A Novel Ethernet Network Layer Protocol for Automotive Network Communications

A Degree Project Report in Computer Science and Engineering

EBBA HÅKANSSON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Degree Project Report 2022

A Novel Ethernet Network Layer Protocol for Automotive Network Communications

EBBA HÅKANSSON

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Novel Ethernet Network Layer Protocol for Automotive Network Communications
EBBA HÅKANSSON

Supervisor: Björn Bergholm, Broccoli Engineering
Supervisor: Neethu Bal Mallya, Chalmers University of Technology
Examiner: Lars Svensson, Chalmers University of Technology

Degree Project Report 2022
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

University of Gothenburg
SE-405 30 Gothenburg
Telephone +46 31 786 0000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2022

A Novel Ethernet Network Layer Protocol for Automotive Network Communications
EBBA HÅKANSSON
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This project investigates the possibility of replacing CAN bus with an Ethernet connection by sending CAN messages in Ethernet frames. A new protocol for Ethernet, called *CANEthernet*, was developed for this purpose and a corresponding EtherType is proposed. CANEthernet packs several CAN messages within an Ethernet frame and different design options for the protocol are discussed and compared. The new protocol can carry all types of CAN messages and does not change the contents of them. CANEthernet is evaluated by comparing it to using a regular CAN bus in regards to throughput, latency and energy efficiency. The evaluation shows that CANEthernet outperforms regular CAN in all evaluated aspects. For throughput CANEthernet can reach 9Gbps over 10Gbps Ethernet, while the fastest CAN throughput is 8Mbps. In latency, CANEthernet can transport a CAN message in 11.2 us while CAN requires 108 us for sending the same CAN message. Finally, CANEthernet uses 10.8 nJ per bit while CAN uses 153 nJ per bit.

Keywords: CAN, CAN FD, Ethernet.

Acknowledgements

This was a degree project for a Degree of Bachelor of Science in Computer Engineering at Chalmers University of Technology. It was written in collaboration with the company Broccoli Engineering. I want to thank my supervisors Neethu Bal Mallya and Björn Bergholm, for helping me.

Ebba Håkansson, Gothenburg, June 2022

List of Acronyms

ACK	Acknowledgment
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-rate
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
EOF	End of Frame
FCS	Frame Check Sequence
IEEE	Institute of Electrical and Electronics Engineers
IFS	Inter Frame Space
IPG	Inter Packet Gap
MAC	Media Access Control
OSI	Open Systems Interconnection
RTR	Remote Transmission
SFD	Start Frame Delimiter
SPI	Serial Peripheral Interface
SOF	Start of Frame

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Today's vehicles are, to a large degree, controlled by computers. A vehicle contains several Electronic Control Units (ECUs). Each ECU is responsible for different parts of the vehicle, such as the engine, steering, and driver controls. Robust and efficient communication between the ECUs is essential for a functioning vehicle. The number of ECUs in vehicles has increased significantly over the last three decades, and with that, the amount of network communication within a vehicle has grown rapidly[2]. Because of this, automotive manufacturers can save a lot of money and resources by making communication within a vehicle more efficient.

1.2 Purpose

The project is intended to determine if communication between nodes in a vehicle is more efficient by using Controller Area Network (CAN) messages or packing several CAN messages into one Ethernet message. The scenario imagined for this project is two nodes within a vehicle that both have a CAN and an Ethernet connection. The nodes need to communicate using CAN messages. This project will help determine whether it is more efficient to use the CAN connection or send the CAN messages over Ethernet. In order to make the Ethernet communication as efficient as possible, a custom EtherType will be created to stow CAN messages.
If the communication is found to be more efficient via Ethernet, the CAN connection could be removed. Such change could lead to thousands of cars being produced with fewer resources and would positively impact the ecological footprint of vehicle production.
The project was done in collaboration with the consulting company Broccoli. The company has many consultants working in the automotive industry, so a project with ties to this industry was appropriate.

1.3 Goal

The project aims to determine the most efficient communication method between nodes in a vehicle - either (i) using CAN messages or (ii) packing several CAN messages into one Ethernet message. The study will look at the current state of vehicle network communications and recent research. The evaluation will primarily

compare the efficiency of the two communication methods in terms of throughput, latency, transmission speed, and amount of overhead. Each method's cost efficiency, energy consumption, ecological footprint, and reliability will also be examined.

1.4 Limitations

Due to time constraints, the project will have some limitations. It will not consider the security aspects of network communication. Security is crucial in designing vehicles, and it would be unethical to develop the internal network without considering it. However, it does not fit within the time frame for this project. Another constraint is that no hardware development will be done; all technical development will only be done with the software. This project will also not consider other types of communication than CAN and Ethernet.

1.5 Thesis Organization

Chapter 2 discusses the plan made for carrying out the project, followed by Chapter 3, which contains the relevant theory on CAN and Ethernet. Chapter 4 describes the methodology of how the new protocol was developed. It discusses the various design aspects, why certain solutions were chosen, and details of the final protocol design.

Chapter 5 contains the hardware and software implementation, including the used hardware, its setup, and the software library created for the project. The chapter also explains how the tests were executed. Following that, Chapter 6 contains the evaluation of the protocol, which compares it against CAN for throughput, latency, and energy efficiency. Finally, Chapter 7 concludes the report with an analysis of the project plan vs. execution and potential future work.

2
Method

The following steps outline how the project will be carried out:

Hardware Setup and Configuration: The first step in the project will be to set up the hardware. The project will use two Arduino boards, two CAN network cards, and two Ethernet network cards. The Arduino boards will be connected together with both a CAN and an Ethernet connection. The hardware will then be configured to send and receive messages. After this, several CAN and Ethernet testing messages will be created. The messages will then be used to test the configuration.

Research: The next step is to research CAN and Ethernet. This phase will begin during the hardware phase while waiting for ordered parts. The research will include specifications of the different protocols used and variations of the protocols. It will also include information on how EtherTypes work and how messages with a custom EtherType can be transmitted and received with the provided hardware.

Development of the proposed EtherType: When the research is finished, the development of the new EtherType will begin. The details of the EtherType as well as the method of storing several CAN messages in an Ethernet message will be calculated to achieve the highest efficiency.

Verification of the proposed EtherType: When the new EtherType has been developed, the project will move to a testing phase. A number of test CAN messages will be created. The messages will then be used to verify that the Ethernet messages are sent and received correctly. To do this, a system for doing the verification will be created.

Evaluation: The final step of the project will be to perform tests and calculations that compare sending CAN messages with sending CAN messages using Ethernet. The two ways of communicating will be compared on several criteria such as time and cost efficiency, ecological footprint, energy consumption and reliability. The evaluation will draw conclusions of what type of communication is appropriate for different situations.

Documentation and Final Report: All the essential information related to the project will be documented throughout the different steps. A final report and a presentation will be made available towards the end of the project.

3
Theory

This chapter describes the hardware and the communication protocols used for the project.

