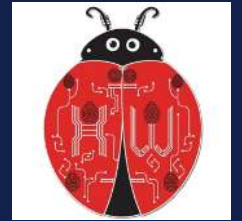


Connected Car Hacking training

02. Building a test bench

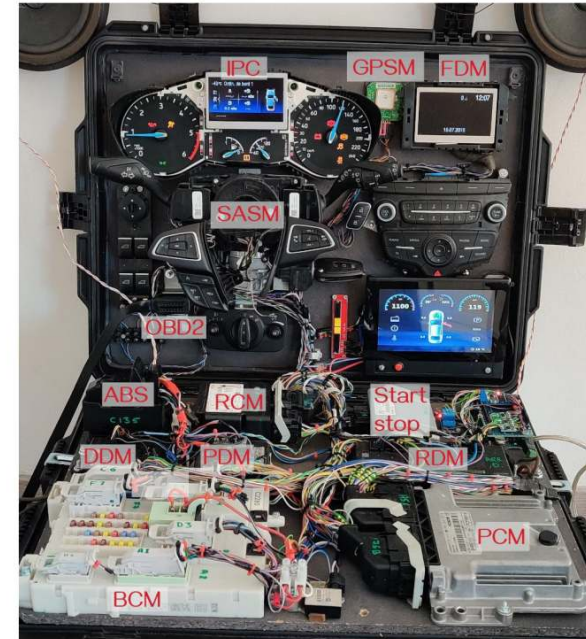
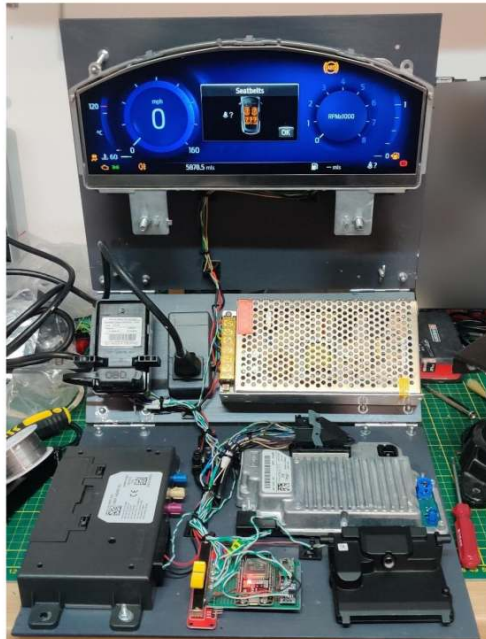
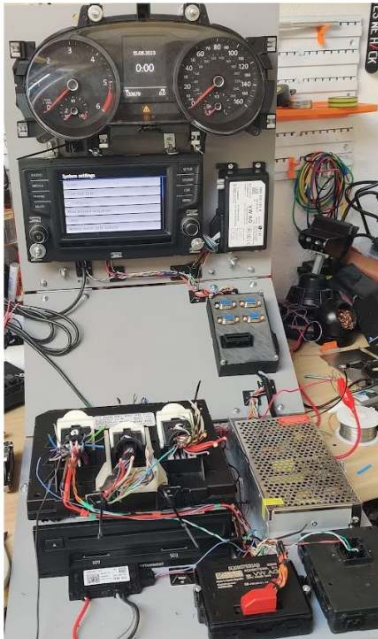
Hardware.io USA 2026



Quarkslab

Usage of an automotive test-bench

- ▶ Have a functional set of **ECUs**
- ▶ No risk of **breaking your car**
- ▶ Control of the various **on-board network/bus** and **I/Os**



Which car to work on?



If you have a choice, choose a model **you could have access to**.

Beware of models **too old** (missing useful functionalities/protocols) or **too recent** (ECU price, communication encryption, ...).

Choosing a **common model** allows to easily get access to spare ECUs.

