CAN Bus

FINAL THESIS

David Garcia Torre
The Faculty of Electrical Engineering and Communication
18/06/2021

ABSTRACT

The objective of this thesis was to theoretically address the most important concepts of the CAN bus as the main part, which is discussed in the first section as an introduction and as a theoretical part.In this thesis the practical objective was to implement a function that calculates the direction that has been taken between two given coordinates. For this we want to save the received coordinates in two variables to return the direction taken so that this makes sense, a library function has been developed that uses the gps L80 module to work, for this it relates all the commands of the possible actions indicated in the datasheet with our instructions

KEYWORDS

CAN, OSI, GPS, Vehicle, Layer, Signal, Voltage, Frame, Bit, Security, Honeypot

Contents

Introduction

A Controller Area Network (CAN bus) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other's applications without a host computer.

It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles to save on copper, but it can also be used in many other contexts. For each device, the data in a frame is transmitted sequentially but if sometimes there is more than one device transmitting at the same time, the highest priority device can continue while the others could back off. Frames are received by all devices, including the transmitting device. [1]

The CAN communications protocol provides the following benefits:

- It offers high immunity to interference, ability to self-diagnose and repair data errors.
- It is a standardized communications protocol, which simplifies and saves the task of communicating subsystems from different manufacturers over a common network or bus.
- The host processor delegates the communications load to an intelligent peripheral, therefore the host processor has more time to perform its own tasks.
- It is a multiplexed network, it considerably reduces cabling and eliminates point-to-point connections, except for snags.

1 Theorical part

The CAN bus is a communications protocol developed by the German firm Robert Bosch GmbH, based on a bus topology for the transmission of messages in distributed environments, in this section we are going to go a little deeper

1.1 About CAN Bus

As vehicles have become more modern and efficient, considering the ease of use for the driver, the environmental impact and passenger safety, have been controlled more parameters of the same, which has resulted in the need to have a central control unit. For example, in automatic cars the engine sends the speed of the vehicle to the transmission, which should tell other modules when to change gears.

Connect all these modules so that they communicate with each other individually and owner became inordinately complex, so the CAN bus protocol was used. The standard was introduced in 1986, the first drivers by Intel and Philips appeared in 1987, but it was not integrated into a vehicle until 1988. Not only its wiring was drastically reduced, weighing 45 kilograms less than the previous model, but the sensors worked in a much faster.

CAN is one of the key protocols for on-board diagnostics (OBD, OnBoard Diagnostics), which have been mandatory since 1996 (OBD-II) in the United States and since 2001 in the European Union (EOBD).[2]

1.1.1 Layers

We can adapt the CAN protocol to the OSI model, and it would deal with the physical level and link. We proceed to describe this equivalence and the tasks that are order each one according to this scheme.[3]

Data link layer

CAN uses as physical infrastructure a twisted pair of cables, CAN-H and CAN-L, each with a 120 ohm resistance. the dominant states and recessive that we have illustrated above are defined as a difference of voltage between them greater or less than the minimum threshold (<0.5v at the input of the receiver or <1.5v at transceiver output

Transfer layer Most of the CAN standard applies to the transfer layer. The transfer layer receives messages from the physical layer and transmits those messages to the object layer. The transfer layer is responsible for bit timing and synchronization, message framing, arbitration, acknowledgement, error detection and signaling, and fault confinement. It performs:

- Fault Confinement.
- Error Detection.
- Message Validation.
- Acknowledgement.
- Arbitration
- Message Framing
- Transfer Rate and Timing
- Information Routing

Physical layer

It defines the aspects of the physical medium for the transmission of data between nodes of a CAN network, the material and electrical characteristics and the transmission of the bit stream through the bus.

Bus voltage levels

Signal transmission on a CAN bus is carried out over two twisted cables. The signals on these cables are called CAN-H (CAN high) and CAN-L (CAN low) respectively. The bus has two defined states: dominant state and recessive state. In the recessive state, the two bus cables are at the same voltage level (common-mode voltage), while in the dominant state there is a voltage difference between CAN-H and CAN-L of at least 1.5 V. The transmission of signals in Differential voltage form, compared to transmission in the form of absolute voltages, provides protection against electromagnetic interference.

