canbus

Release 1.0

Jan Jansen

Oct 14, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

I have an old volvo, and although this is probably a bad indicator for a car enthusiast, I do like car technology.

Cars have evolved, and are now full of electronics. Microcontrollers interconnected via a CAN netwerk.

Very soon my old diesel car will not be legal anymore (although EURO 5, having a particle filter, EGR valve ....)

The value of my car was estimated 500 euro, and this was two years ago ...

Still, for me it is much more valuable! Certainly since I started looking under the hood.

I bought one of those cheap chinese diagnostic tools. At first I did not pay attention, since all I wanted is save a buck on maintenance. This is one of the reasons old cars get expensive and people will get a new one.

The diagnostic tool is called VIDA, it came with a virtual box. As it happens you can find this easily: e.g. https://volvodiag.com/product/virtualbox-build/ (google is your friend)

This is commercial software, so any free copy you will find is of course illegal. From what I found out somebody (Gunnar?) in the US made a copy.

Suppose you're a volvo dealer, you would like to use the same software for all your volvo models, even the older ones.

The software is using an sql-database and XML files, so everything is a parameter. I'm not an expert, but it is very well made, and offers a lot of insight on how professional developers create a highly configurable package. Watch and learn!

A word of warning : the software is not open sourced!

Instead of buying a new electric volvo my attention got caught by Damien Maguire MuskVo. He is involved in the openinverter project : https://openinverter.org/wiki/Main_Page.

As it turns out these car enthusiasts have succesfully converted modern (ecu containing) cars into electric vehicles with modern electrics and batteries.

A big hurdle is exactly the presence of all the electronics. If replacing the ICE (internal combustion engine) by electric, you will somehow have to fool the brain (ECM) that the ICE is still there.

All these modules talk to each other via the CAN bus network. An important step in an EV conversion is getting familiar with this network.

I created a cheap STM32 CAN interface, which consists of an advanced sniffer/sender and an emulator (which will eventually emulate the Volvo ECM)

As you might have guessed Car manufacturers are not really into open source. Even figuring out what is going on the CANBUS can be challenging. This is the reason why I analysed the diagnostic software, to gain further insight into CAN messaging.

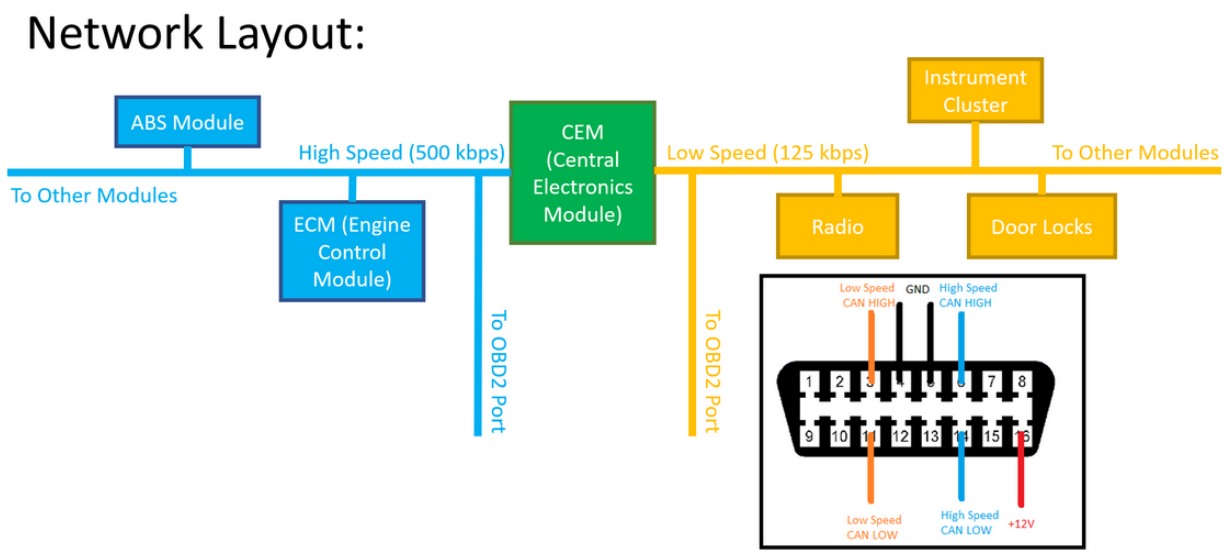
This manual is an account of my journey in the car network, car communication and car diagnostics, which might pave the way for an EV conversion....

CHAPTER

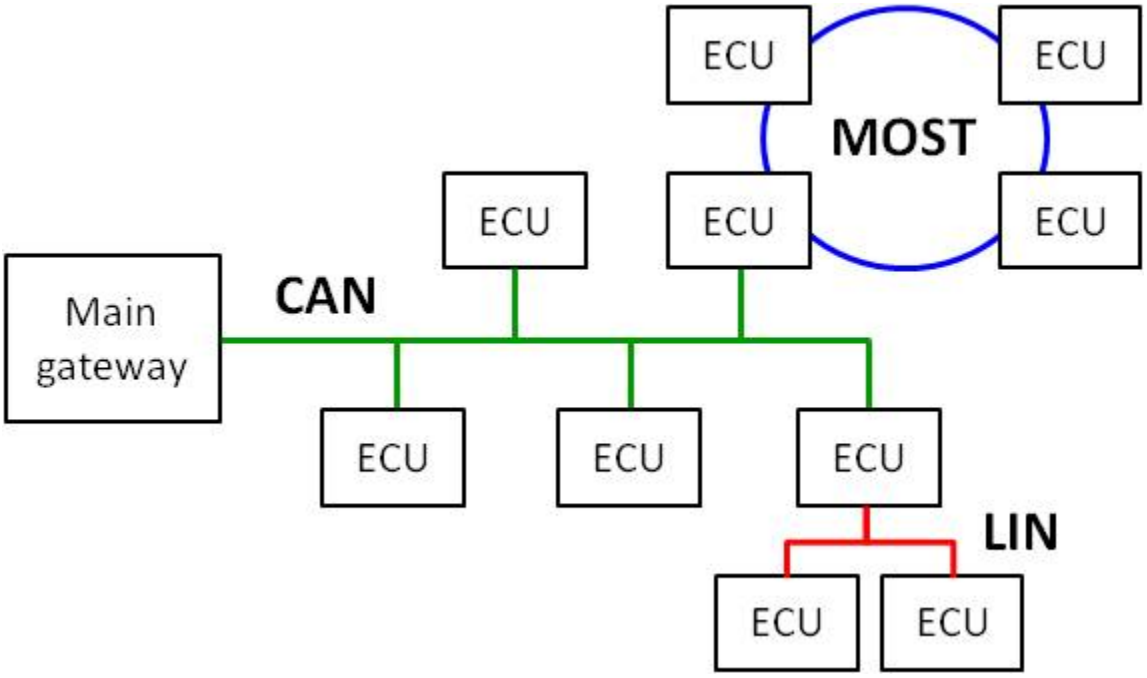
TWO

VOLVO CANBUS AND MODULES

2.1 Volvo Canbus OBD2



2.2 Networks



Controller Area Network (CAN) emerged as a communication network by Bosch GmbH specifically tailored for the automotive industry in 1985. It operates on a message-based broadcast protocol.

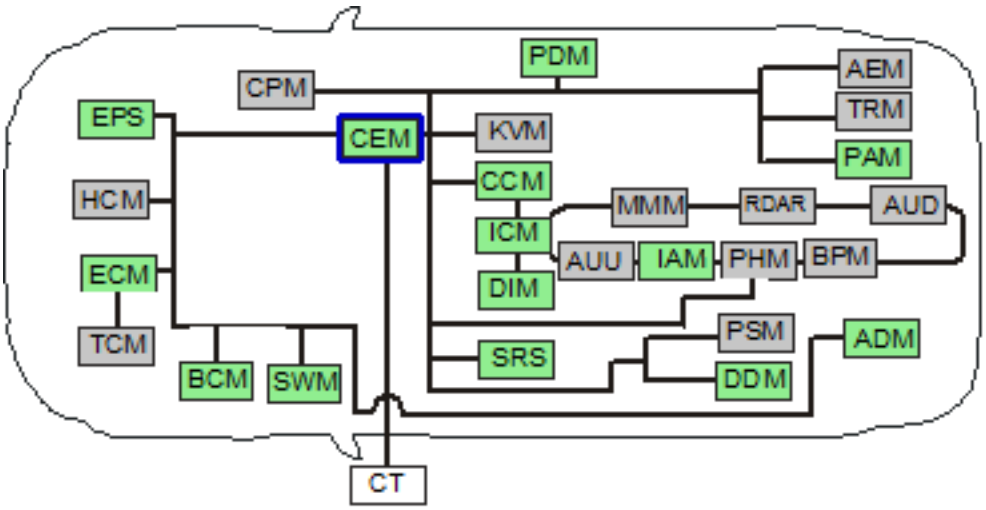
Given its maximum bandwidth of 1 Mbit/s, CAN proves inadequate for transferring video and audio data. Hence, Media Oriented Systems Transport (MOST) serves this purpose . MOST, a fiber-optic ring network, is meticulously engineered for the efficient transmission of large data volumes within automotive systems. Nodes within a MOST network aren't directly accessible; they are only reachable via specialized CAN ECUs functioning as gateways.

To alleviate CAN's load, Local Interconnect Network (LIN) establishes a sub-network on CAN for communicating with low-performance ECUs. Access to LIN is orchestrated through the CAN master node, which oversees slave nodes within this network.

2.3 Volvo V50 modules overview

This is an overview of the modules in a volvo V50.

in the vida software you can find an overview that looks like this :



in the log of the diagnostic session I found a link between:

- the vida xml module ID
- the can message ID ? (to be confirmed)
- the module description

871101	41	CCM
831701	67	DDM
393204	117	IAM
641901	73	SWM
645901	48	EPS
381101	81	DIM
593102	1	BCM
366902	99	PAM
831703	69	PDM
884701	88	SRS
254501	11	ADM
284101	17	ECM
393901	84	ICM
372302	80	CEM
372304	64	CEM

2.4 Modules:

- CEM Central Electronic Module (Behind right side of dash)
- EPS Electrical Power Steering module (Right front corner of engine compt)
- BCM Brake Control Module (At left rear corner of engine compt)
- HCM Headlamp Control Module (Behind left headlight) (Active lights only, (S40/V50))
- SWM Steering Wheel Module (Top of steering column)
- ECM Engine Control Module (Left front of engine compt)

- TCM Transmission Control Module (Rear of engine compt) (Automatic transmission only)
- ADM Additive Dosing Module (Diesel only)

2.5 overview modules

Number	ID	Type
254501	11	ADM
366902	99	PAM
284101	17	ECM
366901	45	KVM
372301	82	AEM
372302	80	CEM
372304	64	CEM
381101	81	DIM
393202	109	AUD
393901	84	ICM
393906	102	MMM
394201	100	PHM
395301	104	SUB
593102	1	BCM
641901	73	SWM
645901	48	EPS
831701	67	DDM
831703	69	PDM
852601	46	PSM
871101	41	CCM
875401	24	CPM
884701	88	SRS
892901	35	TRM
393204	117	IAM
393205	113	RDAR
372306	123	AUU
352902	112	HCM
394202	124	BPM

2.6 Modules for profile

In the file : /system/log/diagnostics/VIDA, is a lot of usefull info.

For instance : vadis_GetallEcuDataForProfile 0b00c8af83f1a8fb.

2.6.1 executing stored procedure

SELECT DISTINCT

t121.commaddress as ecuaddress, t123.identifier as busid, t102.identifier as ecutype, t102.description ecuDescription, t100.identifier diagnosticnumber, t121.id as configId, t121.canidrx as canid, t121.fkT121_Config_Gateway as gatewayConfigId, t122.id as protocolid, t161.folderLevel as folderLevel, t121.priority as [priority]

FROM T120_Config_EcuVariant t120 INNER JOIN

T121_Config T121 ON T120.fkT121_Config = T121.id

INNER JOIN T122_Protocol T122 ON T121.fkT122_protocol = T122.id

INNER JOIN T100_EcuVariant T100 ON T120.fkT100_EcuVariant = T100.id

INNER JOIN T160_DefaultEcuVariant T160 ON T100.id = T160.fkT100_EcuVariant

INNER JOIN T161_Profile T161 on T160.fkT161_Profile = T161.Id

INNER JOIN T123_Bus t123 on T121.fkt123_bus = t123.id

INNER JOIN T101_Ecu t101 on T100.fkt101_ecu = t101.id

INNER JOIN T102_EcuType t102 on t101.fkt102_ecutype = t102.id

INNER JOIN dbo.GetCompatibleProfiles('0b00c8af83f1a8fb') p on t161.id = p.id

WHERE t102.identifier <> 0

ORDER BY t161.folderLevel DESC, t121.priority ASC

2.6.2 table with overview

"ecuaddress","busid","ecutype","ecuDescription", "diagnosticnumber","configId","gatewayConfigId"

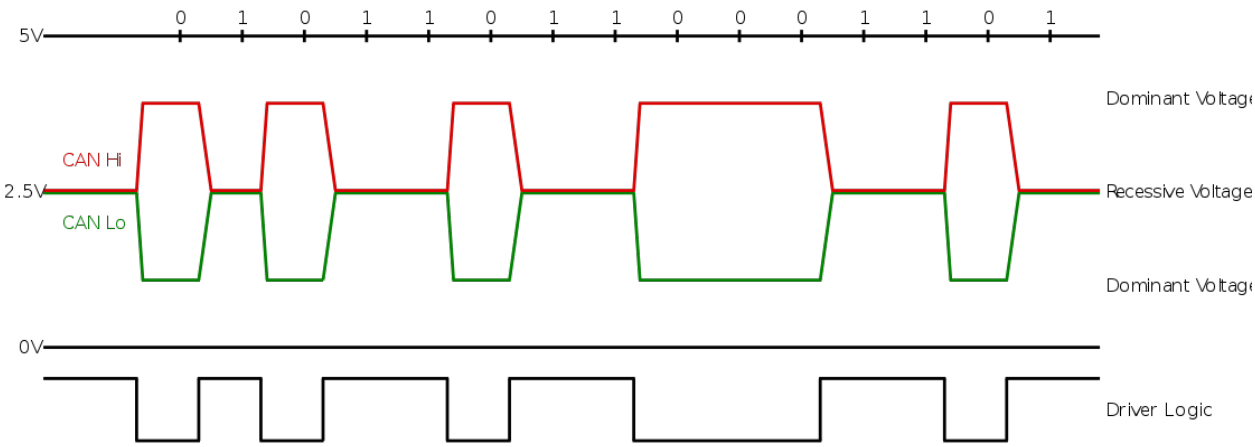
Table 1: EcuAddress for Model

Address	busid	ecutype	descript
0B	1	254501	ADM
11	1	284101	ECM
7B	0	372306	AUU
7C	0	394202	BPM
2D	0	366901	KVM
63	0	366902	PAM
52	0	372301	AEM
40	0	372304	CEM
51	0	381101	DIM
6D	0	393202	AUD
75	0	393204	IAM
71	0	393205	RDAR
54	0	393901	ICM
66	0	393906	MMM
64	0	394201	PHM
68	0	395301	SUB
43	0	831701	DDM
45	0	831703	PDM
2E	0	852601	PSM
29	0	871101	CCM
18	0	875401	CPM
58	0	884701	SRS
23	0	892901	TRM
70	1	352902	HCM
50	1	372302	CEM
01	1	593102	BCM
49	1	641901	SWM
30	1	645901	EPS

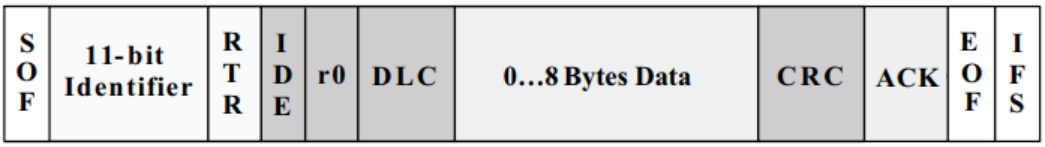
2.7 further info on canbus:

2.8 Volvo uses extended identifiers:

CAN frames follow a defined format: all standard frames have an 11-bit identifier and up to 8 bytes of data. Extended frames allow 29 bit identifiers, but only the same 8 bytes of data. CAN frames also include checksums, and most CAN implementations in microcontrollers will automatically insert / verify checksums in hardware.



Standard CAN



Extended CAN

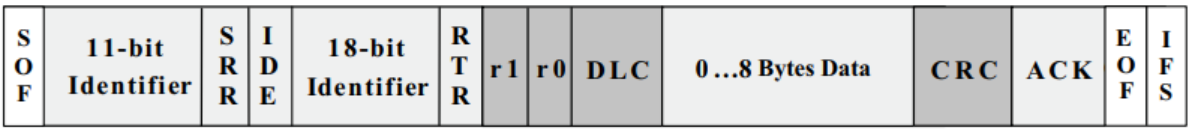
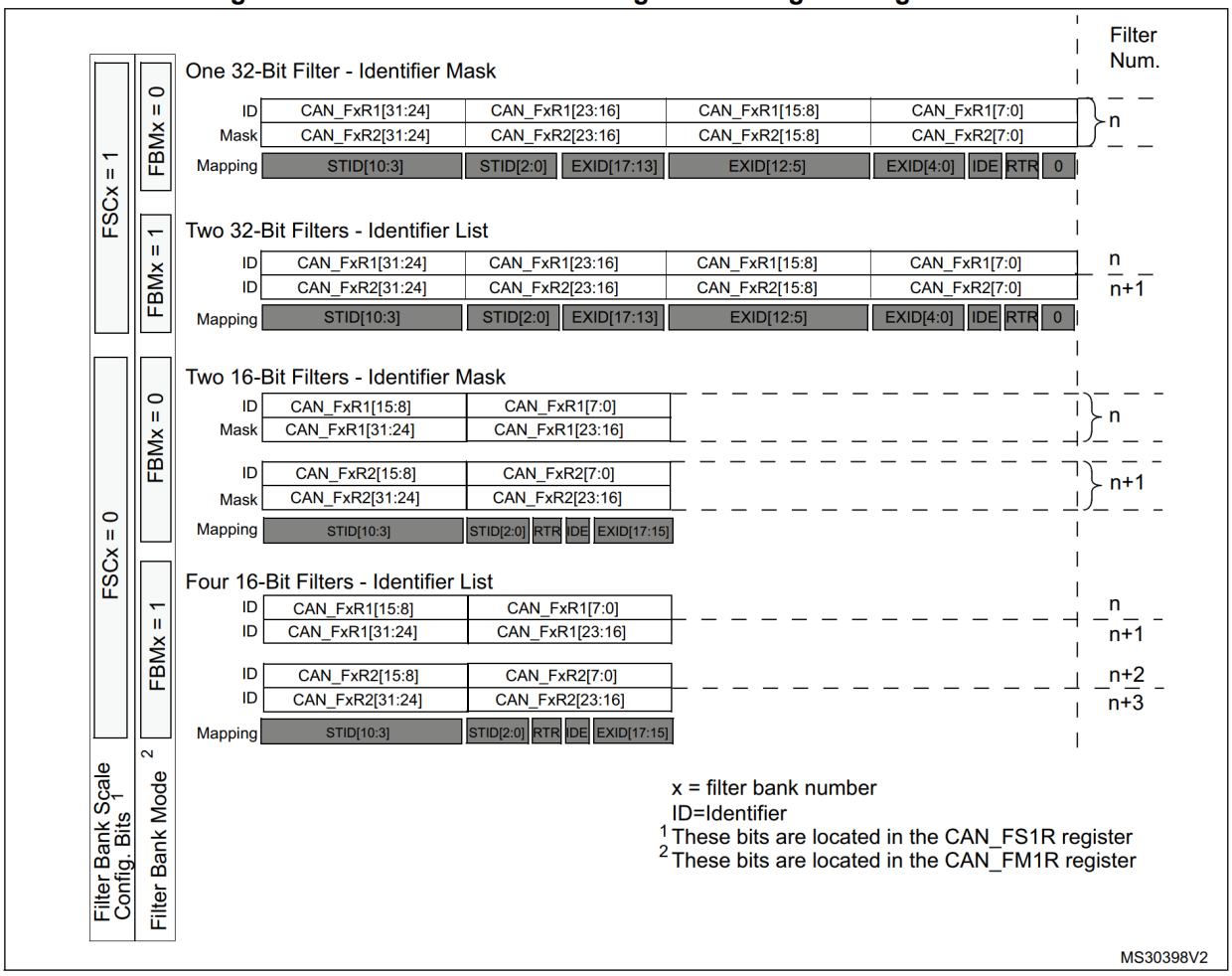


Figure 315. Filter bank scale configuration - register organization



CHAPTER

THREE

VOLVO DIAGNOSTIC SOFTWARE

3.1 Volvo VIDA

Volvo VIDA (Vehicle Information and Diagnostics for Aftersales) is a comprehensive diagnostic software system developed by Volvo Cars to aid in the servicing and maintenance of their vehicles. It provides technicians with the necessary tools to diagnose, troubleshoot, and repair Volvo cars effectively.