3.1 CAN

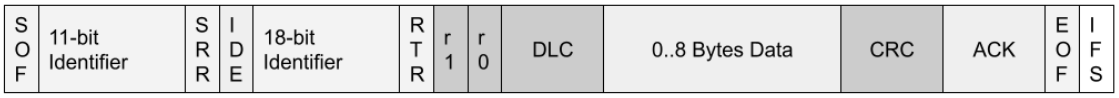
CAN is a communication bus developed in 1986 for use in the automotive industry. Today CAN is used in a wide range of applications, but it is still very commonly used for automotive applications. The protocol is suitable for automotive applications because of its robustness and efficiency. In the 1990s, the increased number of ECUs in vehicles led to a rapid increase in the use of CAN networks within the automotive industry. Manufacturers chose the CAN communication bus for its efficiency and low cost [2]. A new version of CAN, called Controller Area Network Flexible Data-Rate (CAN FD), was developed in 2012. The amount of CAN FD in vehicles has increased since then. CAN was not developed for sending large amounts of data. It is designed for quickly transmitting messages between several nodes and for its real-time properties, quick error detection, fast error recovery, and short reaction times [2] [4].

Nodes on the CAN bus are connected with a two wire bus. The bus can be at one of two states, either recessive or dominant. The base state on the bus is recessive, and it only becomes dominant if at least one node transmits a dominant value. In many parts of CAN messages it is not allowed to send more than five bits of the same value consecutively. To prevent this from happening the transmitter inserts a stuff bit of the opposite value when there are six or more bits of the same value consecutively.

CAN FD differs from classic CAN in that it can send more significant amounts of data per message. Unlike classic CAN, CAN FD can send different parts of a message at different bit rates. With CAN FD, the data field of a message can be sent at up to 8 Mbps, while the rest of the message can be transmitted at up to 1 Mbps. Standard CAN has a maximum speed of 1 Mbps.

Apart from these differences, the two protocols are very similar. There are three types of frames in CAN FD: Data frame, Error frame, and Overload frame. Classic CAN has the same frame types plus a Remote frame. The message types are structured similarly for both protocols. This report does not require detailed knowledge

(a) Classic CAN Data frame



(b) Extended CAN Data frame

(c) Classic CAN FD Data frame

(d) Extended CAN FD Data frame

Figure 3.1: Different Data Frames

of how CAN messages are structured. Unless specified, the other chapters of this report will use the term CAN to refer to both CAN and CAN FD messages. Below is a basic description of the different types of CAN and CAN FD frames.

3.1.1 Data frame

CAN Data frames are used to transmit data. Figure 3.1a shows the fields of a classic CAN Data frame. The classic CAN Data frame has an 11-bit identifier field. There is also an extended CAN Data frame with a 29-bit identifier. The extended CAN data frame is shown in Figure 3.1b. Apart from the identifier, the classic and extended CAN data frames are identical. The identifier field usually contains the identity of the transmitting node. It also sets the priority of the message, which decides which message takes precedence when collision occurs on the bus. This is done by the nodes reading the value of the identifier field on the bus as they transmit, and when noticing a higher value than the one being transmitted by themselves they end the transmission. In essence this means that the node transmitting with the highest value of the identifier field takes precedence. The identifier field can also transmit data, which allows the message to send more than 64 bits. Storing data in the identifier field is especially common in the extended format.

The CAN data frames can contain up to 8 bytes of data, and the DLC field indicates the amount of data in a message. Each CAN message also has a Start Of Frame (SOF) to mark the beginning of a message, an Acknowledgement Field (ACK), a

checksum field (CRC), and an End Of Frame (EOF) to mark the end of the message. The Inter Frame Space (IFS) at the end of each message is 3 bits long [2].

The CAN FD data frame also has a classic frame with an 11-bit identifier, shown in Figure 3.1c and an extended version with a 29-bit identifier, shown in Figure 3.1d. The figures show that CAN and CAN FD data frames are similar, but they have important differences. One main difference is that the CAN FD data frame can send data between 0 and 64 bytes. This means that a CAN FD data frame can be significantly larger than a CAN data frame. Another difference is that CAN FD can send data messages with two different predetermined bit rates. The CAN FD message contains a flag bit before the data field begins, which sets at which of the two bit rates the rest of the message will be sent. With this format, a message can be transmitted with different bit rates for the identifier field and the data field [4].

The CRC and ACK fields are used in the same way for all Data frames. The CRC field contains the result of the cyclic redundancy check calculation. CAN and CAN FD use this field to ensure that the message is correctly built and received by other nodes. Within the ACK field, there is a 1-bit ACK slot set to recessive (1) by the transmitting node. If the receiving node has a successful CRC check, it forces the bus to be dominant (0) during the ACK slot. If the transmitting node reads a dominant (0) signal from the bus at the ACK slot, it knows that the other node has received a correct CRC.

3.1.2 Remote frame

Remote frames are used for requesting data from a node, and they are very similar to data frames. CAN FD does not have a remote frame. There are two important differences between CAN data and remote frames. First, the RTR-bit is set to recessive(1) as opposed to dominant(0) as it is in a data frame. The main difference is that a remote frame has an empty data field. The data length content field now contains the size of the requested data.

3.1.3 Error frame

The Error frame is sent by nodes when they discover an error on the bus. The error frame is identical for both CAN and CAN FD. The first field of the Error frame is the Error flags which consist of 6-12 identical bits that are either dominant(0) or recessive(1). The value of the error flags indicates the type of error. Following that is the error delimiter, composed of 8 recessive(1) bits. This leads to the Error frames size between 14-20 bits long.

3.1.4 Overload frame

The Overload frame is sent by a node that is overloaded and can not participate in more communication. An overload frame can also be used to report some types of errors. The frame is identical to the error frame. CAN receivers can differentiate

Field	Preamble	SoF	MAC destination	MAC source	(802.1Q tag)	Ethertype/ length	Payload	FCS	IPG
Size (bytes)	7	1	6	6	4	2	46-1500	4	12

Figure 3.2: Fields of an Ethernet message

between the two because error frames are sent while a message is transmitted on the bus, while overload frames are sent between messages when the bus is empty. Another difference is that error frame causes re-transmission of the previous frame, while overload frames do not.

3.2 Ethernet

CAN dominates the communication between ECUs in today's vehicles, but Ethernet has emerged as a replacement for automotive communications. The number of connected ECUs within a vehicle and the overall amount of transmissions on automotive networks are increasing. Ethernet may be a good alternative for handling a large amount of data transferred [7] [3].

Ethernet is a collection of standards for computer communications. It covers the physical and the data link layer in the Open Systems Interconnection (OSI) model. All fields of an Ethernet message are shown in Figure 3.2. The 7-byte preamble is used for synchronization and is followed by a 1-byte Start Frame Delimiter (SFD) which indicates the beginning of the message.

Ethernet uses Media Access Control (MAC) fields which contain 6-byte addresses for the transmitting and the receiving node. Following that is an optional field for an 802.1Q tag, which will not be used in this project. The next 2 bytes contain a value. If the value is over 1536, the field contains the EtherType of the message. If it is under or equal to 1500, the value represents the size of the payload. The meaning of the EtherTypes is explained in the next chapter.

The payload of an Ethernet message is 46-1500 bytes long. After the payload there is a 4 byte long Frame Check Sequence (FCS). The FCS is a cyclic redundancy check which allows the receiver to confirm that the received data is not corrupted [11]. The protocol does not state how a receiver should act if it detects a corrupted frame. The message ends with a 12-byte long Inter Packet Gap(IPG) [5].

The EtherType of an Ethernet message implies which protocol is used within the Ethernet packet so the receiving node can correctly interpret the message. EtherTypes are registered by the Institute of Electrical and Electronics Engineers (IEEE) Registration Authority. The current list of registered EtherTypes contains registrations from private companies and non-profit organisations such as the IEEE [6].