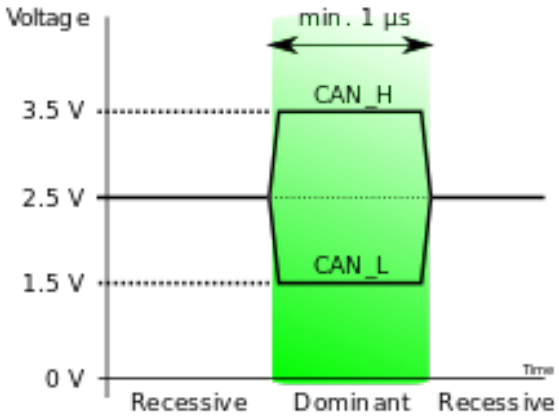

Fig. 1.1: Measured RSS level vs near-water path loss model

Cable and connectors

The different nodes of a CAN bus must be interconnected by a pair of twisted cables with a characteristic impedance of 120 ohms, and it can be screened or unshielded cable . The braided cable provides protection against external electromagnetic interference. And if, in addition, it is shielded, the protection will be greater

Tab. 1.1: Properties of the transmission

Bus length (m)	**Transfer rate (kbit / s)**
40	1000
100	500
200	250
500	100
1000	50

but in exchange for an increase in the cost of the cable.

The CAN standard, unlike other standards such as USB , does not specify any type of connector for the bus and therefore each application may have a different connector. However, there are several commonly accepted formats such as the 9-pin D-sub connector, with the CAN-L signal on pin 2 and the CAN-H signal on pin 7.

The properties of the transmission line limit the bandwidth of the data. Indicatively, the following values are accepted as length limit depending on the bus transfer rate:

1.2 Basic structure

The CAN bus topology is going to maintain a main central line to which they go connected the other modules or ECUs (Electronic Central Unit) . One of the modules can be a sensor, an actuator, or even a gateway for another device to communicate, such as a USB or Ethernet port. The nodes are connected to each other by a two-wire bus, and each requires a CPU, a CAN controller (usually built-in) and a transceiver (to receive and transmit, converts the bus information flow to controller levels and vice versa). If any of the nodes fail, the system continues to function, unless that there are directly dependent modules, which makes it more secure.[4]

It also simplifies the control system and makes it much easier introduce new modules. Any module can alert the controller of the occurrence of an event, that is, you can send information, as well as receive, but not simultaneously. There are two types of bus: the high-speed bus, which uses a single linear bus, and the low speed or fault tolerant one, which can use one linear, star, or multiple stars connected by a linear one.

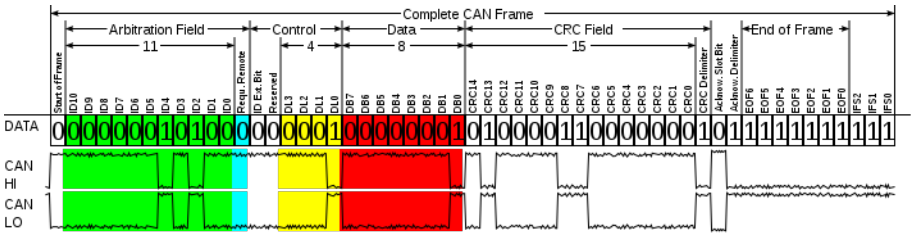

Fig. 1.2: CAN bus basic frame

1.2.1 Frames

The CAN bus frame has different fields that inform about different aspects of the message. The fields of a CAN frame are shown below. Frames use bit stuffing, that is, when 5 equal bits occur, an extra bit of value is introduced otherwise to avoid desynchronization. As will be seen later, there are two types of frames, standard and extended, based on the number of bits of the identifier. During the work, standard frames will be used since using only three nodes does not need a great field of arbitration.[5]

SOF (Start of Frame bit)

Indicates the beginning of the message and allows synchronization of all connected nodes to network. This bit is dominant (logical 0).

Arbitration field

It is made up of 12 bits or 32 bits depending on the type of frame. Inside the field finds the identifier, which indicates the priority of the node. The node with the largest priority is the one with the lowest identifier. The RTR bit is used to distinguish between a remote frame or a data frame. The different CAN Bus frames.

Control field

Made up of 6 bits. The IDE bit indicates with a dominant state that the frame sent is standard. The RB0 bit is reserved and is set in dominant state by the protocol DOG. The rest of the bits, the Data Length Code (DLC) indicates the number of bytes of data it contains the message. An extended frame has an extra bit RB1.

Data field

It can be made up of up to 8 bytes, depending on what we specify in the DLC. The message data is contained in this field.

Cyclic redundancy check field

This 15-bit field detects errors in the transmission of the message. It is delimited with a final bit in recessive state.

Recognition field

The last field of the frame is made up of 2 bits. The transmitting node sends a frame with the ACK (Acknowledge) bit in a recessive state, while the receivers,

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier	11	A (unique) identifier which also represents the message priority
Remote transmission request (RTR)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames
Identifier extension bit (IDE)	1	Must be dominant (0) for base frame format with 11-bit identifiers
Reserved bit (r0)	1	Reserved bit. Must be dominant (0), but accepted as either dominant or recessive.
Data length code (DLC)	4	Number of bytes of data (0–8 bytes)
Data field	0–64 (0-8 bytes)	Data to be transmitted
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)

Tab. 1.2: Frame format

if they have received the message correctly, they will send a message in dominant state. Contains a delimiter bit.