3.1.1 Key Features

1. **Vehicle Information**: VIDA provides detailed information about Volvo vehicles, including technical specifications, wiring diagrams, component locations, and service procedures. This allows technicians to quickly access relevant information needed for diagnosis and repair.
2. **Diagnostic Functions**: VIDA allows technicians to perform comprehensive diagnostics on various vehicle systems, including the engine, transmission, ABS, airbag, climate control, and more. It can read and clear diagnostic trouble codes (DTCs), view live data streams, perform component tests, and program electronic control modules (ECUs).
3. **Software Updates**: VIDA enables technicians to update vehicle software to the latest available versions. This is crucial for ensuring optimal performance, reliability, and compatibility with new technologies.
4. **Guided Diagnostics**: VIDA offers guided diagnostic procedures to assist technicians in identifying and resolving vehicle issues efficiently. These step-by-step instructions help streamline the diagnostic process and minimize errors.
5. **Remote Diagnostics**: VIDA supports remote diagnostics capabilities, allowing technicians to access vehicle data and perform diagnostics remotely. This feature can be particularly useful for troubleshooting complex issues or providing support to technicians in remote locations.
6. **Integrated Parts Catalog**: VIDA includes an integrated parts catalog that provides access to genuine Volvo parts information, including part numbers, descriptions, and illustrations. This simplifies the process of ordering and replacing components during repairs.
7. **Multi-Language Support**: VIDA is available in multiple languages, making it accessible to technicians worldwide.

Overall, Volvo VIDA is a powerful diagnostic tool that helps Volvo technicians efficiently diagnose and repair vehicles, ensuring optimal performance, safety, and reliability for Volvo owners.

3.1.2 databases

The best way to get some relevant data out of the VIDA, is to start a diagnostic sessions. It then creates logfiles under : VIDA/system/log/diagnostics.

```
Trying to open a connection database server (Master DB)
 connection is open
Found database AccessServer
Found database BaseData
Found database CarCom
Found database DiagSwdlRepository
Found database DiagSwdlSession
Found database DiceTiming
Found database imagerepository
Found database EPC
```

3.1.3 SQL database VIDA

the VIDA installation has a SQL server database (microsoft) onboard. There is a way to peek into the tables : - in the VidaConfigApplication.exe - DBUser : sa (in my case) - DBpwd : GunnarS3g3

SQL server import/export wizard - use sql Server Authentication - select the database (eg DiagSwdlSession)

3.1.4 howto peek into the VIDA database:

Volvo VIDA software is designed for multiple Volvo models and multiple model years. To select your model and the appropriate configuration, it reads out the VIN (vehicle identification number). This is made possible by the software's large database of XML files. There is a method to read out these XML files and select the ones that are important for your model.

https://github.com/spnda/volvo_vida_db https://github.com/Tigo2000/Volvo-VIDA/

3.1.5 stored procedures:

in the microsoft sql database, you can find stored procedures, that can give some indication on where VIDA gets its information

```
/****** Object:  Stored Procedure dbo.vadis_GetDiagInit    Script Date: 2006-10-18␣
↪14:59:31 ******/

ALTER  PROCEDURE [dbo].[vadis_GetDiagInit]
@configId int
AS
SELECT     T132_InitValueType.name,
    T131_InitValue.initValue
FROM
  T130_Init
INNER JOIN
         T131_InitValue ON T130_Init.id = T131_InitValue.fkT130_Init
INNER JOIN
         T132_InitValueType ON T131_InitValue.fkT132_InitValueType = T132_InitValueType.
↪id
```

(continues on next page)

(continued from previous page)

```
INNER JOIN
  T121_Config ON T130_Init.id = T121_Config.fkT130_Init_Diag
WHERE
 (T121_Config.id = @configId)
```

3.1.6 example:

this could be a record of interest : CEM,Central electronic module (CEM),Read high beam relay,NA,372302,80,0x504B03041400000008000B1D47

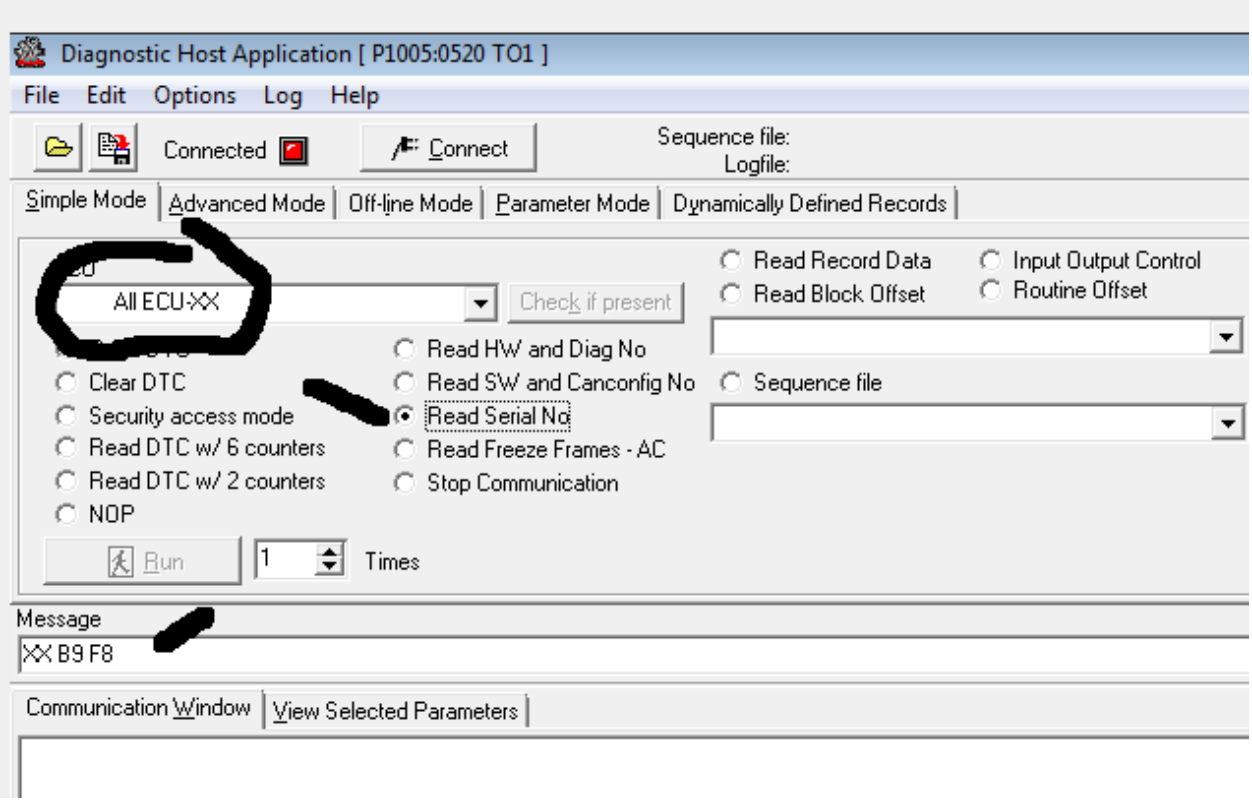
- the number 372302 refers to the CEM
- 0x504.. is a coded file

3.2 GGD-DHA (Generic Global Diagnostic - Diagnostic Host Application) .

“GGD DHA is a system used when developing systems in cars. This program contains all variables that are transferred within the car. It can access all variables and commands allowing reading and/or writing to these variables. It is also possible to read DTCs, part numbers, upload new software to the different modules etc. This system is useful when testing new systems since in VIDA you do not have access to all variables, which you have in GGD DHA, but in GGD DHA you do not get access to the technical description of how you replace parts, troubleshooting etc. However, in GGD DHA it is possible to create automatic sequences that read or write to the variables making it unnecessary to be at the computer all the time. When writing to a variable, it is possible to set the sensor values to the required values making it possible to simulate different conditions in a car even if the conditions have not occurred.”

https://drive.google.com/file/d/1Guan662vouIs1bkY1W7K2SAEcnkzm8A0/view

This software uses databases in (ddb or dds) format. After opening, you can send individual messages.



In the Volvo Diagnostic Host Application, you can save the data in text format.

Peeking in this database (P1005_(D2)_0520_TO1_050202.DDB) give some insight in how the Volvo Can messaging works.

This example is for reading the VIN number

```
DIAG_ITEM [NAME=VIN][SEND_ABLE=TRUE][NOTE=] [FORMAT=OTHER:2] [BASE=HEX] [VALUE=E9]␣
↪[TYPE=BLOFF][INPUT=FALSE]
BEGIN
  RESPONSE_ITEM [NAME=VIN .][SEND_ABLE=FALSE][NOTE=VIN] [NO_OF_BYTES=17] [OFFSET=2]␣
↪[MASK=] [UNIT=.] [PRECISION=0] [SIGNED=U] [BASE=ASCII] [FORMULA=*1] [COMP_VALUE=] [DEP_
↪RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE]
END
```

3.3 GGD-DHA (Hacking - or modifying the database).

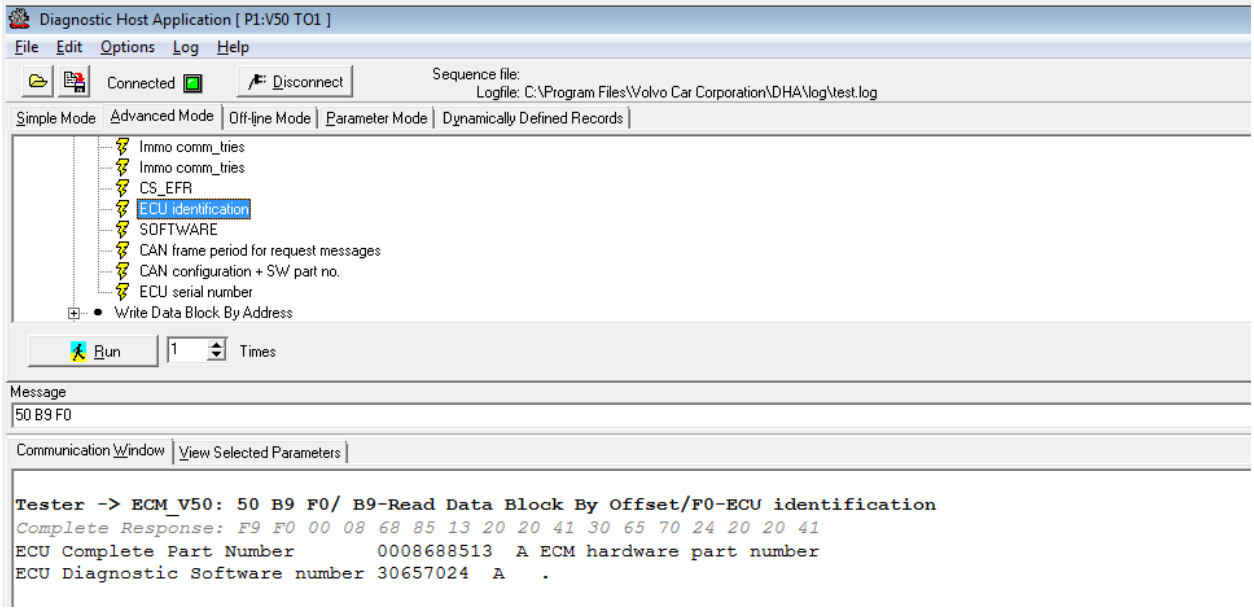
This software uses databases in (ddb, dds) format. The cool thing is that this software works with the DICE cable. ddb database format is text format and can be edited.

I created a small database for working with my V50. The CANBUS messages for the V50 can be found in the VIDA database/XML files.

I created a sample to interact with a stm32, so it sends a serial number upon request, pretending it is an ECM module (0x50). (can be found under the stm32f103c6-canbus directory)

3.3.1 conclusion :

this is an easy way to interact with all the modules in your car at no extra cost, provided you already have the DICE cable, and without programming, modifying the database in a texteditor will do.



CHAPTER

FOUR

ANALYSIS OF THE MSQL VIDA DATABASE

4.1 Vida database analysis

The main focus is the CARCOM database. (since guess, here is the CAN messages get assembled or translated)

something key is the ecu-identification

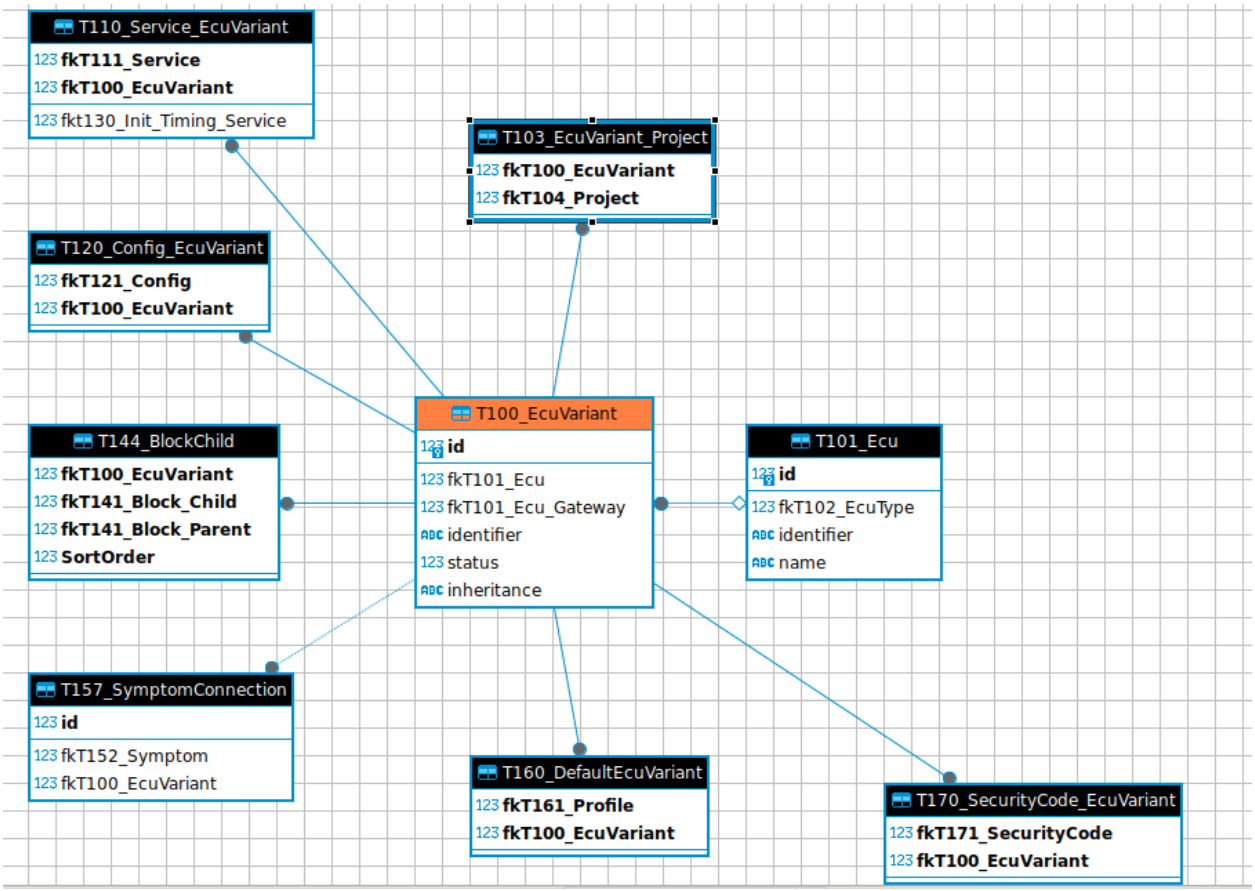
As a starting point the diagnostic session was used. Looking up 31254684 AC diagnosticPartNO in T100, gave EcuvariantID = 2013.

- it refers to T101 and from there to T102

so here I find identifier : 372302, CEM (this also is present in XML files) Suppose this is the CEM under the glove box.

- T120 links to T121

Here if find “29” which refers to extended CANBUS ID



4.2 Vida database analysis (block child)

The main focus is the CARCOM database.

