developed using *Visual Studio*. Once this integration was done, a remote monitor module could be developed, consisting in a *Windows* service and a user interface, also developed using *Visual Studio*'s tools.

Finally, the Product Communication Framework is ready to use in real test systems since it was tested in a controlled environment because the failure of the framework, when used in Production lines' test systems, can lead to the rejection of functional equipment.

Keywords: Infotainment; In-Vehicle Infotainment System; Test Systems; Communication Framework; RS-232; Ethernet; CAN; Visual Studio.

Resumo

Na Indústria Automóvel, os vários componentes são testados cuidadosamente para garantir o níveis de qualidade e, consequentemente, o desempenho e o comportamento esperados. Entre estes componentes estão os Sistemas de *Infotainment*, que usam as suas próprias interfaces, como RS-232, *Ethernet* e *Controller Area Network* (CAN) para comunicar com os sistemas de teste. Portanto, durante desenvolvimento e processos de produção de Sistemas de *Infotainment*, é necessário usar ferramentas diferentes devido aos vários tipos de protocolos de comunicação envolvidos. Estes protocolos devem ser integrados para estabelecer conexões e trocar dados entre sistemas com diferentes interfaces de comunicação e, consequentemente, otimizar os processos de teste envolvidos e minimizar possíveis falhas.

Neste contexto, a presente Dissertação teve como objetivo apresentar o trabalho desenvolvido durante um estágio na APTIVPort - Services, S.S., no Departamento de Engenharia de Testes com o principal objetivo de padronizar e metodizar as práticas atualmente utilizadas para estabelecer comunicações entre Sistemas de *Infotainment* e sistemas de teste, através da criação de um *framework* para caracterizar, estabelecer, controlar e monitorizar comunicações RS-232, Ethernet e CAN. O *framework* deve ser constituído por vários módulos normalizados e integrados, com objectivos diferentes, que trocam dados relevantes entre as suas diversas funções.

Neste âmbito, este trabalho pretende abordar a necessidade de uma *framework* que integra ferramentas e facilita a monitorização de diferentes conexões. Assim, o objetivo principal da *framework* é permitir o uso das mesmas funções para estabelecer conexões de diferentes tipos, integrando ferramentas que têm propósitos semelhantes. A *framework* pode ser usada para testar o equipamento e controlá-lo, enviando comandos e recebendo mensagens através das conexões estabelecidas. A análise de uma determinada conexão também é garantida pela *framework* que permite consultar todas as mensagens trocadas e assinalar mensagens com determinados padrões.

Além disso, a *framework* permite consultar remotamente o conteúdo de uma conexão estabelecida entre equipamentos e sistemas que usam as funções da *framework* para se comunicar através de RS-232, Ethernet ou CAN.

A integração das diferentes ferramentas foi feita de forma incremental, concretamente adicionando primeiro as funções para comunicar utilizando RS-232, depois as funções para comunicar utilizando Ethernet e finalmente as funções para comunicar utilizando CAN a uma biblioteca de *links* dinâmicos (DLL) desenvolvida usando o *Visual Studio*. Assim que essa integração foi feita, um módulo de monitorização remota pode ser desenvolvido, consistindo num serviço *Windows* e numa interface de utilizador, também desenvolvida usando as ferramentas *Visual Studio*.

Finalmente, a denominada *Product Communication Framework* foi concluída e pode para ser utilizada em sistemas de teste reais, tendo sido testada em ambiente controlado uma vez que a falha da *framework*, quando usada em sistemas de teste de linhas de produção, pode levar à rejeição de equipamentos conformes.

Palavras-chave: Infotainment; Sistema de Infotainment de Veículos; Sistemas de teste; Framework de Comunicação; RS-232; Ethernet; CAN; Visual Studio.

Acknowledgments

This Dissertation marks the conclusion of a cycle in my life that would have not been possible without the effort and motivation that I have received all along these years.

Therefore, I would like to thank:

First and above all to my Parents, for giving me the possibility of receiving the best education and giving me values that make me a better person, showing me that without work and humbleness nothing is accomplished. There are not enough words to show how grateful I am.

To Artur, my brother, for always being a role model and giving me all the support and advises that made me not commit errors that otherwise I would have committed.

To Paula, my girlfriend and best friend, for giving me all the patience and love that made me never give up from my objectives, supporting and inspiring me everyday.

To Professor Paulo Portugal, my supervisor, who I had also the privilege of being my teacher during my course, for all the time spent with me and guidance, always giving the best advises to make this Dissertation a better one.

To Eng. Nuno Silva, my co-supervisor, for receiving me at Aptiv, giving me the possibility of develop this project, and for all the time and knowledge shared with me, making possible to lead the project to success.

To Aptiv's collaborators from Braga, specially from Manufacturing Test Engineering Department, for always making me feel integrated in the company during the past months.

To all of you my sincere gratitude,

Manuel Pedro Rebelo

"If the doors of perception were cleansed
everything would appear to man as it is, Infinite."

William Blake

Contents

Abstract i

Resumo iii

Acknowledgments v

List of Figures xii

List of Tables xiii

Abbreviations, Symbols and Units xv

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 3
- 1.4 Dissertation's structure . . . 4

2 Problem domain and technological background 5
- 2.1 Equipment description . . . 5
  - 2.1.1 Aptiv's products . . . 5
  - 2.1.2 Test proceedings . . . 7
  - 2.1.3 Signal generators . . . 9
- 2.2 Interfaces' characteristics . . . 11
  - 2.2.1 RS-232 . . . 12
  - 2.2.2 Ethernet . . . 14
  - 2.2.3 CAN . . . 19
- 2.3 Problem description . . . 24

3 Proposed Solution 25
- 3.1 Requirements . . . 25
- 3.2 Description . . . 27
  - 3.2.1 Main modules . . . 29
- 3.3 Conclusions . . . 32

4 Implementation 33
- 4.1 Product Communication Framework DLL . . . 33
  - 4.1.1 XML configuration file and XML Parser . . . 41
  - 4.1.2 Communication functions . . . 43

4.2 Product Communication Monitor . . . 56
4.2.1 Product Communication Monitor Service . . . 56
4.2.2 Product Communication Monitor UI . . . 57

5 Experimental Validation **61**
5.1 Testing Framework . . . 61
5.1.1 NI TestStand sequences . . . 61
5.1.2 Validation of the Remote Monitoring . . . 64

6 Conclusion and future work **69**
6.1 Conclusions and achievements . . . 69
6.2 Future work . . . 70

A Product Communication Configuration File Example **71**

References **73**

List of Figures

2.1 MIB3-Top general view. 6
2.2 MIB3-Top back unit connectors. 6
2.3 MIB3-Top Main-Connector. 7
2.4 Example of tests performed in Infotainment Systems, when supplying them with RF signals. 8
2.5 MIB3-Top Cross Unit fixture. 8
2.6 MIB3-Top Debug-Connector. 9
2.7 Rohde & Schwarz SMBV100A front and back view. 10
2.8 Rohde & Schwarz SFC Compact Modulator front and back view. 10
2.9 Rohde & Schwarz SFE100 front and back view. 11
2.10 DB-9 Male Connector Pin out, [12]. 12
2.11 Example of connection between DTEs using RS-232. 13
2.12 UART Frame. 14
2.13 OSI model and Ethernet corresponding sub layers. 15
2.14 IEEE 802.3 frame. 17
2.15 The evolution of the Ethernet networks. 18
2.16 Physical CAN Connection according to ISO 11898, [18]. 20
2.17 Schematic and application of a single star topology, [17]. 20
2.18 Schematic and application of a twin star topology, [17]. 21
2.19 Schematic and application of a linear bus topology, [17]. 21
2.20 Schematic and application of a hybrid topology, [17]. 21
2.21 Structure of a CAN Data frame, [17]. 23

3.1 Diagram of the proposed solution. 29
3.2 PCF DLL general view. 30
3.3 Diagram of the Product Communication Monitor. 31
3.4 Example of application for the proposed solution in test systems. 32

4.1 PCF DLL Class diagram. 34
4.2 **Interface Class**. 34
4.3 Example of interaction between an application and a device using PCF DLL's methods. 38
4.4 **Log Class**. 39
4.5 Example of interaction between PCF DLL's Classes. 40
4.6 **SerialDevice Class**. 44
4.7 Initializing a RS-232 Device. 45
4.8 Example of interaction between an application and device using PCF DLL's methods to communicate over RS-232. 46

4.9 Server/Client relationship example for connection-oriented sockets. 47
4.10 **EthernetDevice Class**. 48
4.11 Initializing an Ethernet Device. 49
4.12 Example of interaction between an application and device using PCF DLL's methods to communicate over Ethernet. 51
4.13 XL Driver Library function calls for CAN applications, [32]. 52
4.14 **CANDevice Class**. 53
4.15 Initializing a CAN Device. 54
4.16 Example of interaction between an application and device using PCF DLL's methods to communicate over CAN. 55
4.17 PCMUI homepage. 58
4.18 PCMUI connection to remote system page. 58
4.19 Example of interaction between PCMUI and PCMS. 60

5.1 Test sequence used for testing the framework's functionalities (RS-232 Example). 62
5.2 Test sequence's step used for testing the framework's filtering methods. 64
5.3 Checking PCCF content using *PuTTY*. 65
5.4 Checking available interfaces to monitor using PCMUI. 65
5.5 PCMUI in stream mode. 66
5.6 PCMUI with PCL in memory. 66
5.7 Checking available product numbers to monitor using PCMUI. 67

List of Tables

2.1 IEEE 802.3 Transmission Medium Specifications, [15]. 16

4.1 PCF DLL's Constructor. 35
4.2 PCF DLL's `Init` method. 35
4.3 PCF DLL's `Read` methods. 36
4.4 PCF DLL's `Send` methods. 36
4.5 PCF DLL's `Send_Read` methods. 37
4.6 PCF DLL's `CheckState` method. 37
4.7 PCF DLL's `Stop` method. 38
4.8 PCF DLL's Filter methods. 39

Abbreviations, Symbols and Units

ACK	Acknowledgment
AM	Amplitude Modulation
API	Application Programming Interface
bps	bits per second
CAN	Controller Area Network
CAN FD	Controller Area Network with Flexible Data-Rate
CD	Compact Disc
cm	centimeters
CMOS	Complementary Metal-oxide Semiconductor
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense with Multiple Access and Collision Detect
CTS	Clear to Send
DAB	Digital Audio Broadcasting
dBm	Decibel milliwatt
DCD	Data Carrier Detect
DCE	Data Circuit-Terminating Equipment
DLC	Data Length Code
DLL	Dynamic-link Library
DSR	Data Set Ready
DTE	Data Terminal Equipment
DTMB	Digital Terrestrial Multimedia Broadcast
DTR	Data Terminal Ready
EIA	Electronic Industries Alliance
EMC	Electromagnetic Compatibility
EMI	Electromagnetic interference
EOF	End Of Frame
ESD	Electromagnetic Discharge
FCS	Frame Check Sequence
FM	Frequency Modulation
GHz	Gigahertz
GND	Common Ground
GNSS	Global Navigation Satellite System
HD	High Definition
Hz	Hertz
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
IFG	Interframe Gap
IFS	Inter Frame Space

I/O	Inputs/Outputs
IOC	Input/Ouput Controller
IP	Internet Protocol
ISO	International Organization for Standardization
Kbaud	Kilobaud
KHz	Kilohertz
LAN	Local Area Network
LLC	Logical Link Control
LTE	Long Term Evolution
m	Meter
MAC	Medium Access Control
Mbaud	Megabaud
Mbps	Megabits per second
MHz	Megahertz
NI	National Instruments
OSI	Open System Interconnection
OUI	Organizationally Unique Identifier
PCCF	Product Communication Configuration File
PCF	Product Communication Framework
PCFC	Product Communication Framework Class
PCL	Product Communication Log
PCM	Product Communication Monitor
PCMS	Product Communication Monitor Service
PCMUI	Product Communication Monitor User Interface
RDS	Radio Data System
RF	Radio Frequency
RI	Ring Indicator
RTR	Remote Transmission Request
RTS	Request To Send
RD	Received Data
SAE	Society of Automotive Engineers
SD	Secure Card
SIM	Subscriber Identity Module
SFD	Start Frame Delimiter
SDARS	Satellite Digital Audio Radio Service
TCP	Transmission Control Protocol
TIA	Telecommunications Industry Association
TTL	Transistor-transistor Logic
TV	Televison
TD	Transmitted Data
UART	Universal Synchronous Receiver/Transmitter
UDP	User Datagram Protocol
UI	User Interface
USB	Universal Serial Bus
V	Volt
VICS	Vehicle Information and Communication System
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter 1 is divided in four sections that have as objective to present the context, the motivation, the objective for this work and its structure.

In the first section is presented the context in which this work fits.

The second section presents the motivation for developing this work in the context mentioned before.

The third section is used for giving a description of the main objectives of the present work.

Finally, a fourth section has as objective to present the structure of the Dissertation and it can be used as a guide for consulting this document along with the index.

1.1 Context

Since the automobile first appeared, the Automobile Industry has always involved several sectors and branches of activity.

In the past, vehicles were presented as simple machines, focusing the attention on their mechanical characteristics as power rates and speed limits. Later, the advertisements focused on the visual aspect of the vehicles, presenting them as a ”singular piece of art”.

More recently, the focus was made in the vehicles’ impact on the environment as society has started to worry about the impact in the environment and new standards, including legal requirements, were issued to reduce pollution caused by automobiles [1].

Nowadays, brands show their new automobiles focusing their ads in the technological capabilities that their Infotainment Systems have to offer. This systems’ functionalities appear to be an important differentiation for the final consumer. In other words, a person who buys a new car choose it comparing the technological functionalities that the automobile has to offer, in a very similar way to when someone chooses a new smart phone. This can be observed comparing old advertisements with the new ones.

According to a survey, ”by 2020 it is expected that up to 80% of new vehicles will be connected to digital services”, [2].

The simple auto-radio has now evolved to a complex computer, the Infotainment System, that combines entertainment and information delivery for drivers and passengers [3], keeping the main functionalities of the old auto-radios but has now to deal with various challenges. Some of this challenges are related to the interconnection of devices, using different protocols, wide temperature variations, vibrations caused by the movement of the vehicle and Electromagnetic interference (EMI) that tends to increase with the inclusion of more and more devices in the system.

The lack of ability to self-monitor and actuate to eliminate or reduce certain issues can lead to the failure of certain functionalities of this products which, ultimately, can correspond to the failure of the vehicle as a whole. When this occurs, the costs for reparation can be significant, risking the products' profitability.

In order to avoid failures, during the development and production of Infotainment Systems, various tests are made intensively. Some of these tests are made when the products are fully assembled, consisting in sending diagnostic commands using different interfaces and evaluate the products' response to check if the parameters evaluated are within the corresponding requirements previously defined and that must be complied.

Within these tests, different tools and work sets are required due to the varied types of communication protocols involved. Such as the multiple layers of communication that exist to control and monitor the different devices that are interconnected with the Infotainment Systems.

Normally, the tools used to communicate with the products are not integrated and with the appearance of new ones, the creation of new tools come along. Thus, there is the need to create a standardized tool set in order to communicate with different products over different interfaces.

1.2 Motivation

During the months of July and August of 2018, the author of the present work had the opportunity to hold a Summer Internship at the Multinational Company Aptiv, specifically at its Industrial Site located at Braga city.

In scope of the objectives that were defined for the Internship versus the results to be achieved, the author was able to develop software to support the functional tests of products such as those previously mentioned (Infotainment Systems).

It was an Internship developed in an industrial environment, in real working context and which results met expectations, proving to be of effective use and application in the Company. Admittedly, the Internship allowed the author to apply knowledge assimilated at the Electrical Engineering Course and acquire new knowledge, theoretical and practical.

Besides that, it is important to note that in the different phases of: design, development, testing and production of new products, in the circumstance - auto-radios - , a variety of tools are needed. This tools are usually created from scratch so that all these phases develop and succeed, always in full compliance with the internal and customer requirements.

When new tools are created, they are usually intended to be used only with one product. There are always different approaches to implement those tools and instead of being used by multiple

persons they are used only by the person that created them, since there is difficulty in sharing information and lack of time to teach others how to use them.

There is therefore a need to standardize and to method the process that results from the emergence of new versions and types of products with some of the same / different features / functionalities that are different from previous ones.

It was within this need, to standardize and to method the process of developing new tools to communicate with different Infotainment Systems, that the opportunity of conducting the Master's Dissertation was identified. Acknowledged that the author of this work also has as motivator the internal context of Aptiv, a Multinational Company that practices research, design and develops products and processes from scratch and innovates in an Industry - the Automotive Industry - , highly demanding, competitive and in permanent technological evolution with Customers all around the World.

1.3 Objectives

The main objective of this work was to standardize and to facilitate the practices used to establish communications with Infotainment Systems, through the creation of a framework to characterize, establish, control and monitor RS-232, Ethernet and Controlled Area Network (CAN) communications. Being the mainly used to conduct tests when the products are fully assembled, these interfaces have singular characteristics that demand the use of specific hardware and software that are not integrated in the same work set, despite the interfaces mostly being the same for different products.

Thus, the framework is set to be used by test engineers in their test systems that verify the products requirements and to communicate with devices responsible to generate the various signals distributed in the test lines, such as Digital Audio Broadcasting (DAB), Satellite Digital Audio Radio Service (SDARS) or Vehicle Information and Communication System (VICS). Additionally, the framework can be used to create new applications where there is the need to establish communications using the mentioned interfaces, for example in laboratory.

With these purposes, it is important to note that the framework should consist of several standardized and integrated modules with different objectives that exchange relevant data between their different functions.

A first module concerns the characterization of the environment where the framework will be integrated. The main objective is to define the characteristics of the product to be tested, the characteristics of the test equipment, for example signal generators, and the characteristics of the interfaces used. This way, it is possible that this module also defines the corresponding protocols and the messages' types to be used in the communication between the various elements involved.

A second module concerns the functions responsible for establishing communication channels, sending and receiving messages and also waiting for possible messages. Given the objective of this second module, it must be divided into others, in order to guarantee the multiplicity of protocols covered by this framework, ensuring its flexibility.

Derived from the importance of monitoring an established communication, it is important to ensure that all communications can be analyzed. All contents shall be saved and the access to them shall be also guaranteed. Besides that, the framework must contain functions that allow filtering and signaling messages, which have particular interest in test systems where using those functions anomalies can be detected.

Finally, the framework shall allow users to remotely access data related to a particular communication being performed at another location, for example, test lines located at Production areas.

1.4 Dissertation's structure

The present Dissertation is structured in six chapters that are resumed in the following points:

i. the present chapter, 1, has as objective to describe the context, the objectives, the motivation for this Dissertation and its own structure;

ii. the second chapter, 2, has as objective to describe the involved equipment characteristics, test proceedings, the communication interfaces characteristics and the problem;

iii. a third chapter, 3, has as objective to describe the proposed solution as its requirements;

iv. a fourth chapter, 4, has as objective to describe how the solution's different modules were implemented to address to requirements mentioned before;

v. a fifth chapter, 5, has as objective to describe how the developed framework was tested; and

vi. a final chapter, 6, has as objective to describe the achievements of this work and future work proposals.

At the end of this document, the references and appendix can be found.

Chapter 2

Problem domain and technological background

This chapter 2 is divided in three main sections used to describe the involved equipment characteristics as its test proceedings, the interfaces' characteristics and the problem to be solved.

The first section consists in a general description of the equipment conceived in Aptiv, its characteristics and how this equipment is tested. It is presented a description of test proceedings and equipment used during those tests.

In the second section, a description of the involved communication interfaces used for testing the equipment is made, more precisely RS-232, Ethernet and CAN.

Finally, in the third section, a description of the problem to be solved is presented.

2.1 Equipment description

2.1.1 Aptiv's products

Being present all around the World, with 147000 collaborators in 45 countries, of which 16000 are Engineers and Scientists (approximally 11%), Aptiv is responsible for developing and producing products for some of the main automobile manufacturers such as Audi, Volkswagen, Ferrari and Porsche, more precisely Infotainment Systems, User Experience, Connectivity and Security, Safety Electronics and Autonomous Driving. Presents itself as a "global technology company that develops safer, greener and more connected solutions enabling the future of mobility", [4].

The Company's Braga site designs several Products but only two were conceded to develop this work. One of the products was MIB3-Top (fig.2.1), a model present in Audi's A3, A4 and Q7 vehicles and Porsche's J1 vehicles. The other product was a Volvo Cluster, still in pre series production, that will be present in the new Volvo FH12 truck model and, for this reason, a description of this equipment will not be included in this work.

Figure 2.1: MIB3-Top general view.

MIB3-Top is an Infotainment Central Unit with no Compact Disc (CD), no Secure Digital (SD) card, no Subscriber Identification Module (SIM) function, no buttons neither illumination. To interact with it, a display shall be connected to the MIB3-Top in the connectors 5 and 7 present in the figure 2.2. The connector 6 shall be connected to the vehicle display that works as tachometer. MIB3-Top does not include a Bluetooth or Wireless Local Area Network (WLAN) antenna, but in the Bluetooth case an antenna can be connected to the connector 9.

This unit has one multimedia board with a Samsung processor and it has one main board with a TI J6 processor and a Renesas Input/Output Controller (IOC) processor.

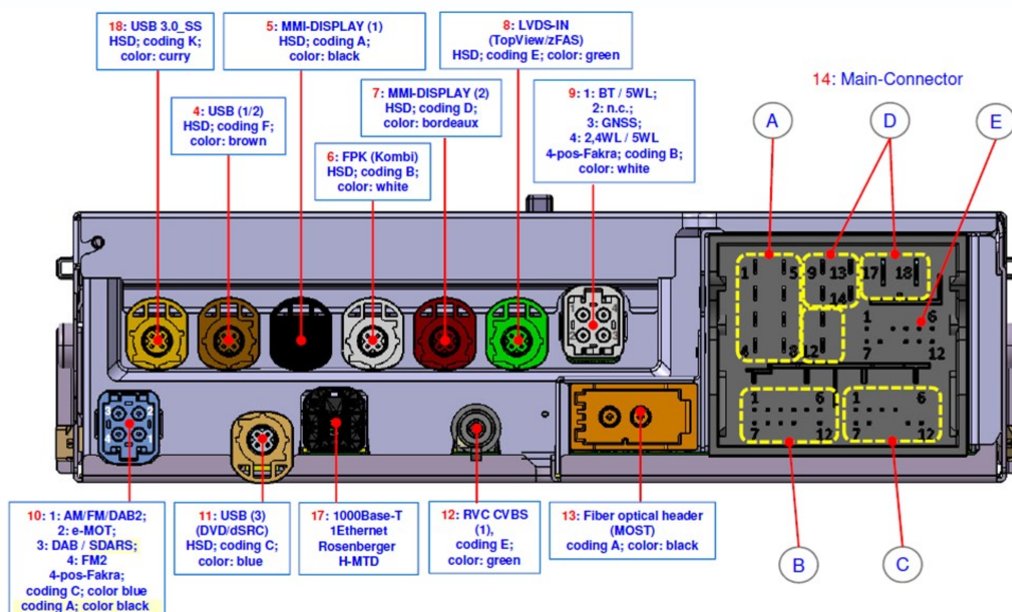


Figure 2.2: MIB3-Top back unit connectors.

Besides the connectors for Universal Serial Bus (USB) interfaces (4, 11 and 18), Ethernet (17), Radio Frequency (RF) signals (10), among others, MIB3-Top Main-Connector (14) (fig.2.3) is of particular importance since it is used to power the unit and to link it to sensors. It is also used to connect and supply analog and digital audio systems and to couple the unit to CAN bus present in the vehicle.