4

Development

The development of a new protocol, called CANEthernet, was the main goal of this project. The protocol should be placed over Ethernet, meaning all data concerning the new protocol will be placed in the 46-1500 byte payload field in the Ethernet message. The EtherType for this new protocol must have a value over 1536, and it can not be an already registered EtherType. The new EtherType was chosen to be 13090, corresponding to the hexadecimal value 0x3322.

The protocol is created to store several CAN messages within an Ethernet message. Additionally the protocol describes how to order the data in the Ethernet message payload to make it as efficient as possible. Efficiency has two sides. First, the more CAN messages that can fit within one Ethernet payload, the more time is saved, and the throughput increases. Second, the data must be structured to make unpacking it as simple as possible for the receiver of the message. In creating this protocol, both aspects of efficiency have been taken into consideration.

Due to a lack of data, some assumptions and decisions had to be made without any exact data to point to. This was mainly the case with the sizes of the CAN messages. There was no knowledge of the average sizes of CAN messages, and therefore the only data that could be used was the possible message sizes.

Another aspect of this was the lack of knowledge on whether all types of CAN messages would be sent using this protocol. For example, had it been known that Error and Overload messages would not be sent with this protocol, it may have been designed differently.

In order to handle this lack of data, the development focused on making the protocol as broad as possible, to allow for all types and sizes of CAN messages.

This chapter describes the different aspects needed to be considered when creating this protocol. It describes different solutions for each aspect that was considered when designing the protocol. In this chapter, CAN refers to both standard CAN and CAN FD.

4.1 Minimizing CAN message size

One of the aspects of creating the protocol is whether it is possible to decrease the sizes of the CAN messages by removing bits. Making the CAN messages smaller would lead to a larger number of them fitting in a single Ethernet message, which would increase speed and throughput.

In theory, this would mean that the transmitting node would remove bits from the CAN messages, which would then be added again at the receiving node. It would require clear rules for which bits would be removed, so that the receiving node knows what bits to add. However, this was not used in the CANEthernet protocol due to two reasons.

The first reason was that there is a large difference in structure between different types of CAN messages. The Error and Overload frames share almost no similarities to the Remote and Data frames. There would need to exist an indicator for each message indicating which type it is. Each type of message would need its own set of rules for which bits to remove and add. This would make the protocol significantly more complicated and challenging to implement and use. In addition, if new types of CAN message types are introduced in the future, the CANEthernet protocol would have to be updated.

The second reason was that the technique of removing bits requires that the CAN messages are correctly built by the sender. If the CAN messages can be faulty, it is impossible to ensure that the correct bits are added by the receiver. CANEthernet is a protocol and can not check that CAN messages are correctly built. That responsibility would fall on the sender. CANEthernet is created to be a general protocol and is not made for a specific use. Therefore, it was decided not to trust that the sender only sends correct CAN messages, since it is unknown what type of transmitter will be used.

Due to the reasons described, the CANEthernet protocol does not use any bit removal to decrease the CAN message sizes.

4.2 Finding beginning and ending of each CAN message

In order for the receiving node to extract the CAN messages from the CANEthernet message, it needs to know when each CAN message begins and ends. This can be done by finding bit patterns that indicate message ending, by parsing the CAN messages and using the contents to calculate its size, or by including a field with the size of each CAN message in the protocol. This section describes several different solutions.

4.2.1 Bit pattern

The first possible solution is for the receiver to identify a bit pattern that indicates when each CAN message ends. That requires there to be such a pattern for all CAN messages. This pattern could be naturally occurring within the CAN message itself. All types of CAN messages ends with 8 recessive(1) bits. While this may seem like a possible bit pattern to use, it is not due to the possibility that the same pattern occurs at other places in CAN messages. The bit pattern could also be added at the end of each CAN message by the transmitting node. However, the same problem occurs since all bit patterns could occur within a CAN message.

Even if there was a bit pattern that could be used to denote the end of a CAN message, it would not be possible to use. As explained in a previous chapter, there is no guarantee that the CAN messages are correctly built. Because of this, there is no possible way to ensure that a particular bit pattern does not occur in a CAN message. Because of the explained reasons, the CANEthernet protocol does not use bit patterns to identify the ending of the CAN message.

4.2.2 Parsing CAN messages

Another solution could be to use the data within the CAN messages to find the size of them. CAN messages of data and remote type have data that indicates whether it is extended or not, if it is CAN FD, and the size of the data field. With this information, it is possible to know the exact size of each message. However, this is not possible to use because the Error and Overflow messages are distinctly different from the data and remote messages. The size of Error and Overflow messages is not set and can not be obtained from information within the messages. In addition, parsing messages could be CPU intensive and increase energy consumption. Because of this, using the internal data of the CAN messages is not an option for finding the ending of the CAN messages in the CANEthernet protocol.

4.2.3 Size field

The last solution is to include a field per CAN message that indicates the size of the CAN message. This solution makes it possible for the receiver to find each CAN message without reading any data in the CAN messages. This is the solution chosen for the CANEthernet protocol. For this solution, it may be beneficial to pad out the CAN messages, because that may lead to shorter CAN-size fields. The following chapter describes padding the messages and the calculations for the best amount of padding.

4.2.4 Padding messages

In this instance, padding a CAN message means adding bits after each message bit to increase the size of the message to the desired size. The CAN messages could be padded out to any size, but this project only considers no padding or padding to multiples of 8 bits (bytes). In order to lower the complexity of the protocol, this

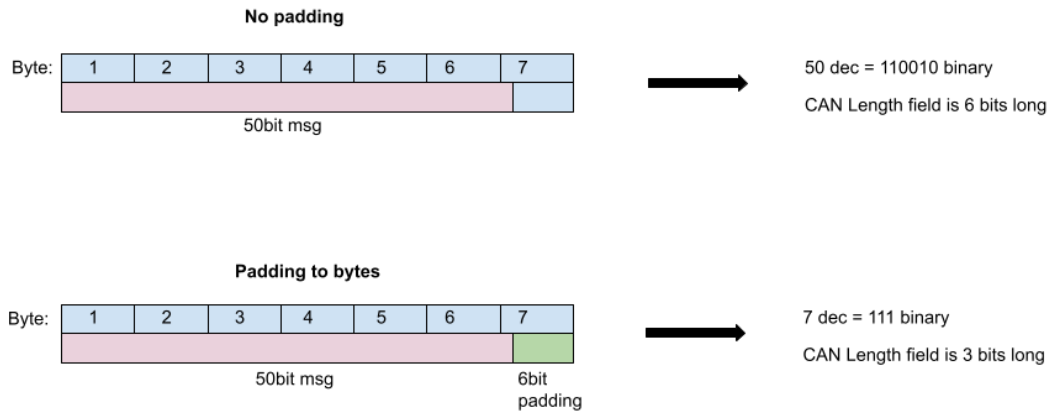


Figure 4.1: Example showing how padding effects size of CAN size field

project only looked at the case where all CAN messages were padded in the same way.