End of frame field

A series of 7 recessive bits indicates the end of the frame.

CAN has four frame types:

- Data frame: a frame containing node data for transmission.
- Remote frame: a frame requesting the transmission of a specific identifier.
- Error frame: a frame transmitted by any node detecting an error.
- Overload frame: a frame to inject a delay between data or remote frame.

Data Frame

The data frame is the only frame for actual data transmission. There are two message formats:

- Base frame format: with 11 identifier bits
- Extended frame format: with 29 identifier bits

The CAN standard requires that the implementation must accept the base frame format and may accept the extended frame format, but must tolerate the extended frame format.[6]

Remote frame

Generally data transmission is performed on an autonomous basis with the data source node sending out a Data Frame.It is also possible, however, for a destination node to request the data from the source by sending a Remote Frame.

There are two differences between a Data Frame and a Remote Frame. Firstly the RTR-bit is transmitted as a dominant bit in the Data Frame and secondly in the Remote Frame there is no Data Field. The DLC field indicates the data length of the requested message.[7]

- RTR = 0 ; DOMINANT in data frame
- RTR = 1 ; RECESSIVE in remote frame

Error frame

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier A	11	First part of the (unique) identifier which also represents the message priority
Substitute remote request (SRR)	1	Must be recessive
Identifier extension bit (IDE)	1	Must be recessive (1) for extended frame format with 29-bit identifiers
Remote transmission request (RTR)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames
Identifier extension bit (IDE)	1	Must be dominant (0) for base frame format with 11-bit identifiers
Identifier B	18	Second part of the (unique) identifier which also represents the message priority
Remote transmission request (RTR)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames
Reserved bits (r1, r0)	2	Reserved bits which must be set dominant (0), but accepted as either dominant or recessive
Data length code (DLC)	4	Number of bytes of data (0–8 bytes)
Data field	0–64 (0-8 bytes)	Data to be transmitted
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)

Tab. 1.3: Extended frame format

The error frame consists of two different fields:

- The first field is given by the superposition of ERROR FLAGS (6–12 dominant/recessive bits) contributed from different stations.
- The following second field is the ERROR DELIMITER (8 recessive bits).

There are two types of error flags:

Active Error Flag

Six dominant bits –> Transmitted by a node detecting an error on the network that is in error state "error active".

Passive Error Flag

Six recessive bits –> Transmitted by a node detecting an active error frame on the network that is in error state "error passive".

There are two error counters in CAN:

1. Transmit error counter (TEC)
2. Receive error counter (REC)

Overload frame

The overload frame contains the two bit fields Overload Flag and Overload Delimiter. There are two kinds of overload conditions that can lead to the transmission of an overload flag:

- The internal conditions of a receiver, which requires a delay of the next data frame or remote frame.
- Detection of a dominant bit during intermission.

Unlike the error frame, the saturation frame only occurs between frames. It is generated by detecting a dominant bit in the interframe space or by not being sending a message for internal problems.

Bit stuffing

To ensure that there are sufficient recessive-dominant transitions to guarantee synchronization, a bit of opposite polarity is inserted after five consecutive bits of

the same polarity. This practice is necessary due to the non-zero coding of the CAN protocol. The inserted bits are removed by the receiver.

All fields of the frame are filled in except for the CRC delimiter, the ACK acknowledgment, and the end of the frame. When a node detects six equal consecutive bits in a field capable of being filled, it considers it an error and emits an active error. An active error consists of six dominant consecutive bits and violates the bit stuffing rule. [8]

The stuffing bit rule implies that a frame can be longer than expected if the theoretical bits of each field in the frame are added together.

Here we can see a frame before and after bit stuffing.

