

IoT-Proj

2022

Contents

1	CAN bus	5
1.1	A simple introduction	5
1.2	So what is an ECU	5
1.3	OSI MODEL	6
1.4	CAN Standard	6
1.4.1	This is where the CAN Standard comes in handy	6
1.5	CAN Definition	8
1.5.1	Characteristic	9
1.6	Top 4 benefits of CAN bus	10
1.7	The CAN bus history	11
1.8	CAN MESSAGE	11
1.9	What is a CAN frame?	12
1.9.1	The 8 CAN bus protocol message fields	13
1.10	CAN bus protocol Characteristic	14
1.10.1	Error Frame	14
1.11	How to log CAN bus data	15
1.11.1	But how do you actually record CAN bus data?	16
1.11.2	How can you interpret can bus data	17
1.12	How do you get the relevant dbc file?	19
1.13	Related protocols	20
1.14	Conclusion	21

2	Project	23
3	ThingSpeak	25
3.1	Introduction	25
3.2	Create Channel	25
3.3	Collect and Analyze Data	26
3.3.1	Read Data from a Channel	26
3.3.2	Calculate the Dew Point	27
3.3.3	Visualize Dew Point Measurement	28
4	MATLAB	31
4.1	Vehicle Network Toolbox	31
4.1.1	MathWorks Virtual Channels	31
4.2	Preparation of the CAN channel	32
4.2.1	Create CAN Channels	32
4.2.2	Configure Channel Properties	35
4.2.3	Start the Channels	35
4.2.4	Create a Message Object	36
4.2.5	Our 2 CAN Message Object	36
4.3	Pack Messages: Send and Transmit	39
4.4	Transmit And Visualize CAN Data Using CAN Explorer	41
4.4.1	Open the CAN explorer	41
4.4.2	Select the Device Channel	41
4.4.3	Configure the Channel Bus Speed	42
4.4.4	Start Monitoring Transmitted Message	42
4.5	Receive and Unpack Messages	45
4.5.1	Start Monitoring Received Message	47
4.6	Conclusion	48
5	Second Example	51
5.1	ID Messages	51

5.2	Create CAN Channels	52
5.2.1	Start the Channels	52
5.2.2	Create a Message Object	53
5.2.3	Our CAN message Object	53
5.3	Transmit And Visualize CAN Data Using CAN Explorer	54
5.3.1	Open the CAN explorer	54
5.3.2	Select the Device Channel	55
5.3.3	Configure the Channel Bus Speed	55
5.4	Pack Messages: Send and Transmit	55
5.4.1	First Iteration	56
5.4.2	Second Iteration	59
5.4.3	Fifth Iteration	61
5.4.4	sixth iteration	62
5.4.5	Conclusion	62
6	DBCAn File	65
6.1	Use DBC-File in CAN Communication	65
6.2	Open the DBC-File	65
6.2.1	canDatabase	66
6.2.2	Code	66
6.3	View Message Information	67
6.3.1	messageInfo	67
6.4	View Signal Information	67
6.5	Create a Message Using Database Definitions	68
6.6	View New Signal Information	69
6.7	Change Signal Information	69
6.8	receive Messages with Database Information	72
6.9	Receive Messages	72
6.10	Examine a Received Message	74
6.11	Extract All Instances of a Specified Message	74

6.12	Extract All the Signal Value	75
6.13	CAN Explorer	76
6.14	Plot Physical Signal Values	77
6.15	Close the DBC-File	79
6.16	Conclusion	79
7	Creation DBC-file	81
7.1	Kvaser Database Editor	81
7.2	Use DBC-File in CAN Communication	83
7.3	Open the DBC-File	83
7.4	View Message Information	84
7.5	View Signal Information	85
7.6	Create a Message Using Database Definitions	86
7.7	View New Signal Information	87
7.8	Change Signal Information	87
7.9	Start and receive Messages with Database Information	91
7.10	CAN Explorer	92
7.11	Conclusion	94
	Bibliography	96

Chapter 1

CAN bus

1.1 A simple introduction

CAN Bus means Controller Area Network. In order to understand what CAN bus is, imagine that your car is like human body: In this context, CAN Bus is the nervous system, enabling communication between different parts of the body. Communication happens between different CAN Bus ‘nodes, which are also ‘Electronic control units’ (ECUs).

The CAN bus was developed by BOSCH as a multi-master message broadcast system, it does not send large blocks of data point-to-point from node A to node B.

Usually, many short messages are broadcast to the entire network, like data sensor, which provides for data consistency in every node of the system.^[1] The goal was to make cars more reliable and safer in order to allow robust serial communication.

The CAN protocol has great popularity in industrial automation, automotive-truck applications and in small quantities in the medical field.

1.2 So what is an ECU

These CAN nodes are like parts of the body, interconnected via the CAN bus. Information from one node can be shared with other nodes. In an automotive CAN bus system, ECUs can be entities like the engine control unit, airbags, audio system etc. A modern car may

have up to 70 ECUs – and each of them may have information that needs to be shared with other parts of the network.

1.3 OSI MODEL

The Open System Interconnection (OSI) also known as model (ISO/OSI) is a conceptual model that describes the universal standard of communication established in 1984 by the International Organization for Standardization (ISO), the main international standardization body. The OSI model allows the interoperability between products from different manufacturers.

This model establishes for the logical network architecture a layered structure composed of a stack of network communication protocols divided into 7 levels, which together perform all the functions of the network, following a logical-hierarchical model. Each layer shares information with the corresponding layer of another node with a communication protocol.

1.4 CAN Standard

1.4.1 This is where the CAN Standard comes in handy

The CAN bus system enables each ECU to communicate with all other ECUs - without complex dedicated wiring. Specifically, an ECU can prepare and broadcast information (e.g. sensor data) via the CAN bus.

The physical communication happens via the CAN bus wiring harness, consisting of two wires, CAN low and CAN high. The broadcasted data is accepted by all other ECUs on the CAN network - and each ECU can then check the data and decide whether to receive or ignore it.

The CAN communication protocol, ISO-11898: 2003, describes how information is passed between devices on a network and conforms to the Open Systems Interconnection(OSI) model.

In more technical terms, the controller area network is described by a data link layer and physical layer.

In the case of high speed CAN, ISO 11898-1 describes the data link layer, while ISO 11898-2 describes the physical layer. The role of CAN is often presented in the 7 layer OSI model as per the illustration.

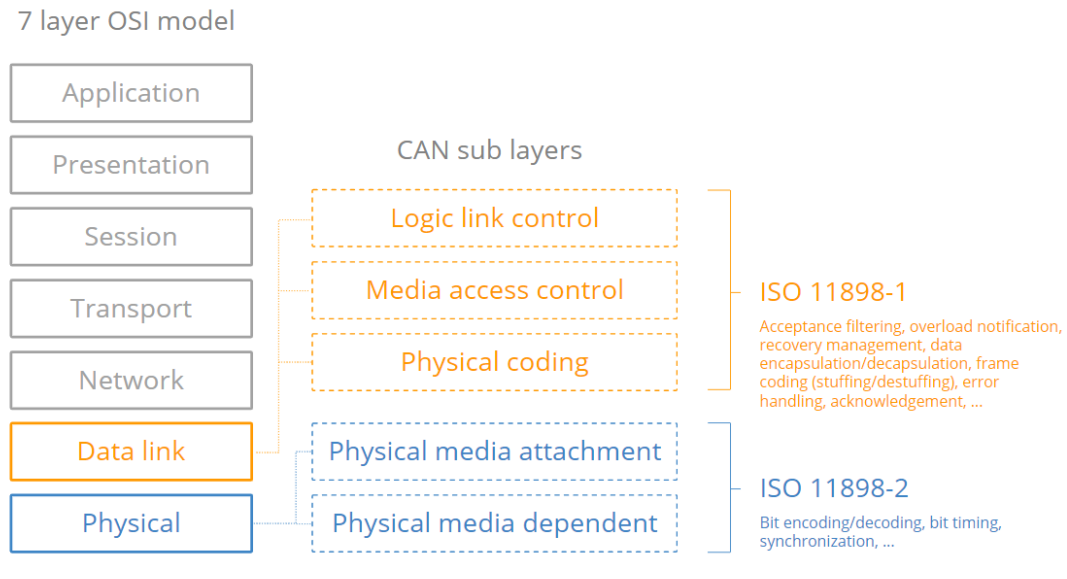
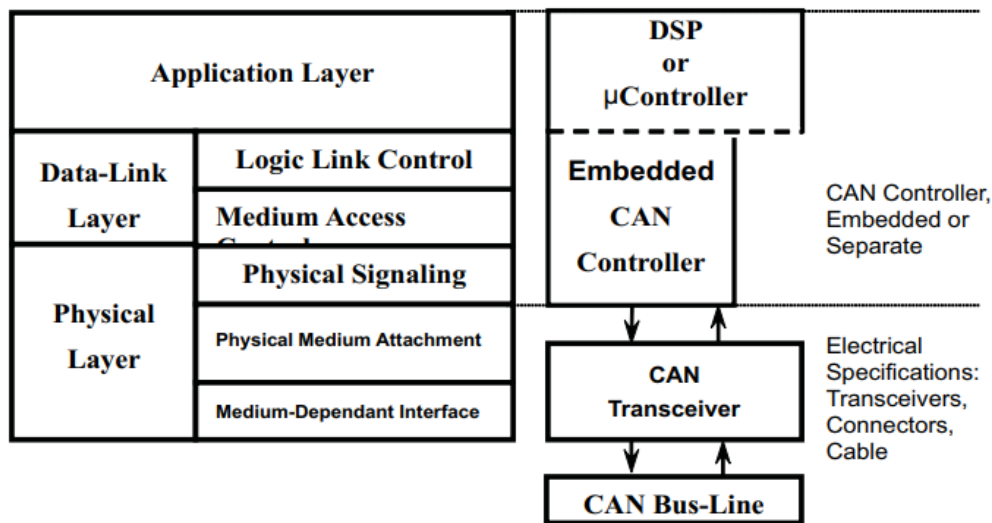


Figure 1.1: ISO 11898

The CAN bus physical layer defines things like cable types, electrical signal levels, node requirements, cable impedance etc. For example, ISO 11898-2 dictates a number of things, including below:

- Baud rate: CAN nodes must be connected via a two wire bus with baud rates up to 1 Mbit/s (Classical CAN) or 5 Mbit/s (CAN FD)
- Cable length: Maximal CAN cable lengths should be between 500 meters (125 kbit/s) and 40 meters (1 Mbit/s)
- Termination: The CAN bus must be properly terminated using a 120 Ohms CAN bus termination resistor at each end of the bus



[1]

Figure 1.1: data link and physical layer

In the context of automotive vehicle networks, you'll often encounter a number of different types of network types but the focus is on the HIGH speed CAN bus:

- High speed CAN bus (ISO 11898): It is by far the most popular CAN standard for the physical layer, supporting bit rates from 40 kbit/s to 1 Mbit/s (Classical CAN). It provides simple cabling and is used in practically all automotive like cars, trucks, buses, tractors, bikes as well as ships plane and more. It also serves as the basis for several higher layer protocols such as OBD2, J1939, NMEA 2000, CANopen etc.

The second generation of CAN is referred to as CAN FD (CAN with Flexible Data-rate)

1.5 CAN Definition

CAN is an International Standardization Organization (ISO) defined serial communications bus originally developed for the automotive industry in order to replace complex wiring harness with a more simple structure: two-wire bus.[1]

1.5.1 Characteristic

Dominant and Recessive Bit

In the protocol CAN bit are defined dominant and recessive.

If one node transmits a dominant bit at the same time another node transmits a recessive bit, the first one will be send on the bus.

In the CAN protocol, the logic bit 0 is defined as the 'dominant' bit and the logic bit 1 as the 'recessive' bit.

CSMA-CD

The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority(CSMA/CD).[1]

- CSMA means that each node on a bus must wait for a prescribed period of inactivity before attempting to send a message (Carrier Sense). Everytime the period of inactivity doesn't attempt, each node has the same possibility to transmit a message (Multiple Access)
- CD means that collisions are resolved through a bit-wise arbitration, based on pre-programmed priority of each message in the identifier field of message. The arbitration is nondestructive, this means that the messages remain intact after the arbitration has been completed even if collisions have been ascertained

Communication with Messages

The CAN protocol is a message based protocol, not addressing based. It means that message itself contains the priority and content of the transmitted data.

Each nodes of the system receive all the messages sent on the bus, give the acknowledge and decide if the message should be immediately discarded or held for processing.

Other important features:

- the CAN protocol is the possibility for each node to request information from other

nodes in order to obtain periodically information. This is called Remote Transmission Request (RTR)

- The nodes can be added to the system without the need to reprogram all the other nodes
- The allocation of priority to messages in the IDENTIFIER is a feature of CAN that makes it particularly attractive for use within a real-time control environment. The lower the binary message identifier number, the higher its priority. An identifier consisting entirely of zeros is the highest priority message on a network because it holds the bus dominant the longest[1]

1.6 Top 4 benefits of CAN bus

There are 4 key benefits to CAN bus that help explain the popularity:

- First, CAN bus is simple and low cost. ECUs communicate via single CAN system instead of via direct complex analogue signal lines - reducing errors, weight, wiring and costs
- Second, it is fully centralized. The CAN bus provides 'one point-of-entry' to communicate with all network ECUs - enabling central diagnostics, data logging and configuration
- Third, CAN is extremely robust. In particular, the system is robust towards electric disturbances and electromagnetic interference - ideal for safety critical applications (e.g. vehicles)
- Fourth, it is efficient. CAN frames are prioritized by ID so that top priority data gets immediate bus access, without causing interruption of other frames or CAN errors

1.7 The CAN bus history

- In terms of the history of CAN bus, it was initially developed by Bosch in 1986 as a solution to the prior challenge of complex point-to-point wiring in cars
- In 1991, Bosch published CAN 2.0, with 2.0 referring to 11-bit CAN IDs and 2.0B referring to 29-bit CAN IDs
- In 1993, CAN was adopted as an international standard, ISO 11898
- In 2003, the standards was separated for the data link and physical layers
- In 2012, Bosch released the next generation of CAN called CAN FD with flexible data-rate
- In 2015, CAN FD was standardized and is gradually being roll out in vehicles and machinery today
- In 2018 more recently, work has also begun on the next generation CAN XL protocol

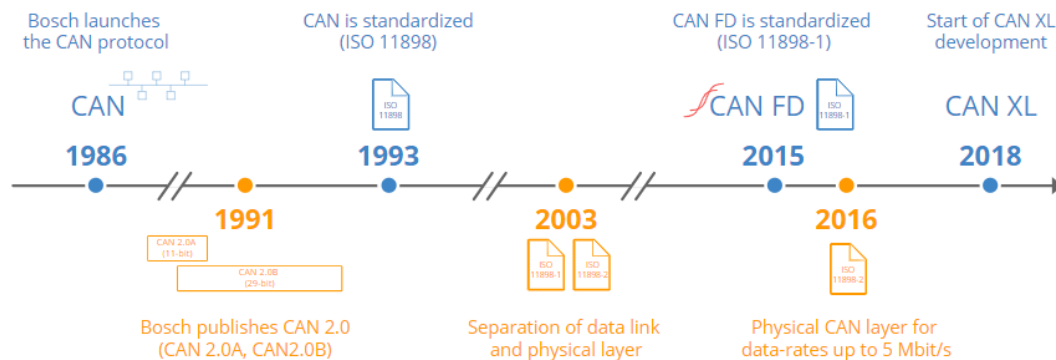


Figure 1.3: CAN history

[?]

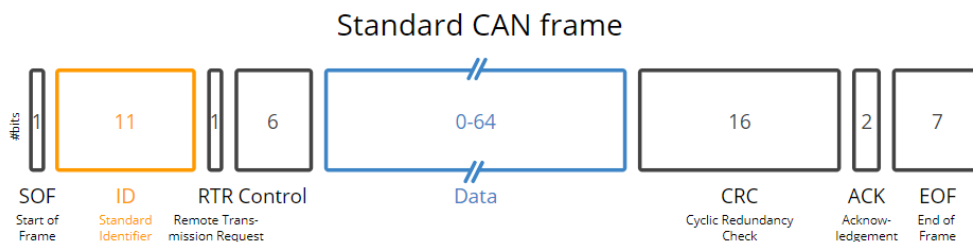
1.8 CAN MESSAGE

CAN protocol defines four different types of messages (or frames).

- The data frame is the most common message type, and comprises the Arbitration Field, the Data Field, the CRC Field, and the Acknowledgement Field. The Arbitration Field Contains an 11-bit identifier and the RTR bit, which is dominant for data frames
- The Remote Frame is used to solicit the transmission of data from another node. It is similar to the data frame, with two differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data
- The Error Frame is transmitted when a node detects an error in a message according with the many protocol errors defined by the CAN standard, and causes all other nodes in the network to send an error frame as well
- The Overload Frame is transmitted by a node that becomes too busy. It is primarily used to provide for an extra delay between messages

1.9 What is a CAN frame?

To further understand how CAN bus works, let's look at the CAN frame. Communication over the CAN bus is done via CAN frames. Below is a standard CAN frame with 11 bits identifier (CAN 2.0A), which is the type used in most cars. The extended 29-bit identifier frame (CAN 2.0B) is identical except the longer ID. It is e.g. used in the J1939 protocol for heavy-duty vehicles. Note that the CAN ID and Data are highlighted - these are important when recording CAN bus data, as we'll see below.



[?]

Figure 1.3: CAN frame

1.9.1 The 8 CAN bus protocol message fields

Let's briefly go through each of the 8 message fields:

- **SOF:** The Start of Frame (SOF) bit marks the start of message, is a 'dominant 0', and is used to tell the other nodes that a CAN node intends to broadcast information
- **ID(11 or 29-bit):** The ID is the frame identifier and it used to prioritize messages on the bus. The lower the binary value, the higher its priority
- **RTR:** The Remote Transmission Request indicates whether a node sends data or requests dedicated data from another node. The bit is dominant when information is required from another node
- **Control:** The Control contains the Identifier Extension Bit (IDE) which is a 'dominant 0' for 11-bit. The IDE bit determines whether the message is a standard or extended frame. It also contains the 4 bit Data Length Code (DLC) that specifies the length of the data bytes to be transmitted (0 to 8 bytes)
- **Data:** The Data contains the data bytes aka payload, which includes CAN signals that can be extracted and decoded for information so contains the actual information to be communicated, 64 bits of application may be transmitted
- **CRC:** The Cyclic Redundancy Check is used to ensure data integrity, it contains the checksum (number of nits transmitted) of the preceding application data for error detection
- **ACK field:** ACK consists of two bits: ACK slot bit and ACK Del. It is used to indicate whether the message has been received correctly. The first bit (Ack slot) is a time interval in which each bus node, regardless of whether it processes or rejects the received data, puts a 'dominant' bit (logic 0) on the bus if the received data is correct (result of the check on the CRC). The second ACK Del bit is used to close the ACK field
- **-EOF:** The EOF marks the end of the CAN frame

-
- IFS-This 7-bit bit interframe space contains the time required by the controller to move correctly received frame to its proper position in a message buffer area

The CAN ID and payload are the most relevant fields in regards to CAN bus data logging.

