

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



# **CAN and Ethernet bus sniffer**

**João Francisco Sá Santos**

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Paulo Portugal

Co-Supervisor: Eng. Tiago Rodrigues

September 21, 2020



# Resumo

O tema deste documento está relacionado com a indústria automóvel, em específico com o desenvolvimento de um *bus sniffer* para ser implementado numa rede de sistemas eletrónicos, conhecidos como *Electronic Control Units (ECU)*, que estão presentes nos automóveis e que são responsáveis por controlar um ou mais dos seus sistemas.

Nos últimos anos as redes CAN (*Controller Area Network*) e Ethernet têm vindo a afirmar-se como sendo duas tecnologias bastante dominantes nesta área quer pela capacidade de suportar velocidades de transferência de dados bastante elevadas no caso da Ethernet quer pela fiabilidade e facilidade de implementação do CAN.

No entanto a dificuldade e perigo de testar estes sistemas em ambiente real origina a necessidade de ter estações de teste virtuais que simulem o comportamento de tais ECUs em ambiente automóvel. Assim, para testar, avaliar, e corrigir o funcionamento de ECUs em tais ambientes, as estações de teste necessitam de acesso à informação que corre nas redes dos automóveis e é desta necessidade que surgem os *bus sniffers*.

O avanço rápido de tecnologias como Ethernet faz com que no mercado estas ferramentas de teste ainda sejam muito caras e que tenham um grande impacto para uma empresa cuja atividade se enquadre e dependa delas. Como tal criar uma plataforma interna à empresa traria uma redução drástica de custos e também permitiria ajustá-la ao seu uso e necessidades.

Esta dissertação foca-se no desenvolvimento de tal plataforma que começa pelo hardware responsável por interceptar a informação das redes automóveis - o *bus sniffer*. Para tal foram estudadas várias opções de *Hardware* no mercado e a mais relevante acabou por se basear num *Raspberry Pi 4* com um *shield* de CAN para a qual foi implementada uma estrutura de *software* que se baseia em 3 *threads*, duas para receber a informação via CAN e Ethernet e outra para enviar a informação para um *Desktop* por Wi-Fi.

Para esta solução foi avaliada a precisão dos *timestamps* através da métrica de desvio temporal da diferença de tempo entre pacotes recebidos consecutivamente em relação à diferença de tempo dos mesmos pacotes quando enviados. As variáveis independentes usadas nos testes foram o volume de informação em cada pacote (número de *bytes*) e o tempo entre envio de cada pacote e para tal foi usada uma conhecida plataforma de testes na indústria automóvel da empresa Vector. O *software* desta plataforma é o *CANoe* que neste caso foi usado com o *hardware* VN5610A por suportar *automotive* Ethernet e CAN. Os testes realizados mostram resultados bastante positivos de desvios médios nas ordens das dezenas de micro segundos e comprovam que o *Raspberry Pi 4* com *shield* para CAN tem capacidade para ser utilizado neste ambiente e que o desempenho pode ser facilmente aumentado sem qualquer custo monetário relevante seguindo as recomendações deixadas no final da dissertação.





# Abstract

This document addresses a topic within the automotive industry, focusing on the development of a bus sniffer to be implemented in a car's network of electronic control units (ECUs) which are responsible for controlling one or more of its systems.

For the last couple of years CAN (Controller Area Network) and Ethernet have been becoming the two most dominant automotive networks. Ethernet because it offers the possibility to satisfy the increasingly higher bandwidth requirements and CAN because of its reliability and ease of implementation.

The danger and struggle to test these systems in a real environment are reasons why having virtual test stations which simulate the behaviour of these ECUs from cars is a must in this industry. To test, evaluate and fix these systems such test stations need access to the information that runs on the automotive network (Ethernet and CAN buses) which is the job of bus sniffers.

The fast development of technologies such as Ethernet in the automotive industry creates an opening in the market that allows tools that support such technologies to be sold at an excessively high price. Because companies that develop ECUs are highly dependent on such tools, replacing those with ones created by the firm would not only drastically reduce its costs but would also allow to adjust it to the company use and needs.

This dissertation focuses on the development of such platform which starts by the Hardware responsible for intercepting the information present on those automotive networks - the bus sniffer. In order to build this tool a couple of solutions were studied with the Raspberry Pi 4 with a CAN breakout board being the fittest one. A software based on 3 threads, 2 to intercept the information on CAN and Ethernet and the other to send the packets to a target computer by Wi-fi was also developed in order to support the Raspberry + CAN board hardware.

This solution was tested doing both volume tests and time stress tests. To evaluate the accuracy of the timestamps it was measured the time deviation between the time difference of packets consecutively received and the same difference from those packets when sent. Those packets were sent using a recognized test platform which is widely accepted in the automotive industry and that is owned by Vector. The software of this platform is the CANoe which was used in combination with the Hardware VN5610A because it supports both automotive Ethernet and CAN. The tests which were made show very positive results with average deviations in the order of tens of micro seconds and which prove that the Raspberry Pi 4 with the CAN shield can easily be used in this environment and its performance can be improved without any cost if the recommendations from the end of this document are followed.



# Acknowledgements

I want to express my gratitude towards Professor Paulo Portugal for completely changing the idea that I had of a Supervisor in the most positive way. Every single time that I needed, Professor Paulo Portugal was available and provided very relevant and fundamental opinions to guide me during my work, in particular during the pandemic situation.

I also want to thank my Co-Supervisor, Eng<sup>o</sup> Tiago Rodrigues for the ongoing availability to answer my questions and doubts and being both very critical when needed and very rewarding when deserved. Also to Eng<sup>a</sup> Sandra Costa, Eng<sup>a</sup> Jana Seidel and to the team I was part of during this period my thanks for embracing me and making the whole time in Bosch more pleasurable.

And last but definitely not least, a huge thank to my girlfriend, Filipa Raimundo for putting up with me during every day of this dissertation, both on good and bad days. To my family for supporting me during during my whole education, and particularly this course and to my closest friends for creating with me some of the best moments and memories that I will ever have.

João Santos



*Every mistake  
is a lesson*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Motivation . . . . .	2
1.3	Goals . . . . .	2
1.4	Structure . . . . .	3
<b>2</b>	<b>State of the art</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	CAN . . . . .	6
2.2.1	CAN protocol . . . . .	6
2.2.2	Standard CAN or Extended CAN . . . . .	7
2.2.3	A CAN Message . . . . .	7
2.2.4	CAN-FD . . . . .	12
2.3	Ethernet . . . . .	16
2.3.1	Ethernet protocol . . . . .	16
2.3.2	Automotive Ethernet . . . . .	19
2.3.3	100/1000 BASE-T1 Ethernet . . . . .	20
2.4	Available Sniffer tools . . . . .	24
2.4.1	VN5610A and VN5610 . . . . .	25
2.4.2	Universal EMC Device . . . . .	26
2.5	Summary and main conclusions . . . . .	27
<b>3</b>	<b>Proposed architecture</b>	<b>29</b>
3.1	Requirements analysis . . . . .	29
3.2	Previous work . . . . .	30
3.3	Hardware architecture . . . . .	32
3.3.1	Requirements fulfilment . . . . .	33
3.3.2	Bus sniffer hardware setup . . . . .	35
3.4	Bus sniffer software implementation . . . . .	35
3.4.1	Bus sniffer input parameters . . . . .	36
3.4.2	Bus sniffer implementation design . . . . .	37
3.4.3	Bus sniffer output . . . . .	42
3.5	Summary . . . . .	43
<b>4</b>	<b>Tests and requirements evaluation</b>	<b>45</b>
4.1	Evaluation Conditions . . . . .	45
4.2	Results . . . . .	46
4.2.1	Typical use . . . . .	48

4.2.2	Time stress test . . . . .	49
4.2.3	Packet size stress test . . . . .	52
4.3	Performance Evaluation . . . . .	54
<b>5</b>	<b>Conclusions and Future work</b>	<b>55</b>
5.1	Conclusions . . . . .	55
5.2	Future work . . . . .	56
	<b>References</b>	<b>57</b>



# List of Figures

2.1	CAN protocol stack, adapted from [3]	6
2.2	Standard CAN: 11-Bit Identifier [3]	7
2.3	Extended CAN: 29-Bit Identifier [3]	8
2.4	The Inverted Logic of a CAN Bus [3]	9
2.5	Arbitration on a CAN Bus. Squares are dominant bits and straight lines are recessive. [3]	10
2.6	CAN-FD transmission phases [5]	12
2.7	Size comparison between CAN and CAN-FD frames [5]	13
2.8	Frame formats comparison between Classical CAN, Extended CAN and CAN-FD, adapted from [3]	14
2.9	Two CAN FD frame formats: The IDE bit is recessive in FFFF, the RRS bit is always dominant, and the value of the SRR bit doesn't matter [5]	15
2.10	Ethernet protocol stack [8]	16
2.11	The original Ethernet implementation: shared medium, collision-prone. All computers trying to communicate share the same cable, and so compete with each other VS. Modern Ethernet implementation: switched connection, collision-free. Each computer communicates directly only with its own switch, without competition for the cable with others.	17
2.12	An Ethernet packet [10]	18
2.13	Data conversion from MII to MDI [25]	23
2.14	VN5610A/VN5640 with 100 BASE-T1 option [29]	25
2.15	Universal EMC device [30]	26
3.1	Diagram block representation of OM14510-SJA1105TJP board [32]	30
3.2	OM14510-SJA1105 with MCP2517FD CAN breakout setup. Adapted from [32]. (1) Automotive Ethernet links between ECUs and OM14510-SJA1105 board and OM14510-SJA1105 and media converter. (2) CAN link between ECU and a CAN breakout board (MCP2561FD) (3) SPI connection between the CAN breakout board and the OM14510-SJA1105 board (the diagram of this connection is merely representative to ease readability. The real connection is made with the extension connector located on the right side of the board) (4) USB-B connection from the computer to the OM14510-SJA1105 board (only needed to flash code) (5) Ethernet link between media converter and the computer	31
3.3	Location of connectors and main ICs on Raspberry Pi 4 [34]	33

3.4	Raspberry simplified block diagram (Raspberry Pi 4 on the left versus previous versions on the right) . . . . .	33
3.5	PiCAN2 shield for CAN interface mounted on top of a Raspberry . . . . .	34
3.6	Packets software time stamping in linux . . . . .	34
3.7	Raspberry Pi 4 + PiCAN bus sniffer hardware block diagram. (1) Ethernet 1000/100/10BASE-T1 link between ECUs Ethernet network and the Media Converter (2) Ethernet 1000/100/10BASE-T link between the Media Converter and the Raspberry Pi 4 (3) CAN link between ECUs CAN network and the PiCAN2 breakout board (4) SPI link between the PiCAN 2 breakout board and the Raspberry Pi 4 (In reality the breakout board is mounted on top of the Raspberry Pi 4. The space in between is on the picture just to ease visibility) (5) Wi-fi connection between the Raspberry Pi 4 and the Desktop . . . . .	35
3.8	Command line error on input parameters . . . . .	36
3.9	Software design of the <i>CanEthAnalyz</i> (Dummy code pieces present in the diagram are just for representation) . . . . .	38
3.10	Graphical illustration of a raw socket [40] . . . . .	40
3.11	Command line output example from <i>CanEthAnalyz</i> . . . . .	43
4.1	Hardware block diagram of Raspberry Pi 4 + PiCAN bus sniffer in test environment using VN5610A to generate traffic and 1000Base-T1 Media Converter by <i>technica engineering</i> to convert between Base-T1 to Base-T. . . . .	45
4.2	CANoe packet generation setup . . . . .	46
4.3	CANoe traces - Ethernet trace on the left and CAN trace on the right (on the CAN trace the message and timestamp is updated on the same place) . . . . .	48
4.4	Wireshark log capture from the packets that are sent by the Raspberry Pi 4 . . . . .	49
4.5	CAN results for 50ms . . . . .	50
4.6	CAN results for 10ms . . . . .	50
4.7	CAN results for 5ms . . . . .	50
4.8	CAN results for 1ms . . . . .	50
4.9	Ethernet results for 50ms . . . . .	51
4.10	Ethernet results for 10ms . . . . .	51
4.11	Ethernet results for 5ms . . . . .	52
4.12	Ethernet results for 1ms . . . . .	52
4.13	Results for 100 bytes on each packet . . . . .	53
4.14	Results for 200 bytes on each packet . . . . .	53
4.15	Results for 400 bytes on each packet . . . . .	53
4.16	Results for 800 bytes on each packet . . . . .	53
4.17	Results for 1500 bytes on each packet . . . . .	54

# List of Tables

2.1	In-vehicle domains and the level of criticalities [18] . . . . .	20
2.2	100BASE-T1 3 bit to 2 ternary mapping [26] . . . . .	23





# Acronyms and Abbreviations

AUI	Attachment Unit Interface
AVB	Audio Video Bridging
BRS	Bit Rate Switch
CAN	Controller Area Network
CAN-FD	Controller Area Network Flexible Data-rate
CD+AMP	Collision Detection and arbitration on message priority
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DEI	Drop Eligible Indicator
DLC	Data Length Code
ECU	Electronic Control Unit
EDL	Extended Data Length
EOF	End Of Frame
ESI	Error State Indicator
FBFF	Flexible Data-rate base frame format
FCS	Frame check sequence
FDF	Flexible Data-rate format
FEFF	Flexible Data-rate extended frame format
FSB	Fixed Stuff Bits
GPS	Global Positioning System
IDE	Identifier Dominant Extension
IFS	Inter Frame Space
ISO	International Standardization Organization
LIN	Local Interconnect Network
LSB	Least Significant Bit
MAC	Media Access Control
MOST	Media Oriented Systems Transport
OBD	On Board Diagnostic
OSI	Open Systems Interconnection
PCP	Priority Code Point
PDU	Protocol Data Unit
PHY	Physical Layer transceiver circuitry
RTR	Remote Transmission Request
SAE	Society of Automotive Engineers
SFD	Start Frame Delimiter
SOF	Start Of Frame
SRR	Substitute Remote Request
TCP	Transmission Control Protocol
TDC	Transceiver Delay Compensation
UDP	User Datagram Protocol
VID	VLAN Identifier

# Chapter 1

## Introduction

### 1.1 Context

In its early days, cars were simply seats on top of wheels mounted to an engine, which provided mobility. With time these vehicles started having roofs, comfortable seats and more power to run faster. Eventually multiple kinds of signal lights, information display and ignition switches were added and electronics proved its significance.

Nowadays in the automotive industry, as new technologies develop and different system architectures for cars are studied and implemented, the need for communication between these systems increases. Also, there is a tendency to introduce increasingly more infotainment devices: multimedia systems and different devices with informative purpose, as well as systems to allow communication with external networks, all of them having a more complex graphical user interface (i.e. Global Positioning System). All these devices produce an increasingly larger volume of information which requires relatively high data rates to send it to the control units which will process the information. For this purpose there are a couple of network solutions that are used, the main one still being the Controller Area Network bus (CAN) which is safe and reliable but does not support a communication rate greater than 5 Mb/second, as it was not intended for high-traffic in a peer-to-peer network. For this reason and also because modern cars integrate an increasingly higher number of functionalities with high bandwidth, real-time, and reliability requirements, Ethernet is being used as the solution for these cases since it offers the possibility to satisfy these bandwidth requirements and enables the usage of temporal redundancy mechanisms to increase the reliability of the communication network.

This increase of complexity and information in these networks allied with the critical time constrains that this area has, made it crucial for engineers to have a very precise and reliable way to check, test, debug and validate their designs throughout the design cycles of a hardware-based product as well as helping in later phases of a product life cycle, in examining communication interoperability between systems and between components, and clarifying hardware support concerns.

When used together, the combined strengths of CAN and Ethernet can create a very powerful

and robust network which is the ideal use case for the automotive industry. As such a bus analyzer that supports both technologies and accommodates all the aspects mentioned in the last paragraph will have great value as these two networks will thrive in the future of the Automotive networks.

## 1.2 Motivation

With the introduction of electronic technologies in vehicles, the intercommunication of the devices proved to be of much importance but also started to be a lot to handle. This made it difficult for developers to understand how new technologies behave in the car environment and if the information that is being sent and received by each ECU (Electronic Control Unit) is correct.

For this reason automotive network analyzers were created. Its ambition is to aid on the task of monitoring the automotive network inside a car by intercepting the packets that are transported on each network and hasten the identification of problems therefore helping the design, test and validation of the whole system. Since CAN is a technology that has been widely used for the last decades in the automotive industry there are many tools to analyze it. On the other hand Automotive Ethernet has been growing and improving only in the past years and it is being commonly adopted in automotive networks it is crucial to develop new tools that provide support for this technology. Having the possibility to analyze these two automotive networks will allow projects to be developed faster and more easily by facilitating the task of debugging software that is recently added to the system.

## 1.3 Goals

The main goal of this dissertation is to develop a CAN and Ethernet bus sniffer to be used in automotive networks. This sniffer shall be easy to use and adapted to the company needs while remaining low cost. The application should intercept the data from the Ethernet and CAN buses, timestamp it and send to another device.

The specific goals requested by Bosch are:

- Develop a proof of concept for a bus sniffer to intercept CAN and Ethernet networks. This includes choosing hardware and developing firmware;
- Allow the intercepted information to be used in a software to be developed by Bosch.
- Validate and evaluate the system using real ECUs (Electronic Control Unit) or a testing environment provided by Bosch

Finally, there are a few expensive tools like this available in the market but this project aims to substitute those while being cheap, scalable and owned by the company therefore easing tool support and enabling further development with newer technologies.



## 1.4 Structure

The rest of this document is structured as follows:

- [Chapter 2](#) [State of the art] - Presentation of fundamental concepts of the two automotive networks that will be used as well as an overview of the existing solutions for network sniffers in the market.
- [Chapter 3](#) [Proposed architecture] - Presentation of a platform that was used previously and description in detail of the Hardware and Software architecture proposed and implemented in this dissertation.
- [Chapter 4](#) [Tests and requirements evaluation] - Validation and evaluation of the proposed solution in terms of accuracy of timestamps and reliability of the system under two different network metrics and presentation and discussion of the results.
- [Chapter 5](#) [Conclusions and Future work] - Conclusion of the dissertation, including the work done, main results achieved and future work.



# Chapter 2

## State of the art

### 2.1 Introduction

This chapter covers topics that are considered to be essential to understand the technical aspects laid out in [Chapter 1](#) and which will be the foundations for the solution that was developed during this Dissertation.

It is divided into the following sections:

- [CAN](#) - An explanation about the most important aspects of the CAN and CAN-FD (CAN flexible Data-rate) protocols;
- [Ethernet](#) - A summary of one of the most used technologies in networks and why and how it is being adopted in the Automotive industry;
- [Available Sniffer Tools](#) - Analysis of the currently available test benches on the market and why another solution is needed;

## 2.2 CAN

The history of Controller Area Network (CAN) starts more than 30 years ago. At the beginning of the 1980s a group of engineers at Bosch GmbH were looking for a serial bus system suitable for use in passenger cars. The most popular solutions adopted at that time were considered inadequate for the needs of most automotive applications. The bus system, in fact, had to provide a number of new features that could hardly be found in the already existing fieldbus architectures. The new communication protocol was presented officially in 1986 with the name of Automotive Serial Controller Area Network at the Society of Automotive Engineers (SAE) congress held in Detroit. It was based on a multi master access scheme to the shared medium that resembled the well-known carrier sense multiple-access (CSMA) approach. The peculiar aspect, however, was that CAN adopted a new distributed nondestructive arbitration mechanism to solve contentions on the bus by means of priorities implicitly assigned to the colliding messages. Moreover, the protocol specifications also included a number of error detection and management mechanisms to enhance the fault tolerance of the whole system. [1]

### 2.2.1 CAN protocol

CAN is a serial communications bus standard defined by the International Standardization Organization (ISO) [2] originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. The specification calls for high immunity to electrical interference and the ability to self-diagnose and repair data errors. These features have led to CAN's popularity in a variety of industries including building automation, medical, and manufacturing.