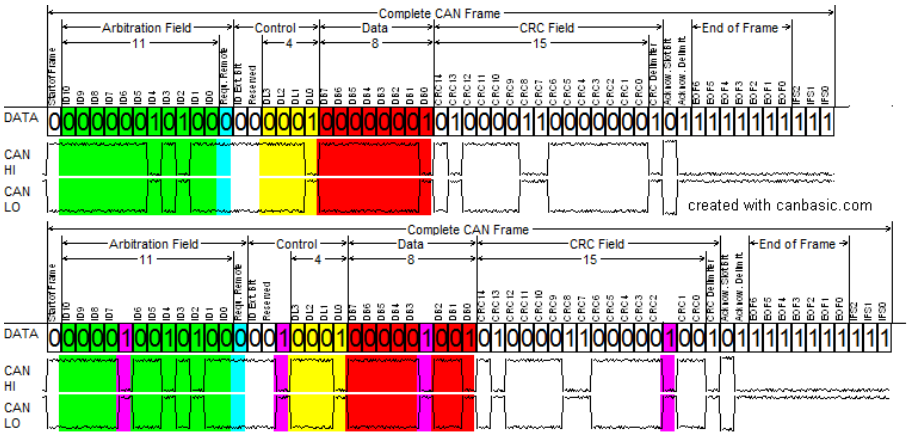

Fig. 1.3: Compare a CAN bus frame before and after bit stufing

1.3 Security

Implementing security measures in vehicle networks is complicated due to the restrictions imposed by their nature. It works in a time-controlled environment, in which messages have to be transmitted quickly and with a certain priority, since they are critical systems (a plot that indicates that the brakes are to be activated, delays cannot be allowed), in addition to synchronizing the clock ticks and time slots to transmit.

We also have to take into account that ECUs have a capacity of limited computation, as we are in embedded systems, in addition to the great It would cost the manufacturer to equip them with processors that powerful enough to carry out tasks that up to recently they had not been considered worrisome. For example, S12XD microcontrollers, specifically designed for motorsports, have a 16-bit CPU, 4KB EEPROM, and internal flash memory of 512 KB. It is common to find flash memory instead of RAM (which would have higher speed) in the ECUs for their

lower cost. Those who are in charge simpler or less relevant tasks have 8-bit CPUs, and in cases more bizarre, 32-bit, such as the V850E1 used in Toyota vehicles that are suspected they had software bugs causing them to speed up too much

Putting these two considerations together, the size of the packages cannot be excessive: they have to reach their destination and be processed in a certain time. This makes redundancy or authentication codes unable to occupy too much space, since they would occupy the bandwidth that is due dedicate to messages with importance.

Here we present several solutions proposed by both teams of research from universities and private companies that are dedicated to this industrial sector, having in several cases different proposals aimed at to solve the same problem. [9]

1.3.1 Firewall

Some companies have developed a firewall for CAN with the intention of limiting malicious traffic. For example, Genivi has a device that connects to a server to have the rules that allow or not the frames, avoiding that those that do not comply with them enter through the telematics module. Each rule is sign with a device-specific key and update over-the-air, passing as CAN frames to the firewall, which validates and stores them.

The rules of this firewall have a validation method with a mask and a filter: the ID is compared with the mask and only the bits remaining at 1 are will match the filter with an AND. If they pass the filter, they will be intercepted. They also have several fields to apply transformations to them, discard them or pass them without modification.

1.3.2 Encryption

Another bet is to encrypt the messages on the buses. This is especially dedicated because most protocols work too simple and add the burden of performing encryption on drives with so little processing power like ECUs is not a simple task. Without However, progress is being made in this regard.

Trillium has developed a technology called SecureCAN, designed to encrypt and manage payload keys of less than 8 bytes, unlike others technologies that require larger blocks. Thanks to the possibility of encrypting Frames of this small size can ensure the confidentiality of the CAN and LIN messages in real time. Symmetric encryption allows encryption, transmit and decrypt with a threshold of one millisecond, according to the company itself. However, they point out the need for a firewall despite the encryption.

The TrilliumCipher algorithm combines transposition, substitution, and matching algorithms time multiplexing, including the timestamp within the ciphertext itself, to avoid replay attacks.

1.3.3 Honeypots

Honeypots are rogue systems that make the attacker believe that they are found in a vulnerable network or system in order to monitor and analyze its activity, as well as being aware of possible new attacks. They can organize in networks (called honeynets).

It has been suggested that the honeypots connect to the gateway of the remote connections and simulate the internal network of the vehicle, performing a realistic simulation. Taking into account the cost, size and capacity of the computation needed to do this could be too much, as it should have their own hardware and not affect the operation of the real network.

1.4 Applications

1.4.1 Types

CAN was initially developed for automotive applications and therefore the protocol platform is the result of existing needs in the automotive area. The International Organization for Standardization (ISO, International Organization for Standardization) defines two types of CAN networks: a high-speed network (up to 1 Mbit / s), under the ISO 11898-2 standard, designed to control the motor and interconnect the electronic control units (ECU); and a fault tolerant low speed network (less than or equal to 125 kbit / s), under the ISO 11519-2 / ISO 11898-3 standard, dedicated to the communication of the internal electronic devices of a car such as door control , sunroof, lights and seats.