1.10 CAN bus protocol Characteristic

1.10.1 Error Frame

Along the bus there can be several causes for which a transmission error or error reception occurs.

The three main ones are:

- Each node can have different sampling point and therefore due to a disturbance it may occur that two nodes read two different logic levels
- Two different nodes can have different thresholds of recognition of logical levels
- The signal along the bus is attenuated and therefore there may be degradation that generate a read error

A nodes sent an error frame when reveals an error in a Remote Frame or Data Frame.

The Error flag consists of 6 bits that can be dominant or recessive, resulting in two types of Error Flag:

- Active Error Flag, 6 dominant bits is sent if the node has an Error status Active
- Passive Error Flag, 6 recessive bits that can be overwritten by an Active Error Flag but cannot overwrite the ACK and EOF fields.

Error handling

The error detecting procedure and the consequent isolation or faults make CAN very reliable.

Error detecting is divided into the following steps:

-
- The CAN controller of a node detects an error
 - Immediately send an Error Frame
 - The corrupted message is ignored by all nodes on the network
 - The CAN controller updates its status
 - The message is retransmitted (eventually it will have to compete, for access to the bus, with other messages)

The CAN protocol defined different kind of error:

- Monitoring: each node compares the bits it sends with those actually present on the bus and in case of discrepancy there is a BIT error
- CRC: each receiver node computes the CRC sequence corresponding to the received message and compares it with the one that the transmitter has queued to the message itself. If there is a difference between the two sequences, there is a CRC error
- Frame control: if the receiver detects that they are not the 5 possible CAN frame structures have been respected, generates an Error
- Sending the acknowledgment bit: each node that receives correctly a D.F. or a R.F. is required to send a bit dominant in the ACK field of this frame. If the overwrite does not take place, the ACK slot bit remains recessive and the transmitter sends an error message: there is Acknowledgment Error only if none of the other nodes send a dominant bit

1.11 How to log CAN bus data

To truly understand the CAN bus protocol, we find it useful to explain how to record and decode CAN bus data. CAN data acquisition is done across many use cases today. For example, CAN data from cars can be used to reduce fuel cost, improve driving, test prototype parts – and for insurance optimization.

Similarly, CAN data from trucks, buses, tractors etc. can be used in fleet management to reduce costs or improve safety.

Another increasingly popular use case for CAN data is predictive maintenance. Vehicles and machinery can be monitored via IoT CAN loggers that communicate data to remote servers. This data can in turn be used to predict and avoid breakdowns.

Finally, CAN loggers are increasingly being used as blackboxes in vehicles and machinery, providing data for diagnostics and for example warranty dispute handling.

1.11.1 But how do you actually record CAN bus data?

As mentioned, two CAN fields are important for CAN logging: The CAN ID and the Data.

A common way to record CAN bus data in practice is to use CAN bus data logger like the CANedge. With this, it is possible record timestamped raw CAN frames into a log file on an SD card.

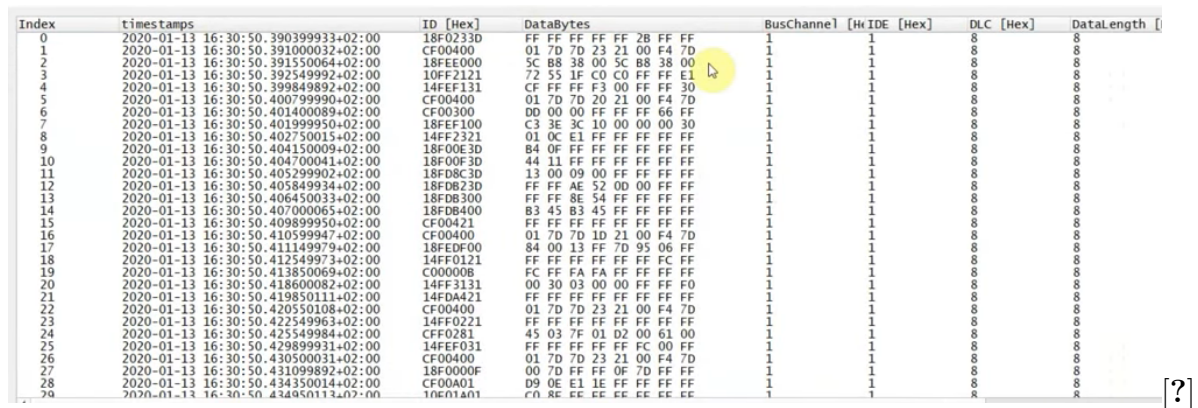
This first step is to connect your CAN logger to your CAN bus. Typically this involves using an adapter cable:

- Cars: In most cars, you simply use an OBD2 adapter to connect. In most cars, this will let you log raw CAN data, as well as perform requests to log OBD2 or UDS (Unified Diagnostic Services) data
- Heavy duty vehicles: To log J1939 data from trucks, excavators, tractors etc you can typically connect to the J1939 CAN bus via a standard J1939 connector cable (deutsch 9-pin)
- Maritime: Most ships/boats use the NMEA 2000 protocol and enable connection via an M12 adapter to log marine data
- CANopen: For CANopen logging, you can often directly use the CiA 303-1 DB9 connector (i.e. the default connector for our CAN loggers), optionally with a CAN bus extension cable

- Contactless: If no connector is available, a typical solution is to use a contactless CAN reader - e.g. the CANCrocodile. This lets you log data directly from the raw CAN twisted wiring harness, without direct connection to the CAN bus (often useful for warranty purposes)
- Other: In practice, countless other connectors are used and often you'll need to create a custom CAN bus adapter - here a generic open-wire adapter is useful

When you've identified the right connector and verified the pin-out, you can connect your CAN logger to start recording raw CAN frames.

To understand what this looks like, let's load a log file.



Index	timestamps	ID [Hex]	DataBytes	BusChannel	[H] IDE [Hex]	DLC [Hex]	DataLength [
0	2020-01-13 16:30:50.390399933+02:00	18F0233D	FF FF FF FF FF 2B FF FF	1	1	8	8
1	2020-01-13 16:30:50.391000032+02:00	CF00400	01 7D 7D 23 21 00 F4 7D	1	1	8	8
2	2020-01-13 16:30:50.391550064+02:00	18FEE000	5C B8 38 00 5C B8 38 00	1	1	8	8
3	2020-01-13 16:30:50.392549992+02:00	10FF2121	72 55 1F C0 C0 FF FF E1	1	1	8	8
4	2020-01-13 16:30:50.399849892+02:00	14FEF131	CF FF FF F3 00 FF FF 30	1	1	8	8
5	2020-01-13 16:30:50.400799990+02:00	CF00400	01 7D 7D 20 21 00 F4 7D	1	1	8	8
6	2020-01-13 16:30:50.401400089+02:00	CF00300	0D 00 00 FF FF FF 66 FF	1	1	8	8
7	2020-01-13 16:30:50.401999950+02:00	18FEF100	C3 3E 3C 10 00 00 00 30	1	1	8	8
8	2020-01-13 16:30:50.402750015+02:00	14FF2321	01 0C E1 FF FF FF FF FF	1	1	8	8
9	2020-01-13 16:30:50.404150009+02:00	18F00E3D	84 0F FF FF FF FF FF FF	1	1	8	8
10	2020-01-13 16:30:50.404700041+02:00	18F00F3D	44 11 FF FF FF FF FF FF	1	1	8	8
11	2020-01-13 16:30:50.405299902+02:00	18FD8C3D	13 00 09 00 FF FF FF FF	1	1	8	8
12	2020-01-13 16:30:50.405849934+02:00	18FD823D	FF FF AE 52 0D 00 FF FF	1	1	8	8
13	2020-01-13 16:30:50.406450033+02:00	18FD8300	FF FF 8E 54 FF FF FF FF	1	1	8	8
14	2020-01-13 16:30:50.407000065+02:00	18FD8400	83 45 B3 45 FF FF FF FF	1	1	8	8
15	2020-01-13 16:30:50.409899950+02:00	CF00421	FF FF FF FF FF FF FF FF	1	1	8	8
16	2020-01-13 16:30:50.410599947+02:00	CF00400	01 7D 7D 1D 21 00 F4 7D	1	1	8	8
17	2020-01-13 16:30:50.411149979+02:00	18FEDF00	84 00 13 FF 7D 95 06 FF	1	1	8	8
18	2020-01-13 16:30:50.412549973+02:00	14FF0121	FF FF FF FF FF FF FC FF	1	1	8	8
19	2020-01-13 16:30:50.413850069+02:00	C00000B	FC FF FA FA FF FF FF FF	1	1	8	8
20	2020-01-13 16:30:50.418600082+02:00	14FF3131	00 30 03 00 00 FF FF F0	1	1	8	8
21	2020-01-13 16:30:50.419850111+02:00	14FD4421	FF FF FF FF FF FF FF FF	1	1	8	8
22	2020-01-13 16:30:50.420550108+02:00	CF00400	01 7D 7D 23 21 00 F4 7D	1	1	8	8
23	2020-01-13 16:30:50.422549963+02:00	14FF0221	FF FF FF FF FF FF FF FF	1	1	8	8
24	2020-01-13 16:30:50.425549984+02:00	CF0281	45 03 7F 01 D2 00 61 00	1	1	8	8
25	2020-01-13 16:30:50.429899931+02:00	14FEF031	FF FF FF FF FF FC 00 FF	1	1	8	8
26	2020-01-13 16:30:50.430500031+02:00	CF00400	01 7D 7D 23 21 00 F4 7D	1	1	8	8
27	2020-01-13 16:30:50.431099892+02:00	18F0000F	00 7D FF FF 0F 7D FF FF	1	1	8	8
28	2020-01-13 16:30:50.434350014+02:00	CF00A01	D9 0E E1 1E FF FF FF FF	1	1	8	8
29	2020-01-13 16:30:50.434950113+02:00	18F01A01	C0 8E FE FE FE FE FE FE	1	1	8	8

Figure 1.4: Log file
This specif log file contains J1939 data.

Using a CAN bus software toll called asammdf, we can view the raw data in a tabular structure and you'll see the link back to the CAN frame field discussed earlier – including in particular the CAN ID and data payload.

If we look at this raw CAN bus data sample, you will probably notice something: Raw CAN bus data is not human-readable.

1.11.2 How can you interpret can bus data

To interpret it, we need to decode the CAN frames into scaled engineering values aka physical values. For example speed as measured in km/h.

To extract the physical value of a CAN signal, the following information is required:

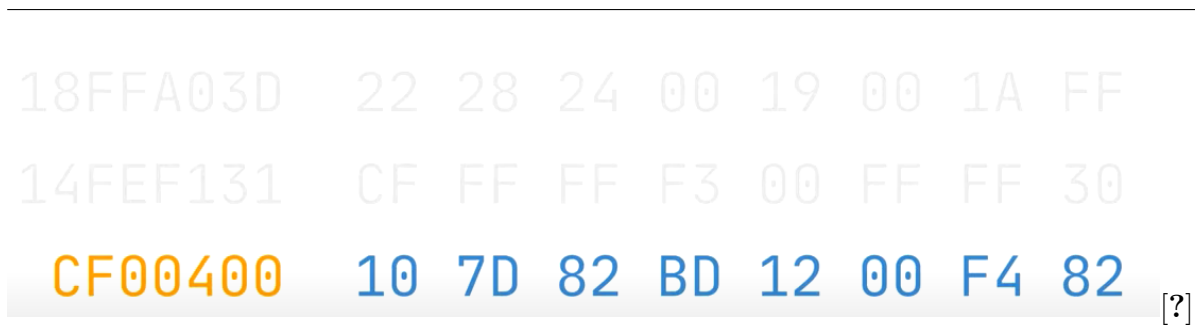


Figure 1.5: CAN data

- Bit start: Which bit the signal starts at
- Bit length: The length of the signal in bits
- Offset: Value to offset the signal value by
- Scale: Value to multiply the signal value by

Let's look at how this works step-by-step: The first thing to understand is that each CAN frame contains a number of CAN signals (or parameters) within the CAN databytes. For example, a CAN frame with a specif CAN ID may carry data for 2 CAN signals as illustrated.

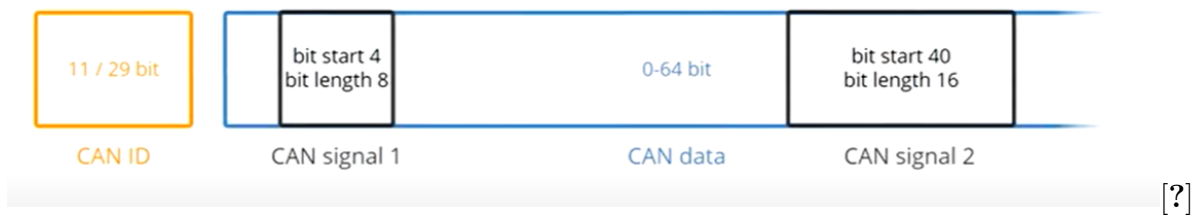


Figure 1.6: CAN frame

To extract a CAN signal, you 'carve out' the relevant bits, take the decimal value and perform a linear scaling:

$$Physical_value = offset + scale * raw_value_decimal$$

To extract the physical value of a CAN signal, a specif set of inforamation is required. In this example, the start bit is 24 and the bit length is 16. The EngineSpeed signal is

CAN ID	Data bytes
0CF00400	FF FF FF 68 13 FF FF FF

[?]

Figure 1.7: Engine speed signal

little-endian and we therefore need to reorder the byte sequence from 6813 to 1368. Next, we convert the hex string to decimal and apply the standard linear conversion used in most CAN decoding.

$$621 \text{ rpm} = 0.125 * 4968 + 0$$

[?]

Figure 1.8: result

In practice, the information required for extracting CAN bus signals is contained in a standardized file format. The most common format is called 'CAN databases'.

In summary, if you have a tool for recording raw CAN bus data and relevant DBC file, then you can extract human-readable data, which you can use for practically any use case.

1.12 How do you get the relevant dbc file?

Most CAN based assets today use a proprietary CAN bus signal encoding.

This is the case in practically all cars, motorcycles, factory machinery etc. Here, the decoding rules are proprietary and only known to the manufacturer – and they differ across different models and brands. However, some exceptions exist: In most cars today, you can record OBD2 data by requesting specific parameters from the car like speed, throttle position etc.

In case the CAN data is proprietary, you can attempt to reverse engineer the data using CAN bus sniffer tools. It is also possible to attempt to leverage public DBC files created through the reverse engineering efforts of others.

1.13 Related protocols

We've mentioned a couple of protocols related to CAN bus.

In simple terms, CAN bus provides the basic means of communicating data – but not a lot more. Therefore a number of 'higher layer protocols' have been developed to enable more advanced functionalities.

Let's briefly outline the most common ones:

- First we have J1939, which is the standard in-vehicle network for heavy-duty vehicles. J1939 parameters are identified by a suspect parameter number or (SPN) and these are bundled in a parameter group identified as a PG number (or PGN). The maritime NMEA 2000 protocol and the agricultural ISOBUS protocol are derived from J1939
- Second, we have OBD2. In short, 'on-board diagnostics' is a self-diagnostic and reporting capability that can for example be used by mechanics to troubleshoot issues in cars. OBD2 specifies diagnostic trouble codes (aka DTCs) and real time data (like speed and RPM), which can be recorded via OBD2 loggers
- Next, we have CANopen which is used widely in embedded control applications, including in industrial automation. It is based on CAN, meaning that a CAN bus data logger is also able to log raw CANopen data. These higher layer protocols will increasingly be based on CAN FD, which is the next generation of Classical CAN.

The CAN FD standard increases the data payload from 8 to 64 bytes and allows for a higher data bit rate. This enables increasingly data-intensive use cases like electric vehicles

1.14 Conclusion

The CAN bus is an intriguing bus that combines the extreme flexibility of a multi-master broadcast protocol with excellent reliability and fault-tolerance. For these reasons, it has seen ever-growing adoption in the industry since the 90s.

In the next chapter we will see example inherent to CAN bus protocol using Matlab with the possibility to analyze CAN bus data.

Chapter 2

Project

In the next chapters, after a brief introduction to the tools and software used, some examples will be discussed to better understand the functioning of the CAN protocol by analyzing some of its characteristics, discussed in the previous chapter, such as a detailed study of the data transported by the CAN frames or the acknowledge.

So the examples that will be shown have the purpose of providing a concrete representation of the CAN protocol by deepening what previously discussed with real values.

First of all we will acquire data using Thingspeak in order to simulate the collection of data carried out by a sensor, which will subsequently be packaged in a CAN frame to be sent through the CAN bus, so as to be able to analyze the transport and provide a simulation of how real data are sent through the protocol by ECUs.

Second, we will simulate a CANbus protocol through Matlab, in the specific we will use MathWorks Virtual Channels tools, that allow us to send CANframe.

Third, we will use CAN Explorer to analyze CANframes and CANbus protocol for each iteration.

Chapter 3

ThingSpeak

3.1 Introduction

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. You can send data to ThingSpeak from your devices, create instant visualizations of live data. [4]

With MATLAB analytics inside ThingSpeak, you can write and execute MATLAB code to perform preprocessing, visualizations, and analyses. ThingSpeak enables engineers and scientists to prototype and build IoT systems without setting up servers or developing web software.[4]

3.2 Create Channel

The first step shows how to create a new channel to collect analyzed data.

- Sign in to ThingSpeak[pagina]
- Click Channels > MyChannels
- Click New channel
- Create a new channel:
 - Name: Dew Point Measurement

-
- Field 1: Temperature
 - Field 2: Humidity
 - Field 3: Dew Point

- Save the Channel

Now your channel is available for future use by clicking Channels > My Channels.[5]

3.3 Collect and Analyze Data

This step shows how to read temperature and humidity data from ThingSpeak channel 12397.