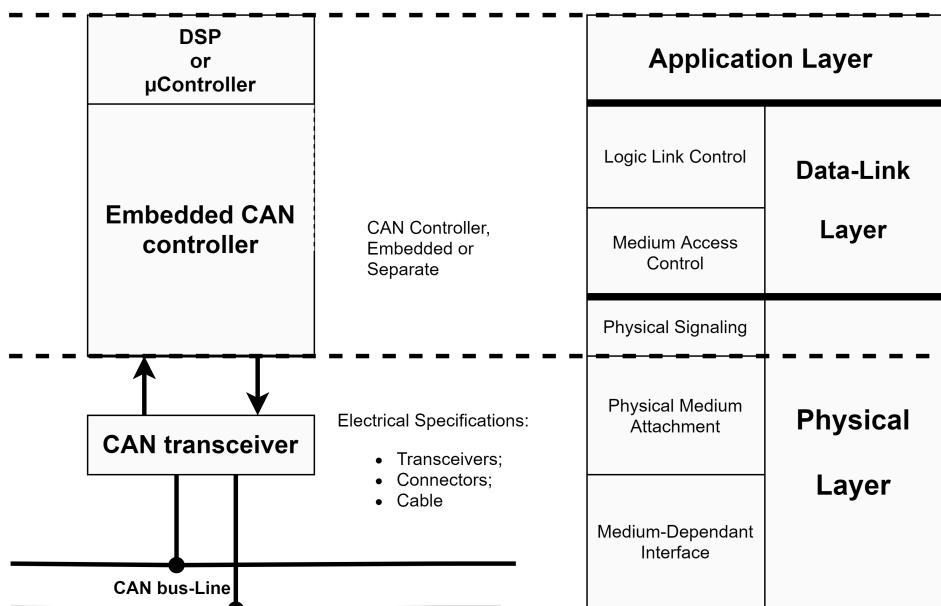


Figure 2.1: CAN protocol stack, adapted from [3]

The CAN communications protocol, ISO-11898: 2003, describes how information is passed between devices on a network and conforms to the Open Systems Interconnection (OSI) model that is defined in terms of layers. As depicted in [fig. 2.1](#), the ISO 11898 architecture defines the lowest two layers of the seven layer OSI/ISO model: the data-link layer and physical layer.

### 2.2.2 Standard CAN or Extended CAN

The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP). CSMA (Carrier Sense Multiple Access) means that each node on a bus must wait for a prescribed period of inactivity before attempting to send a message. CD+AMP (Collision Detection and arbitration on message priority) means that collisions are resolved through a bit-wise arbitration, based on the priority of each message which is contained in the identifier field of a message. The higher priority identifier always wins bus access. That is, if two messages begin transmitting simultaneously, the one with the lowest ID (highest priority) wins and the CAN controller with the higher ID will back off and retry once the bus is available. Since every node on a bus takes part in writing every bit "as it is being written," an arbitrating node knows if it placed the logic-high bit on the bus. The ISO-11898:2003 Standard, with the standard 11-bit identifier, provides signaling rates from 125 kbps to 1 Mbps. The standard was later amended with the "extended" 29-bit identifier. The standard 11-bit identifier field in [Figure fig. 2.2](#) provides for  $2^{11}$ , or 2048 different message identifiers, whereas the extended 29-bit identifier in [Figure fig. 2.3](#) provides for  $2^{29}$ , or 537 million identifiers.

### 2.2.3 A CAN Message

#### 2.2.3.1 CAN frame

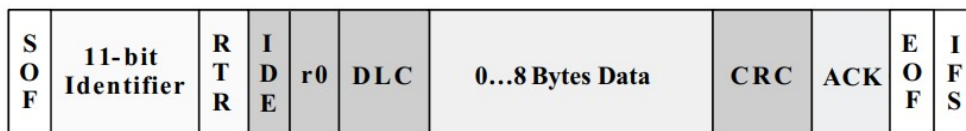


Figure 2.2: Standard CAN: 11-Bit Identifier [3]

The meaning of the bit fields in a CAN frame ([fig. 2.2](#)) are:

- SOF – The single dominant start of frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.
- Identifier-The Standard CAN 11-bit identifier establishes the priority of the message. The lower the binary value, the higher its priority.
- RTR – The single remote transmission request (RTR) bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the specified node. The responding data is also received by all nodes and used by any node interested. In this way, all data being used in a system is uniform.

- IDE – A dominant single identifier extension (IDE) bit means that a standard CAN identifier with no extension is being transmitted. A recessive bit indicates that more identifier bits follow. The 18-bit extension follows IDE making a total of 29 bits for the identifier.
- r0 – Reserved bit (for possible use by future standard amendment).
- DLC – The 4-bit data length code (DLC) contains the number of bytes of data being transmitted.
- Data – 8/16/24/32/48/64 bits of application data may be transmitted
- CRC – CAN frames contain a safeguard based on a CRC polynomial: The transmitter calculates a check sum from the transmitted bits and provides the result within the frame in the CRC field. The receivers use the same polynomial to calculate the check sum from the bits as seen on the bus-lines. Then self-calculated check sum is compared with the received one to see if it is correct or not.
- ACK – Every node receiving a correct message overwrites this recessive bit in the original message with a dominant bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the sending node repeats the message after re arbitration. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter.
- EOF – This end-of-frame (EOF), 7-bit field marks the end of a CAN frame (message) and disables bit stuffing, indicating a stuffing error when dominant. When 5 bits of the same logic level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data.
- IFS – This 7-bit inter frame space (IFS) contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area.

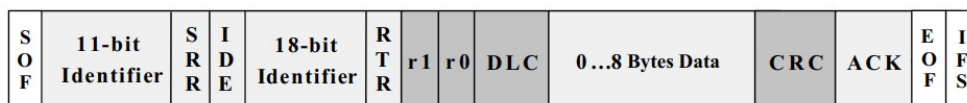


Figure 2.3: Extended CAN: 29-Bit Identifier [3]

As shown in [fig. 2.3](#), the Extended CAN message has a 29 bit identifier, and is the same as the Standard message with the addition of:

- SRR – The substitute remote request (SRR) bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- IDE – A recessive bit in the identifier extension (IDE) indicates that more identifier bits follow. The 18-bit extension follows IDE.

- r1 – Following the RTR and r0 bits, an additional reserve bit has been included ahead of the DLC bit.

### 2.2.3.2 Arbitration

A fundamental CAN characteristic shown in [fig. 2.4](#) is the opposite logic state between the bus, and the driver input and receiver output. Normally, a logic-high is associated with a one, and a logic-low is associated with a zero - but not so on a CAN bus. This is why CAN transceivers have the driver input and receiver output pins passively pulled high internally, so that in the absence of any input, the device automatically defaults to a recessive bus state on all input and output pins.[\[3\]](#) Also, CAN receivers measure differential voltage on the bus to determine the bus level. Since 3.3V transceivers generate the same differential voltage ( $\geq 1.5V$ ) as 5V transceivers, all transceivers on the bus (regardless of supply voltage) can decipher the message. In fact, the other transceivers can't even tell there is anything different about the differential voltage levels. [\[4\]](#)

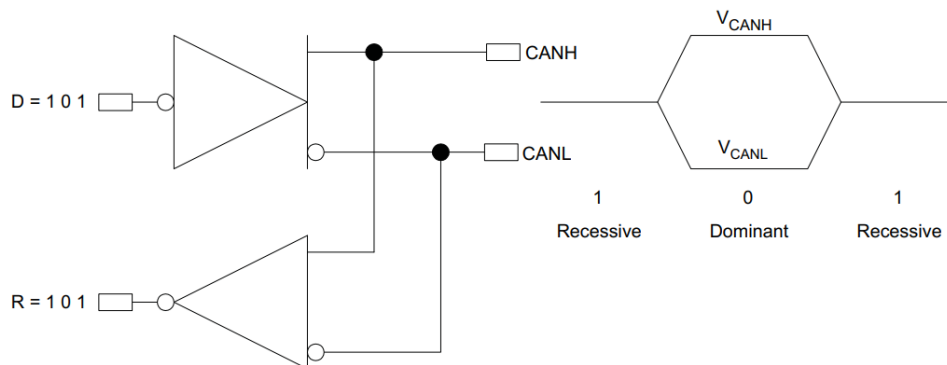


Figure 2.4: The Inverted Logic of a CAN Bus [\[3\]](#)

Bus access is event-driven and takes place randomly. If two nodes try to occupy the bus simultaneously, access is implemented with a nondestructive, bit-wise arbitration. Nondestructive means that the node winning arbitration just continues on with the message, without the message being destroyed or corrupted by another node. In terms of logic, such behavior is an WIRED-AND. Physically, AND-logic can be implemented by a so-called open collector circuit.

[Figure 2.5](#) displays the CAN arbitration process that is handled automatically by a CAN controller. Because each node continuously monitors its own transmissions, as node B's recessive bit is overwritten by node C's higher priority dominant bit, B detects that the bus state does not match the bit that it transmitted. Therefore, node B halts transmission while node C continues on with its message. Another attempt to transmit the message is made by node B once the bus is released by node C. This functionality is part of the ISO 11898 physical signaling layer, which means that it is contained entirely within the CAN controller and is completely transparent to a CAN user.

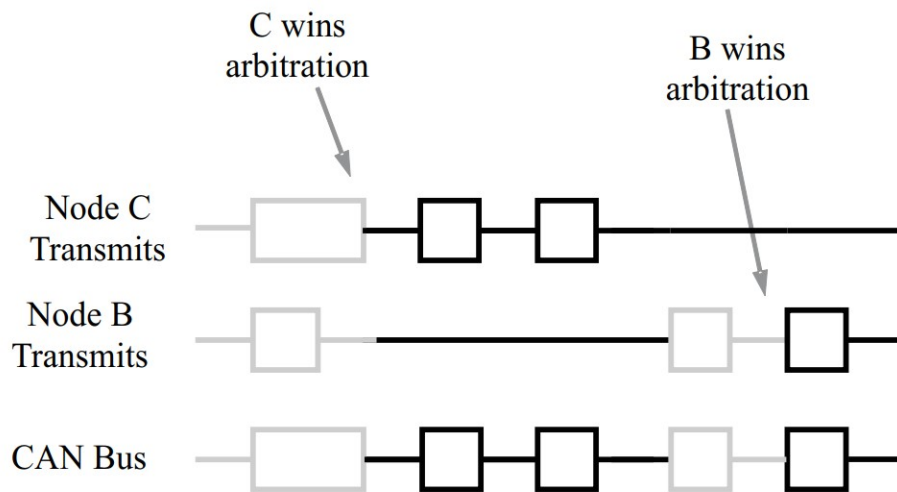


Figure 2.5: Arbitration on a CAN Bus. Squares are dominant bits and straight lines are recessive. [3]

The allocation of message priority is up to a system designer, but industry groups mutually agree on the significance of certain messages. For example, a manufacturer of motor drives may specify that message 0010 is a winding current feedback signal from a motor on a CAN network and that 0011 is the tachometer speed. Because 0010 has the lowest binary identifier, messages relating to current values always have a higher priority on the bus than those concerned with tachometer readings.

The allocation of priority to messages in the identifier is a feature of CAN that makes it particularly attractive for use within a real-time control environment. The lower the binary message identifier number, the higher its priority. An identifier consisting entirely of zeros is the highest priority message on a network because it holds the bus dominant the longest. Therefore, if two nodes begin to transmit simultaneously, the node that sends a last identifier bit as a zero (dominant) while the other nodes send a one (recessive) retains control of the CAN bus and goes on to complete its message. A dominant bit always overwrites a recessive bit on a CAN bus. Note that a transmitting node constantly monitors each bit of its own transmission. This is the reason for the transceiver configuration of Figure 2.4 in which the CANH and CANL output pins of the driver are internally tied to the receiver's input. The propagation delay of a signal in the internal loop from the driver input to the receiver output is typically used as a qualitative measure of a CAN transceiver.



### 2.2.3.3 Message Types

The four different message types, or frames (see [Figure 2.2](#) and [Figure 2.3](#)), that can be transmitted on a CAN bus are the data frame, the remote frame, the error frame, and the overload frame.

The **data frame** is the most common message type, and comprises the Arbitration Field, the Data Field, the CRC Field, and the Acknowledgment Field. The Arbitration Field contains an 11-bit identifier in [Figure 2.2](#) and the RTR bit, which is dominant for data frames. In [Figure 2.3](#), it contains the 29-bit identifier and the RTR bit. Next is the Data Field which contains zero to eight bytes of data, and the CRC Field which contains the 16-bit checksum used for error detection. Last is the Acknowledgment Field.

The intended purpose of the **remote frame** is to solicit the transmission of data from another node. The remote frame is similar to the data frame, with two important differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data.

The **error frame** is a special message that violates the formatting rules of a CAN message. It is transmitted when a node detects an error in a message, and causes all other nodes in the network to send an error frame as well. The original transmitter then automatically retransmits the message. An elaborate system of error counters in the CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frames.

The **overload frame** is mentioned for completeness. It is similar to the error frame with regard to the format, and it is transmitted by a node that becomes too busy. It is primarily used to provide for an extra delay between messages.

A frame is considered to be valid when the last bit of the ending EOF field of a message is received in the error-free recessive state. A dominant bit in the EOF field causes the transmitter to repeat a transmission.

### 2.2.3.4 Error Checking and Fault Confinement

The robustness of CAN may be attributed in part to its abundant error-checking procedures. The CAN protocol incorporates five methods of error checking: three at the message level and two at the bit level. If a message fails any one of these error detection methods, it is not accepted and an error frame is generated from the receiving node. This forces the transmitting node to resend the message until it is received correctly. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by its controller after an error limit is reached.

Error checking at the message level is enforced by the CRC and the ACK slots displayed in [Figure 2.2](#) and [Figure 2.3](#). The 16-bit CRC contains the checksum of the preceding transmitted bits for error detection with a 15-bit checksum and 1-bit delimiter. The ACK field is two bits long and consists of the acknowledge bit and an acknowledge delimiter bit.

Also at the message level is a form check. This check looks for fields in the message which must always be recessive bits. If a dominant bit is detected, an error is generated. The bits checked are the SOF, EOF, ACK delimiter, and the CRC delimiter bits

At the bit level, each bit transmitted is monitored by the transmitter of the message. If a data bit (not arbitration bit) is written onto the bus and its opposite is read, an error is generated. The only exceptions to this are with the message identifier field which is used for arbitration, and the acknowledge slot which requires a recessive bit to be overwritten by a dominant bit.

The final method of error detection is the bit-stuffing rule. As explained in the ISO 11898-1, to secure synchronization senders must transmit a complementary bit at the latest after transmitting five homogeneous bits even if a complementary bit followed the five homogeneous bits. If that doesn't happen an error is generated. Stuffing ensures that rising edges are available for on-going synchronization of the network and that a stream of bits are not mistaken for an error frame, or the seven-bit interframe space that signifies the end of a message. Stuffed bits are removed by a receiving node's controller before the data is forwarded to the application.

With this logic, an active error frame consists of six dominant bits—violating the bit stuffing rule. This is interpreted as an error by all of the CAN nodes which then generate their own error frame. This means that an error frame can be from the original six bits to twelve bits long with all the replies. This error frame is then followed by a delimiter field of eight recessive bits and a bus idle period before the corrupted message is retransmitted. It is important to note that the retransmitted message still has to contend for arbitration on the bus. [3]

## 2.2.4 CAN-FD

With the constant improvement of car technology in the automotive industry and the constantly increase of ECUs in a car, the needed bandwidth increased to values over 1Mbps. This meant that a new protocol supporting higher bit rates than CAN was necessary and this was when CAN-FD (flexible data-rate) emerged.

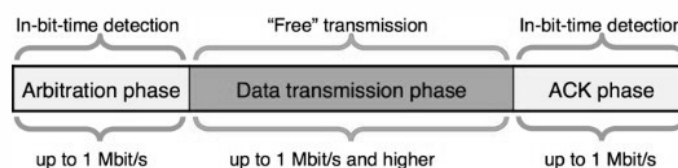


Figure 2.6: CAN-FD transmission phases [5]

In 2011, Bosch started the CAN FD development in close cooperation with car makers and other CAN experts. The improved protocol overcomes CAN limits: There is the possibility to transmit data faster than with 1 Mbit/s and the payload (data field) is now up to 64 bytes long and not limited to 8 bytes anymore. The primary difference between the classical CAN and CAN-FD is the Flexible Data (FD). Using CAN-FD, ECUs can dynamically switch to different data-rate and with larger or smaller message sizes. Enhanced features in CAN-FD includes the capability to dynamically select and switch to faster or slower data rate, as and when required, and to pack more data within the same CAN frame and transport it over the CAN bus in less time.

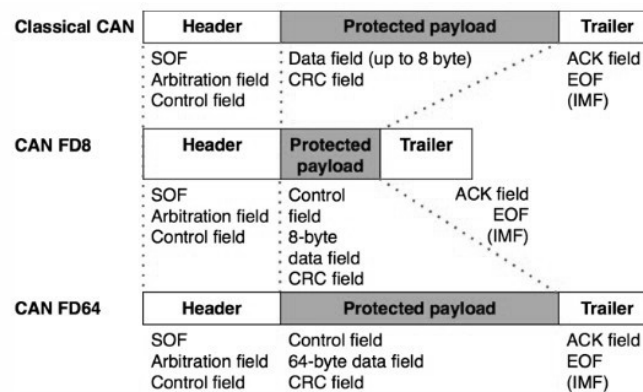


Figure 2.7: Size comparison between CAN and CAN-FD frames [5]

As depicted in [Figure 2.7](#), larger payloads improve the protocol efficiency and lead to a higher throughput.