- Passenger vehicles, trucks, buses (gasoline vehicles and electric vehicles)
- Agricultural equipment
- Electronic equipment for aviation and navigation
- Industrial automation and mechanical control
- Elevators, escalators
- Building automation
- Medical instruments and equipment
- Pedelecs
- Model Railways/Railroads
- Ships and other maritime applications

- Lighting Control Systems

1.4.2 Vehicles

How can we use this technology in vehicles? Well, in a lot of devices integrated in a car for example. Let's think about the electric windows, the climate control, the central locking, the sunroof, the electric seats, the injection control unit and all its sensors, the instrument panel, the controls on the steering wheel, the multimedia systems ...

In fact, such is the number of devices that at present, to guarantee the speed and robustness of communications, there is usually not a single CAN bus, but there are several sub-buses in the vehicle. A bus for electronic engine management, another for air conditioning and entertainment, another for security issues (alarms, central locking, ABS) etc ...

Any electronic device connected to the bus can send messages and the rest listen to it. Each type of message carries an identifier. Listeners decide which messages interest them and which ones are not. For the thing to work, the electrical devices take turns "talking" one at a time.

Another use of this protocol is for diagnostic services and vehicle data collection. Cars have a special connector called OBD that is usually found under the steering wheel. This connector allows us to access the CAN buses of the car. With an adapter we can connect a computer, smartphone or similar and thus we will find out everything that is cooked inside our car.

The CAN protocol was the brainchild of Bosch in 1982 and the first production model to assemble it was the 1992 Mercedes-Benz E-Class. The CAN bus has become a de facto standard and is now used in the vast majority of countries. Automobiles being manufactured and also beginning to enter the motorcycle industry.

2 Practical Experiments

To exemplify all the theoretical knowledge, we are going to develop some programs that could be used in the GPS L80 module.

Fig. 2.1: Module L80

2.1 GPS Program developed

This GPS is a project made with the intention of making a final project in which we talk about the CAN Bus, a technology widely used these days which we consider that serves to acquire a very practical and useful knowledge for future employment.

The objective of this program is to be able to have the software developed to be able to make a practical application in the GPS L80 module where we will be able to verify the correct operation of the same one.

The software that we are going to use consists of two parts, the program that installed and executed will make the commands / orders that we program it to do. The other part is the library, which is responsible for the commands that we use in the GPS program, the device itself understands them as this built to do things based on the commands you need and that process needs a translation, and this is responsible for the library, in which we tell him that we want to do depending on the order that comes to him.

In this case, let's first talk about the C program.

The objective is to make a code which receives two points and tells me in function of the first one, the direction in which it has moved until it reaches the second one. For this the program receives two inputs, which are the two points, which have two coordinates each. In the program, the two coordinates are used to form a vector, using the vector product and with the help of mathematics we can easily calculate the angle between two vectors and compare it with the X-axis to have all the results in function of it.

The results obtained are the angle that is closer to the axis being always less than 180 degrees, to solve this problem and get a result between 0 and 360 degrees, we have to take the complementary to have a result in the range we want. Also, to avoid coding errors, we have added the exception that the end point is the same as the initial, and in that case the angle obtained is 0.

```c
#include <stdio.h>
#include <math.h>
#define PI 3.14159265

int main () {

  float v1x,v2x,v1y,v2y;
  double m, ret,vecx,vecy;

  while(1){

    printf("Enter the coordinates of the first point (X / Y) \n");
    scanf("%f %f",&v1x,&v1y);
    printf("Enter the coordinates of the second point (X / Y) \n");
    scanf("%f %f",&v2x,&v2y);

    vecx=v2x-v1x; //Calculate vect X
    vecy=v2y-v1y; //Calculate vect Y

    //Calculate angle between vectors
    m=(vecx/(sqrt((vecx*vecx)+(vecy*vecy))));
    ret = acos (m) *  (180.0 / PI);

    //This is the way i choose the angles that can be more than 180
   taking the complementary
    if(v1y>v2y){
      ret=360-ret;
    }
    if((v1x==v2x)&&(v1y==v2y)){
      ret=0;
    }

    printf("The angle is: %.2f\n", ret);

  }
```

```
35   return 0;
36 }
```

2.2 Library for module L80

In this case, we are going to talk about the library, which is a set of functional implementations, coded in a programming language, that provides a well-defined interface to the functionality that is invoked.

For this we need the information found in the datasheet which in our case is GNSS SDK commands manual since we are working with GNSS Module Series using version 1.4 of 2017.