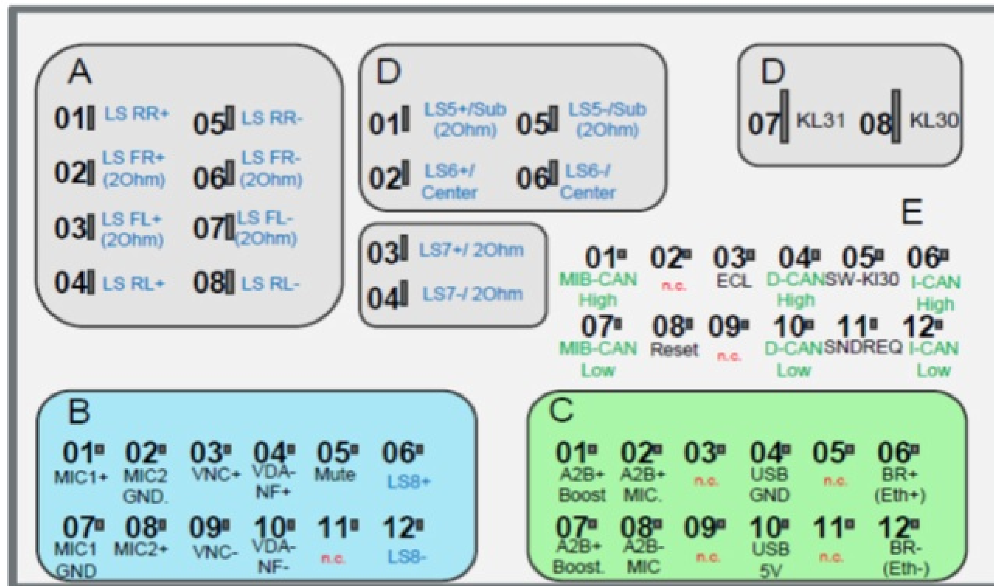


Figure 2.3: MIB3-Top Main-Connector.

2.1.2 Test proceedings

The framework to be developed is intended to be included in test systems responsible for the Final Tests of the units produced. Units are placed in cross units fixtures that establish the connection between the different unit interfaces and the test system. Then, test systems use *National Instruments (NI) TestStand* [5] to run test sequences responsible to verify the state of the unit being tested.

These tests contain segments in which test systems communicate with the units using specific interfaces. Using these interfaces, test sequences start by commanding the product to go to diagnostic mode, by sending a special diagnostic command. Then, the modules present in the Infotainment Systems are ready to be queried with diagnostic commands to which they respond. The responses are then processed by the functions present in the test sequences, verifying the parameters being evaluated.

An example of the tests (fig. 2.4) performed is when cross units fixtures are supplied with RF signals, that for they turn, they supply to the unit being tested. Once those signals are generated with known values, it is possible to compare them with the measured ones, checking if they either match or not.

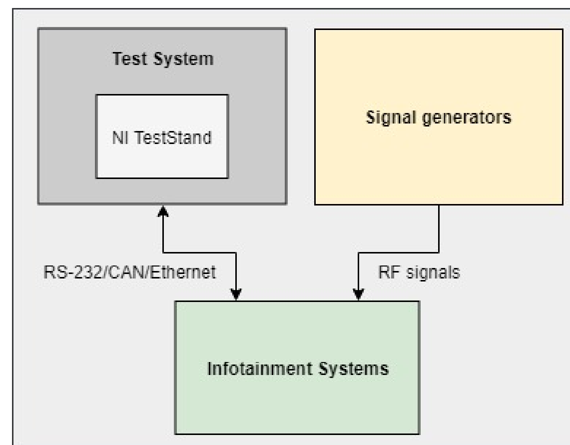


Figure 2.4: Example of tests performed in Infotainment Systems, when supplying them with RF signals.

Furthermore, these test sequences permit the execution of different tests in parallel, testing independent modules of the product at the same time, in order to optimize the time and the available resources.

When the test is finished, the unit is released by the fixture and manually removed from it, going further in the production process if it passed all the tests or going to laboratory to be analyzed otherwise.

Test systems present in the production line of the MIB3-Top (fig.2.5) use RS-232 and also use, more recently, Ethernet to RS-232 adapters such as *Brainboxes* ES-257 [6] (a device that enables the connection of two RS232 interfaces to an Ethernet network) to communicate with the product's processors being tested, diagnosing the product's components states, checking if they comply with the specific defined requirements.

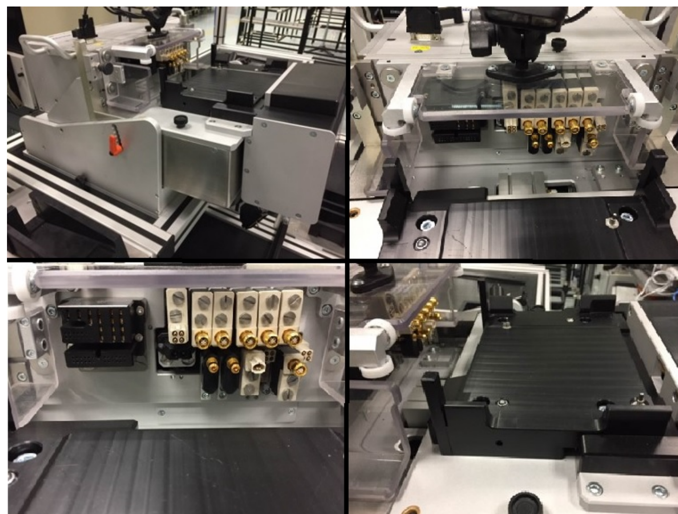
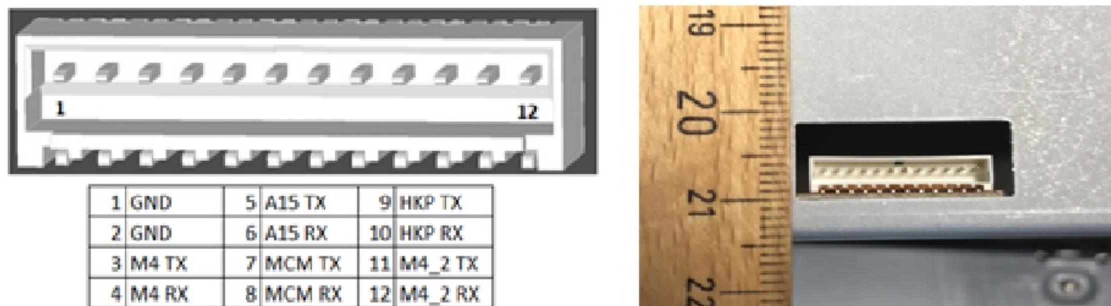


Figure 2.5: MIB3-Top Cross Unit fixture.

For MIB3-Top, the communication is established via RS-232 using the connector (fig.2.6) present in the front of MIB3-Top units, specifically created for debug and diagnostic purposes.



1	GND	5	A15 TX	9	HKP TX
2	GND	6	A15 RX	10	HKP RX
3	M4 TX	7	MCM TX	11	M4_2 TX
4	M4 RX	8	MCM RX	12	M4_2 RX

Figure 2.6: MIB3-Top Debug-Connector.

It is important to notice that diagnostic commands sent to products' RS-232 interface have a specific format, concretely composed by a prefix field, a group identification field, a test identification field, an operation field, a status information field, a number of bytes field and data bytes field. Depending on the operation field, a check sum field can be added to the command.

In the Volvo Cluster test systems, diagnostic commands are sent to the product over it's CAN interface. In both cases, the devices respond to the commands with pre-defined responses that make the test systems able to verify different requirements that must be complied by the product being tested. Since this Volvo Cluster has a screen, some of the tests consist, for example, in checking its brightness, by sending a command to set it to a pre-defined value, which is then verified using the test systems sensors.

In this case, as others where there is the need to establish CAN connections, Aptiv's Engineers use *Vector* devices as interface between their products and test systems, such as VN1610 [7], providing an USB interface to access a CAN network.

2.1.3 Signal generators

Since there are different RF signals used by Infotainment Systems, there is the need to use signal generators to generate these signals with known values in order to provide them across the test systems. Measuring a value as power or frequency of a signal in a device and comparing it to the generated one permits evaluate if the device's components are complying with the defined requirements.

Signal generators are used not only in test systems but also in laboratory where Engineers have the objective of test independently a certain module of a device, for example a SDARS module.

To facilitate the use of signal generators in the mentioned environments, remote controlling them is very useful. It is with this purpose that identifying the commonly used signal generators and respective characteristics is important. Therefore, the signal generators in use are:

- Rohde & Schwarz SMBV100A [8];

  The SBMV100A (fig.2.7) offers a very high output level and short setting times. At the same time, can be equipped with an internal baseband generator to allow generation of a number of digital standards (e.g. Long Term Evolution (LTE), LTE-Advanced, IEEE802.11ac, Global Navigation Satellite System (GNSS)). This device has a frequency range from 9 kHz to 6 GHz and a level range from -145 dBm to +18 dBm.

Figure 2.7: Rohde & Schwarz SMBV100A front and back view.

  It is used in Aptiv's test lines to generate SDARS and VICS signals and according to its manual it can be controlled remotely sending commands using its Ethernet interface.

- Rohde & Schwarz SFC Compact Modulator [9];

  The SFC Compact Modulator (fig.2.8) is a multi-standard signal source. It supports real time coding for all conventional digital and analog television (TV) and audio broadcasting standards such as Digital Terrestrial Multimedia Broadcast (DTMB), Digital Audio Broadcasting (DAB), Frequency Modulation (FM) and Radio Data System (RDS). The SFC Compact Modulator is equipped with a built-in computer. This device has a frequency range from 30 MHz to 800 MHz.

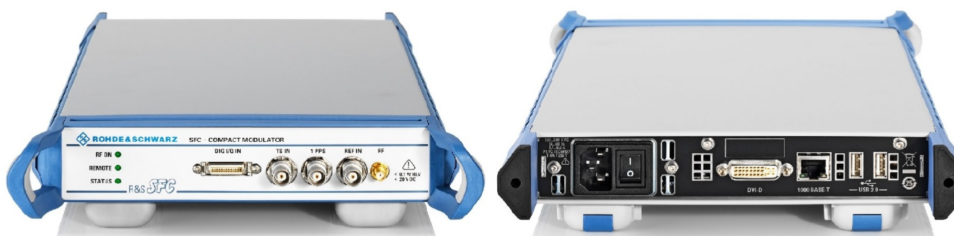


Figure 2.8: Rohde & Schwarz SFC Compact Modulator front and back view.

  It is used in Aptiv's test lines to generate DAB signals and according to its manual it can be controlled remotely sending commands using its Ethernet interface.

- Rohde & Schwarz SFE100 [10];

  The SFE100 (fig.2.9) is a multi-standard test transmitter providing real time coding for broadcast signals. It supports all common digital and analog TV standards and a number of audio broadcasting standards such as Amplitude Modulation (AM), FM and RDS. This device has a frequency range from 100 kHz to 2.7 GHz.

Figure 2.9: Rohde & Schwarz SFE100 front and back view.

  It is used in Aptiv’s test lines to generate High Definition (HD) radio signals and according to its manual it can be controlled remotely sending commands using its Ethernet interface.

It is possible to conclude that it is possible to control remotely all the mentioned signal generators using their Ethernet interface.

2.2 Interfaces’ characteristics

One of the main goals of the framework is to contend functions that make the user able to communicate with the automotive Products using a specified interface and, if desired, monitor that communication for analysis and testing proposes.

To build such functions, it is very important to understand the characteristics of the different involved layers in a specific type of communication. It is important to know how each one vary from one to another, besides the importance of knowing the expected behavior of the equipment chosen to communicate with.

In the following subsections are described the main concepts of the involved interfaces, specifically RS-232, Ethernet and CAN, which are the principal interfaces used to send diagnostic commands by test systems present in the production lines of the automotive products referenced above.

2.2.1 RS-232

RS-232 was first introduced in the 60's as a communication standard for serial transmission data. Currently, RS-232 complies with the standard for serial interfaces defined in TIA-232-F: Interface Between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) Employing Serial Binary Data Interchange, [11].

It was a standard widely used in computer systems, but meanwhile it has been substituted for other communication standards, mostly by USB. Nevertheless, RS-232 is still used in many applications, like in the scope of this work, to communicate with Infotainment Systems.

2.2.1.1 Physical interface

Despite the standard define different connectors, the focus will be given to the case of the equipment developed in Aptiv. In these devices, that use RS-232 to communicate, the connector used is a DB-9, (fig.2.10), which contains 9 pins but only three are used for each interface in the mentioned application. The connection is established between a DTE and a DTE.

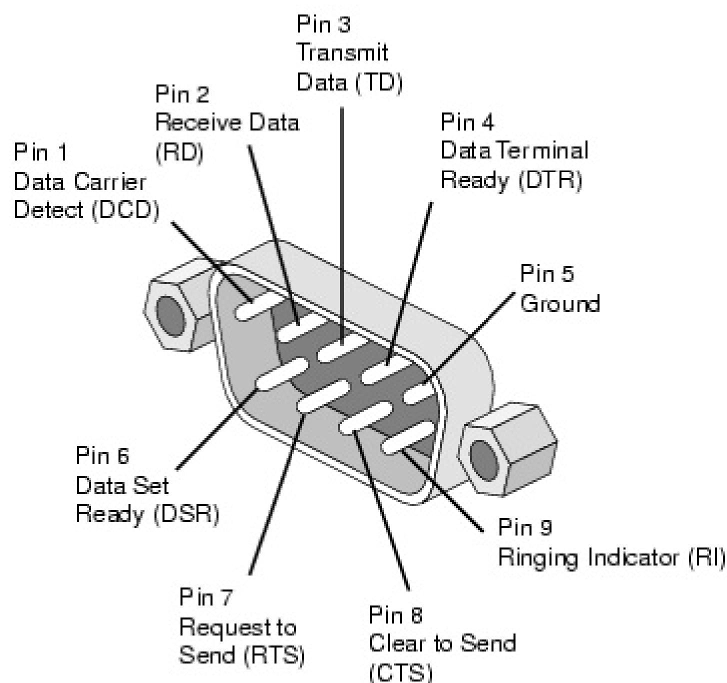


Figure 2.10: DB-9 Male Connector Pin out, [12].

In this type of applications (fig. 2.11), the Transmitted Data (TD) is responsible to carry data from a DTE to a DTE, the Received Data (RD) is responsible to carry data in the opposite direction and the Common Ground (GND) used as return path for the signals. The other pins can be short circuited in each side, namely connecting Request to Send (RTS) and Clear to Send (CTS) and

connecting Data Set Ready (DSR), Data Carrier Detect (DCD) and Data Terminal Ready (DTR). The remaining pin, Ring Indicator (RI), is not used.

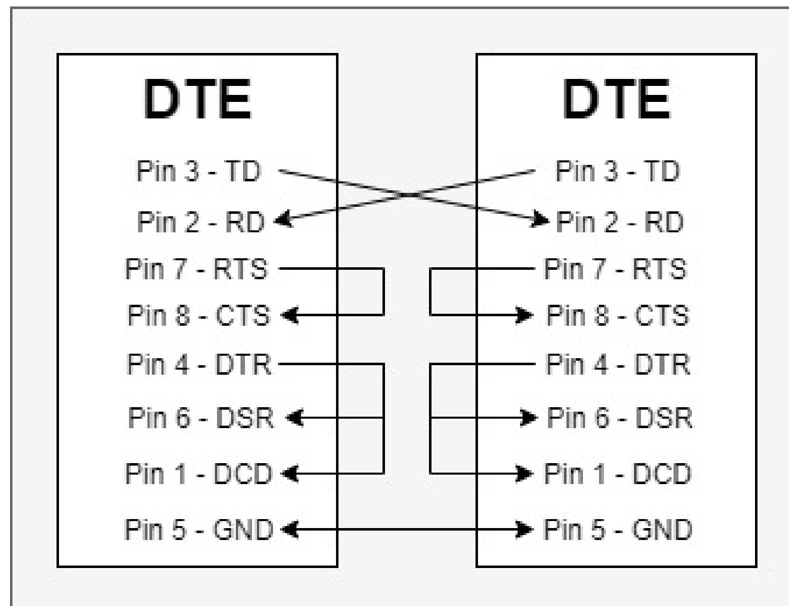


Figure 2.11: Example of connection between DTEs using RS-232.

Depending on the baud rate and cable type, the distance between two devices is limited.

2.2.1.2 Data and Control signals

Being a serial communication protocol, RS-232 transmits data one bit at a time, over a single wire. Using an unbalanced voltage transmission, bits are represented by voltages levels, positive and negative, and the communication can be bilateral, that is RS-232 can operate in a full-duplex mode in which is possible to devices connected using the standard to send and receive data at the same time. To operate in such way, three signals are essential, namely TD, RD and GND.

In RS-232 applications that use Universal asynchronous receiver-transmitter (UART), in which data is transmitted using frames (fig.2.12), it starts with the so called start bit that correspond to a bit with logic value 1. When a device is not transmitting, the logic value is 0. When a change occurs in this value, that indicates to the receiver that a frame is incoming. The following bits, defined by the User in the parameters of the communication, correspond to the data bits and are transmitted starting with the least significant bit and finishing with the most significant one.

If parity is used, it corresponds to the following bit. When a frame is received, the DCE counts the number of data bits with value equal to one and checks if it is either an odd or even number, if parity bit does not match it, a transmission error occurred.

Finally, the frame has one or two bits used to signal the end of the frame, called stop bits and correspond to bits with logic value 0.

Figure 2.12: UART Frame.

According to Axelson, J. [13], the remaining lines (RTS, CTS, DSR, DCD, DTR and RI) are flow-control and other status and control signals. ”The RS232 standard defines uses for all of the signals, but applications are free to use the signals in any way as long as both ends agree on what the signals mean. Many links use the RTS and CTS flow-control signals. In the most commonly used protocol, each computer uses an output bit to let the other computer know when it's okay to send data. The DCE asserts CTS when ready to receive data, and the DTE asserts RTS when ready to receive data. Before transmitting, a computer reads the opposite computer's flowcontrol output. If the signal's state indicates that the receiving computer isn't ready for data, the transmitting computer waits. In links that use DTR and DSR, each computer typically asserts its output on power up to indicate that the equipment is present and powered.”

RI, DCD, DSR and CTS are outputs on the DCE, and the first two can have any use if not used for their original purposes, that is detecting a ring signal on a phone line (RI) and detecting the presence of a carrier, respectively.

This author [13] also states that ”RS-232 logic levels are defined as positive and negative voltages rather than the positive-only voltages. Bits with logic value 1 are represented with voltages levels between -15 V and -3 V and bits with logic value 0 are represented with voltages levels between +3 V and +15V with respect to the common ground”.

2.2.2 Ethernet

Ethernet is a Local Area Network (LAN) technology that allows the connection between various devices within a relatively low-cost and flexible network system. It can be used in small networks as in very large networks, with different applications.

The Open Systems Interconnection (OSI) model describes how networking hardware and software work together, dividing their tasks into seven separate layers. OSI was developed in 1978 by International Organization of Standardization (ISO). The lower layers correspond to the standards used to describe how a LAN system moves bits around, in which Ethernet is included, while higher layers correspond to abstract notions as the reliability of data transmission and how the user gets the data. Ethernet is included in the two lower layers of OSI model, corresponding to physical and data link layers (fig.2.13).

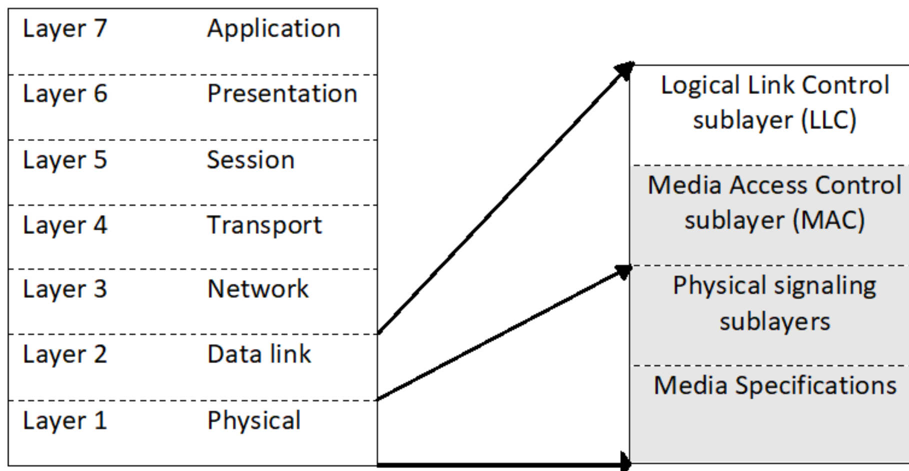


Figure 2.13: OSI model and Ethernet corresponding sub layers.

An Ethernet Local Network is composed by hardware and software that work together to deliver digital data between different devices. To make it possible, four basic elements are combined, composing an Ethernet System. This four building blocks are the follow [14] :

1. the *physical medium*, which consists of the cables and other hardware used to carry digital Ethernet signals between computers attached to the network;
2. the *signaling components*, which consists of a standardized electronic devices that send and receive signals over an Ethernet channel;
3. the *frame*, which is a standardized set of bits used to carry data over the System; and
4. the *Medium Access Control (MAC) protocol*, which consists of a set of rules embedded in each Ethernet interface that allow multiple computers to access the shared Ethernet channel in a fair manner.

On the other hand, the mentioned blocks can be grouped in two parts, one corresponding to the physical layer and other to the MAC layer. Signaling components and physical medium belong to the first one and the frame and MAC protocol belong to the other.

2.2.2.1 Physical medium and signaling components

The signaling components are all the hardware used to send and receive signals over the physical medium, which is all the hardware components responsible to carry the transmitted signals. This hardware differ depending on the type of cabling used and the Ethernet system's speed.

In Ethernet systems, signaling components refer to all the specific transceivers used in the Ethernet cards interfaces and repeater hubs, since they are responsible to generate, send and receive the signals used in the Ethernet system.

In the table 2.1 are presented the specifications of the most used Ethernet media systems, being the 10BASE-T and 100BASE-T the most common.

Table 2.1: IEEE 802.3 Transmission Medium Specifications, [15].

Characteristic	Ethernet Value	10base5	10base2	10baseT	10baseFL	100baseT
Data rate (Mbps)	10	10	10	10	10	100
Signaling method	Baseband	Baseband	Baseband	Baseband	Baseband	Baseband
Maximum segment length (m)	500	500	185	100	2000	100
Media	50-ohm coax (thick)	50-ohm coax (thick)	50-ohm coax (thick)	Unshield twisted-pair cable	Fiber-optic	Unshield twisted-pair cable
Topology	Bus	Bus	Bus	Star	Point-to-point	Bus

The 10BASE-T systems might use as signaling components an Ethernet interface with a built-in 10BASE-T transceiver or a repeater hub equipped with 10BASE-T ports, among others. While for building a 10BASE-T twisted-pair segment, the media components used are an unshielded twisted-pair cable with Category 3, which applies to 100 ohm unshielded twisted pair cables and associated connecting hardware whose transmission characteristics are specified up to 16 MHz, or better and a eight-position RJ-45-style modular connector. The twisted-pair cables are rated in Category according to the Telecommunications Industry Association/Electronic Industries Alliance (TIA/EIA) 568 standard [16] which describes the specifications the cables met.

The most worldwide used Fast Ethernet media system is 100BASE-T, allowing transmissions with data rates of 100 Mbps.

The 100BASE-T systems might use as signaling components an Ethernet interface with a built-in 100BASE-T transceiver, a Medium-Independent Interface or an external 100BASE-T transceiver. While for building a 100BASE-T twisted-pair segment, the media components used are an unshielded or shield twisted-pair cable and a eight-position RJ-45-style modular connector that meets Category 5 specifications, which applies to 100 ohm unshielded twisted pair cables and associated connecting hardware whose transmission characteristics are specified up to 100 MHz.

2.2.2.2 The Ethernet frame

Ethernet communication consists in the exchange of standardized frames between stations, in which the bits are formed up in specified fields as presented in the figure 2.14 and in accordance with the official Ethernet standard Institute of Electrical and Electronics Engineers (IEEE) 802.3.