Illustration: [\[link\]](#)



Which ECU to focus on



- ▶ **Instrument Cluster:** provides a lot of outputs to understand what's happening on the bench, good starting point
- ▶ **Infotainment:** a lot of connectivity (Wi-Fi, Bluetooth, sometime 3GPP), verbose logs
- ▶ **Telematic:** external connectivity using 3GPP modem, most critical attack surface, but may require infotainment unit to have active traffic to backend
- ▶ **Body Control Module:** core of the car, handle most of the time the wake-up frames, locking systems...
- ▶ **Gateway:** critical security component, could also handle wake-up frames
- ▶ **Other ECUs:** depending on your needs and topics of interest (RF/RFID with RKE/PKE, Airbag crash data logger, V2G...)

Where to find target ECUs



- ▶ Scrapyard
 - + Could check/test on-site if the ECU is working
 - + Guarantee to have access to the connectors
 - Could be difficult to find a specific model
 - Dismantling the car by yourself requires adequate tools and some time
- ▶ Internet
 - + Large range of ECUs available
 - Sometime you may receive a broken ECU...
 - Connectors are often missing

We do recommend finding **websites that list other parts for the same vehicle**, to avoid any fitment issues.

Moreover, spending a few more bucks on an ECU with **its connector** will save you time.



Recommended bench tools



- ▶ **Lab voltage regulator:** you may need to power over 12V some ECU (approx. 14V when a vehicle is running), w/ peak current of 6/8A for some units
- ▶ **Wago connectors:** easily interconnect bus, power lines...
- ▶ **Dupont headers:** useful when you don't have the ECU's connector
- ▶ **120 ohms resistors:** complete missing termination on CAN bus



Training test-bench

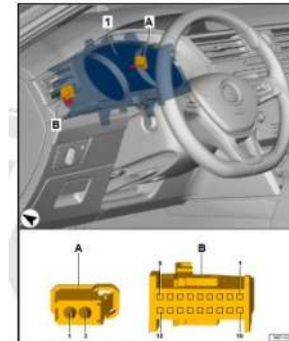
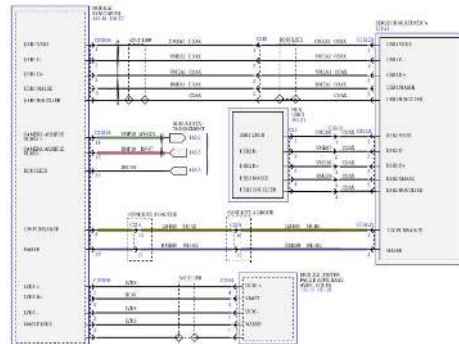


Six ECUs will be used to create a test-bench for this training:

- ▶ **Instrument Cluster (IC)**: base ECU to get a visual display of the state of the bench
- ▶ **Body Control Module (BCM)**: manages doors, lights, controls and ignition state
- ▶ **Gateway (GW)**: filters and routes CAN messages between the different CAN buses
- ▶ **In-Vehicle Infotainment unit (IVI)**: multimedia unit which can be paired with smartphone for sending/receiving calls or broadcasting music
- ▶ **Telematic Control Unit (TCU)**: handles Internet connectivity and emergency calls
- ▶ **Fleet Management Unit (FMU)**: used for fleet sharing application

Getting technical information

- ▶ Look for technical documentation, to **identify ECU** (type, location), **available networks** and how the OBD-II port is wired.
- ▶ Mechanic forums are good, but you'll get only a few schematics
- ▶ Every manufacturer has a website dedicated to **mechanic workshop**, but access needs subscription (could be hourly access). You will find exact **diagram**, ECU **pinout**, and sometimes you can download **firmware updates**
- ▶ Try to find CAN database (.dbc), like on comma.ai's GitHub: <https://github.com/commaai/opendbc>