3.3.1 Read Data from a Channel

Read the humidity and temperature from the public WeatherStation channel Fields 3 and 4, and write that data to Fields 2 and 1, respectively, of your Dew Point Measurement channel. Dew point is calculated and written to Field 3.[6]

Use MATLAB Analysis to read, calculate, and write your data:

- Go to the Apps tab, and click MATLAB Analysis.
- New > Custom > Create
- Name field: Dew Point Calculation
- In the MATLAB Code field, enter the following lines of code.

```
1 %You read data from the public ThingSpeak channel 12397 - Weather Station, and write it into your new channel.
2 readChId = 12397;
3 %Add your channel ID AND Write API Key. See readme.
4 writeChId = XXXXXXX;
5 writeKey = 'XXXXXXXXXXXXXXXXXX';
6
7 %Read the humidity and temperature from the public WeatherStation channel Fields 3 and 4
8 [temp,time] = thingSpeakRead(readChId,'Fields',4,'NumPoints',20);
9 humidity = thingSpeakRead(readChId,'Fields',3,'NumPoints',20);
```

Figure 1.1: Read Data

In Figure 1.1 this part of code will be executed:

- ReadChId: Save the public Weather Station channel ID to variables.
- writeChId: Save your channel Id to variables.
- writeKey: Save you Write API Key to a variable.
- thingSpeakread: Read the latest 20 points (NumPoints) of temperature data with timestamps (Field 4) and humidity data (Field 3) from the public Weather Station channel and saves into variables: Temp, Time, humidity. [6]

3.3.2 Calculate the Dew Point

This steps of code calculate the dew point using temperature and humidity readings:

```
11 %Convert the temperature from Fahrenheit to Celsius.
12 tempC = (5/9)*(temp-32);
13
14 %Specify the constants for water vapor (b) and barometric pressure (c).
15 b = 17.62;
16 c= 243.5;
17 %Calculate the dew point in Celsius.
18 gamma = log(humidity/100) + b*tempC./(c+tempC);
19 dewPoint = c*gamma./(b-gamma)
20
21 %Write data to your channel.
22 thingSpeakWrite(writeChId,[temp,humidity,dewPoint],'Fields',[1,2,3],...^M
23 'TimeStamps',time,'Writekey',writeKey);
```

Figure 1.2: Dew Point

In Figure 1.2 this part of code will be executed:

- tempC: Convert the temperature from Fahrenheit to Celsius.
- a,b: Specify the constants for water vapor (b) and barometric pressure (c)
- gamma, dewPointF: Calculate the dew point in Celsius.

- ThingSpeakWrite: Write data(temp, humidity, dewPoint) to your Dew Point Measurement channel (writeChId) into Fields 1,2,3.

The Dew Point Measurement channel now shows charts with channel data from each Field.[6]

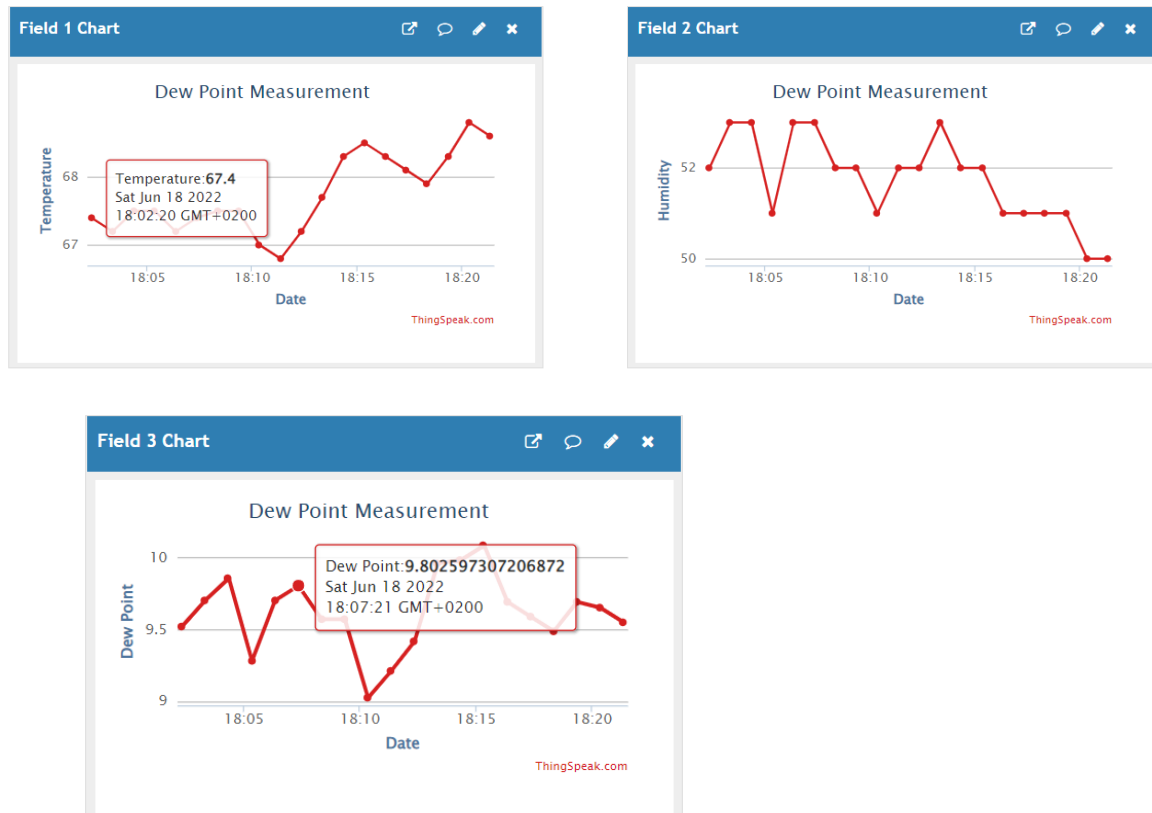


Figure 1.3: Channel Field

3.3.3 Visualize Dew Point Measurement

Use the MATLAB Visualizations app to visualize the measured dew point data, temperature, and humidity from your Dew Point Measurement channel.

Apps > MATLAB Visualization, and click New to create a visualization.

Alternately, you can click MATLAB Visualization in your Dew Point Measurement channel view.[6]

- Select the Custom template and click Create.
- Name the visualization "Dew Point."

-
- Create variables for your Dew Point Measurement channel ID and your Read API Key.
 - Read data from your channel fields.
 - Plot the data

All these steps are performed in the following code:

```
1
2 readChId = XXXXXXXX
3 readKey = 'xxxxxxxxxxxxxxxxxxxx';
4
5 [dewPointData,timeStamps] = thingSpeakRead(readChId,'fields',[1,2,3],...
6     'NumPoints',20,'ReadKey',readKey);
7
8     plot(timeStamps,dewPointData);
9     xlabel('TimeStamps');
10    ylabel('Measured Values');
11    title('Dew Point Measurement');
12    legend({'Temperature','Humidity','Dew Point'});
13    grid on;
```

Figure 1.4: Visualize Dew Point

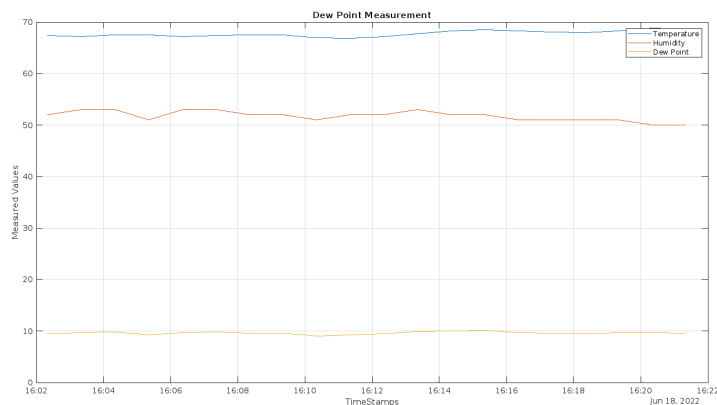


Figure 1.5: the plot output

Chapter 4

MATLAB

This step shows how to simulate a CANbus through MathWorks Virtual Channels.

In the specific we will send CANframes through a virtual CANbus with 2 channels and we will analyze characteristics of CANbus protocol through Simulink.

4.1 Vehicle Network Toolbox

Vehicle Network Toolbox provides MATLAB functions and Simulink blocks for sending, receiving, encoding, and decoding CAN, CAN FD, J1939, and XCP messages. The toolbox simplifies communication with in-vehicle networks and lets you monitor, filter, and analyze live CAN bus data or log and record messages for later analysis and replay. You can simulate message traffic on a virtual CAN bus.[7]

4.1.1 MathWorks Virtual Channels

To facilitate code prototyping and model simulation without hardware, Vehicle Network Toolbox provides a MathWorks virtual CAN device with two channels. These channels are identified with the vendor "MathWorks" and the device "Virtual 1", and are accessible in both MATLAB and Simulink. The two virtual channels belong to a common device, so you could send a message on channel 1 and have that message received on channel 1 and channel 2. Since the virtual device is an application-level representation of a CAN/CAN FD bus without an actual bus, the following limitations apply:[9]

-
- The virtual interface does not perform low level protocol activities like arbitration, error frames, acknowledgment, and so on.
 - Although you can connect multiple channels of the same virtual device in the same MATLAB session or in Simulink models running in that MATLAB session, you cannot use virtual channels to communicate between different MATLAB sessions.

4.2 Preparation of the CAN channel

In this section we will go into more details on the initialization of the virtual channel. These steps will be performed:

- CAN channel initialization: Creation of the two MathWorks virtual CAN channels
- Channel Properties: CAN channel's configuration
- Start the Channels

4.2.1 Create CAN Channels

First of all we create two MathWorks virtual CAN channels that allow us to send CAN-frames from devices to devices:

```
29 %channel initialization: one for transmission(1) and one for reception(2).
30 canch1 = canChannel('MathWorks','Virtual 1',1);
31 canch2 = canChannel('MathWorks','Virtual 1',2);
```

Figure 3.1: CAN channel

canChannel(vendor, device, devicechannelindex)

Let's see specifically what this function does.

CanChannel returns a CAN channel connected to a device from a specified vendor.

- Vendor: CAN device vendor, specified as 'MathWorks', 'kvaser', 'vector'.

-
- Device: CAN device to connect channel to, specified as a character, vector or string. Valid values depend on the specified vendor.
 - index: CAN device channel port or index, specified as a numeric value.
 - Output: CAN device channel, returned as a **can.Channel object**.

The can.Channel object is showed in figure:

```
canch1 =

Channel with properties:

Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'

Status Information
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'

Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000
    SJW: []
    TSEG1: []
    TSEG2: []
    NumOfSamples: []

Other Information
    Database: []
    UserData: []
```

Figure 3.2: CAN channel

This property are used to examine or configure CAN channel setting.

–Device Information:

- The DeviceVendor property indicates the name of the device vendor.
- This property is read-only and display the device type to which the channel is connected

-
- This property is read-only and it indicates the channel index on which the specified CAN is configured
 - This property is read-only and displays the serial number of the device connected to the CAN
 - This property is read-only and it indicates the communication protocol for which the CAN channel is configured, either CAN or CAN FD

–**Status Information:**

- The Running property indicates the state of the CAN according to the following values: - false (default):The channel is offline - true: The channel is online
- The MessagesAvailable property displays the total number of messages available to be received by a CAN channel
- The MessagesReceived property indicates the total number of messages received since the channel was last started
- The MessagesTransmitted property indicates the total number of messages transmitted since the channel was last started
- The InitializationAccess property indicates if the configured CAN channel object has full control of the device channel, according to the following values:
 - true: Has full control.
 - false: Does not have full control
- InitialTimestamp indicates when channel started.
- Indicate settings of message acceptance filters

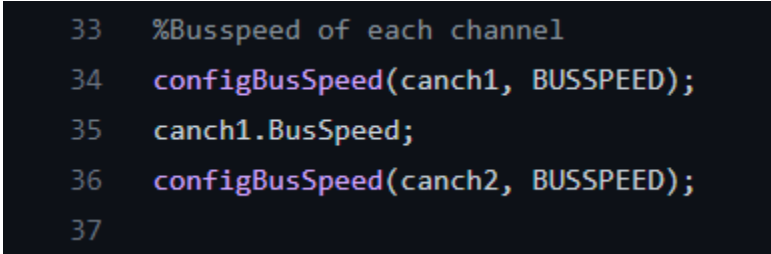
Recap

After a brief introduction on the parameters of the function we explain the figure 3.1 in which we have created 2 channels: canch1 and canch2, both of them we are using MathWorks Vendor. This is mean we are using the virtual channel provided by MathWorks

that allow us to connect a device to the channel, in our example we assume 2 different devices for endpoint, with 2 different channels.

4.2.2 Configure Channel Properties

You can set the behavior of your CAN channel by configuring its property values.[8] In figure 3.2 we are configuring the BUSSPEED for each channel, hence we are setting the speed of the CAN channel.



```
33 %Busspeed of each channel
34 configBusSpeed(canch1, BUSSPEED);
35 canch1.BusSpeed;
36 configBusSpeed(canch2, BUSSPEED);
37
```

Figure 3.3: Configure Channel

`configBusSpeed(canch, busspeed)`

`ConfigBusSpeed(canch,busspeed)` sets the speed of the CAN channel in a direct form that uses baseline bit timing calculation factors. Unless you have specific timing requirements for your CAN connection, use the direct form of `configBusSpeed`. Also note that you can set the bus speed only when the CAN channel is offline. The channel must also have initialization access to the CAN device.

4.2.3 Start the Channels

After you configure their properties, start both channels. Then view the updated status information of the first channel.[8]

Now we are ready to simulate CAN bus protocol: Start function starts the CAN channel 'canch1' and 'canch2' on the CAN bus to send and receive messages.

```
38 %start comunicacion channel
39 start(canch1);
40 start(canch2);
```

Figure 3.4: Channel initialization

4.2.4 Create a Message Object

After we set all the property values as desired and our channels are running, we are ready to transmit and receive messages on the CAN bus.

To transmit a message, create a message object and pack the message with the required data.[8]

In this step we are creating two CAN Message objects that will contain our data measured by the sensor:

- The first one, according to ID (IDDew = 200), transmit the DewPoint measurement
- The second one, according to ID (IDTemp = 300), transmit the temperature measurement
- Both CAN Message object are Standard in the specific: False = Standard, True = Extended
- Both of them have Datalength = uint8 array, with size specified by the data length.

Note, the lowest the ID, the higher is the priority.

```
42 %Message creation according to the ID
43 messageout = canMessage(IDDew, false, 8);
44 messageout2 = canMessage(IDTemp, false, 8);
45
```

Figure 3.5: CAN message

4.2.5 Our 2 CAN Message Object

Now focus on our CAN Message Objects:

```

messageout =
  Message with properties:
    Message Identification
      ProtocolMode: 'CAN'
      ID: 200
      Extended: 0
      Name: ''
    Data Details
      Timestamp: 0
      Data: [0 0 0 0 0 0 0 0]
      Signals: []
      Length: 8
    Protocol Flags
      Error: 0
      Remote: 0
    Other Information
      Database: []
      UserData: []

messageout2 =
  Message with properties:
    Message Identification
      ProtocolMode: 'CAN'
      ID: 300
      Extended: 0
      Name: ''
    Data Details
      Timestamp: 0
      Data: [0 0 0 0 0 0 0 0]
      Signals: []
      Length: 8
    Protocol Flags
      Error: 0
      Remote: 0
    Other Information
      Database: []
      UserData: []

```

Figure 3.6: CAN Object

In figure 3.6 we have created two CAN message Objects with different IDs and same format Standards. Let's see some explanation about function properties in the next section.

canMessage(id, extended, datalength)

CanMessage creates a CAN message object from the raw message information.

–Input Arguments:

- ID of the message, specified as a numeric value. If this ID used an extended format, set the extended argument to true
- extended indicates whether the message ID is standard or extended type, specified as true or false. The logical value true indicates that the ID type is extended, false indicates standard type
- datalength is the length of the message data, specified as an integer value of 0 through 8, inclusive
- messagename is the name of the message definition, specified as a character vector or string

–Output Arguments:

-
- CAN message, returned as a CAN message object, with `can.Message` Properties

–**Message Identification:**

These properties are used to examine or configure message setting.

- The `ProtocolMode` property indicates the communication protocol for which the CAN message is configured, either CAN or CAN FD
- The `ID` property represents a numeric identifier for a CAN message. The values range:
 - 0 through 2047 for a standard identifier
 - 0 through 536,870,911 for an extended identifier
- The `Extended` property is the identifier type for a CAN message.
 - False: The identifier type is standard (11 bits)
 - True: The identifier type is extended (29 bits)
- The `Name` property displays the name of the message, as a character vector value

–**Data Details:**

- The `Timestamp` property displays the time at which the message was received on a CAN channel
- Use the `Data` property to define the raw data in a CAN message. The data is an array of `uint8` values, based on the data length you specify in the message
- The `Signals` property allows you to view and edit decoded signal values defined for a CAN message.
- Length of the CAN message in bytes, specified as a `uint8` value. This indicates the number of elements in the Data vector
- Length code of the CAN FD message data, returned as a `uint8` value

–**Protocol Flags:**

-
- The BRS property indicates that the CAN FD message bit rate switch is set
 - The ESI property indicates that the CAN FD message error state indicator flag is set
 - The Error property indicates if true that the CAN message is an error frame
 - Use the Remote property to specify the CAN message as a remote frame.
 - False(default): The message is not a remote frame
 - True: The message is a remote frame

4.3 Pack Messages: Send and Transmit

Now everything is ready for packing our data sensor into the CAN messages object and send them through the virtual channels.

Let's focus on Figure 3.7 representing our script:

```
%Sending messages through the channel1
for i = 1:a
    %Pack messages
    pack(messageout, double(dewPointData(i)), 0, 64, 'LittleEndian');
    messageout.Data;
    pack(messageout2, double(Temp(i)), 0, 64, 'LittleEndian');
    messageout2.Data;
    %transmit the messages trough channel1
    transmit(canch1, messageout);
    transmit(canch1, messageout2);
    canch1; %Remove the semicolon to show the channel status
end
```

Figure 3.7: CAN message

We are packing our data sensor in the corresponding CAN Messages Objects using the pack function and send them into the channel with the transmit function.

In the specific: After we have created the message object, we pack it with the required data.[8]

- Pack function pack signal data into CAN message.

-
- messageout1/2 - CAN messages name, specified as a CAN message object.
 - double(dewPointData) - Value of signal to pack into message, specified as a numeric value, in our case the Dew Point and temperature measurement
 - startbit = 0 - Signal starting bit in the data, specified as a single or double value. This is the least significant bit position in the signal data
 - signalsize = 64 - Length of the signal in bits, specified as a numeric value
 - byteorder = 'LittleEndian' - Signal byte order format, specified as 'LittleEndian' or 'BigEndian'.
- The instruction messageout.Data return as in output the data field of the CAN Message Object. The first data packed is showed in Figure 3.8.
 - Now we can transmit the packed message. Use the transmit function, supplying the channel canch1 and the message as input arguments.[8]
 - canch1 - CAN channel used for transmit messages
 - messageout/2 - Message to transmit

```
ans =

1×8 uint8 row vector

    23    246    40    21    63     9    35    64

ans =

1×8 uint8 row vector

    154    153    153    153    153    217    80    64
```

Figure 3.8: CAN message.Data (decimal)

In this first example we are assumed that there are no acknowledgement required by the protocol, this is mean that we transmit all the message at the BUSSPEED.