#### 2.2.4.1 CAN-FD protocol

The Control Field in classical CAN frames contains reserved bits, which are specified to be transmitted dominantly. In the CAN-FD frame, the reserved bit r0 after the IDE bit (Classical CAN) or after the RTR bit (r1 - Classical extended CAN) is defined as Extended Data Length (EDL) bit and is transmitted recessively. The following bits are new in CAN-FD compared with CAN:

- EDL - Extended Data Length
- r1, r0 reserved (transmitted dominantly)
- BRS - Bit Rate Switch
- ESI - Error State Indicator

The DLC values from 0000b to 1000b still code a Data Field length from 0 to 8 byte, while the DLC values from 1001b to 1111b are defined in CAN-FD to code Data Fields with a length of 12, 16, 20, 24, 32, 48 and 64 byte respectively. The EDL bit distinguishes between the normal CAN frame format and the CAN-FD frame format. The value of the BRS bit decides, whether

the bit-rate in the Data-Phase is the same as in the Arbitration-Phase (BRS dominant) or whether the predefined faster bit rate is used in the Data-Phase (BRS recessive). In CAN-FD frames, the EDL bit is always recessive and followed by the dominant r0 bit. This provides an edge for resynchronization before an optional bit-rate switch. The edge is also used to measure the transceiver’s loop delay for the optional TDC (Transceiver Delay Compensation). In CAN-FD frames, the transmitter’s error state is indicated by ESI, dominant for error active and recessive for error passive. This simplifies network management. There are no CAN-FD remote frames, the bit at the position of the RTR bit in normal CAN frames is replaced by the dominant r1 bit. However, normal CAN remote frames may optionally be used in CAN-FD systems. Receivers ignore the actual values of the bits r1 and r0 in CAN-FD frames. [6]

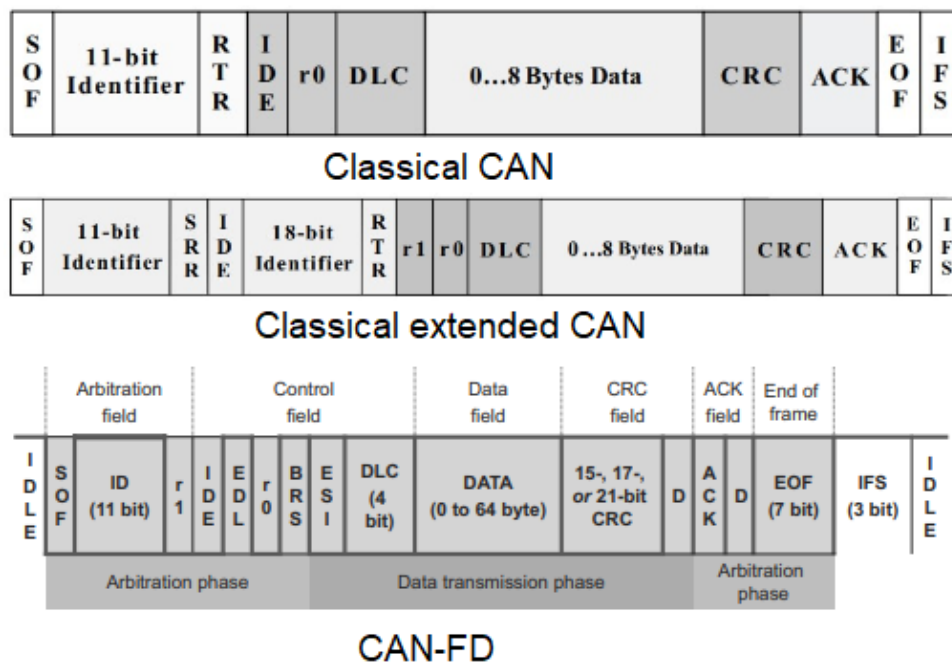


Figure 2.8: Frame formats comparison between Classical CAN, Extended CAN and CAN-FD, adapted from [3]

The CAN-FD protocol controller has to also support Classical CAN frames. Both CAN protocols (Classical as well as CAN-FD) are internationally standardized in ISO 11898-1:2015 [7]. CAN-FD data frames with 11-bit identifiers use the FBFF (FD base frame format) and those with 29-bit identifiers use the FEF (FD extended frame format). The CAN FD protocol doesn’t support remotely requested data frames.

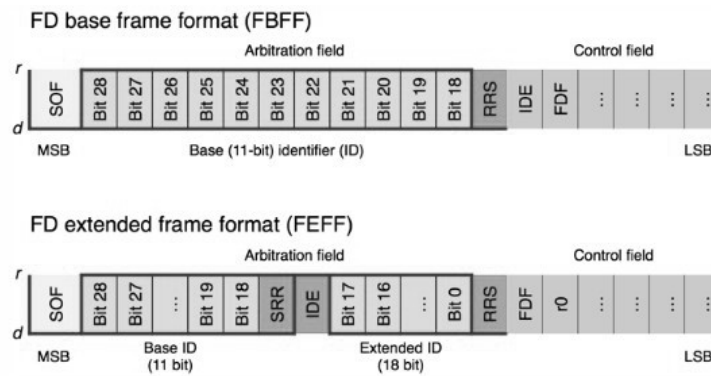


Figure 2.9: Two CAN FD frame formats: The IDE bit is recessive in FEFF, the RRS bit is always dominant, and the value of the SRR bit doesn't matter [5]

The control field comprises additional bits not provided by the Classical CAN data frames. The FDF (FD format) bit indicates the usage of FD frame formats. At the sample-point of the BRS (bit-rate switch) bit, the bit-rate switch is performed. This guarantees a maximum of robustness. The following ESI bit provides information about the error status: a dominant value indicates an error active state.

During the standardization process of the CAN FD protocol, some additional safe guards were introduced in order to improve the communication reliability. This is why the CRC field comprises 17-bit (for frames with payloads up to 16 byte) or 21-bit (for frames larger than 16 byte) polynomials and an 8-bit stuff-bit counter, plus a parity bit. The CRC field uses fixed-stuff bits (FSB) with an opposite value of the previous bit. All these safe guards guarantee that all single failures are detected under all conditions. Even the possibility to detect multiple failures has been improved. [5]

## 2.3 Ethernet

Ethernet is a technology that connects wired local area networks (LANs), metropolitan area networks (MAN) and wide area networks (WAN) and enables locally distributed computing stations to communicate with each other. It also describes how network devices can format and transmit data so other devices can recognize, receive and process the information.

Ethernet has been around in various forms since the early 1970s. The current incarnation of Ethernet is defined by the IEEE standard known as 802.3. [8]

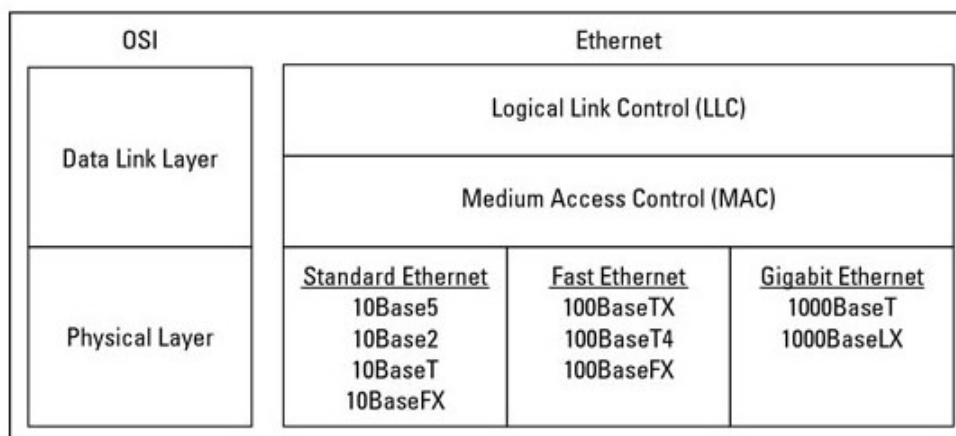


Figure 2.10: Ethernet protocol stack [8]

Like CAN, Ethernet operates at the first two layers of the OSI model — the Physical and the Data Link layers and for the information to be successively sent and received by an Ethernet cable it needs a protocol for communication between two devices as well as an interface between the Data Link layer and the physical layer. Analogously to CAN, Ethernet has its own structure and for the interface it also relies on the MAC (Media Access Control) sub-layer. This sub-layer has two main functions: data encapsulation by doing framing, addressing and error detection and media access management by doing medium allocation (collision avoidance) and contention resolution (collision handling).

### 2.3.1 Ethernet protocol

Ethernet was originally based on the idea of computers communicating over a shared coaxial cable acting as a broadcast transmission medium.

Original Ethernet's shared coaxial cable (the shared medium) traversed a building or campus to every attached machine. Carrier sense multiple access with collision detection (CSMA/CD) governed the way the computers shared the channel. This scheme was simpler than competing Token Ring or Token Bus technologies. Computers are connected to an Attachment Unit Interface (AUI) transceiver, which is in turn connected to the cable (with thin Ethernet the transceiver is integrated into the network adapter). While a simple passive cable is highly reliable for small

networks, it is not reliable for large extended networks, where damage to the wire in a single place, or a single bad connector, can make the whole Ethernet segment unusable.

Since all communication happens on the same wire, any information sent by one computer is received by all, even if that information is intended for just one destination. The network interface card interrupts the CPU only when applicable packets are received: the card ignores information not addressed to it. Use of a single cable also means that the data bandwidth is shared, such that, for example, available data bandwidth to each device is reduced when two nodes are simultaneously active because of collisions.

A collision happens when two stations attempt to transmit at the same time. They corrupt transmitted data and require stations to re-transmit. The lost data and re-transmission reduces throughput. In the worst case, where multiple active hosts connected with maximum allowed cable length attempt to transmit many short frames, excessive collisions can reduce throughput dramatically.

In a modern Ethernet, the stations do not all share one channel through a shared cable or a simple repeater hub; instead, each station communicates with a switch, which in turn forwards that traffic to the destination station. In this topology, collisions on a link are only possible if station and switch attempt to communicate with each other at the same time and the connection is not full duplex (which it is in the vast majority of the cases nowadays). Furthermore, the 10BASE-T standard introduced a full duplex mode of operation which became common with Fast Ethernet and standard with Gigabit Ethernet. In full duplex, switch and station can send and receive simultaneously, and therefore modern Ethernet is completely collision-free.

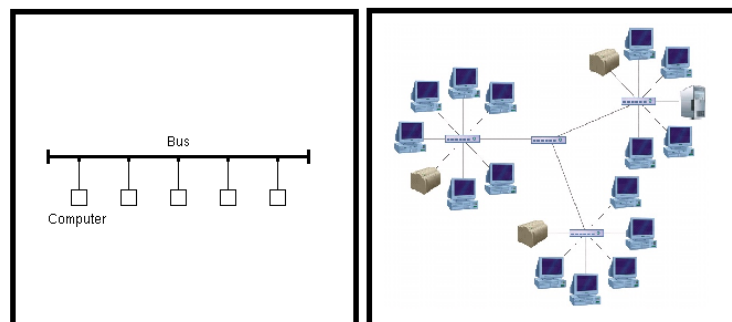


Figure 2.11: The original Ethernet implementation: shared medium, collision-prone. All computers trying to communicate share the same cable, and so compete with each other VS. Modern Ethernet implementation: switched connection, collision-free. Each computer communicates directly only with its own switch, without competition for the cable with others.

### 2.3.1.1 Frame format

The internal structure of an Ethernet frame is specified in IEEE 802.3 [9]. The Figure below shows the complete Ethernet packet and the frame inside, as transmitted, for the payload size up to the MTU (maximum transmission unit) of 1500 octets. Some implementations of Gigabit Ethernet and other higher-speed variants of Ethernet support larger frames, known as jumbo frames.

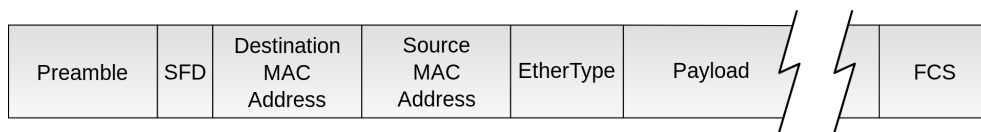


Figure 2.12: An Ethernet packet [10]

If a VLAN is used with the IEEE 802.1Q standard four additional bytes are used between the Source MAC address and the EtherType. The first half is used to identify the frame as an IEEE 802.1Q-tagged frame, the same way as the EtherType field is used but with a constant value of 0x8100 or 0x88A8. The other two bytes contain the Priority code point (PCP - 3 bits), the Drop eligible indicator (DEI - 1 bit) and the VLAN identifier (VID - 12 bit) to specify the VLAN to which the frame belongs.

### Physical layer initial frames:

- **Preamble and start frame delimiter:** An Ethernet packet starts with a seven-octet preamble and one-octet start frame delimiter (SFD). The preamble consists of a 56-bit (seven-byte) pattern of alternating 1 and 0 bits, allowing devices on the network to easily synchronize their receiver clocks, providing bit-level synchronization. It is followed by the SFD to provide byte-level synchronization and to mark a new incoming frame. [10]
- **SFD:** The SFD is the eight-bit (one-byte) value that marks the end of the preamble, which is the first field of an Ethernet packet, and indicates the beginning of the Ethernet frame. The SFD is designed to break the bit pattern of the preamble and signal the start of the actual frame.[10] The SFD is immediately followed by the destination MAC address, which is the first field in an Ethernet frame. SFD is the binary sequence 10101011 (0xAB, decimal 171 in the Ethernet LSB (Least Significant Bit) first ordering).[10]

A physical layer transceiver circuitry (PHY for short) is required to connect the Ethernet MAC to the physical medium. The connection between a PHY and MAC is independent of the physical medium and uses a bus from the media independent interface family (MII, GMII, RGMII, SGMII, XGMII). Fast Ethernet transceiver chips utilize the MII bus, which is a four-bit (one nibble) wide bus, therefore the preamble is represented as 14 instances of 0xA, and the SFD is 0xA 0xB (as nibbles). Gigabit Ethernet transceiver chips use the GMII bus, which is an eight-bit wide interface, so the preamble sequence followed by the SFD would be 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAB (as bytes).

### Data Link Layer frames:

- **Header:** The header features destination and source MAC addresses (each six octets in length), the EtherType field and, optionally, an IEEE 802.1Q tag or IEEE 802.1ad tag. The EtherType field is two octets long and it can be used for two different purposes. Values of 1500 and below mean that it is used to indicate the size of the payload in octets, while



values of 1536 and above indicate that it is used as an EtherType, to indicate which protocol is encapsulated in the payload of the frame. When used as EtherType, the length of the frame is determined by the location of the interpacket gap and valid frame check sequence (FCS). As said before, the IEEE 802.1Q tag or IEEE 802.1ad tag, if present, is a four-octet field that indicates virtual LAN (VLAN) membership and IEEE 802.1p priority. The first two octets of the tag are called the Tag Protocol IDentifier (TPID) and double as the EtherType field indicating that the frame is either 802.1Q or 802.1ad tagged. 802.1Q uses a TPID of 0x8100. 802.1ad uses a TPID of 0x88a8.

- **Payload:** The minimum payload is 42 octets when an 802.1Q tag is present and 46 octets when absent. When the actual payload is less, padding bytes are added accordingly. The maximum payload is 1500 octets. Non-standard jumbo frames allow for larger maximum payload size.
- **Frame check sequence (FCS):** The frame check sequence (FCS) is a four-octet cyclic redundancy check (CRC) that allows detection of corrupted data within the entire frame as received on the receiver side. The standard states that the FCS value is computed as a function of the protected MAC frame fields: source and destination address, length/type field, MAC client data and padding (that is, all fields except the FCS).

#### Physical layer final frames:

- **End of frame** The end of a frame is usually indicated by the end-of-data-stream symbol at the physical layer or by loss of the carrier signal; an example is 10BASE-T, where the receiving station detects the end of a transmitted frame by loss of the carrier. Later physical layers use an explicit end of data or end of stream symbol or sequence to avoid ambiguity, especially where the carrier is continually sent between frames; an example is Gigabit Ethernet with its 8b/10b encoding scheme that uses special symbols which are transmitted before and after a frame is transmitted.[11] [12]
- **Interpacket gap:** Interpacket gap is idle time between packets. After a packet has been sent, transmitters are required to transmit a minimum of 96 bits (12 octets) of idle line state before transmitting the next packet.

### 2.3.2 Automotive Ethernet

With new technologies like smartphone connectivity, navigation systems and interactive feedback systems in cars, Automotive technology and sensor quality is rapidly evolving. In order to achieve the bandwidth requirements, automotive manufacturers have been adding wiring harness to the system, requiring high investment costs. Ethernet deployment can reduce these costs, however this protocol was not originally designed to be a real time or deterministic system [13]. A major requirement of automotive Ethernet is its ability to support a reliable and deterministic interconnection of nodes, since many subsystems in a car are safety-critical. [Table 2.1](#) summarizes

the main automotive functional domains and categorizes them in terms of reliability requirements [14]. As shown in the table, the domains with high critical levels require a real-time control behavior in a car. These critical systems are designed to protect and assist the driver in order to avoid or at least decrease the chances of an accident. Existing automotive networks such as LIN (Local Interconnect Network) [15], MOST (Media Oriented Systems Transport) [16] and FlexRay [17] are time-triggered protocols, which have a defined scheduling that effectively enables the identification of unexpected states of the protocol. On the other hand, since traditional Ethernet is an event-triggered protocol, it is not appropriate to hard real-time systems because of the unpredictability of latency, jitter and message arrivals. Deploying Ethernet in the automotive domain requires additional features to the original protocol to ensure a safe and reliable driving experience. [18]

Domain	Devices	Reliability requirements
Powertrain	Engine control, fuel injection, valve control	High
Chassis	Vehicle dynamics control (VDC), Anti-lock brake system (ABS), Air-bag control	High
Body & Comfort	Climate control, locks, window and seat control	Low
Driver assistance	Rear and front view cameras, park assistance systems	Medium
Telematics/Infotainment	Car stereos, speakers, GPS, Internet	Low

Table 2.1: In-vehicle domains and the level of criticalities [18]

### 2.3.3 100/1000 BASE-T1 Ethernet

100BASE-T1 Ethernet was the first communication protocol that was widely accepted and used in the automotive industry. Some vehicles have used 100BASE-TX for on board diagnostic (OBD) scan tools. However, 100BASE-TX was not able to grow within the automotive ecosystem because it requires two twisted-pair cables and does not meet strict Comité International Spécial des Perturbations Radioélectriques (CISPR) 25 Class 5 radiated emissions limits [19].

On 30 June 2016, Ethernet 1000Base-T1 was approved by the IEEE-SA Standards Board [20]. This protocol reaches the capability of transmitting at a rate of a gigabit per second, using a cable of up to 15m, and supports previous standards (10/100Base-T1). The differences between 100Base-T1 and 1000Base-T1 are only related to the data rate and the modulation as it will be explained in section 2.3.3.2.

#### 2.3.3.1 General design

Fast Ethernet (100BASE-TX) is an extension of the 10 megabit Ethernet standard. It runs on twisted pair or optical fiber cable in a star wired bus topology, similar to the IEEE standard

802.3i [21] called 10BASE-T, itself an evolution of 10BASE5 (802.3 [21]) and 10BASE2 (802.3a [21]). Fast Ethernet devices are generally backward compatible with existing 10BASE-T systems, enabling plug-and-play upgrades from 10BASE-T. Most switches and other networking devices with ports capable of Fast Ethernet can perform auto negotiation, sensing a piece of 10BASE-T equipment and setting the port to 10BASE-T half duplex if the 10BASE-T equipment cannot perform auto negotiation itself. The standard specifies the use of CSMA/CD for media access control. A full-duplex mode is also specified and in practice all modern networks use Ethernet switches and operate in full-duplex mode, even as legacy devices that use half duplex still exist.

A Fast Ethernet adapter can be logically divided into a media access controller (MAC), which deals with the higher-level issues of medium availability, and a physical layer interface (PHY). The MAC is typically linked to the PHY by a four-bit 25 MHz synchronous parallel interface known as a media-independent interface (MII), or by a two-bit 50 MHz variant called reduced media independent interface (RMII). In rare cases the MII may be an external connection but is usually a connection between ICs in a network adapter or even two sections within a single IC. The specs are written based on the assumption that the interface between MAC and PHY will be an MII but they do not require it. Fast Ethernet or Ethernet hubs may use the MII to connect to multiple PHYs for their different interfaces.

The MII fixes the theoretical maximum data bit rate for all versions of Fast Ethernet to 100 Mbit/s. The information rate actually observed on real networks is less than the theoretical maximum, due to the necessary header and trailer (addressing and error-detection bits) on every Ethernet frame, and the required inter packet gap between transmissions.