The *preamble* field allows the interfaces on the network to synchronize themselves with the incoming data stream, preventing the important data fields will not be lost due to signal start-up delays present in 10 Mbps Ethernet interfaces. Although signal start-up delays are avoided in Fast Ethernet systems, due the use of more complex mechanisms for encoding signals, the preamble is maintained in this systems to provide compatibility with the original Ethernet frame. According to the IEEE 802.3 specification the *Start Frame Delimiter* (SFD) consists on one byte in which the last two bits are 1, and it can be considered as part of the preamble.

56 bits	8 bits	48 bits	48 bits	16 bits	46 to 1500 bytes	32 bits
Preamble	SFD	Destination Address	Source Address	Length or Type	Data/LLC	Frame Check Sequence

Figure 2.14: IEEE 802.3 frame.

The *destination address* consists in a 48 bit Ethernet address called the interface's physical or hardware address where the frame is intended to be delivered. This field may contain a multicast address, a standard broadcast address or a unicast address. When a frame is received by an Ethernet interface, it is read at least to this field, so that they can ignore the rest of the frame if this destination address does not match their own or the ones their programmed to receive.

The *source address* corresponds to the physical address of the interface that sent the frame. This field and the *destination address* are composed by two 24 bits parts, one called Organizationally Unique Identifier (OUI), forming the first half of the physical address of any Ethernet interface, acquired by the Ethernet equipment manufactures from IEEE. The second part is assign by the vendor as each interface is manufactured.

The follow field is either a *length* or *type* field. In the first version of IEEE 802.3 standard, published in 1985, the *type* field was not included, being this field only called *length* field. In 1997, *type* field was added to the mentioned standard. The hexadecimal value present in the field indicates the manner that it is being used. If the decimal value of the field is less than or equal to 1500, then this field is being used as a *length* field, indicating the number of Logical Link Control (LLC) data octets that are present in the *data* field of the frame. If the decimal value of this field is greater than or equal to 1536, then this field is being used as a *type* field, indicating the type of protocol data being carried in the *data* field of the frame.

The *data* field contains a minimum of 46 bytes and a maximum of 1500 bytes. If the number of LLC data octets is inferior than the minimum, then data octets will be added to make the data field large enough. When the frame is received, the length of valid data in this field is determined using the *length* field.

The last field in the IEEE 802.3 frame is called *Frame Check Sequence* (FCS) or *Cyclic Redundancy Check* (CRC) and it is used to check if the data transmitted is correct. This 32 bit field is calculated using all the fields in the transmitted frame except the *preamble* and the SFD. When an interface receives a frame it calculates the CRC and compares it to the present in the frame, if both values match it means that no errors occurred during the transmission of the frame over the Ethernet channel.

2.2.2.3 Medium Access Control protocol

When transmitting a frame on the original mode of a Ethernet operation, a half-duplex shared Ethernet channel, *Medium Access Control Protocol* defines a set of rules, such as Carrier Sense with Multiple Access and Collision Detection (CSMA/CD), that devices present in the network must follow.

Mainly, a interface connected to this type of channel must know when it can transmit and receive and must be able to detect and respond if a collision occurs. To do so, the following steps must be followed:

- if a signal is being transmitted in the channel, the present condition is called ”carrier”;
- if a device wants to transmit a frame, it waits until the channel is idle, waiting for an absence of ”carrier”;
- when the channel is idle, the interface waits a period called *interframe gap* (IFG), and then transmits its frame; and
- if two interfaces transmit at the same time, a collision occurs. The interfaces detect the signal collision and reschedule their frame transmission. This mechanism is called *collision-detection*.

Once data transmission is made one frame at a time and every interface use the same rules to access the Ethernet channel, the use of this algorithms ensures a fair chance to use the network for each interface present.

With the evolution of the Ethernet networks, figure 2.15, concretely with emergence of the full-duplex operation mode, approved for adoption into the IEEE 802.3 standard in March 1997, simultaneous communication between a pair of stations is allowed. For full-duplex operation, the media system must provide independent transmit and receive data paths. This constitutes a major advantage compared with the original operation mode, once it doubles the total bandwidth and the timing requirements of a shared channel no longer limit the maximum segment length, being the only limits set by the signaling capabilities of the media segment.

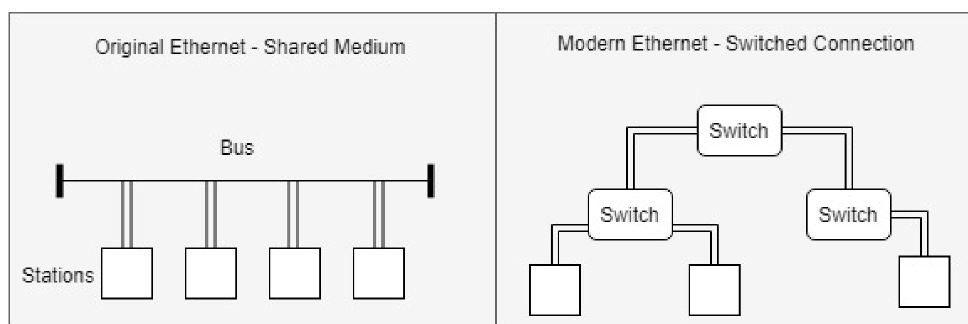


Figure 2.15: The evolution of the Ethernet networks.

In the full-duplex operation mode the connection is made by exactly two stations in a point-to-point link, medium is not shared and consequently CSMA/CD algorithms are not needed. Both stations must be configured to operate in the same operation mode, otherwise communication errors will occur.

In this operation mode, the MAC protocol is used to provide mechanisms that control when Ethernet frames are sent, allowing stations to interact in real time to control the data flow.

2.2.3 CAN

CAN is a communication protocol which was introduced by Robert Bosch GmbH in 1986 as a serial bus system originally developed to by used in passenger automobiles. Mercedes Benz was the first car manufacturer to apply it for networking body electronics of its S-Class vehicles. Although, it is now used in several different vehicles from trucks to planes, as well as in industrial applications.

"Since 1994–1995, CAN is the most common protocol for automotive applications, and is described in the ISO standards 11898-1 to 11898-5. In Part 1 of ISO 11898 (ISO 11898-1), the data link layer and physical layer are set according to the ISO reference model ISO/IEC 7498-1. The "High Speed" CAN bus access (up to 1 Mbps) is specified in ISO 11898-2 and mainly used in the propulsion of a vehicle. The "Low Speed" CAN (40...125 kbps) for the comfort section is described in ISO 11898-3. ISO 11898-4 allows a time-triggered communication to ensure a smooth data transfer with high communication traffic", [17]. ISO 111898-5 is an extension of Part 2 and describes High-speed medium access unit with low-power mode.

In the following subsections, it will be presented CAN main features, namely CAN physical layer, CAN data link layer with the typical CAN frame and higher-layer protocols.

2.2.3.1 Physical Layer

For CAN systems there are two main concepts for the Physical Layer, namely the Fault-Tolerant Low-Speed CAN Physical Layer and the High-Speed CAN Physical Layer. These concepts differ in the maximum data rate, 125 kbits/s for low-speed CAN and 1Mbit/s high-speed CAN, and differ in the bus termination. The focus will be taken in high-speed CAN since it is the one used in the communication between products and test systems.

CAN Physical Layer (fig.2.16) consists in a transceiver, a connector, a network or a data bus, a CAN coil (optional), electromagnetic compatibility (EMC) and electrostatic discharge (ESD) protection devices (optional).

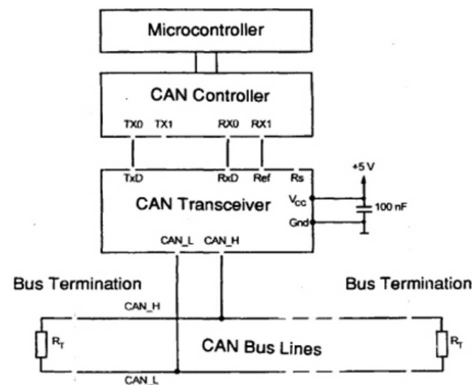


Figure 2.16: Physical CAN Connection according to ISO 11898, [18].

For the two different Physical Layers concepts, the transceiver is identical, it transmits and receives the physical data to and from the bus using differential voltage signals which are converted in the receiver to a logical signal.

The standard, defined by ISO, states the use of two 120 ohms resistors in each termination of the bus for high-speed CAN, which reduces the echo on the bus and makes the communication more reliable.

CAN can have several topologies:

- Single star (fig.2.17) - In this architecture, all transceivers are connected to a single point and only one termination resistor (60 ohm) is used. The maximum length of each wire is 9 meters and the maximum baud rate is 500 kbaud;

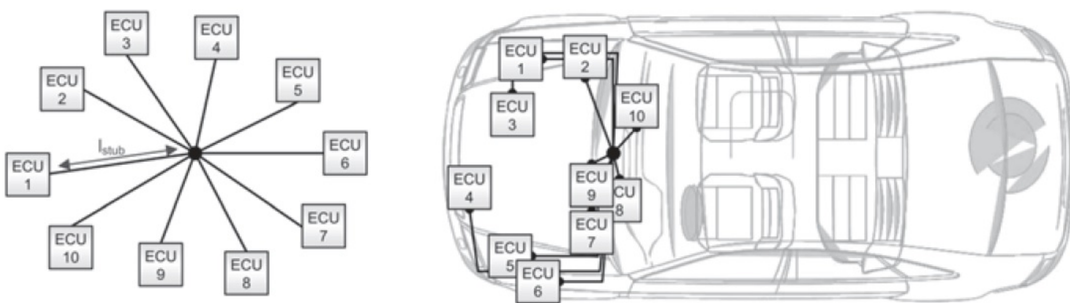


Figure 2.17: Schematic and application of a single star topology, [17].

- Twin Star (fig.2.18) - In this architecture, two single stars are connected to each other, with the termination resistors located in the center of each star. The maximum baud rate is equal to the single star topology;

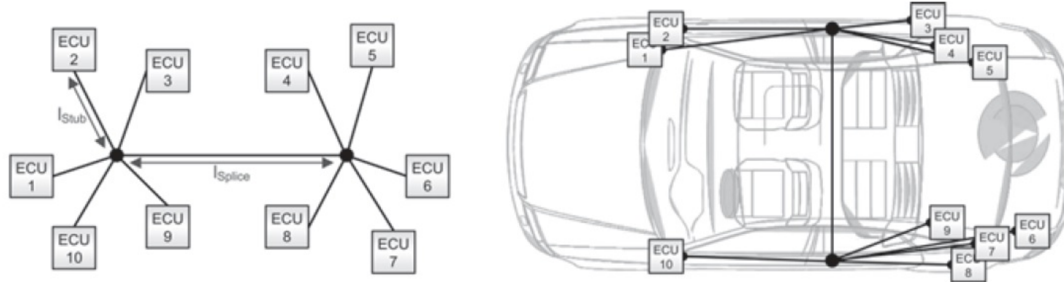


Figure 2.18: Schematic and application of a twin star topology, [17].

- Linear Bus (fig.2.19) - This architecture is the most used in industrial applications, allowing baud rates up to 1 Mbaud. The connection between the transceiver and the bus is limited to a maximum of 30 cm and the bus length should be below 40 m; and

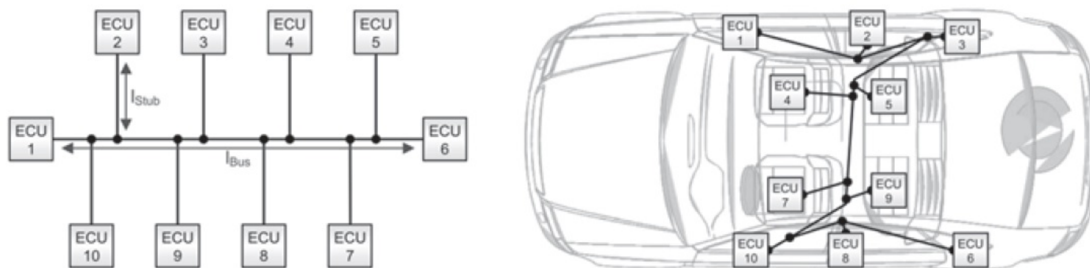


Figure 2.19: Schematic and application of a linear bus topology, [17].

- Hybrid (fig.2.20) - This architecture consists in a combination of a single star and a linear bus. This topology allows baud rates up to 1 Mbaud but the maximum length of the wire is smaller than in the other topologies. The cost for this solution is higher than for the others.

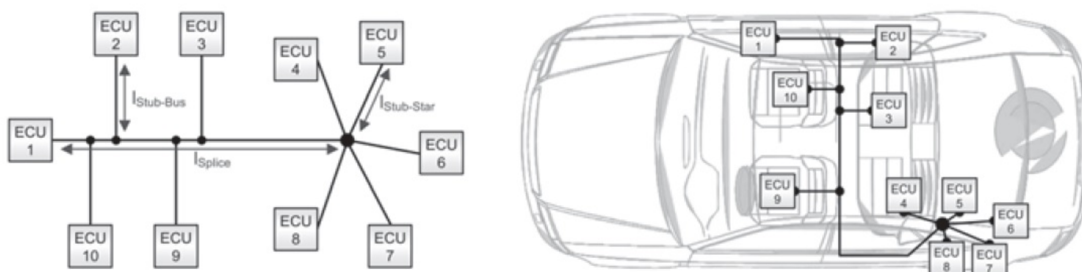


Figure 2.20: Schematic and application of a hybrid topology, [17].

2.2.3.2 Data link layer

The data link layer of CAN protocol is divided into two separate layers, as in the Ethernet's data link layer these two layers are MAC, responsible to manage the use of the bus, in order to avoid collisions, frame encoding and decoding, and signaling errors, and LLC, responsible to give the User a proper interface between lower and higher layers being used in the CAN system, this interfaces are called communication services.

CAN protocol's MAC mechanism is based on the CSMA scheme. This means that before a node transmit, it must know the state of network. If the network is in a idle state, the node can transmit its frame otherwise it must wait for the end of the current transmission. When two nodes transmit at same time a collision occurs, but unlike Ethernet, CAN resolve contentions in a deterministic way, using the identifier field of the frames, that as mentioned before the ones with lowest identifiers have higher priority. This way, CAN ensures that maintenance of the bandwidth available.

ISO specification states that LLC sublayer of CAN provides two communication services, one used to broadcast the value of a specific object over the network and one used to ask for the value of a specific object to be broadcast by its remote producer. This way CAN can be applied to Producer/Consumer and Master/Slave based systems.

Furthermore, CAN controllers can be classified as BasicCAN or FullCAN. For BasicCAN controllers, there are one transmit and one receive buffer, as in UART, and frame filtering is made by application layers. In the FullCAN case, a variable number of internal buffers can be configured to either receive or transmit particular messages and frame filtering is implemented directly in the CAN controller.

2.2.3.3 The CAN frames

Although CAN frames belong to the data link layer, it was considered relevant to have a subsection specifically to describe them.

CAN protocol defines four types of frames: Data frame, Error frame, Remote frame and Overload frame. Each one have different objectives but only Data frame transports message data. The others are used for fault containment, triggering and synchronization.

For Data frame, message is divided into specific fields with fixed length. In CAN there are two different structures for CAN data frame (fig.2.21), distinguished by the size of the identifier field.

A frame with 11-bit identifier field (Standard Frame) has the following composition:

- starts with a single bit called *Start of Frame*, used to signal the start of a frame and for synchronize all network nodes;

- the next field is called *Arbitration Field* and it is composed by 12 bits. This bits contain the 11-bits Identifier, used as logical address and indicate the message's priority. The lower the numerical value is, higher the priority is. *Arbitration Field* last bit is called *Remote-Transmission-Request* (RTR) and it is used to identify the frame as containing data from its

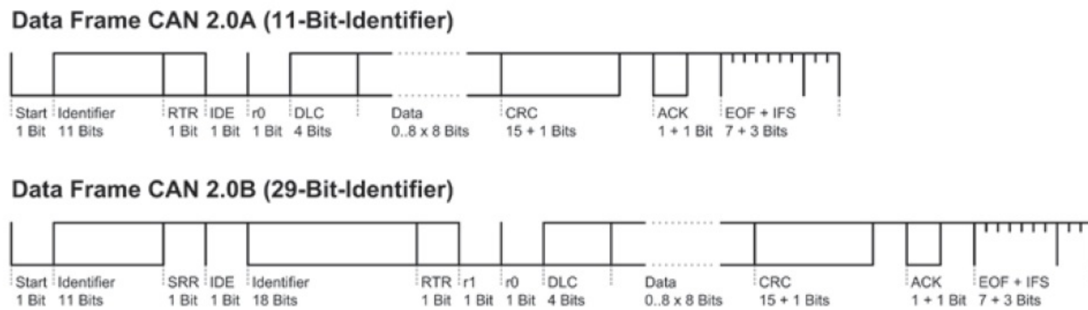


Figure 2.21: Structure of a CAN Data frame, [17].

identifier or a frame with no data but triggers the actual transmitter of the identifier to send a frame with data;

- following RTR is the *Control Field* composed by 6 bits that are also separated in different fields. The first bit of this field is the Identifier Extension Flag used to indicate if the data frame is either a one with 11 bit identifier or a 29 bit identifier, and it comes low or high respectively and indicates if the identifier is completed. The next bit *r0* is reserved. Finally, *Control Field* last four bits contain the Data Length Code (DLC) to indicate the length of the following field;

- *data field* has variable size (0 to 8 bytes) and contains the data of the message;

- used for fault detection, the following field is called *CRC field* and it is 16 bits long. It contains the checksum for the preceding bits in the first 15 bits and is followed by one bit called *CRC* Delimiter in high level;

- the next field of the frame is *Acknowledge Field*. This field has one bit called *ACK Slot*, transmitted in a high level excepting to be overwritten to low level by a receiver node. This only indicates that the frame was at least recognized by at least one active node as a correct one. As in *CRC field*, the first field, *ACK Slot*, is followed by a one bit delimiter in high level;

- After *ACK field*, the frame has the called *End Of Frame* (EOF) 7 bits long field, that as the name says, it indicates the end of the data frame; and

- finally, the frame has the *Inter Frame Space* (IFS) 3 bits long field, used to separate the current frame from the following one.

Extended frames, frames with 29 bit identifiers, differ from the standard frames in the *Arbitration Field* and in the first bit of the *Control Field*. *Arbitration Field* consists in 32 bits divided in three parts, the 11 most significant bits are the *Base Identifier*, followed by two high bits called the *Substitute Remote Request* and the *Identifier Extension Flag*. The 18 least significant bits correspond to the *Identifier Extension*. The last bit of the Arbitration Field is the RTR.

2.2.3.4 Higher-layer protocols

In the past few years, several higher-level application protocols have been defined, in order to reduce the resources involved in designing and implementing automation and networked embedded systems. These protocols rely on the CAN data link layer to transfer messages between nodes of this type of network. ”The main aim of such protocols is to provide a usable and well-defined set of service primitives, which can be used to interact with embedded devices in a standardized way. Reduced design and development costs are easily achieved thanks to standardization”, [19].

Currently, several solutions are available such as CANopen [20] for embedded control systems, Society of Automotive Engineers (SAE) J1939 [21] used in trucks and other vehicles, DeviceNet [22] used in factory automation applications, among others.

2.3 Problem description

Along with the appearance of new products, Aptiv’s Engineers are used to develop new communication tools despite using similar tools on previous products. This results in a waste of resources, both human and time. On one hand allocating someone to create a new communication tool that could already created is dispensable, on the other there is difficulty in sharing the information to use those tools correctly.

Furthermore, during Infotainment Systems’ life cycles there is the need to use different communication interfaces to interact and control them. In early phases, when products are being developed, communication interfaces are used to debug the equipment behavior. In later phases, when products are being produced, there is the need to control the equipment when it is being tested, which demands tools that can be integrated in *NI TestStand*. In laboratory, both for repair and current product engineering, there is also the need of communication tools to control and debug Infotainment Systems.

When a certain system is communicating with a Infotainment System, it is important to monitor that connection. In the case of test systems, located in Production areas, it is desirable to access remotely the communication content being exchanged between these systems and the products in test.

It is in the need of standardized tool to communicate with Infotainment Systems, well documented in order to be used by Aptiv’s Engineers of various areas, that the problem was identified. To solve the problem presented, it shall be created a flexible tool that can be used to establish communications between Infotainment Systems and other systems, for example test systems, using different communication interfaces, namely RS-232, Ethernet and CAN. Besides permitting the exchange of messages between the mentioned systems, the solution must permit to consult the communication content both locally and remotely. The filtering of messages must also be guaranteed.

Chapter 3

Proposed Solution

This chapter 3 is divided in three main sections that have as objective to present the proposed solution, the requirements that it must address and the consequent conclusions.

In the first section are presented the requirements that the solution must comply in order to fulfill the objectives defined for this work.

Derived from the points contained in the mentioned section, a description of the proposed solution is presented in the second section of this chapter.

In the last section are presented the conclusions derived from the architecture presented in 3.2.

3.1 Requirements

In accordance with the needs derived from the problem to be solved, it was elaborated a list of requirements and goals that its solution must comply.

It is desired that the solution has enough flexibility so it can be used by test systems responsible for the final tests in the Production lines of different products. It is also desired that the solution can be used by Test Engineers during the development of new test sequences when a new product is being prepared for production as to be used in laboratory and repair locations where products are debugged.

Thus, the requirements that the solution must address are:

i. contain functions that are used to define the characteristics of the Aptiv's product to be tested;

ii. contain functions that are used to define the characteristics of the test equipment, as signal generators, to be controlled;

iii. contain functions that are used to define the characteristics of the communication interfaces to be used by the systems where the mentioned functions are used;

iv. deal with multiple connections (maximum 4);

v. contain functions that establish RS-232 connections;

vi. contain functions that establish Ethernet connections;

vii. contain functions that establish CAN connections;

viii. contain functions that make enable synchronous and asynchronous connections between the systems where the mentioned functions are used and the devices that are intended to communicate with, namely Aptiv's products and test equipment;

ix. contain functions to send, receive and wait for messages of the different connection types mentioned before;

x. give to the User the ability to send and receive messages over different communication interfaces using the same functions despite the involved interfaces;

xi. contain functions that are used to save all the content of a established connection;

xii. the number of communication logs saved must be limited to 5;

xiii. contain functions that are used to flag a certain message defined by the user in a established connection;

xiv. contain a monitor service that makes User able to remotely access data from certain connection established between a remote device and certain product, for example in a production line;

xv. work on *Windows* Operative System (minimum version is *Windows 7*);

xvi. contain functions that can be integrated in test sequences from *NI TestStand*; and

xvii. contain functions written in C# or C++.

Analyzing the described requirements, it is possible to separate them in different groups that concern different objectives.

The first one, from i to iv, establish the intention to have functions that are used to define the characteristics of the equipment involved in a certain communication to be established. Consequently, this information must be stored somewhere, must be editable and there must be a way to parse it to the functions used for communicating. The number of different devices described is limited to the number of connections that can be established, in this case four.

A second group, from v to xii, states the intention to establish connections using RS-232, Ethernet or CAN between a certain system and a device with the selected interface. With the connection established the User shall be able to send, receive and wait for messages using the same functions despite the interface being used. The solution must save all the communication in order to be consulted during and after the communication. The number of logs saved must be limited to 5, because in a test system a lot of units are tested and if the system keeps saving all the logs, storing can be become a problem.

A third group, xiii and xiv, states the intention to monitor the communications established using the functions described before. The monitoring process shall be done using functions to be used locally, this is in the system where the communication functions are being used. The monitoring process shall be also done remotely, in other words, the communication content must be available to consult from other system that is not connected directly to the device that is communicating with system where communication functions are being used. This is of particular interest, for consulting the content of a communication established between a test system in the Production line and the unit being tested by it, without using the test system itself.

Finally, a fourth group, from xv to xvii, concerns the environment where the solution can be integrated. Since test systems use Windows as operative system, along with *NI TestStand* to run tests on Aptiv's products, it is imperative that the solution can be integrated in both. C# or C++ are the programming languages that can be used to develop scripts to be used on *NI TestStand*.