With this file what we do is to translate the functionality of our program in commands that understand the module that will apply them, for this what we do is to create a series of functions that depending on what gets them to be called do a procedure or another.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int gen_checksum(const char * command, int size ){
5
6   //We received the command of the function an the size
7   int i=0,checksum=0;
8   char hex[256]={0}; //We use 256 because 255 is the length max and we
      have a empty one at the end
9
10  checksum=command[0];
11
12  for(i=1;i<size;i++){
13    checksum=checksum^command[i]; //Xor all the vytes
14  }
15
16  return checksum; //return the checksum
17 }
```

```
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36 /*
37 On this next functions we receive a string of data and some variables,
       we use the sprintf command to join them.
38 We use 256 of buffer because 255 is the length max and we have a empty
       one at the end.
39 The sprintf command line uses the buffer to join the sting with the
       variables to make possible make the checksum
40 We transmit the huart the info
41 */
42
43 void Change_NMEA_Port_Default_Baud_Rate( UART_HandleTypeDef *huart,int
       baudrate) {
44         int bytes = 0;
45         int checksum = 0;
46         char buffer[256] = {0};
47
48         bytes = sprintf(buffer, "%s,%d", "PQBAUD,W", baudrate);
49
50         checksum = gen_checksum(buffer, bytes);
51
52         bytes = sprintf(buffer, "%s,%d,%X\r\n", "PQBAUD,W", baudrate,
       checksum);
53
54         HAL_UART_Transmit(huart, buffer, bytes, 100);
55 }
```

The full code is on the appendix.

3 Conclusion

Thesis conclusion. As a conclusion to the final thesis, we can divide everything done into two parts, a theoretical part where it has been explained what the CAN bus is, the different layers have been discussed, and the basic structure of a frame transported in the CAN bus, security measures implemented to protect this communication and finally a series of practical applications in the market today.

In the second part of the thesis, it is divided into two subparts. First a C program that from two points entered as coordinates on the screen takes us out a direction in which the object is directed from the first point to the second, this direction will be an angle between 0 and 360 degrees. Finally, so that the L80 module, which is the test object in which the system should have been tested, works.

Bibliography

[1] "About can bus." [Online]. Available: http://www.auterraweb.com/aboutcan.html

[2] "What is can bus (controller area network)." [Online]. Available: https://dewesoft.com/daq/what-is-can-bus

[3] W. Buchanan, "Can bus," *Computer Busses*, p. 333–343, 2000.

[4] N. Liang and D. Popovic, "The can bus," *Intelligent Vehicle Technologies*, p. 21–64, 2001.

[5] B. Hunting, "Can bus system: Understanding the basics," Apr 2020. [Online]. Available: https://knowhow.napaonline.com/can-bus-system-understanding-basics/

[6] C. Electronics, "Can bus explained - a simple intro (2021)." [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en

[7] "Controller area network." [Online]. Available: http://www.esd-electronics-usa.com/Controller-Area-Network-CAN-Introduction.html

[8] "Can bus explained - a simple intro (2020)," Jul 2017. [Online]. Available: https://www.youtube.com/watch?v=FqLDpHsxvf8

[9] "Can bus," Jun 2021. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus

Symbols and abbreviations

CAN	Controller Area Network
GPS	Digital Signal Processing
OBD	OnBoard Diagnostic
OSI	Open System Interconnection
EOBD	European On Board Diagnostic
ECU	(Electronic Central Unit
CPU	Central Processing Unit
SOF	Start of Frame
RTR	Remote transmission request
DLC	Data Length Code
CRC	Cyclic redundancy check
ACK	Acknowledge
TEC	Transmit error counter
REC	Receive error counte
EEPROM	Electrically Erasable Programmable Read-Only Memory
RAM	Random Access Memory
ISO	International Organization for Standardiza-tion)
OBD	On Board Diagnostics

List of appendices

A L80 Library

Here is the full code of L80 Library designed

```
#include <stdio.h>
#include <string.h>

int gen_checksum(const char * command, int size ){

  //We received the command of the function an the size
  int i=0,checksum=0;
  char hex[256]={0}; //We use 256 because 255 is the length max and we
   have a empty one at the end

  checksum=command[0];

  for(i=1;i<size;i++){
    checksum=checksum^command[i]; //Xor all the vytes
  }

  return checksum; //return the checksum
}


/*
On this next functions we receive a strinf of data and some variables,
    we use the sprintf command to join them.
We use 256 of buffer because 255 is the length max and we have a empty
    one at the end.
The sprintf command line uses the buffer to join the sting with the
    variables to make possible make the checksum
We transmit the huart the info
*/

void Change_NMEA_Port_Default_Baud_Rate( UART_HandleTypeDef *huart,int
    baudrate) {
        int bytes = 0;
        int checksum = 0;
        char buffer[256] = {0};

        bytes = sprintf(buffer, "%s,%d", "PQBAUD,W", baudrate);

        checksum = gen_checksum(buffer, bytes);

        bytes = sprintf(buffer, "%s,%d,%X\r\n", "PQBAUD,W", baudrate,
    checksum);

```