Figure 4.1 shows an example of how padding a 50-bit long CAN message to full bytes can make the CAN-size field shorter. Without padding, the CAN-size field would need to contain the number 50, which requires a CAN-size field of 6 bits. If the message is padded out to full bytes, it becomes 7 bytes long. To describe 7 bytes, the CAN-size field only needs 3 bits. This means that the size field becomes 6-3=3 bits shorter when padded to bytes. However, the downside to padding is the space needed for the padding. When the CAN message was padded to full bytes, 6 bits were added to its end. When considering this, the padding did not save space in this instance. Whether padding saves space or not is dependent on the size of the message and the size of the padding.
Because of the lack of data on the prevalence of the different sizes of CAN messages, it is not possible to calculate the ultimate amount of padding for this protocol. In the absence of the data, it was chosen that the CANEthernet protocol will pad out CAN messages to full bytes. Designing the protocol to include padding will make it easier to change the protocol to a different type of padding if needed.

When a CANEthernet message with padded CAN messages arrives, the receiver must find where the CAN message begins and the padding ends. This can be solved by the knowledge that all types of CAN messages end on a recessive(1) bit. If the padding is done with only dominant(0) bits, the receiver can find the spot where there is a recessive(1) bit by going backward through the padded message. Since it needs to find the end of each CAN message, the work that the receiver needs to do increases. In this project, it is not possible to estimate the amount of work needed by the receiver, and it is therefore not considered in the protocol creation.

To calculate the needed size of the CAN-size fields the largest possible CAN message must be considered (596 bits). When padded out to full bytes, the message becomes 75 bytes. To describe 75, the CAN-size fields for the CANEthernet protocol are 1 byte. This also provides the possibility to handle an increase of maximum CAN messages sizes as the size field allows up to 255 bytes.

4.3 Indicating end of the message

The receiving node has to know when the message ends, i.e., when all CAN messages have been received. Similarly to the previous section, it can be done either by the receiver parsing the message to find a bit pattern that indicates the ending. It can also be done with a field in the CANEthernet protocol header indicating either the number of CAN messages or the size of the whole CANEthernet frame.

4.3.1 Bit pattern

In the case of the receiver parsing the message and looking for the specific bit pattern, there needs to be a distinct pattern that indicates the end of the whole message. This pattern could be added to the message by the sender. But, as explained in the previous chapter, there is no bit pattern that could not occur in a CAN message. Therefore, it is not possible to use a bit pattern to identify the end of the message.

4.3.2 Size field

The next option is a field in the CANEthernet header indicating the size of the whole message. In this case, there may be padding done to decrease the size of this field. An example of how padding could work is shown in Table 4.1. The table shows how long the size field would be for a 4702-bit long message when sent with 64-bit padding and without padding. Without padding, the size field is 13 bits, making the total number of bits 4715. If padded to 64 bits, the padding would be 34 bits which would make the message 4736 bits long. The size field has to represent 4736/64 = 74, which is sent in binary and requires 7 bits. This makes the total number of used bits for the 64-bit padded message 4743 bits. In this example, it would be more efficient not to pad the message as compared to padding it to 64 bits but the difference is very small. However, it is not possible to determine the most efficient padding with the current known data, as was explained in the previous chapter.

Padding	Full msg size	Size-field size	Total used bits
None	4702 bits	13 bits	4715 bits
To 64bit	4736 bits	7 bits	4743 bits

Table 4.1: Example: 4702-bit message with 64-bit padding and without padding

4.3.3 Number of CAN messages

Another option is to have a field in the header that indicates how many CAN messages are in the CANEthernet message. With this information, the receiver knows when it has extracted all CAN messages. The size of this field depends on the number of CAN messages in the message. The minimum amount of CAN messages is always 1, but the maximum number of CAN messages depends on how the other parts of the protocol is designed.

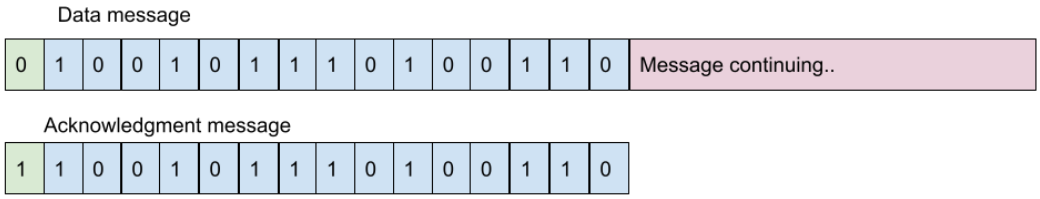


Figure 4.2: Example showing how data and acknowledgment message are built

Both the number of CAN messages field and the size field were possible solutions, and there was no data to calculate which would be more efficient. The option of a field with the number of CAN messages was chosen for CANEthernet because of its simplicity.

4.4 Acknowledgment

The CAN protocol uses cyclic redundancy checks to confirm that a correctly built message has been received. During the transmission, the receiver confirms that the message is correct by forcing a dominant (0) bit at a certain place. Ethernet does not have any acknowledgment. Because it is designed to replace CAN transmissions, the CANEthernet protocol should use some type of acknowledgment to achieve the same message security as CAN. Cyclic redundancy checks are an extensive and complicated area. Since this was not the focus of this project, it was not implemented in CANEthernet.

A simpler way of acknowledging that a message has been received is to answer with a type of acknowledgment message when transmission has been received. The CANEthernet implements this by having a 2-byte long acknowledgment field at the beginning of the CANEthernet header. The first bit of the acknowledgment field indicates if the message is an acknowledgment message. The bit is set to 0 for data transmissions and 1 for acknowledgment transmissions. The remaining 15 bits of the acknowledgment field are a number chosen by the sender of a data message. The receiving node then answers with an acknowledgment message, with the first bit of the acknowledgment field set to 1 and the remaining bits the same as the received messages acknowledgment field.

Figure 4.2 shows an example of what the acknowledgment field could look like. The first message is the data message which has the first bit set to 0. The second message has the same acknowledgment field with the first bit set to 1. This message does not contain any data, it is only meant to show that the data message was received.

The solution of using an acknowledgment field and message only notifies the sender that a message with that acknowledgment field has been received. It does not check that the message is correctly built or that the correct data was received.

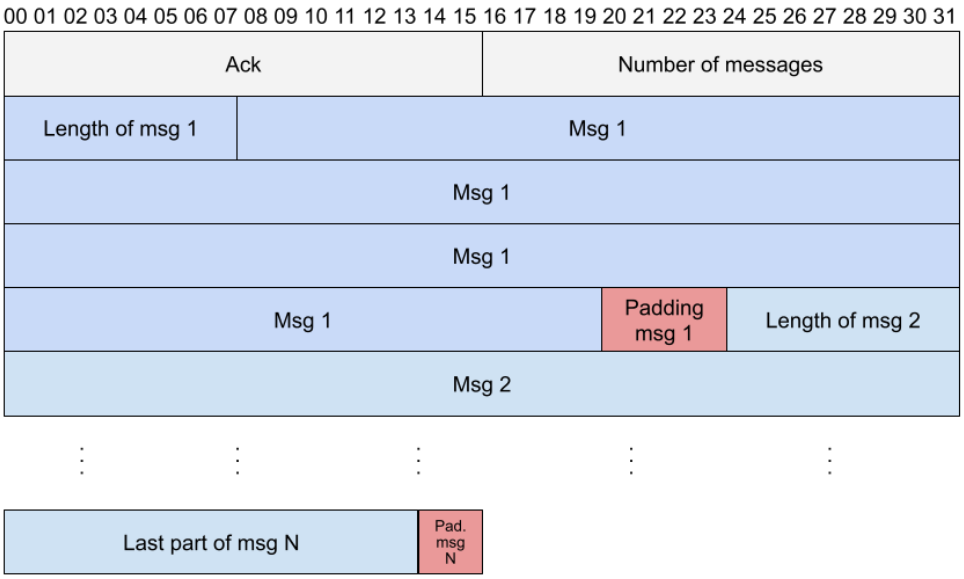


Figure 4.3: Overview of the CANEthernet protocol

4.5 The CANEthernet protocol

This section describes the design of the created CANEthernet protocol. The CANEthernet protocol has fields for acknowledgment, number of CAN messages, and size of the CAN messages.