100BASE-T1 and 1000BASE-T1 was developed to meet the needs of an automotive system. It keeps the same frame format as the 100Base-TX and only modifies the physical layer to allow communications at 100 and 1000 Mbps speeds over communication distances of at least 15m only requiring an unshielded single twisted-pair cable. 100/1000BASE-T1 emissions profile stay under CISPR 25 Class 5 Annex G stripline method and provides a robust and efficient signaling scheme with high spectral efficiency which is required by automotive applications. This limits the signal bandwidth but improves return loss, reduces cross talk, and ensures that automotive Ethernet standard passes the stringent automotive electromagnetic emission requirements. [19], [14], [22]. 100/1000BASE-T1 standardizes the in-vehicle ecosystem to a network architecture, simplifying the overall communication between ECUs and possibly even eliminating the need for older or less prevalent protocols such as Media Oriented Systems Transport (MOST) or FlexRay. 100/1000BASE-T1 can enable the communication of audio, video, connected car, firmware/software and calibration data within vehicles using the audio video bridging (AVB) collection of Ethernet protocols over unshielded single twisted-pair cable. The AVB collection of standards has low and deterministic latency, synchronized nodes and traffic shaping [19], [23]. These aspects are important for communicating the various types of information in automotive systems and enable 100/1000BASE-T1 to carry different types of data with various priorities (low data rate and high priority vs. high data rate and low priority, as well as time synchronization). [24]

### 2.3.3.2 100/1000 BASE-T1 physical layer

While 100BASE-T1 operates with a unique 4-bit to 3-bit (4B3B), 3-bit to 2-ternary pair (3B2T) and three level pulse amplitude modulation (PAM3) encoding scheme to achieve reduced emissions compared to Fast Ethernet (100BASE-TX), 1000BASE-T1 uses a 80B/81B PAM3 modulation at 750 Mb/s (three bits transmitted as two ternary symbols). A 100/1000BASE-T1 PHY performs all necessary scrambling and encoding before transmission over the unshielded single twisted-pair cable. 100/1000BASE-T1 is transparent from a MAC in that the existing Media Independent Interface (MII) has not changed. There are four main xMIIs currently in use today for 100/1000BASE-T1: [24]

- MII:
  - 4-bit-wide data interface
  - Receive and transmit controls
  - Receive and transmit clocks
  
- Reduced Media Independent Interface (RMII):
  - 2-bit-wide data interface
  - Receive and transmit controls
  - Single clock reference
  
- Reduced Gigabit Media Independent Interface (RGMII):
  - 4-bit-wide data interface
  - Receive and transmit controls
  - Receive and transmit clocks
  
- Serial Gigabit Media Independent Interface (SGMII):
  - 2-pin low-voltage differential signaling (LVDS) receive path
  - 2-pin LVDS transmit path
  
- r1–Following the RTR and r0 bits, an additional reserve bit has been included ahead of the DLC bit.

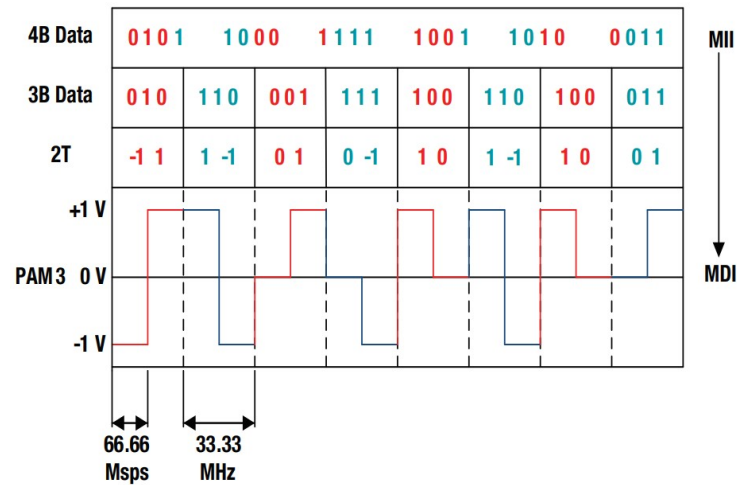


Figure 2.13: Data conversion from MII to MDI [25]

For example, a 100BASE-T1 PHY communicating with a MAC via RGMII will receive four parallel bits clocked at 25 MHz (100 Mbps total). The PHY converts these four bits to three bits and increases the clock frequency to 33 1/3 MHz to maintain the 100 Mbps bit rate. (If a frame is not divisible by three, the PHY adds stuffing bits to enable proper conversion. The link partner removes these stuffing bits before transfer to the MAC.) Using each group of three bits, a ternary pair (2T) is generated by the PHY based off of the symbol map shown in Figure 2.2.

3-bit data	TA	TB
0 0 0	-1	0
0 0 1	0	1
0 1 0	-1	1
0 1 1	0	1
1 0 0	1	0
1 0 1	0	-1
1 1 0	1	-1
1 1 1	0	-1

Table 2.2: 100BASE-T1 3 bit to 2 ternary mapping [26]

Finally, the ternary pair vector (TA, TB) is transmitted using three-level pulse amplitude modulation (PAM3) at a 66 2/3 MHz fundamental frequency. Figure 3 shows the data converted from MII to the Medium Dependent Interface (MDI) through the PHY. Looking at the PAM3 signal, it is easy to see that every 33 1/3 MHz period will represent three bits of data and thus transfer data at 100 Mbps. This signal is sent using three voltage levels (+1 V, 0 V and -1 V) with less than 2.2 V peak-to-peak (when measured with 100  $\Omega$  differential termination) [26].

## 2.4 Available Sniffer tools

As mentioned in the [Chapter 1](#) with the introduction of telematics and other electronic technologies of vehicles, the intercommunication of the devices proved to be a lot to handle. To make it easier for developers to understand how their technologies are working in the car environment and if the information that is being sent and received by each ECU (Electronic Control Unit) is the correct, ways of debugging the system started being created.

One of the most important ones is the way to monitor the network inside a automobile by identifying the information that is transported on the buses and hasten the identification of problems.

As for the moment there are 3 available options for sniffer tools in the market which meet the requirements for this project but each one has its disadvantages. The aim of this dissertation is to fully replace these equipments without its disadvantages.

Those 3 tools are:

- Vector VN5610A from Vector
- Vector VN5610 from Vector
- Universal EMC Device from Technica Engineering

### 2.4.1 VN5610A and VN5610

The VN5610A and VN5640 are compact and very powerful interfaces with USB host connection for accessing Ethernet and CAN (FD) networks. There are many use cases for the VN5610A and VN5640 such as Ethernet monitoring, frame- as well as load generation or synchronous tracing of Ethernet frames with other bus systems such as CAN. Furthermore the VN5640 interface offers with its numerous Ethernet channels a powerful platform for extensive analysis, simulation or complex testing tasks within an Automotive Ethernet network. [27]

Particularly, the VN5610(A) supports the Ethernet physical layer 10BASE-T, 100BASE-T1 (OPEN Alliance BroadR-Reach), 100BASE-TX and 1000BASE-T and enables the transparent monitoring and logging of Ethernet data streams and CAN events with minimal latency times and high resolution time stamps. (in the order of 1  $\mu$ s) [28]



Figure 2.14: VN5610A/VN5640 with 100 BASE-T1 option [29]

#### Common features of VN5610 and VN5610A:

- Support of two independent Ethernet ports, available as 2x RJ45 or 1x D-SUB9;
- Support of standard Ethernet (10BASE-T/100BASE-TX/1000BASE-T);
- Support of two independent CAN/CAN FD channels, available as 1x D-SUB9;
- High resolution time stamps for Ethernet frames;
- High resolution time stamps for CAN/CAN FD frames;
- Software and hardware time synchronization of multiple Vector network interfaces;
- Internal three-way-routing in/monitor/out;
- Robustness, power supply and temperature ranges suitable for automotive and industrial applications

### Differences between VN5610 and VN5610A:

- **VN5610:**
  - Support of BroadR-Reach physical layer
- **VN5610A:**
  - Support of 100BASE-T1 (OPEN Alliance BroadR-Reach);
  - Support of one digital input/output (e. g. for DoIP Activation Line);

### Disadvantages:

The only, although big, disadvantage of these two devices are the huge costs. Each box costs around 2000 euros which is already a lot of money for a company that needs to have several but, added to that, there is also a yearly cost for the software license which is close to 7000 euros. This disadvantage makes it obvious why a low cost Hardware which performs as good or even better for the company needs would be of great value.

## 2.4.2 Universal EMC Device

The Universal EMC Device is a testing equipment which can perform data logging and traffic generation under high electromagnetic radiations and without influencing measurements.

It has one Fiber optic Ethernet connection to access all the automotive buses. The device will take care automatically of the conversion between Gigabit Ethernet and the automotive standard buses, bidirectionally. It features CAN/CAN FD 100BASE-T1 and others.



Figure 2.15: Universal EMC device [30]



**Universal EMC Device features:**

- 6x CAN/CAN-FD;
- 4x 100BASE-T1 Ports;
- 2x SFP GB Ethernet optical ports;
- 2x LIN channels;
- 1x FlexRay Channel A;
- 1x OBD port (Fast Ethernet);

**Disadvantages:**

The disadvantages of this device are not having automotive Gigabit Ethernet and not being able to modify the software that supports this hardware.

## 2.5 Summary and main conclusions

Vehicles now routinely accommodate multiple cameras, on-board diagnostics, advanced driver assistance systems, infotainment systems, in-dash displays etc. With all the added hardware and software comes a massive demand for bandwidth and along with it comes the desire for networks with an open architecture that is scalable, future-proof, and can maintain multiple systems and devices.

Many of these requirements are beyond the capabilities of CAN, while Ethernet promises to enhance performance and allow powerful and valuable applications. Yet, CAN Bus networks address a variety of high level safety and performance features that Ethernet cannot match such as the low cost, the ability to function in a challenging electrical environment, a high degree of real-time capability, excellent error detection and fault confinement capabilities and ease of use. For these reasons both of these networks will coexist in the future of the automotive networks which is why a bus sniffer that supports both interfaces will be very useful.

As for the market for bus analyzers in the area of automotive applications there are 4 or 5 companies that produce and sell such products (Keysight / IXIA, Spirent SmartBits, Telemotive, Vector) but since these products use fairly new technologies [31] all of them suffer from the same problem - very high prices, mainly when we think of scalability. For this reason, a device which could perform as well or even better as those, but developed by Bosch for its own use would be of great significance.



## Chapter 3

# Proposed architecture

The previous chapter was focused on exposing the necessary knowledge and the state of the art solutions that can be used as a bus sniffer for CAN and Ethernet. In this chapter, we will detail what are the requirements for this project, an option that was previously accepted as a solution but has been discarded after some careful inspection, the reasons that led to choose a new solution and how it is implemented.

### 3.1 Requirements analysis

In today's automotive industry, performance standards and safety requirements are continually increasing. Tests are more rigorous and have higher standards. This fact in combination with the progress of the technologies used in the automotive industries drive the needs and requirements of testing equipment to equally high standards.

In particular, the bus sniffer to be developed in this dissertation is going to be integrated in a test system that is expected to be able to work with the latest technologies available in the industry. Its requirements are:

- Support for both 100 and 1000 Mbps Ethernet BASE-T1;
- Support for CAN-FD;
- Real-time packet capture with precise time stamps: average accuracy in the order of tens of micro seconds for packets that are sent with a period higher than 1ms;
- Packets need to be sent in chunks by Wi-Fi to a remote Desktop;
- The whole packet that was captured has to be sent with the addition of the timestamp of capture and the interface;
- Low cost : < 500 euros;

Unlike 100 Mbps Ethernet and CAN, Gigabit Ethernet is still a fairly new technology with its last standard defined in the second half of 2018 [31] which makes it the "bottleneck" technology when searching board options to develop a bus sniffer.

The need of precise time stamps and packet grouping means that a simple "switch" redirecting packets from the buses to the destination isn't enough. The packets need some sort of processing to save the timestamps in its payload and to gather them until a certain condition is met so that it can all be sent at once to the destination.

Finally, the most important requirement that will distinguish this tool from the ones available in the market is being low cost. A total price of 500 euros in comparison to the option that is used in the company (and to most of the options in the market) will reduce in 96% the costs of one tool (comparison using a VN5610A + a CANoe annual license).

### 3.2 Previous work

This dissertation is a continuation of a project which has been already started, also in a thesis environment, by Bosch in 2019. During this dissertation and after some development and research, the board that had been chosen before was discarded. This section aims to briefly explain why this option was taken and why it was later discarded.

An analysis of multiple boards to create a bus sniffer for CAN and Ethernet has been made back then and it comprised the following options:

- SJA1105Q-EVB;
- SJA1105SMBEVM;
- ADSP-SC584 DSP;
- OM14510 - SJA1105TJP;

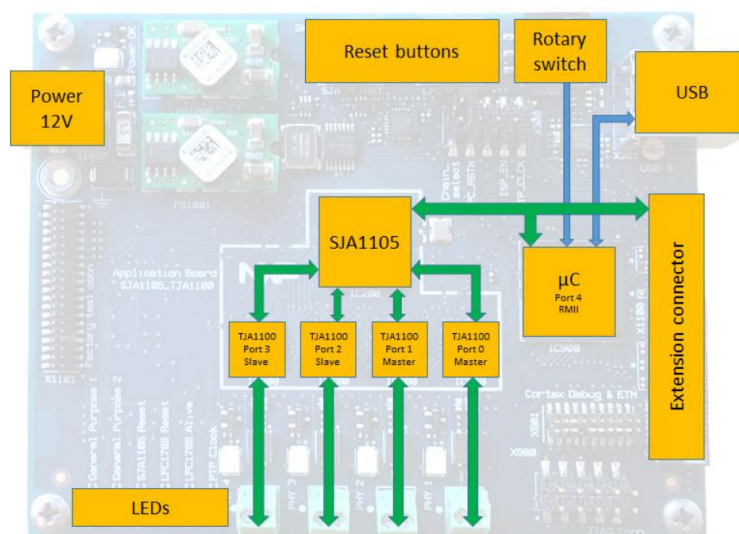


Figure 3.1: Diagram block representation of OM14510-SJA1105TJP board [32]

The OM14510 - SJA1105TJP was the board that was chosen. Its primary functional components are shown in Figure 3.1.

The given reasons to sustain this choice, according to what was known back then, were not needing a NDA to get full access to the board information which the other options needed. Having an IEEE 802.3-compliant 5-port automotive Ethernet switch with the option to individually configure each of the five ports to operate at 100Mbit/s or 1000Mbit/s. Having the option to connect a CAN breakout board via SPI to the general purpose pins of the board by using the extension connector and to reprogram the on-board  $\mu$ C (a LPC1788 by NXP) to handle the packets both from the CAN and the Ethernet side and finally costing only 500 euros. The setup that was expected to achieve is represented in Figure 3.2.

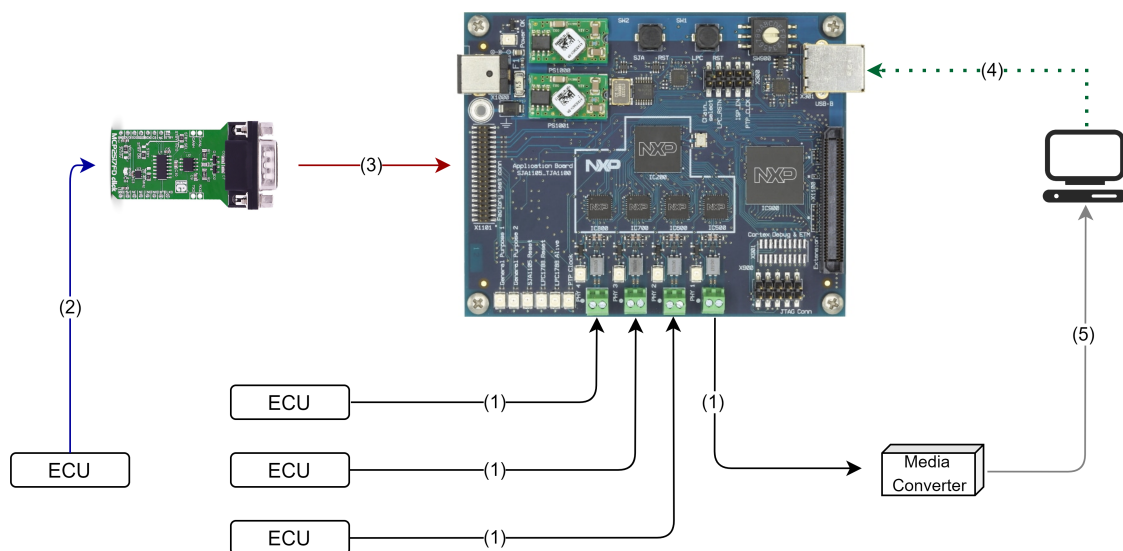


Figure 3.2: OM14510-SJA1105 with MCP2517FD CAN breakout setup. Adapted from [32].

- (1) Automotive Ethernet links between ECUs and OM14510-SJA1105 board and OM14510-SJA1105 and media converter.
- (2) CAN link between ECU and a CAN breakout board (MCP2561FD)
- (3) SPI connection between the CAN breakout board and the OM14510-SJA1105 board (the diagram of this connection is merely representative to ease readability. The real connection is made with the extension connector located on the right side of the board)
- (4) USB-B connection from the computer to the OM14510-SJA1105 board (only needed to flash code)
- (5) Ethernet link between media converter and the computer

While developing the project some step backs which weren't considered when the board was chosen in 2019 started discrediting the choice of this board. After reprogramming the  $\mu$ C from the ground, implementing a serial connection via USB-B between the computer and the  $\mu$ C, implementing the SPI communication between the  $\mu$ C and the board switch (a SJA1105) and setting

up the initial configuration of the switch the number of step backs built up to such a number that made this option unfeasible.

The list of conclusions that were taken during the mentioned development were:

- Due to privacy constrains, NXP refused the request of the source code of  $\mu C$ 's firmware which would contain all the code to initialize the board;
- The SPI communication through the extension connector to the CAN breakout board is not possible because the chip select signal that is connected to those pins is the same chip select that is used on the SPI communication with the SJA1105 switch and there are no general purpose pins available. This means that the only use of these extension SPI pins is to debug the communication;
- Even though the switch supports 1000 Mbps communication, the four TJA1100 transceivers that are on the board only support 100BASE-T1;
- The initialization of the four Ethernet transceivers would also have to be done by the  $\mu C$  and there are no signals carried to the extension pins to debug the communication;
- There is a 5th port on the switch that is connected to both the  $\mu C$  and carried to the extension connector. This port can work with gigabit Ethernet if an adequate transceiver is connected to the respective extension pins but if done so the connection to the  $\mu C$  cannot be used and vice-versa excluding the hypothesis of receiving information via gigabit Ethernet and process the packets in the  $\mu C$ .

These points completely disprove the premises that supported the previous choice of this board. This meant that, for the moment, it would be better to find another solution which could still check all the requirements that were proposed and if still, in the future, this switch should be used, a PCB made specifically for it should be created using the LPC1788, the CAN controller and transceiver and lastly the adequate number of gigabit Ethernet transceivers.

### 3.3 Hardware architecture

Having the previous project put aside, the more adequate hardware solution considering the needed requirements but also the disposable time was a Raspberry Pi 4 with a PiCAN2 shield. Even though it doesn't support CAN-FD, this shield was chosen due to its immediate availability at FEUP, which would eliminate any travel delays related to the COVID-19 pandemic, and also because the same company (SK Pang [33]) has a similar shield that supports CAN-FD and which makes the transition trivial since it won't require significant changes in software if any at all. This time the choice for this Hardware was well studied and supported by strong arguments.





Figure 3.5: PiCAN2 shield for CAN interface mounted on top of a Raspberry

- Even though Raspbian, which is a Debian based distro of Linux, is not intended for real time systems, if the results that we obtain are within the ranges of this report [36] (i.e. with delays lower than  $350\mu\text{s}$  for 95% of the cases) then it is good enough to make a proof of concept of what we want to achieve here. In the future if better performance is needed that kernel can be easily updated with multiple patches [36] [37] that turn it into a real-time operating system.
- Although hardware time stamping is not supported by the Raspberry's Ethernet transceiver (Broadcom BCM54213), the software time stamping provided by Linux has a very decent performance ( $< 10\ \mu\text{s}$  standard deviation [38]). Using the `SO_TIMESTAMP` option, the Linux kernel will timestamp the packet as soon as the kernel is notified about its arrival. This way the delay between the packet arriving and entering user-space to have a timestamp is avoided.