The definition and analysis of the requirements above were the start point for elaborating a solution that integrates different communication types and makes the User capable of monitor locally and remotely the desired communication.

3.2 Description

To ensure that all requirements, separated in four groups in 3.1, are fulfilled, the proposed solution is the creation of a framework composed by different modules that together can characterize, establish, control and monitor RS-232, Ethernet and CAN communications.

The figure 3.1 shows the proposed solution diagram. This solution includes:

- a standardized Extensible Markup Language (XML) file called Product Communication Configuration File (PCCF) created specifically to be used to store the characteristics of the interfaces that the User intends to communicate with and the paths where all contents of the communications and the errors that can occur should be saved.

  XML was the format chosen because it enables the creation of different tags, that can be organized in a hierarchical way, used to store pertinent information without using too much memory. Furthermore, this type of file is easily editable;

- a Dynamic-Link Library (DLL) constituted by different functions. One of these functions is responsible to extract all the information from the previous mentioned XML file and build an object. This object can be then used by the remaining DLL's functions that make the User able to open communication channels, send, receive and wait for possible messages. These functions save communication contents and errors/exceptions that can occur in log files called Product Communication Logs (PCLs). This DLL also include functions to filter and signal messages defined by the user and check the current state of certain interface.

  This DLL shall be written in C# due the fact that functions used in test sequences can use .Net, which is a framework with a lot of useful resources. Having a DLL with all these functions makes the framework more flexible, since it can be used not only on *NI TestStand*

but also in other applications where it is intended to communicate using RS-232, Ethernet and CAN interfaces; and

- a monitor solution constituted by a Windows service that works as a Transmission Control Protocol/Internet Protocol (TCP/IP) server in the system where the framework is being used and a User Interface (UI) to monitor remotely the communications established using the framework's DLL functions, working as a client that communicate with the Windows service.

  It is important to mention that IP objective is to hide the underlying physical network by creating a virtual network view, using addresses that are represented by a 32 bit unsigned binary value. This protocol is connectionless packet delivery meaning that the dependency on specific computing centers that use hierarchical connection-oriented networks is minimized. Packets sent by IP might be lost, arrive out of order, or even be duplicated. IP assumes higher layer protocols will address these anomalies, [29].

  TCP objective is to provide a reliable logical circuit or connection service between pairs of processes, [29]. Since it assumes that lower-level protocols, as IP, are not reliable, TCP guarantees that reliability. TCP allows applications to stream data transfer, since it groups data bytes to be transferred to the destination into TCP segments, which are passed to the below layer like IP. TCP assigns a sequence number to each byte transmitted and waits for a positive acknowledgment (ACK) from the destination TCP layer. Case ACK is not received after a timeout interval, data is transmitted again. This sequence numbers are also used by the receiving TCP to order the segments and to eliminate duplicate segments. ACK is also used to indicate to the sender the number of bytes that can be received. TCP allows multiplexing through the use of ports and provides full duplex for bidirectional concurrent data streams.

  This way, this service can be integrated in any Windows system and it enables the connection of multiple devices that intend to consult PCLs, either they are using the UI specifically designed to do so or using other software that can establish TCP/IP connections. Since this type of service is always running on background, it shall also be responsible to manage the logs saved in the system.

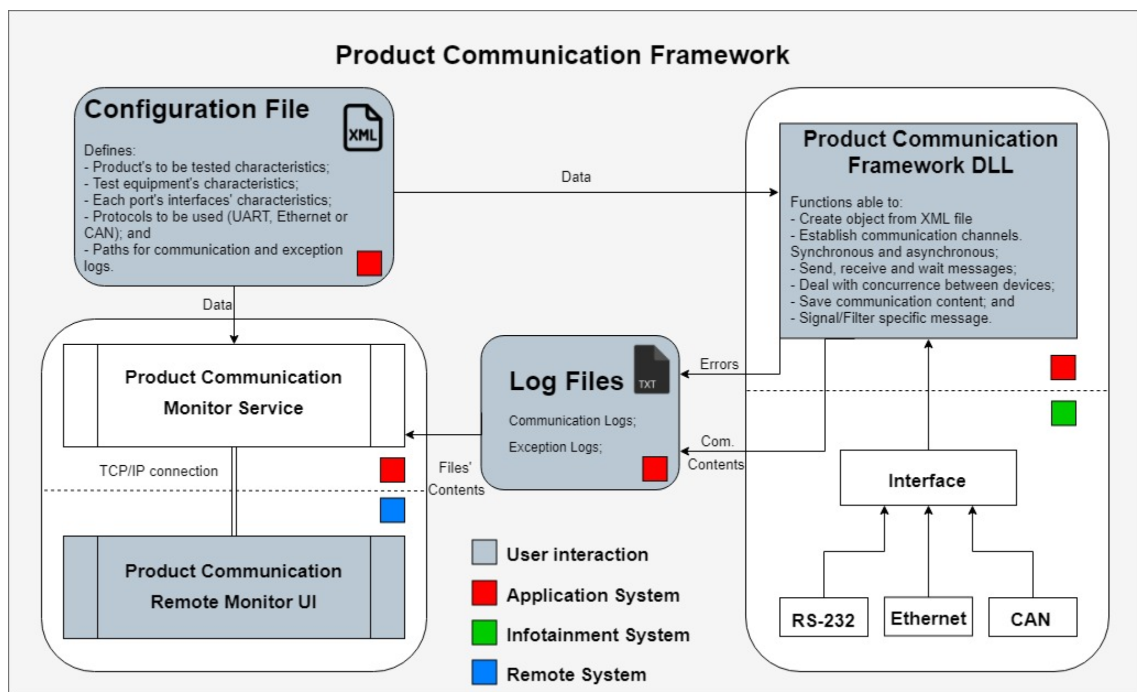


Figure 3.1: Diagram of the proposed solution.

The solution presented can be easily implemented in test systems. Different products and equipment can be described in the configuration file which can be edited at any time. The functions from the DLL, can be integrated in *NI TestStand* test sequences, and independently the interface type, the methods invoked are the same. With the Windows service installed, the communication content is automatically managed and can be consulted from any device that is in the same network, using a UI specifically design to do it or using other software for TCP/IP connections. The communication content can also be consulted locally, once it is always available in the log files.

3.2.1 Main modules

This section focus in the main modules of the solution, namely the Product Communication Framework (PCF) DLL and the Product Communication Monitor (PCM) since together they address the main objectives for this work.

3.2.1.1 Product Communication Framework DLL

Product Communication Framework DLL can be itself divided in three different groups of functions that have particular objectives, such as parsing configuration information, communicate with interfaces and analyze the established communications. Figure 3.2 gives a general view of PCF DLL and shows the hierarchal call trees for its methods.

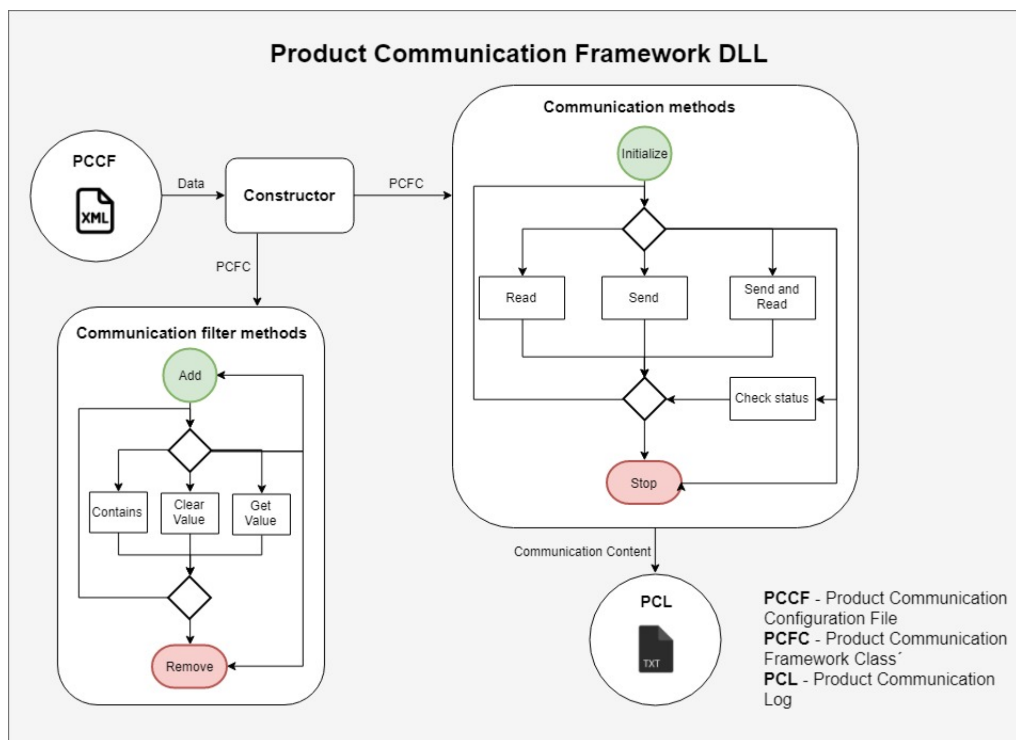


Figure 3.2: PCF DLL general view.

A first part concerns parsing the information contained in the XML configuration file, PCCF, into an object concretely a new instance of **Product Communication Framework Class** (PCFC). This permits that the information from the XML configuration file can be used by the remaining functions of the PCF DLL. In other words, after creating PCFC using the class constructor, the User can call other methods from the PCF DLL using the created PCFC.

A second part is constituted by all the methods that permit the User to communicate with the different devices specified in the XML configuration file and check their status. These methods include the functions responsible to initialize a specified device and establish the connection between the host and the device using the corresponding interface.

After the initialization, the User can use some of the PCF DLL methods to read incoming messages and send messages to the device. The PCF DLL also includes methods to send and wait for an expected response within a time interval.

If intended, all the exchanged messages are saved in a PCL file created in the pre-defined paths from the XML configuration file. To terminate a connection, the User can call a stop method. At any moment, the User can check a certain communication status calling the corresponding method.

The last part of PCF DLL includes functions to create and manage the communication filters, that is, a group of functions which main objective is to flag if a certain message occurred. Concretely, the User have access to methods that permit adding and removing filters. With those filters created, the User can then use a method to check if the filter occurred and a method to check when it happened.

3.2.1.2 Product Communication Monitor

The main objective of Product Communication Monitor is to make the User able to monitor remotely the communications established using PCF DLL's functions. To do so, PCM is constituted by two modules (fig.3.3):

i. Product Communication Monitor Service (PCMS), a Windows service that can be installed in the system where PCF DLL is being used; and

ii. Product Communication Monitor User Interface (PCMUI), a UI application to be used in a remote system.

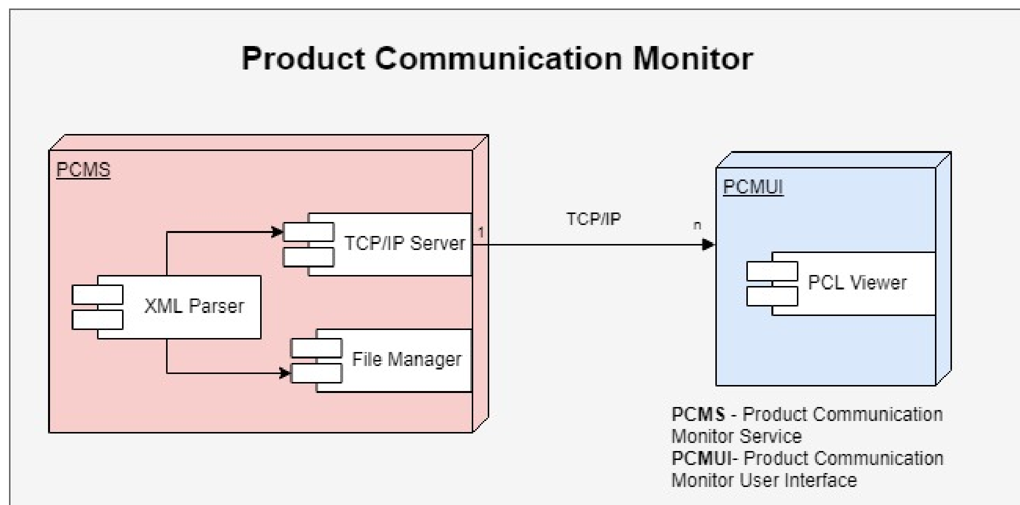


Figure 3.3: Diagram of the Product Communication Monitor.

Despite PCM and PCF not being directly connected, they both use PCLs to share fundamental information.

The main idea is to monitor communications established using PCF DLL remotely is quite simple. PCF DLL's functions write the communication contents in PCLs and PCMS consults those files to be able to transmit the information stored there. Then, using PCMUI the User can establish a connection with a certain system that have PCMS installed in order to have access to the System's information. PCMS shall be responsible to manage the files stored in the system in order to avoid too many PCLs to be stored.

PCMS shall be flexible enough to permit the simultaneous connections of different applications other than PCMUI. It shall respond to specific commands that make the User able to remotely consult the interfaces characteristics connected to the system and consult the PCLs of established connections. Consequently PCMS, must content a XML parser in order to extract information from PCCF in a similar way PCF constructor does.

Despite other applications can be connected to PCMS to obtain information from the system, the main objective for building a UI specifically to monitor the communications established using PCF is to make the User able to download a copy of the PCLs to the remote system.

3.3 Conclusions

After defining the requirements that the solution for the problem must comply it was possible to design the solution's architecture.

Using a standardized XML file for saving configuration parameters along with a DLL, the proposed solution guarantees the flexibility to be integrated in different applications where communication over RS-232, Ethernet or CAN is a must, such in test systems.

The solution presented also guarantees the possibility of monitoring the connection established using the DLL's functions both locally and remotely. Locally, communication content is saved in log files and can be consulted at any time. Remotely, the access to these log files is guaranteed by a *Windows* service along with an UI designed specifically for purpose, or a TCP/IP client.

An example of application for the proposed solution is illustrated in figure 3.4 where the framework's modules are signaled in red. Inside a test system, the communication functions are used inside *NI TestStand* test sequences to communicate with Infotainment Systems and Signal generators. The configuration file and the log files are saved in it too. The *Windows* service is optional depending if the user intends to permit the remote access to the communications established using the framework's functions. To have this type of access, there is the UI inside the remote systems.

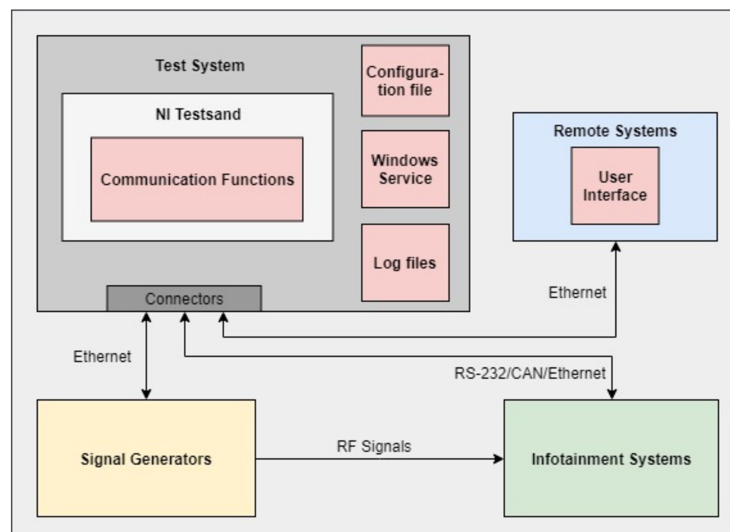


Figure 3.4: Example of application for the proposed solution in test systems.

The solution's architecture is the guide line to its implementation that is presented in the following chapter 4.

Chapter 4

Implementation

This chapter 4 is divided in two main sections that have as objective to describe how the solution's different modules are implemented to address the requirements previously defined in the chapter 3.

In the first section is presented how the XML module, responsible for parsing the description of the interfaces to an object, is implemented and presents how the different interfaces functions are developed in order to be integrated in a unique module responsible to contain all the communication functions

A second section describes how the Product Communication Monitor module is implemented.

4.1 Product Communication Framework DLL

The implementation of PCF DLL was made using *Microsoft Visual Studio 2017* along with its template for building DLLs in C#. This way, the DLL created can be included in different applications and address the requirements xvi and xvii previously defined in 3.1.

PCF DLL (fig. 4.1) is composed by several Classes but only **PCF Class** can be accessed by the User.

PCF Class has public methods that can be grouped in three groups, one concerning parsing information from PCCF and build a **PCF Class** object to be used by the other groups of methods, one concerning communication functions and one concerning functions to filter selected messages.

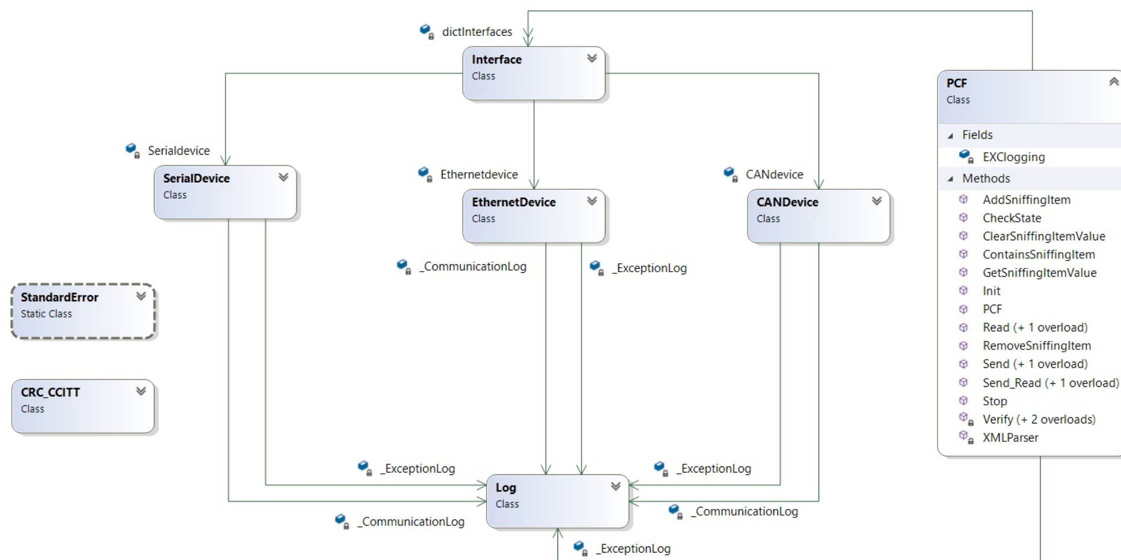


Figure 4.1: PCF DLL Class diagram.

Every object of the **PCF Class** has one field that is a **Dictionary Class** [23] object containing instances of the **Interface Class** (fig. 4.2), built using a private method from **PCF Class** called `XMLParser` that will be presented with detail in the subsection 4.1.1.

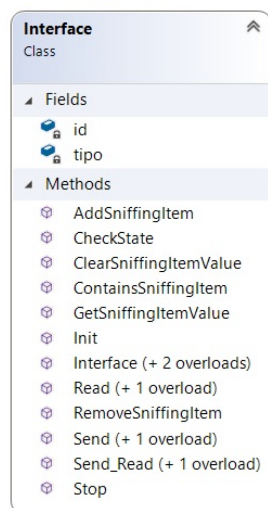


Figure 4.2: **Interface Class**.

Interface Class has three different constructors, each one corresponding to a different interface type. To make this association, the constructors of the **Interface Class** call the constructor of the corresponding device Class to build an object from the desired type. **Interface Class** has a field that identifies the type of communication and it is used by remaining methods to make the link between **PCF Class**' methods and the corresponding device's methods.

Thus, **Interface Class** enables **PCF Class** to inherit fields and methods from **SerialDevice Class**, **EthernetDevice Class** and **CanDevice Class**, which implementation will be presented with detail in the subsections 4.1.2.1; 4.1.2.2 and 4.1.2.3, respectively. This way, the User can use the same functions independently the interface to be used to communicate, addressing the requirement x defined in 3.1.

In addition, **PCF Class** methods return an integer that correspond to an error code from **StandardError Class**. Hence, this functions when included on test sequences of *NI TestStand* can be easily checked for their right functionality.

To build an object from the **PCF Class**, with the parameters of the interfaces defined in the configuration file, the User shall call the `PCF` method present in the table 4.1.

Table 4.1: PCF DLL's Constructor.

Method	Arguments	Returns	Description
`PCF`	**string** *path* **ref int** *error*	**PCF Class** instance	This constructor builds an instance of the **PCF Class** for the file saved in a specified path passed as argument in *path* and it returns an error code in the *error* field passed by reference.

To establish a connection, the User shall use the `Init` method present in the table 4.2.

Table 4.2: PCF DLL's `Init` method.

Method	Arguments	Returns	Description
`Init`	**string** *interfaceName* **string** *interfaceId*	0 if no error occurred -1 if an error occurred -10 if an exception occurred	This function is used to initialize the communication with the selected interface passed as argument in *interfaceName*. The *interfaceId* argument is used to identify the correspondent product in the communication logs.

Once a connection is established with a certain device, the User can read received messages that are saved in a **Queue Class** [24] instance by using the `Read` methods present in the table 4.3. Considering the format of certain messages, only used in RS-232 and Ethernet connections, as mentioned in 2.1.2, there is the need to read messages divided in separated fields. These fields correspond to the ones of the diagnostic commands, which are a prefix field, a group identification field, a test identification field, an operation field, a status information field, a number of bytes field and data bytes field.

Table 4.3: PCF DLL's `Read` methods.

Method	Arguments	Returns	Description
`Read`	**string** *interfaceName* **ref string** *msg*	0 if no error occurred -1 if there is no message to be read -2 if there is connection problem -10 if an exception occurred	This function is used to read a single message passed as reference in *msg* from the selected interface passed as argument in *interfaceName*.
`Read`	**string** *interfaceName* **string** *responseManu-facturingPrefix* **string** *groupId* **string** *testId* **string** *operation* **ref string** *statusInformation* **ref int** *nDataBytes* **ref string** *data* **double** *timeOut*	0 if no error occurred -1 if there is no message to be read -2 if there is a connection problem -3 if the selected interface is invalid -4 if no matching message was found -5 if arguments passed have unexpected size -6 if there was check sum error -10 if an exception occurred	This function is used to read a single message within a timeout in milliseconds passed as argument in *timeOut* from the selected interface passed as argument in *interfaceName*. The prefix field shall be passed as argument in *responseManu-facturingPrefix*, the group identification field shall be passed as argument in *groupId*, the test identification field shall be passed as argument in *testId* and the operation field shall be passed as argument in *operation*. The status information field from the received message is passed by reference in *statusInfor-mation*, the number of bytes read is passed by reference in *nDataBytes* and the received data field is passed by reference in *data*.

The DLL has functions that enables the User to send messages over a certain connection, namely the `Send` methods, presented in the table 4.4. As in the `Read` methods, it was taken in consideration the format of the diagnostic commands. In this case, it is possible to send a message divided in separated fields.

Table 4.4: PCF DLL's `Send` methods.

Method	Arguments	Returns	Description
`Send`	**string** *interfaceName* **string** *msg*	0 if no error occurred -10 if an exception occurred	This function is used to send a single message passed as argument in *msg* to the selected interface passed as argument in *interfaceName*.
`Send`	**string** *interfaceName* **string** *requestManu-facturingPrefix* **string** *groupId* **string** *testId* **string** *operation* **string** *data*	0 if no error occurred -3 if the selected interface is invalid -5 if arguments passed have unexpected size -10 if an exception occurred	This function is used to send a single message to the selected interface passed as argument in *interfaceName*. The prefix field shall be passed as argument in *requestManufacturingPrefix*, the group identification field shall be passed as argument in *groupId*, the test identification field shall be passed as argument in *testId*, the operation field shall be passed as argument in *operation* and the data field shall be passed as argument in *data*.

Derived the need to send a message and capture immediately the respective response, it was developed the `Send_Read` methods that enable to do so, both for raw messages or messages separated in the fields for diagnostic commands, as mentioned in 2.1.2. These methods are presented

in the table 4.5.