4.5.1 Finding beginning and ending of each CAN message

The CANEthernet protocol uses a one byte long field per CAN message indicating the size of the CAN message. All CAN messages in the protocol are padded to whole bytes and the padding is done with 0s.

4.5.2 Indicating end of the message

To find the end of a message the CANEthernet protocol includes a field indicating the number of CAN messages stored in the CANEthernet message. The size of this field depends on the other parts of the protocol. Because it has been decided to use 1 byte for message size per CAN message and 2 bytes for acknowledgment, the maximum number of CAN message (N) is calculated with: 3 b

$$N = \lfloor \frac{MaxSpace - 2}{minCAN + 1} \rfloor$$

where *MaxSpace* is the largest possible size of the message (1500) byte) and *minCAN* is the smallest possible byte padded CAN message (2 byte).

The calculation does not take into account the size of the number of messages field, since it has very little impact. Calculating the formula with the known numbers gives that the maximum number of CAN messages *(N)* is 499. To represent this number the protocol needs 9 bits. For simplicity the field is extended to be two full bytes long.

4.5.3 Acknowledgment

Each message in CANEthernet begins with a 2 byte long acknowledgment field. A node sends a data message with the first bit of the field set to 0 and the rest of the field as a random number. When a node receives a message it responds with an acknowledgment message, which only contains the acknowledgment field. The field is identical as in the received message except for the first bit which is set to 1.

4.5.4 Order of protocol fields

In order to make the acknowledgment messages as short as possible, it is the first field in the protocol. This way the acknowledgment message only has to contain the two byte acknowledgment field. Following that is the two byte long field indicating the number of CAN messages. After that comes the 1 byte field for the CAN size of message 1, followed by CAN message 1. Next is the next CAN size field followed by CAN message 2. The rest of the CAN messages are stored the same way. An overview of this can be seen in Figure 4.3.

5

Implementation

The implementation of the project included setting up the hardware using Arduino boards and creating a reusable software library to enable the testing. The requirement included (i) sending a CANEthernet message containing CAN messages, (ii) receiving the CANEthernet message, (iii) unpacking the CAN messages from the CANEthernet message, and then (iv) sending the unpacked CAN messages over a CAN bus. The CAN messages must be received and checked to match the original CAN messages from the CANEthernet message. The following sections discuss the chosen hardware setup and the software library.

5.1 Hardware Setup

This project used two Arduino UNOs[1] for testing the created protocol. Arduino UNO is a board that uses the ATmega328P microcontroller. The Arduino board is programmed using C++ with the addition of some Arduino-specific functions. The project also used Ethernet modules of the model ETH Click[10] from Microe. ETH Click has an Ethernet controller and uses the SPI serial interface. Modules of model CAN Bus Click[9] from Mikroe were also used. The CAN Bus Click module has an MCP2515 CAN controller from Microchip and a CAN transceiver. It uses the SPI serial interface. The project used Arduino UNO click shields[8], an extension to the Arduino UNO board that makes connecting the Click-modules easier.

Each Arduino was equipped with one Ethernet module and one CAN module to do this implementation. The Ethernet modules were then connected to each other, and the CAN modules were connected to each other as well. Figure 5.1 illustrates the setup, and Figure 5.2 shows a photo of the actual hardware setup. The thought behind this setup was that one Arduino would create a CANEthernet message that contains CAN messages. This Arduino would then send the CANEthernet message to the second Arduino via an Ethernet connection. The second Arduino would unpack the CAN messages and send them via the CAN connection to the first Arduino. The first Arduino would then compare the received CAN messages to the CAN messages in the previously sent CANEthernet message. If the CAN messages were identical, it would indicate that the software, hardware and protocol were correctly done.

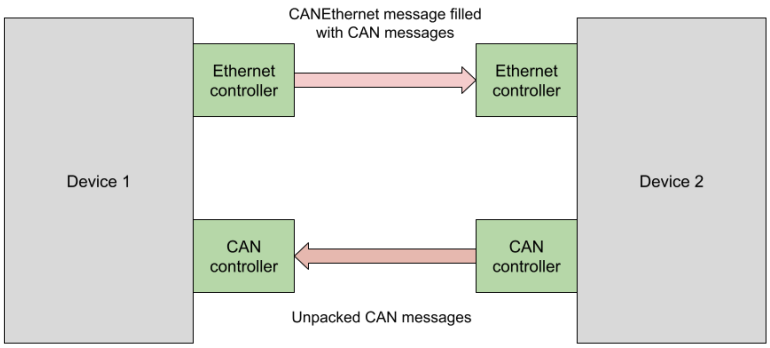


Figure 5.1: Setup

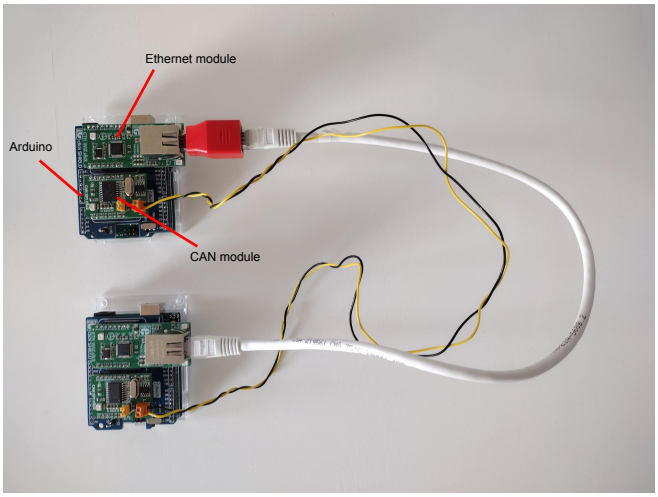


Figure 5.2: Photo of the hardware setup

5.2 Software library

A software library was developed for testing and verification purposes. The library can generate CAN test messages, assemble them into a CANEthernet frame, and send them over Ethernet using the correct EtherType. It can also receive and unpack the CANEthernet frame into CAN messages. The library facilitated testing of the new protocol and may also make further development of CANEthernet easier. The code is written in C/C++. The main part of the library is divided into two C++ files with corresponding header files.

- ***Message_functions.cpp*** contains methods for creating CAN test messages and assembling them into a CANEthernet message. Since the focus of the library is testing, the CAN test messages it creates are made up of randomized bytes with random sizes (within the boundaries dictated by CAN frame size rules). The file also contains methods for handling the ACK field, adding MAC addresses, and adding the EtherType.

- ***Parse_functions.cpp*** contains methods for reading from a CANEthernet message. It contains methods for retrieving ACK, number of messages, EtherType, and actual CAN messages. It also includes a method for checking if the ACK of a received message corresponds to the ACK of a sent message.