List of manufacturers' technical data

Audi/VW	https://erwin.audi.com/ & https://erwin.vw.com/	Mercedes	http://www.startekinfo.com/
BMW	http://www.bmwtechinfo.com/	MG	https://rmi.mgmotor.eu
BYD	https://eurmi.byd.com	Mini	http://www.minitechinfo.com/
Citroen	http://public.servicebox.peugeot.com/pages/index.jsp	Mitsubishi	http://www.mitsubishitechinfo.com/
Fiat	https://www.technicalinformation.fiat.com/	Nissan	https://www.nissan-techinfo.com/home.aspx
Ford	https://www.etis.ford.com/	Peugeot	http://public.servicebox.peugeot.com/
Honda	https://techinfo.honda.com/rjanisis/logon.aspx	Porsche	https://techinfo2.porsche.com/
Hyundai	http://www.hyundaitechinfo.com/	Renault	https://newdialogys.renault.com/
Jeep	http://www.techauthority.com/	Saab	http://epsiportal.com/Site/SAAB
Kia	https://kiatechinfo.snapon.com/default.aspx	Smart	http://www.smarttekinfo.com/SmartTek/



List of manufacturers' technical data

Land Rover	http://www.landrovertechinfo.com/	Tesla	https://service.teslamotors.com/
Lexus	https://techinfo.lexus.com/	Toyota	http://techinfo.toyota.com/
Mazda	https://www.mazdaserviceinfo.com/	Volvo	http://www.volvotechinfo.com/

Lab 1 - reading technical documentation



Lab 1 - Reading technical documentation



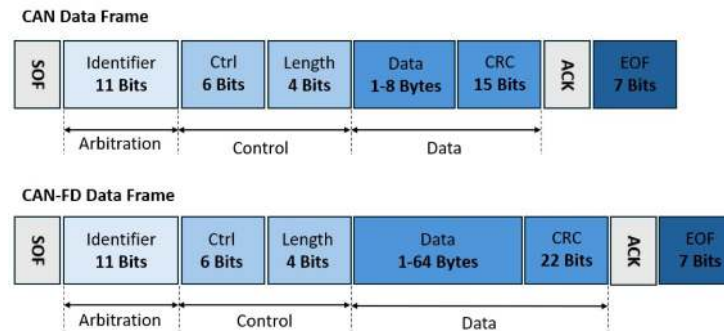
For this lab, you'll study wiring documentation, coming from or inspired by different manufacturers technical data

- ▶ Complete challenges **Test bench - 01. Documentation**

CAN bus: key concepts

- ▶ **Broadcasted** messages, a CAN message has an **Arbitration ID** and a **payload**
- ▶ Arbitration ID can be **standard** (11 bits: 0x000 to 0x7FF) or **extended** (29 bits: up to 0x1FFFFFFF)
- ▶ The Arbitration ID is used for **anti-collision**, the **lower the ID the higher the priority**
- ▶ Payload of **8 bytes** for CAN, **64 bytes** for CAN-FD and **2048 bytes** for CAN-XL

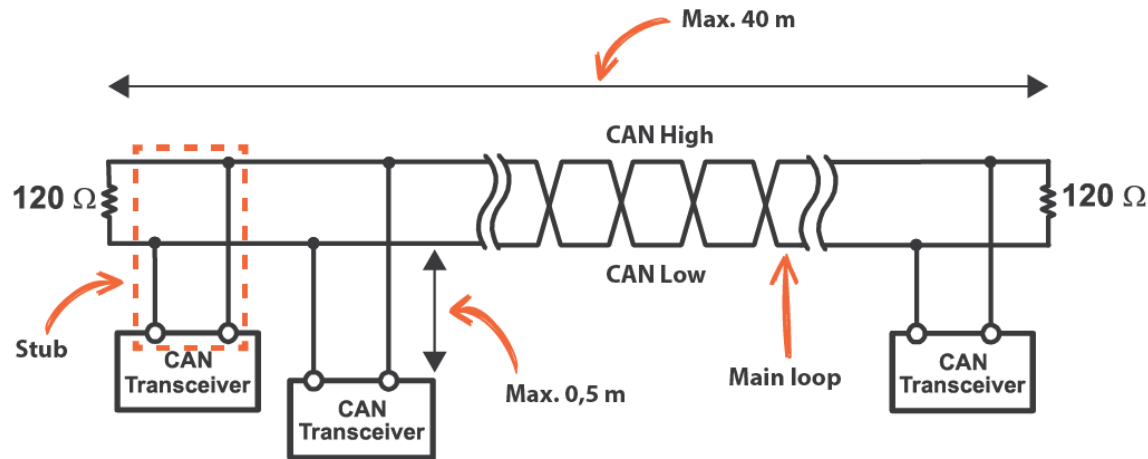
Illustration: [\[link\]](#)