4.4 Transmit And Visualize CAN Data Using CAN Explorer

CAN explorer is configured to receive data using MathWorks Virtual Channel1.

Now we will focus on analyzing messages through CAN open.

4.4.1 Open the CAN explorer

Open the CAN Explorer app using the `canExplorer` command. Alternatively, you can find CAN Explorer in the MATLAB® Apps tab.

4.4.2 Select the Device Channel

When the app first opens, it displays all the accessible CAN channels from devices connected to the system.

Select MathWorks Virtual 1 Channel 1 from the available devices. Then the app finishes opening and looks like this, with the selected device highlighted in a blue outline.

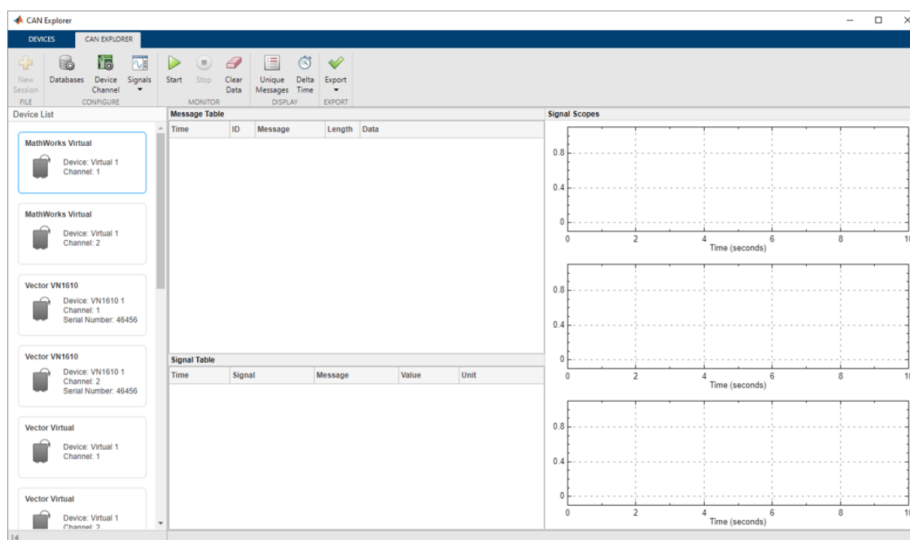


Figure 3.9: Select Device Channel[10]

4.4.3 Configure the Channel Bus Speed

Configure the channel bus speed if the desired network speed differs from the default value.

- To open the Device Channel Configuration dialog, select Device Channel in the toolbar
- This example uses the default bus speed at 500000 bits per second. Confirm the current device channel configuration and click OK

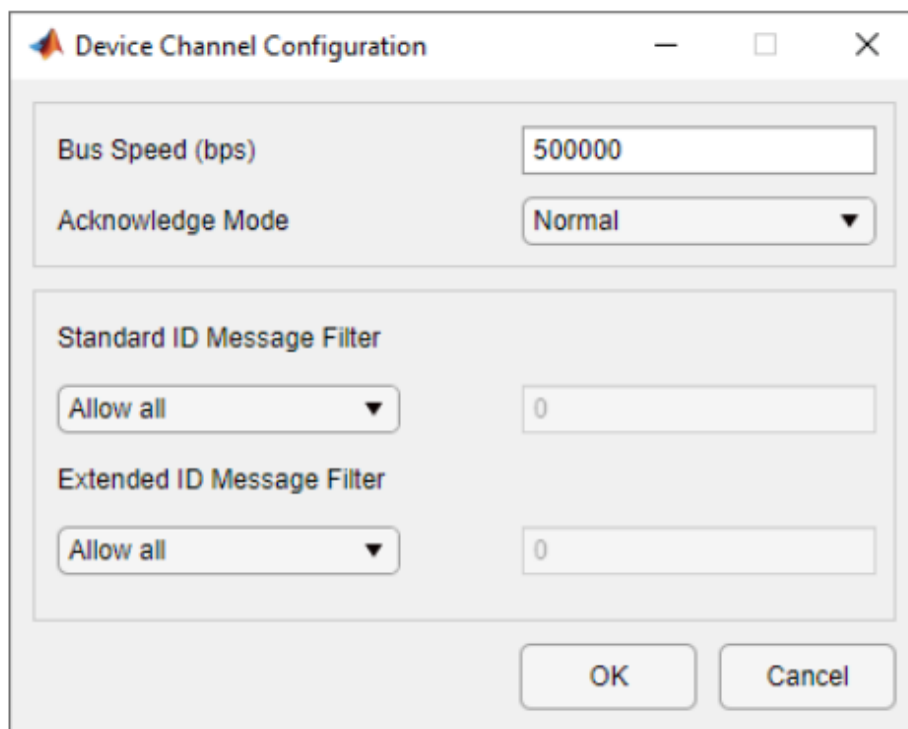


Figure 3.10: Channel Bus Speed [10]

4.4.4 Start Monitoring Transmitted Message

Now we start monitoring the DewPointData transmitted over the canch1 in CAN explorer before starting the replay to avoid losing any data. Click Start in the toolbar.

According to Figure 3.7 we are sending, at each iteration of the FOR Loop, through canch1 the dewpoint and temperature measurement without waiting the acknowledgement

that we discussed in the introduction of CAN bus protocol. This means we are sending all the data at once.

The canch1 channel status and the CAN explorer monitoring at the first iteration are showed in Figure 3.11 and 3.12. As you can see in the virtual channel 1 there are 2 available message given by the first iteration. Can Explorer show us the ID messages, Temperature ID(12C) = 300 and Dewpoint ID(C8) = 200, the conversion from hex to decimal match with the ID that has been defined before.

canch1 =

Channel with properties:

Device Information

```
DeviceVendor: 'MathWorks'
Device: 'Virtual 1'
DeviceChannelIndex: 1
DeviceSerialNumber: 0
ProtocolMode: 'CAN'
```

Status Information

```
Running: 1
MessagesAvailable: 2
MessagesReceived: 0
MessagesTransmitted: 2
InitializationAccess: 0
InitialTimestamp: 14-Sep-2022 17:56:04
FilterHistory: 'Standard ID Filter: Allow All'
```

Channel Information

```
BusStatus: 'N/A'
SilentMode: 0
TransceiverName: 'N/A'
TransceiverState: 'N/A'
ReceiveErrorCount: 0
TransmitErrorCount: 0
BusSpeed: 250000
```

Figure 3.11: Canch1 channel status

If we do a step over the time, our expectation is that the number of message available increases until we send all the data and the CAN Explorer monitor all the messages due

Message Table				
Time	ID	Message	Length	Data
34.057220	12C		8	9A 99 99 99 99 D9 50 40
32.249526	C8		8	17 F6 28 15 3F 09 23 40

Figure 3.12: CAN Explorer status

to the fact we are sending data without waiting for a ACK. Figure 3.13 and 3.14 shows an intermediate step.

```
canch1 =

Channel with properties:

Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'

Status Information
    Running: 1
    MessagesAvailable: 14
    MessagesReceived: 0
    MessagesTransmitted: 14
    InitializationAccess: 0
    InitialTimestamp: 14-Sep-2022 18:14:59
    FilterHistory: 'Standard ID Filter: Allow All'

Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 250000
```

Figure 3.13: Canch1 channel Intermediate status

Message Table				
Time	ID	Message	Length	Data
60.656495	12C		8	00 00 00 00 00 E0 50 40
59.917340	C8		8	0C 39 9C 03 8A 23 23 40
57.705034	12C		8	9A 99 99 99 99 D9 50 40
57.559256	C8		8	53 9F C4 08 EE 9A 23 40
40.364072	12C		8	CD CC CC CC CC CC 50 40
38.588431	C8		8	24 3B 9F 8D 39 66 23 40
35.486931	12C		8	00 00 00 00 00 E0 50 40
34.413403	C8		8	4E 8A 33 99 4D 8F 22 40
32.528891	12C		8	00 00 00 00 00 E0 50 40
32.345860	C8		8	B8 57 58 1D 48 B5 23 40
28.048104	12C		8	CD CC CC CC CC CC 50 40
27.699264	C8		8	24 3B 9F 8D 39 66 23 40
23.406699	12C		8	9A 99 99 99 99 D9 50 40
23.159392	C8		8	17 F6 28 15 3F 09 23 40

Time	ID	Message	Length	Data
37.610022	12C		8	CD CC CC CC CC CC 50 40
36.505956	C8		8	24 3B 9F 8D 39 66 23 40
32.720829	12C		8	00 00 00 00 00 E0 50 40
31.922542	C8		8	4E 8A 33 99 4D 8F 22 40
29.820597	12C		8	00 00 00 00 00 E0 50 40
28.058446	C8		8	B8 57 58 1D 48 B5 23 40
17.858212	12C		8	CD CC CC CC CC CC 50 40
17.658035	C8		8	24 3B 9F 8D 39 66 23 40
15.394991	12C		8	9A 99 99 99 99 D9 50 40
15.313925	C8		8	17 F6 28 15 3F 09 23 40

Figure 3.14: CAN Explorer Intermediate status canch1 and canch2.

In Figure 3.14 is possible to see each message that has been sent according to its timetable from canch1 and all the messages that can be received from canch2. According to our assumption we receive all the message at once.

4.5 Receive and Unpack Messages

In this section we will receive the messages, unpack them, analyze the canch2 status and eventually the data will be monitored by means of CAN Explorer.

Let's focus on figure 3.15 representing the receive part of our script:

The receive function returns a timetable of CAN messages received on the CAN channel. The number of messages returned is less than or equal to messagesrequested contained in the canch channel object.

Input Arguments

- CAN channel = Canch2, specified as a CAN channel object. This is the channel by which WE access the CAN bus

```

for j = 1:(a+b)
    canch2; %Remove the semicolon to show the channel status
    % receive the message trough the channel2
    messagein = receive(canch2,1);
    canch2;
    if messagein.ID == ID Dew
        %Unpack a Message
        value(c) = unpack(messagein, 0, 64, 'LittleEndian', 'double')
        c = c+1;
    elseif messagein.ID == ID Temp
        value2(d)= unpack(messagein, 0, 64, 'LittleEndian', 'double')
        d = d +1;
    end
end
end

```

Figure 3.15: Receive CAN Messages

- `messagerequested` is the maximum number of messages to receive, specified as a positive numeric value or `Inf`

Output Arguments

- `messagein` = CAN messages from the channel, returned as a timetable of messages or an array of CAN message objects

After our channel receives a message the if condition simulate the ID check by the second device, according to the ID we unpack the message in two different array.

Input Arguments

- CAN Message = `messagein`, specified as a CAN message object, from which to unpack the data.
- `Startbit` = 0, signal starting bit in the data, specified as a single or double value. This is the least significant bit position in the signal data.
- `Signal size` = 64, ength of the signal in bits, specified as a numeric value. Accepted values for `signalsize` are from 1 through 64, inclusive.
- `Byte order` = `LittleEndian`, Signal byte order format, specified as `'LittleEndian'` or `'BigEndian'`.

- Datatype = double, Data type of unpacked value, specified as a character vector or string

Output Arguments

- Value = value of message data, returned as a numeric value of the specified data type.

4.5.1 Start Monitoring Received Message

Now we can start monitoring the DewPointData received over the canch2 in CAN Explorer.

Figure 3.16 shows us the canch2 status before receiving the message with all the messages available but no one received and after with one message received. Figure 3.17 shows the messagein object and the first correct DewPointData received and stored in value.

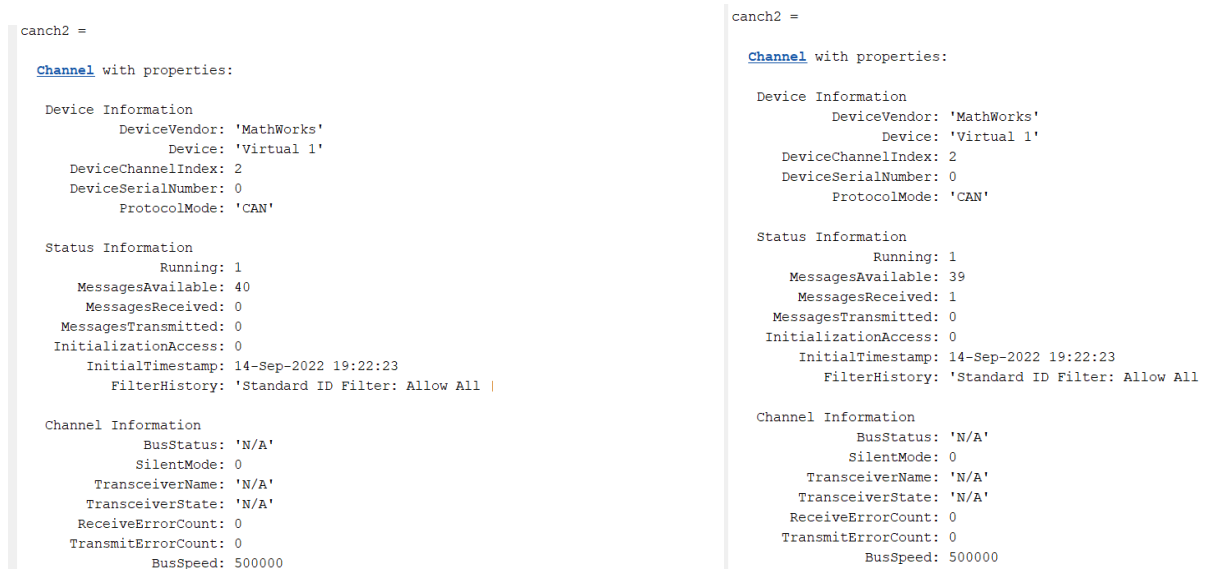


Figure 3.16: canch2 channel before and after receiving first message

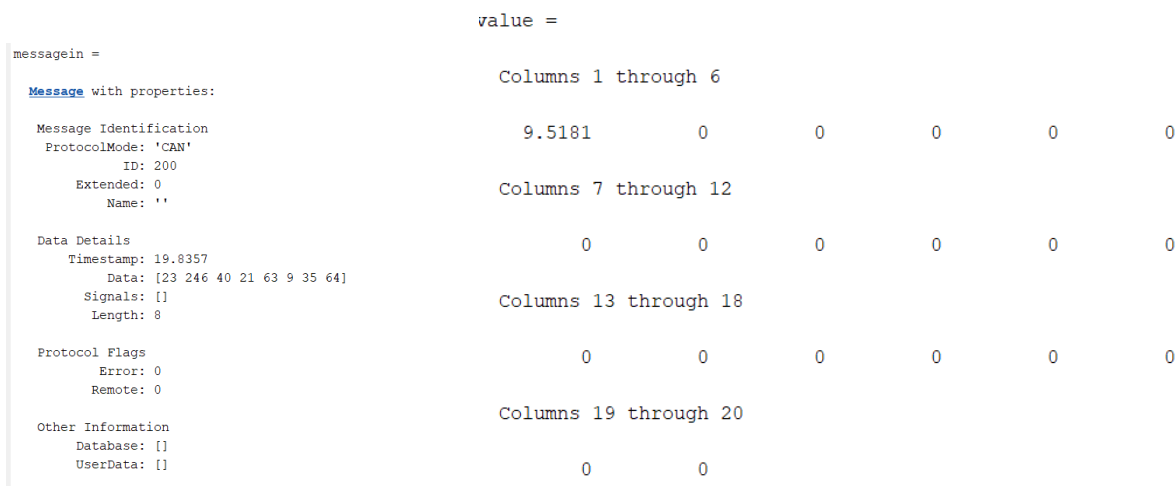


Figure 3.17: Messagein and first value

If we do a step over the time, our expectation is that the number of available messages is decreasing and the number of Received messages is increasing. All the data has been

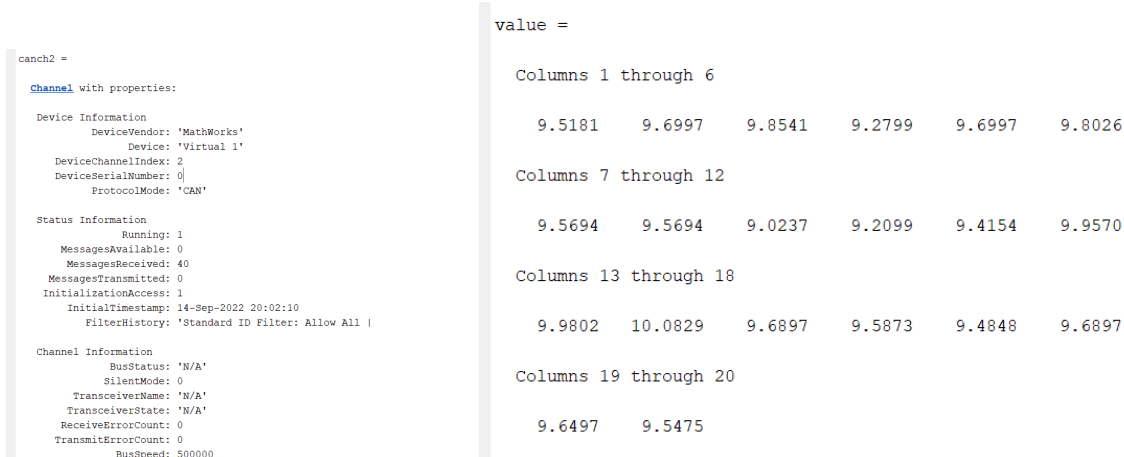


Figure 3.18: canch2 status and value at the last iteration

transmitted correctly from device 1 to device2 through the 2 virtual channel canch1 for sending and canch2 for receiving.

4.6 Conclusion

In this first example we have simulated the functioning of the CAN protocol at a high level, in the specif we try to simulate CAN COMMUNICATION between devices the device 1 has transmitted all the DewPointMeasurement collected by the sensor at once trough the MathWorks Virtual Channel 1 and the second device has received all the data

through the MathWorks Virtual Channel 2 and stored all the data in a vector. This in general how data are broadcasted and shared in CAN bus protocol.

Chapter 5

Second Example

In this new example we try to create a script through Matlab that allow us to represent the acknowledgment of the CAN bus. Mathlab and MathWorks Virtual Channel don't give us the possibility to use property of interest of the CAN protocol, so we try with our own hand to focus on some characteristic as the acknowledgment.

As we said in the CAN bus chapter communicate with messages, each nodes of the system receive all the messages sent on the bus, give the acknowledge and decide if the message should be immediately discarded, so in this example the device 1 send the new DewPointData only after it has received the ACK message from the device 2. We have a two way communication from the nodes, but the ACK is not send using the appropriate field, we simulate it using the ID.