The function definitions in both files are included in Appendix A.

Testing was done according to the previously described plan using the software library. The testing confirmed that the CANEthernet protocol is functional on the hardware and during the conditions used in this project. No issues or errors were discovered. The successful testing indicates that CANEthernet should work on other Ethernet devices as well.

6
Evaluation

The evaluation of the CANEthernet protocol consists of comparing sending several CAN messages with transmitting those same CAN messages in an Ethernet message with the proposed CANEthernet protocol. CAN and CANEthernet are compared on three main points; throughput, latency, and energy efficiency. The calculations are based on viewing the CAN messages as data and all other bits as overhead. For clarity, this section will refer to Ethernet frames with the CANEthernet protocol simply as CANEthernet messages.

The calculations do not consider CAN stuff bits since it would significantly complicate the formulas and only make a small difference. It is assumed that there is no competing traffic on the network and that there are no errors.

6.1 Throughput

Throughput depends on several factors, such as the type of CAN, type of Ethernet and the amount of other traffic on the network. The analysis in this project is made using the highest theoretically possible throughputs.

6.1.1 CAN

The throughput for CAN depends on the size of the CAN messages. When CAN messages are sent, the CAN controller adds a 3-bit inter frame space (IFS) after each message. The IFS is viewed as the overhead in the CAN throughput calculations. The throughput *(TP)* of a standard CAN message (not CAN FD) can be calculated using:

$$TP = \left(\frac{size \times speed}{size + 3} \right) \tag{6.1}$$

where *size* is the size of the CAN message, *speed* is the transmission speed and *+3* is the overhead due to IFS.

Figure 6.1 shows the throughput of standard CAN at 1Mbps. It is evident that as the message size gets larger, the throughput increases. This is due to the overhead being constant per message. The maximum throughput is 0.97Mbps and is achieved when sending the largest possible message size, which is 108 bits.

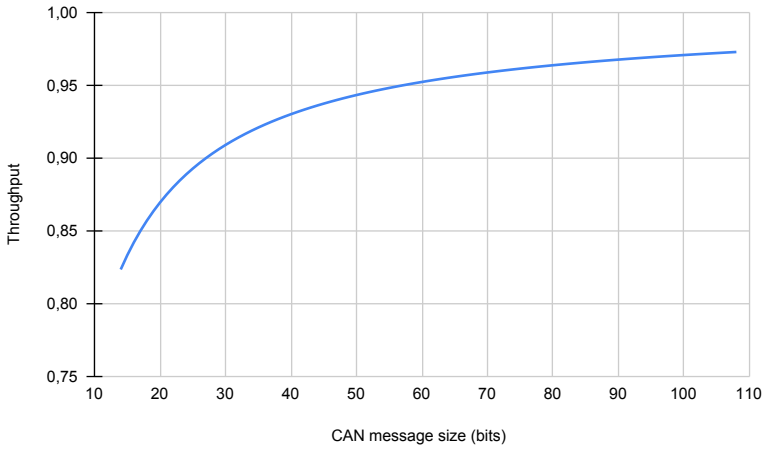


Figure 6.1: Throughput of standard CAN at 1Mbps

When calculating throughput for CAN FD, the dual bit rate for data frames must be included. The maximum speed is assumed to be 8Mbps for the data field and 1Mbps for other parts of the message. The data field is between 22-538 bits long. The rest of the message is between 27-46 bits long. CAN FD has a constant overhead of 3 bits, which leads to the the maximum throughput at the largest possible frame size. For CAN FD, it is 584 bits. The calculation for maximum throughput assumes the speed for the full message as 8Mbps. Since the non-data part of the frame is very small compared to the data field, the maximum speed can be approximated to 8Mbps. Thus, the maximum throughput is calculated to be 8Mbps using the Equation 6.1.

6.1.2 CANEthernet

A CANEthernet message has an overhead consisting of 26 bytes for the Ethernet header, 2 bytes for the Acknowledgment, 2 bytes for the number of CAN fields, the 1-byte size field for each CAN message, and a 12-byte inter packet gap(IPG). It has in total 1496 bytes for storing the CAN messages and their 1-byte size fields.

In order to calculate the CANEthernet throughput, all CAN messages are assumed to have the same size. This may not be the scenario in a real-life application of the protocol, but it is a necessary assumption to do the calculations. With this assumption, the number of CAN messages *(N)* that can fit in a CANEthernet message can be found using:

$$N = \left\lfloor \frac{1496}{\left(\lfloor \frac{CANMsgSize}{8} \rfloor + 1 + 1 \right)} \right\rfloor \tag{6.2}$$

where *CANMsgSize* is the size of a CAN message which is the same for all CAN messages, the first *+1* is the CAN message padding, and the second *+1* is the size field for a CAN message.

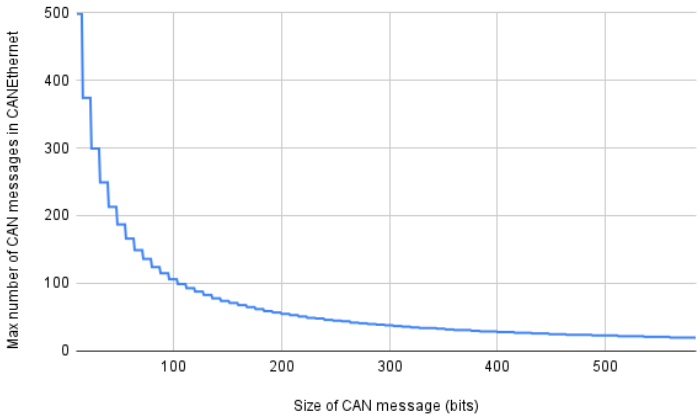


Figure 6.2: Maximum number of CAN messages in CANEthernet frame

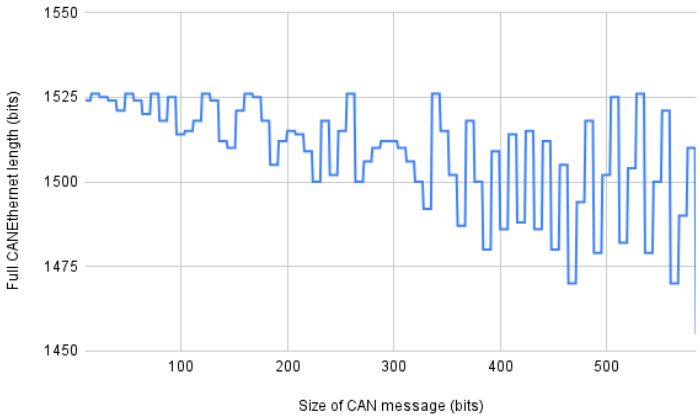


Figure 6.3: Size of CANEthernet when filled with different size CAN messages

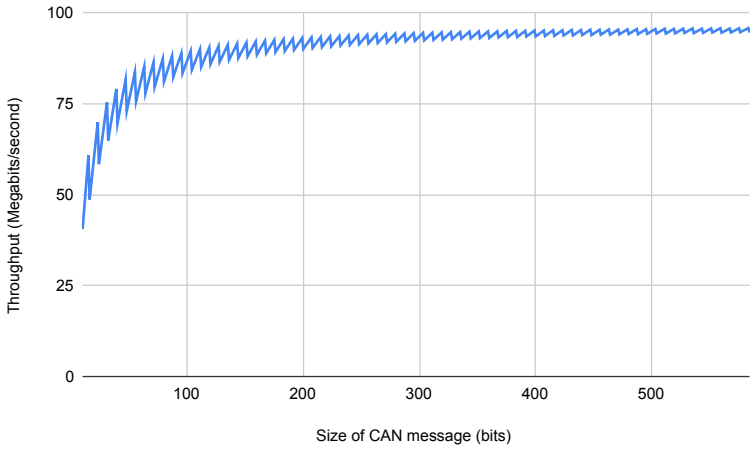
Figure 6.2 plots the value of N for all possible sizes of CAN messages. The sharp angle shapes are due to the padding of the CAN messages. For example, the frame can carry an equal amount of 1-bit and 8-bit messages as the 1-bit messages are padded to the 8 bits.