CAN bus wiring

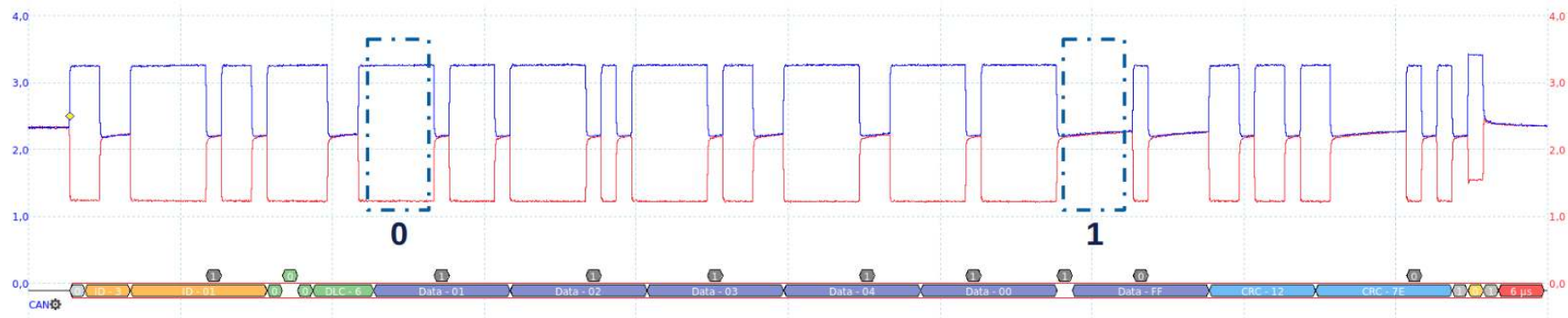


- ▶ **Two wires:** twisted pair with a **CAN High** and **CAN Low** wire
- ▶ The bus is terminated by a **120 ohms resistor** to prevent signal reflection



CAN bus signaling

- ▶ **Differential signaling:** the voltage difference on each wire defines the signal sent (fault resistant)
- ▶ **Dominant state (0):** CAN High is at $\sim 3.5\text{V}$, CAN Low is at $\sim 1.5\text{V}$
- ▶ **Recessive state (1):** CAN High drops to $\sim 2.5\text{V}$, CAN Low level increases to $\sim 2.5\text{V}$



Type of CAN frames



- ▶ **Data frame:** sent by a transmitter node to all other nodes
- ▶ **Error frame:** sent by any node detecting an error
- ▶ **Remote frame:** sent by a node to request the transmission of a data frame with the same identifier
- ▶ **Overload frame:** flow control, injects an extra delay after a data or remote frame