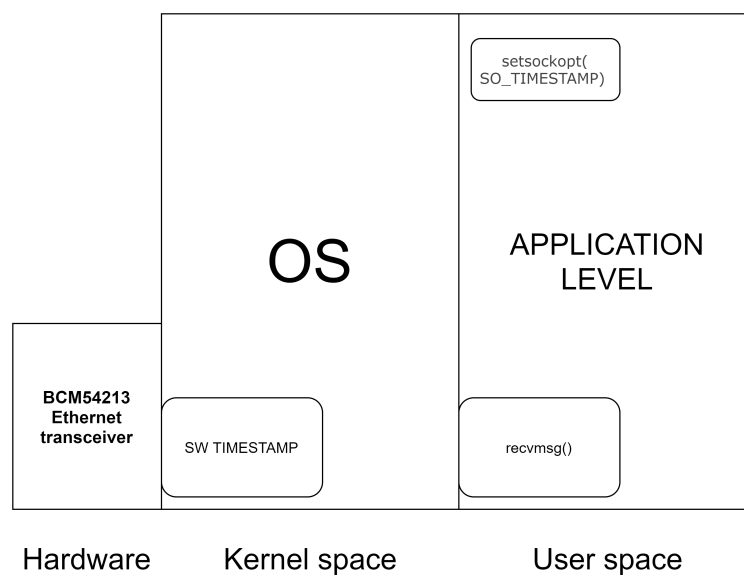


Figure 3.6: Packets software time stamping in linux



### 3.3.2 Bus sniffer hardware setup

The Raspberry Pi 4 + PiCAN 2 system will work as the bus sniffer. The packets generated by the multiple ECUs that are connected to the Ethernet network to be tested will be sent to the Raspberry Pi 4 using a Media converter in between to convert the Ethernet link from automotive Ethernet (BASE-T1) which is used in the ECUs environment (1) to the normal BASE-T Ethernet (RJ45 cable) used by the Raspberry Pi 4(2). In parallel the CAN network will be connected (3) directly to the PiCAN 2 board which will subsequently receive the packets and deliver them through SPI (4) to the Raspberry Pi 4. In turn, when a certain software condition (which will be detailed later in [section 3.4](#)) is met the Raspberry Pi 4 will send the packets that has collected both from Ethernet and from CAN to a target Desktop via Wi-fi (5).

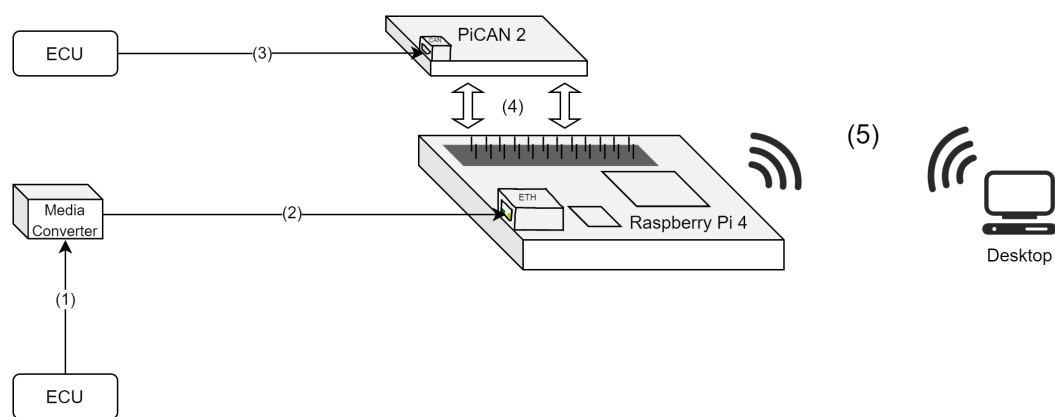


Figure 3.7: Raspberry Pi 4 + PiCAN bus sniffer hardware block diagram.

- (1) Ethernet 1000/100/10BASE-T1 link between ECUs Ethernet network and the Media Converter
- (2) Ethernet 1000/100/10BASE-T link between the Media Converter and the Raspberry Pi 4
- (3) CAN link between ECUs CAN network and the PiCAN2 breakout board
- (4) SPI link between the PiCAN 2 breakout board and the Raspberry Pi 4 (In reality the breakout board is mounted on top of the Raspberry Pi 4. The space in between is on the picture just to ease visibility)
- (5) Wi-fi connection between the Raspberry Pi 4 and the Desktop

This whole setup can be multiplied any number of times to work in parallel using the same target Desktop or different ones. The total cost of a setup such as the one presented in [Figure 3.7](#) including the Raspberry Pi 4, the PiCAN 2 board and the media converter can be as low as 200€ and will never exceed the 300€.

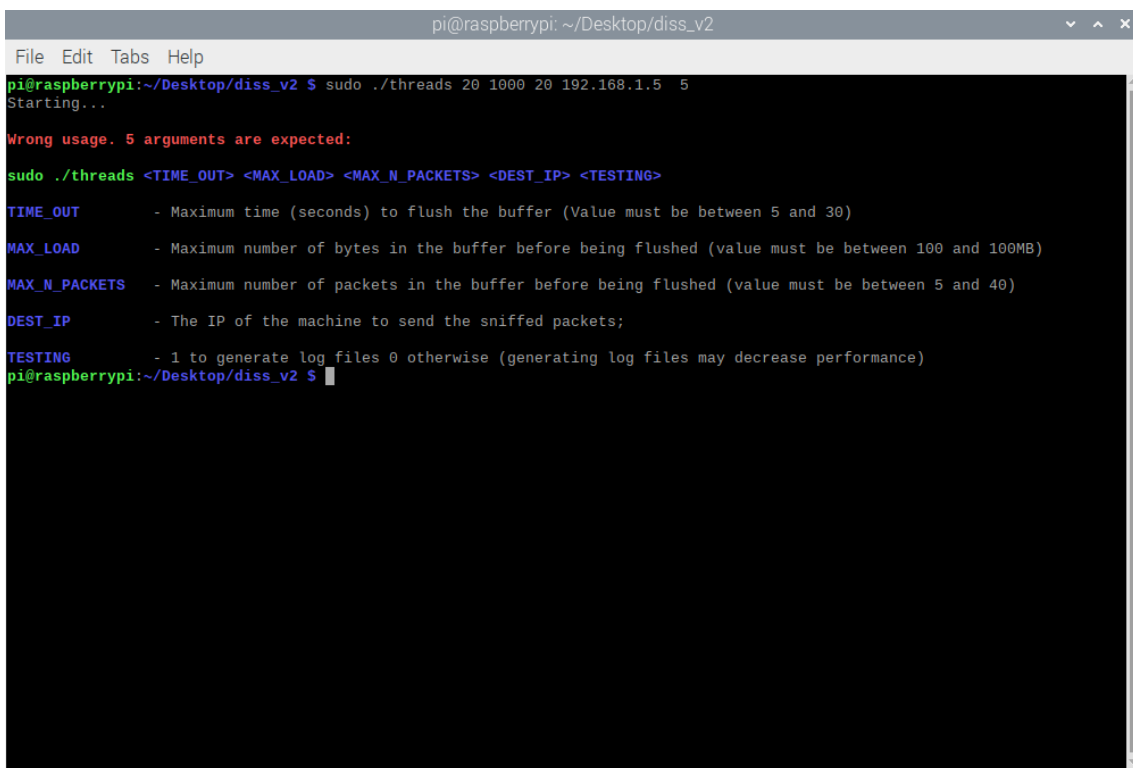
## 3.4 Bus sniffer software implementation

To sniff the Ethernet and CAN buses and send this data to the target Desktop, a C based program named *CanEthAnalyz* was developed. This Software supports a setup such as the one shown in [Figure 3.7](#). This program is given the highest priority to run on the Linux OS in order to be scheduled for as much time as possible. The program creates 3 threads, one for each interface

used (CAN, Ethernet and Wlan). The *CanEthAnalyze* expects some parameters as inputs which are detailed in 3.4.1 and, while running, outputs some information on the command line. 3.4.3.

### 3.4.1 Bus sniffer input parameters

As mentioned previously, the *CanEthAnalyze* expects some parameters as input. In fact, the program needs 5 inputs to start running. Some of them exist to give the user some freedom to choose how the program should behave and to allow different implementations on the upward design of the software on the desktop while others are necessary as a functional part of the program. Without any of these 5 parameters the program won't start and will display an error message followed by an explanation for a proper call to the program as shown in fig. 3.8



```

pi@raspberrypi: ~/Desktop/diss_v2
File Edit Tabs Help
pi@raspberrypi:~/Desktop/diss_v2 $ sudo ./threads 20 1000 20 192.168.1.5 5
Starting...
Wrong usage. 5 arguments are expected:
sudo ./threads <TIME_OUT> <MAX_LOAD> <MAX_N_PACKETS> <DEST_IP> <TESTING>
TIME_OUT      - Maximum time (seconds) to flush the buffer (Value must be between 5 and 30)
MAX_LOAD      - Maximum number of bytes in the buffer before being flushed (value must be between 100 and 100MB)
MAX_N_PACKETS - Maximum number of packets in the buffer before being flushed (value must be between 5 and 40)
DEST_IP       - The IP of the machine to send the sniffed packets;
TESTING       - 1 to generate log files 0 otherwise (generating log files may decrease performance)
pi@raspberrypi:~/Desktop/diss_v2 $

```

Figure 3.8: Command line error on input parameters

The five parameters are:

<TIME\_OUT> <MAX\_LOAD> <MAX\_N\_PACKETS> <DEST\_IP> <TESTING>

The <TIME\_OUT> <MAX\_LOAD> and <MAX\_N\_PACKETS> define the conditions which have to be met to flush the packets that are on the sending buffer from the Raspberry Pi 4 to the target desktop. The packets will be flushed as soon as any one of the three conditions are met. These parameters are the ones that will modify the functionality of the program mainly on the way/timing that the Desktop receives the packets that are sent by the Raspberry Pi 4.

- Thus the <TIME\_OUT> parameter defines for how long the Raspberry Pi 4 can stay without communicating with the Desktop. If the time out value (in seconds) is reached, the

Raspberry Pi 4 will flush the transmit buffer with however many packets there are. This parameter must be a value between 5 and 30.

- Similarly the `<MAX_LOAD>` is a parameter that defines when the Raspberry Pi 4 should flush the transmit buffer. This value (in bytes) is compared to the sum of all the bytes in the packets that are on the transmit buffer and as soon as it is reached the Raspberry Pi 4 flushes the buffer to the target Desktop. This value must be between 100 bytes and 100MB.
- Finally the `<MAX_N_PACKETS>` specifies the amount of packets that shall stay on the transmit buffer before it gets flushed. As soon as `<MAX_N_PACKETS>` packets get in the buffer, the Raspberry Pi 4 will flush it to the Desktop. This parameter must be a value between 5 and 40.
- The most essential parameter from the user point of view is the `<DEST_IP>`. It must be the local IP of the Desktop to which the the flushed packets must be sent. An incorrect IP will allow the program to run without the packets being sent to the desired device.
- Lastly, the `<TESTING>` is a binary parameter which should be set to 1 if the user wants to generate log files for all the packets received in each interface and 0 otherwise. This parameter should be used with caution because generating log files not only slightly decreases performance but might also might leave the Raspberry Pi 4 out of memory if the program is left running for too long, since all the packets received are mirrored to the log files.

### 3.4.2 Bus sniffer implementation design

The basic structure of the *CanEthAnalyz* is represented on [fig. 3.9](#). The core of the program relies on 3 threads running in loop that share information between them. One thread for each interface used: One thread to receive packets on the Ethernet interface, another to receive on the CAN interface and the last to send the information over wi-fi to the target Desktop.

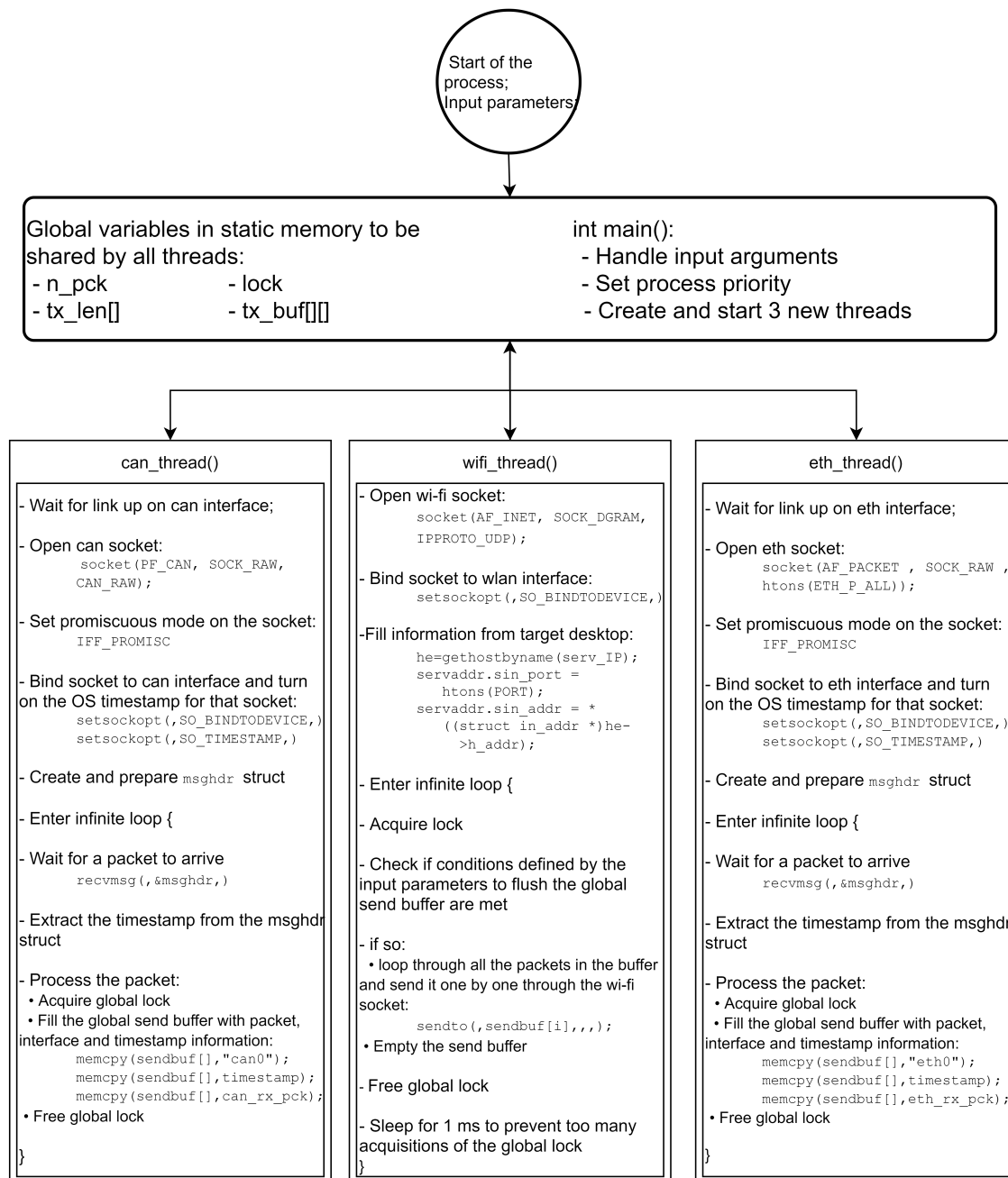


Figure 3.9: Software design of the *CanEthAnalyz* (Dummy code pieces present in the diagram are just for representation)

These 3 threads have to share the information contained in the packets that each receive. For this reason there are a couple of variables that are global on the system to be accessed by any thread, obviously with the proper use of a mutex to allow concurrency.

- The `n_pck` variable is used to keep track of the number of packets on the transmit buffer. It is incremented by the Ethernet thread and the CAN thread every time a packet is received on the respective interface and it is reset when the wi-fi thread flushes the buffer.

- `tx_buf[][]` is a multidimensional char array that is used as the transmit buffer. The first index indicates the position of the packet in the buffer and the second dimension is used as a string to store the interface on which the packet was captured, the timestamp and the whole packet that was captured.
- The `tx_len[]` is an array that in each position stores the size in bytes of each string of the `tx_buf[][]`

Finally a mutex `lock` is also kept global to handle the concurrency to the 3 variables mentioned above.

After all the initial setup is done, in the `int main()`, the input parameters from 3.4.1 are checked and the timer for the packets's timestamps is started. This means that all the timestamps in the packets are relative timestamps corresponding to the launch of the program. Also, here the priority of the process is set to `-20` which is the maximum priority for a process in a UNIX system. This is done to make the OS schedule the *CanEthAnalyz* for as much time as possible since a packet can arrive at any possible time. As mentioned in section 3.3.1 even though Raspbian is not a real time operating system and therefore we can't guarantee that our tasks will be scheduled when we need, linux still does a decent job scheduling it. According to [36], the median deviation to schedule a task in a user space program for a Linux distro such as Raspbian is  $67\mu\text{s}$ , in 95% of the cases the deviation is below  $307\mu\text{s}$  but in some exceptions the scheduler can deviate to 17ms. For a proof of concept this performance suffices but for regular use the installation of a real time dual kernel such as Xenomai [39] is recommended. This way these values can be as low as  $9\mu\text{s}$  for the median deviation,  $18\mu\text{s}$  for 95% of the cases and  $37\mu\text{s}$  as the worst case. Lastly, the `int main()` creates 3 new threads, each one responsible for an interface, and then it exits.

The 3 newly created threads are `void * eth_thread()`, `void * can_thread()` and `void * wifi_thread()`. The Ethernet and CAN threads are responsible for receiving the packets on their respective interface and to place it on the transmit buffer. The Wi-Fi thread is responsible to check if the conditions to flush the transmit buffer are met and to send it if so. The behaviour of these threads will be explained in the following paragraphs.

`void * eth_thread()` and `void * can_thread()` are very identical threads. They start by waiting for a link up on the interface. Once that happens a new raw socket is opened for both Ethernet and CAN.

```
1 socket( AF_PACKET , SOCK_RAW , htons(ETH_P_ALL));
2 socket( PF_CAN , SOCK_RAW , CAN_RAW);
```

Raw sockets are used to receive raw packets. This means packets received at the Ethernet layer will directly pass to the raw socket. Stating it precisely, a raw socket bypasses the normal TCP/IP processing and sends the packets to the specific user application, unlike other sockets such as stream sockets and data gram sockets that receive data from the transport layer which contain only the payload without headers (fig. 3.10). [40].

For this application we need to forward the whole packet as it was sent with all the headers from the Ethernet Layer which is the reason why raw sockets are used.