Using N, the size of the complete Ethernet message (*FullCANEthSize*) in *bytes* can be calculated using:

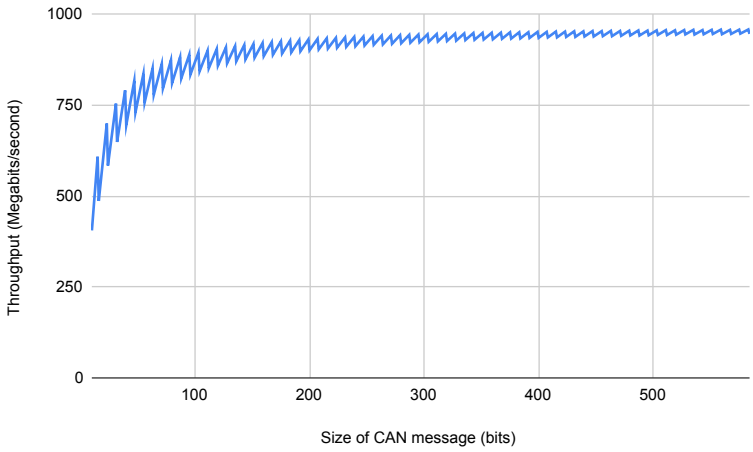
$$FullCANEthSize = \left(\lfloor \frac{CANMsgSize}{8} + 1 + 1 \rfloor \times N \right) + 2 + 2 + 26 \qquad (6.3)$$

where *+2*, *+2* and *+26* are the bytes required for the number of CAN messages field, Acknowledgement field and the Ethernet header.

Figure **??** plots *FullCANEthSize* for CAN messages of different sizes. The plot clearly indicates that CANEthernet message size varies depending on the CAN message size.



(a) CANEthernet throughput at 100Mbps



(b) CANEthernet throughput at 1Gbps

Figure 6.4: Throughput of CANEthernet

Using N and *FullCANEthSize*, the throughput *(TP)* in *bits per second* can be calculated using:

$$TP = \frac{N \times CANMsgSize}{(FullCANEthSize + 12) \times 8} \times EthSpeed \tag{6.4}$$

where *+12* is the interpacket gap and *EthSpeed* is the speed of the Ethernet connection.

Figures 6.4a and 6.4b show the throughput for CANEthernet at 100Mbps and 1Gbps. The shapes of the plots are identical, and they only differ on the scale of the vertical axis. The plot's sawtooth shape comes from two factors. The first is how much padding (wasted bits) is required. The second factor is that as messages increase in size, fewer will fit into a CANEthernet frame.

Several simplifications have been done for the calculations. They assume all CAN messages are equal in size, with no competing traffic and no errors. The ACK frames are not included either because their small size would barely impact the result. While keeping the assumptions in mind, the plots show that when CAN messages are not very small, the throughput is over 80% of the Ethernet speed.

6.1.3 Summary

The highest possible throughput for CAN is achieved using a full CAN FD message and sending it at a speed of 8Mbps. Thus, CAN gives a maximum throughput under 8Mbps, while CANEthernet can reach 9Gbps on 10Gbps Ethernet, or 90Mbps on 100Mbps Ethernet. This shows that CANEthernet can have a significantly larger throughput than CAN.

6.2 Latency

The latency of a connection depends on multiple factors, like the amount and size of the CAN messages. This project will only make calculations and analysis of best-case latency. The calculations will compare the latency of sending a single 108-bit CAN message with and without using the CANEthernet protocol.

6.2.1 CAN

Using $1Mbps$ CAN, the calculation for the latency is $108bits/1Mbps = 108us$.

6.2.2 CANEthernet

The minimum size of an Ethernet frame is 64 bytes, which would be able to carry a 108-bit CAN message. When the message is received, the receiver sends an Acknowledgment. This message would also be 64 bytes long with an additional 12-byte inter packet gap. Table 6.1 shows the latency calculation for transmitting and receiving one 108-bit CANEthernet message at $100Mbps$, giving a total latency of $11.2us$ for the CANEthernet message.

Message type	Latency
Data	$(64 \times 8\ ) \ / \ 100 = 5.1us$
Ack	$(64 + 12) \times 8 \ / \ 100 = 6.1us$

Table 6.1: Latency calculation for a CANEthernet message

6.2.3 Summary

When sending a 108-bit CAN message over $1Mbps$ CAN, the latency is $108us$. When sending a 108-bit CAN message with a CANEthernet message, the latency is $11.2us$ over $100Mbps$ Ethernet. In this situation there is a latency gain for using CANEthernet.

6.3 Energy efficiency

The power consumption and energy efficiency depend on several factors, like the components used, the amount of data transmitted per time unit, etc. The CAN and Ethernet may have different power savings when in idle mode. Thus, it is not straightforward to compare the power consumption between CAN and Ethernet with the CANEthernet protocol. This section contains simple power consumption modeling based on the components used in the implementation.

6.3.1 CAN

The CAN module used for the Arduinos is the 3.3V CAN Bus Click module. It has two main components, the SN65HVD230 CAN transceiver from Texas Instruments and the MCP2515 CAN controller from Microchip.

- SN65HVD230 consumes 17.5 mA, which corresponds to 58 mW
- MCP2515 consumes 5 mA, which corresponds to 16.5 mW

Thus, a total power of 149 mW for both the sending and receiving components. Considering the maximum throughput of 0.97 Mbps on a 1 Mbps connection from Section 6.1.1, the total consumption will be 153nJ per bit.

6.3.2 CANEthernet

The Ethernet communication is done using a 3.3V Eth Wiz Click module. There is no power consumption specification for the module, but its main component is a W5500 Ethernet controller. The W5500 datasheet indicates that at the highest speed (100 Mbps), transmitting consumes 132mA, corresponding to 436 mW and receiving consumes 128 mA, corresponding to 422 mW.

Device	Power consumption
Sender	436 mW / 80 Mbps = 5.5 nJ per bit
Receiver	422 mW / 80 Mbps = 5.3 nJ per bit

Table 6.2: Power consumption for CANEthernet

Considering the maximum throughput of 80Mbps on a 100Mbps Ethernet connection from Section 6.1.2, the power consumption per bit are calculated in Table 6.2. The calculations show that sending and receiving one bit equates to a power usage of 10.8 nJ. The power consumed for creating and unpacking the CANEthernet messages is not included.

6.3.3 Summary

For the hardware used in this project, CAN uses 153nJ per bit while CANEthernet uses 10.8 nJ per bit. With this hardware there is a large difference in power efficiency.