Anatomy of a CAN frame

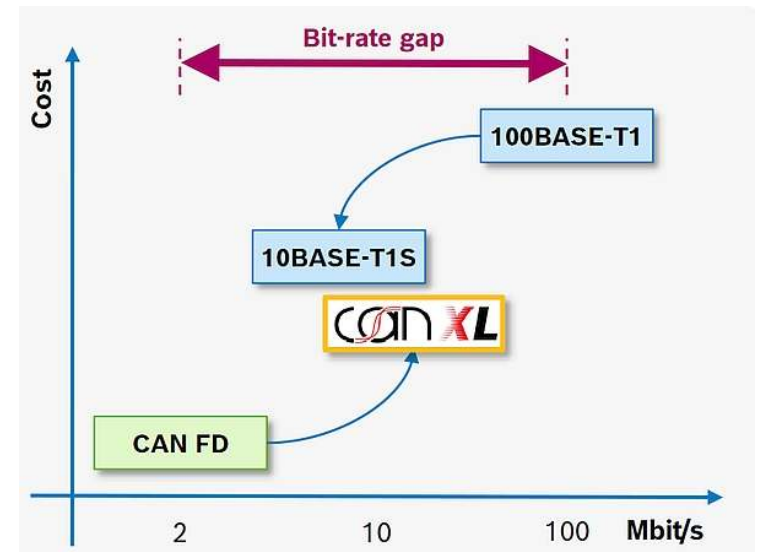


- ▶ **Arbitration ID:** from 0x000 to 0x7FF (11 bits) in standard mode, up to 0x1FFFFFFF (29 bits) in extended mode
- ▶ **RTR:** defines if it's a data frame or remote frame
- ▶ **IDE:** defines the arbitration ID mode (standard/extended)
- ▶ **r0:** recessive (1) for CAN-FD frames
- ▶ **Data:** 8 bytes, up to 64 bytes in CAN-FD
- ▶ **CRC/ACK:** used for error detection
- ▶ **Bit stuffing:** if 5 same bits are consecutive, an opposite bit is added to the frame

CAN technologies and speed



- ▶ **CAN:** Automotive and industrial protocol developed in 1986 by Bosch
 - Speed of 10, 20, 62.5, 125, 250, **500**, 800 and 1 000 Kb/s
 - 8 bytes of payload
- ▶ **CAN-FD:** Flexible Data-Rate introduced in 2012
 - Speed up to 8 Mb/s
 - 64 bytes of payload
 - Arbitration phase limited to 1Mb/s to be backward compatible with classic CAN
- ▶ **CAN-XL:**
 - Speed up to 20 Mb/s
 - 2048 bytes of payload
 - Can tunnel Ethernet frames



Setting up the CAN adapter

- ▶ To interact with the CAN bus, you'll need to set your adapter
- ▶ To find the interface name, use command `ip a`

```
$ ip a
[...]
3: can0: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN group default qlen 1000
    link/can
4: can1: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN group default qlen 1000
    link/can
```



Setting up the CAN adapter (1/2)

- ▶ To activate a CAN interface, the desired speed must be set first (here 500Kbps)

```
$ sudo ip link set can0 up type can bitrate 500000 dbitrate 500000 fd on  
# To support classic CAN (no-fd)  
$ sudo ip link set can0 up type can bitrate 500000
```

- ▶ If you need to change the speed of the interface, you'll need to bring it down first

```
$ sudo ip link set can0 down
```

- ▶ You can also rename the interface, to match the CAN bus label

```
$ sudo ip link set can0 name CAN-HS
```



Setting up the CAN adapter (2/2)

- ▶ An important step is to set a bigger buffer when sending a large amount of CAN frames

```
$ sudo ip link set can0 txqueuelen 1000
```

- ▶ If everything is correctly set, you must get the following output:

```
$ sudo ip a | grep can
3: can0: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UP group default qlen 1000
    link/can
[...]
```



Reading a CAN bus: candump

- ▶ The can-utils library has many tools to work with CAN bus
- ▶ **candump** displays every message going through the bus

```
$ candump -x -a can0  
  
can0 TX - - 2e0 [3] 02 06 64      '..d'  
can0 TX - - 130 [3] 02 06 64      '..d'  
can0 RX - - 121 [2] 01 46         '.F'  
can0 RX - - 124 [2] 01 46         '.F'  
can0 TX - - 128 [8] 07 33 68 65 65 6C 70 6F '.3heelpo'
```



Candump - filters

- ▶ Filters can be applied to only display specific arbitration ID or a range of ID using masks
- ▶ For each filter, add “,” followed by the desired arbitration ID then “:” and the desired mask
- ▶ A binary comparison is made with the mask, only ID matching mask “1” bit are displayed

```
$ candump -x -a can0,123:7FF,030:7F0
```

```
can0 TX - - 123 [3] 02 06 64      '..d'  
can0 TX - - 031 [3] 02 06 64      '..d'  
can0 RX - - 036 [2] 01 46         '.F'  
can0 RX - - 123 [2] 01 46         '.F'  
can0 TX - - 03F [8] 07 33 68 65 65 6C 70 6F '.3heelpo'
```

Write on a CAN bus: cansend

- ▶ **cansend** is the most basic tool to send data over the CAN bus

```
$ cansend can0 321#C0FFEE
```