Table 4.5: PCF DLL's `Send_Read` methods.

Method	Arguments	Returns	Description
`Send_Read`	**string** *interfaceName* **string** *msg* **string** *responsePattern* **ref string** *response* **double** *timeOut*	0 if no error occurred -1 if the pattern was not found -2 if there was an error matching the pattern -10 if an exception occurred	This function is used to send a message passed as argument in *msg* to the selected interface passed as argument in *interfaceName* and wait for an expected pattern passed as argument in *responsePattern* within a timeout in milliseconds passed as argument in *timeOut*. If the pattern occurs, the correspondent response is returned in *response* passed by reference.
`Send_Read`	**string** *interfaceName* **string** *requestManu-facturingPrefix* **string** *groupId* **string** *testId* **string** *operation* **string** *txData* **string** *responseManu-facturingPrefix* **ref string** *statusInfor-mation* **ref int** *nDataBytes* **ref string** *rxData* **double** *timeOut*	0 if no error occurred -1 if there is no message to be received or if the selected interface is off -2 if there is a connection problem -3 if the selected interface is invalid -4 if the expected message was not found -5 if the arguments have unexpected size -6 if there was a check sum error -10 if an exception occurred	This function is used to send a single message to the selected interface passed as argument in *interfaceName* and wait for the correspondent response within a timeout in milliseconds passed as argument in *timeOut*. The prefix field of the message to be sent shall be passed as argument in *requestManufacturingPrefix*, the group identification field shall be passed as argument in *groupId*, the test identification field shall be passed as argument in *testId*, the operation field shall be passed as argument in *operation* and the data field of the message to be sent shall be passed as argument in *data*. If the expected response occurs, the correspondent status information is returned in *statusInformation*, the number of data bytes received is returned in *nDataBytes* and the data received is returned in*rxData* all passed by reference.

At any time, the User can verify the state of a connection using the `CheckState` method presented in the table 4.6.

Table 4.6: PCF DLL's `CheckState` method.

Method	Arguments	Returns	Description
`CheckState`	**string** *interfaceName*	-1 if the state is unknown 0 if the state is Run 1 if the state is Stop 2 if the state is Stopped 3 if the state is Running	This function is used to check the state of a certain communication established with the selected interface passed as argument in *interfaceName*.

The User can terminate a certain connection using the `Stop` method presented in the table 4.7. After using this method, the User can reinitialize the same connection by calling the `Init` method, previously presented in the table 4.2.

Table 4.7: PCF DLL's `Stop` method.

Method	Arguments	Returns	Description
`Stop`	**string** *interfaceName*	0 if no error occurred -10 if an exception occurred.	This function is used to stop the communication with the selected interface passed as argument in *interfaceName*.

The figure 4.3 illustrates the usage of some of the PCF DLL's methods presented until this point. In this figure, it is possible to verify that, once a connection is established, all received messages are saved and using the `Read` method, the User will have access to them, in a logic of first in, first out.

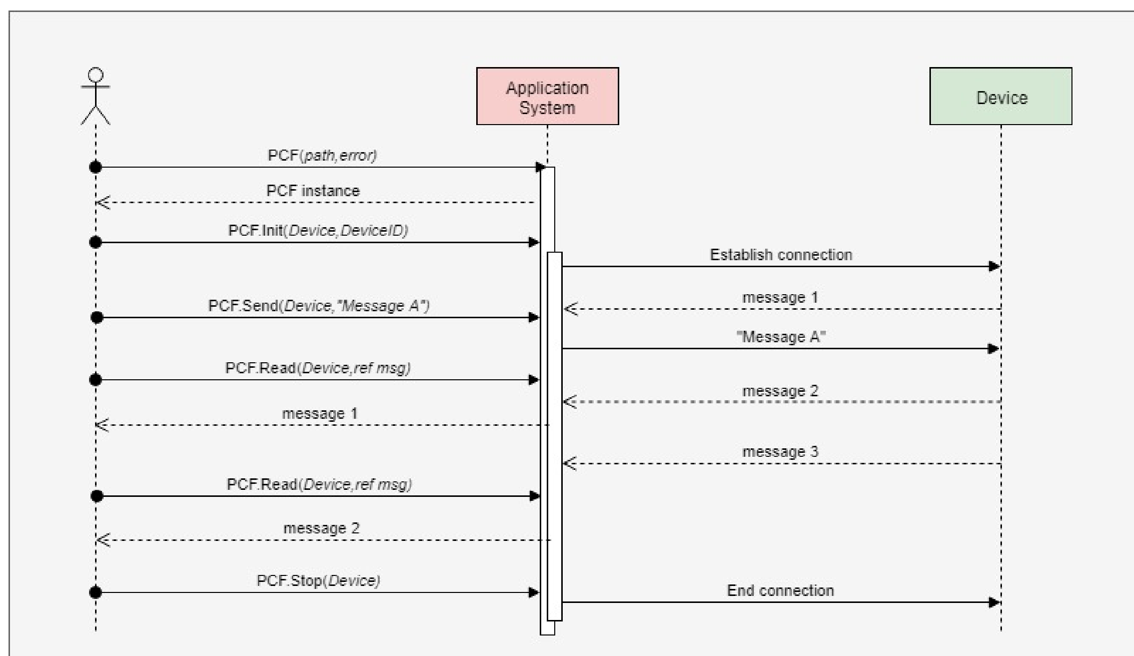


Figure 4.3: Example of interaction between an application and a device using PCF DLL's methods.

PCF DLL has also other public methods which objective is to filter and signal messages defined by the User, in order to address the requirement xiii previously defined in 3.1. The mentioned methods are presented in the table 4.8.

Table 4.8: PCF DLL's Filter methods.

Method	Arguments	Returns	Description
`AddSniffingItem`	**string** *interfaceName* **string** *itemName* **string** *pattern*	—	This function is used to add a pattern to be filtered passed as argument in *pattern* in the communication established using the selected interface passed as argument in *interfaceName*. The user shall define a name for the filter, passing it as an argument in *itemName*.
`RemoveSniffingItem`	**string** *interfaceName* **string** *itemName*	—	This function is used to remove a filter with the name passed as argument in *itemName* from the selected interface passed as argument in *interfaceName*.
`ContainsSniffingItem`	**string** *interfaceName* **string** *itemName*	True if the filter exists false otherwise	This function is used to check if a certain filter with specified name passed as argument in *itemName* exists for the interface with the name passedas argument in *interfaceName*.
`GetSniffingItemValue`	**string** *interfaceName* **string** *itemName*	The captured a message, otherwise the function will return *null*.	This function is used to check if a certain filter with specified name passed as argument in *itemName* exists for the interface with the name passed as argument in *interfaceName*.
`ClearSniffingItemValue`	**string** *interfaceName* **string** *itemName*	—	This function is used to clear the capture value of a filter with specified name passed as argument in *itemName* for the interface with the name passed as argument in *interfaceName*.

Furthermore, once a connection is established, every received and sent messages are saved in PCLs automatically, just by defining a correct path for saving it in the PCCF. Additionally, every expected errors that can occur using a specified interface will be also saved in a log if a path for it is defined in PCCF. To do so, **PCF Class**, **SerialDevice Class**, **EthernetDevice Class** and **CANDevice Class** use **Log Class**' (fig. 4.4) objects and correspondent methods. More precisely, each device has one **Log Class** object to save the communication content and one **Log Class** object to save errors.

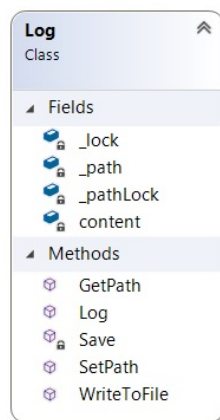


Figure 4.4: **Log Class**.

Objects created using **PCF Class**' constructor have one **Log Class** object to save all the errors, independently of the type of the interface in use by the correspondent **PCF Class** object.

To make the interaction between PCF DLL's Classes more clear, the figure 4.5 shows a simple example where using an abstract Device, that could be RS-232, Ethernet or CAN, some of the methods from PCF DLL are used. First, information about the device is extracted from the configuration file using **PCF Class**' constructor (presented at table 4.1), in order to construct an Interface instance that for its turn will use the constructor from the "DeviceClass". Then, the communication with the device is initialized using the `Init` method from the DLL (presented at table 4.2). In this example, if the initialization succeed, the method `Send_Read` (presented at table 4.5) is invoked to send a message to the abstract Device and wait for an expected response. Finally, the `Stop` method (presented before at table 4.7) is used to terminate the connection with the Device.

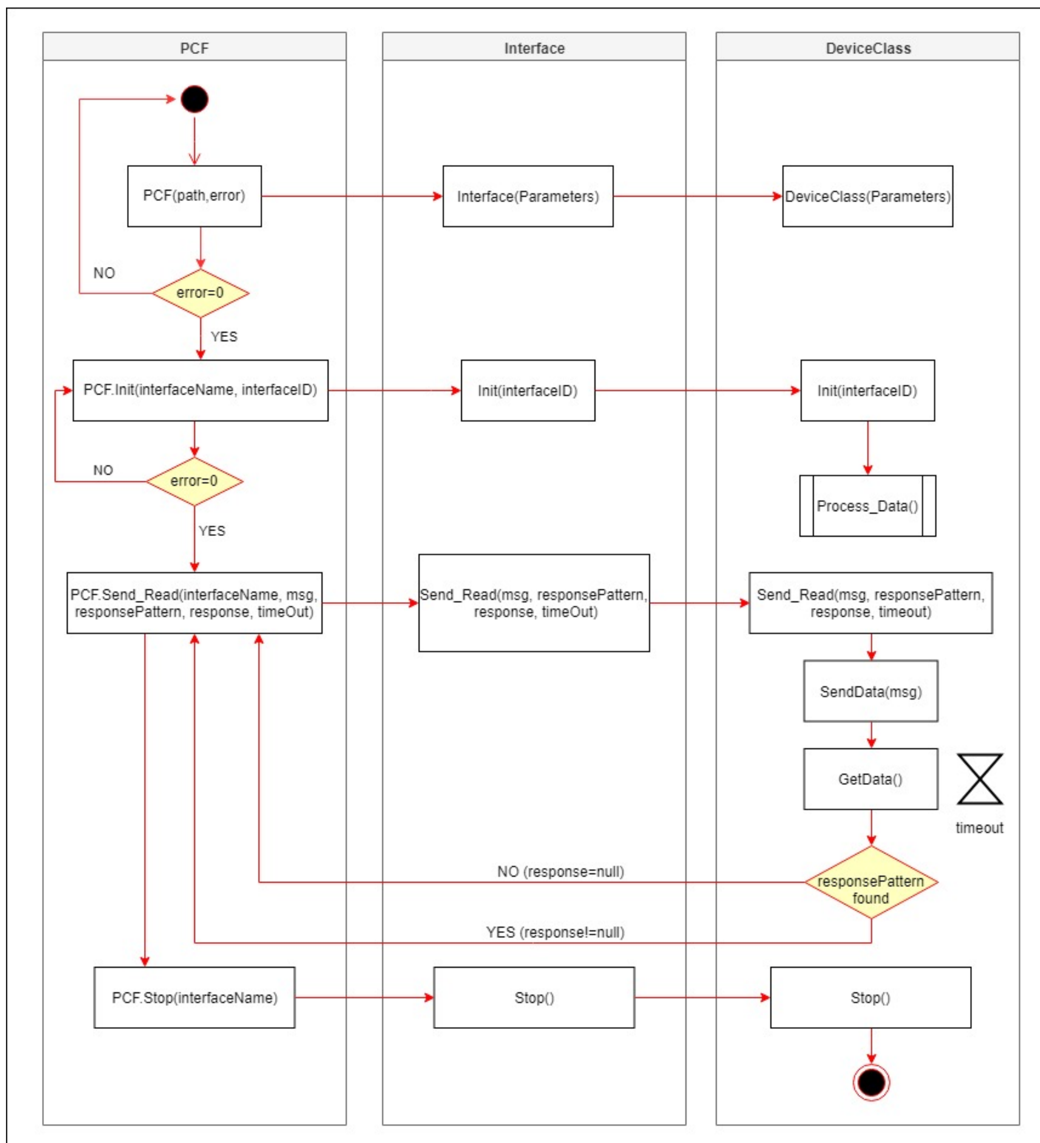


Figure 4.5: Example of interaction between PCF DLL's Classes.

With the top view of PCF DLL presented, it is important to describe how lower level modules are implemented, namely the XML parser and the different interfaces' functions. This description is present in the following subsections, 4.1.1 and 4.1.2.

4.1.1 XML configuration file and XML Parser

The requirements specified in 3.1, more precisely i, ii and iii, state the need to specify the characteristics of the interfaces that the User intend to use and transform that information in such way that makes it to be usable by the functions present in the PCF DLL.

It is within this scope that first was defined the format of PCCF, choosing for each interface type which characteristics must be defined in order to it's values be used by the correspondent functions. Then, it was defined a method able to parse this information to an **PCF Class** object when it is created. Please consult appendix A to see a PCCF example with a RS-232 device, an Ethernet Device and a CAN device.

Independently of the device which is intended to communicate with, it must be defined an unique identification (ID) for each device in PCCF. This ID will be then used by the User to call PCF DLL methods, corresponding to the *interfaceName* field in its methods.

Each device has also a group of parameters defined in PCCF corresponding to its characteristics that vary with the communication type, also specified within these parameters. As mentioned before, each device has as parameter a path for saving the communication content and another path to save eventual errors that can occur.

The remaining parameters vary with communication type of the correspondent device. For devices that use an RS-232 interface, the parameters to be defined in PCCF are the correspondent port name, baud rate, parity, number of data bits, number of stop bits, timeout for writing and a timeout for reading. For devices that use an Ethernet interface, the parameters to be defined in PCCF are the correspondent IP address, port number, address family code and buffer size. Finally, for devices that use a CAN interface, the parameters to be defined in PCCF are the number of the hardware type, channel number and baud rate. All this parameters are explained with detail in the correspondent subsections of the different interfaces in 4.1.2.

As example, for a PCCF with only one Ethernet device, the content of the configuration file would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<Devices>
<ExceptionLogPath>C:\Users\User\Desktop\PCF\PCF_exceptionLOG.txt
   </ExceptionLogPath>
<Device>
    <id>EthernetDeviceExample</id>
        <Parameters>
                <CommunicationType>Ethernet</CommunicationType>
                <IP>10.238.227.20</IP>
```