We also introduce the error frame, it is transmitted when a node detects an error in a message according with the many protocol errors defined by CAN Standard, also in this case we don't use the correctly error frame object, MathWork don't allow us, but we use a specific Data Message for error frame.

5.1 ID Messages

We are defining our ID messages for representing CAN bus characteristic as error frame and ACK.

Figure 5.1 show us the specif ID for each kind of CAN Messages Objects and the

```
1  %def ID messages
2  IDTemp = 300;
3  IDHum = 400;
4  IDDew = 500;
5  IDError = 001;
6  ErrorFrame = 00000000;
```

Figure 5.1: ID Messages

specific Data Frame uses for communicate Error.

- IDTemp : Temperature Can Message
- IDHum : Humidity CAN Message
- IDDew ; DewPointData
- IDError: Error Frame ID
- ErrorFrame : Data Frame for IDError messages

5.2 Create CAN Channels

As in the previous example we are creating two different channels, canch1 and canch2, in order to simulate two devices communication. We will not explain this steps anymore.

```
29 %hannel initialization: one for transmission(1) and one for reception(2).
30 canch1 = canChannel('MathWorks','Virtual 1',1);
31 canch2 = canChannel('MathWorks','Virtual 1',2);
```

Figure 5.2: Receive CAN Messages

5.2.1 Start the Channels

Now we can star both channels: canch1 and canch2.

```
38  %start comunicacion channel
39  start(canch1);
40  start(canch2);
```

Figure 5.3: Start Channels

5.2.2 Create a Message Object

After we set all the property value as desired and our channels are running, we are ready to transmit and receive messages on the CAN bus. To transmit a message, create a message objects and pack the message with the required data.

In this step we are creating two CAN Message object that will contain our data measured by the sensor:

- messageoutDwe: It is the CAN message object to transmit the DewPoint measurement
- messageoutTemp : define the CAN Message object to transmit temperature measurement
- messageoutError : define the CAN Message object to transmit Error Frame
- messageACK : define the CAN Message object to transmit ACK.

5.2.3 Our CAN message Object

In Figure 5.4 we have created four CAN Message Object with different ID and same format Standard. The canMessage's property are the same explained in the previous chapter.

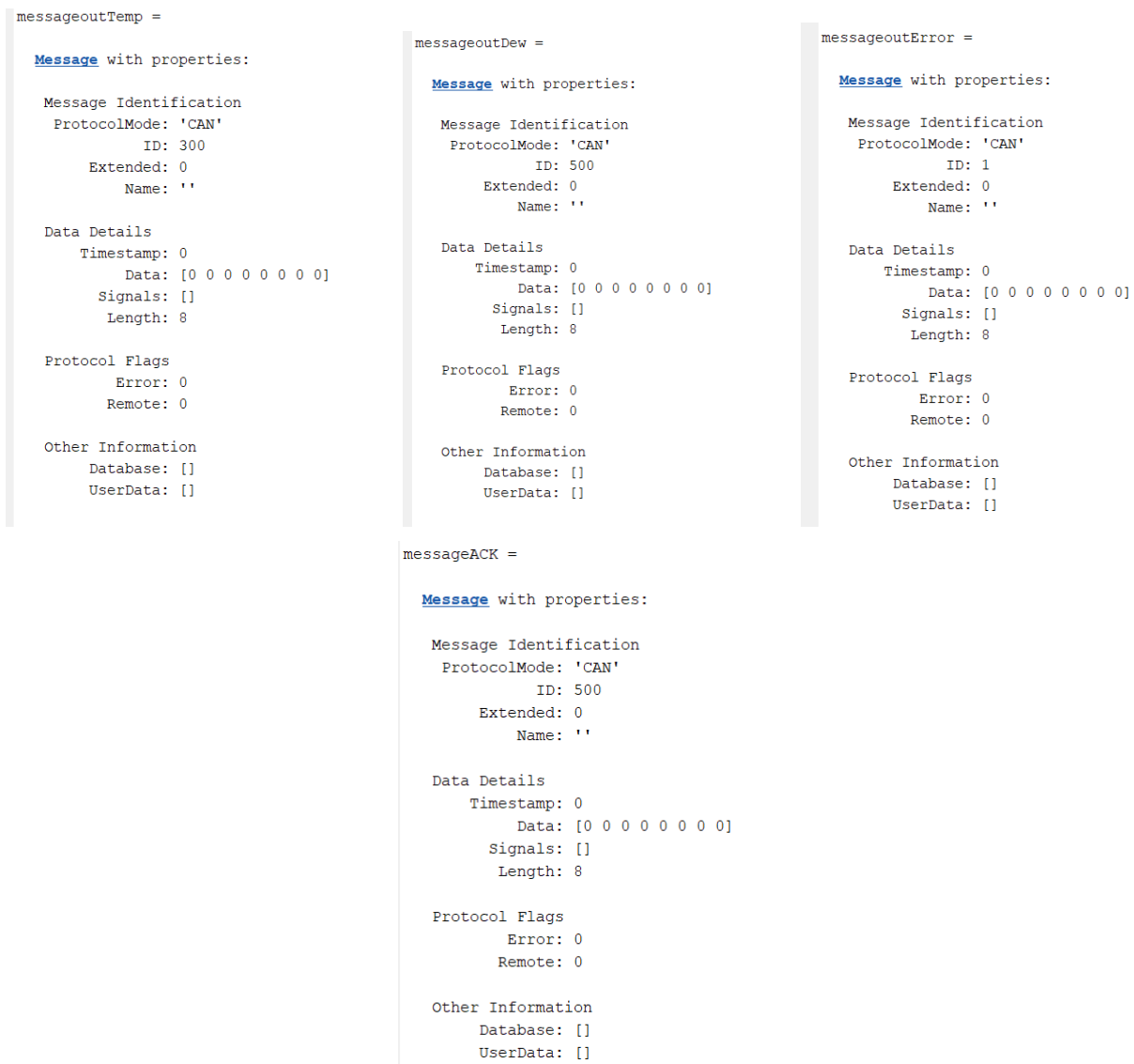


Figure 5.4 CAN Message Objects

5.3 Transmit And Visualize CAN Data Using CAN Explorer

CAN explorer is configured to receive data using MathWorks Virtual Channel1.

Now will focus on analyzing messages through CAN open.

5.3.1 Open the CAN explorer

Open the CAN Explorer app using command `canExplorer`. Alternatively, you can find CAN Explorer in the MATLAB® Apps tab.

5.3.2 Select the Device Channel

When the app first opens, it displays all the accessible CAN channels from devices connected to the system.

Select MathWorks Virtual 1 Channel 1 from the available devices. Then the app finishes opening and looks like this, with the selected device highlighted in a blue outline.

5.3.3 Configure the Channel Bus Speed

Configure the channel bus speed if the desired network speed differs from the default value.

- To open the Device Channel Configuration dialog, select Device Channel in the toolbar
- This example uses the default bus speed at 500000 bits per second. Confirm the current device channel configuration and click OK

5.4 Pack Messages: Send and Transmit

Now we are ready for pack our data sensor into the CAN Messages Object and send them through the virtual channel. The first device transmit a new message only if the second device send an ACK that confirm the reception message. Let's focus on Figure 5.5 A representing our sending code and 5.5 B receive code:


```

43 %send dewdatapoint, wait for ack and send again
44 while n <= a
45     canch1
46     if n == 1 || n == 5
47         pause(0.5)
48         messageinack = receive(canch1,2);
49         if messageinack(1).ID == messageinack(2).ID
50             pack(messageoutDew, double(dewPointData(i)), 0, 64, 'LittleEndian');
51             messageoutDew.Data;
52             transmit(canch1, messageoutDew);
53             i = i+1;
54         elseif messageinack(2).ID == IDError
55             pack(messageoutDew, double(dewPointData(i)), 0, 64, 'LittleEndian');
56             messageoutDew.Data;
57             transmit(canch1, messageoutDew);
58             i = i+1;
59             n = n-1;
60         end
61     elseif n == 5
62         pause(0.5)
63         messageinack = receive(canch1,2);
64         pack(messageoutTemp, 0, 0, 64, 'LittleEndian');
65         messageoutTemp.Data;
66         transmit(canch1, messageoutTemp);
67     else
68         pack(messageoutDew, double(dewPointData(i)), 0, 64, 'LittleEndian');
69         messageoutDew.Data;
70         transmit(canch1, messageoutDew);
71         i = i+1;
72     end
73
79     if n==1
80         messageinDew = receive(canch2,1),
81         canch2
82         value(c) = unpack(messageinDew, 0, 64, 'LittleEndian', 'double');
83         c = c+1;
84         pack(messageACK, 0, 0, 64, 'LittleEndian')
85         messageACK.Data;
86         transmit(canch2, messageACK);
87     elseif n == 1
88         messageinDew = receive(canch2, 2);
89         if messageinDew(2).ID == IDDew
90             %Unpack a Message
91             value(c) = unpack(messageinDew(2), 0, 64, 'LittleEndian', 'double');
92             c = c+1;
93             pack(messageACK, 0, 0, 64, 'LittleEndian')
94             messageACK.Data;
95             transmit(canch2, messageACK);
96         elseif messageinDew(2).ID == IDError
97             pack(messageoutError, ErrorFrame,0, 64, 'LittleEndian');
98             messageoutError.Data;
99             transmit(canch2, messageoutError);
100     end
101 end

```

Figure 5.5: A) Send code B) Receive code

5.4.1 First Iteration

The first IF in row 46 is not executed only in the first and fifth iteration. It is used in order to show an example of error message.

In the first iteration we execute the else statement at row 67 and this means we are packing our data sensor in the corresponding CAN Messages Objects, DewPoint Messages, using the pack function and transmit it within the channel canch1 with the transmit function.(This step is performed from row 67 to row 71)

```

71_ canch1
canch1 =
Channel with properties:
Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'
Status Information
    Running: 1
    MessagesAvailable: 1
    MessagesReceived: 0
    MessagesTransmitted: 1
    InitializationAccess: 1
    InitialTimestamp: 31-Oct-2022 18:12:25
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000

73_ canch2
canch2 =
Channel with properties:
Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 2
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'
Status Information
    Running: 1
    MessagesAvailable: 1
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: 31-Oct-2022 18:12:25
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000

```

Figure 5.6: A) Canch1 Status B) Canch2 Status

Figure 5.6 shows the Canch1 status after sending the first message by the first device.

After sending the message, as showed in figure 5.5 B, the second device receive the

message by running the first IF condition in row 79. Specifically the device receives the message within canch2 (as showed in figure 5.5 B and 5.8) using the receive function, unpack the message and get the first data measurement. Now it is ready to send the ack, packing a new message with the ID = messageACK trough the canch2. The messageACK is a CAN Message Objects with the ID Dew as defined before.(This step is performed from row 79 to row 86)

Hence, we are simulating the ack characteristic through a specific CAN Message Objects: MessageACK.

This step end our first iteration.

Figure 5.7 show us the CAN Message object.

```
messageinDew =  
  
    Message with properties:  
  
    Message Identification  
    ProtocolMode: 'CAN'  
    ID: 500  
    Extended: 0  
    Name: ''  
  
    Data Details  
    Timestamp: 29.8110  
    Data: [23 246 40 21 ... ]  
    Signals: []  
    Length: 8  
  
    Protocol Flags  
    Error: 0  
    Remote: 0  
  
    Other Information  
    Database: []  
    UserData: []
```

Figure 5.7: Message object

In figure 5.8 A is showed the Canch2 status which confirms the receipt of the first message by the second device.

```

canch2 =
  Channel with properties:
    Device Information
      DeviceVendor: 'MathWorks'
      Device: 'Virtual 1'
      DeviceChannelIndex: 2
      DeviceSerialNumber: 0
      ProtocolMode: 'CAN'
    Status Information
      Running: 1
      MessagesAvailable: 0
      MessagesReceived: 1
      MessagesTransmitted: 0
      InitializationAccess: 1
      InitialTimestamp: 31-Oct-2022 18:12:25
      FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
    Channel Information
      BusStatus: 'N/A'
      SilentMode: 0
      TransceiverName: 'N/A'
      TransceiverState: 'N/A'
      ReceiveErrorCount: 0
      TransmitErrorCount: 0
      BusSpeed: 500000

```

Figure 5.8: A) Canch2 Status

Figure 5.9 A shows as the Canch2 status after sending the ACK message at the end of the first iteration. In status information we can see 1 message received(see figure 5,8), 1 message transmitted (the ACK) and 1 message available because according to Matlab virtual channel, the device is single, otherwise in figure 5.9 B is showed the first iteration monitored with CAN Explorer.

As said during the first iteration's explanation, the first device sent a message with ID = 1F4 with the Dew Measures (second row), and the second device sent an ACK with same ID and no data (second row).

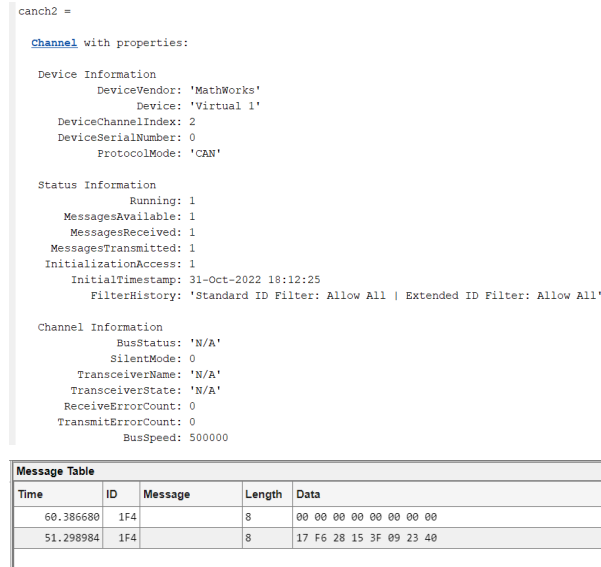


Figure 5.9: A) Canch2 Status and B) CAN explorer after first message after ACK message

5.4.2 Second Iteration

In the second and more iteration excluded the fifth, in figure 5.5 A, we are running the first IF Condition at row 49 and this means that the first device receives the message within canch1 at row 48 and compares the ID with the own message. Note that messaginack is a vector with the previous message and the ACK message. If the ID in the messageACK is the same as the previous message, the first device performs the IF Condition at row 49.

Hence, the first device packs a new DewPoint measurement and transmits it within the canch1. (this step is performed from row 49 to row 53 in figure 5.5 A)

```

canch1 =
  Channel with properties:
    Device Information
      DeviceVendor: 'MathWorks'
      Device: 'Virtual 1'
      DeviceChannelIndex: 1
      DeviceSerialNumber: 0
      ProtocolMode: 'CAN'
    Status Information
      Running: 1
      MessagesAvailable: 2
      MessagesReceived: 0
      MessagesTransmitted: 1
      InitializationAccess: 1
      InitialTimestamp: 31-Oct-2022 18:12:25
      FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
    Channel Information
      BusStatus: 'N/A'
      SilentMode: 0
      TransceiverName: 'N/A'
      TransceiverState: 'N/A'
      ReceiveErrorCount: 0
      TransmitErrorCount: 0
      BusSpeed: 500000
  ...

canch1 =
  Channel with properties:
    Device Information
      DeviceVendor: 'MathWorks'
      Device: 'Virtual 1'
      DeviceChannelIndex: 1
      DeviceSerialNumber: 0
      ProtocolMode: 'CAN'
    Status Information
      Running: 1
      MessagesAvailable: 1
      MessagesReceived: 2
      MessagesTransmitted: 2
      InitializationAccess: 1
      InitialTimestamp: 31-Oct-2022 18:12:25
      FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
    Channel Information
      BusStatus: 'N/A'
      SilentMode: 0
      TransceiverName: 'N/A'
      TransceiverState: 'N/A'
      ReceiveErrorCount: 0
      TransmitErrorCount: 0
      BusSpeed: 500000

```

Figure 5.10 : A) Canch1 Status pre-second iteration B) Canch1 Status post-second iteration

In figure 5.10 A, is showed the Canch1 Status before receiving the ACK Message, in fact in the channel 2 message are available, the previous sent by the first device and the ACK sent by the second device. The figure 5.10 B shows the Canch1 Status after sending the message. (from row 49 to row 53 in figure 5.5 A).

The second device for the remaining iteration performs the ELSEIF Condition at row 87, receives the message within canch2 and performs the IF condition, it unpacks the message, gets the value and send again a new ACK to the first device within canch2. (This step is performed from row 87 to row 95 figure 5.5 B)

In figure 5.11 A and B is showed the Canch2 Status after sending the second ACK message.

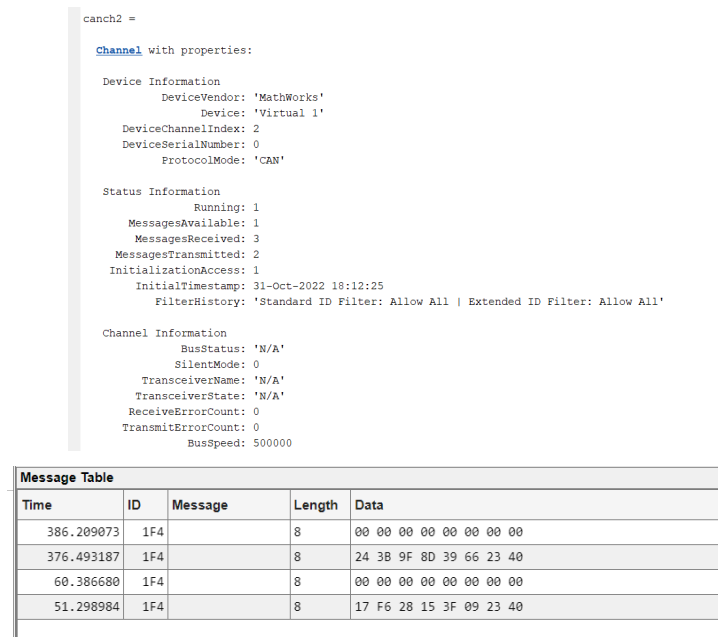


Figure 5.11: A Canch2 Status and B CAN Explorer after second iteration

5.4.3 Fifth Iteration

We arrived at the fifth iteration: In the fifth iteration the first device receive the ACK perform the ELSEIF at row 61 and pack a message with a different ID, in this case the ID Temperature, that it is not expected or negligible, (we knew device must ignore message but in this case we assume that is a corrupted message), by the second device and transmit the message within canch1. (This step is performed row 61 to 66)

The second device perform the ELSEIF Condition at row 96 due to the fact the ID is different from the expected so it pack a CAN Message Object (according to our assumption it assume is corrupted but simply must be ignored) using the ID error and transmit the message through canch2.(This step is performed row 87,88 and from 96 to 99)

Message Table				
Time	ID	Message	Length	Data
540.228545	1		8	00 00 00 00 00 00 00 00
533.568149	12C		8	00 00 00 00 00 00 00 00
525.432821	1F4		8	00 00 00 00 00 00 00 00
515.922802	1F4		8	4E 8A 33 99 4D 8F 22 40
509.771045	1F4		8	00 00 00 00 00 00 00 00
502.635129	1F4		8	B8 57 58 1D 48 B5 23 40
386.209073	1F4		8	00 00 00 00 00 00 00 00
376.493187	1F4		8	24 3B 9F 8D 39 66 23 40
60.386680	1F4		8	00 00 00 00 00 00 00 00
51.298984	1F4		8	17 F6 28 15 3F 09 23 40

Figure 5.12: CAN explore after fifth iteration

5.4.4 sixth iteration

In the next iteration the first device receives the message, unpacks the message and it understands through the ID that there is an error and sends again the old message. (This step is performed from row 46 to 48 and from 54 to 59)

This was a simple example that simulates the characteristic ack and error detection of the CAN bus protocol according to some compromises with the goal to give a provide a better understanding of how the CANBUS protocol works.