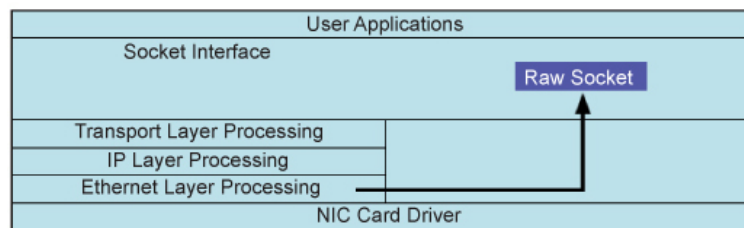


Figure 3.10: Graphical illustration of a raw socket [40]

Then the sockets are set to promiscuous mode so that every packet that is on the network is captured instead of only the packets that are addressed to the Raspberry Pi 4.

```

1 struct ifreq eth;
2 strcpy(eth.ifr_name, "eth0");
3 eth.ifr_flags |= IFF_PROMISC;
4 ioctl(eth0_sock, SIOCSIFFLAGS, &eth);

```

Afterwards the sockets are bound to their respective interface and set to have the OS timestamp option using the following calls:

```

1 setsockopt(socket, SOL_SOCKET, SO_BINDTODEVICE, devname, strlen(devname));
2 setsockopt(socket, SOL_SOCKET, SO_TIMESTAMP, &timestamp_on, sizeof(timestamp_on));

```

To complete the setup before entering the loop a `struct msg_hdr` has to be prepared to use the `recvmsg` call. This call is the only that allows to receive messages from sockets with ancillary data, in which is contained the timestamp from the OS.

```

1 iov.iov_base = msgbuf;
2 iov.iov_len = MSGBUFSIZE;
3 msg.msg_iov = &iov;
4 msg.msg_iovlen = 1;
5 msg.msg_name = &_sin;
6 msg.msg_namelen = sizeof(struct sockaddr_ll);
7 msg.msg_control = control;
8 msg.msg_controllen = 1024;

```

Entering the loop, the `recvmsg(eth0_sock, &msg, 0)` call is used in a blocking way. This means that the thread will be "sleeping" until there is an interrupt that signals `recvmsg()` that a packet arrived. Once it arrives the timestamp is retrieved from the ancillary data:

```

1 for (cmsg = CMSG_FIRSTHDR(&msg) ;
2      cmsg && (cmsg->cmsg_level == SOL_SOCKET) ;
3      cmsg = CMSG_NXTHDR(&msg, cmsg) )
4 {

```

```

5     if (cmsg->cmsg_type == SO_TIMESTAMP)
6         tv = *(struct timeval *) CMSG_DATA(cmsg);
7     }

```

At this point all the needed information about the packet is available, which means, that the only thing that is left before ending the loop is to put it all together in an array and add it to the shared transmit buffer. Since this array will be unwrapped in the desktop its structure must always be the same. The adopted format was:

- Bytes 1 to 4 (4 bytes): Interface in which the packet was received
- Bytes 5 to 12 (8 bytes): hexadecimal value of the timestamp
- Remaining bytes : The packet received including headers and payload

Because the transmit buffer and associated variables are shared across all the threads the mutex has to be used here.

```

1  /* Acquire lock to handle to shared variables */
2  pthread_mutex_lock(&lock);
3
4  /* First 4 bytes of the payload are the name of the interface */
5  memcpy(sendbuf[n_pck], "eth0", 4);
6  tx_len[n_pck] += 4;
7
8  /* Next 8 bytes are the timestamp in usec. Last byte is the MSB */
9  memcpy(&sendbuf[n_pck][tx_len[n_pck]], &usec_time, sizeof(double));
10 tx_len[n_pck] += sizeof(double);
11
12 /* The remaining of the buffer is the received packet */
13 memcpy(&sendbuf[n_pck][tx_len[n_pck]], msgbuf, data_size);
14 tx_len[n_pck] += data_size;
15 n_pck++;
16 pthread_mutex_unlock(&lock);

```

Having the `can_thread()` and `eth_thread()` adding packets to the transmit buffer in loop it is now responsibility of the `wifi_thread()` to look for the conditions to flush that buffer. Similarly to the other threads, a socket is created and associated to the respective interface. This time though, the interface is Wlan and instead of being a raw socket it is a UDP (User Datagram Protocol) socket. That means that each one of the arrays in the transmit buffer will be encapsulated by UDP to be sent to the target Desktop and that the information from the Desktop has to be stored in a `struct sockaddr_in`.

```

1  /* Filling information from server*/
2  memset(&servaddr, 0, sizeof(servaddr));
3  he=gethostbyname(serv_IP);
4  servaddr.sin_family = AF_INET;

```

```

5 servaddr.sin_port = htons(PORT);
6 servaddr.sin_addr = *((struct in_addr *)he->h_addr);

```

Thereupon starts the loop. At each iteration the lock is acquired to check if the global variables meet the conditions defined by the input parameters 3.4.1. If so the transmit buffer is flushed to the target desktop and the global variables such as the transmit buffer, number of packets in the buffer, number of bytes in the buffer and timer are reset. The lock is then handed over and the thread is put to sleep for 1ms in order to prevent too many acquisitions of the lock and to let the other threads use it.

```

1  while(1) {
2      /* lock to check shared variables */
3      pthread_mutex_lock(&lock);
4
5      if(n_pck >= MAX_N_PACKETS || load >= MAX_LOAD || t_over) {
6
7          /*Send all the packets in the buffer */
8          for(int i = 0 ; i < n_pck ; i++) {
9              n_bytes = sendto(wifi_sock, sendbuf[i], tx_len[i],
10                 MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));
11                 memset(sendbuf[i],0,1024);
12                 memset(&tx_len[i],0,sizeof(int));
13             }
14             n_pck = 0;
15             load = 0;
16             t_over = false;
17             alarm(TIME_OUT);
18         }
19         pthread_mutex_unlock(&lock);
20         /* usleep to prevent too many acquisitions of the lock */
21         usleep(1000);
22     }

```

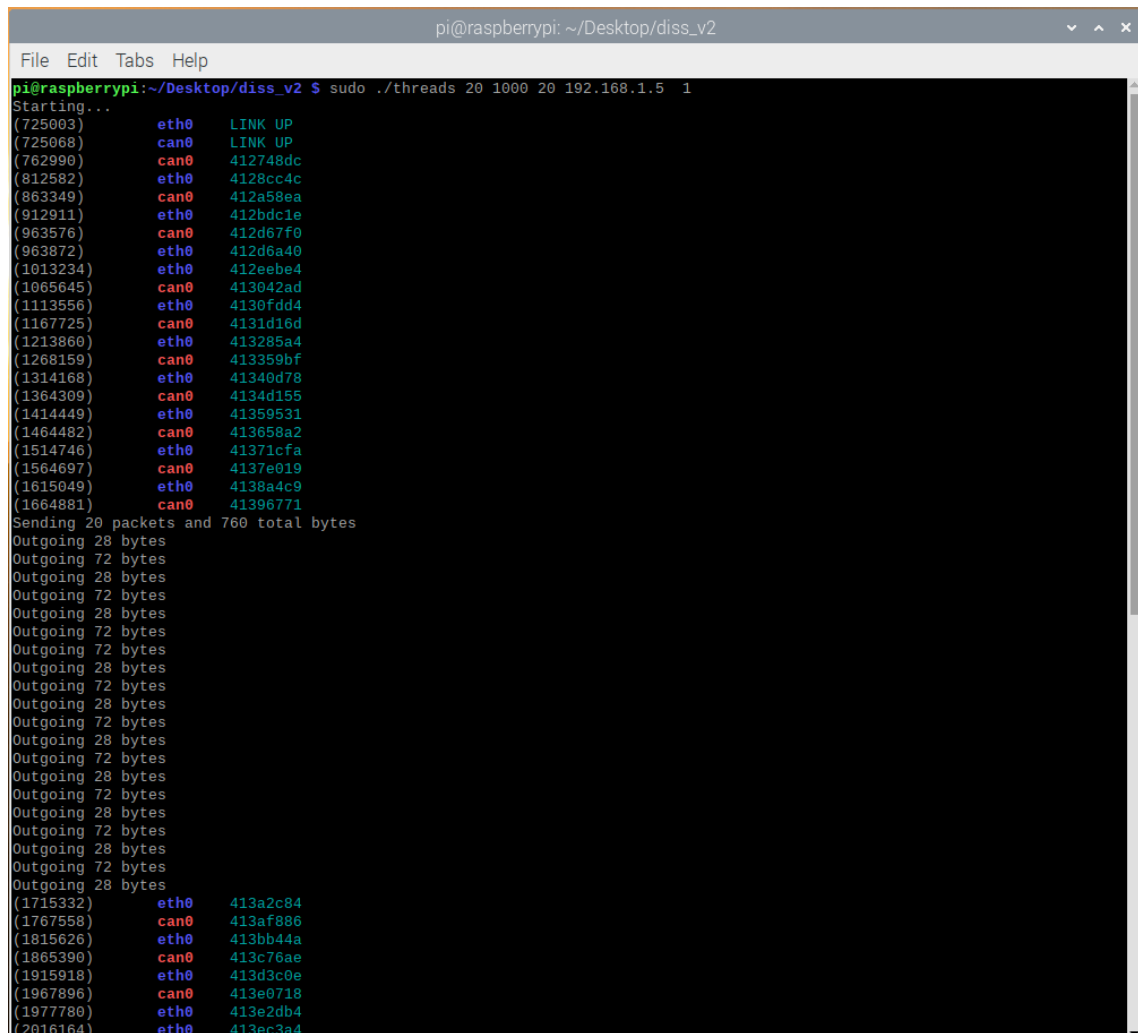
### 3.4.3 Bus sniffer output

The *CanEthAnalyze*, as explained in section 3.4.2 sends the information that runs on the CAN and Ethernet buses to a target desktop. Other than that, it also outputs real time information on the command line of the Raspberry Pi 4. Every time a new packet arrives at an interface, *CanEthAnalyze* displays on the command line the timestamp of the packet, the interface in which it was captured, the timestamp value in hexadecimal and for Ethernet packets, if the packet is TCP (Transmission Control Protocol) or UDP, it also identifies it. When the transmit buffer is flushed the number of packets sent as well as how many bytes were sent in each packet is also shown in the terminal. An example is shown in fig. 3.11.

In case the <TESTING> from the input parameters 3.4.1 is activated a log file for each receiving interface and for the wi-fi interface is created with the same information that is displayed on



the terminal , plus the time difference between the arrival of the present packet and the previous one.



```

pi@raspberrypi: ~/Desktop/diss_v2
File Edit Tabs Help
pi@raspberrypi:~/Desktop/diss_v2 $ sudo ./threads 20 1000 20 192.168.1.5 1
Starting...
(725003) eth0 LINK UP
(725068) can0 LINK UP
(762990) can0 412748dc
(812582) eth0 4128cc4c
(863349) can0 412a58ea
(912911) eth0 412bdc1e
(963576) can0 412d67f0
(963872) eth0 412d6a40
(1013234) eth0 412eebe4
(1065645) can0 413042ad
(1113556) eth0 4130fdd4
(1167725) can0 4131d16d
(1213860) eth0 413285a4
(1268159) can0 413359bf
(1314168) eth0 41340d78
(1364309) can0 4134d155
(1414449) eth0 41359531
(1464482) can0 413658a2
(1514746) eth0 41371cfa
(1564697) can0 4137e019
(1615049) eth0 4138a4c9
(1664881) can0 41396771
Sending 20 packets and 760 total bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
Outgoing 72 bytes
Outgoing 28 bytes
(1715332) eth0 413a2c84
(1767558) can0 413af886
(1815626) eth0 413bb44a
(1865390) can0 413c76ae
(1915918) eth0 413d3c0e
(1967896) can0 413e0718
(1977780) eth0 413e2db4
(2016164) eth0 413ec3a4

```

Figure 3.11: Command line output example from *CanEthAnalyz*

The software to receive the packets on the target Desktop will be developed by BOSCH in the future according to its needs but to validate the operation of the *CanEthAnalyz* we can use Wireshark and see the packets received.

### 3.5 Summary

This chapter started by exposing the project requirements indicated by Bosch which are mostly focused on hardware. Then it was presented the work that has been done previously for this project, it was explained what was done in the past, what was done on that platform during this thesis and why that option was discontinued. After that, it was revealed that the new platform would be based on a Raspberry Pi 4, it was demonstrated why this is a good choice, the arguments which supported this option and what could be its limitations. Adding to that was exposed the Hardware details of

the Raspberry Pi 4 + PiCAN2 and how it could fulfil the requirements purposed by Bosch as well as the setup in which this Bus sniffer is expected to operate.

Finally, the software implementation of the bus sniffer was the last topic presented in this section. It was divided in the input parameters, the general design of the implementation and the output parameters. There were presented the five input parameters where three of them define the conditions which have to be met to flush the packets that are on the sending buffer from the Raspberry Pi 4 to the target desktop and the two others decide to which IP the received packets should be forwarded and if the *CanEthAnalyz* should generate log files for all the packets received in each interface. After the explanation of the initial setup, the implementation of the program was detailed, starting by the main thread and global variables followed by the explanation of the function of each one of the three threads. Lastly, the outputs provided by *CanEthAnalyz* were shown; they are composed of 1) the real-time information about the arrival and dispatch of the packets; and 2) the log files generated when the program is terminated.

# Chapter 4

## Tests and requirements evaluation

This chapter presents the Raspberry Pi 4 + PiCAN2 shield performance evaluation when employing the *CanEthAnalyz* in an automotive testing environment using certified tools. The evaluation metrics are described and the performed evaluation is explained. The results are then analyzed and the main conclusions are pointed out.

### 4.1 Evaluation Conditions

In order to evaluate the performance of a Bus analyzer it has to be inserted in a network which can be customized according to the variables that are going to be evaluated. This could be done using another Raspberry Pi to create traffic but to add more integrity to the tests it was used a tool which is widely used in engineering for networking of electronic systems in the automobile and related industries. CANoe is a software tool for development, test and analysis of individual ECUs and entire ECU networks which is owned by Vector. Particularly it allows to generate CAN and Ethernet traffic using an adequate Hardware such as the VN5610A. It should be noted that even such credited software/hardware doesn't have perfect precision and that a small slice of the eventual inaccuracies are also due to the generation of data.

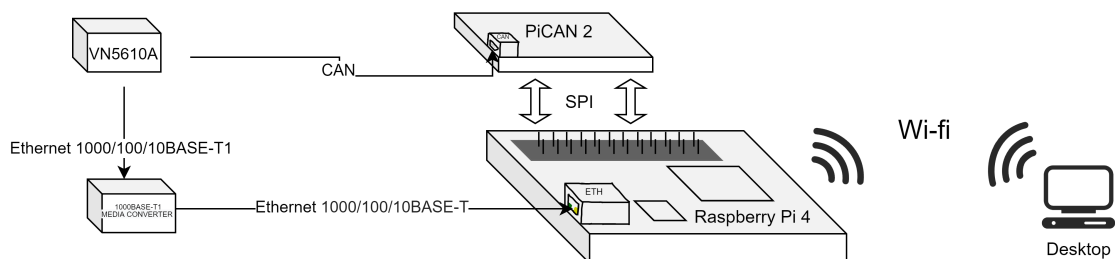


Figure 4.1: Hardware block diagram of Raspberry Pi 4 + PiCAN bus sniffer in test environment using VN5610A to generate traffic and 1000Base-T1 Media Converter by *technica engineering* to convert between Base-T1 to Base-T.

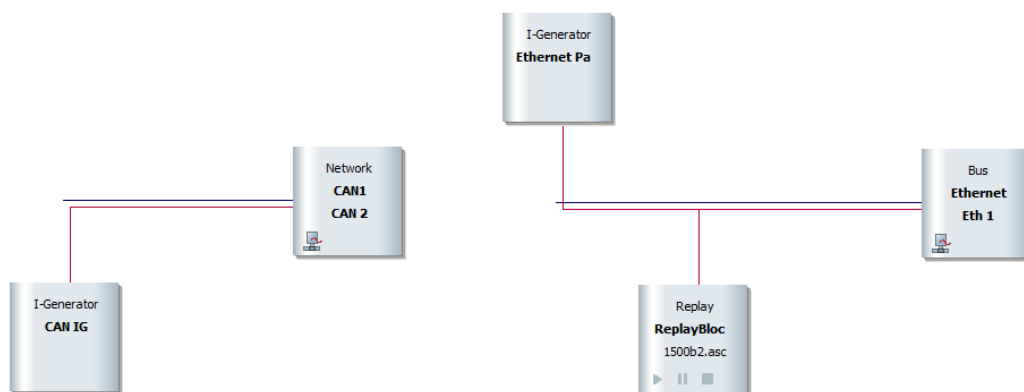
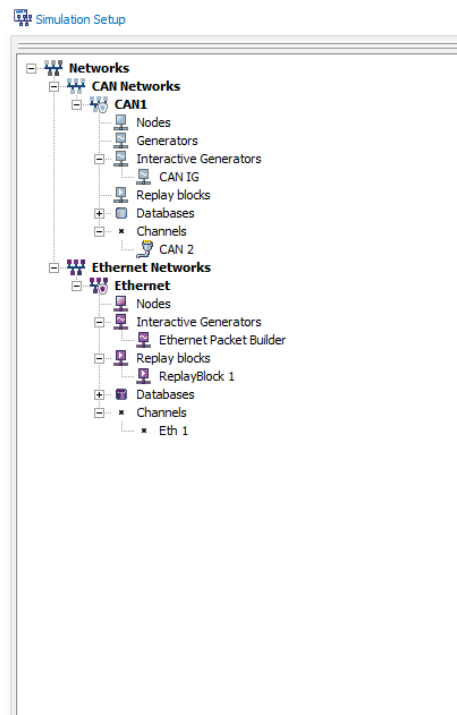


Figure 4.2: CANoe packet generation setup

In [fig. 4.1](#) is a representation of the Hardware setup that was used during the tests. In [fig. 4.2](#) is shown the CANoe window where the traffic that is sent by the VN5610A for both the CAN and Ethernet networks is simultaneously generated.

## 4.2 Results

The main ambition of these tests is to demonstrate that the Raspberry can perform well and meet the requirements to work as a bus sniffer even under high traffic and to verify the premises indicated in [Section 3.3.1](#), mainly the ones that come from [36].

The tests which were made comprised two main variables: The time in between delivery of messages to test the accuracy of the software time stamping by the OS under time stress and the size of each packet to test the processing power of the raspberry to handle and modify the packets and observe how it affects the timestamps. The packets were sent periodically in a loop using CANoe and received on the Raspberry. Because CANoe isn't able to accurately send packets every X milliseconds and always introduces a small jitter between each loop, it was measured both the difference between the send of each packet on the CANoe side and the difference between each reception on the Raspberry Pi 4. After that the 2 differences are compared to see the delays that are introduced by the Raspberry Pi 4 + PiCAN shield. To clarify the analysis method used an example is shown below.

For example if CANoe sends 5 packets with the timestamps from the first column and the Raspberry receives them with the timestamps from the second column (which can be on a different relative scale):

CANoe	Raspberry
6ms	5ms
14ms	15ms
26ms	27ms
35ms	34ms
47ms	44ms

This means that the differences that will be compared to calculate the metric that will be used are:

CANoe	Raspberry
$14 - 6 = 8\text{ms}$	$15 - 5 = 10\text{ms}$
$26 - 14 = 12\text{ms}$	$27 - 15 = 12\text{ms}$
$35 - 26 = 9\text{ms}$	$34 - 27 = 7\text{ms}$
$47 - 35 = 12\text{ms}$	$44 - 34 = 10\text{ms}$

These values from the table above will then be subtracted which will provide the following delays:

Time deviations
$10 - 8 = 2\text{ms}$
$12 - 12 = 0\text{ms}$
$7 - 9 = -2\text{ms}$
$10 - 12 = -2\text{ms}$

The measures are taken from the logs of the Raspberry and from the logs of CANoe and the relative time bases are converted into the same. By using the time stamps from both sides we

assure that the jitter produced by CANoe and VN5610A does not influence our results, which would be the case if only logs from the Raspberry were used.