T144 blockchild - query with ID=2013 (ecuvariant)

quite a few fk141 blockchild ID (extract)

guess these are all parameters from the CEM

- 4419201 Total distance (T141 block)
- 4419685 Global Data
- 4419034 Ambient Temperature

4.2.1 example: T141 Ambient temperature

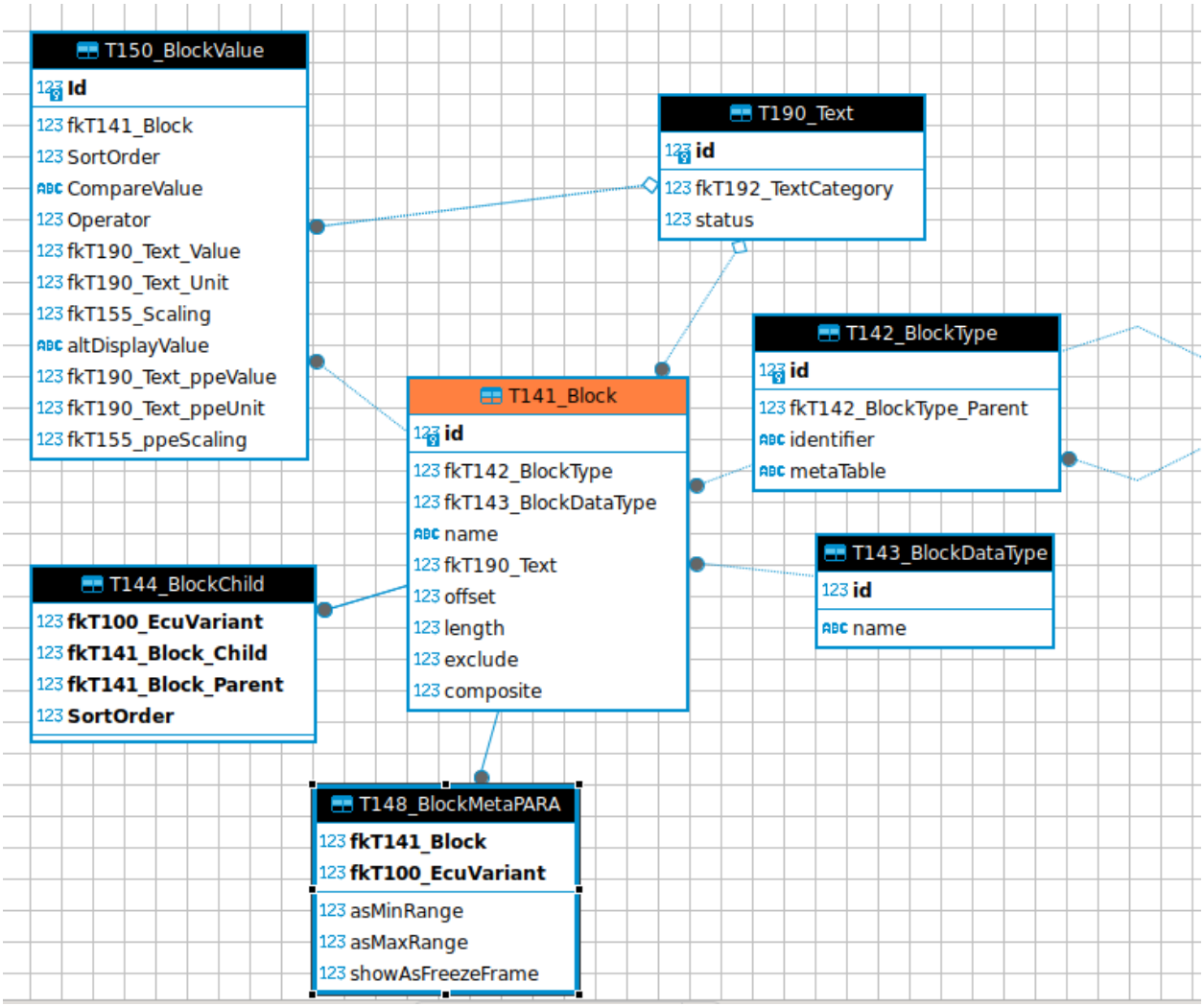
ID= 44119304 4419034 ambient temperature

- blocktype 5
- fk143_datatype 50
- length 16

-> blocktype 5 = REID -> datatype 50 = HEX

4.2.2 example: T141 Accelerator pedal

4419314 Accelerator pedal blocktype 5, fk143_datatype 50, length 16



4.3 Carcommunication and scripts

in the CARCOM database (T102), the modules are defined by a number: example the CEM = 372302

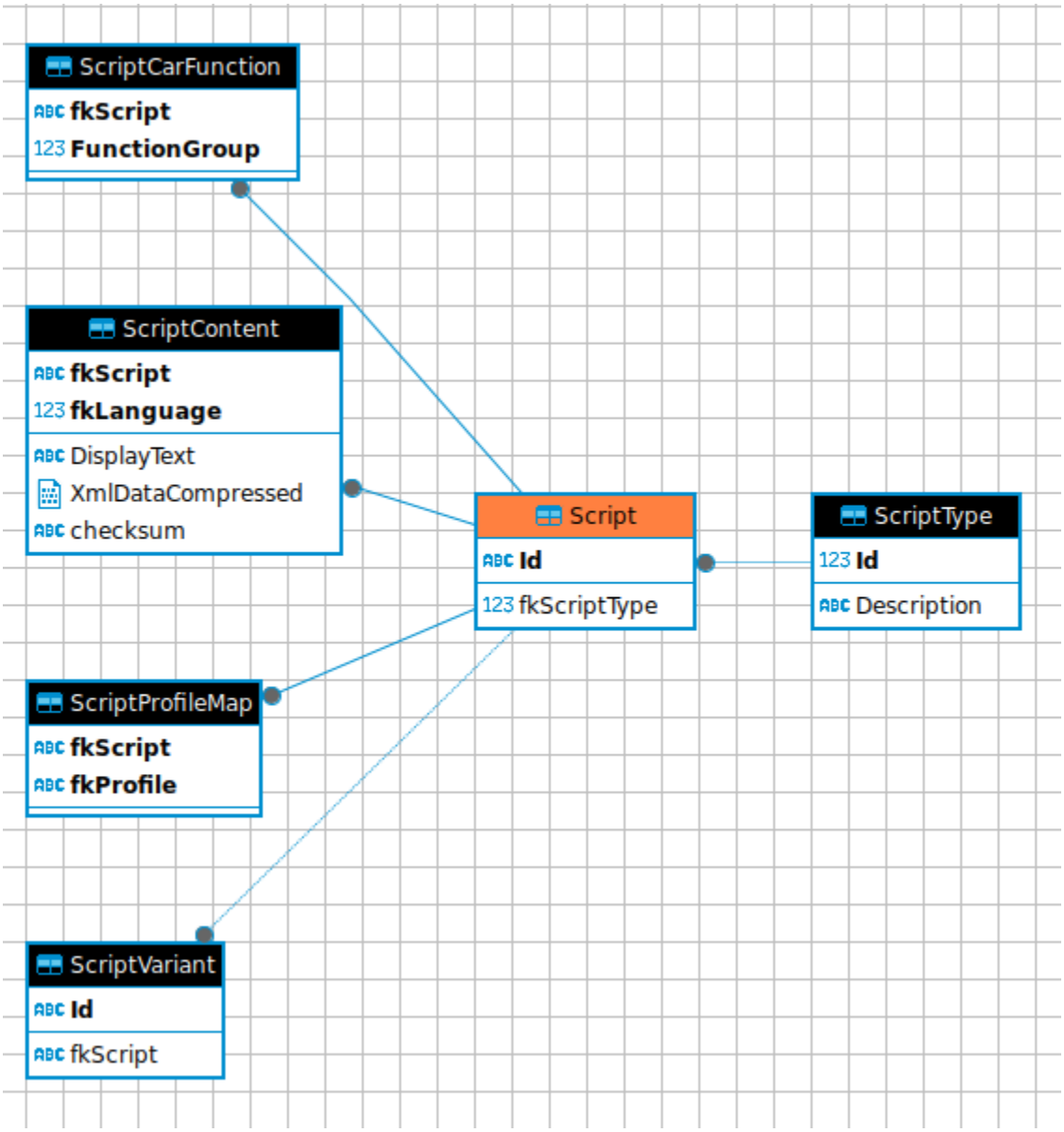
scripttype statusindentifier

in the DIAGSWDL database (ScriptCarFunction), Functiongroup (372302 exactly the same number refers to numerous script numbers)

Example : 0900c8af81b6a712 which is SUN_SENSOR,_VOLTAGE,_STATUS

4.3.1 in the XML file :

```
property class="Parameter" id="nev9960842n1-nev9960863n1"
identifier ecu="372304" ecuMode="EcuType" id="nev9960842n1-nev9960865n1" read="4" type=
↪"REID" value="5E01"
```



4.3.2 ScriptType

Checking for our purpose VehCommSpecification:

VehCommSpecification EcuIdentification CarStatus CarConfig ClearDtc ReadVin StatusIdentifier ReadDtc ReadDtcEcm Dro TransportMode CustomerParameter PreProg EcuReplace PreCompleteReadout AlarmMode KeyPos DownloadMode EcuIdentificationHighSpeed EcuIdentificationLowSpeed ClockValue PostReset PreSwdl

4.4 Freeze Frame Param (diagnostic session)

in CARCOM T148 blockmetaPARA is a parameter “showAsFreezeFrame”

the DIAGSWDL database contains data from a diagnostic session.

Vehicle speed refers to blockId, paramId (CARCOM, T141_Block)

id	fkT142	fkT143	name	fkT190_Text	offset	length	exclude	composite
4760627	8	2	Vehicle speed	24409	96	8	0	0

- offset = 96
- length = 8
- fkT142 = PARAM
- fkT143 = “unsigned”

CHAPTER

FIVE

DECODING EXPLAINED:

In order to save space, they compressed scripts using zip.

To read the script, you'll have some work …

so 0x504… is xml_data_compressed_hex

First we remove the "0x" prefix and convert the remaining hex string to binary data

xml_data_bin = bytes.fromhex(xml_data_compressed_hex[2:])

this data we write to disk as a zip file (and subsequently uncompress it using unzip)

```
import csv
import subprocess
import os

def sanitize_filename(filename):
    invalid_chars = [' ', '\\', '/']
    for char in invalid_chars:
        filename = filename.replace(char, '_')
    return filename

# Replace 'your_input.csv' with the actual CSV file name
input_csv_file = 'modified_exportvida.csv'

with open(input_csv_file, 'r', newline='', encoding='utf-8') as csvfile:
    csv_reader = csv.reader(csvfile)

    for row in csv_reader:
        # Extract fields from the CSV row
        fk_script = row[0]
        fk_language = row[1]
        display_text = row[2]
        xml_data_compressed_hex = row[6]

        # Sanitize the display_text for a filename
        sanitized_display_text = sanitize_filename(display_text)

        # Remove the "0x" prefix and convert the remaining hex string to binary data
        xml_data_bin = bytes.fromhex(xml_data_compressed_hex[2:])

        # Create a sanitized filename with .zip extension and write the binary data to it
        output_file_name = f'{display_text}_{row[4]}.zip'
```

(continues on next page)

(continued from previous page)

```
        with open(output_file_name, 'wb') as output_file:
            output_file.write(xml_data_bin)

        print(f"Saved binary data for {display_text} to {output_file_name}")
```

5.1 volvo model logic

mdl544yr2005eng167, mdl545yr2005eng167, mdl542yr2006eng167, mdl544yr2006eng167, mdl545yr2006eng167, mdl542yr2007eng167, mdl544yr2007eng167, mdl545yr2007eng167, mdl533yr2007eng167, mdl544yr2008eng167, mdl545yr2008eng167, mdl533yr2008eng167, mdl542yr2008eng167, mdl533yr2008eng199, mdl542yr2008eng199, mdl545yr2008eng199, mdl544yr2008eng199, mdl533yr2009eng199, mdl544yr2009eng199, mdl545yr2009eng199, mdl542yr2009eng199, mdl544yr2010eng199, mdl545yr2010eng199, mdl542yr2010eng199, mdl533yr2010eng199, mdl533yr2011eng199, mdl544yr2011eng199, mdl545yr2011eng199, mdl542yr2011eng199, mdl544yr2012eng199, mdl545yr2012eng199, mdl533yr2012eng199, mdl542yr2012eng199

The decoded XML file contains a description of all the models it applies to.

mdl545yr2008eng167

- mdl545 is a volvo estate (multiwagon)
- yr2008 is the constructionyear : 2008
- eng167 is a 1.6 diesel engine

So is it important to select only those XML files in which your model appears.

5.2 sniffing in XML files

unzip Read_out_MOST_nodes_372302.zip

372302 is the CEM

after unzipping we get a file like this one : 0900c8af81d1cb6c

we can have a look: *xmllint –format 0900c8af81d1cb6c | less*

- first we need to check if our model mdl545yr2008eng167 is in this file, if not it applies to other models

5.3 diagnostic extract CEM example

```
372302,100,REID,DD06,Strömförsörjningsläge
372302,2,BLOFF,FC,Serviceintervall
372302,4,REID,3F98,ECM start signal
372302,4,REID,5F30,Coolant water temp
372302,100,REID,4007,Workshop test (TPMS) status
372302,100,REID,4125,Batteritemperatur
372302,100,REID,DD02,MECU spänningsmatning
372302,100,REID,DD07,Strömförsörjningsläge
372302,100,REID,DD00,Global tid
372302,100,REID,DD00,Global real time
372302,4,REID,3F02,Clutch pedal sensor
```

5.4 reading accelerator pedal position via CAN

<identifier ecu=”284101” ecuMode=”EcuType” read=”4” type=”REID” value=”0141”>

5.5 Brake light switch

<identifier ecu=”284101” ecuMode=”EcuType” read=”4” type=”REID” value=”005C”>

5.6 Turn indicator

641901,4,REID,0005,Turn indicator

5.7 Hidden features

There are numerous scripts, for instance about Vehcom, which could be Vehicle communication ?

<identifier ecu=”372304” read=”4” type=”REID” value=”9A00”> <name=”Interior light, relay” textid=”74202”/>

this one is for activating the feature?

<identifier ecu=”372302” stop=”16” type=”REID” value=”8F31” write=”15”> <name=”Rear wash motor relay” textid=”13829”>

<identifier ecu=”372304” read=”4” type=”REID” value=”8D04”> <name=”Brake light, left” textid=”72127”/>

<identifier ecu=”372304” read=”4” type=”REID” value=”8D05”> <name=”Brake light, right” textid=”72128”/>

<identifier ecu=”372304” read=”4” type=”REID” value=”8D01”> <name=”Brake light switch” textid=”4716”/>

5.8 CAN diagnostic messages HOWTO

CHAPTER

SIX

READING THE VIN (VEHICLE IDENTIFICATION NUMBER)

VIN stands for Vehicle Identification Number. It is a unique code assigned to every motor vehicle when it's manufactured. VINs are composed of 17 characters (digits and capital letters) and serve as a vehicle's fingerprint, providing information about its manufacturer, model, features, and more. VINs are used for various purposes including vehicle registration, tracking recalls, insurance records, and theft prevention.

At the start of the VIDA communication, it does not know your Volvo. (VIDA does not know nothing)

6.1 sniffing vida dice

- notice ID = 0x726
- DB 0 = 0x03 (meaning 3 bytes of data?)
- DB 1 = 0x22
- DB 2 = 0xf1 (address 0xF114 as specified in xml)
- DB 3 = 0x14

The following script I got from the internet, but it shows that it loads a script 'Read VIN first readout (Odometer Value) (No immo check)' *'Read_VIN_first_readout_(Odometer_Value)_(No_immo_check)': 0900c8af8184c004*

```
[CarStatus       ][00F][Info]    Init CarStatus
[ScriptProvider  ][00F][Event]   Database: DiagSwdlRepository, SP: GetScript,  Type:
↪'ReadVin' ScriptId: '' Language: 'en-US' EcuType: '-1' Profile: 'EMPTYPROFILE,'
[ScriptProvider  ][00F][Info]    Fetched script: 'VCC-225200-1 1.8' title: 'Read VIN␣
↪first readout (Odometer Value) (No immo check)'
[Script          ][00F][Info]    Running script: 'VCC-225200-1 1.8' title: 'Read VIN␣
↪first readout (Odometer Value) (No immo check)'
[CarComRepository][011][Event]   Database: CarCom, SP: general_GetEcuId, EcuId: 30728270␣
↪AA, Result: 821
[DiagnosticVehCom][011][Info]    Using diagnostic part number '30728270 AA' for system␣
↪type: '0'
[CarComRepository][011][Event]   Database: CarCom, SP: vadis_GetEcuVariantData,␣
↪EcuVariant: 821
```

6.2 Fetched script

The script with title 'VCC-225200' got fetched. As it happens this script can be found in a VIDA database table.

fkScript,fkLanguage,DisplayText,XmlDataCompressed,checksum

'Read_VIN_first_readout_(Odometer_Value)_(No_immo_check)'

uncompressing this : 0900c8af8184c004

which is an XML file

6.3 xml content

as it happens this part of the script applies to my car There is a value : 0xF114, sending this to the ECU should get some VIN values.

```
<node class="Components.ReadVinComponent" id="nev9333590-nev15368413n1" nodeid="EuCD" x=
↪"620" y="275">
  <property class="Int" id="nev9333590-nev15368414n1" name="CanHiSpeedValue" type="In"␣
↪value="500000" visible="0"/>
  <property class="Int" id="nev9333590-nev15368415n1" name="UseRelay" type="In" value="0
↪" visible="0"/>
  <property class="Byte" id="nev9333590-nev15368416n1" name="BusId" type="In" value="1"␣
↪visible="0"/>
  <property class="String" id="nev9333590-nev15368417n1" name="Identifier" type="In"␣
↪value="F114" visible="0"/>
  <property class="String" id="nev9333590-nev15368418n1" name="Mode" type="Internal"␣
↪value="EcuVariant" visible="1"/>
  <property class="Int" id="nev9333590-nev15368419n1" name="ServiceId" type="In" value=
↪"100" visible="0"/>
  <property class="String" id="nev9333590-nev15368420n1" name="EcuName" type="Out" value=
↪"" visible="1"/>
  <property class="Response" id="nev9333590-nev15368421n1" name="Result" type="Out"␣
↪value="" visible="1"/>
  <property class="String" id="nev9333590-nev15368422n1" name="EcuVariant" type="In"␣
↪value="30728270 AA" visible="0"/>
  <extension id="nev9333590-nev15368423n1"/>
</node>
```

Now what happens next ?

```
[CarComRepository][010][Event] Database: CarCom, SP: general_GetEcuId,
EcuId: 30728270 AA, Result: 821
```

It gets to the CarCom database (just a microsoft sql mdf file-somewhere at the back of this manual there is a procedure to access these files under linux)

SP: general_getEcuID

- SP = stored procedure
- Carcom = database

select id from T100_Ecuvariant where identifier = @ecuIdentifier

so it will get the Ecu variant in the T100 table where EcuId="30728270 AA"

- it comes up with ecuvariant 821
- next a stored procedure vadis_GetEcuVariantData is executed

the following is an example of such a stored procedure, it might not be entirely correct, since I used ocr software ..