Message Table				
Time	ID	Message	Length	Data
753.150920	1F4		8	00 00 00 00 00 00 00 00
742.815363	1F4		8	24 3B 9F 8D 39 66 23 40
540.228545	1		8	00 00 00 00 00 00 00 00
533.568149	12C		8	00 00 00 00 00 00 00 00
525.432821	1F4		8	00 00 00 00 00 00 00 00
515.922802	1F4		8	4E 8A 33 99 4D 8F 22 40
509.771045	1F4		8	00 00 00 00 00 00 00 00
502.635129	1F4		8	B8 57 58 1D 48 B5 23 40
386.209073	1F4		8	00 00 00 00 00 00 00 00
376.493187	1F4		8	24 3B 9F 8D 39 66 23 40
60.386680	1F4		8	00 00 00 00 00 00 00 00
51.298984	1F4		8	17 F6 28 15 3F 09 23 40

Figure 5.13: CAN explorer after sixth iteration

5.4.5 Conclusion

In this second example we try to simulate the main characteristic of the CAN protocol with some compromises due the Virtual Channel. In specific we try to simulate CAN Communication with ACK and error messages between two devices.

The first device has transmitted the DewPoint Measurement collected by the sensor one by one within the MathWorks Virtual Channel 1 after it has received the ACK message that the second device has sent, furthermore we have simulated an error with the corresponding management in order to represent another characteristic of the CAN Protocol.

In figure 5.11 are showed all the monitored message through CAN Explorer. In each row there are time, ID , Length and the data.

canExplorerMsgs_2022_Nov_03_183213						
42x8 timetable						
	Time	1 ID	2 Extended	3 Name	4 Data	5 Length
1	51.299 sec	500	0"		[23,246,40,21,63,9,35,64]	8
2	60.387 sec	500	0"		[0,0,0,0,0,0,0]	8
3	376.49 sec	500	0"		[36,59,159,141,57,102,35,64]	8
4	386.21 sec	500	0"		[0,0,0,0,0,0,0]	8
5	502.64 sec	500	0"		[184,87,88,29,72,181,35,64]	8
6	509.77 sec	500	0"		[0,0,0,0,0,0,0]	8
7	515.92 sec	500	0"		[78,138,51,153,77,143,34,64]	8
8	525.43 sec	500	0"		[0,0,0,0,0,0,0]	8
9	533.57 sec	300	0"		[0,0,0,0,0,0,0]	8
10	540.23 sec	1	0"		[0,0,0,0,0,0,0]	8
11	742.82 sec	500	0"		[36,59,159,141,57,102,35,64]	8
12	753.15 sec	500	0"		[0,0,0,0,0,0,0]	8
13	808.84 sec	500	0"		[83,159,196,8,238,154,35,64]	8
14	809.62 sec	500	0"		[0,0,0,0,0,0,0]	8
15	810.15 sec	500	0"		[12,57,156,3,138,35,35,64]	8
16	810.69 sec	500	0"		[0,0,0,0,0,0,0]	8
17	811.25 sec	500	0"		[12,57,156,3,138,35,35,64]	8
18	811.77 sec	500	0"		[0,0,0,0,0,0,0]	8
19	812.29 sec	500	0"		[10,51,17,197,34,12,34,64]	8
20	812.81 sec	500	0"		[0,0,0,0,0,0,0]	8
21	813.33 sec	500	0"		[131,230,67,49,123,107,34,64]	8
22	813.86 sec	500	0"		[0,0,0,0,0,0,0]	8
23	814.4 sec	500	0"		[137,127,23,228,168,212,34,64]	8
24	814.92 sec	500	0"		[0,0,0,0,0,0,0]	8
25	815.45 sec	500	0"		[189,169,130,244,251,233,35,64]	8
26	815.98 sec	500	0"		[0,0,0,0,0,0,0]	8
27	816.5 sec	500	0"		[76,48,74,133,221,245,35,64]	8
	Time	1 ID	2 Extended	3 Name	4 Data	5 Length
19	812.29 sec	500	0"		[10,51,17,197,34,12,34,64]	8
20	812.81 sec	500	0"		[0,0,0,0,0,0,0]	8
21	813.33 sec	500	0"		[131,230,67,49,123,107,34,64]	8
22	813.86 sec	500	0"		[0,0,0,0,0,0,0]	8
23	814.4 sec	500	0"		[137,127,23,228,168,212,34,64]	8
24	814.92 sec	500	0"		[0,0,0,0,0,0,0]	8
25	815.45 sec	500	0"		[189,169,130,244,251,233,35,64]	8
26	815.98 sec	500	0"		[0,0,0,0,0,0,0]	8
27	816.5 sec	500	0"		[76,48,74,133,221,245,35,64]	8
28	817.02 sec	500	0"		[0,0,0,0,0,0,0]	8
29	817.55 sec	500	0"		[173,104,51,77,113,42,36,64]	8
30	818.07 sec	500	0"		[0,0,0,0,0,0,0]	8
31	818.59 sec	500	0"		[124,87,108,224,37,97,35,64]	8
32	819.12 sec	500	0"		[0,0,0,0,0,0,0]	8
33	819.64 sec	500	0"		[130,238,84,123,176,44,35,64]	8
34	820.16 sec	500	0"		[0,0,0,0,0,0,0]	8
35	820.72 sec	500	0"		[98,251,25,163,58,248,34,64]	8
36	821.24 sec	500	0"		[0,0,0,0,0,0,0]	8
37	821.78 sec	500	0"		[124,87,108,224,37,97,35,64]	8
38	822.3 sec	500	0"		[0,0,0,0,0,0,0]	8
39	822.83 sec	500	0"		[180,17,127,51,171,76,35,64]	8
40	823.36 sec	500	0"		[0,0,0,0,0,0,0]	8
41	823.89 sec	500	0"		[220,121,14,73,86,24,35,64]	8
42	824.41 sec	500	0"		[0,0,0,0,0,0,0]	8

Figure 5.11: CAN Explorer

Chapter 6

DBCCan File

During our introduction of CAN protocol, we have talked about CAN Database file. In this chapter we are going to see a new example in order to better understand what a DBC File is and how to read it.

In the specific we load a DBC-file on Matlab and we simulate a CAN communication through Matlab Virtual Channel that we already saw in the previous example.

This example has been performed using Matlab.

6.1 Use DBC-File in CAN Communication

This example shows how to create, receive and process messages using information stored in DBC-files.

6.2 Open the DBC-File

We are going to use a DBCfile already present in Matlab examples folder. We include this DBC-file in our project folder.

First of all open file "demoVNTCANdbFiles.dbc" using the function canDatabase.

6.2.1 canDatabase

canDatabase function allows us to create a handle to the specified database file. You can specify a file name, a full path or a relative path.[11]

CAN database, returned as a database object with can Database Properties.

6.2.2 Code

In figure 6.1 A shows the code that resumes what we said until now, instead in figure 6.1 B is showed the DBC-file objects.

```
%Open DBC-Files
db = canDatabase("demoVNT_CANdbFiles.dbc")
db.Messages
```



```
db =
Database with properties:
    Name: 'demoVNT_CANdbFiles'
    Path: 'C:\Program Files\MATLAB\R2022a\examples\vnt\data\demoVNT_CANdbFiles.dbc'
    UTF8_File: 'C:\Program Files\MATLAB\R2022a\examples\vnt\data\demoVNT_CANdbFiles.dbc'
    Nodes: {}
    NodeInfo: [0x0 struct]
    Messages: {5x1 cell}
    MessageInfo: [5x1 struct]
    Attributes: {}
    AttributeInfo: [0x0 struct]
    UserData: []
```

Figure 6.1: A) Code B) DBC-File object

Now examine the Messages property to see the names of all messages defined in this file using db.Messages.

```
ans =
5x1 cell array
    'DoorControlMsg'
    'EngineMsg'
    'SunroofControlMsg'
    'TransmissionMsg'
    'WindowControlMsg'
```

Figure 6.2: db.Messages

6.3 View Message Information

Use the function `messageInfo` to view information for message `EngineMsg` including the identifier data length and signal list.

6.3.1 `messageInfo`

The `messageInfo` function returns a structure with information about the CAN messages in the specified database `candb`(sixth row and third image in figure 6.3), if we add the message's name it returns also information about the specified message(fifth row and second image in figure 6.3).

This step is performed in figure 6.3.

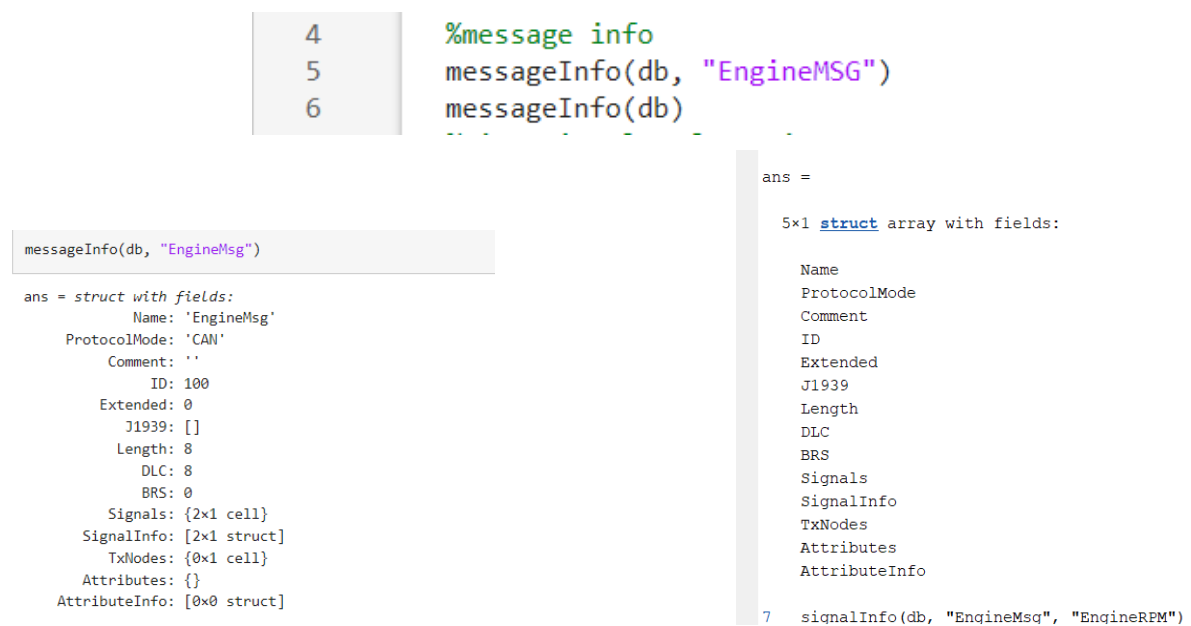


Figure 6.3: Message Info: Engine and structure information

6.4 View Signal Information

Use `signalInfo` function to view information for signal "EngineRPM" in CAN Message `EngineMsg`, including type, byte ordering size, and scaling values that translate raw signals to physical values.[11]

This is shown in Figure 6.4.

```

7 %view signal information
8 signalInfo(db, "EngineMsg", "EngineRPM")

```

```

ans =

  struct with fields:

    Name: 'EngineRPM'
    Comment: ''
    StartBit: 0
    SignalSize: 32
    ByteOrder: 'LittleEndian'
    Signed: 0
    ValueType: 'Integer'
    Class: 'uint32'
    Factor: 0.1000
    Offset: 250
    Minimum: 250
    Maximum: 9500
    Units: 'rpm'
    ValueTable: [0x1 struct]
    Multiplexor: 0
    Multiplexed: 0
    MultiplexMode: 0
    RxNodes: {0x1 cell}
    Attributes: {}
    AttributeInfo: [0x0 struct]

```

Figure 6.4: Message Signal Information

It is also possible query information on all signals in message EngineMsg at once as for example the factor or the offset. This is shown in Figure 6.5

```

10 signalInfo(db, "EngineMsg")

```

```

ans =

  2x1 struct array with fields:

    Name
    Comment
    StartBit
    SignalSize
    ByteOrder
    Signed
    ValueType
    Class
    Factor
    Offset
    Minimum
    Maximum
    Units
    ValueTable
    Multiplexor
    Multiplexed
    MultiplexMode
    RxNodes
    Attributes
    AttributeInfo

```

Figure 6.5: Message Info: Engine and structure information

6.5 Create a Message Using Database Definitions

Now, we create a new message by specifying the database and the message name Enginemsg to have the database definition applied. CAN signals in this message are represented in engineering units in addition to the raw data bytes.[11]

Figure 6.6 A show the code and figure 6.6 B show the CAN message object, in the specific is shown the ID, Protocol Mode, Name and Data.

```

11 %create Message Using Database Definitions
12 msgEngineInfo = canMessage(db, "EngineMsg")

```

```

msgEngineInfo =

  Message with properties:

    Message Identification
      ProtocolMode: 'CAN'
      ID: 100
      Extended: 0
      Name: 'EngineMsg'

    Data Details
      Timestamp: 0
      Data: [0 0 0 0 0 0 0 0]
      Signals: [1x1 struct]
      Length: 8

    Protocol Flags
      Error: 0
      Remote: 0

    Other Information
      Database: [1x1 can.Database]
      UserData: []

```

Figure 6.6: A) CAN Message Code B) CAN Message object

6.6 View New Signal Information

We can use the Signals property to see signal values for the specific message. We can directly write to and read from these signals to pack and unpack data from the message.[11]

In Figure 6.7 is shown the code and the two value contained into MsgEngine CAN message: VehicleSpeed and EngineRPM.

```

13 %view Signal Information
14 msgEngineInfo.Signals

```

```

ans =

  struct with fields:

    VehicleSpeed: 0
    EngineRPM: 250

```

Figure 6.7: CAN signal Information

6.7 Change Signal Information

In Figure 6.8 is shown the code that allow us to change Signal Information EngineRPM, vehicleSpeed and view it.

```

15 %change Signal Information
16 msgEngineInfo.Signals.EngineRPM = 5000
17 msgEngineInfo.Signals
18 msgEngineInfo.Signals.VehicleSpeed = 63
19 msgEngineInfo.Signals

```

Figure 6.8: New signal Code

We can write directly to the signal EngineRPM to change its value, in this case 5000. (16 and 17 row) In Figure 6.9 shown the CAN message object with the new data, note the EngineRPM has been update with the written value 5000.

```

msgEngineInfo =
  Message with properties:
    Message Identification
      ProtocolMode: 'CAN'
      ID: 100
      Extended: 0
      Name: 'EngineMsg'
    Data Details
      Timestamp: 0
      Data: [140 185 0 0 0 0 0 0]
      Signals: [1x1 struct]
      Length: 8
    Protocol Flags
      Error: 0
      Remote: 0
    Other Information
      Database: [1x1 can.Database]
      UserData: []

```

Figure 6.9: New Signal Information

Figure 6.10 show the current signal values back and note that EngineRPM has been update with the written value as we said:

```
ans =  
  
    struct with fields:  
  
    VehicleSpeed: 0  
    EngineRPM: 5000
```

Figure 6.10:

It also important know when a value is written to the signal as in this case, it is translated, scaled and packed into the message data using the database definition. Now we are written a new value to the vehicleSpeed signal.

Figure 6.11 shown the vehicleSpeed CAN Message with the new value.

```
msgEngineInfo =  
  
    Message with properties:  
  
    Message Identification  
    ProtocolMode: 'CAN'  
    ID: 100  
    Extended: 0  
    Name: 'EngineMsg'  
  
    Data Details  
    Timestamp: 0  
    Data: [140 185 0 0 63 ... ]  
    Signals: [1x1 struct]  
    Length: 8  
  
    Protocol Flags  
    Error: 0  
    Remote: 0  
  
    Other Information  
    Database: [1x1 can.Database]  
    UserData: []
```

Figure 6.11:

Figure 6.12 shown the current signal value, note that vehicleSpeed has been updated with the written value as we said:


```
ans =

  struct with fields:

    VehicleSpeed: 63
    EngineRPM: 5000
```

Figure 6.12:

6.8 receive Messages with Database Information

Now after some analysis about the CAN messages object contained into DBC-file we can attach a database to a CAN channel that receives messages to apply database definitions to incoming messages automatically. It is important to know the database decode only messages that are defined. All other messages are received in their raw form.

In this step we are creating the channel as usual(row21) and attach our DBC-file for the MathWorks Virtual Channel 1 (row22) as shown in figure.

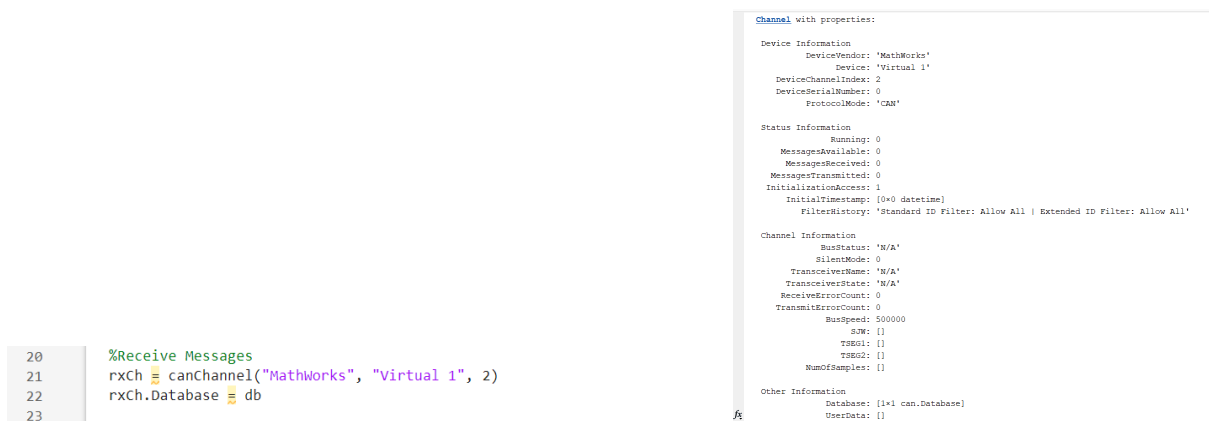


Figure 6.13: A) CAN Channel Initialization B) Channel's Structure

6.9 Receive Messages

Now start the channel (row24), generate some messages traffic (row 25) and receive messages with physical message decoding. (row 26)

The function generateMsgsDB is shown in Figure 6.14, it is a function provided by

Matlab for creates and transmits CAN messages for demo purposes, must be added to the folder, this function periodically transmits multiple CAN messages at various periodic rates with changing message data.