7
Conclusion

A new protocol called CANEthernet and an associated EtherType have been developed. It allows using Ethernet instead of a CAN bus to send CAN messages. The CANEthernet protocol has been tested and verified using CAN and Ethernet hardware.

CAN and CANEthernet were compared regarding latency, energy efficiency and throughput using assumptions on CAN message size, CAN and Ethernet versions, etc. For all three aspects, CANEthernet outperforms CAN. Despite the assumptions made, it is still clear that CANEthernet, in general, performs better than CAN in these aspects. The energy efficiency results is the most uncertain, since it depends on the hardware used. Calculations in this project was done on the hardware used for testing, however the hardware in vehicles may differ greatly from the testing hardware. The results indicates that vehicular manufacturers should consider using Ethernet instead of CAN when possible.

7.1 Project Execution

The method described in Chapter 2 has been followed without major changes. The planned schedule for the project is presented in the Gantt chart in Figure 7.1. The time spent on the different aspects has varied slightly from the original plan, but the tasks were ordered as in the planning report. The research phase was shorter than anticipated, while the development took more time. Verification and evaluation also required more time than expected. The plan included four weeks towards the end of the project for exclusively writing the report. But this couldn't be met as other tasks took more time than expected. However, overtime work enabled finishing the report and the project on time. The purpose (Section 1.2) to determine if communication between two nodes in a vehicle is more efficient using CAN messages or packing several CAN messages in an Ethernet message has been fulfilled. All parts of the goal (Section 1.3) have been met.

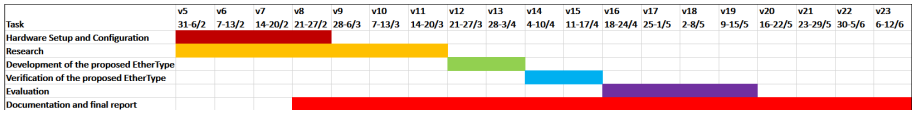


Figure 7.1: The planned schedule for the project

7.2 Future work

Future work can include further exploration of the protocol architecture and better modeling of the evaluations considered. One possible direction is defining the types of CAN messages sent with CANEthernet. For example, removing Error and Overload frames may shorten the size field. Another potential direction is gathering data about CAN message size distribution in a real scenario and using the data to make more realistic evaluations. Finally, the evaluation of energy consumption could be improved by using the parameters from real devices used in vehicular applications.

Bibliography

[1] Arduino. *Arduino Uno Rev3*. URL: https://store.arduino.cc/products/arduino-uno-rev3.

[2] Robert I. Davis et al. "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised". In: *Real-Time Syst.* 35.3 (Apr. 2007), pp. 239–272. DOI: 10.1007/s11241-007-9012-7.

[3] Timo Häckel et al. *Secure Time-Sensitive Software-Defined Networking in Vehicles*. 2022. arXiv: 2201.00589 [cs.NI].

[4] Florian Hartwich and Robert P. Bosch. *CAN with Flexible Data-Rate*. Specification Version 1.0. Bosch, Apr. 2012.

[5] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600. DOI: 10.1109/IEEESTD.2018.8457469.

[6] Internet Assigned Numbers Authority. *IEEE 802 numbers*. July 2021. URL: https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[7] Philipp Meyer et al. *Network Anomaly Detection in Cars: A Case for Time-Sensitive Stream Filtering and Policing*. 2021. arXiv: 2112.11109 [cs.NI].

[8] Mikroe. *Arduino UNO click shield*. URL: https://www.mikroe.com/arduino-uno-click-shield.

[9] Mikroe. *CAN SPI Click 3.3V*. URL: https://www.mikroe.com/can-spi-33v-click.

[10] Mikroe. *ETH WIZ Click*. URL: https://www.mikroe.com/eth-wiz-click.

[11] William Stallings. *Data and Computer Communications*. 9th Edition. Pearson, 2014.

A

Appendix 1

Listing A.1: Methods from *Message_functions.cpp*

```
//Returns EtherType
uint16_t getEtherType(const uint8_t * msg){
  return twoByteToDecimal(msg, 12);
}

//Returns ACK
uint16_t getACK(const uint8_t * msg){
  return twoByteToDecimal(msg, 14);
}

//Returns Number of messages
uint16_t getNrMessages(const uint8_t * msg){
  return twoByteToDecimal(msg, 16);
}

//Returns the two bytes starting at offset as decimal
uint16_t twoByteToDecimal(const uint8_t * ptr, int offset){
  const uint8_t* tempPtr = ptr + offset;
  const uint16_t* twoBytePtr = (const uint16_t *) tempPtr;

  return *twoBytePtr;
}

//Retrieves the CAN message with length field at offset
int getCANMessage(const uint8_t * fullMsg, uint8_t * CANMsg, int offset
    ){
    int len = fullMsg[offset];
    for(int i = 0; i<len+1; i++){
        CANMsg[i] = fullMsg[i+offset+1];
    }

    return len;
}

//Checks if received ack corresponds to sent ack
bool isAckCorrect(uint16_t send_ack, uint16_t receive_ack){
  if (send_ack >> 15 != 0) return false;
  if (receive_ack >> 15 != 1) return false;

  return (send_ack & 0xEFFF) == (receive_ack & 0xEFFF);
}
```

Listing A.2: Methods from *Parsing_functions.cpp*

```
//Randomizes CAN-msg
static uint8_t randomCANmsg(uint8_t * write_pos, int max_can_len){
  long randnum = random(2, max_can_len);
  write_pos[0] = randnum;
    for(int i = 1; i < randnum+1; i++){
      write_pos[i] = random(0, UINT8_MAX);
    }
  return randnum;
}

//Set first bit of ack
static uint16_t setTopAckBit(uint16_t ack, bool isResponse){
  if(isResponse){
    return ack | (1 << 15);
  } else{
    return ack & 0xEFFF;
  }
}

//Adds random ack number
void addAck(uint8_t* msg){
  uint16_t ack = setTopAckBit(random(0,UINT16_MAX), false);
  uint16_t* temp_ptr = (uint16_t *) (msg+14);
  *temp_ptr = ack;
}

//Adds response ack field
void setResponseAck(const uint8_t * msg, uint8_t * response_msg){
  uint16_t ack = setTopAckBit(getACK(msg), true);
  uint16_t* temp_ptr = (uint16_t *) (response_msg+14);
  *temp_ptr = ack;
}

//Adds nr of messages
static void addNrOfMessges(uint8_t* msg, uint16_t n_messages){
  uint16_t* temp_ptr = (uint16_t *) (msg+16);
  *temp_ptr = n_messages;
}

//Adds EtherType
void addEtherType(uint8_t* msg){
  msg[12] = 0x33;
  msg[13] = 0x22;
}

//Adds MAC addresses
void addMac(uint8_t* msg, const uint8_t * address, const uint8_t *
    remote){
  const int mac_len = 6;
  memcpy(&msg[0], remote, mac_len);
  memcpy(&msg[mac_len], address, mac_len);
}
```

```
//Adds can messages, return number of messages
int addCANMessages(uint8_t * msg, int max_total_len, int max_can_len){
  int len_index = 18;
  uint16_t n_messages = 0;
  while(len_index<max_total_len){
    int msg_len = randomCANmsg(msg+len_index, max_can_len);
    len_index += msg_len+1;
    n_messages++;
  }
  addNrOfMessges(msg, n_messages);
}
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden

CHALMERS