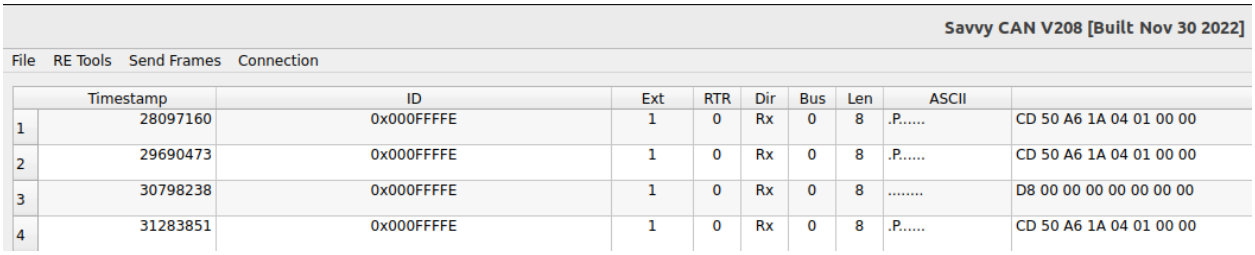
```
SELECT
T100_EcuVariant. identifier AS DiagNumber,
102_EcuType.identifier AS EcuTypeldentifier,
T101_Ecu.identifier AS Eculdentifier,
T121_Config.commAddress AS EcuAddress,
T100_EcuVariant.id AS EcuVariantid,
T102_EcuType.description AS EcuName,
T121_Config.id As Configid,
T121_Config.fkt121_Config_Gateway As GatewayConfigId,
T122_Protocol.id AS Protocolld
FROM
T100_EcuVariant
INNER JOIN
T101_Ecu ON T100_EcuVariant.fkT101_Ecu = T101_Ecu.id
INNER JOIN
T102_EcuType ON T101_Ecu.fkT102_EcuType
INNER JO1N
T120_Config_EcuVariant ON T100_EcuVariant.id = T120_Config_EcuVariant.fkT100_Ecu
Variant
INNER JOIN
T121_Config ON T120_Config_EcuVariant.fkT121_Config = T121_Config.id
INNER JOIN
T122_Protocol ON T121_Config.fkT122_pratocol
WHERE (T100_EcuVariant.id = @ecuVariantid)
ORDER BY Priority
```

6.4 Stored procedure : general_GetEcuId

CHAPTER

SEVEN

VOLVO DIAGNOSTIC MESSAGES



Savvy CAN V208 [Built Nov 30 2022]

File RE Tools Send Frames Connection

	Timestamp	ID	Ext	RTR	Dir	Bus	Len	ASCII	
1	28097160	0x000FFFFE	1	0	Rx	0	8	.P.....	CD 50 A6 1A 04 01 00 00
2	29690473	0x000FFFFE	1	0	Rx	0	8	.P.....	CD 50 A6 1A 04 01 00 00
3	30798238	0x000FFFFE	1	0	Rx	0	8	.....	D8 00 00 00 00 00 00 00
4	31283851	0x000FFFFE	1	0	Rx	0	8	.P.....	CD 50 A6 1A 04 01 00 00

7.1 Sniffing the VIDA diagnostic messages:

ID 0x000FFFFE (this is the VIDA tool identifier)

```
000FFFFE CD 11 A6 01 96 01 00 00
    |     |  |  |  |    |
    |     |  |  |  |    '--------- Number of responses expected (query only)
    |     |  |  |  '-------------- Operation ID/Sensor ID (01 96)
    |     |  |  '----------------- Type of operation
    |     |  '-------------------- Module id in CEM
    |     '----------------------- Message length (always C8 + data byte length)
    '--------------------------- Start of frame
```

7.2 applied to a real message:

(message) CD 50 A6 1A 04 01 00 00

```
message length = 0xCD-0xC8=5
0x50 module ID CEM
0xA6 Read Current Data By Identifier
0x1A04 372302,4,REID,1A04,Ignition key position
0x1 the number of responses expected

so VIDA is checking key position
```

(message) CD 50 A6 1A 02 01 00 00

```
372302,4,REID,1A02,Read 30-supply
guess VIDA is checking the voltage of the CEM
```

7.3 Guessing the reply

We are expecting a response on this: so module 50 has to reply the ID of replies should be 0x01200021

(message) 01200021 | CE 50 E6 1A 04 0C 0D 00

Our request was an A6, so E6 must mean response by identifier. The identifier was the key position : 0x1A04

Now remains to guess which values represent the key position ?

I found a clue in the DHA database :

> RESPONSE_ITEM [NAME=IgnitionKeyPos Ignition key not inserted][SEND_ABLE=FALSE][NOTE=IgnitionKeyPos] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=Ignition key not inserted] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00000111] [COMP_VALUE==0b00000000] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE] RESPONSE_ITEM [NAME=IgnitionKeyPos Pos 0][SEND_ABLE=FALSE][NOTE=IgnitionKeyPos] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=Pos 0] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00000111] [COMP_VALUE==0b00000100] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE] RESPONSE_ITEM [NAME=IgnitionKeyPos Pos I][SEND_ABLE=FALSE][NOTE=IgnitionKeyPos] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=Pos I] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00000111] [COMP_VALUE==0b00000101] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE] RESPONSE_ITEM [NAME=IgnitionKeyPos Pos II][SEND_ABLE=FALSE][NOTE=IgnitionKeyPos] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=Pos II] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00000111] [COMP_VALUE==0b00000110] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE] RESPONSE_ITEM [NAME=IgnitionKeyPos Pos III][SEND_ABLE=FALSE][NOTE=IgnitionKeyPos] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=Pos III] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00000111] [COMP_VALUE==0b00000111] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE] RESPONSE_ITEM [NAME=IgnitionKeyPosQF][SEND_ABLE=FALSE][NOTE=IgnitionKeyPosQF] [NO_OF_BYTES=1] [OFFSET=3] [MASK=] [UNIT=] [PRECISION=2] [SIGNED=U] [BASE=DEC] [FORMULA=x&0b00011000/8] [COMP_VALUE=] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_CHECK=TRUE]

END

7.4 Top tip : finding a script

All the Volvo scripts are under compressed format in an SQL table. My model is mdl545yr2009. Suppose I'm interested in the CCM (climate)module: this has the following ID = 871101.

```
grep 871101 0900* > tempfile # this gets me all the scripts for the module
grep mdl545yr2009 tempfile > investigatefile # this gets me only the scripts for my model
```

7.5 Script 0900c8af81d49c30 (EcuIdentification)

in this script all the modules are listed, together with their ID. Remarkable is this in the XML file : <node class="Components.VehComm" id="nev10084941n1-nev12929369n1" nodeid="ServiceB9F0" >

This same "B9F0" you can see in the initial communication. (logs)

7.6 Using VIDA logs

under VIDA/system/log/Diagnostics each diagnostic sessionlog is saved. it is filled with readable messages:

<Request EcuAddress="0B" DiagnosticPartNo="30670330 A" CompletePartNo="0008621154 A" Message="B9F0" OrderPositi

<Response Message="F9F000086211542020413067033020204 1"/>

</Request>

0B corresponds to the ADM module

<Request EcuAddress="50" DiagnosticPartNo="31254684 AC" CompletePartNo="0031254903 " Message="B9F8" Order

<Response Message="F9F800000000928700000000000000000000000000000000000000000000000000FFFFFFFFFFFFFF

</Request> <Request EcuAddress="50" DiagnosticPartNo="31254684 AC" CompletePartNo="0031254903 " Message="B9F5" OrderPosition="14">

<Response Message="F9F5003125467820414300004400003129619720414100004000003129611020414100FE8000"/>

</Request> <Request EcuAddress="50" DiagnosticPartNo="31254684 AC" CompletePartNo="0031254903 " Message="B9F7" OrderPosition="15">

<Response Message="F9F700307720412020200031253518202041070605040320204200000000020202 0FFFFFFFFFFF

</Request>

7.7 Using SQL MDF viewer

this tool lets you peek in de MDF (database) files without sqlserver.

not only data but also stored procedures

CHAPTER

EIGHT

MAKING SENSE

8.1 Reading a partnumber

8.1.1 Reading part number from ECU 0x50

```
---> ID=000ffffe data=50 88 00 00 00 00 00 00
<--- ID=00000003 data=50 8e 00 00 31 25 49 03
Part Number: 000031254903
```

```
CAN_HS ---> ID=000ffffe data=cb 50 b9 f0 00 00 00 00
CAN_HS <--- ID=01000003 data=8f 50 f9 f0 00 08 68 85
CAN_HS <--- ID=01000003 data=09 13 20 20 41 30 65 70
CAN_HS <--- ID=01000003 data=4c 24 20 20 41 00 00 00
```

First byte in reply is a technical field, like rolling counter, start (0x80)/end(0x40 flag of the multipart message, etc. The P/N is:

08 68 85 13 == 8688513

8.1.2 quick analysis

data = cb 50 b9 f0

- length = 3 (cb-c8)
- module = 50 (CEM)
- type of operation = b9 (read datablock by offset)
-

8.1.3 ECU Identification

```
B9F0
  can0  00400003   [8]  8F 50 F9 F0 00 08 69 07
  can0  00400003   [8]  09 19 20 20 20 30 72 89
  can0  00400003   [8]  4C 79 20 41 41 00 00 00

B9F1 - B9F4
  can0  00400003   [8]  CC 50 7F B9 12 00 00 00
```

8.1.4 Downloadable Software Part Number

```
B9F5
  can0  00400003   [8]  8F 50 F9 F5 00 30 78 63
  can0  00400003   [8]  09 53 20 41 41 00 00 C0
  can0  00400003   [8]  0A 00 00 30 78 62 13 20
  can0  00400003   [8]  0B 41 41 00 00 40 00 00
  can0  00400003   [8]  0C 30 72 89 81 20 41 41
  can0  00400003   [8]  4C 00 00 70 00 00 00 00
```

8.1.5 hex2human

CAN Config Part Number: 0030786353204141 (30786353 AA) Flash Sector Start Address: 0000C000

Application Software Part Number: 0030786213204141 (30786213 AA) Flash Sector Start Address: 00004000

Local Config Part Number: 0030728981204141 (30728981 AA) Flash Sector Start Address: 00007000

8.2 Response

suppose you send a request for information over the canbus, you want to listen for a certain respons as well.

How is this done ?

In the request message, there the “type of operation” is defined :

for example:

- A1 No Operation Performed (keep alive)
- A3 Security Access Mode
- A5 Read Current Data By Offset
- A6 Read Current Data By Identifier
- A7 Read Current Data By Address
- A8 Set Data Transmission
- A9 Stop Data Transmission

In the vida msql database (T111 service table) these values are defined. In this table is a field (definition) that describes how this value should “behave”.

8.3 example for code B9 (Read Data Block By Offset)

```
<request>
    <item name="Service" length="8" />
    <item name="Identifier" length="8" />
</request>
<response>
    <block id="ID2_1" type="BLOFF" length="8" name="Bloff" />
    <dynamiclist type="PARAM" name="Parameters" blockName="Parameter" parent="ID2_1" />
</response>
```

This XML snippet describes a structure for a request and a response.

In the request section:

- There are three items: - “Service” with a length of 8 units. - “Identifier” length of 8 units.

In the response section: - BLOFF (T142 blocktype) this refers to metaTable 146 (is not in database, but guess it is constructed out of XML file data depending on volvo model) - ID2_1 (cannot find any reference in database ...) - PARAM (T142 blocktype) refers to metaTable 148 (T148_metaPara) which gives all the parameters for an ECUvariant

8.4 VIDA diagnostic logs

When using VIDA diagnostic tools, at first it tries to get vehicle data. As it happens, this might give some insight.

(extract) <Request EcuAddress=”0B” DiagnosticPartNo=”30670330 A” CompletePartNo=”0008621154 A” Message=”B9F0” OrderPosition> <Response Message=”F9F000086211542020413067033020204 1”/>

In the Response Message I can see some of the request

30670330202041 similar to ”30670330 A” *0008621154202041* similar to ”0008621154 A”

8.5 apply this to real life example

```
CAN_HS ---> ID=000ffffe data=cb 50 b9 f0 00 00 00 00
CAN_HS <--- ID=01000003 data=8f 50 f9 f0 00 08 68 85
CAN_HS <--- ID=01000003 data=09 13 20 20 41 30 65 70
CAN_HS <--- ID=01000003 data=4c 24 20 20 41 00 00 00
```

- First byte in reply is a technical field (8F)
- Second byte is CEM ID (50)
- Third (F9) is like reply to B9
- Fourth (F0)
- Fifth 00
- PARTNUMBER 08688513 (08688513 A)
- SPACES 20 20
- ASCII A
- DIAGNOSTICPARTNUMBER (30657024 A)
- SPACES 20 20
- ASCII A
- PARTNUMBER cannot be found in the database *maybe something like a serialnumber*
- in (T100_ecuvariant) the DIAGNOSTICPARTNUMBER is found and refers to CEM 372302

8.6 using the DHA (diagnostic host application) to get a clue

this was easier to understand, and performs likely the same action

```
DIAG_ITEM [NAME=ECU: HW partnumber and DIA diagnostic number][SEND_ABLE=TRUE][NOTE=]␣
↪[FORMAT=OTHER:2] [BASE=HEX] [VALUE=F0] [TYPE=BLOFF][INPUT=FALSE]
BEGIN
  RESPONSE_ITEM [NAME=ECU Complete Part Number .][SEND_ABLE=FALSE][NOTE=ECU Complete␣
↪Part Number] [NO_OF_BYTES=8] [OFFSET=2] [MASK=] [UNIT=.] [PRECISION=2] [SIGNED=U]␣
↪[BASE=5 BCD 3 ASCII] [FORMULA=x] [COMP_VALUE=] [DEP_RESPITEM=CHECKOK] [DEP_RESPITEM_
↪CHECK=TRUE]
  RESPONSE_ITEM [NAME=ECU Diagnostic Software number .][SEND_ABLE=FALSE][NOTE=ECU␣
↪Diagnostic Software number] [NO_OF_BYTES=7] [OFFSET=10] [MASK=] [UNIT=.] [PRECISION=2]␣
↪[SIGNED=U] [BASE=4 BCD 3 ASCII] [FORMULA=x] [COMP_VALUE=] [DEP_RESPITEM=CHECKOK] [DEP_
↪RESPITEM_CHECK=TRUE]
END
```

the response is defined :

- offset 2 and 8 databytes
- offset 10 and 7 databytes

according to this the PARTNUMBER would be *00*08688513

8.7 test-setup

I used a setup on a stm32 to simulate the accelerator pedal.

stm32f103c6-canbus

This microcontroller sends values to the CANBUS, which can be picked up by the dice software.

In the dice diagnostic software this parameter can be display graphically. (graph)

```
frame.id = 0x00222222; //pretend this is the CEM module
frame.extended_id = true; //volvo uses 29bit
frame.rtr = false;
frame.len = 8;
// Populate the payload bytes with accelerator pedal 0x3F92
frame.data[0] = 0xCE; //length = 5
frame.data[1] = 0x50; //CEM module
frame.data[2] = 0xE6; //reply to A6
frame.data[3] = 0x3F; //accel pedal
frame.data[4] = 0x92; //16bit value
frame.data[5] = 0x24; //value
frame.data[6] = 0x01; //number of messages
frame.data[7] = 0x00;
CAN::transmit(frame);
```

CHAPTER

NINE

HACKING : MODIFYING STANDARD SETTINGS

I browsed some forums, and sometimes I would find traces of ECU modders, hackers. Sometimes this offers some insight into how something actually works.

9.1 Checks between the ECM, CEM and BCM

The check between the ECM, CEM, and BCM occurs when the ignition key is turned to position II (takes approximately 100 ms). After the CEM has approved the key, the ECM is informed that this is the case.

The following checks occur before the engine is started:

9.1.1 Check 1

The ECM transmits a request to the BCM to transmit its serial number. The ECM checks that the serial number corresponds to the serial number it has stored. If the serial number is correct, the start conditions have been met.

9.1.2 Check 2

At the same time as the ECM checks the BCM, the ECM transmits a random number and a code to the CEM. The CEM checks that the code corresponds to the code it has stored. If the code is correct, the CEM transmits another code back to the ECM. The ECM checks the code and compares it to the code it has stored. If it is correct, another start condition has been met.

When Check 1 and Check 2 are complete and approved, the CEM permits the ECM to activate the relays for the starter motor and the engine can be started.

from the manual

Checks between the ECM, CEM and BCM The check between the ECM, CEM and BCM occurs when the ignition key is turned to position II (takes approximately 100 ms). After the CEM has approved the key, the ECM is informed that this is the case. The following checks occur before the engine is started: Check 1 The ECM transmits a request to the BCM to transmit its serial number. The ECM checks that the serial number corresponds to the serial number it has stored. If the serial number is correct the start conditions have been met. Check 2 At the same time as the ECM checks the BCM, the ECM transmits a random number and a code to the CEM. The CEM checks that the code corresponds to the code it has stored. If the code is correct the CEM transmits another code back to the ECM. The ECM checks the code and compares it to the code it has stored. If it is correct another start condition has been met. When Check 1 and Check 2 are complete and approved the CEM permits the ECM to activate the relays for the starter motor and the engine can be started.

9.2 Pincodes

CEM PIN, ECM PIN, IMMO code, and other codes in Volvo cars: why some units can't be easily cloned via OBD and can't be used from another vehicle without modifications.

Every Volvo since 1998 has two basic security levels: CEM and ECM. In newer vehicles, additional units like IEM, VGM, etc., are integrated into this security system.

The content of the programming memory in these units is essentially divided into two parts:

- The first programming part of the memory requires a unique PIN code to overwrite the software, dictating how the unit behaves. For example, the software version and configuration table in the CEM unit or, typically from 2008 onwards, the engine maps in the ECM unit.
- The second programming part is memory requiring an IMMO PIN, which protects the EEPROM memory where synchronization codes are stored, typically essential for starting the vehicle. It involves synchronizing a unique number between the central CEM unit and the ECM engine unit. If this code does not match, the vehicle will not start. In newer vehicles, additional units like the BCM brake unit, the SCL steering wheel lock, ignition keys, remote controls, and lastly, audio system units that communicate over the MOST optical network, are integrated into the synchronization system.

For most vehicles, the CEM/ECM PIN and CEM IMMO can be decoded using VDASH or VDD. This allows most vehicle operations to be performed without the need to remove the unit and physically connect it to a specific programmer. Units that cannot be decoded via OBD can usually be decoded by removing and reading the unit, known as "on the bench."

Another way to obtain the PIN and IMMO code is to upload software from the official VIDA diagnostic and capture the programming key during the upload. However, Volvo considers this method illegal and in direct violation of the Volvo VIDA licensing agreement.