```
                <Port>9001</Port>
                <AddressFamily>2</AddressFamily>
                <BufferSize>1024</BufferSize>
                <CommunicationLogPath>C:\Users\User\Desktop\LOGS
                   \Com\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\User\Desktop\LOGS\
                   Error\</ExceptionLogPath>
        </Parameters>
</Device>
</Devices>
```

As previously mentioned, there is the need to transform the parameters specified in the PCCF into variables that can be used inside the methods available in PCF DLL. To accomplish this objective, PCF DLL has a private method called `XMLParser` that has as arguments a string that must indicate the path where the PCCF is saved and an integer passed by reference that correspond to an error that can be returned in it, for example if a parameter is missing. After analyzing PCCF, this method returns a **Dictionary Class** instance that has as primary keys the different IDs of the specified devices and as correspondent value an object from **Interface Class**.

`XMLParser` method uses System.XML Namespace [25] from .NET framework to open and read data present in PCCF, more precisely **XmlDocument Class**' constructor and the `Load` method to open the configuration file then, using **XmlNode Class**, the method is able to parse all the information from the file to **List Class** objects, one for each device defined in PCCF. After that, each **List** created is passed to a **Dictionary Class** object that as primary keys the ID of the devices and as corresponding value a **List** of parameters. Finally, `XMLParser` goes through each element of the **Dictionary Class** object created and using the **Interface Class**' constructors, it adds to the **Dictionary Class** object to be returned by the `XMLParser` method the ID of the device and the correspondent **Interface Class** object.

In example presented, `XMLParser` creates a **List Class** instance with a single object corresponding to the Ethernet device specified in the file.

Then, the **List Class** instance created is passed to a **Dictionary Class** object with a single entrance. In this case, the primary key would be ”EthernetDeviceExample” and the correspondent value a **List Class** instance with the parameters: ”Ethernet”, ”10.238.227.20”, ”9001”, ”2”, ”1024”, ”C:\Users\User\Desktop\LOGS\Com\”, ”C:\Users\User\Desktop\LOGS\Error\”.

Going through the **Dictionary Class** object created, the first value saved in the **List Class** object is the communication type that is used to distinguish which **Interface Class** constructor has to be invoked to create the right object. In this case, `XMLParser` is dealing with an Ethernet Interface, consequently the **Interface Class** object created is of this type which use the remaining values of the **List Class** instance as parameters when invoking the **EthernetDevice Class** constructor.

At last, `XMLParser` returns a **Dictionary Class** instance with a single entrance, the primary

key is once again ”EthernetDeviceExample” and the correspondent value the **Interface Class** object which type is Ethernet and parameters the ones saved in the PCCF. To communicate with this device, the argument *interfaceName* of the PCF DLL functions would be ”EthernetDeviceExample”.

4.1.2 Communication functions

The following sections have as objective to describe how functions used to communicate with the interfaces that the framework must support were implemented.

The implementation of PCF DLL was done adding to it a Class for each interface incrementally. Creating a Class for each interface permitted to maintain the overall structure of the DLL and permits that new interfaces that were not included in the requirements can be added to it with minimal changes in the source code, avoiding also that certain interface’s functions are dependent of other interface’s functions, which shows how modular this DLL is.

The development of the mentioned Classes was done in the order that is presented in the following sections, that is PCF DLL first version supported only RS-232 interfaces, a second version supported RS-232 and Ethernet interfaces and the last version complies with all interfaces defined in the requirements from 3.1, which are RS-232, Ethernet and CAN interfaces.

4.1.2.1 RS-232 functions

The main objective to implement the following functions is to address the requirements v and ix specified before in 3.1, that is establish RS-232 connections and send, receive and wait for messages using this connection type.

SerialDevice Class (fig. 4.6) methods can be divided in three groups:

i. the first one concerns the constructor for this Class. As mentioned before, it is used by `XMLParser` method along with the respective **Interface Class** constructor when a RS-232 device is present in PCCF and consequently a new instance of this class must be created in order to communicate with it;

ii. a second group includes all the methods responsible for guarantee the establishment of the connection, enabling the possibility of send, receive and wait for messages, save those messages and check the state of the connection; and

iii. a third group concerns the methods used to filter messages in the connection.

SerialDevice Class was implemented using **SerialPort Class** [26] from .NET framework that is part of System.IO.Ports namespace [27] that contains classes for controlling serial ports.

SerialDevice Class constructor creates an instance from **SerialPort Class** setting its proprieties. The constructor also defines the remaining fields for the **SerialDevice Class** object that it is creating. These fields are the name for the instance and the paths for creating the files where the communication content and errors shall be saved. It also creates a **Queue Class** instance to save received messages and a **Dictionary Class** instance to handle the filters that can be created.

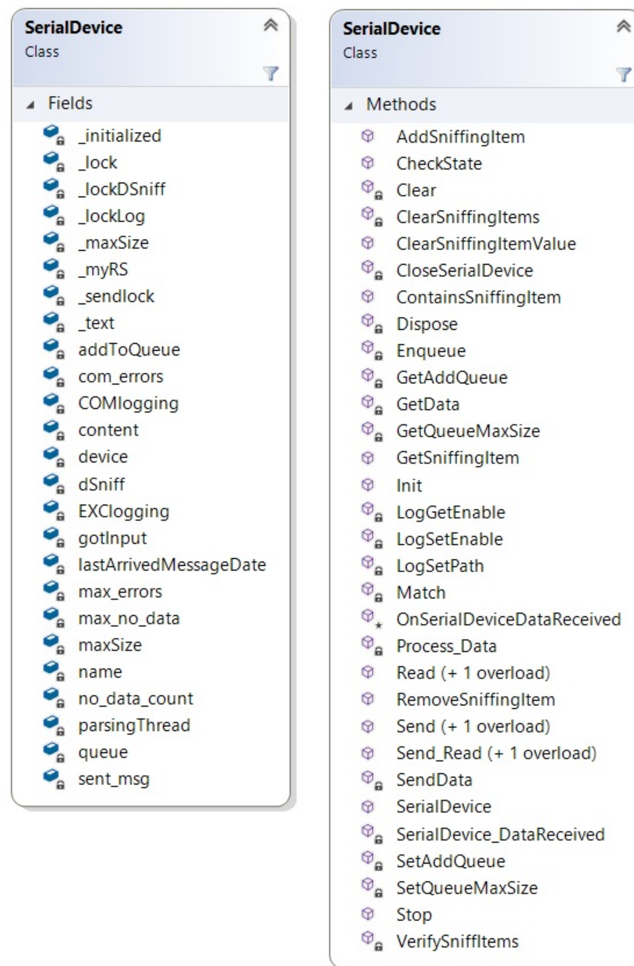


Figure 4.6: **SerialDevice Class**.

To establish the connection the method used is `Init` which verifies the state of the Port that is intended to be used. If it is not being used and its a valid one, this method uses **SerialPort Class**' `Open` method to open it. Finally, it creates a thread (`Process_Data`) responsible to handle received messages and put them in the **Queue Class** instance and in the PCL created for saving them. Every time a message is received, it is verified if it contains any pattern from the filters that can be included in the filter's **Dictionary Class** object. This initialization is illustrated in the figure 4.7.

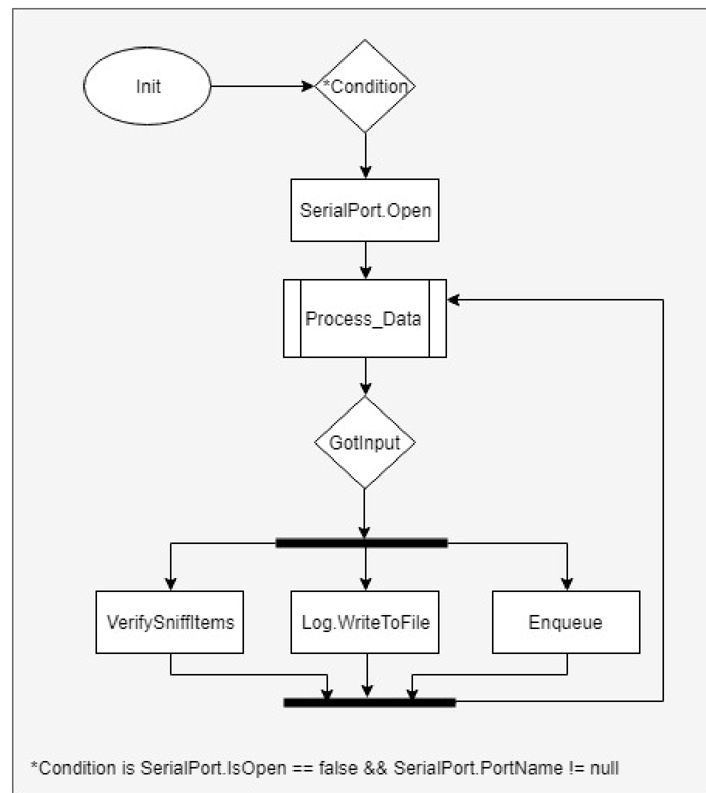


Figure 4.7: Initializing a RS-232 Device.

SerialDevice Class contains two `Read` methods, one to read a message as it was received and other to read a message with a specific format as explained in the respective method from PCF Class. Both methods use **SerialDevice Class**' `GetData` method to get messages from the **Queue Class** instance where received messages are being saved.

The two existing `Send` methods in **SerialDevice Class**, one to send a raw message and other to send a message that is composed by several fields as explain in the respective method from **PCF Class**, use the same **SerialDevice Class** method to effectively send a message using the established connection, this is `SendData` method. This method uses `WriteLine` method from **SerialPort Class** to do so and if successful it saves the sent message in the PCL.

Additionally, in **SerialDevice Class**, the method to send and read a raw message uses `SendData` and `GetData` methods directly while the equivalent method for the predefined format of messages mentioned uses the respective `Send` and `Read` methods for such format.

Once the format mentioned can have a field to contain check sum bytes, it was created a class, **CRC_CCITT Class**, used to generate those field in the `Send` method adding it to the message to be sent and used to verify that field on a received message, returning an error if the calculated check sum is not equal to the received one.

To terminate the established connection, **SerialDevice Class** contains the `Stop` method which dispose all the resources being used by the respective instance and closes the port that is being used by the **SerialPort Class** instance with its `Close` method.

The communication state can be checked with the `CheckState` method which returns the _myRs field that indicates if the connection is either in Run, Stop, Stopped or Running states.

In order to enable the use of the methods that read and send messages included in different threads running at same time, `GetData` and `SendData` methods use the lock statement along with a respective object, _lock and _sendlock. The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released, [28]. This statement is also used in the functions used to save messages in PCL and in the functions that filter messages.

The figure 4.8 illustrates the usage of some of the PCF DLL's methods to communicate with a device using a RS-232 interface. It is possible to observe the interaction between Classes and how messages are sent and received. In the **SerialDevice** line is included the **SerialPort Class** functions calls.

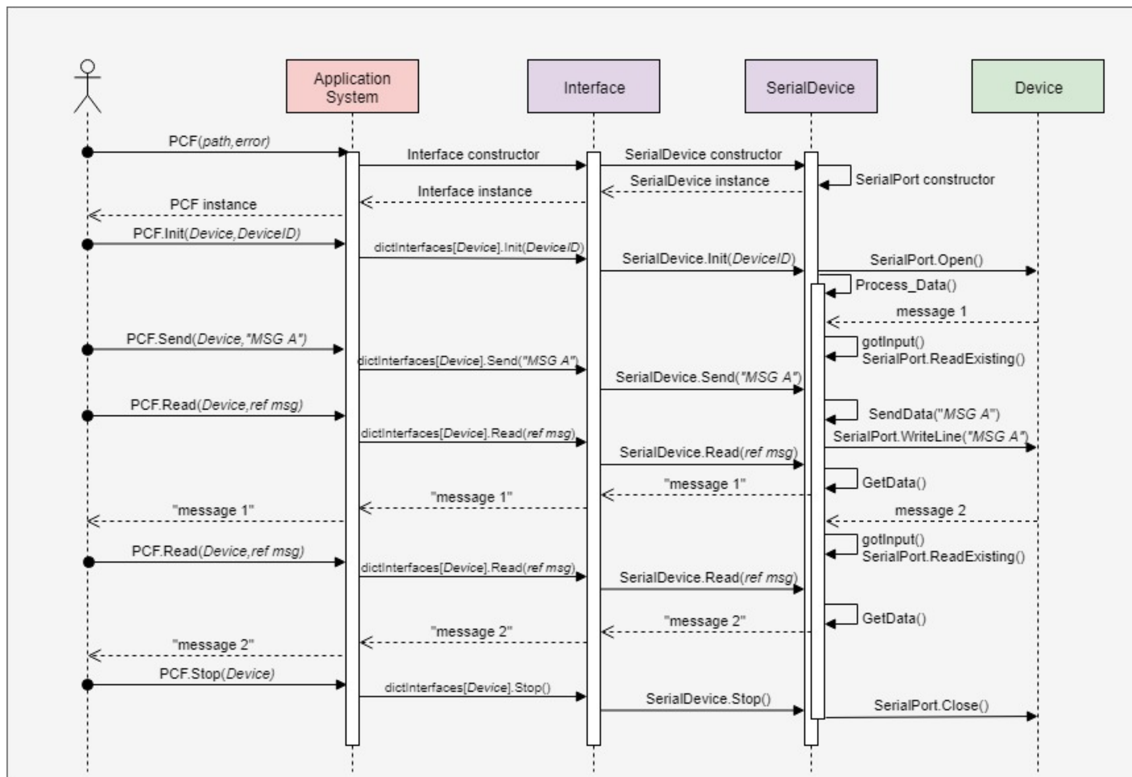


Figure 4.8: Example of interaction between an application and device using PCF DLL's methods to communicate over RS-232.

Finally, **SerialDevice Class** includes the methods that interact directly with the filter's **Dictionary Class** instance. This methods permit to add new filters, verify messages comparing them to existing filters, clear saved filtered messages and remove filters from the filter's **Dictionary Class** instance.

4.1.2.2 Ethernet functions

In this work, Ethernet connections use the association between TCP/IP protocols.

To implement a Ethernet connection over TCP/IP, the typical approach is to use sockets in a server/client relationship (fig.4.9).

The socket interface is an application programming interface to the communication protocols that is used by processes to request network services from the operating systems.

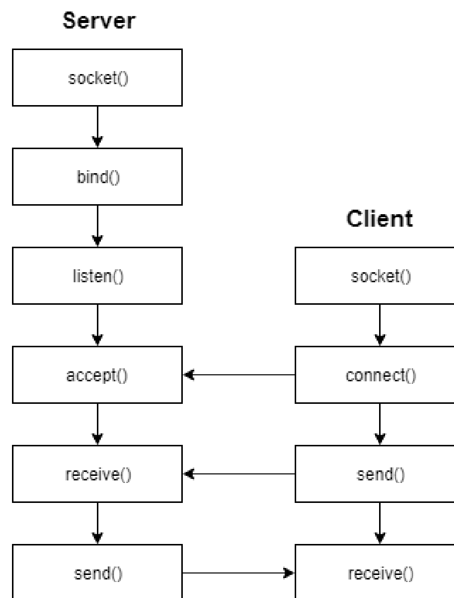


Figure 4.9: Server/Client relationship example for connection-oriented sockets.

Socket Class [30] from .NET is one of the available frameworks that implement the Berkeley sockets interface using different methods and properties that allow to perform synchronous and asynchronous data transfer using TCP/IP. This Class belongs to System.Net.Sockets Namespace [31] which provides a managed implementation of the Windows Sockets interface.

For this type of connection-oriented protocol, **Socket Class** allows the creation of a server that can listen for connections using the `Listen` method. To specify the local IP address and port number, the `Bind` method shall be called before the `Listen` method. For processing any incoming connection request, the `Accept` method can be used and it returns a Socket to be used to communicate data with the remote host. This returned Socket is then used to call the `Send` or `Receive` methods. To connect to a listening host, the method to be used is `Connect`.

In the client side the Constructor shall be used to initialize a new instance of the **Socket Class**.

If it is pretended to communicate data asynchronously, the methods `BeginConnect` and `EndConnect` are used to connect with a listening host while `BeginSend`, `EndSend`, `Begin-Receive` and `EndReceive` methods are used to communicate data. In this case, the incoming connecting request can be processed using `BeginAccept` and `EndAccept` methods.

Finally, when data transfer is finished, the `Shutdown` method is used to disable the Socket and after being called, the `Close` method is used to release all resources associated with it.

The main objective to implement a Class specifically to Ethernet connections is to address the requirements vi and ix specified in 3.1, that is establish Ethernet connections and send, receive and wait for messages using this connection type.

EthernetDevice Class (fig. 4.10) methods can be divided in three groups:

i. the first one concerns the constructor for this Class. As mentioned before, it is used by `XMLParser` method along with the respective **Interface Class** constructor when a Ethernet device is present in PCCF and consequently a new instance of this class must be created in order to communicate with it;

ii. a second group includes all the methods responsible for guarantee the establishment of the connection, enabling the possibility of send, receive and wait for messages, save those messages and check the state of the connection; and

iii. a third group concerns the methods used to filter messages in the connection.

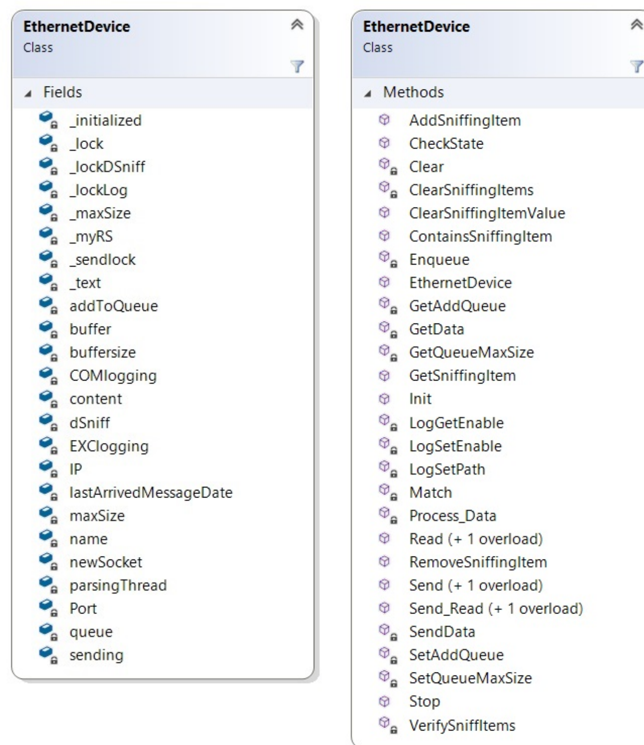


Figure 4.10: **EthernetDevice Class**.

EthernetDevice Class constructor creates an instance from **Socket Class** setting its proprieties. The constructor also defines the remaining fields for the **EthernetDevice Class** object that it is creating. More precisely, it defines the name for the instance and the paths for creating the files where the communication content and errors shall be saved. It also creates a **Queue Class**

instance for save received messages and a **Dictionary Class** instance to handle the filters that can be created.

To establish the connection the method used is `Init` which connects to a remote end point defined in PCCF, using **Socket Class**' `Connect` method. Then, it creates a thread (`Process_Data`) responsible to handle received messages and put them in the **Queue Class** instance and in the PCL created for saving them. This thread uses the `Receive` method from **Socket Class**. Every time a message is received, it is verified if it contains any pattern from the filters that can be included in the filter's **Dictionary Class** instance. This initialization is illustrated in the figure 4.11.

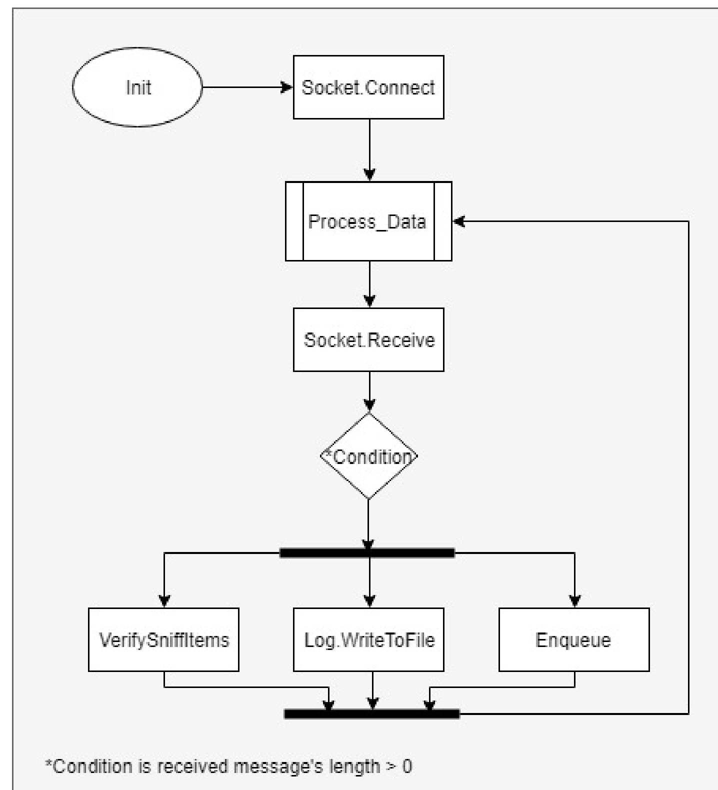


Figure 4.11: Initializing an Ethernet Device.

EthernetDevice Class contains two `Read` methods, one to read a message as it was received and other to read a message with a specific format as explained in the respective method from **PCF Class**. Both methods use **EthernetDevice Class**' `GetData` method to get messages from the **Queue Class** instance where received messages are being saved.

There are two `Send` methods in **EthernetDevice Class**, one to send a raw message and other to send a message that is composed by several fields as explain in the respective method from **PCF Class**. Both methods, use the same **EthernetDevice Class** method to effectively send a message using the established connection, this is `SendData` method. In its turn, this method uses `Send` method from **Socket Class** to do so and if successful it saves the sent message in the PCL.

Additionally, in **EthernetDevice Class**, the method to send and read a raw message uses `SendData` and `GetData` methods directly while the equivalent method for the predefined format of messages mentioned uses the `Send` and `Read` methods for such format.

As mentioned before, as in **SerialDevice Class**, once the format mentioned can have a field to contain check sum bytes, it was created a class, **CRC_CCITT Class**, used to generate those field in the `Send` method adding it to the message to be sent and used to verify that field on a received message, returning an error if the calculated check sum is not equal to the received one.

Despite the format mentioned being specific for RS-232 connections, there is the need to implement functions capable of dealing with it in Ethernet connections since some systems use Ethernet to Serial converters to communicate with certain devices. From the point of view of the application where PCF is being used, the connection is Ethernet only. In this work, it was used *Brainboxes* ES-257, a device that enables the connection of two RS232 interfaces to an Ethernet network.

To terminate the established connection, **EthernetDevice Class** contains the `Stop` method which dispose all the resources being used by the respective instance and closes the connection with the end point that is being used by the **Socket Class** instance, using the `Shutdown` method to disable the socket and the `Close` method to release all resources associated with the Socket object.

The communication state can be check with the `CheckState` method which returns the _myRs field that indicates if the connection is either in Run, Stop, Stopped or Running states.

Like in **SerialDevice Class**, in order to enable the use of the methods that read and send messages included in different threads running at same time, `GetData` and `SendData` methods use the lock statement along with a respective object, _lock and _sendlock. This statement is also used in the functions used to save messages in PCL and in the functions that filter messages.

The figure 4.12 illustrates the usage of some of the PCF DLL's methods to communicate with a device using an Ethernet interface. It is possible to observe the interaction between Classes and how messages are sent and received. In the **EthernetDevice** line is included the **Socket Class** functions calls.

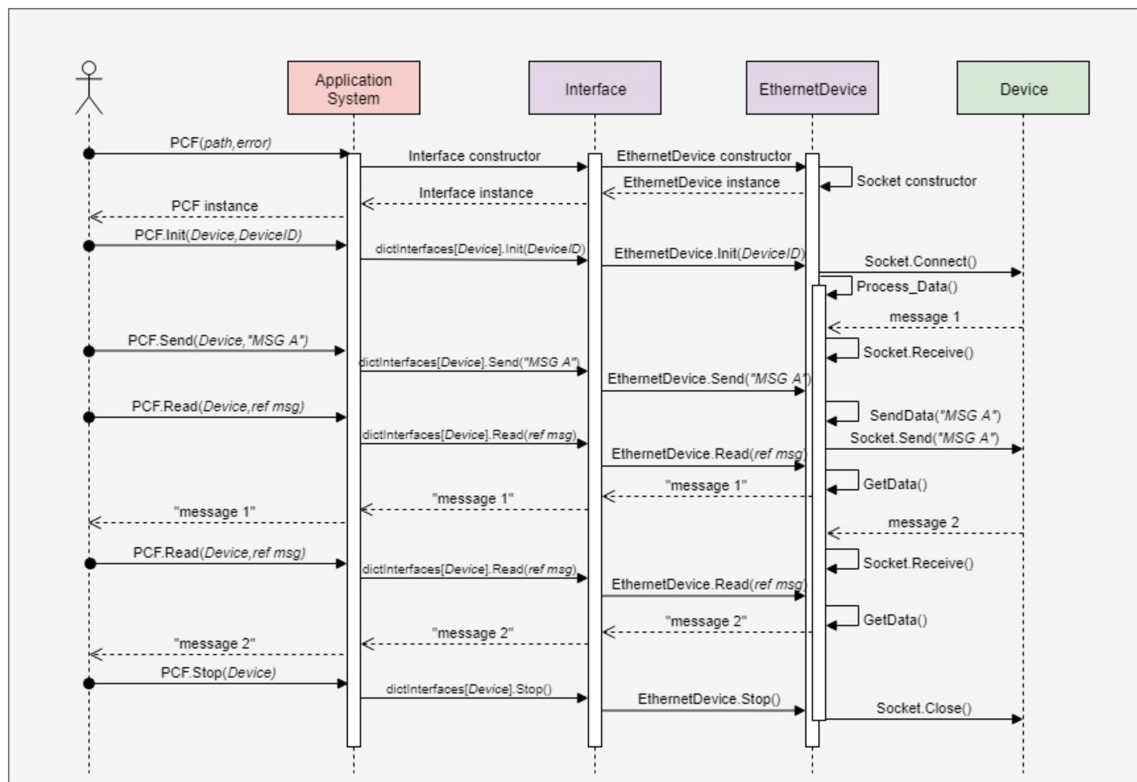


Figure 4.12: Example of interaction between an application and device using PCF DLL's methods to communicate over Ethernet.

Finally, **EthernetDevice Class** includes the methods that interact directly with the filter's **Dictionary Class** instance. This methods permit to add new filters, verify messages comparing them to existing filters, clear saved filtered messages and remove filters from the filter's Dictionary.

4.1.2.3 CAN functions

As mentioned before, to establish CAN connections, Aptiv's Engineers use *Vector* devices as interface between their products and test systems. In order to use these devices, *Vector* provides various tools such as applications like *CANalyser*, *CANape* and *CANoe*, and a DLL, XL Driver Library [32], which enables the development of applications for CAN, among other protocols, on supported *Vector* devices, which applies to the implementation of PCF DLL's CAN functions.

Besides XL Driver Library function calls, *Vector Hardware Config* tool is required to set up the hardware settings.

With that said, it became imperative to use part of the XL Driver Library function calls for CAN applications (fig.4.13) in the implementation of **CANDevice Class** (fig.4.14). Thus, this functions' calls were integrated in different methods of the **CANDevice Class** in order to keep a structure similar to the other interfaces Classes.

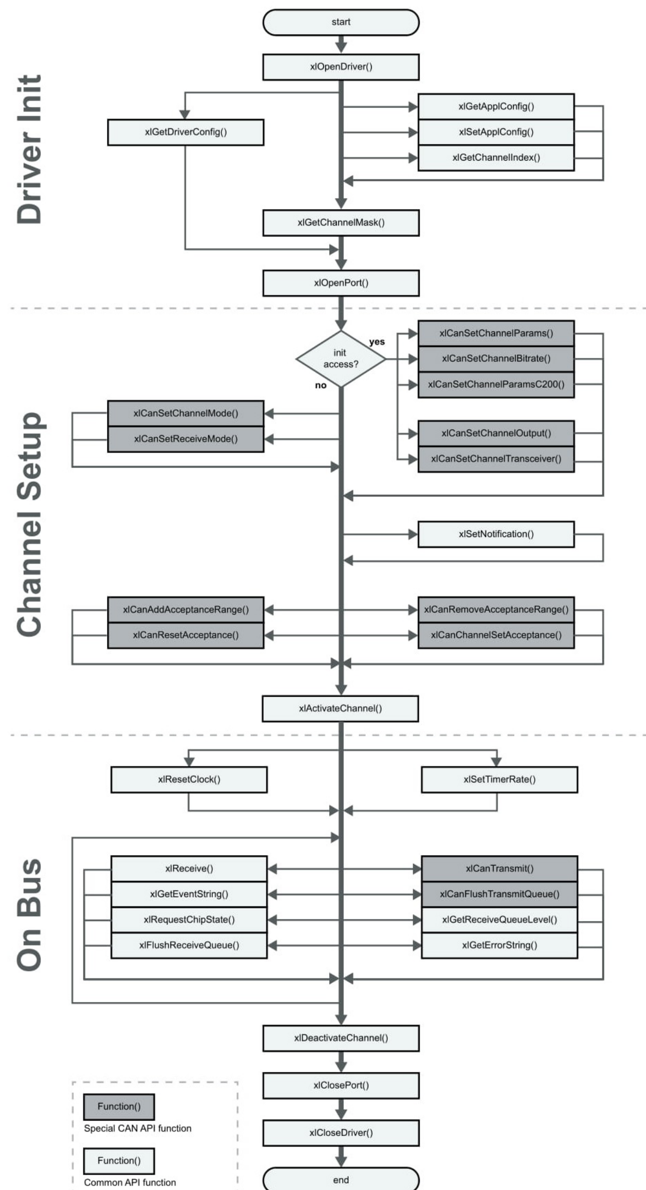


Figure 4.13: XL Driver Library function calls for CAN applications, [32].

The main objective to implement a Class specifically to CAN connections is to address the requirements vii and ix specified before in 3.1, that is establish CAN connections and send, receive and wait for messages using this connection type.

Following the structure of the others devices classes, **CANDevice Class** (fig.4.14) methods can be divided in three groups:

i. the first one concerns the constructor for this Class. As mentioned before, it is used by `XMLParser` method along with the respective **Interface Class** constructor when a CAN device is present in PCCF and consequently a new instance of this class must be created in order to communicate with it;

ii. a second group includes all the methods responsible for guarantee the establishment of the connection, enabling the possibility of send, receive and wait for messages, save those messages and check the state of the connection; and