- It access the database file used for the example that is provided by Matlab, create the messages to send using the canMessage function, this function build CAN message based on user-specified structure, in this case the structure provided by the DBC-file shown in Figure 6.2 (canMessage was also explained in the previous example)
- Next it register each message on the channel at a specified period rate
- Run for several seconds increasing the message data regularly
- Set new signal data randomly
- At the end it stops the channel and returns the monitoring channel

In figure 6.15 is shown the first rows of received messages created. (row 28)

```

24 start(rxCh);
25 generateMsgsDb();
26 rxMsg = receive(rxCh, Inf, "OutputFormat", "timetable")
27 %View the first few rowa of received messages
28 head(rxMsg)
29

```

Figure 6.14: Code for creating and monitoring messages

ans =

8x8 timetable

Time	ID	Extended	Name	Data	Length	Signals	Error	Remote
2.0831 sec	100	false	('EngineMsg')	[[0 0 0 0 0 0 0 0]]	8	(1x1 struct)	false	false
2.0831 sec	200	false	('TransmissionMsg')	[[0 0 0 0 0 0 0 0]]	8	(1x1 struct)	false	false
2.0831 sec	400	false	('DoorControlMsg')	[[0 0 0 0 0 0 0 0]]	8	(1x1 struct)	false	false
2.0831 sec	600	false	('WindowControlMsg')	[[0 0 0 0]]	4	(1x1 struct)	false	false
2.0831 sec	800	false	('SunroofControlMsg')	[[0 0 0 0]]	2	(1x1 struct)	false	false
2.1144 sec	100	false	('EngineMsg')	[[0 0 0 0 0 0 0 0]]	8	(1x1 struct)	false	false
2.1301 sec	200	false	('TransmissionMsg')	[[4 0 0 0 0 0 0 0]]	8	(1x1 struct)	false	false
2.1457 sec	100	false	('EngineMsg')	[[177 128 0 0 53 0 0 0]]	8	(1x1 struct)	false	false

Figure 6.15: First rows of received messages

Now we can end the simulation of received messages by stopping the channel.

```

30      %stop the channel
31      stop(rxCh);
32      clear rxCh
33

```

Figure 6.16: Stop channel

6.10 Examine a Received Message

Now we are ready to analyze the CAN Communication that we are performed using DBC-file.

```

34      %examine a Received Message
35      rxMsg(10, :);
36      rxMsg.Signals(10);
37

```

Figure 6.17: Code

First of all we inspect a received message to see the applied database decoding.

```

ans =
1x8 timetable

```

Time	ID	Extended	Name	Data	Length	Signals	Error	Remote
1.7727 sec	200	false	{'TransmissionMsg'}	{[0 0 0 0 0 0 0 0]}	8	{1x1 struct}	false	false

Figure 6.18: First Message

It is possible to derive the value contained in the CAN Message if we know the offset, the factor as illustrated in theory chapter, hence the first CAN Message contains $\text{EngineRPM} = 250$.

$$\text{EngineRPM} = \text{offset}(250) + \text{DataDecimal}(0) * \text{factor}(0.1000) = 250$$

This data for the formula has been shown in Figure 6.4 and the result in Figure 6.7. More information will be shown later during the analysis with the CAN Explorer.

6.11 Extract All Instances of a Specified Message

Now, we can extract all instances of message EngineMsg:

```

38 %extract All Instances of a Specified Message
39 allMsgEngine = rxMsg(strcmpi("EngineMsg", rxMsg.Name), :);
40 head(allMsgEngine)
41

```

Figure 6.19: Code for Extract All Instances

This are the first few instances of this specific message. (row 40)

```

ans =
8x8 timetable

```

Time	ID	Extended	Name	Data	Length	Signals	Error	Remote
0.094806 sec	100	false	{'EngineMsg'}	{[0 0 0 0 0 0 0 0]}	8	{1x1 struct}	false	false
0.12257 sec	100	false	{'EngineMsg'}	{[0 0 0 0 0 0 0 0]}	8	{1x1 struct}	false	false
0.14977 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false
0.18809 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false
0.21777 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false
0.24834 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false
0.2803 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false
0.31421 sec	100	false	{'EngineMsg'}	{[177 128 0 0 53 0 0 0]}	8	{1x1 struct}	false	false

Figure 6.20: Engine Head

6.12 Extract All the Signal Value

In this step we use canSignalTimetable to repackage signal data from message EngineMsg into a signal timetable, in such a way we can plot the vehicle speed that it was sent through the CAN Channel.

```

42 %plot Physical Signal Values
43 signalTimetable = canSignalTimetable(rxMsg, "EngineMsg");
44 head(signalTimetable)
45

```

Figure 6.21: Code for Plot

View the first rows of the signal timetable.

```
ans =  
8x2 timetable  
  
      Time      VehicleSpeed      EngineRPM  
      _____      _____      _____  
0.1145 sec           0           250  
0.14815 sec          0           250  
0.18323 sec          0           250  
0.21317 sec          0           250  
0.24361 sec          0           250  
0.26829 sec          50          3569.6  
0.3007 sec           50          3569.6  
0.33857 sec          50          3569.6
```

Figure 6.22: Head Vehicle Speed

6.13 CAN Explorer

In Figure 6.23 are shown some of the data sent through the Virtual Channel and monitored by the CAN Explorer. In each row there are time, ID, Length and the data.

Is possible derive the value of EngineRPM and VehicleSpeed contained in Figure6.23, first of all we need to convert data in decimal and using the following formula:

$$\text{EngineRPM} = \text{offset}(250) + \text{DataDecimal}(X) * \text{factor}(0.1000)$$

Examine different example generated during the simulation:

2.068300	64	8	AC 81 00 00 32 00 00 00
2.689739	64	8	B1 83 00 00 37 00 00 00
3.082196	64	8	5B 85 00 00 37 00 00 00

Figure 6.23: Example

Engine(hex)	Engine(dec)	Speed(hex)	Speed(dec)
81 AC	33196	32	50
83 B1	33713	37	55
85 5B	34139	37	55

Figure 6.23: Example

As said before the 8 bit most significant are the EngineRPM and the less significant 8 bit are the VehicleSpeed. Now we need to use the formula, the offset and factor was shown in Figure 6.4, the DataDecimal is given starting from the hexadecimal Data and according to the byte order, in this case Little-Endian so we need to switch AC and 81.

This is mean we convert 81 AC in decimal 33196 and we use the formula:

EngineRPM = offset(250) + DataDecimal(33196)*factor(0.1000)= 3569.6

VehicleSpeed 32 = 50km/h

EngineRPM = offset(250) + DataDecimal(33713)*factor(0.1000)= 3621.3

VehicleSpeed 37 = 55km/h

EngineRPM = offset(250) + DataDecimal(34139)*factor(0.1000)= 3663.9

VehicleSpeed 37 = 55km/h

Message Table				
Time	ID	Message	Length	Data
2.120770	04		8	AC 01 00 00 32 00 00 00
2.121943	C8		8	04 00 00 00 00 00 00 00
2.100704	64		8	AC 81 00 00 32 00 00 00
2.068300	64		8	AC 81 00 00 32 00 00 00
2.061470	C8		8	04 00 00 00 00 00 00 00
2.043622	190		8	02 00 00 00 00 00 00 00
2.043619	64		8	00 00 00 00 00 00 00 00
2.013176	C8		8	00 00 00 00 00 00 00 00
2.013173	64		8	00 00 00 00 00 00 00 00
1.983237	64		8	00 00 00 00 00 00 00 00
1.964890	C8		8	00 00 00 00 00 00 00 00
1.948156	64		8	00 00 00 00 00 00 00 00
1.914522	320		2	00 00
1.914518	258		4	00 00 00 00
1.914514	190		8	00 00 00 00 00 00 00 00
1.914510	C8		8	00 00 00 00 00 00 00 00
1.914506	64		8	00 00 00 00 00 00 00 00

Figure 6.24: CAN Explorer

6.14 Plot Physical Signal Values

Now we are ready for viewing the data contained in the canSignalTimetable and in specif we focus on the VehicleSpeed. This value was randomly generated using the function generateMsgDb and sent within the CAN Virtual Channel using the CAN Message objects

defined inside of DBC-Files.

In Figure 6.25 is shown the code for the plot using the signalT timetable VehicleSpeed.

```
46 %plot the values of signal VehicleSpeed
47 plot(signalT timetable.Time, signalT timetable.VehicleSpeed)
48 title("Vehicle Speed from EngineMsg", "FontWeight", "bold")
49 xlabel("Timestamp")
50 ylabel("Vehicle Speed")
51
```

Figure 6.25: Code for plot

Now we can plot values of EngineRPM signal over time. The result is shown in Figure 6.26.

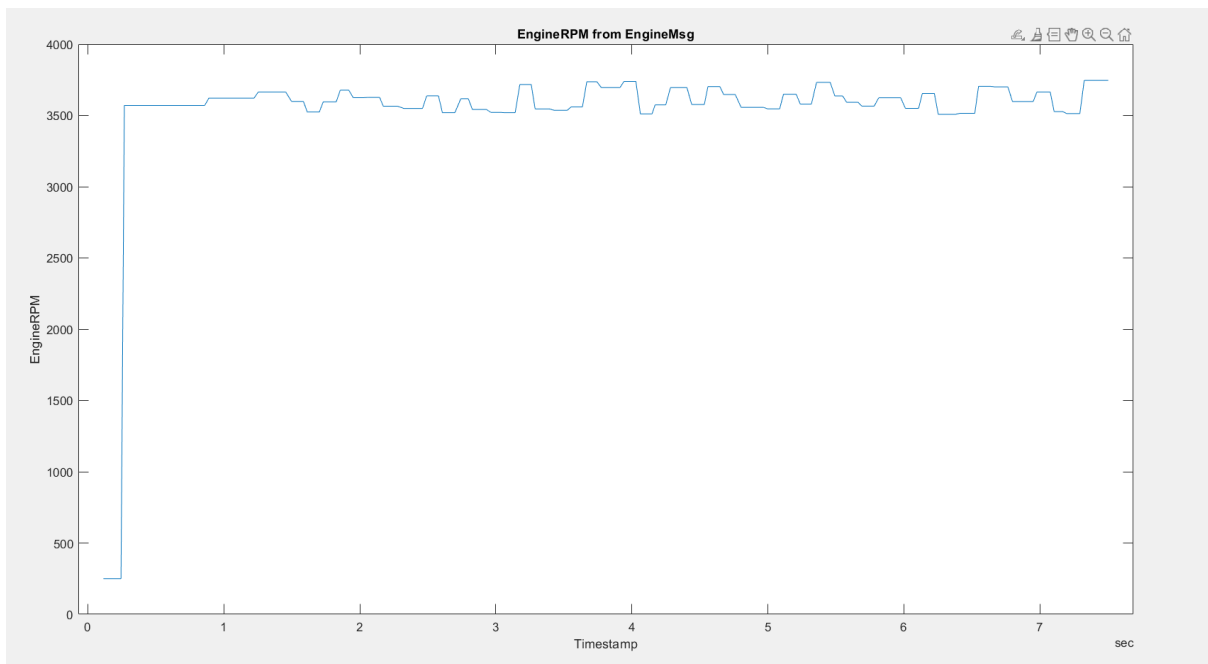


Figure 6.26: Plot EngineRPM

Now we can plot values of Vehicle Speed signal over time. The result is shown in Figure 6.27.

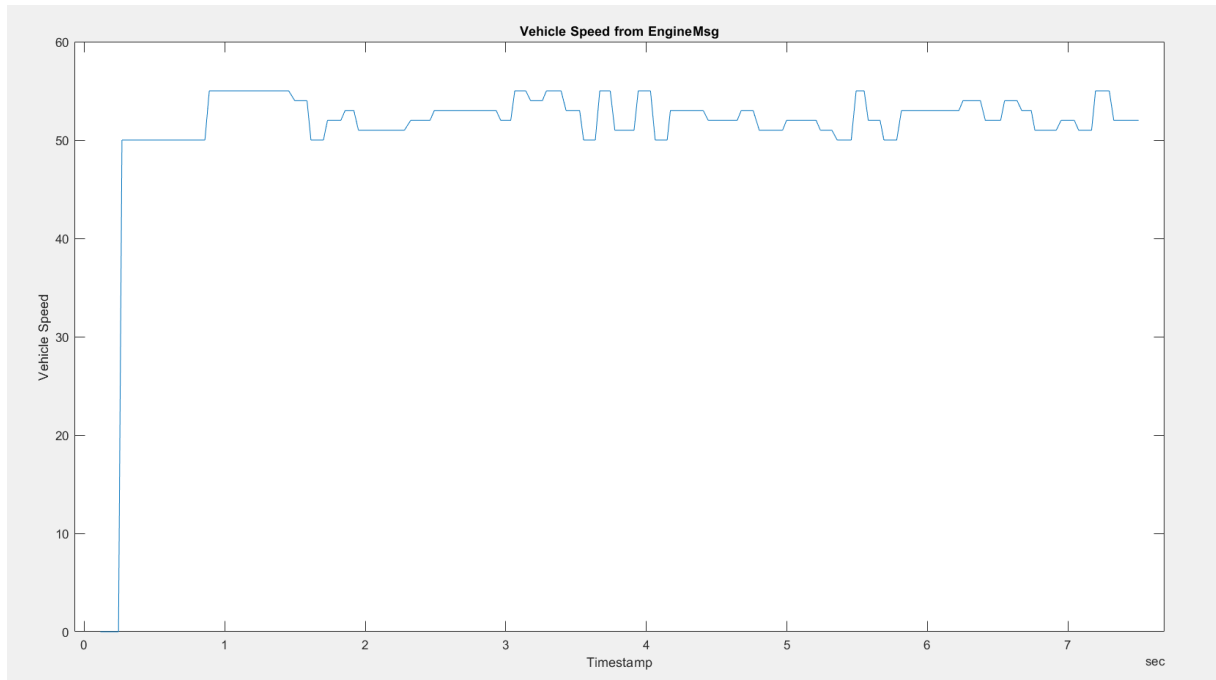


Figure 6.27: Plot Vehicle Speed

6.15 Close the DBC-File

Close access to the DBC-file by clearing its variable from the workspace.

```
51  
52      %close the DBC-file  
53      clear db  
54
```

Figure 6.28: Close DBC-File

6.16 Conclusion

In this example is possible to simulate a real communication through the Virtual Channel, using real CAN Message objects EngineMsg contained in the DBC File example. The messages are generated using a specific function given by Matlab, analyzed within CAN Explorer and shown.

Chapter 7

Creation DBC-file

In this chapter it will be shown how to create your own DBC-File and perform the same example seen in the previous chapter.

7.1 Kvaser Database Editor

Kvaser Database Editor allows to edit or create a DBC-File, in this case the goal is to create a DBC that contains an EngineMSg and a DewPointMeasureMsg.

DBC-Msg	Signal1	Signal2
EngineMsg	EngineRPM	VehicleSpeed
DewPointMsg	Temperature	DewPoint

After opening Kvaser Database it is possible create the new CAN Message Object within the Message/Signal View selecting AddMessage as shown in Figure 7.1.

After this step rename the new Message:

- EngineMsg:
 - ID = 100
 - Frame Format = Standard
 - DLC = 8
- DewPointMsg

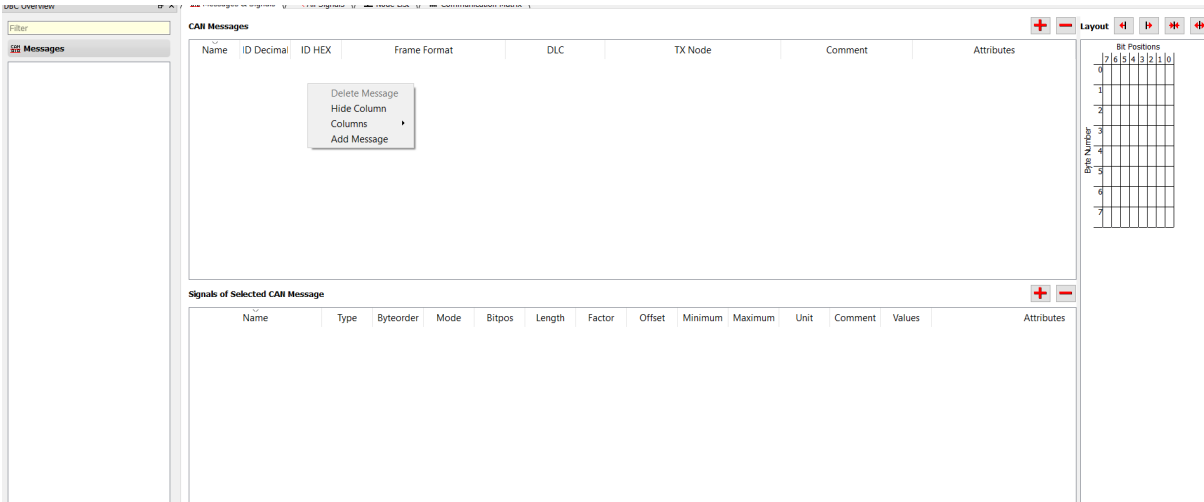


Figure 7.1: Create a New CAN Message Objects [12]

- ID = 200
- Frame Format = Standard
- DLC = 8

Figure 7.2 shown the CAN Messages created in the DBC-File and that is going to use in the future example.

CAN Messages							
	Name	ID Decimal	ID HEX	Frame Format	DLC	TX Node	Comment
1	EngineMsg	100	064	Standard	8	---	This is the Engine Message that contains EngineRPM and ve...
2	DewPointMsg	200	0c8	Standard	8	---	This is the DewPointMsg that contains Temperature and Dp...