9.2.1 some examples:

- CEM PIN – for changing vehicle configuration (e.g., activating navigation, changing the US>EU market, removing the speed limiter, etc.), uploading a newer software version

  decoding via VDASH over OBD – P3, SPA 2015-2018, SPA 2021 if we know the VGM PIN, P2GGD (V8, 3.2i, D5 147kW), P1 GGD (typically 2011+) decoding via VDD – P1, P2 from MY2005+, P3, SPA 2015-2018, SPA 2021 if we know the VGM PIN

- CEM IMMO – adding ignition keys, changing CEM-ECM synchronization codes, pairing a new SCL or SCU, pairing audio system components

  decoding via VDASH P2 except for V8, 3.2i, D5 147kW, P3 except for V40/V40CC

- ECM PIN – performance optimization, EGR / DPF removal, deactivation of error codes

  not needed for P2, not needed for P1 until approximately MY2011 decoding via OBD at P3 DENSO units, SPA 2015-2018 using UCBP cannot be decoded at P3 5-cylinder diesel engines

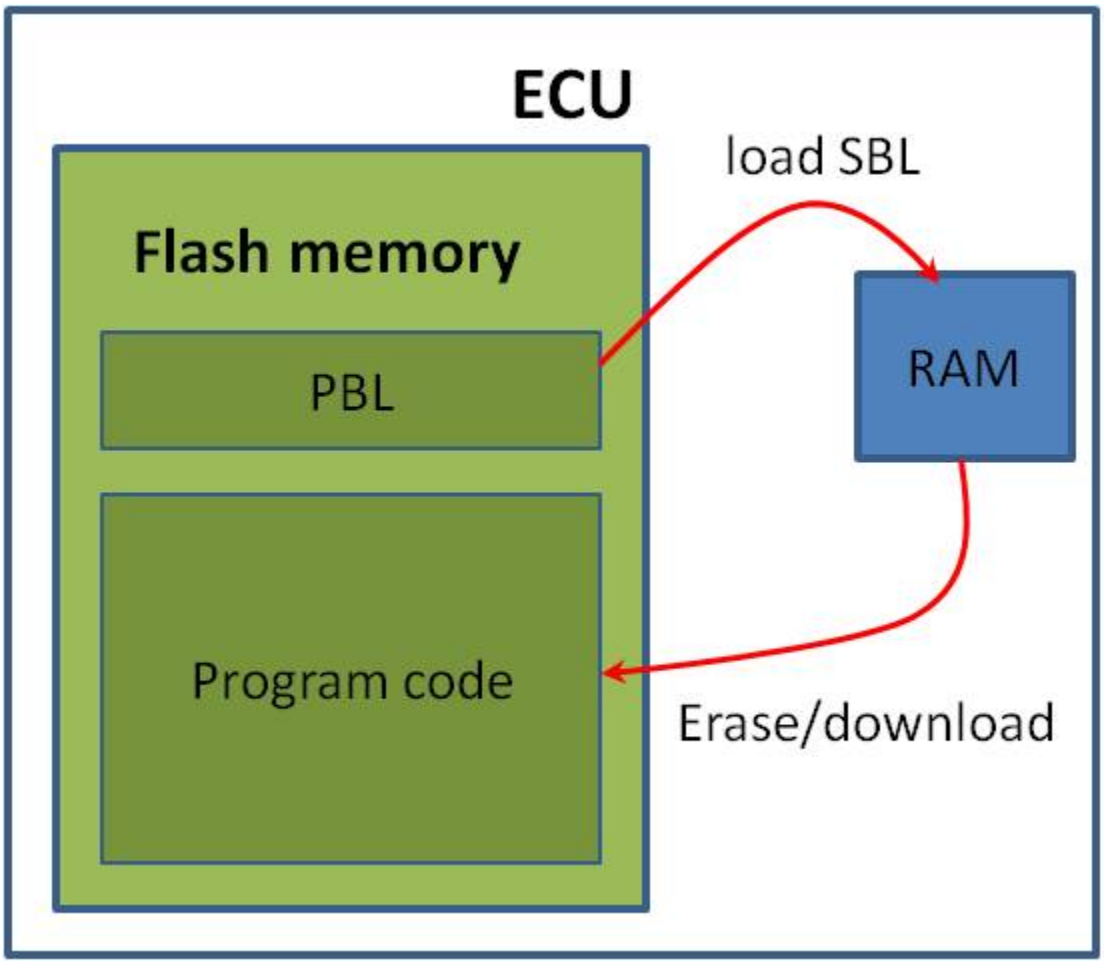
- ECM IMMO – changing the CEM-ECM synchronization key, pairing the BCM unit

  decoding via VDASH P1 1.6D 80kW and D5 and 2.4D 5-cylinder, P2 except for V8, 3.2i, D5 147kW, DENSO VEA P3 and SPA 2014 – 2018

- VGM PIN – only in vehicles with the iCUP (Android) system for erasing diagnostic codes throughout the vehicle and for any other vehicle operations

  Cannot be decoded via OBD, must be obtained from the official Volvo VIDA diagnostics

9.3 Bootloaders



When initializing, the ECU first initiates the boot loader. The boot loader comprises two distinct parts: a primary boot loader (PBL) and a secondary boot loader (SBL). In its default mode, the PBL loads the application software stored in the flash memory. Due to specific constraints such as a small memory size (16k) and the inability to be modified post-production, the PBL is unsuitable for programming or updating data in the flash memory. For these tasks, the SBL is utilized. The SBL is transferred by the primary loader into RAM and subsequently activated. It then assumes responsibility for managing the flash memory, including tasks such as erasing or reprogramming, and the software download process.

9.4 The software download process

To download data, the ECU should be put in programming mode. First the SBL is downloaded into RAM by means of PBL. After SBL is activated, clear the flash memory. The next step is the download of data files. Finally the ECU is reset to erase RAM and put back in default mode.

9.5 Keys

CEM EEPROM:

From the xx256 MCU eeprom. I found based on some other forum posts where the keys are stored (transponder portion at least). They are each 4 bytes and there are a maximum of 6 keys that can be added to the car.

YV1MS682442020566 MCU2 EEPROM (4K)

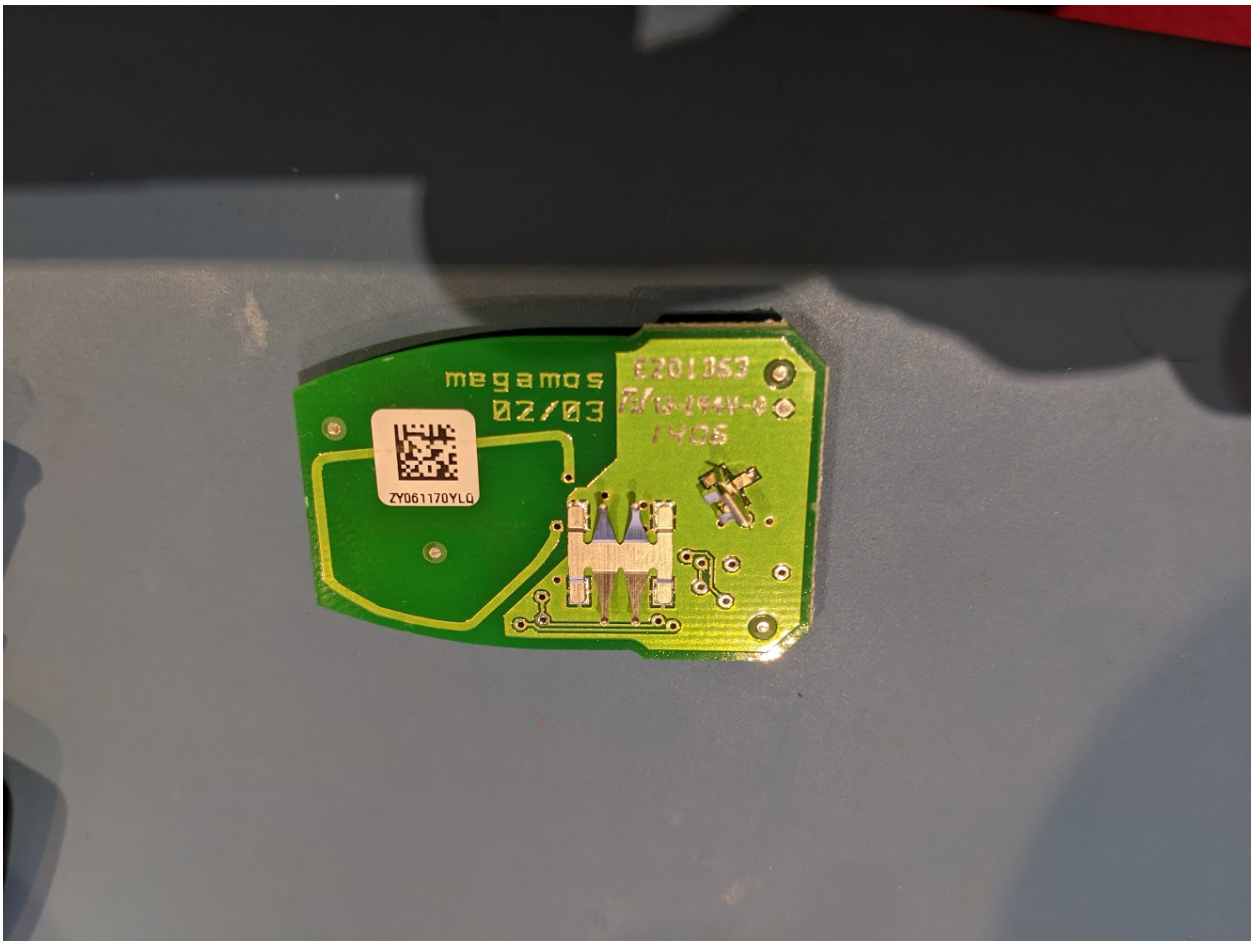
I do not know if there are any checksums on this.

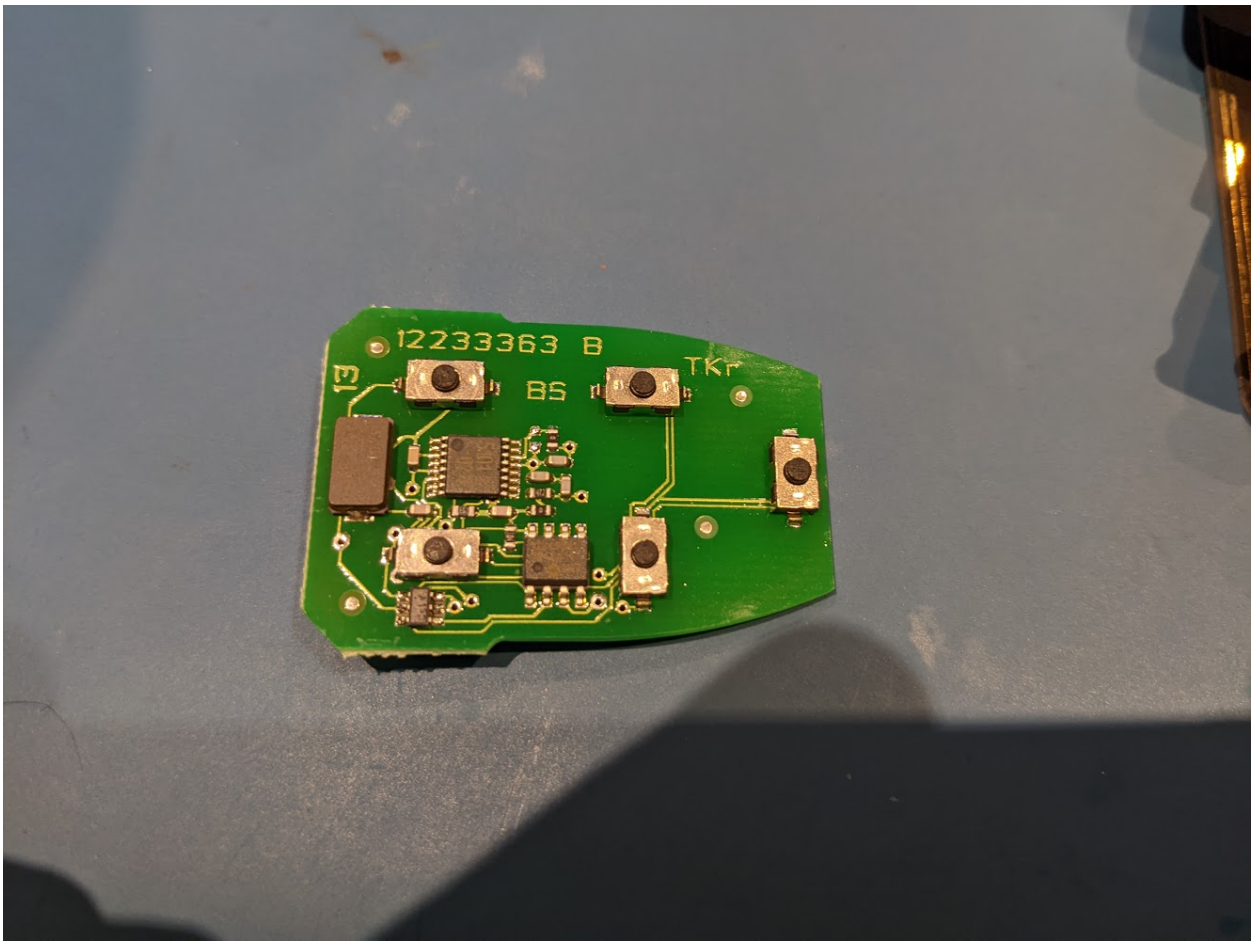
Above you can see that the vehicle has 4 transponders/keys added, with two sets of 4 bytes blank for slots 5 and 6. On the ICM display, if I go into the information section it also reports there are 4 keys.

The keys I believe are ID48 Megamos transponders.

9.5.1 Key Fob:

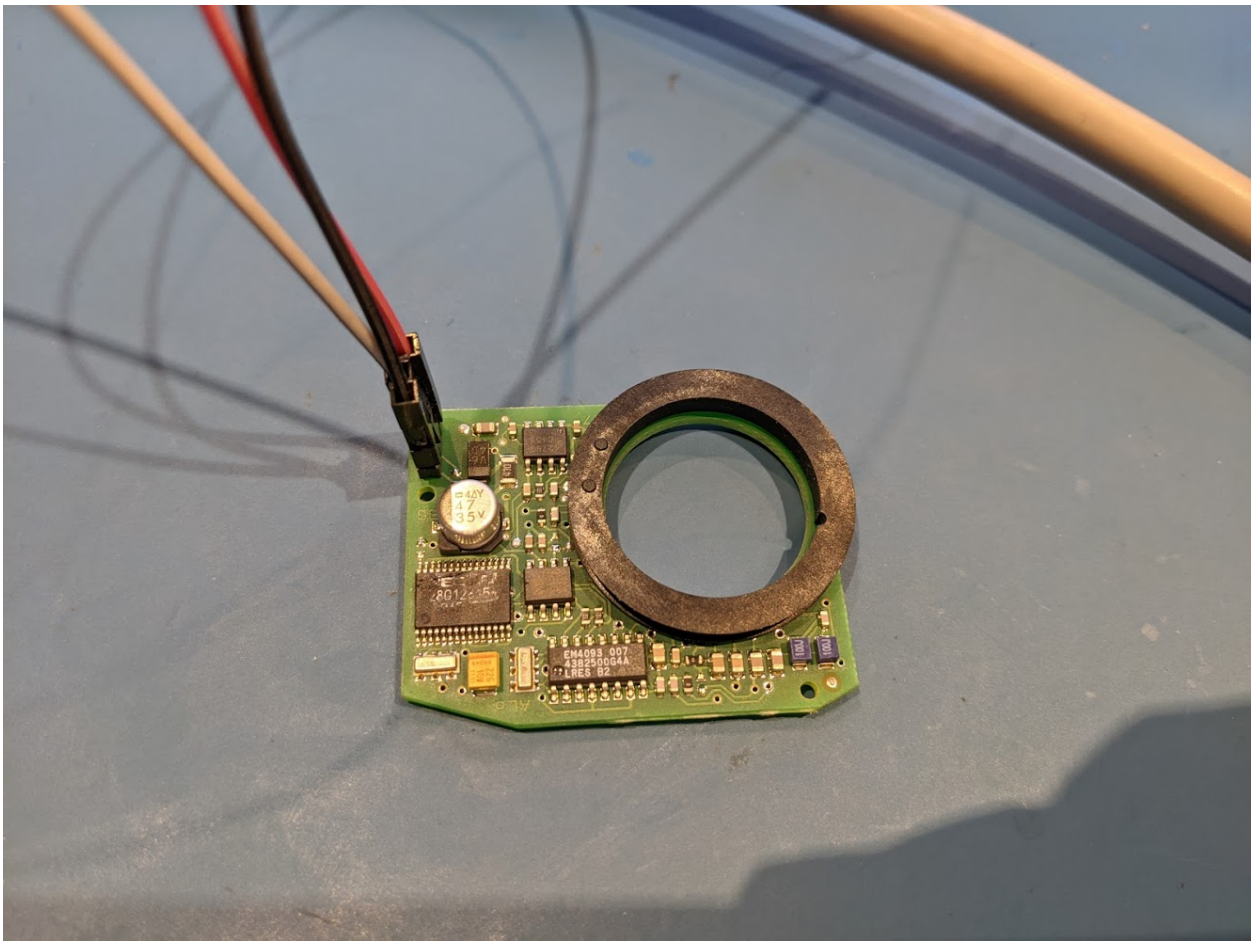
There is the active component that I believe transmits at 315MHz when you push the buttons. This is picked up by a receiver on top of the DIM (Cluster). The passive part is in the plastic housing, a small glass RFID transponder that when inserted into the ignition is verified.


megamos
02/03
ZY061170YLQ
E201353
1405



9.5.2 Ignition Transponder Reader:

This is a 3 wire device, 12V, GND and LIN. The LIN bus is shared with the SCL (Steering Column Lock) module, which are both connected to the CEM.


47
35V
EM4095 007
LRES B2

CHAPTER

TEN

DBC (DATABASE CONTAINER) FILES: THE KEY TO DECODING CAN BUS MESSAGES

https://www.csselectronics.com/pages/dbc-editor-can-bus-database

10.1 example : VehicleSpeed

Vehiclespeed for a Volvo C30 has CAN ID 2104136, but this if followed by the speed data. In a dbc file is defined how this speed can be decoded from CAN messages. **BO_** 2182103350 VehicleSpeedBCM: 8 BCM

SG_ VehicleSpeed : 55|16@0+ (0.01,0) [0|65535] ” km/h” XXX

10.2 short intro

In the world of automotive communication systems, Controller Area Network (CAN) bus plays a crucial role in facilitating communication between various electronic control units (ECUs) within vehicles. Understanding and decoding CAN bus messages is essential for tasks such as vehicle diagnostics, monitoring, and control.

DBC (Database Container) files serve as the cornerstone for decoding CAN bus messages. Developed by Vector Informatik, DBC files are structured databases that define the parameters, signals, and message formats used in CAN bus networks. These files provide a standardized format for describing the communication protocols and message data within a CAN network.

Within a DBC file, each message is defined with its identifier, data length, and the signals contained within it. Signals represent specific data parameters transmitted within CAN messages, such as vehicle speed, engine RPM, or sensor readings. Each signal is associated with attributes such as scaling factors, offsets, and units, which are crucial for interpreting the raw data received from the CAN bus.