### 4.2.1 Typical use

First we present a typical use of the program:

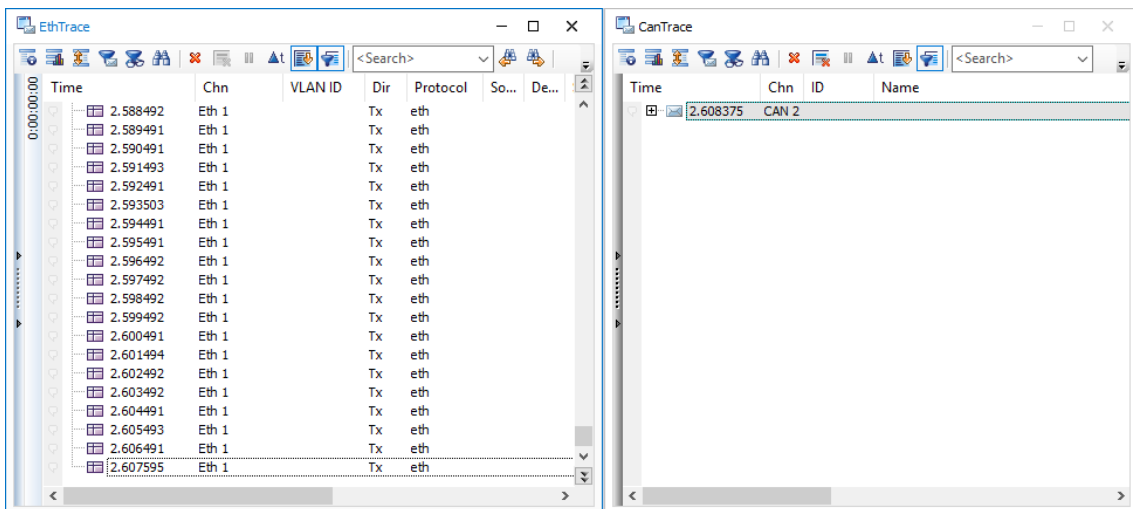


Figure 4.3: CANoe traces - Ethernet trace on the left and CAN trace on the right (on the CAN trace the message and timestamp is updated on the same place)

On [fig. 4.3](#) we can see the traces from CANoe software with the messages being sent both on the Ethernet and the CAN interfaces. Looking closely to the timestamps on the Ethernet side it can be seen small jitters on the timestamp of the packets that are sent by CANoe every millisecond. The average of the jitter is  $\pm 5\mu\text{s}$  with occasional deviations of  $\pm 50\mu\text{s}$ . If only logs from the raspberry were to be used, this jitter would influence the results, but since we use the logs both from the CANoe and from the Raspberry and compare the time difference between the sending of consecutive packets and the arrival of the same consecutive packets, this jitter does not influence the results.

On the desktop side we can see the wireshark window ([fig. 4.4](#)) with the packets that were sniffed and processed by the Raspberry Pi 4 and then sent over wi-fi to the Desktop. As we can see the payload of the packet starts by the 4 bytes which contain the interface in which the packet was captured (in this case eth0), then 8 bytes which is the timestamp of the packet and then the full raw packet with the headers and payload that was intercepted by the Raspberry Pi 4 + PiCAN 2 shield.

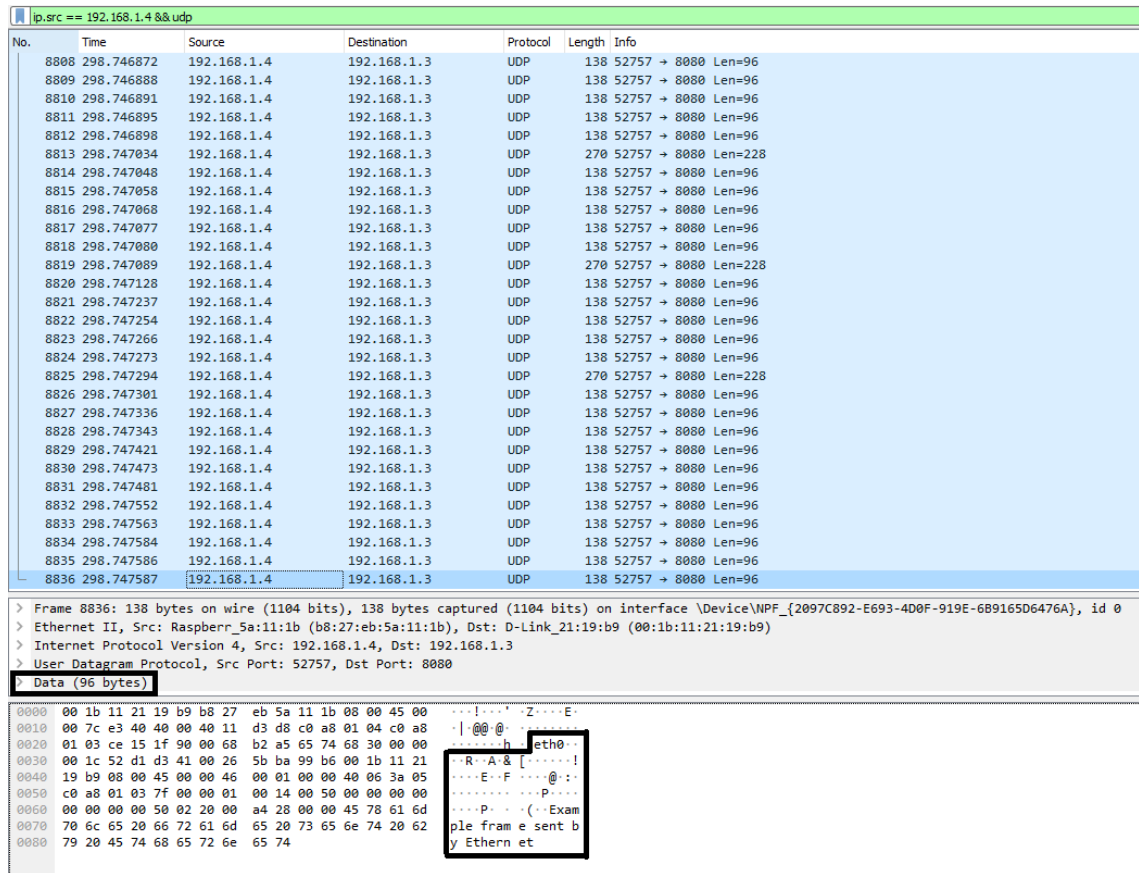


Figure 4.4: Wireshark log capture from the packets that are sent by the Raspberry Pi 4

### 4.2.2 Time stress test

For this test it was evaluated the performance of the Raspberry Pi relatively to the time between the arrival of packets. In the CANoe a packet is sent both by Ethernet and CAN periodically and then the performance of the Raspberry is evaluated regarding the time deviation between consecutive packets to a reference value (which is the interval between the send of those packets measured in the CANoe log) as referred in [section 4.2](#). The results are presented first for CAN and then Ethernet and comprise the values of 50ms, 10ms, 5ms and 1ms for 150 thousand of packets received. This range of X values was chosen in accordance to the system in which it will be used. On the left side of the following images, starting on [4.5](#), is the spectrum for the deviation from the reference value of all the received packets and on the right is the cumulative percentage of packets received regarding its time deviation.

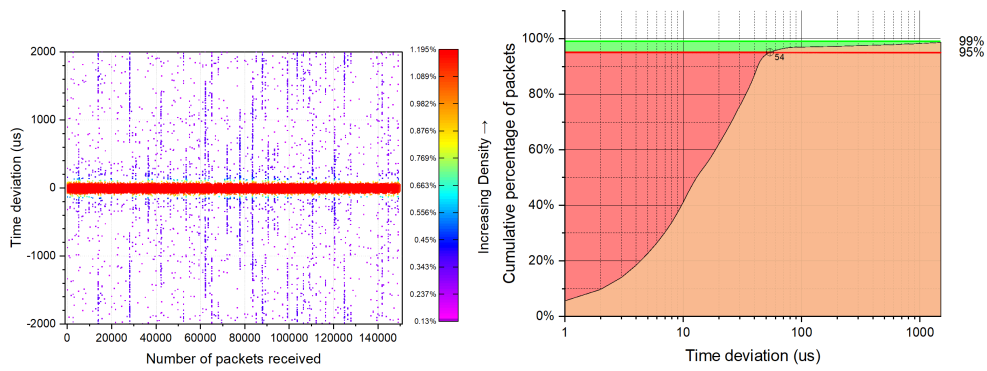


Figure 4.5: CAN results for 50ms

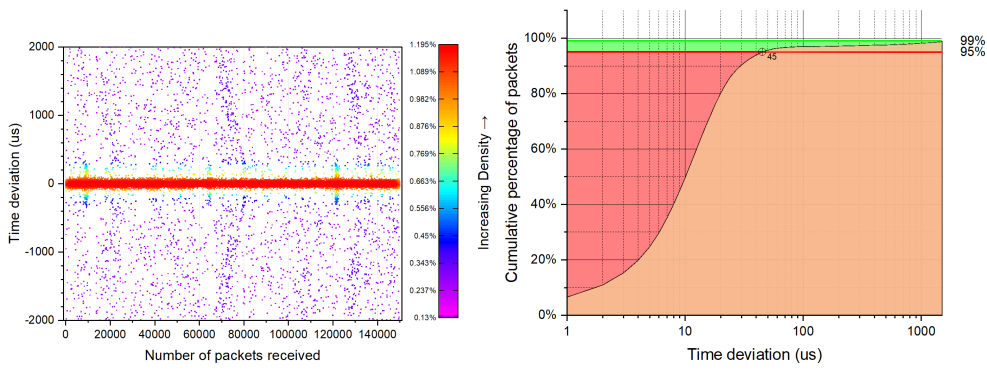


Figure 4.6: CAN results for 10ms

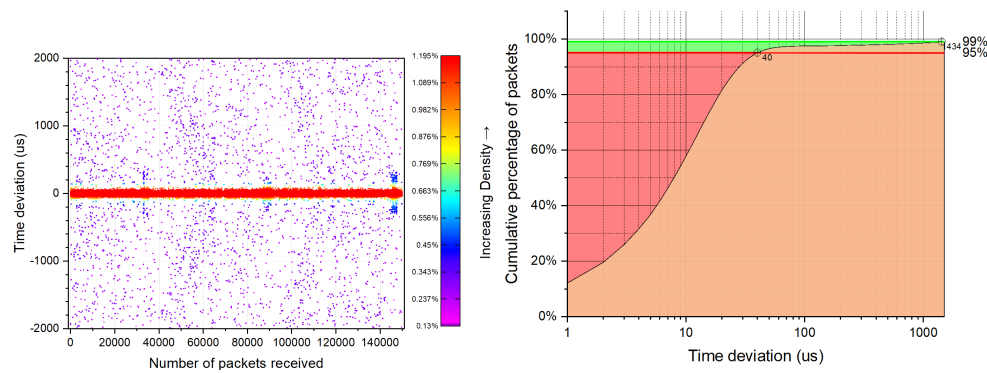


Figure 4.7: CAN results for 5ms

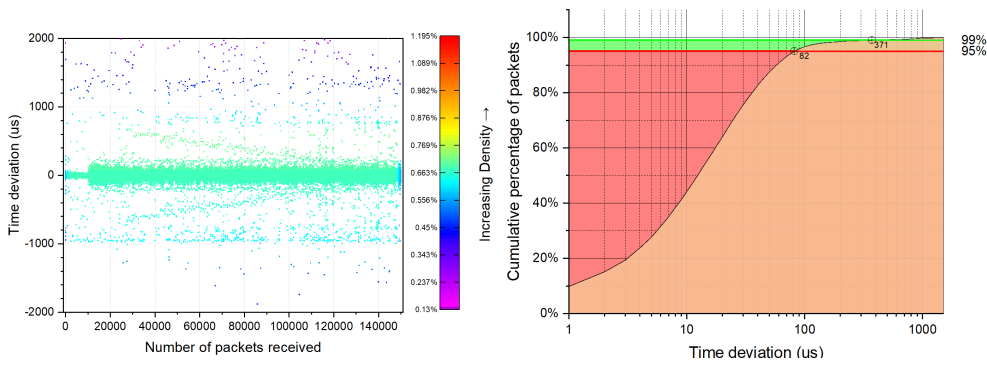


Figure 4.8: CAN results for 1ms



On the left of the Figures above are shown the multiple spectrums of the packets received by CAN from 50ms to 1ms. It can be seen that the vast majority of the packets performs as expected even though some of the packets arrive with a significant delay. Analyzing the right graphs of the Figures we can see that 95% of the packets arrive with the delay that was expected in [section 3.4.2](#) by [36] where it is stated that the median of the results is  $67\mu s$  and in 95% of the cases is below  $307\mu s$ . From 50ms to 5ms the delays are very similar but as we get to the lower values of 1ms, even though the delays from all the packets as a whole are compacted in a smaller range (as can be seen by the presence of a value for the 99% of packets) which might happen because the scheduler from the OS needs to schedule this task more often, the accuracy of the timestamps degrades probably due to having the controller operating at a more demanding rate.

For Ethernet we can observe some similarities with CAN in the way that the spectrum evolves from 50ms to 1ms. The 50, 10 and 5 ms graphs are very identical and the accuracy worsens at 1ms, just like in CAN, which allows the same conclusions to be drawn. Although the major difference is in the reliability of the timestamps for the ethernet interface. As we can see, both by a cleaner spectrum and by the presence of 99% values which are very close to the 95% ones, the time stamping of the Ethernet is almost guaranteed to happen with an accuracy higher than  $30\mu s$  for intervals higher than 5ms or  $160\mu s$  for intervals around 1ms. This is almost 2 times more than the accuracy that was predicted in [section 3.4.2](#) by [36].

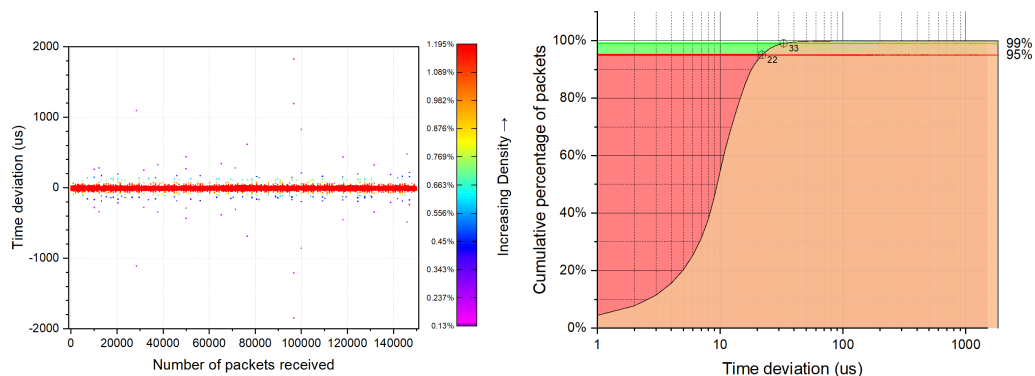


Figure 4.9: Ethernet results for 50ms

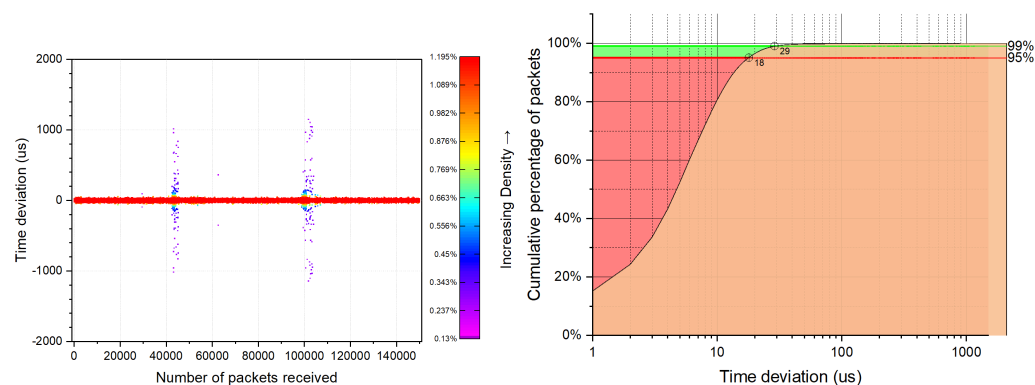


Figure 4.10: Ethernet results for 10ms

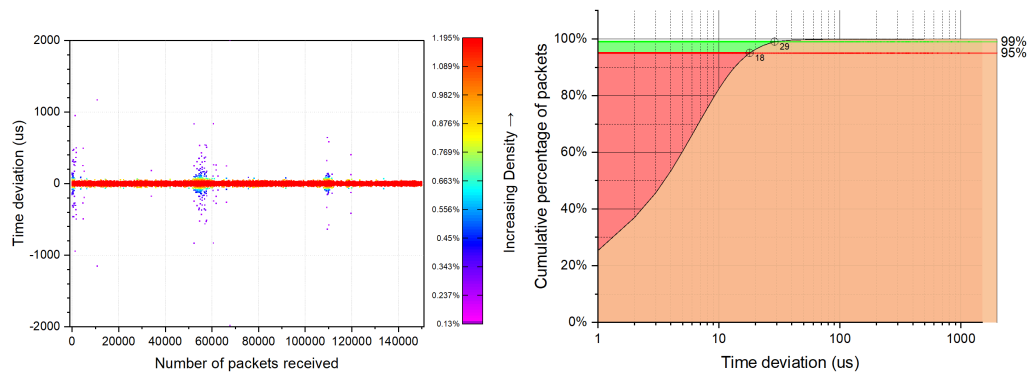


Figure 4.11: Ethernet results for 5ms

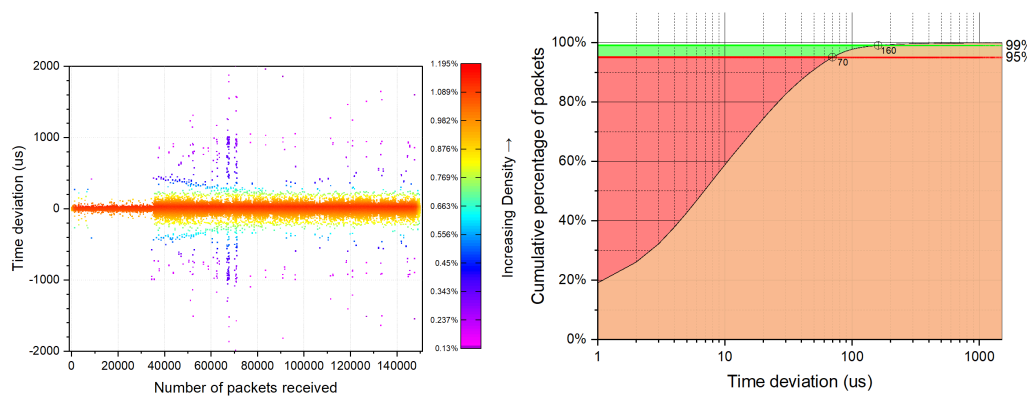


Figure 4.12: Ethernet results for 1ms

This big difference between the behaviour of time stamping in CAN and Ethernet might happen due to the CAN shield not being embedded in the circuit board of the Raspberry Pi 4, therefore having 2 connections before the timestamp is applied to the packet (the CAN connection and the SPI connection) and because of how the OS handles the communications from the Ethernet hub and the extension pins.

With the results from this test we can conclude that the *CanEthAnalyze* performs as good or even better than expected in [section 3.4.2](#) for the range of values in which it will operate (higher than 1ms [Section 3.1](#)) and that arguably it could operate with constrains lower than 1ms.

### 4.2.3 Packet size stress test

In [Section 4.2.2](#) we tested the *CanEthAnalyze* for multiple timing constrains but for Ethernet the packets were always only 100 bytes long. Since we got satisfactory results we are now going to take the strictest test from time test (loops of 1ms) and vary the size of the packet's payload. For different sizes the same parameters as in [Section 4.2.2](#) will be evaluated. The largest protocol data unit (PDU) that can be communicated in a single Ethernet II packet transaction is 1500 bytes. Therefore the test was made for packets with 100, 200, 400, 800 and 1500 bytes. Similarly as before in the left side of the following images is the spectrum for all the received packets and on the right is the cumulative percentage of packets received relative to its time deviation.