Figure 7.2: New CAN Message[12]

Now select the EngineMsg and in **Signals of Selected CAN Message** *Add New Signal to Message*:

Signals of Selected CAN Message

	Name	Type	Byteorder	Mode	Bitpos	Length	Factor	Offset	Minimum	Maximum	Unit	Comment	Values
1	EngineRPM	Signed	Intel	Signal	0	32	0,1	200	250	9000	RPM		
2	VehicleSpeed	Signed	Intel	Signal	32	32	1	0	0	150	Km/H		

Figure 7.3: Signal of EngineMsg[12]

Now select the DewPointMsgMsg and in **Signals of Selected CAN Message** *Add New Signal to Message*:

Signals of Selected CAN Message

	Name	Type	Byteorder	Mode	Bitpos	Length	Factor	Offset	Minimum	Maximum	Unit	Comment	Values
1	Temperature	Signed	Intel	Signal	0	32	0,1	26	0	100	°C		
2	DewPoint	Signed	Intel	Signal	32	32	1	0	0	200	°C Td		

Figure 7.4: Signal of DewPointMsg[12]

7.2 Use DBC-File in CAN Communication

Now it possible receive and process messages using information stored in DBC-files created before using Matlab as the previous example.

7.3 Open the DBC-File

Use the DBC-File created and include it in our project folder.

First of all open the DBC-File created in the previous section: NAMEDBC-FILE.dbc using the function canDatabase.

```
1 db = canDatabase("ProvaDBCF.dbc")
2 db.Messages
```

Figure 7.5: Open DBC Code

Now examine the Messages properly to see the names of all messages defined in this DBC-file.

```

db =
Database with properties:
    Name: 'ProvaDBCF'
    Path: 'C:\Users\Marco\OneDrive\Desktop\iotprog\candbc\ProvaDBCF.dbc'
    UTF8_File: 'C:\Users\Marco\AppData\Local\Temp\tpf2dd2f4f_bc77_4e39_8f43_2070170b29e0'
    Nodes: {}
    NodeInfo: [0x0 struct]
    Messages: [2x1 cell]
    MessageInfo: [2x1 struct]
    Attributes: {}
    AttributeInfo: [0x0 struct]
    UserData: []

```

Figure 7.6: CAN Database

```

3 %message info
4 messageInfo(db, "EngineMsg")
5 messageInfo(db, "DewPointMsg")
6 messageInfo(db)
7

```

```

ans =

2x1 cell array

    {'DewPointMsg'}
    {'EngineMsg' }

```

Figure 7.8: CAN Database Message

7.4 View Message Information

Use messageInfo function to view information for message EngineMsg, including the identifier, data length and a signal list.

```

ans =

struct with fields:
    Name: 'EngineMsg'
    ProtocolMode: 'CAN'
    Comment: 'This is the EngineMsg that contains EngineRPM and VehicleSpeed'
    ID: 100
    Extended: 0
    J1939: []
    Length: 8
    DLC: 8
    BRS: 0
    Signals: [2x1 cell]
    SignalInfo: [2x1 struct]
    TxNodes: [0x1 cell]
    Attributes: {}
    AttributeInfo: [0x0 struct]

```

```

ans =

struct with fields:
    Name: 'DewPointMsg'
    ProtocolMode: 'CAN'
    Comment: 'This is the DewPointMsg that contains Temperature and DewPointMeasure'
    ID: 200
    Extended: 0
    J1939: []
    Length: 8
    DLC: 8
    BRS: 0
    Signals: [2x1 cell]
    SignalInfo: [2x1 struct]
    TxNodes: [0x1 cell]
    Attributes: {}
    AttributeInfo: [0x0 struct]

```

Figure 7.9: A) EngineMsg B) DewPointMsg

7.5 View Signal Information

Use `signalInfo` function to view information for signal `EngineRPM` in `CAN Message EngineMsg` and signal `DewPoint` in `DewPointMsg` Message, including type, byte ordering size, and scaling values that translate raw signals to physical values.

```
8 %view signal Information
9 signalInfo(db, "EngineMsg", "EngineRPM");
10 signalInfo(db, "DewPointMsg", "DewPoint");
11 signalInfo(db, "EngineMsg");
12 signalInfo(db, "DewPointMsg");
```

Figure 7.10: Signal Information

In figure 7.11 are shown the CAN Message structures for each CAN Message. The value as name, byte order and units coincide with those defined during the creation of the DBC file.

<pre>ans = struct with fields: Name: 'EngineRPM' Comment: '' StartBit: 32 SignalSize: 32 ByteOrder: 'LittleEndian' Signed: 1 ValueType: 'Single' Class: 'single' Factor: 8 Offset: 200 Minimum: 0 Maximum: 5000 Units: 'RPM' ValueTable: [0x1 struct] Multiplexor: 0 Multiplexed: 0 MultiplexMode: 0 RxNodes: {0x1 cell} Attributes: {} AttributeInfo: [0x0 struct]</pre>	<pre>ans = struct with fields: Name: 'DewPoint' Comment: '' StartBit: 0 SignalSize: 32 ByteOrder: 'LittleEndian' Signed: 1 ValueType: 'Single' Class: 'single' Factor: 1 Offset: 0 Minimum: 0 Maximum: 200 Units: '°C Td' ValueTable: [0x1 struct] Multiplexor: 0 Multiplexed: 0 MultiplexMode: 0 RxNodes: {0x1 cell} Attributes: {} AttributeInfo: [0x0 struct]</pre>
---	--

Figure 7.11: A) EngineMsg B) DewPointMsg

7.6 Create a Message Using Database Definitions

Now, create a new message by specifying the database and the message name Enginemsg to get the database definition applied. CAN signals in these messages are represented in engineering units in addition to the raw data bytes. Figure 7.12 shows the code.

```
13 %create Message Using Database Deinitions
14 msgEngineInfo = canMessage(db, "EngineMsg")
15 msgEngineInfo2 = canMessage(db, "DewPointMsg")
```

Figure 7.12: Code

Figure 7.13 shows the CAN message objects, in the specific is shown the ID, Protocol Mode, Name and Data.

<pre>msgEngineInfo = Message with properties: Message Identification ProtocolMode: 'CAN' ID: 100 Extended: 0 Name: 'EngineMsg' Data Details Timestamp: 0 Data: [0 0 0 0 0 0 0 0] Signals: [1x1 struct] Length: 8 Protocol Flags Error: 0 Remote: 0 Other Information Database: [1x1 can.Database] UserData: []</pre>	<pre>msgEngineInfo2 = Message with properties: Message Identification ProtocolMode: 'CAN' ID: 200 Extended: 0 Name: 'DewPointMsg' Data Details Timestamp: 0 Data: [0 0 0 0 0 0 0 0] Signals: [1x1 struct] Length: 8 Protocol Flags Error: 0 Remote: 0 Other Information Database: [1x1 can.Database] UserData: []</pre>
---	--

Figure 7.13: A) EngineMsg Info B) DewPointMsg Info

7.7 View New Signal Information

Use the Signals property to see signal values for the specific message. Directly write to and read from these signals to pack and unpack data from the message.[\[11\]](#)

In Figure 7.14 is shown the two values contained into EngineMsg and DewPointMsg CAN message: VehicleSpeed, EngineRPM, Temperature and DewPoint.

```
ans =  
  
  struct with fields:  
  
    EngineRPM: 200  
    VehicleSpeed: 0  
  
ans =  
  
  struct with fields:  
  
    Temperature: 200  
    DewPoint: 0
```

Figure 7.14: Signals value

7.8 Change Signal Information

In Figure 7.15 is shown the code that allows us to change Signal Information EngineRPM, vehicleSpeed, DewPoint, Temperature and view it.

```
19 %change Signal Information  
20 msgEngineInfo.Signals.EngineRPM = 5000;  
21 msgEngineInfo.Signals;  
22 msgEngineInfo.Signals.VehicleSpeed = 63;  
23 msgEngineInfo.Signals;
```

Figure 7.15: New Signal Code

Write directly to the signal EngineRPM to change its value, in this case 5000. In Figure 7.16 is shown the CAN message object with the new data, note the EngineRPM has been updated with the written value 5000.


```
msgEngineInfo =  
  
    Message with properties:  
  
        Message Identification  
        ProtocolMode: 'CAN'  
        ID: 100  
        Extended: 0  
        Name: 'EngineMsg'  
  
        Data Details  
        Timestamp: 0  
        Data: [0 0 0 0 0 0 22 68]  
        Signals: [1x1 struct]  
        Length: 8  
  
        Protocol Flags  
        Error: 0  
        Remote: 0  
  
        Other Information  
        Database: [1x1 can.Database]  
        UserData: []  
  
ans =  
  
    struct with fields:  
  
        EngineRPM: 5000  
        VehicleSpeed: 0
```

Figure 7.16: New Signal Information

It also important know when a value is written to the signal as in this case, it is translated, scaled and packed into the message data using the database definition. Now create a new value to the vehicleSpeed signal.

Figure 7.17 show the vehicleSpeed CAN Message with the new value.

```
msgEngineInfo =  
  
    Message with properties:  
  
        Message Identification  
        ProtocolMode: 'CAN'  
        ID: 100  
        Extended: 0  
        Name: 'EngineMsg'  
  
        Data Details  
        Timestamp: 0  
        Data: [0 0 124 66 0 0 22 68]  
        Signals: [1×1 struct]  
        Length: 8  
  
        Protocol Flags  
        Error: 0  
        Remote: 0  
  
        Other Information  
        Database: [1×1 can.Database]  
        UserData: []  
  
ans =  
  
    struct with fields:  
  
        EngineRPM: 5000  
        VehicleSpeed: 63
```

Figure 7.17: New Value

Do the same for the DewPointMsg, write directly to the signal Temperature to change its value, in this case 18. In Figure 7.18 is shown the CAN message object with the new data.

```
msgEngineInfo2 =  
  
    Message with properties:  
  
    Message Identification  
        ProtocolMode: 'CAN'  
            ID: 200  
        Extended: 0  
        Name: 'DewPointMsg'  
  
    Data Details  
        Timestamp: 0  
        Data: [0 0 0 0 0 0 54 195]  
        Signals: [1x1 struct]  
        Length: 8  
  
    Protocol Flags  
        Error: 0  
        Remote: 0  
  
    Other Information  
        Database: [1x1 can.Database]  
        UserData: []  
  
ans =  
  
    struct with fields:  
  
        Temperature: 18  
        DewPoint: 0
```

Figure 7.18: New Signal Information

Do the same for the DewPoint signal, write a new value 63.

```

msgEngineInfo2 =

  Message with properties:

    Message Identification
      ProtocolMode: 'CAN'
      ID: 200
      Extended: 0
      Name: 'DewPointMsg'

    Data Details
      Timestamp: 0
      Data: [0 0 124 66 0 0 54 195]
      Signals: [1x1 struct]
      Length: 8

    Protocol Flags
      Error: 0
      Remote: 0

    Other Information
      Database: [1x1 can.Database]
      UserData: []

ans =

  struct with fields:

    Temperature: 18
    DewPoint: 63

```

Figure 7.19: New Value

7.9 Start and receive Messages with Database Information

Now after some analysis about the CAN messages object contained into DBC-file we can attach a database to a CAN channel that receives messages to apply database definitions to incoming messages automatically. It is important to know the database decode only messages that are defined. All other messages are received in their raw form.

It is possible start the channel, figure 7.20, generate some messages traffic and receive messages with physical message decoding

The function `generateMsgsDB2`, it is a function provided by matlab for creates and transmits CAN messages for demo purposes, must be added to the folder, this function periodically transmits multiple CAN messages at various periodic rates with changing message data.

```
rxCh =
  Channel with properties:
    Device Information
      DeviceVendor: 'MathWorks'
      Device: 'Virtual 1'
      DeviceChannelIndex: 2
      DeviceSerialNumber: 0
      ProtocolMode: 'CAN'
    Status Information
      Running: 0
      MessagesAvailable: 0
      MessagesReceived: 0
      MessagesTransmitted: 0
      InitializationAccess: 1
      InitialTimestamp: [0x0 datetime]
      FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
    Channel Information
      BusStatus: 'N/A'
      SilentMode: 0
      TransceiverName: 'N/A'
      TransceiverState: 'N/A'
      ReceiveErrorCount: 0
      TransmitErrorCount: 0
      BusSpeed: 500000
```

Figure 7.20: Channel

7.10 CAN Explorer

In Figure 6.23 are shown some of the data sent through the Virtual Channel and monitored by the CAN Explorer. In each row there are time, ID, Length and the data.

Is possible derive the value of EngineRPM and VehicleSpeed contained in Figure6.23, first of all we need to convert data in decimal and using the following formula:

$$\text{EngineRPM} = \text{offset}(200) + \text{DataDecimal}(X) * \text{factor}(0.1)$$

$$\text{DewPoint} = \text{offset}(26) + \text{DataDecimal}(X) * \text{factor}(0.1)$$

Examine different example generated during the simulation:

Message Table				
Time	ID	Message	Length	Data
7.869358	C8		8	03 01 00 00 34 00 00 00
7.854428	64		8	99 83 00 00 33 00 00 00
7.824044	C8		8	03 01 00 00 34 00 00 00
7.824041	64		8	99 83 00 00 33 00 00 00
7.790166	64		8	99 83 00 00 33 00 00 00
7.774130	C8		8	03 01 00 00 34 00 00 00
7.758877	64		8	2F 83 00 00 33 00 00 00
7.727834	C8		8	F5 00 00 00 33 00 00 00
7.727831	64		8	2F 83 00 00 33 00 00 00
7.697143	64		8	2F 83 00 00 33 00 00 00
7.681534	C8		8	F5 00 00 00 33 00 00 00
7.666062	64		8	2F 83 00 00 33 00 00 00
7.634274	C8		8	F5 00 00 00 32 00 00 00
7.628122	64		8	2F 83 00 00 33 00 00 00
7.597325	C8		8	FE 00 00 00 32 00 00 00
7.597320	64		8	E5 88 00 00 35 00 00 00
7.566005	64		8	E5 88 00 00 35 00 00 00

Figure 7.21: CAN Explorer

Engine(hex)	Engine(dec)	Speed(hex)	Speed(dec)
16 81	3504.6	36	54
4C 88	3739.2	36	54
E9 82	3551.3	35	53
2F 83	3608.3	33	51

Figure 7.22: Example EngineMsg

As said before the most significant 8 bit are the EngineRPM and the less significant 8 bit are the VehicleSpeed. Now we need to use the formula that has been mentioned, furthermore the offset and factor have been shown in Figure 7.3, the DataDecimal is given starting from the hexadecimal Data and according to the byte order, since in this case we are dealing with Little-Endian, we need to switch 16 and 81.

This means we convert 81 16 in decimal 33046 and we use the formula:

$$\text{EngineRPM} = \text{offset}(200) + \text{DataDecimal}(33046) * \text{factor}(0.1) = 3504.6$$

$$\text{VehicleSpeed} = \text{offset}(0) + \text{DataDecimal}(54) * \text{factor}(1) = 54 \text{ km/h}$$

$$\text{EngineRPM} = \text{offset}(200) + \text{DataDecimal}(34892) * \text{factor}(0.1) = 3739.2$$

$$\text{VehicleSpeed} = \text{offset}(0) + \text{DataDecimal}(54) * \text{factor}(1) = 54 \text{ km/h}$$

$$\text{EngineRPM} = \text{offset}(200) + \text{DataDecimal}(33513) * \text{factor}(0.1) = 3551.3$$

$$\text{VehicleSpeed} = \text{offset}(0) + \text{DataDecimal}(53) * \text{factor}(1) = 53 \text{ km/h}$$

$$\text{EngineRPM} = \text{offset}(200) + \text{DataDecimal}(33583) * \text{factor}(0.1) = 3608.3$$

$$\text{VehicleSpeed} = \text{offset}(0) + \text{DataDecimal}(51) * \text{factor}(1) = 51 \text{ km/h}$$

Now we are going to do the same operation for the DewPointMsg which is contained in the DBC File.

Temperature(hex)	Temperature(dec)	DewPoint(hex)	DewPoint(dec)
FD	51.3	34	52
FC	51.2	35	53
F3	50.3	33	51
FE	51.4	32	50

Figure 7.23: Example DewPointMsg

As said before the most significant 8 bit are the Temperature and the less significant 8 bit are the DewPoint. Now we need to use the formula that has been mentioned, furthermore the offset and factor have been shown in Figure 7.4, the DataDecimal is given starting from the hexadecimal Data and according to the byte order, since in this case we are dealing with Little-Endian.

This is mean we convert FD in decimal 253 and we use the formula:

$$\text{Temperature} = \text{offset}(26) + \text{DataDecimal}(253) * \text{factor}(0.1) = 51.3$$

$$\text{DewPointMeasure} = \text{offset}(0) + \text{DataDecimal}(52) * \text{factor}(1) = 52$$

$$\text{Temperature} = \text{offset}(26) + \text{DataDecimal}(252) * \text{factor}(0.1) = 51.2$$

$$\text{DewPointMeasure} = \text{offset}(0) + \text{DataDecimal}(53) * \text{factor}(1) = 35$$

$$\text{Temperature} = \text{offset}(26) + \text{DataDecimal}(243) * \text{factor}(0.1) = 50.3$$

$$\text{DewPointMeasure} = \text{offset}(0) + \text{DataDecimal}(51) * \text{factor}(1) = 33$$

$$\text{Temperature} = \text{offset}(26) + \text{DataDecimal}(254) * \text{factor}(0.1) = 51.4$$

$$\text{DewPointMeasure} = \text{offset}(0) + \text{DataDecimal}(50) * \text{factor}(1) = 50$$

7.11 Conclusion

In this example it is possible to simulate a real communication through the Virtual Channel, using a real Database CAN created using Kvaser Database Editor, it contains CAN

Message objects EngineMsg and DewPointMsg. The first it's composed by EngineRpm signal and VEHICLESped signal the second is composed by Temperature signal and Dew-Point singal.

The messages are generated using a specific function given by Matlab, analyzed within CAN Explorer and shown a real communication among different devices.

Bibliography

- [1] <https://www.ti.com/lit/an/sloa101b/sloa101b.pdf>
↑ 5, 8, 9, 10
- [2] <https://www.microst.it/tutorial/can.htm>
↑
- [3] <https://www.iso.org/standard/63648.html>
↑
- [4] <https://it.mathworks.com/help/thingspeak/>
↑ 25
- [5] <https://it.mathworks.com/help/thingspeak/collect-data-in-a-new-channel.html>
↑ 26
- [6] <https://it.mathworks.com/help/thingspeak/analyze-your-data.html>
↑ 26, 27, 28
- [7] <https://it.mathworks.com/help/vnt/getting-started-with-vehicle-network-toolbox.html>
↑ 31
- [8] <https://it.mathworks.com/help/vnt/ug/can-communication-session.html>
↑ 35, 36, 39, 40
- [9] <https://it.mathworks.com/help/vnt/ug/mathworks-virtual-channels.html>
↑ 31
- [10] <https://it.mathworks.com/help/vnt/ug/use-can-explorer-data-reception-visualization.html>
↑ 41, 42
- [11] cita <https://it.mathworks.com/help/vnt/ug/candatabase.html>
↑ 66, 67, 68, 69, 87
- [12] [kvaser-databse-editor-userguide.pdf](#)
↑ 82, 83