DBC files also include information about the physical layout of the data within CAN messages, such as the byte order (big-endian or little-endian) and the bit arrangement of individual signals. This information is essential for correctly parsing and decoding the data received from the CAN bus.

In practical applications, DBC files are used in conjunction with software tools known as CAN bus analyzers or decoders. These tools parse incoming CAN bus data using the information provided in the DBC file, allowing users to interpret and visualize the data in a human-readable format. This enables engineers, developers, and technicians to monitor vehicle performance, diagnose faults, and develop new automotive functionalities with ease.

In summary, DBC files serve as invaluable resources for decoding CAN bus messages, providing a standardized and structured approach to understanding the communication protocols used in automotive networks. With DBC files and the appropriate decoding tools, unlocking the wealth of data flowing through the CAN bus becomes accessible, empowering automotive professionals to innovate and optimize vehicle systems effectively.

CHAPTER

ELEVEN

OPENMOOSE

Since I have a Dice cable (usb to OBD2), I wondered if this is usable to send/receive custom messages. Openmoose is a project to re-write the ECU using the dice cable, so figured it should be usable for diagnostic purposes.

http://www.openmoose.net/

https://github.com/rlinewiz/OpenMoose

Read and write your Volvo ECU (Bosch ME7) using a DiCE cable.

11.1 modding

jetbrains dotPeek was used to have a look at the J2534.dll.

I used a vbox which contained all the microsoft development, but it crashed often (my fault?).

https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

11.2 code

The following code was found in the J2534.dll, and shows how a can data-message is composed. This can be used to send custom messages.

```
   public static readonly CANPacket msgCANReadECMConfig = new CANPacket(new byte[8]
{
  (byte) 203,
  (byte) 122,
  (byte) 185,
  (byte) 245,
  (byte) 0,
  (byte) 0,
  (byte) 0,
  (byte) 0
});
```

The extended ID, if not provided is a standard diagnostic EID like 0xFFFFFE.

```
public CANPacket(byte[] data)
{
  this.data = new byte[data.Length + 4];
  this.data[0] = (byte) 0;
```

(continues on next page)

(continued from previous page)

```
  this.data[1] = (byte) 15;
  this.data[2] = byte.MaxValue;
  this.data[3] = (byte) 254;
  for (int index = 0; index < data.Length; ++index)
    this.data[index + 4] = data[index];
  this.protocolType = PROTOCOL_TYPE.CAN_XON_XOFF;
}
```

The EID can be given as a parameter as well.

```
public CANPacket(byte[] data, int eid)
{
  byte[] eid1 = new byte[4]
  {
    (byte) 0,
    (byte) 0,
    (byte) 0,
    (byte) eid
  };
  eid1[2] = (byte) (eid >> 8);
  eid1[1] = (byte) (eid >> 16);
  eid1[0] = (byte) (eid >> 24);
  this.setupCANPacketEID(data, eid1);
}
```

CHAPTER

TWELVE

USING CANDUMP

on linux, first load the modules, then start the deamon with 500K baud (-s6)

```
modprobe can
modprobe can-raw
modprobe slcan
slcand -s6 -S2000000 /dev/ttyACM1 can0
ifconfig can0 up
```

12.1 using DICE cable and modding OpenMoose

after modifying OpenMoose, I get to see my custom-message (22,23,24,25,26,27,28) which proves that the DICE cable can be used!

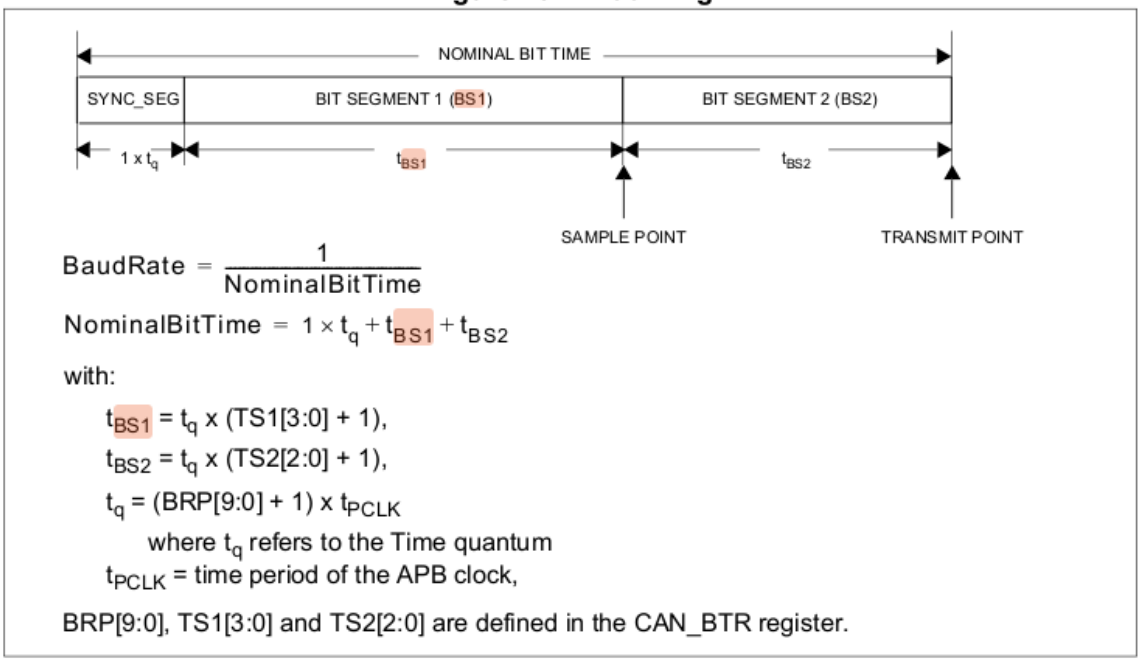
```
root@naj-Latitude-5520:~# candump can0 -ta
 (1710516027.391935)  can0  000FFFFE   [8]  D8 00 00 00 00 00 00 00
 (1710516028.391723)  can0  000FFFFE   [8]  D8 00 00 00 00 00 00 00
 (1710516028.594302)  can0  000FFFFE   [8]  16 17 18 19 1A 1B 1C 1D
 (1710516029.391926)  can0  000FFFFE   [8]  D8 00 00 00 00 00 00 00
```

CHAPTER

THIRTEEN

BAUD & BIT

Figure 234. Bit timing



In the context of STM32 microcontrollers, the APB (Advanced Peripheral Bus) clock is one of the buses used for connecting peripherals, and it's typically derived from the system clock (HCLK). The time period of the APB clock refers to the duration of one clock cycle on this bus. It is the reciprocal of the frequency.

The formula to calculate the time period (T) is:

[T = frac{1}{text{APB clock frequency}}]

Where:

- (T) is the time period.
- (text{APB clock frequency}) is the frequency of the APB clock.

So, if you have the frequency of the APB clock, you can calculate the time it takes for one clock cycle.

For example, if the APB clock operates at 36 MHz, the time period ((T)) would be:

[T = frac{1}{36 , text{MHz}} approx 27.78 , text{ns}]

Understanding the time period of the APB clock is important when configuring peripherals that depend on timing, such as setting up the baud rate for communication peripherals like USART, SPI, or CAN. These peripherals often have timing parameters specified in terms of clock cycles or time intervals, so knowing the time period allows you to set these parameters accurately.

CHAPTER

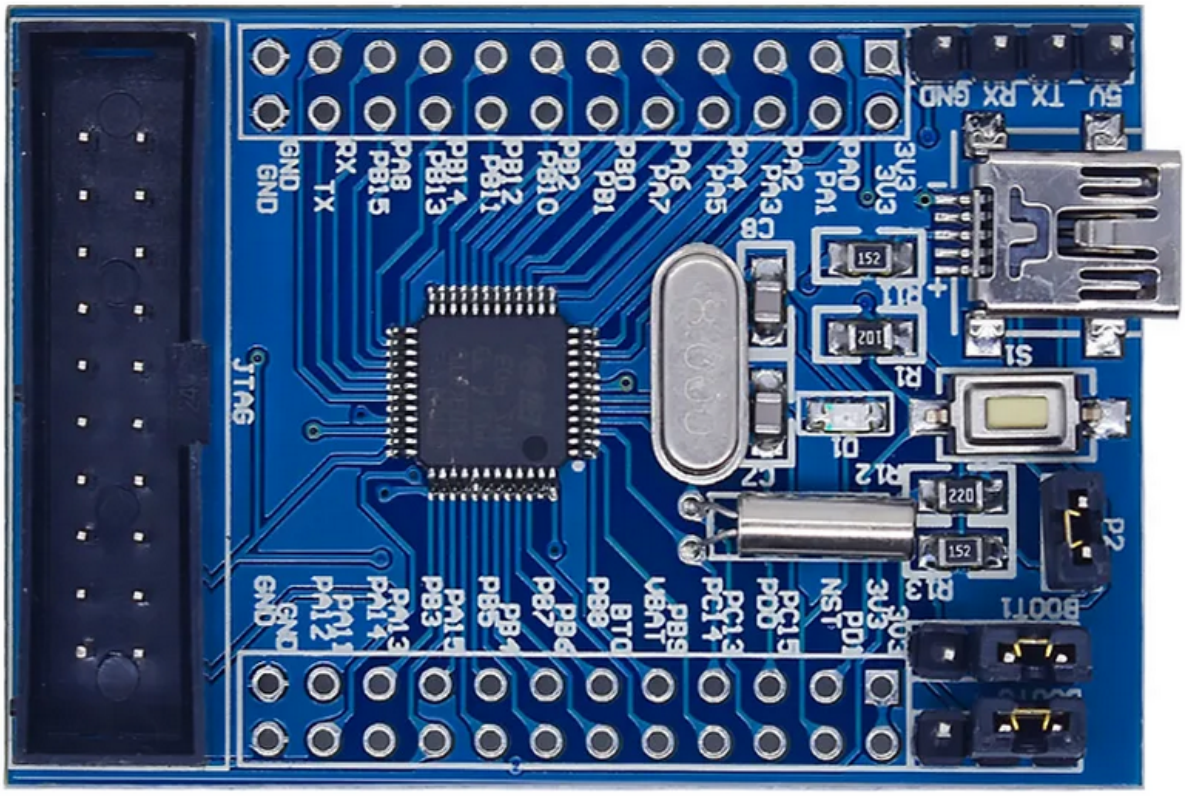
FOURTEEN

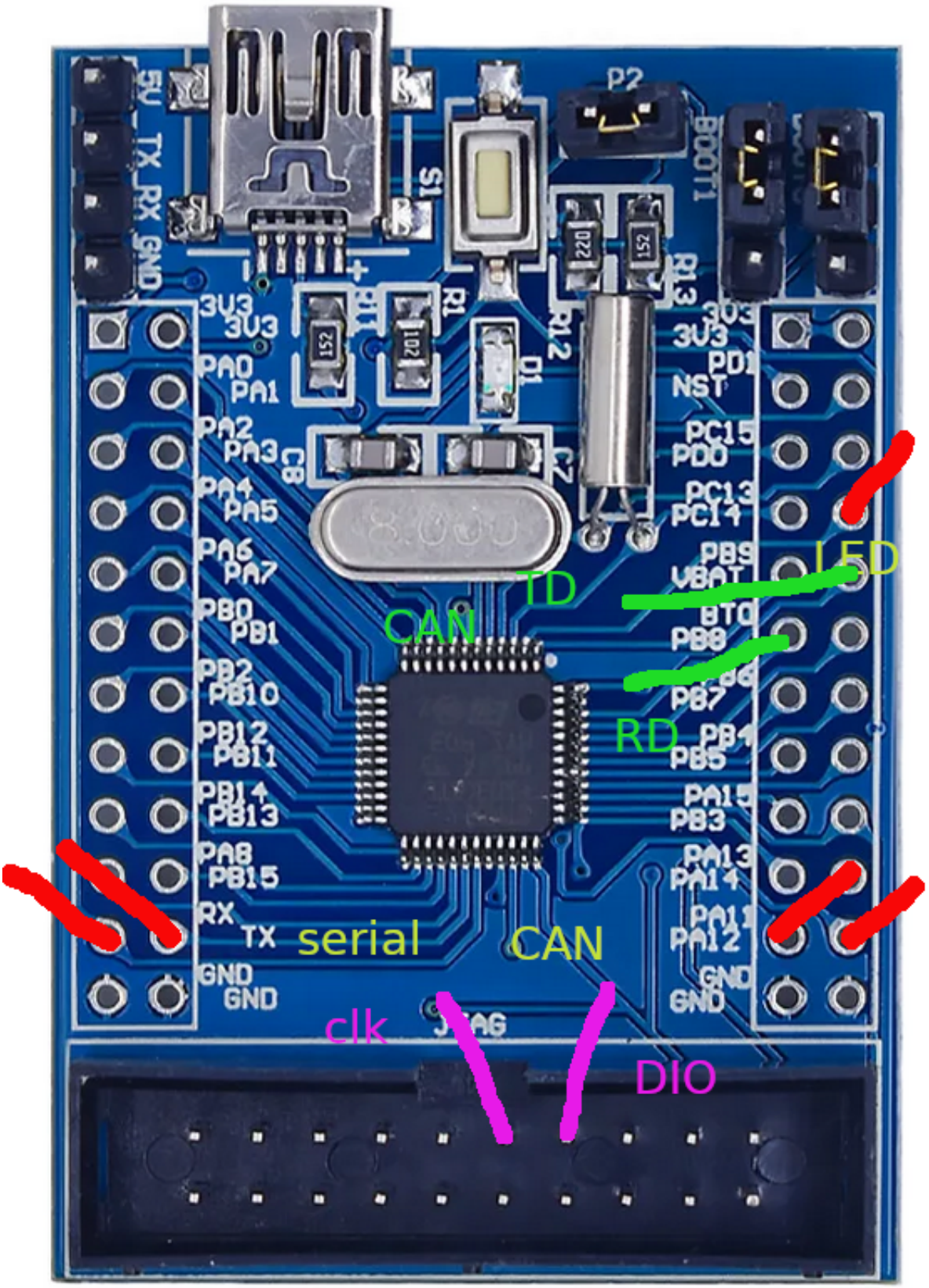
STM MICROCONTROLLERS USED WITH CAN

Since I was inspired by the openinverter project to acquire some new skills (https://openinverter.org/wiki/Main_Page), I went trough my drawer to scramble all the STM hardware together, since that is what they use. The programming is done using libopencm3.

Eventually I bought some cheap stm32 boards which were low spec, but still usefull for CAN communication.

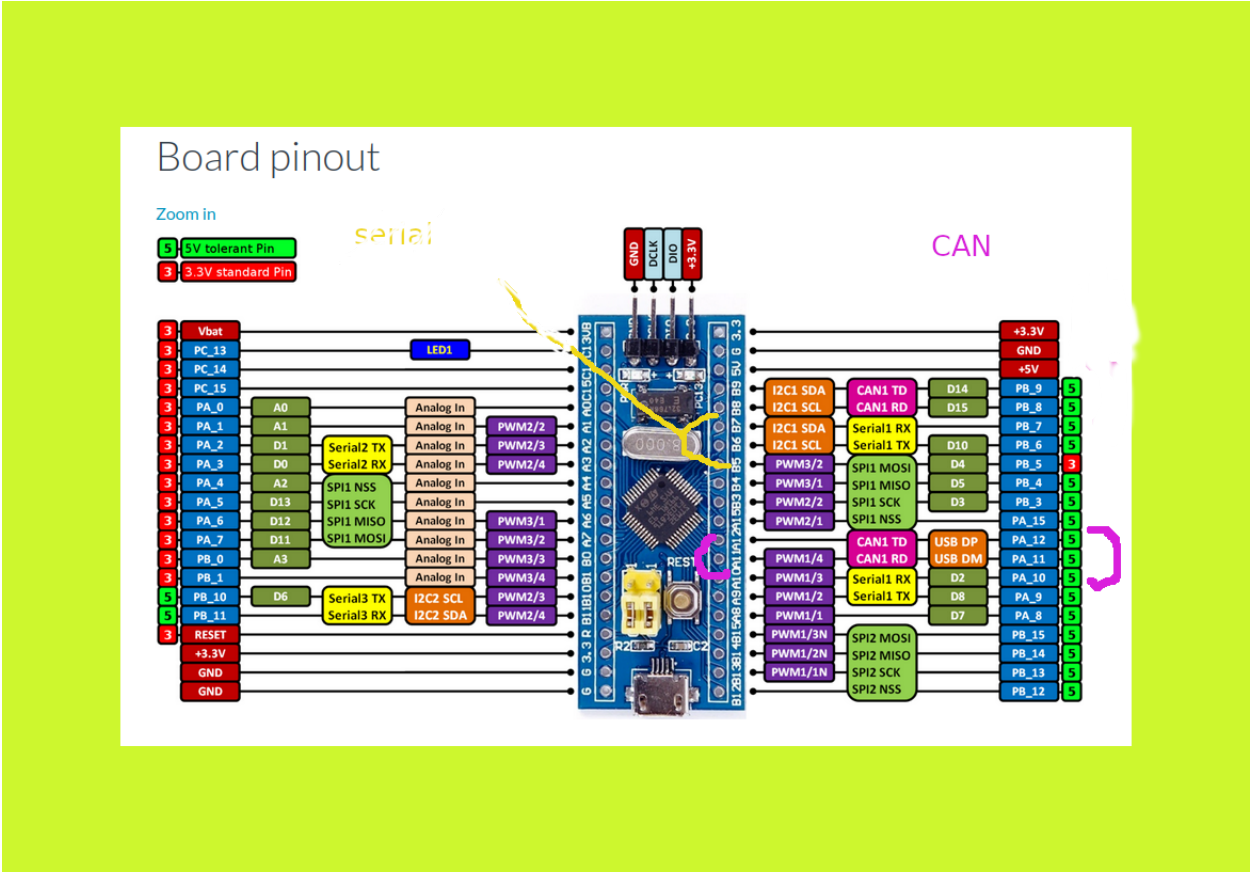
14.1 stm32f103




5V
TX
RX
GND
3V3
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PB0
PB1
PB2
PB10
PB12
PB11
PB14
PB13
PA8
PB15
RX
TX
GND
GND
P2
BOOT1
S1
R1
R11
R12
R13
D1
C8
C7
8.000
PD1
NST
PC15
PD0
PC13
PC14
PB9
VBAT
PB8
PB7
PB5
PA15
PB3
PA13
PA14
PA12
GND
GND
TD
CAN
RD
LED
serial
CAN
clk
DIO


Session 1
Saleae Logic
5 M samples
500 kHz
+8820800 µs
+8820820 µs
+8820840 µs
+8820860 µs
+8820880 µs
+8820900 µs
+8820920 µs
+8820940 µs
+8820960 µs
CAN: Bits
Show this row
Show All
Hide All
Stuff bit
Bit
CAN: Fields
ID
Full ID
CAN payload data
Start of frame
End of frame
Identifier
Extended identifier
Full identifier
Identifier extension bit
Data length count
CRC sequence
CRC delimiter
ACK slot
ACK delimiter
CAN: Warnings
Can decoding