iii. a third group concerns the methods used to filter messages in the connection.

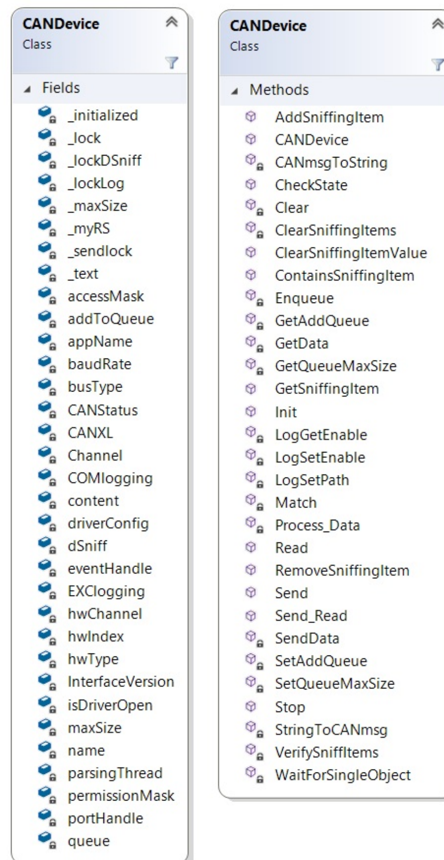


Figure 4.14: **CANDevice Class**.

CANDevice Class constructor defines the fields needed to use a *Vector* device to communicate using CAN protocol, namely the hardware type, that corresponds to the device being used, the channel of this device and the baud rate. The constructor also defines the remaining fields for the **CANDevice Class** object that it is creating. These fields are the name for the instance and the paths for creating the files where the communication content and errors shall be saved. The constructor also creates a **Queue Class** instance to save received messages and a **Dictionary Class** instance to handle the filters that can be created.

To establish the connection the method used is `Init` which verifies in first place if the *Vector* driver is already being used, if not `Init` method uses `XL_OpenDriver` and `XL_GetDriver-Config` to configure the driver. After this, `Init` method verifies if PCF DLL is already defined as an application in *Vector Hardware Config*, if it is not defined it uses `XL_SetApplConfig` function call, otherwise it uses `XL_GetApplConfig` to read the channel assignments. Then,

XL_GetChannelMask, XL_OpenPort, XL_CanSetChannelBitRate and XL_ActivateChannel are invoked by this order. Finally, the Init method creates a thread (Process_Data) responsible to handle received messages and put them in the **Queue Class** instance and in the PCL created for saving them. This thread waits for receiving messages using XL_Receive function call. Since XL Driver defines a structure for CAN messages, there is the need to convert this structure into a string, which accomplished using CANDevice Class' CANmsgToString method. Every time a message is received, it is verified if it contains any pattern from the filters that can be included in the filter's **Dictionary Class** instance. This initialization is illustrated in the figure 4.15.

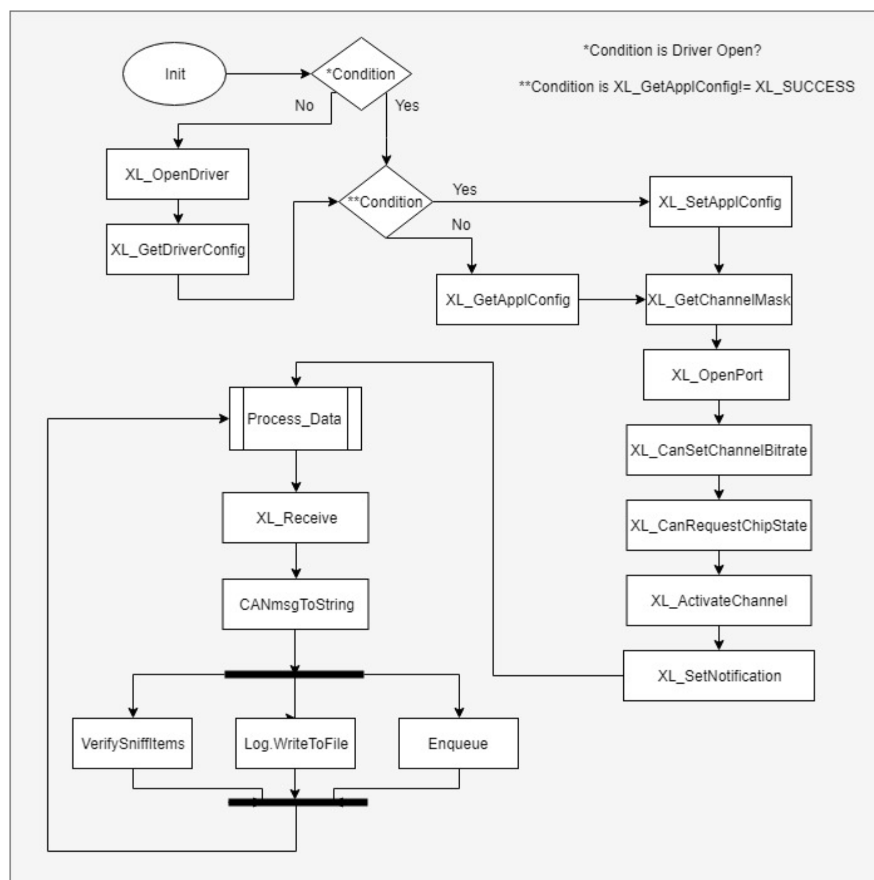


Figure 4.15: Initializing a CAN Device.

Contrary to the other devices classes, the format mentioned for diagnostic commands using RS-232 does not apply to CANDevice Class.

Thus, **CANDevice Class** contains a Read method, that uses **CANDevice Class**' GetData method to get messages from the **Queue Class** instance where received messages are being saved.

The only Send method in **CANDevice Class**, receives as argument a string with a specific format (i.e "ID:0x1111|Data:0x01 0x02 0x03"), calls SendData method where this message is converted to the CAN message structure from XL Driver using StringToCANmsg method and

then effectively send message using `XL_CanTransmit` function call from XL Driver Library. If successful it saves the sent message in the PCL.

Additionally, in **CANDevice Class** the method to send a message and wait for a specified response it is used its `SendData` and `GetData` methods directly.

To terminate the established connection, **CANDevice Class** contains the `Stop` method which dispose all the resources being used by the respective instance and closes the connection using `XL_DeactivateChannel` and `XL_ClosePort` function calls from XL Driver Library.

The communication state can be check with the `CheckState` method which returns the _myRs field that indicates if the connection is either in Run, Stop, Stopped or Running states.

As in other devices, in order to enable the use of the methods that read and send messages included in different threads running at same time, `GetData` and `SendData` methods use the lock statement along with a respective object, _lock and _sendlock. This statement is also used in the functions used to save messages in PCL and in the functions that filter messages.

The figure 4.16 illustrates the usage of some of the PCF DLL's methods to communicate with a device using an CAN interface. It is possible to observe the interaction between Classes and how messages are sent and received. In the **CANtDevice** line is included the XL Driver functions calls.

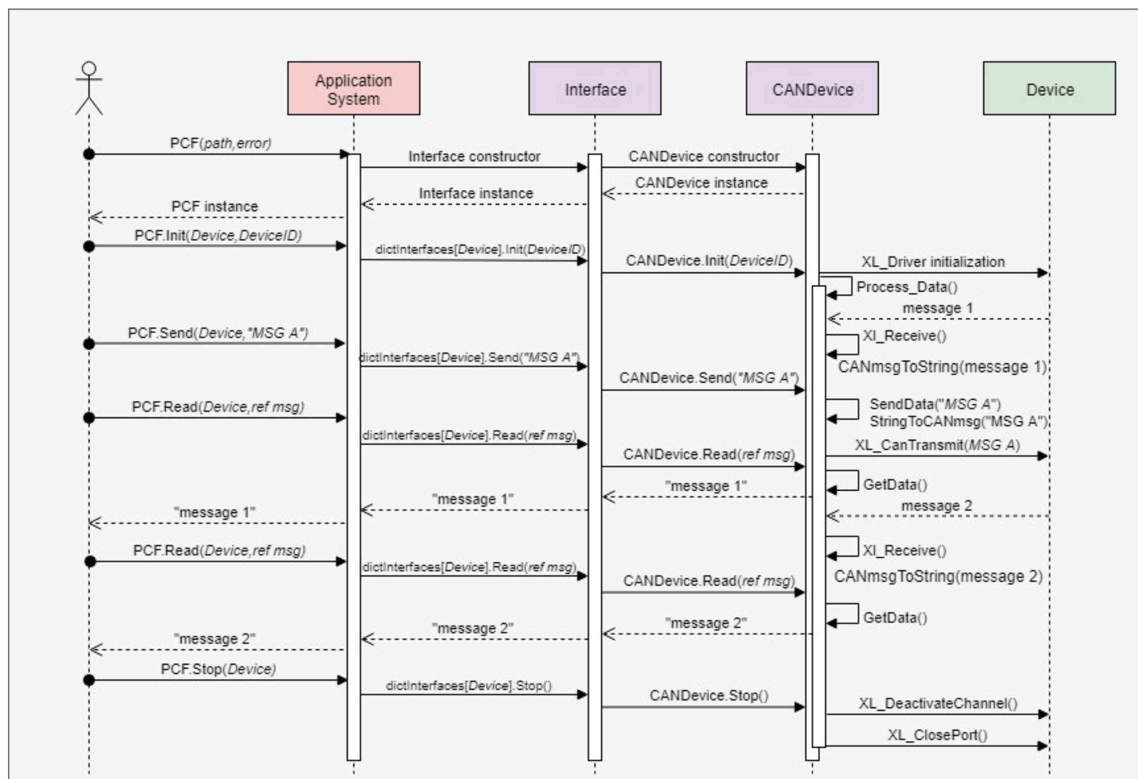


Figure 4.16: Example of interaction between an application and device using PCF DLL's methods to communicate over CAN.

Finally, as in the other Device Classes, **CANDevice Class** includes the methods that interact directly with the filter's **Dictionary Class** instance.

4.2 Product Communication Monitor

The following sections have as objective to describe how the applications that make possible to remotely access the communication logs created using PCF DLL were implemented.

In a first phase, it was implemented a *Windows* Service, called *Product Communication Monitor Service*, responsible to give access to the communication logs created using PCF DLL functions. Working as a TCP/IP server, this service permits the connection of multiple clients to the system where PCF DLL functions are being used. Furthermore, if desired, this service can manage the number of communication logs stored in the system.

Despite the multiple software that permits to communicate with PCMS, it was implemented an application specifically design to communicate with these service after its creation, in order to give a more user friendly interface and permit to download the PCLs to a remote location where this user interface is running.

4.2.1 Product Communication Monitor Service

Using *Visual Studio*'s template for*Windows* Services, it was implemented the *Product Communication Monitor Service* with the main objective to address the requirement xiv defined before in 3.1. This service shall be installed in the systems where PCF DLL is being used in order to remotely access the PCLs, otherwise this is not possible.

Once PCMS is started, two different threads are initialized, one creates the Server and the other manage the PCLs in the system. Both threads use a similar Parser to `XMLParser` from PCF DLL in order to obtain the information from PCCF and use a Log Class (fig. 4.4) instance to store errors that can occur in both threads into log files.

Thus, PCMS main functionalities are divided in two main groups:

i. the first one concerns the functions responsible to create a TCP/IP Server, responsible to communicate with multiple clients; and

ii. a second group includes all the methods responsible to manage PCLs in the system.

Similar to the methods used to implement the communication functions for Ethernet devices, **Socket Class** from .Net is used in the implementation of the TCP/IP server of the PCMS. This time, the methods used are `Bind` and `Listen` in order to wait for new connections from remote systems that want to access PCLs. When a client connects to the server, a new Socket is created in a Async Call Back with `BeginAccept` method from the mentioned Class. Finally, `Shutdown` and `Close` methods are used to destroy the socket when the connection with the client terminates.

With the connection established, the service waits for a pre-defined command from the client in order to give the respective response:

- ”-l” - list all available devices, respective interface and product numbers available to monitor;

- ”-lc” - list all available devices, respective interface’s parameters and product numbers available to monitor;
- ”-ml” followed by identifier integer and product number (i.e. -ml0[1234]) - load the PCL of the respective device;
- ”-ms” followed by identifier integer (i.e. -ms0) - continuously monitor the most recent device for given interface;
- ”-msui” followed by identifier integer (i.e. -msui0) - command used by PCMUI to continuously monitor the most recent device for given interface;
- ”-c” - end connection with the client; and
- ”-h” - list available commands and respective functionality.

The number of available interfaces to monitor is limited to four as specified in the requirement iv from 3.1.

The thread responsible to manage PCLs, parses the information from PCCF to know where PCLs are saved and which devices are defined. Then, PCMS uses **DirectoryInfo Class** [33] from .Net framework to load information from the directory where, for a certain device defined in PCCF, PCLs are saved. PCMS manipulates the information retrieved by `GetFiles` method in order to limit the number of files for a certain device to five, deleting exceeding files using `Delete` method from **File Class** [34].

PCMS has as editable settings the name of PCCF and the name of exception logs files that must be contained in the directory where the service is installed. Another setting mutable is the Port that shall be opened for incoming connections from clients to the Service’s server. Finally, the file manager can be either activated or deactivated using a Boolean setting that works as a flag to indicate if it is pretended to have the file manager running or not.

The User can install the described service manually [35], running the command ”installutil” (i.e ”installutil pcms.exe”) in the Developer Command Prompt for *Visual Studio*, or install the Service automatically using the installer implemented in *Visual Studio* using the Setup Project template.

After PCMS is installed, communication with this service can be established using any telnet client like *PuTTY* or PCMUI, the software developed specifically with this purpose and that is described in the following section 4.2.2.

4.2.2 Product Communication Monitor UI

The main objective accomplished with the implementation of PCMUI is enable the User able to download to his system a selected PCL stored in another system where PCF DLL functions are being used to communicate with different devices while creating the respective communication logs and where PCMS is installed, giving remote access to those logs.

In addition, PCMUI permits the user to see the messages flow live in a connection that is being established using PCF DLL functions.

PCMUI was implemented using *Microsoft Visual Studio* along with its *Windows* Forms Application project template to be written in C#.

When the application developed starts, the main Form (fig. 4.17) appears and the User shall introduce a name, a IP address and a port number corresponding to the system where PCMS is running and which intends to connected with. This Form is always open while PCMUI is running, which enables the connection to multiple remote systems or multiple connections to the same remote system.

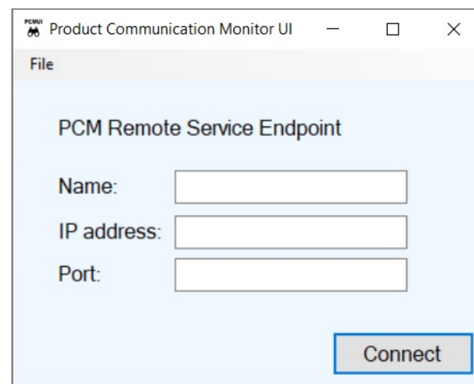


Figure 4.17: PCMUI homepage.

Once the User introduces valid parameters and clicks on connect, a new client is created, using **Sock Class**' methods, and a new Form (fig. 4.18) appears, corresponding to the connection established with PCMS and where several information is presented.

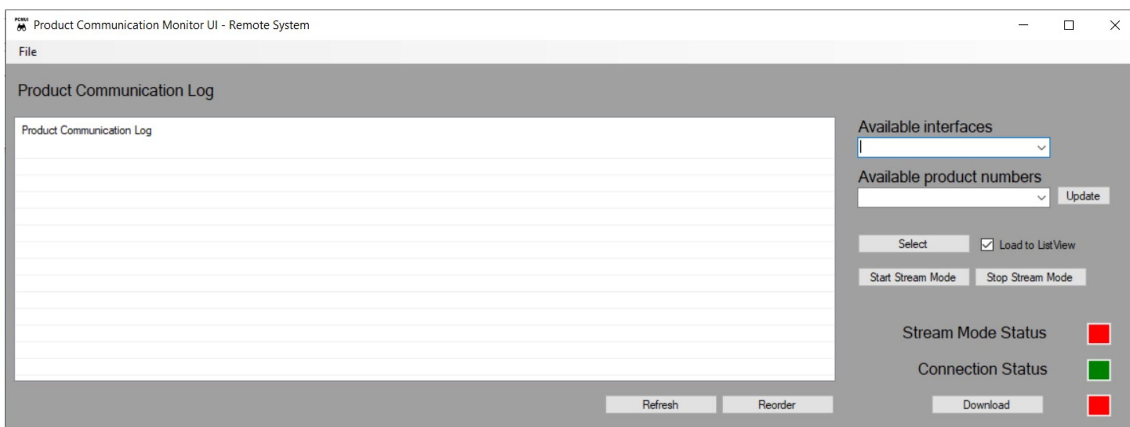


Figure 4.18: PCMUI connection to remote system page.

In this Form, the user can consult the available interfaces to monitor. Selecting an interface, the respective available product numbers are loaded into the ”Available product numbers” combo

box. At anytime, the User can check for new available product numbers for a selected interface, clicking on the **Update** button.

With an interface and a product number chosen and clicking in the **Select** button, the PCMUI loads into memory the PCL received from PCMS, if ”Load to ListView” check box is selected, the PCL received is also loaded to the List View present in the Form. The **Refresh** button loads again to the list view the selected PCL and the **Reorder** button inverts the order of the PCL showed in the List View.

When a PCL is saved in memory, the User can click on **Download**. Doing this, will pop up a new form where specifying a name for the new file, a valid path to save the file and clicking on **Save** will create a copy of the received PCL in the system where PCMUI is running. To do so, first PCMUI checks if the specified path is a valid one, then if it does not exist, the directory is created using the `Create` method from the **DirectoryInfo Class**. Finally, PCMUI uses `WriteAllLines` method from **File Class** to copy the PCL from the memory to a new text file.

PCMUI also enables the User to monitor continuously a certain communication established in the remote system. To do so, the User must select the most recent product number, which is signaled with the character ***** before its name, and click on **Start Stream Mode**. By doing this, the User will activate a timer that when ticks will request PCMS the last lines of the PCL from the selected product number, then compares it to the previously received to check if new messages were exchanged between the remote system and the selected device. If two consequent messages are equal then the mentioned timer is stopped and stream mode is deactivated. While stream mode is active, it can be deactivated by clicking on **Stop Stream Mode** button.

There is a lot of memory consumption since PCMUI requires the allocation of a considerable memory to receive the PCLs using the buffer of a socket, to save those content into memory and to load it to the List View. To mitigate this consumption, PCMUI uses **GC Class** [36] to control the system garbage collector, automatically reclaiming unused memory, concretely, from time to time, `Collect` and `WaitForPendingFinalizer` methods are called.

PCMUI can be installed in a certain *Windows* system using the respective installer created using Setup Project template from *Microsoft Visual Studio*.

The figure 4.19 illustrates an example of an interaction between PCMUI and PCMS. In this example, the User connects to PCMS using PCMUI, in order to download to his system a communication log file of a certain device with a specified product number.

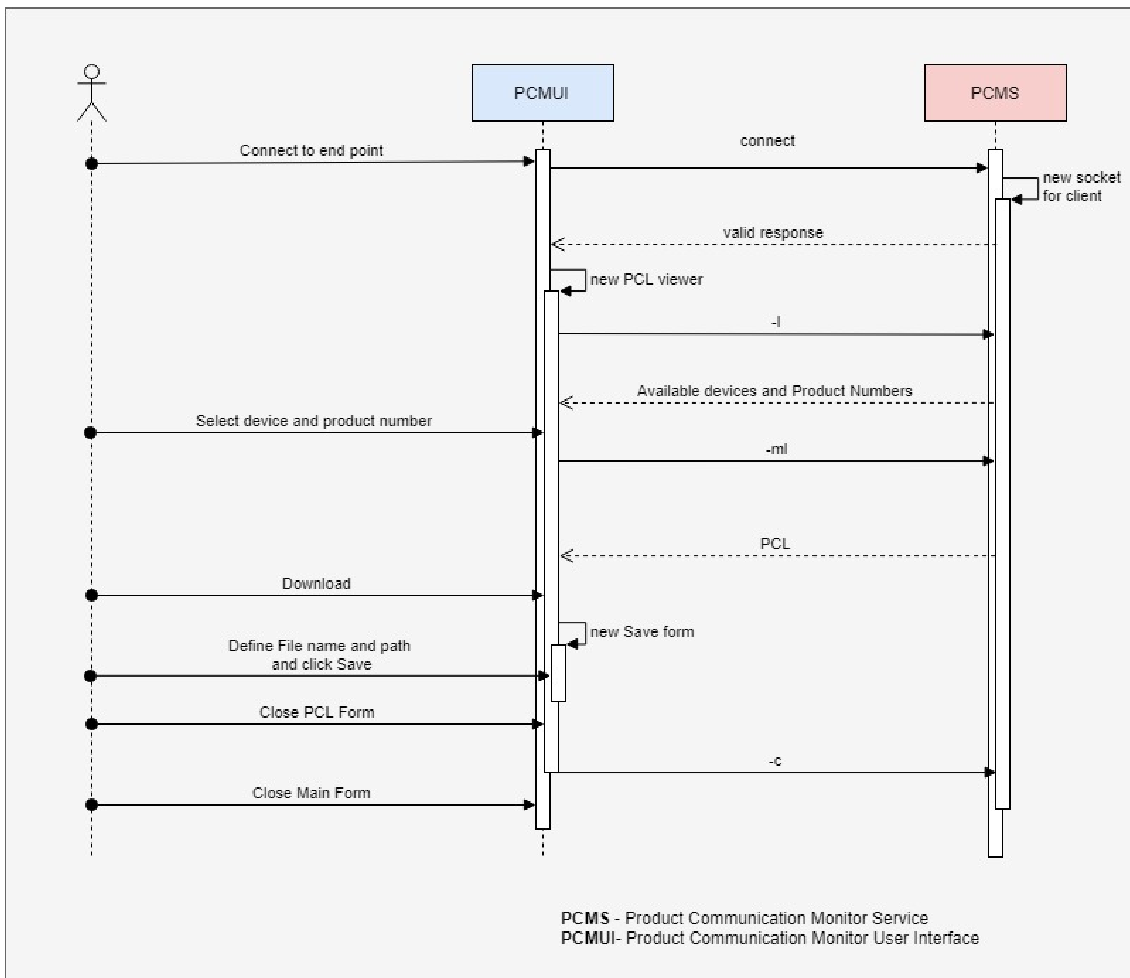


Figure 4.19: Example of interaction between PCMUI and PCMS.

Chapter 5

Experimental Validation

This chapter 5 is constituted by a single section used to present the experimental validation.

5.1 Testing Framework

During the development of the PCF DLL, communication functions for each interface were built and then debugged in order to guarantee their correct functionalities.

Besides the debug done, several parts of code inside the functions that prevent errors that could stop the applications where PCF DLL is being used. Additionally, there are code that verifies parameters to guarantee that unexpected behaviors that could lead to wrong results do not happen.

With the framework development completed, it is important to test it in a controlled environment before it can be used in real applications like in Production lines. Testing in such environment prevents the framework functions to pass wrong variables from the products to the test systems which could lead to consider good products to be defective ones.

Since one of the objectives of the framework is to be included in other applications as in test systems, *NI TestStand* was the platform chosen to perform the experimental tests.

The tests presented are an example of the ones done for the different interfaces, in order to illustrate the method used to test all of them. In the case presented, it was created a test sequence to communicate with MIB3-Top using an RS-232 interface. While the sequence was running, verifying the DLL's functions correct performance, it was used PCMS and PCMUI to monitor remotely the connection.

Due to confidentiality issues some of the figures contain black boxes covering the diagnostic commands used.

5.1.1 NI TestStand sequences

Since test systems use *NI TestStand* sequences to test the products, there is the need to build new sequences using the PCF DLL functions in order to use the framework functionalities.

The sequence used in the following example was developed in order to be used with MIB3-TOP directly with RS-232 interface or using *Brainboxes* ES-257, just by defining a RS-232 interface or an Ethernet interface, respectively, in the configuration file and then initialize it using the correspondent name. In the following example, the interface used was RS-232, as mentioned before.

The sequence developed (fig.5.1) is separated in three parts, each one constituted by different steps that use the methods present in the DLL, with the objective of validate them.

Test UUTs - TestSequence.seq [12] ... TestSequence.seq [Executing]

Steps

Step	Description	Settings	Module Time	Status
Setup (2)				
Load Configuration	Call Load Configuration in <Current File>		0.0314151	Passed
Init Interface	Call Init Interface in <Current File>		0.0548625	Passed
<End Group>				
Main (16)				
Enter Diag	Call Enter Diag in <Current File>		0.3183697	Passed
Terminal Filter	Call Terminal Filter in <Current File>		0.1607197	Passed
Get IOC SW Version	{21}, Numeric Limit Test, x > 5, Call SW Versions in <Current ...		0.9814762	Passed
Get CPA SW Version	Call SW IDs in <Current File>		0.2004469	Passed
Get INIC SW Version	Call SW IDs in <Current File>		0.1415306	Passed
Sniffing IOC Version	Call Sniffing in <Current File>		1.1358639	Passed
Read Board ID From Persistece	Call PS Read in <Current File>		0.0918462	Passed
Read A2B Configuration	Call PS Read in <Current File>		0.0833726	Passed
Write A2B Configuration	Call PS Write in <Current File>		0.168769	Passed
Read A2B Configuration	Call PS Read in <Current File>		0.1687588	Passed
Power Check	Call Power Check in <Current File>		5.3956239	Passed
Wait	TimeInterval(20)			Done
Recovery Mode	Call RecoveryMode step1 SOP1 in <Current File>		35.0514594	Passed
Primary Source	Call Primary Source in <Current File>		0.1585815	Passed
Set Frequency	Call Set Frequency in <Current File>		0.4754933	Passed
Set VOLUME	Call Set Volume in <Current File>		0.1635791	Passed
<End Group>				
Cleanup (1)				
Stop Interface	Call Close Interface in <Current File>		0.4381475	Passed
<End Group>				

Steps Variables Report

Figure 5.1: Test sequence used for testing the framework's functionalities (RS-232 Example).

The first part is called Setup. It starts with **Load Configuration**, that is used to invoke the constructor of the PCF object, to be used by the remaining steps. In this case, the configuration file, PCCF, content is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Devices>
<ExceptionLogPath>C:\Users\gjfz41\Desktop\PCF v3\
   PCF_exceptionLOG.txt</ExceptionLogPath>
<Device>
        <id>MIB3RS</id>
        <Parameters>
                <CommunicationType>Serial</CommunicationType>
                <portname>COM4</portname>
                <baudrate>115200</baudrate>
                <parity>None</parity>
                <databits>8</databits>
                <stopbits>One</stopbits>
                <WriteTimeout>1000</WriteTimeout>
                <ReadTimeout>1000</ReadTimeout>
```

```
                <CommunicationLogPath>C:\Users\gjfz41\Desktop\
                   LOGS-FINAL TESTS\Com\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\gjfz41\Desktop\LOGS-
                   FINAL TESTS\Error\</ExceptionLogPath>
        </Parameters>
</Device>
<Device>
    <id>MIB3ETH</id>
        <Parameters>
                <CommunicationType>Ethernet</CommunicationType>
                <IP>10.238.227.20</IP>
                <Port>9001</Port>
                <AddressFamily>2</AddressFamily>
                <BufferSize>1024</BufferSize>
                <CommunicationLogPath>C:\Users\gjfz41\Desktop\
                   LOGS-FINAL TESTS\Com\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\gjfz41\Desktop\LOGS-
                   FINAL TESTS\Error\</ExceptionLogPath>
        </Parameters>
</Device>
</Devices>
```