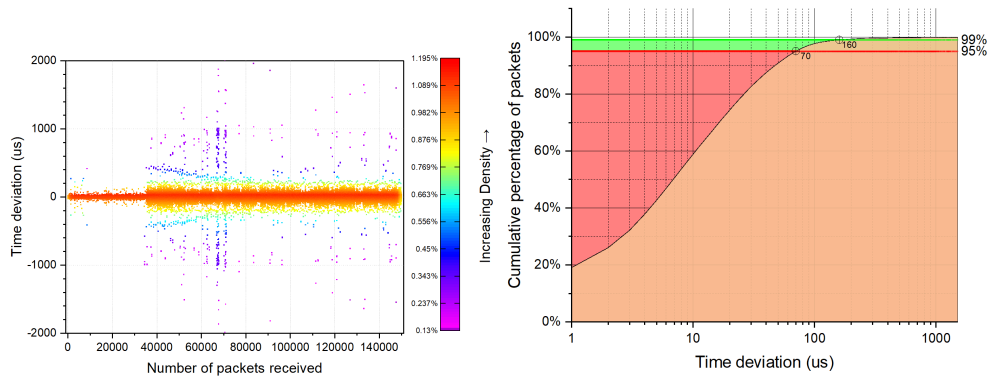


Figure 4.13: Results for 100 bytes on each packet

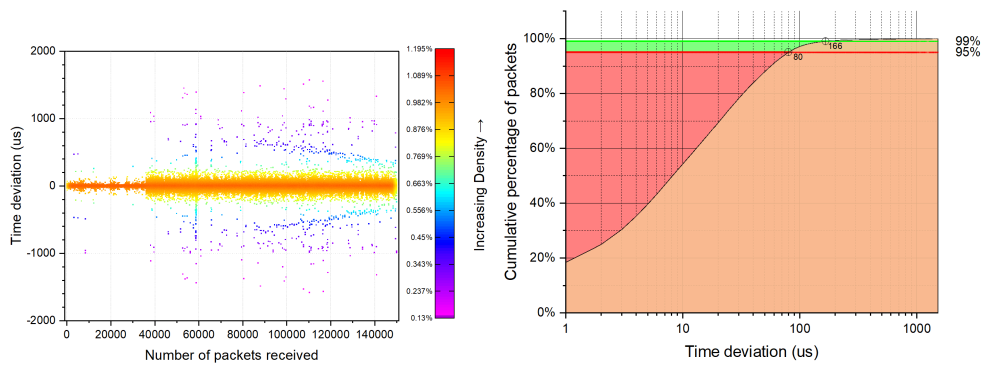


Figure 4.14: Results for 200 bytes on each packet

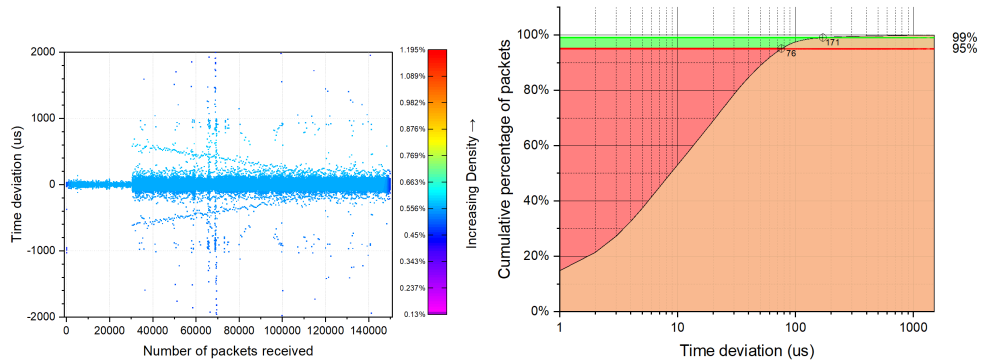


Figure 4.15: Results for 400 bytes on each packet

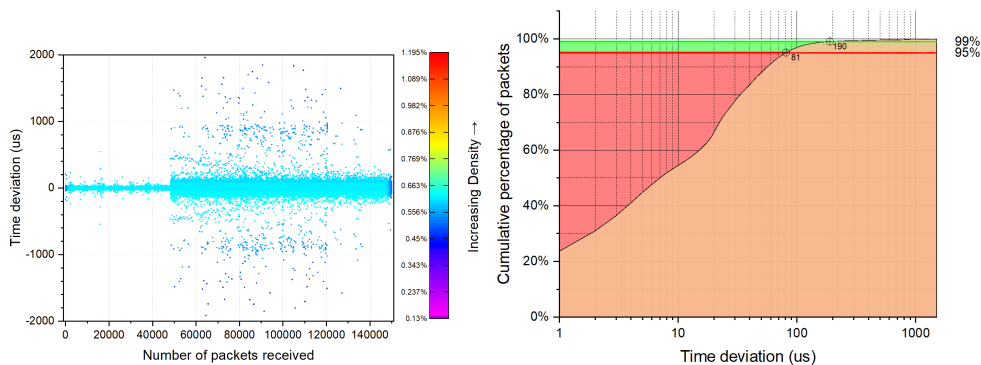


Figure 4.16: Results for 800 bytes on each packet

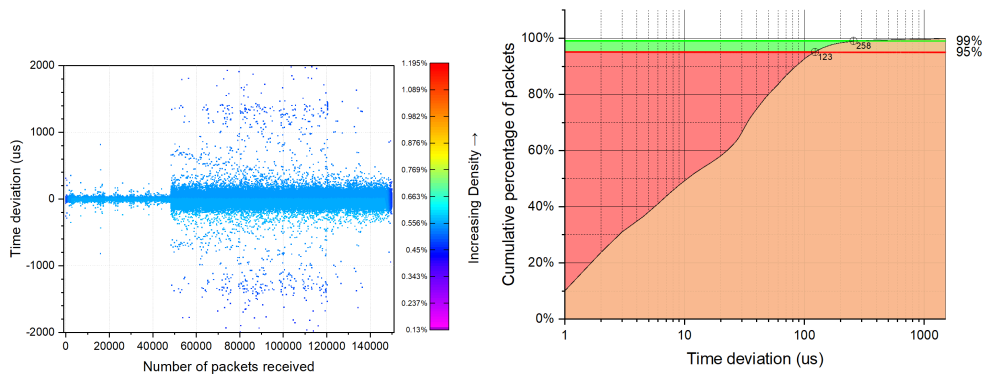


Figure 4.17: Results for 1500 bytes on each packet

As we can see in the graphs above, the time deviation between the packets gradually increases from  $70\mu\text{s}$  for packets with 100 bytes to  $123\mu\text{s}$  (1500 bytes) for 95% of the packets. This is still a very low value which falls in way less than half of the delay that was initially predicted in [section 3.4.2](#) by [36]. Although, even with the processing power of Quad core Cortex-A72 (ARM v8) that runs at 1.5GHz on the Raspberry Pi 4 we can still see that the amount of data that needs to be processed in every packet influences the accuracy of the timestamps.

With this last test we can conclude that all the tests pass the requirements that were proposed in [Section 3.1](#) based on [36], in terms of accuracy by providing more arguments, this time regarding processing stress, which confirm that the *CanEthAnalyz* can perform with an accuracy lower than the proposed  $100\mu\text{s}$  for the range of values in which it will typically operate (higher than 1ms as referred in [Section 3.1](#)). As for the reliability which can be evaluated by analysing the values of the 95% and 99% of the packets we conclude that even though it is within the expected range by [36] ( $307\mu\text{s}$ ) for Ethernet, the CAN side could still be improved a lot if we could control how the OS handles the communication with the external pins, by, for example using a real time OS.

### 4.3 Performance Evaluation

From the results shown in this chapter it is possible to conclude that, regarding timing and processing constrains, the *CanEthAnalyz* can maintain a good deviation average that completely supports the requirement of average values lower than  $100\mu\text{s}$  but it is still influenced by the scheduler from the OS which hurts its reliability as we can see by the values of the 95% and 99% of the packets that in some situations go substantially above the  $100\mu\text{s}$ . These results align with what was expected and described in [section 3.4.2](#) and that was based on [36] which also show that by changing the OS of the raspberry from Raspbian to a real time these values can be considerably decreased. The results also hint that the bus sniffer could support stricter requirements, but further testing using other testing platforms would be required since CANoe doesn't allow to generate packets with periodicity lower than 1ms. One possible solution for a new testing platform would be to have another Raspberry Pi with a real time distro generating the traffic.

## Chapter 5

# Conclusions and Future work

### 5.1 Conclusions

Due to their powerful strengths, Ethernet on the bandwidth requirements and CAN on the safety and reliability, these two types of networks will still be used for a long time in the future of the Automotive industry. For this reason its important that the bus analyzer uses the last technologies from these networks.

Previously to this dissertation a market research to find a board which could support the requirements of gigabit Ethernet and CAN was conducted and led to a choice of a board (OM14510 - SJA1105TJP) that ultimately could not fulfil its purpose. A lot of progress was made using that platform which included a SJA1105 switch and a LCP1788 $\mu$ C and it can be used in the future if an option to use these chips but in a different board is taken. Unfortunately, in the end it was discovered that this board could not support gigabit Ethernet due to its transceivers, nor CAN because the communication by SPI to the  $\mu$ C was not feasible.

Later in this dissertation a software architecture for a Raspberry Pi 4 + PiCAN2 to create a bus sniffer was proposed and implemented. It uses the Gigabit Ethernet port from Raspberry Pi 4 and a PiCAN2 board connected by SPI as interface for CAN and the software is based on 3 threads, one for each interface (CAN and Ethernet) and another to send the packets to a Desktop by Wi-fi.

In order to evaluate the Bus sniffer, a setup by Vector was used. The combination of the VN5610A + CANoe was used to generate the network both for CAN and Ethernet traffic. The evaluation was focused on the timestamps accuracy of the packets received and comprised two separated metrics: 1) The time between each packet and 2) the size of the packets. The evaluation carried out allowed to conclude that the Raspberry Pi 4 + PiCAN2 can maintain a good accuracy but that is still influenced by the scheduler from the OS which has a negative impact on its reliability.

In the end, the objectives of this dissertation were fully achieved. The final result costs only 300€ which is still less than the OM14510-SJA1105TJP solution (500€) and its definitely less than the currently used solution (2000€+ 7000€) even though this setup has 2 CAN and Ethernet channels compared to only one on the Raspberry setup . There were some difficulties that occurred

during this dissertation caused by the COVID-19 pandemic, which prevented the access to material and there was also the mistake of not double checking the previous work about the OM14510 - SJA1105TJP board and developing code for a platform that in the end was found that didn't meet the needs to accomplish the requirements.

## 5.2 Future work

As future work, there are some improvements to the bus sniffer developed in this dissertation that could be made:

- To improve the accuracy and reliability of the software a real time dual kernel such as Xenomai, as mentioned in [section 3.4.2](#), should be installed in the Raspberry Pi 4;
- Switch the PiCAN2 board for a PiCANFD board to allow the reception of CAN-FD packets and configure the SPi communication to a value higher than 12MHz (to support the maximum bit rate of CAN-FD - 12 Mbit/s);
- To take full advantage of the full-duplex of Gigabit Ethernet, swap the interface to send the packets to the Desktop from Wi-fi to the gigabit Ethernet. To accomplish that a switch would be required to separate the Desktop channel and the Automotive channel.

Also, if in the future it is decided to use the SJA1105 switch and the LPC1788 $\mu$ C with adequate gigabit transceivers it should be made on a custom PCB to take advantage of the  $\mu$ C potential and interfaces and to integrate the Ethernet and CAN transceivers and controllers on the same board.

# References

- [1] Gianluca Cena and Adriano Valenzano. Controller area network: A survey. In *The Industrial Communication Technology Handbook*, chapter 13, pages 13–1 to 13–21. CRC Press, September 2017.
- [2] Iso (international standardization organization). <https://www.iso.org/home.html>. Accessed: 11-02-2020.
- [3] Steve Corrigan. Introduction to the controller area network (can). Technical report, Ti - Texas Instruments, 2016. [Online; accessed 31-January-2020]. URL: <http://www.ti.com/lit/an/sloa101b/sloa101b.pdf>.
- [4] Jason Blackman and Scott Monroe. Overview of 3.3v can (controller area network) transceivers. Technical report, Ti - Texas Instruments, 2013. [Online; accessed 31-January-2020]. URL: <http://www.ti.com/lit/an/s11a337/s11a337.pdf>.
- [5] CiA. Can fd. Technical report, CiA - CAN in Automation, 2019. [Online; accessed 01-February-2020]. URL: <https://www.can-cia.org/can-knowledge/can/can-fd/>.
- [6] Can with flexible data-rate. [Online; accessed July 30th, 2020]. URL: <https://can-newsletter.org/uploads/media/raw/d6554699cf2144326280f2f1555b97ea.pdf>.
- [7] ISO International Organization for Standardization. Iso 11898-1:2015 road vehicles — controller area network (can) — part 1: Data link layer and physical signalling. Technical report, International Organization for Standardization, 2015. [Online; accessed 10-February-2020]. URL: <https://www.iso.org/standard/63648.html>.
- [8] Network basics: Ethernet protocol. [Online; accessed 02-February-2020]. URL: <https://www.dummies.com/programming/networking/network-basics-ethernet-protocol/>.
- [9] ieee.org. Ieee standard for ethernet. In *IEEE Standard for Ethernet*, chapter 13, page 108. AUTOSAR, 2016. doi:10.1109.
- [10] ieee.org. "802.3-2018 – ieee standard for ethernet". In *"802.3-2018 – IEEE Standard for Ethernet"*. IEEE Standards Association, 2018. URL: [https://standards.ieee.org/standard/802\\_3-2018.html](https://standards.ieee.org/standard/802_3-2018.html), doi:10.1109.
- [11] Charles E. Spurgeon. Ethernet: The definitive guide. In *Ethernet: The Definitive Guide*, pages pp. 41, 47. IEEE Standards Association, June 2014.

- [12] ieee.org. "40.1.3.1 physical coding sublayer (pcs)". In *Physical Coding Sublayer (PCS)*. IEEE Standards Association, December 2012.
- [13] Kirsten Matheus and Thomas Königseder. *Automotive Ethernet*. Cambridge University Press, 2014. doi:10.1017/CBO9781107414884.
- [14] L. Lo Bello. "the case for ethernet in automotive communications". Technical report, SIGBED Review Special Issue on the 10th International Workshop on Real-time Networks (RTN 2011), December 2011. [Online; accessed 02-February-2020]. URL: [https://www.researchgate.net/profile/Lucia\\_Lo\\_Bello/publication/241622560\\_The\\_case\\_for\\_ethernet\\_in\\_automotive\\_communications/links/580b307408aecba934fbda29/The-case-for-ethernet-in-automotive-communications.pdf](https://www.researchgate.net/profile/Lucia_Lo_Bello/publication/241622560_The_case_for_ethernet_in_automotive_communications/links/580b307408aecba934fbda29/The-case-for-ethernet-in-automotive-communications.pdf).
- [15] EricHackett. Lin protocol and physical layer requirements. Technical report, TI - Texas Instruments, 2018. [Online; accessed 11-February-2020]. URL: <http://www.ti.com/lit/an/slla383/slla383.pdf>.
- [16] Vector. Most - the high-speed multimedia network technology. Technical report, Vector, 2019. [Online; accessed 11-February-2020]. URL: <https://www.vector.com/int/en/know-how/technologies/networks/most/>.
- [17] National Instruments. Flexray automotive communication bus overview. Technical report, National Instruments, 2019. [Online; accessed 11-February-2020]. URL: <https://www.ni.com/pt-pt/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html>.
- [18] F. L. Soares, D. R. Campelo, Y. Yan, S. Ruepp, L. Dittmann, and L. Ellegard. Reliability in automotive ethernet networks. In *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 85–86, March 2015. doi:10.1109/DRCN.2015.7148990.
- [19] R. Boatright C.M. Kozierok, C. Correa and J. Quesnell. *Automotive Ethernet – The Definitive Guide*. Intrepid Control Systems, 2014.
- [20] Ieee p802.3bp 1000base-t1 phy task force. [Online; accessed August 18th, 2020]. URL: <https://www.ieee802.org/3/bp/>.
- [21] IEEE 802 LAN/MAN Standards Committee. Ieee standard for ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pages 1–5600, Aug 2018. doi:10.1109/IEEESTD.2018.8457469.
- [22] D.A. Abaye. "BroadR-Reach Technology: Enabling One Pair Ethernet.". Broadcom Corp., 2012.
- [23] D. Pannell. "Audio Video Bridging (AVB) Assumptions IEEE 802.1 AVB Plenary.". IEEE 802.1 AVB, 2007.
- [24] Donovan Porter. 100base-t1 ethernet: the evolution of automotive networking. Technical report, Ti - Texas Instruments, 2018. [Online; accessed 03-February-2020]. URL: <http://www.ti.com/lit/wp/szzy009/szzy009.pdf>.



- [25] “ieee 100base-t1 (formerly oabr).” [Online; accessed February 4, 2020]. URL: [https://elearning.vector.com/index.php?&wbt\\_ls\\_seite\\_id=1588373&root=378422&seite=vl\\_automotive\\_ethernet\\_introducion\\_en](https://elearning.vector.com/index.php?&wbt_ls_seite_id=1588373&root=378422&seite=vl_automotive_ethernet_introducion_en).
- [26] “ieee standard for ethernet amendment 1: Physical layer specifications and management parameters for 100mbps operation over a single balanced twisted pair cable (100base-t1).”, 2015. URL: [https://standards.ieee.org/standard/802\\_3bw-2015.html](https://standards.ieee.org/standard/802_3bw-2015.html).
- [27] Vn5610a/vn5640 powerful and multifunctional usb network interfaces for automotive ethernet and can. [Online; accessed February 7, 2020]. URL: [https://assets.vector.com/cms/content/products/VN56xx/docs/VN5610A\\_VN5640\\_FactSheet\\_EN.pdf](https://assets.vector.com/cms/content/products/VN56xx/docs/VN5610A_VN5640_FactSheet_EN.pdf).
- [28] Vn5610/vn5610a ethernet/can interface. [Online; accessed February 7, 2020]. URL: [https://assets.vector.com/cms/content/products/VN56xx/docs/VN5610\\_Manual\\_EN.pdf](https://assets.vector.com/cms/content/products/VN56xx/docs/VN5610_Manual_EN.pdf).
- [29] Vn5610a/vn5640 option 100 base-t1. [Online; accessed February 11, 2020]. URL: <https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vn56xx/#c9340>.
- [30] Vn5610a/switched based universal emc device. [Online; accessed February 11, 2020]. URL: <https://technica-engineering.de/produkt/switched-based-universal-emc-device/>.
- [31] 802.3-2018 - ieee standard for ethernet. [Online; accessed August 3rd, 2020]. URL: [https://standards.ieee.org/standard/802\\_3-2018.html](https://standards.ieee.org/standard/802_3-2018.html).
- [32] Sja1105 application board. [Online; accessed August 18th, 2020]. URL: <https://www.nxp.com/docs/en/user-guide/AH1508.pdf>.
- [33] Sk pang site. [Online; accessed September 9th, 2020]. URL: <https://skpang.co.uk/>.
- [34] Raspberry pi 4 ic. [Online; accessed August 18th, 2020]. URL: [https://en.wikipedia.org/wiki/Raspberry\\_Pi#/media/File:RaspberryPi\\_Model\\_4B.svg](https://en.wikipedia.org/wiki/Raspberry_Pi#/media/File:RaspberryPi_Model_4B.svg).
- [35] Raspberry pi 4 specs and benchmarks. [Online; accessed August 3rd, 2020]. URL: <https://magpi.raspberrypi.org/articles/raspberry-pi-4-specs-benchmarks>.
- [36] How fast is fast enough? choosing between xenomai and linux for real-time applications. [Online; accessed August 3rd, 2020]. URL: <https://pdfs.semanticscholar.org/9eb5/1dbe38fb23034e80b8664d8281996d2a5ef6.pdf>.
- [37] Raspberry pi: Preempt-rt patching tutorial for kernel 4.14.y. [Online; accessed August 3rd, 2020]. URL: <https://lemariva.com/blog/2018/02/raspberry-pi-rt-preempt-tutorial-for-kernel-4-14-y>.
- [38] Ying Li, Bob Noseworthy, Jeff Laird, Timothy Winters, and Timothy Carlin. A study of precision of hardware time stamping packet traces. *2014 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pages 102–107, 11 2014. doi:10.1109/ISPCS.2014.6948700.

- [39] Xenomai page. [Online; accessed August 5th, 2020]. URL: <https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home>.
- [40] A guide to using raw sockets. [Online; accessed August 18th, 2020]. URL: <https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/>.