14.1.1 Bluepill



14.2 stm32f103

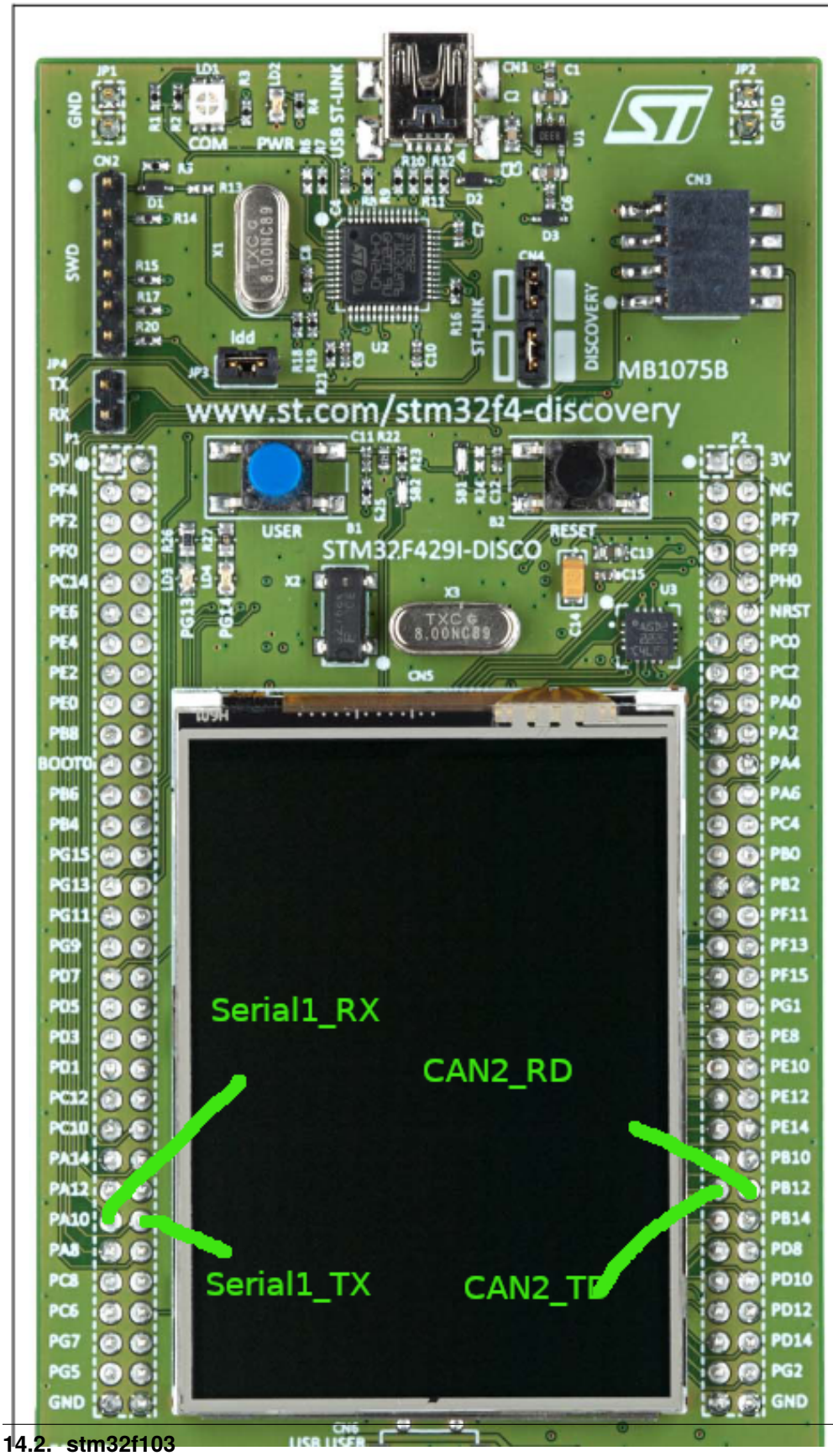
```
static void MX_CAN2_Init(void);
PB6 (TX) en PB12 (RX)

void HAL_CAN_MspInit(CAN_HandleTypeDef* hcan)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(hcan->Instance==CAN2)
  {
  /* USER CODE BEGIN CAN2_MspInit 0 */

  /* USER CODE END CAN2_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_CAN2_CLK_ENABLE();
    __HAL_RCC_CAN1_CLK_ENABLE();

    __HAL_RCC_GPIOB_CLK_ENABLE();
    /**CAN2 GPIO Configuration
    PB12      ------> CAN2_RX
    PB6      ------> CAN2_TX
```

(continues on next page)

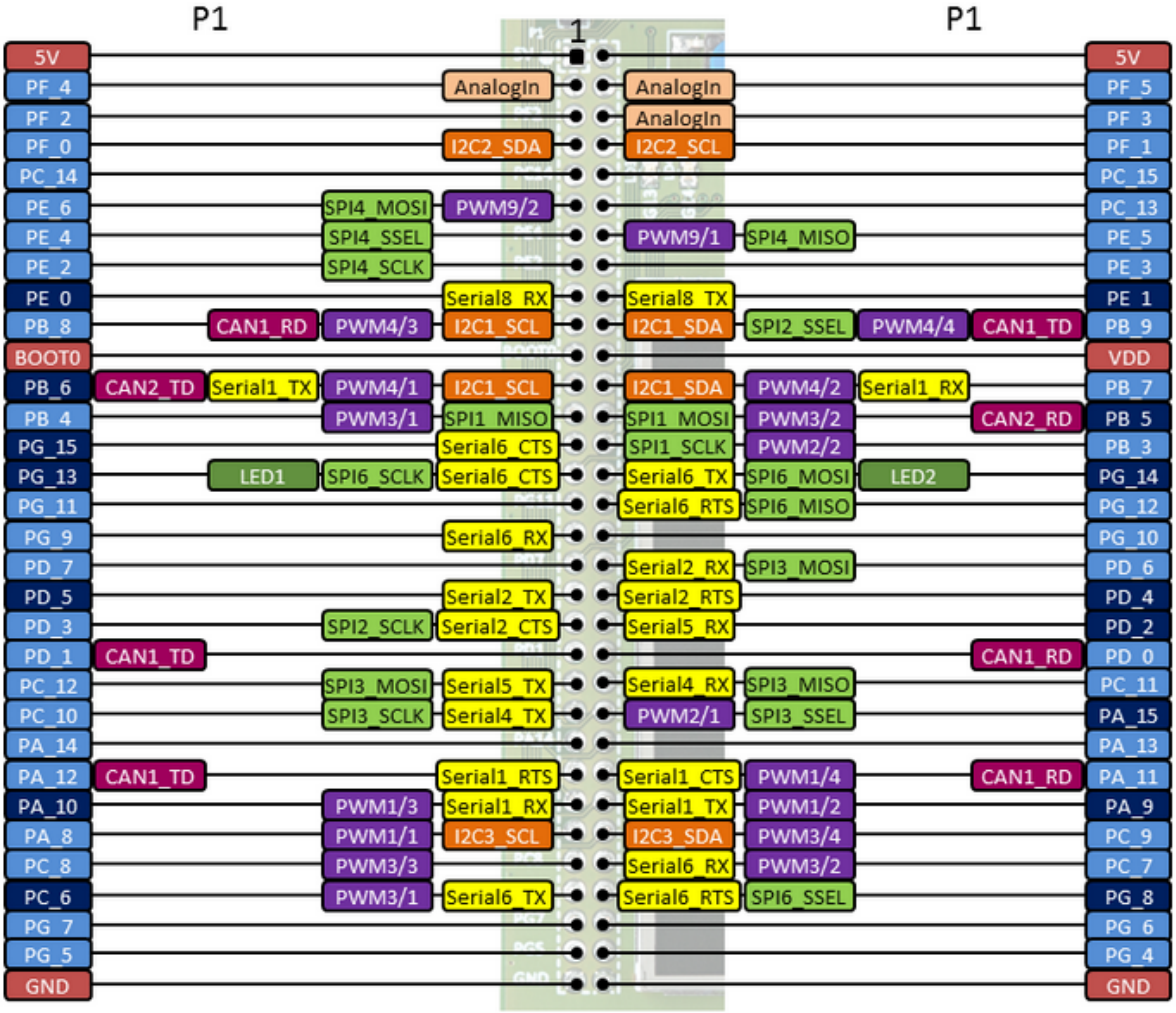
(continued from previous page)

```
    */
    GPIO_InitStruct.Pin = GPIO_PIN_12|GPIO_PIN_6;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF9_CAN2;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* USER CODE BEGIN CAN2_MspInit 1 */

  /* USER CODE END CAN2_MspInit 1 */
  }
}
```

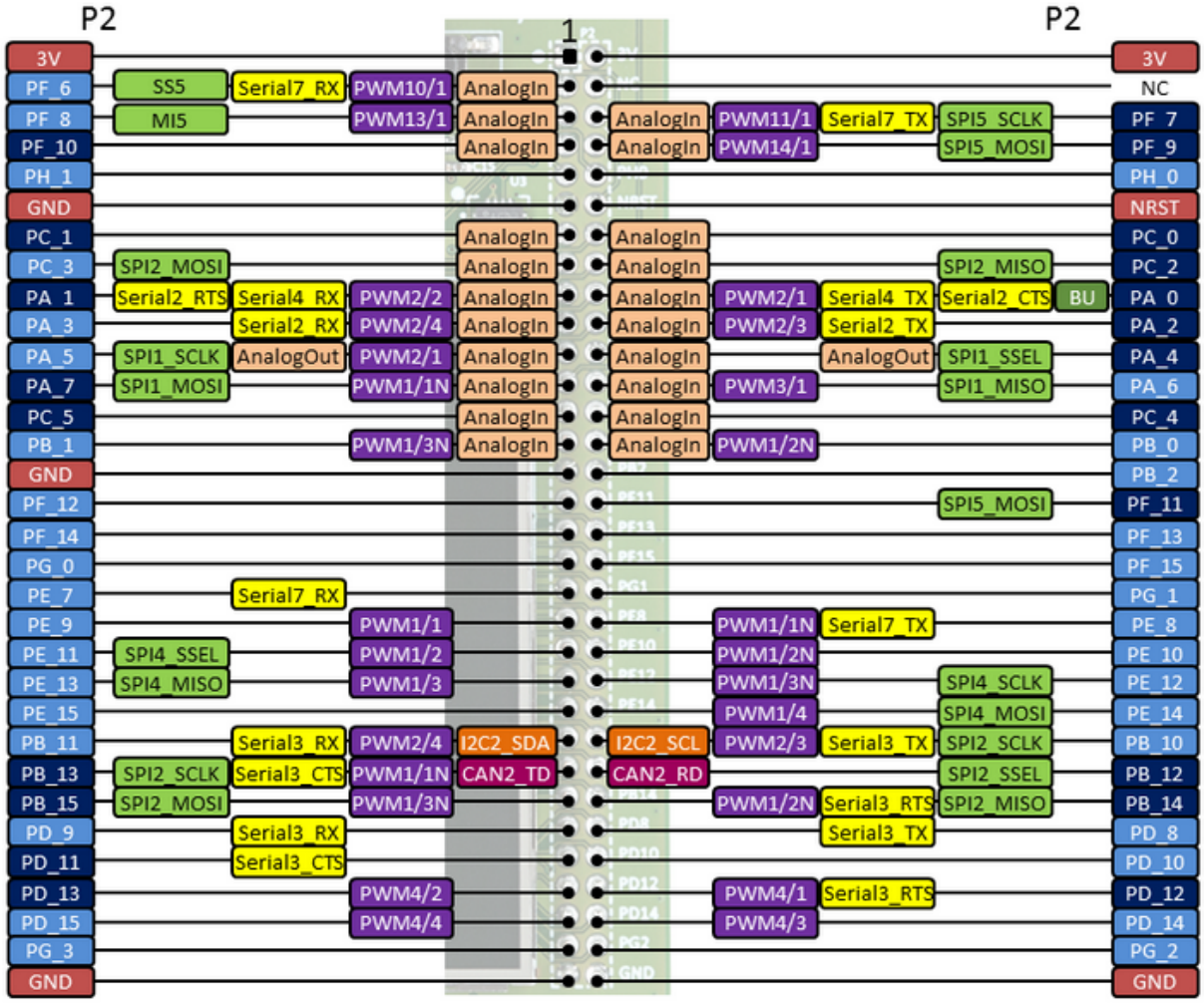
14.2.1 board overview



14.2.2 pinheader left



14.2.3 pinheader right

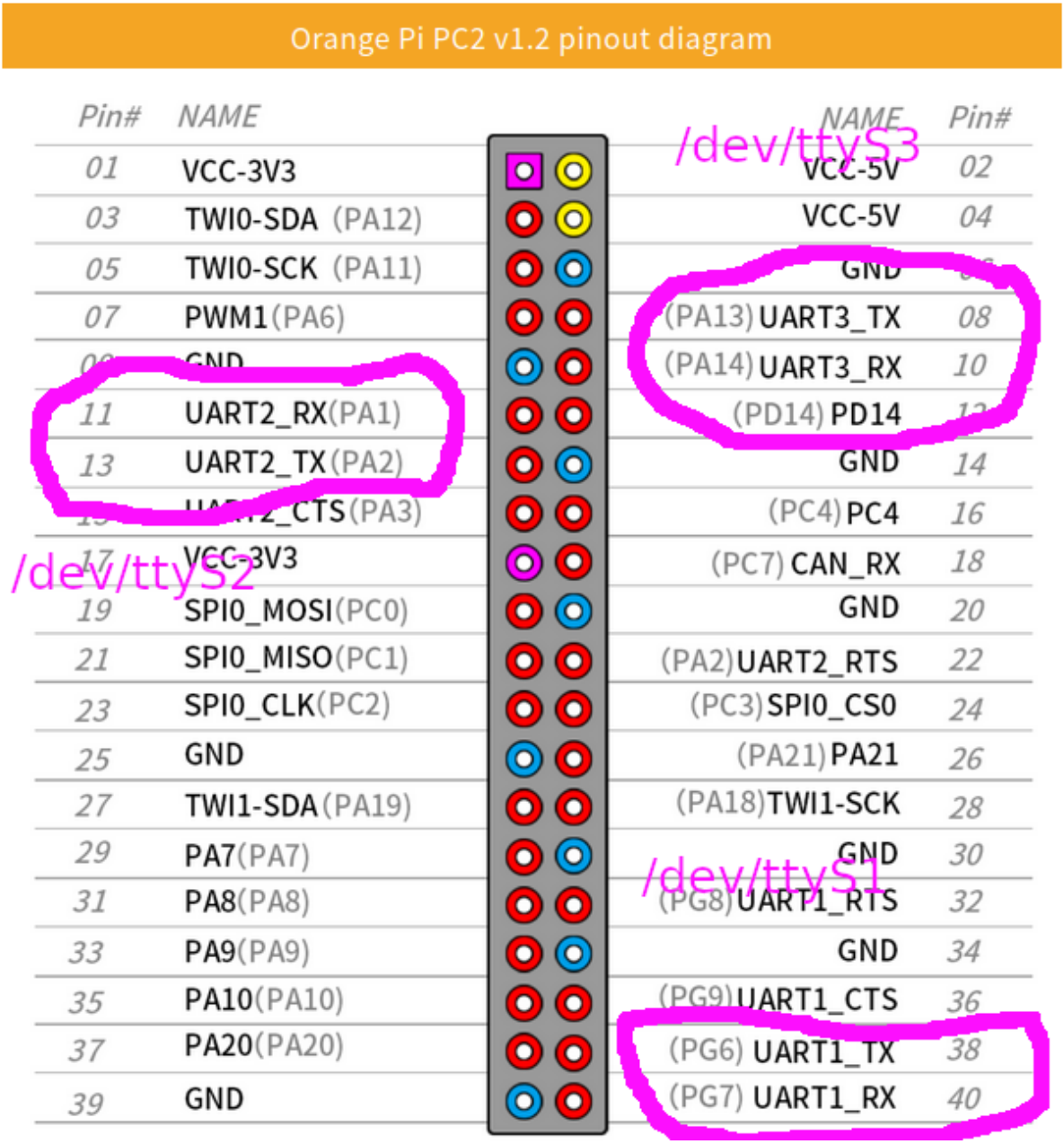


14.3 using a serial port

Due to non working usb-uart plugs, I used what I had : an orange pi pc, running armbian. This allowed me to configure 3 serial ports.

Later on in my project I configured a stm32f103c6 to be used as an uart bridge.

14.3.1 using the GPIO header of an orange pi



I use armbian on a orange pi.

```
sudo armbian-config
```

In the Armbian configuration utility, navigate to *System* > *Hardware* and select the appropriate option to apply device tree overlays.

CHAPTER

FIFTEEN

WHAT IS A SOCKETCAN INTERFACE?

SocketCAN is a set of open-source CAN drivers and a networking stack contributed by Volkswagen Research to the Linux kernel. CAN (Controller Area Network) is a standard communication protocol used in embedded systems such as automotive and industrial applications for message-based communication between microcontrollers and other devices.

SocketCAN provides a standardized API (Application Programming Interface) for CAN communication in the Linux environment. It allows applications to communicate with CAN devices using a set of socket-based interfaces similar to those used for networking. This abstraction makes it easier for developers to work with CAN devices, as they can use familiar socket programming techniques.

Key features of SocketCAN include:

1. **Standardized API**: SocketCAN provides a consistent API for CAN communication, making it easier for developers to write applications that work with different CAN devices.
2. **Socket-based Interface**: Applications interact with CAN devices using sockets, which allows for easy integration into existing network programming paradigms.
3. **Support for Multiple CAN Devices**: SocketCAN supports multiple CAN devices simultaneously, allowing for complex CAN networks to be easily managed.
4. **Integration with Linux Kernel**: SocketCAN is integrated into the Linux kernel, making it readily available on Linux-based systems without the need for additional drivers.

Overall, SocketCAN simplifies the development of applications that need to communicate over CAN networks on Linux systems, providing a standardized and efficient interface for CAN communication.

15.1 Configuring SocketCAN interface : stm32multiserial

STM32F103 Bluepill devices are known for their affordability and versatility. These microcontroller units (MCUs) can be easily configured to function as a virtual COM port, making them popular choices for various embedded systems and DIY projects.

However, a peculiar design flaw limits the simultaneous use of the CAN (Controller Area Network) port and the Virtual COM port on these devices. This limitation poses a challenge for applications requiring both functionalities concurrently.

To circumvent this limitation, I adopted a workaround solution: utilizing two STM32 devices instead of one. By dedicating one device to forward data to the serial port and another to forward this data to the virtual COM port, I effectively bypassed the restriction. Fortunately, the low cost of these Bluepill devices made this workaround feasible, despite the need for additional hardware.