```
38          HAL_UART_Transmit(huart, buffer, bytes, 100);
39  }
40
41
42  void Enable_Disable_PQEPE_Sentence_Output(UART_HandleTypeDef *huart,int
        mode, int save) {
43
44    int bytes = 0;
45      int checksum = 0;
46      char buffer[256] = {0};
47
48      bytes = sprintf(buffer, "%s,%d,%d", "PQEPE,W", mode, save);
49
50    checksum = gen_checksum(buffer, bytes);
51
52      bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQEPE,W", mode, save,
       checksum);
53
54    HAL_UART_Transmit(huart, command, sizeof(command), 100);
55  }
56
57
58  void Set_type_1PPS_output_PPS_pulse_width(UART_HandleTypeDef *huart,int
        type, int width) {
59
60
61    int bytes = 0;
62      int checksum = 0;
63      char buffer[256] = {0};
64
65      bytes = sprintf(buffer, "%s,%d,%d", "PQ1PPS,W", type, width);
66
67    checksum = gen_checksum(buffer, bytes);
68
69      bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQ1PPS,W", type, width,
        checksum);
70
71    HAL_UART_Transmit(huart, command, sizeof(command), 100);
72
73    }
74
75
76  void Change_FLP_mode(UART_HandleTypeDef *huart, int mode, int save) {
77
78    int bytes = 0;
79      int checksum = 0;
80      char buffer[256] = {0};
```

```
81
82      bytes = sprintf(buffer, "%s,%d,%d", "PQFLP,W", mode, save);
83
84    checksum = gen_checksum(buffer, bytes);
85
86      bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQFLP,W", mode, save,
      checksum);
87
88    HAL_UART_Transmit(huart, command, sizeof(command), 100);
89
90  }
91
92
93  void Enable_Disable_GPTXT_sentence_output(UART_HandleTypeDef *huart,
      int mode, int save) {
94
95
96    int bytes = 0;
97      int checksum = 0;
98      char buffer[256] = {0};
99
100     bytes = sprintf(buffer, "%s,%d,%d", "PQTXT,W", mode, save);
101
102   checksum = gen_checksum(buffer, bytes);
103
104     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQTXT,W", mode, save,
      checksum);
105
106   HAL_UART_Transmit(huart, command, sizeof(command), 100);
107
108 }
109
110
111 void Enable_Disable_ECEFPOSVEL_Sentence_Output(UART_HandleTypeDef *
      huart, int mode, int save) {
112
113   int bytes = 0;
114     int checksum = 0;
115     char buffer[256] = {0};
116
117     bytes = sprintf(buffer, "%s,%d,%d", "PQECEF,W", mode, save);
118
119   checksum = gen_checksum(buffer, bytes);
120
121     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQECEF,W", mode, save,
      checksum);
122
```

```
123   HAL_UART_Transmit(huart, command, sizeof(command), 100);
124
125 }
126
127
128 void Start_Stop_odometer_reading(UART_HandleTypeDef *huart, int mode,
      int initialdistance) {
129
130   int bytes = 0;
131     int checksum = 0;
132     char buffer[256] = {0};
133
134     bytes = sprintf(buffer, "%s,%d,%d", "PQODO,W", mode,
      initialdistance);
135
136   checksum = gen_checksum(buffer, bytes);
137
138     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQODO,W", mode,
      initialdistance, checksum);
139
140   HAL_UART_Transmit(huart, command, sizeof(command), 100);
141
142 }
143
144
145 void Switching_WGS84(UART_HandleTypeDef *huart, int mode, int save) {
146
147   int bytes = 0;
148     int checksum = 0;
149     char buffer[256] = {0};
150
151     bytes = sprintf(buffer, "%s,%d,%d", "PQPZ90,W", mode, save);
152
153   checksum = gen_checksum(buffer, bytes);
154
155     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQPZ90,W", mode, save,
      checksum);
156
157     HAL_UART_Transmit(huart, command, sizeof(command), 100);
158
159 }
160
161
162 void PQGLP_Set_GLP_Mode(UART_HandleTypeDef *huart, int mode, int save)
      {
163
164   int bytes = 0;
```

```
    int checksum = 0;
    char buffer[256] = {0};

    bytes = sprintf(buffer, "%s,%d,%d", "PQGLP,W", mode, save);

  checksum = gen_checksum(buffer, bytes);

    bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQGLP,W", mode, save,
   checksum);

  HAL_UART_Transmit(huart, command, sizeof(command), 100);
}


void Enable_disable_velocity_sentence_output(UART_HandleTypeDef *huart,
    int mode, int save) {

  int bytes = 0;
    int checksum = 0;
    char buffer[256] = {0};

    bytes = sprintf(buffer, "%s,%d,%d", "PQVEL,W", mode, save);

  checksum = gen_checksum(buffer, bytes);

    bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQVEL,W", mode, save,
   checksum);