The **Setup** is concluded when the interface is initialized, using the `Init` function from the DLL. If the sequence is paused at this point, it can be verified that communication is established as supposed consulting the respective PCL, where received messages are being saved.

The **Main** part starts by sending the command that makes the device in test to enter in diagnostic mode, using the `Send_Read` method for diagnostic commands in order to first send the respective command and then read the expected response. With the diagnostic mode activated, the device is ready to receive other diagnostic commands.

After entering in diagnostic mode, the step **Terminal Filter** is used to filter the messages sent by the device, restricting them to responses to the commands sent.

The following three steps, **Get IOC SW Version**, **Get CPA SW Version** and **Get INIC SW Version** also use the `Send_Read` method for diagnostic commands in order to get the software versions present in the various processors of the device in test.

Sniffing IOC Version (fig.5.2) has a similar objective as **Get IOC SW Version**, but in this step the methods used and respective functionalities intended to verify are different. First, it is added an item to the filter structure using the `AddingSniffingItem` method, in this case it is intended to check if "Version:" occurs. Then it is sent a message, concretely "version", using the `Send` method for raw messages. After sending the message, `GetSniffingItemValue` is invoked within a timeout in order to check if the filter occurred. Finally, the filter is removed from structure using `RemoveSniffingItem` method.

Steps: Sniffing

Step	Description	Settings
Setup (1)		
Add Sniffing Key	Action, ProductCommunicationFramework...	
<End Group>		
Main (6)		
Label		
send Value	Numeric Limit Test, x == 0, ProductCommu...	
Get Value		
Wait	TimeInterval(1)	
Get Value	Action, ProductCommunicationFramework...	
Goto	Goto Get Value	Precondition, Post Expression, Post Action
<End Group>		
Cleanup (1)		
Remove Sniffing Key	Action, ProductCommunicationFramework...	
<End Group>		

Figure 5.2: Test sequence's step used for testing the framework's filtering methods.

After using filtering methods, the PCF DLL methods for communicating were used to write values in memory and then verify if those values were indeed written in the memory of the device.

Besides this validation, it was also performed other tests that permit to verify if the functions used to communicate work indeed. The **Power Check** step turns off and then turns on one of the antennas present in the device. Using a multimeter, it was verified that when the step occurs, the voltage of the antenna drops and then comes up to the initial level.

The **Recovery Mode** step is used to send a command that permits to control the Inputs/Outputs (I/O) of the device, with the objective of choose the output source, set its frequency and set its volume, in the following steps, namely **Primary Source**, **Set Frequency** and **Set VOLUME** respectively. Connecting a speaker to the radio, it was verified that radio started to play audio when the commands are sent on these steps.

Finally, the **Cleanup** part of the sequence is used to stop the communication with the device in test using the PCF DLL's `Stop` method. After running this step the sequence is finished and consulting the respective PCL it was verified that the communication is terminated.

5.1.2 Validation of the Remote Monitoring

After building the sequence presented in 5.1.1 using PCF DLL functions and validate its results, the monitor modules of the framework can be tested.

In this validation test, PCMS is running in the system where the sequence is used. Both use the same PCCF, previously presented in 5.1.1, in order to guarantee that interfaces defined are the same and no errors occur when the communication established by the sequence is being monitored.

The PCCF content can be checked establishing a connection with PCMS using for example *PuTTY* (fig.5.3) or other software similar to it and sending the command "-lc". In this example any product have been tested consequently, there is any available product to monitor.

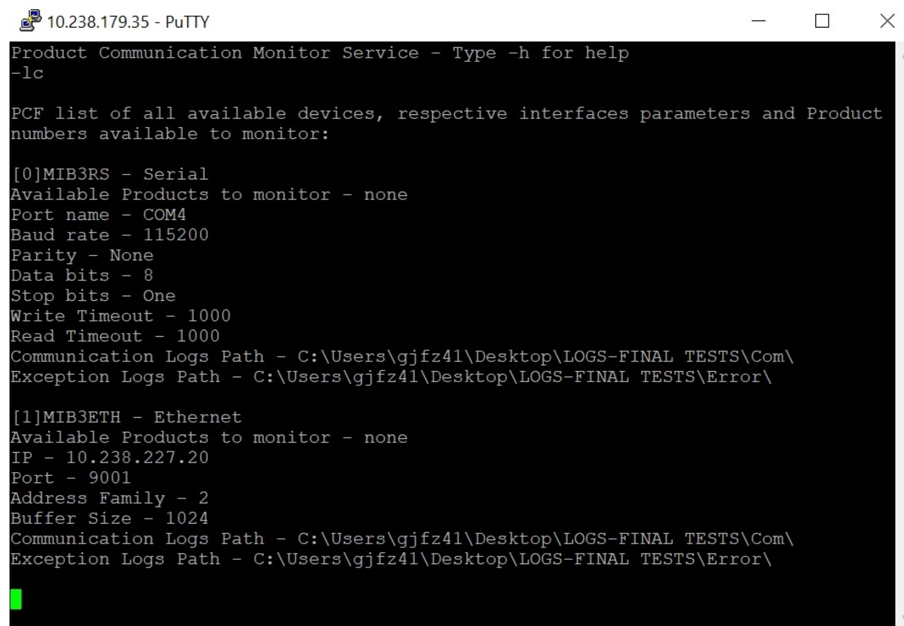


Figure 5.3: Checking PCCF content using *PuTTY*.

Using PCMUI, the available interfaces can be consulted establishing a connection with PCMS (fig.5.4).

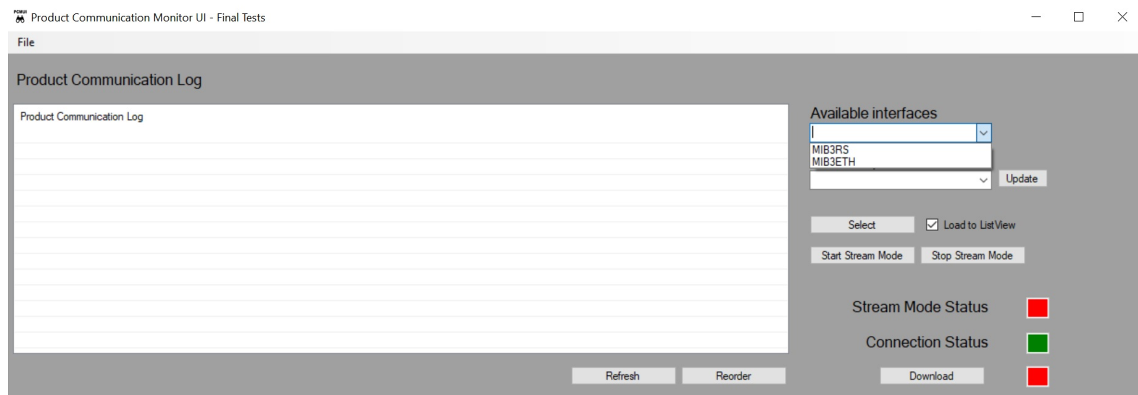


Figure 5.4: Checking available interfaces to monitor using PCMUI.

With the test sequence running, the product number of the device will be then available to be monitored. In the following example, it was used the interface called MIB3RS and defined as product number 12345. Selecting both and starting stream mode, the communication content can be consulted remotely through PCMUI (fig.5.5) while the test sequence and the device exchange messages.

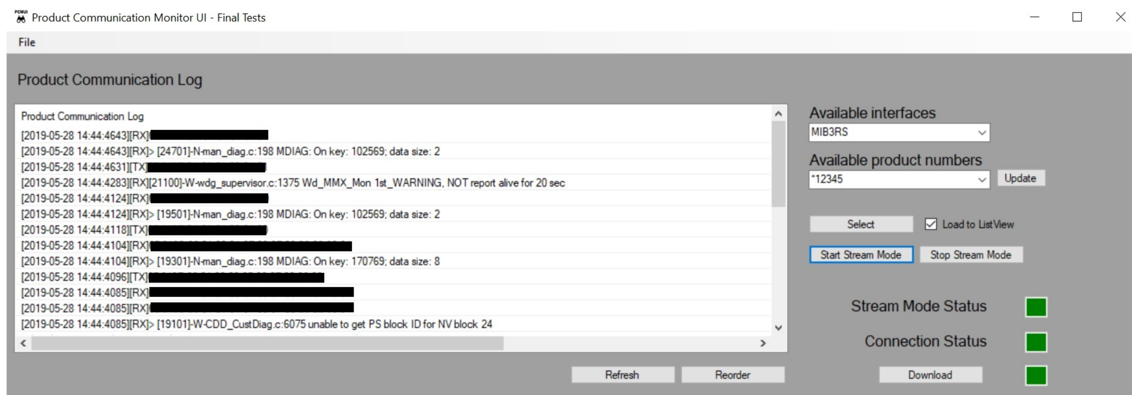


Figure 5.5: PCMUI in stream mode.

When the sequence is finished, it is possible to load to the PCMUI the complete PCL (fig.5.6) and save it on the system where the UI is running. This can be achieved by selecting the right interface and product number and then clicking on download.

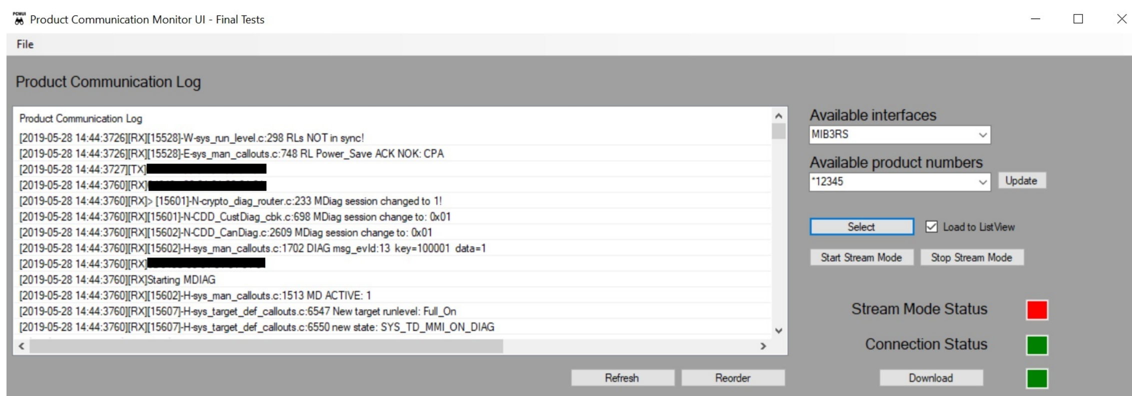


Figure 5.6: PCMUI with PCL in memory.

Then, it was verified that the PCL saved in the system where the test sequence was running and PCMS is running is equal to the one received by PCMUI.

Finally, the test sequence was used five more times, changing only the product number of the device being tested. By doing this, it was possible to verify that PCMS is indeed managing the amount of files stored, deleting the oldest ones and that the product number more recent is the one indicated by PCMUI (fig.5.7).

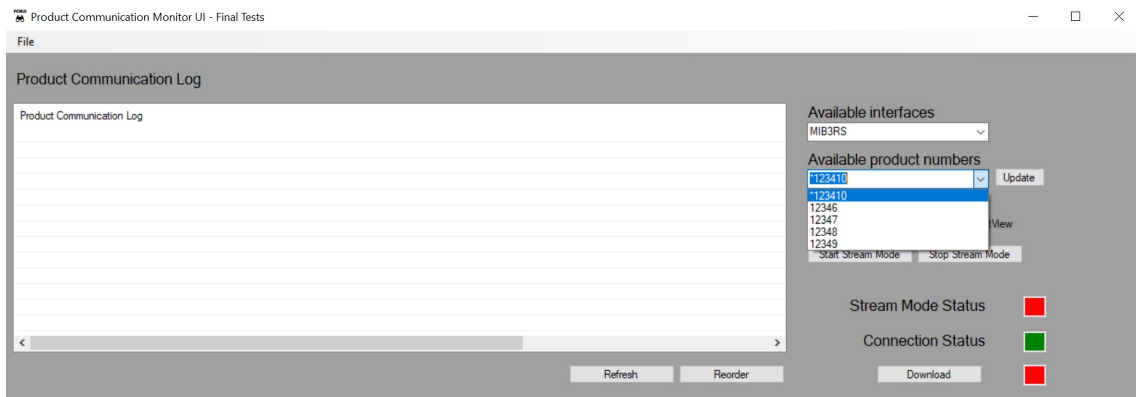


Figure 5.7: Checking available product numbers to monitor using PCMUI.

Chapter 6

Conclusion and future work

This chapter 6 is divided in two main sections used to present this work's conclusions, achievements and proposals for future work.

In the first section are presented the main conclusions and achievements, resulted from the developed work.

In the second section are suggested possible developments and future work proposals, in which the work already done can be improved.

6.1 Conclusions and achievements

Before the development of Product Communication Framework, Aptiv's Engineers had to develop communication tools along with the appearance of new products despite using similar tools to communicate with older products. This procedure resulted in a waste of resources, both human and time versus associated costs, and resulted also in difficulty in sharing the information needed to use those tools, since proper documentation about them was not created. Another issue was that for different communication types, different tools were required.

By adopting PCF, Aptiv's Engineers have now a tool-set that facilitates not only the development of new products but also their tests in Production lines. PCF integrates tools required to communicate over different interfaces, enabling the User to use the same functions for different interfaces, namely RS-232, Ethernet and CAN.

The developed functions are prepared to detect various errors that can occur when a communication is established. This way, connection problems can be distinguished from functional problems of a device. PCF's functions also have components to register those errors in log files. PCF is also composed by functions that can be used to filter certain messages defined by the User.

Furthermore, PCF gives the capability of monitor remotely the connections established, which is of particular interest when the communication functions are being used in Production lines' test systems, facilitating the access to the content exchanged between these systems and the products being tested. This is achieved by using of PCF's *Windows* Service along with a TCP/IP Client, that can be the PCF's Monitor UI.

The framework developed was tested in order to be used in real test systems, but it has enough flexibility to be used to support the development of new applications where communication over RS-232, Ethernet or CAN is a requirement.

The experimental tests presented in this work enables PCF to be used in test systems. For products already in production is difficult to change what is already done, since these changes take a lot of time to be made. On the contrary, for new products, PCF can be very useful to develop the parts of the test sequences where communication over RS-232, Ethernet or CAN is required.

Finally, in order to PCF be used by different Engineers, it was created an User Manual. This documented procedure has all the relevant information not only to use the framework, but also to understand its architecture. Additionally it was given a tutorial, on a *Skype* meeting with the Test Engineering team from Portugal, Germany and Poland. The framework has already been used by Engineers in Aptiv, concretely in its Tech Center in Braga and in the repair site.

It is possible to conclude that all the initial objectives were achieved. The PCF complies with all the requirements defined before its implementation, solving a problem that Aptiv's Engineers faced as a team. It is a very flexible framework that can be used in the testing and production of Infotainment Systems, supported by documentation that makes it easier to use.

The development of this tool-set allowed the contact with the highly demand Industry where Aptiv is inserted. Being a work developed in such environment revealed to be a reinforcement not only professional but also personal. The contact with people of different areas permitted learning competencies, including soft skills, that otherwise would not be possible.

6.2 Future work

PCF's modular design enables the integration of new communication tools. The *Visual Studio* projects are all commented and along with the User Manual, it is easy to introduce new interfaces to the framework. The framework architecture was developed with that in mind, since there are many different communication interfaces in Infotainment Systems. The ones already integrated are the mainly used to test the equipment.

The framework's functions could be integrated in a microcontroller in order to concentrate all the connectors in a small device, saving physical space, for example in laboratory applications.

The filter methods can be used to detect recurrent problems if defined by the User. Other option would be if this could be done automatically, using filters defined in a data base that is connected to the system where the functions are being used. This would also permit that different types of filters could be created. The creation of a module to communicate with data bases would be needed.

Improving the Product Communication Frameworks is possible by integrating in it modules that have communication capabilities. Being Communication the main aim of this framework, adding such capabilities would possibility the usage of the framework in applications from different areas of the Infotainment Systems' Production while interconnecting them.

Appendix A

Product Communication Configuration File Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Devices>
<ExceptionLogPath>C:\Users\UserName\Desktop\PCF\PCFexceptionLog.
   txt</ExceptionLogPath>
<Device>
        <id>RS232Example</id>
        <Parameters>
                <CommunicationType>Serial</CommunicationType>
                <portname>COM4</portname>
                <baudrate>115200</baudrate>
                <parity>None</parity>
                <databits>8</databits>
                <stopbits>One</stopbits>
                <WriteTimeout>1000</WriteTimeout>
                <ReadTimeout>1000</ReadTimeout>
                <CommunicationLogPath>C:\Users\UserName\Desktop\
                   PCF\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\UserName\Desktop\PCF\
                   </ExceptionLogPath>
        </Parameters>
</Device>
<Device>
        <id>EthernetExample</id>
        <Parameters>
                <CommunicationType>Ethernet</CommunicationType>
                <IP>10.238.111.20</IP>
```

```
                <Port>9001</Port>
                <AddressFamily>2</AddressFamily>
                <BufferSize>1024</BufferSize>
                <CommunicationLogPath>C:\Users\UserName\Desktop\
                   PCF\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\UserName\Desktop\PCF\
                   </ExceptionLogPath>
        </Parameters>
</Device>
<Device>
        <id>CANExample</id>
        <Parameters>
                <CommunicationType>CAN</CommunicationType>
                <HardwareType>59</HardwareType>
                <Channel>1</Channel>
                <BaudRate>500000</BaudRate>
                <CommunicationLogPath>C:\Users\UserName\Desktop\
                   PCF\</CommunicationLogPath>
                <ExceptionLogPath>C:\Users\UserName\Desktop\PCF\
                   </ExceptionLogPath>
        </Parameters>
</Device>
</Devices>
```

References

[1] The evolution of car advertising. https://www.appnova.com/from-selling-men-dreams-to-ugc-the-evolution-of-car-advertising-in-histo
Accessed: 2019-01-07.

[2] Pierre Audoin Consultants cited in Consultancy UK (2015). BearingPoint: 80% of new vehicles connected by 2020. https://www.consultancy.uk/news/2353/bearingpoint-80-percent-of-new-vehicles-connected-by-2020/. Accessed: 2019-01-07.

[3] The Royal Society for the Prevention of Accidents. Road Safety Factsheet. https://www.rospa.com/rospaweb/docs/advice-services/road-safety/vehicles/infotainment-systems-factsheet.pdf, October 2018. Accessed: 2019-01-07.

[4] Aptiv's website. https://www.aptiv.com/. Accessed: 2019-01-07.

[5] NI TestStand - Getting Started with TestStand. https://www.ni.com/pdf/manuals/373436f.pdf. Accessed: 2019-01-14.

[6] Brainboxes Ethernet to Serial, ES - Range Product Manual. http://www.brainboxes.com/files/catalog/product/ES/ES-257/documents/ES%20Range%20Product%20Manual%203.5.pdf. Accessed: 2019-03-25.

[7] VN1600 Interface Family Manual. https://assets.vector.com/cms/content/products/VN16xx/docs/VN1600_Interface_Family_Manual_EN.pdf. Accessed: 2019-04-08.

[8] Rohde & Schwarz SMBV100A Operating Manual. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_common_library/dl_manuals/gb_1/s/smbv/SMBV100A_OperatingManual_en_17.pdf. Accessed: 2019-01-14.

[9] Rohde & Schwarz SFC Compact Modulator Operating Manual. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_common_library/dl_manuals/gb_7/sfc_2/SFC_GettingStarted_en_06.pdf. Accessed: 2019-01-14.

[10] Rohde & Schwarz SFE100 Operating Manual. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_common_library/dl_manuals/gb_7/sfe_100/SFE100_GettingStarted_en_13.pdf. Accessed: 2019-01-14.

[11] Telecommunications Industry Association. TIA-232-F Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange. Standard ANSI/TIA/EIA-232-F, American National Standards Institute, Arlington, USA, 1997.

[12] D-sub 9 Connector Pinout. https://www.db9-pinout.com/. Accessed: 2019-01-14.

[13] J. Axelson. *Serial Port Complete: The Developer's Guide, Second Edition*. Complete Guides Series. Lakeview Research, 2007.

[14] Charles E. Spurgeon. *Ethernet The Definitive Guide*. O'Reilly, First edition, 2000.

[15] Rion Hollenbeck. The IEEE 802.3 Standard (Ethernet): An Overview of the Technology, September 2001.

[16] Telecommunications Industry Association. Commercial Building Telecommunications Cabling Standard – Part 2: Balanced Twisted-Pair Cabling Components. Standard ANSI/TIA/EIA-568-B.2-2001, American National Standards Institute, Arlington, USA, 2001.

[17] Wolfhard Lawrenz. *CAN System Engineering From Theory to Pratical Applications*. Springer, Second edition, 2013.

[18] CAN A Serial Bus System - Not Just For Vehicles. http://www.esd-electronics-usa.com/Controller-Area-Network-CAN-Introduction.html. Accessed: 2019-01-14.

[19] Gianluca Cena and Adriano Valenzano. Controller Area Networks for Embedded Systems. In *Embedded Systems Handbook: Networked Embedded Systems*, pages 15–25. Taylor and Francis Group, 2009.

[20] CiA - CANopen - The standardized embedded network. https://www.can-cia.org/canopen/. Accessed: 2019-01-14.

[21] SAE J1939 - Serial Control and Communications Heavy Duty Vehicle Network. Standard, Society of Automotive Engineers, USA, 2018.

[22] DeviceNet Overview. https://www.odva.org/Technology-Standards/DeviceNet-Technology/Overview. Accessed: 2019-01-14.

[23] Dictionary<TKey,TValue> Class. https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2? Accessed: 2019-03-04.

[24] Queue Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue. Accessed: 2019-03-04.

[25] System.XML Namespace Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.xml. Accessed: 2019-03-04.

[26] SerialPort Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.io.ports.serialport. Accessed: 2019-03-11.

[27] System.IO.Ports Namespace Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.io.ports. Accessed: 2019-03-11.

[28] Lock statement Documentation. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement. Accessed: 2019-03-12.

[29] Lydia Parziale, David T.Britt, Chuck Davis, Jason Forrester, Wei Liu, Carolyn Matthews, and Nicolas Rosselot. *TCP/IP Tutorial and Technical Overview*. IBM, Eighth edition, 2006.

[30] Socket Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket. Accessed: 2019-03-25.

[31] System.Net.Sockets Namespace Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets. Accessed: 2019-03-25.

[32] XL Driver Library Manual. https://assets.vector.com/cms/content/products/XL_Driver_Library/Docs/XL_Driver_Library_Manual_EN.pdf. Accessed: 2019-04-08.

[33] DirectoryInfo Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.io.directoryinfo. Accessed: 2019-04-22.

[34] File Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.io.file. Accessed: 2019-04-22.

[35] How to: Install and uninstall Windows services. https://docs.microsoft.com/en-us/dotnet/framework/windows-services/how-to-install-and-uninstall-services. Accessed: 2019-04-29.

[36] GC Class Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.gc. Accessed: 2019-05-06.