Although unconventional, this approach proved effective in achieving the desired functionality without compromising on cost-effectiveness. It underscores the adaptability and flexibility of STM32F103 Bluepill devices in addressing real-world constraints and challenges in embedded systems development.

The serial ports of the Orange Pi PC seemed limited to 115200 baud. When sampling 500k baud CAN traffic, this becomes a problem. The STM32 devices allow for higher baudrates. I modified the original STM32MultiSerial UART2 port to allow for higher baudrates.

on github is this stm32multiserial repository, which is also suited for the stm32f103c6 so stm32f103xx.

STM32MultiSerial GitHub Repository

if all goes well this device will show up as (dmesg)

```
[ 6497.544752] usb 3-2: Product: STM32 Virtual ComPort
[ 6497.544756] usb 3-2: Manufacturer: STMicroelectronics
[ 6497.544759] usb 3-2: SerialNumber: 00000000001A
[ 6497.550462] cdc_acm 3-2:1.0: ttyACM0: USB ACM device
[ 6497.551289] cdc_acm 3-2:1.2: ttyACM1: USB ACM device
```

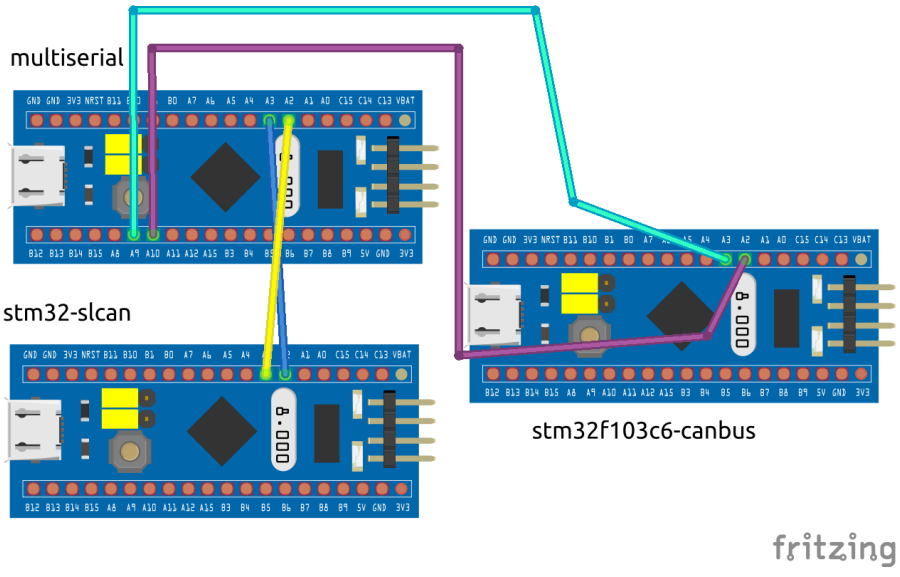
in this case /dev/ttyACM0 and /dev/ttyACM1 will be usable as virtual com port

this works together with the serial can :

STM32-SLCAN

```
sudo modprobe can
sudo modprobe can-raw
sudo modprobe slcan
sudo slcand -s5 -S2000000 /dev/ttyUSB0 can0 # CAN Speed 5 ->250 kBaud, 2,000,000 Baud␣
↪serial
(or)
su slcand -s4 -S2000000 /dev/ttyACM1 can0 #125kBaud CAN and stm32multiserial device
ifconfig can0 up
```

the -S2000000 is the serial baudrate, which is very high, standard usb-uart devices might not be able to cope



CHAPTER

SIXTEEN

LINUX AT HOME

I prefer linux to windows, but a lot of software in the CAN space is windows. I resolved this partly by using a virtualbox, porting the VIDA database to linux. My cheap logic analyzer proved very useful in analyzing CAN traffic.

16.1 Using mssql database on linux

Turns out it is possible to use ms-sqlserver under linux

https://learn.microsoft.com/en-us/sql/linux/quickstart-install-connect-ubuntu?view=sql-server-ver16&tabs=ubuntu2004

controlling the beast ....

```
systemctl status mssql-server
systemctl stop mssql-server
systemctl disable mssql-server
systemctl enable mssql-server
```

A nice GUI would be cool!!

https://dbeaver.io sudo add-apt-repository ppa:serge-rider/dbeaver-ce sudo apt-get update sudo apt-get install dbeaver-ce

And now .. a way to get Volvo vida files accessible under linux. This allows for exporting data and having access to stored procedures. (the tools on windows seemed rather limiting...)

```
/opt/mssql-tools/bin/sqlcmd -S . -U sa -P GunnarS3g3 -Q "CREATE DATABASE BaseData_Data␣
↪ON (FILENAME = '/var/opt/mssql/data/BaseData_Data.MDF'),(FILENAME = '/var/opt/mssql/
↪data/BaseData_Data_log.ldf' ) FOR ATTACH_REBUILD_LOG"

/opt/mssql-tools/bin/sqlcmd -S . -U sa -P GunnarS3g3 -Q "CREATE DATABASE CARCOM ON␣
↪(FILENAME = '/twee/volvo/CarComRT_Data.MDF'),(FILENAME = '/twee/volvo/CarcomRT_Log.LDF
↪' ) FOR ATTACH_REBUILD_LOG"

/opt/mssql-tools/bin/sqlcmd -S . -U sa -P GunnarS3g3 -Q "CREATE DATABASE␣
↪DIAGSWDLREPOSITORY ON (FILENAME = '/twee/volvo/DiagSwdlRepository_Data.MDF'),(FILENAME␣
↪= '/twee/volvo/DiagSwdlRepository_log.LDF ' ) FOR ATTACH"

/opt/mssql-tools/bin/sqlcmd -S . -U sa -P GunnarS3g3 -Q "CREATE DATABASE DIAGSWDLSESSION␣
↪ON (FILENAME = '/twee/volvo/DiagSwdlSession_Data.MDF'),(FILENAME = '/twee/volvo/
↪DiagSwdlSession_log.LDF' ) FOR ATTACH"
```

In Dbeaver the stored procedure can be executed, by right clicking. It then opens a new windows, where you can add the parameter.

To execute a stored procedure in SQL Server that requires a parameter, you can use the EXEC statement followed by the name of the stored procedure and provide the parameter value within parentheses. Here's how you can pass a parameter to the stored procedure [dbo].[vadis_GetEcuVariantData]:

```
USE [CARCOM]
DECLARE @return_value int
DECLARE @parameter_name int = 821
EXEC @return_value = [dbo].[vadis_GetEcuVariantData]@parameter_name

SELECT       'Return Value' = @return_value
```

it return DiagNumber : 30728270 AA

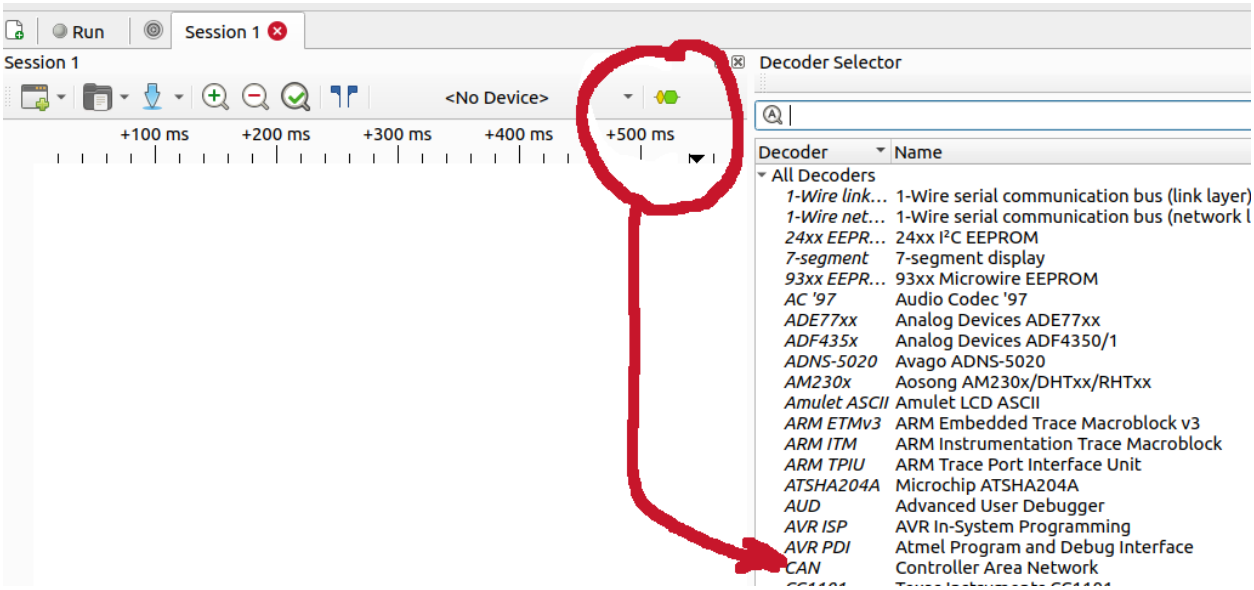
DiagNumber	EcuTypeI	EcuI	Add	Ecu	Nam	CfgId	Gwy	ProtocolId
30728270 AA	372,302	CEM	52	821	CEM	611		7

16.2 Logic Analyzer

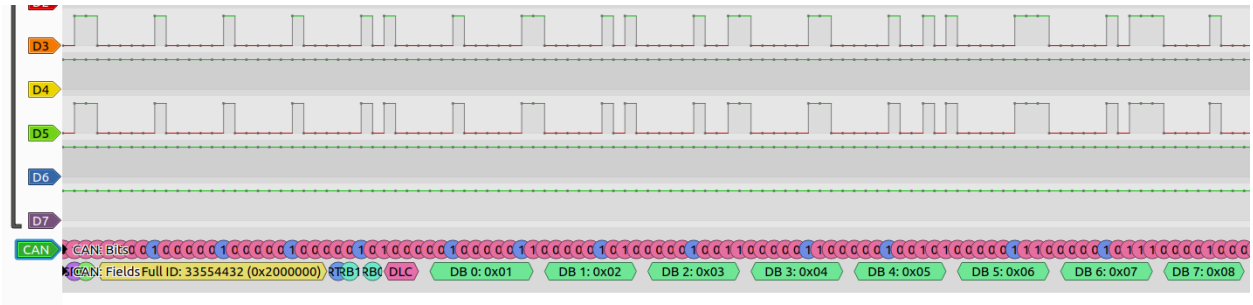
https://www.saleae.com/

There is a way to sniff Canbus traffic using a logic analyzer.

On linux I installed the PulseView software.



important remark : specify the canbus-speed : 125000 or 500000



16.3 virtual box on linux

The software of the VIDA diagnostic tool is installed on windows 7.

The software comes as an oracle vbox and is a windows 7 image.

installing this under ubuntu 22 proved challenging

16.3.1 the problem is USB 3

- install the VirtualBox Extension Pack 7.0.12 (files/tools/extension pack manager)
- sudo usermod -a -G vboxusers $USER

-Win7: you need to install the "Intel USB 3.0 eXtensible Host Controller Driver" in your Windows 7 guest: Update 2019-11-20: Intel seems to have removed the driver from its official site, you can get it from the Archive.org site: http://web.archive.org/web/201901092235 … Driver.zip

- now the SETEK dice adapter should be visible

16.4 SavvyCAN

https://github.com/collin80/SavvyCAN

I used a prebuild version, since compiling QT stuff is not my cup.

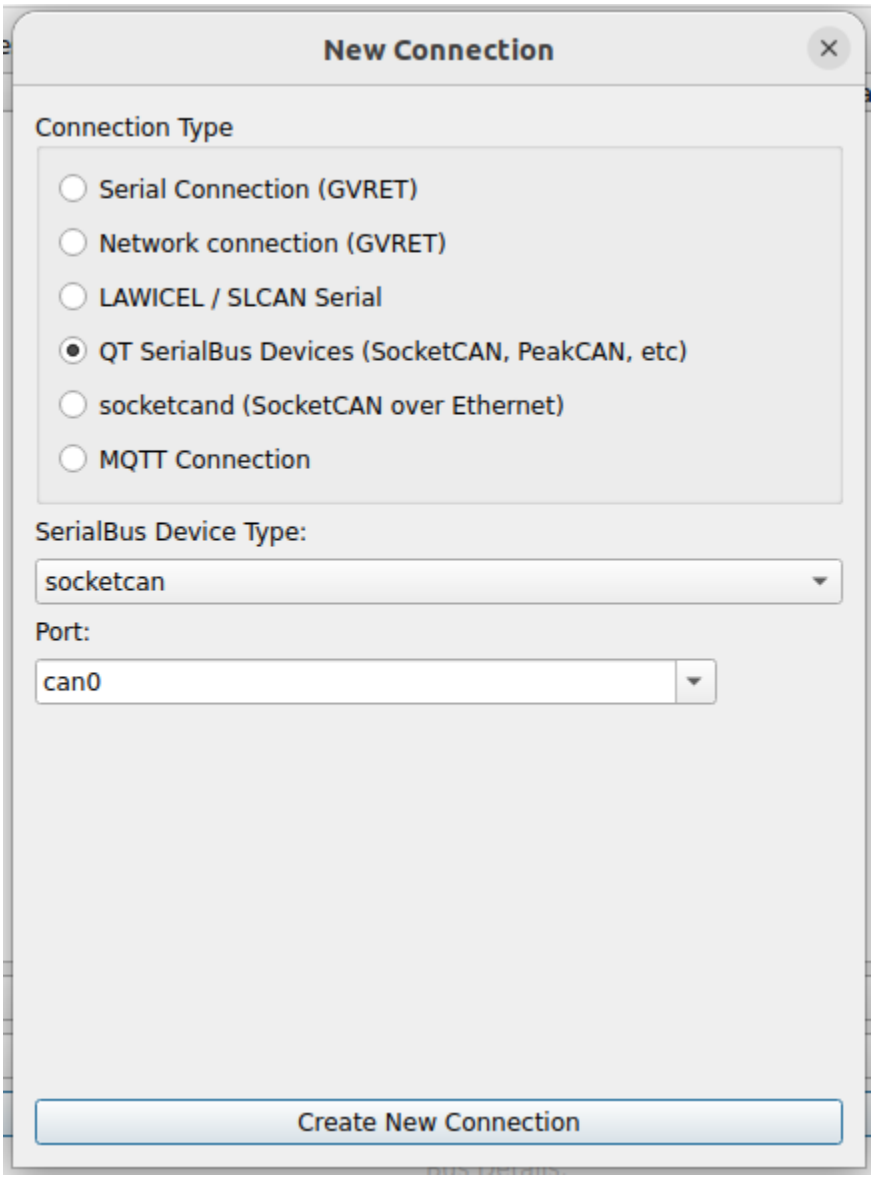
SavvyCAN-x86_64.AppImage

16.4.1 using socketcan

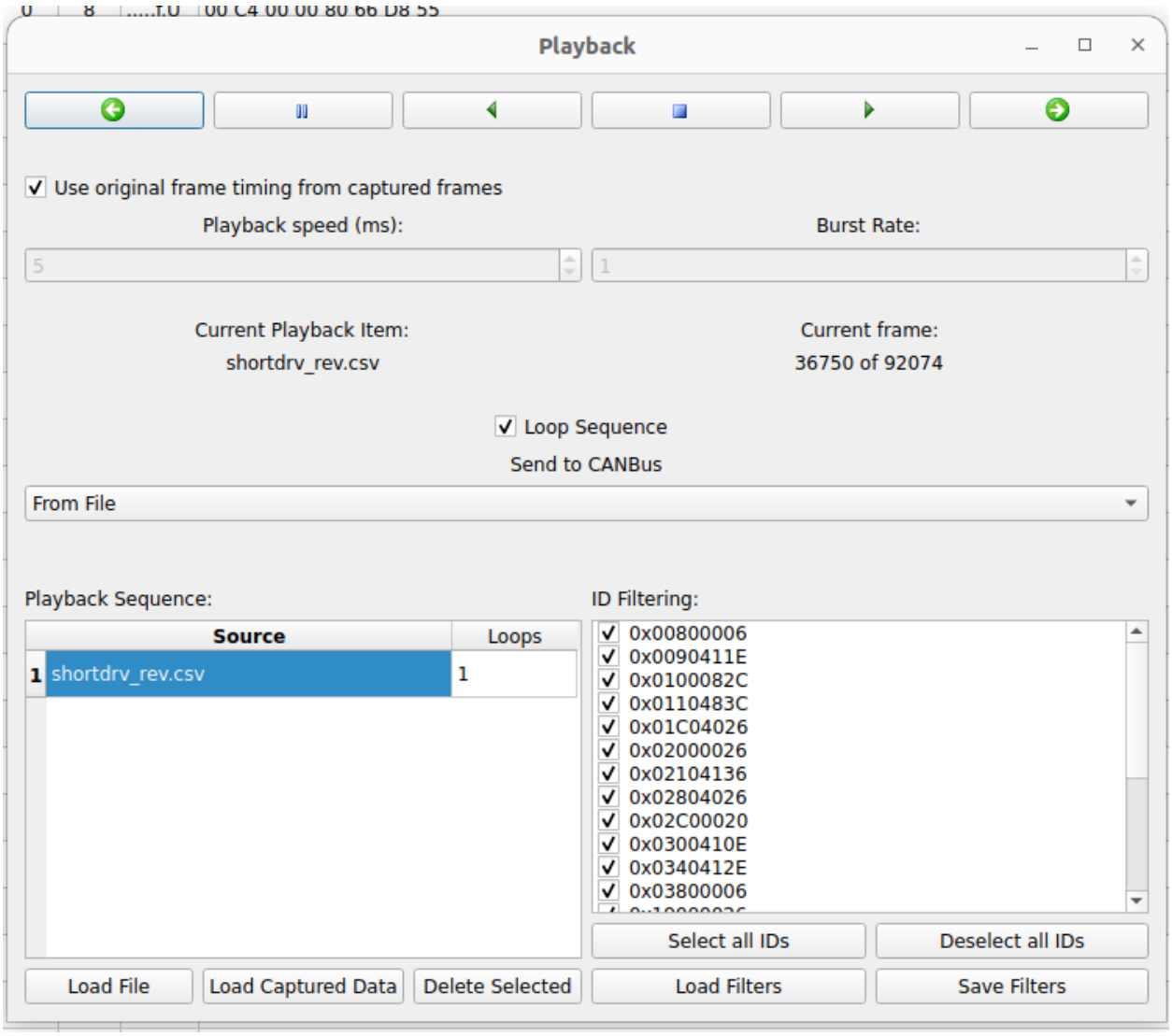
recall :

```
modprobe can
modprobe can-raw
modprobe slcan
slcand -s6 -S2000000 /dev/ttyACM1 can0
ifconfig can0 up
```

16.4.2 howto configure savvycan for use with socketcan



16.4.3 playback over socketcan



CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search