  HAL_UART_Transmit(huart, command, sizeof(command), 100);

}


void Enable_Disable_jamming_detection_function(UART_HandleTypeDef *
   huart, int mode, int save) {

  int bytes = 0;
    int checksum = 0;
    char buffer[256] = {0};

    bytes = sprintf(buffer, "%s,%d,%d", "PQJAM,W", mode, save);

  checksum = gen_checksum(buffer, bytes);

    bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQJAM,W", mode, save,
   checksum);

```

```
207   HAL_UART_Transmit(huart, command, sizeof(command), 100);
208
209 }
210
211
212 void Enable_Disable_Return_Link_Message_Output(UART_HandleTypeDef *
      huart, int mode, int save) {
213
214   int bytes = 0;
215     int checksum = 0;
216     char buffer[256] = {0};
217
218     bytes = sprintf(buffer, "%s,%d,%d", "PQRLM,W", mode, save);
219
220   checksum = gen_checksum(buffer, bytes);
221
222     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQRLM,W", mode, save,
      checksum);
223
224   HAL_UART_Transmit(huart, command, sizeof(command), 100);
225
226 }
227
228
229 void Configure_Parameters_Geo_fence(UART_HandleTypeDef *huart,int GEOID
      ,int mode,int shape,int latitude0,int longitude0,int latitude1/
      radius,int longitude1,int latitude2,int longitude2,int latitude3,
      int longitude3 ) {
230
231   int bytes = 0;
232     int checksum = 0;
233     char buffer[256] = {0};
234
235     bytes = sprintf(buffer, "%s,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d", "
      PQGEO,W",GEOID,mode,shape,latitude0,longitude0,latitude1/radius,
      longitude1,latitude2,longitude2,latitude3,longitude3);
236
237   checksum = gen_checksum(buffer, bytes);
238
239     bytes = sprintf(buffer, "%s,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%X\r\n
      ", "PQGEO,W", GEOID,mode,shape,latitude0,longitude0,latitude1/
      radius,longitude1,latitude2,longitude2,latitude3,longitude3,
      checksum);
240
241   HAL_UART_Transmit(huart, command, sizeof(command), 100);
242
243 }
```

```
244
245
246 void Configure_Parameters_Precision(UART_HandleTypeDef *huart,int
    latitudebits,int longitudebits,int altitudebits,int save) {
247
248   int bytes = 0;
249     int checksum = 0;
250     char buffer[256] = {0};
251
252     bytes = sprintf(buffer, "%s,%d,%d,%d,%d,%d", "PQPREC,W",
    latitudebits,longitudebits,altitudebits, save);
253
254   checksum = gen_checksum(buffer, bytes);
255
256     bytes = sprintf(buffer, "%s,%d,%d,%d,%d,%d,%X\r\n", "PQPREC,W",
    latitudebits,longitudebits,altitudebits, save, checksum);
257
258   HAL_UART_Transmit(huart, command, sizeof(command), 100);
259
260 }
261
262
263 void Enable_Disable_GBS_Sentence_Output(UART_HandleTypeDef *huart, int
    mode, int save) {
264
265   int bytes = 0;
266     int checksum = 0;
267     char buffer[256] = {0};
268
269     bytes = sprintf(buffer, "%s,%d,%d", "PQGBS,W", mode, save);
270
271   checksum = gen_checksum(buffer, bytes);
272
273     bytes = sprintf(buffer, "%s,%d,%d,%X\r\n", "PQGBS,W", mode, save,
    checksum);
274
275   HAL_UART_Transmit(huart, command, sizeof(command), 100);
276
277 }
```

B Check the checksum

I used a code only to check that the checksum was working right:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  /*-
5  This code is used for check the funtcion gen_checksum used on the
       library
6  We use to calculate the checksum a XORing each byte, so this fucntion
       takes input and xors it byte by byte to return the final value.
7  */
8  int gen_checksum(const char * command, int size ){
9
10   int i=0,checksum=0;
11   char hex[255]={0}; //Declarate the string we are going to use
12
13   checksum=command[0];
14
15   for(i=1;i<size;i++){
16     checksum=checksum^command[i]; //Xor all bytes from 1 to the size of
       the string
17   }
18
19   return checksum;
20 }
21
22 void main(){
23
24   int variable=0,i=0;
25   char hex[255]={0};
26   char buffer[255]="PQBAUD,W,115200"; //introduce the sting data on the
       string
27   variable=gen_checksum(buffer,strlen(buffer)); //we sent to the
       function the string and the length
28   printf("%X \n",variable);
29
30 }
```

On the botton on the code in line 24 I can use what ever I want to check the functionality.