- ▶ An empty frame can also be sent

```
$ cansend can0 321#
```

- ▶ To send a CAN-FD frame, use 2 “#” followed by the **BRS/ESI/RES** flags on 1 nibble
- ▶ **BRS** is 1 if there is a **bitrate switch** between the Arbitration ID and the data, 0 otherwise You can add a period after the flag to help reading the command

```
$ cansend can0 321##0.C0FFEE
```

Generating CAN messages

- ▶ **cangen** allows to generate CAN messages at a specific interval
 - ▶ -l: **Arbitration ID** of the message
 - ▶ -L: **Length** of the message
 - ▶ -D: **Data** to send
 - ▶ -g: **gap** in millisecond between two messages
- ▶ It is useful to send **arbitrary data** on a CAN bus or send **periodically** a specific CAN frame
- ▶ It supports two magic values, **i** for incremental and **r** for random

```
$ cangen can0 -I 321 -L 3 -D COFFEE -g 100
```

```
$ cangen can0 -I i -L 8 -D r -g 100
```



Fuzzing a CAN bus in a real car

Beware when fuzzing a CAN bus in a real car

When a **crash occurs**, the Airbag sends a **crash frame** to the Engine Control Module (ECM) and/or the Body Control Module, to:

- ▶ unlock the doors
- ▶ stop the injection pump
- ▶ activate the warning signal
- ▶ trigger the emergency call



Sending random data in a real car could trigger the crash status, and may brick the car

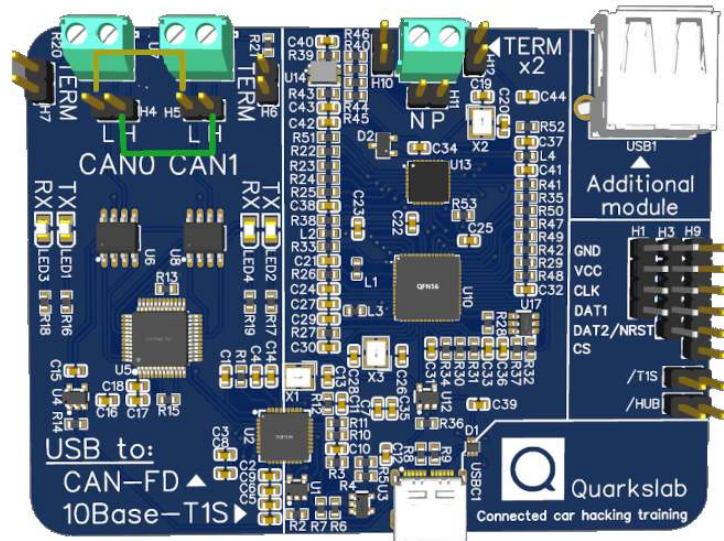
Avoid fuzzing Arbitration ID below 0x300 in a real vehicle!

🧪 Lab 2 - CAN 101



Lab 2 - Sending and receiving CAN messages

- ▶ Plug your CAN adapter with an USB-C cable to your computer.
- ▶ Connect respectively pins L to L and H to H for **CAN0** and **CAN1**



Lab 2 - Sending and receiving CAN messages

- ▶ Configure both interfaces

```
$ sudo ip link set can0 up type can bitrate 500000 dbitrade 500000 fd on
$ sudo ip link set can0 txqueuelen 1000
$ sudo ip link set can1 up type can bitrate 500000 dbitrade 500000 fd on
$ sudo ip link set can1 txqueuelen 1000
```

- ▶ Complete challenges **Test bench - CAN 101**



1st ECU: Instrument Cluster

Instrument Cluster



- ▶ The **Instrument Cluster** will be the first ECU we will work with
- ▶ Having a display, it is useful to get an idea of its **state** and test payload of various CAN frames
- ▶ We will power it using the **Super Multitool 2000** current sensor pins



Polarity warning

Before wiring the Instrument Cluster, perform all the required challenges to get a visual confirmation of the +5V and GND pins, to avoid any short circuit that could damage the device



Lab 3 - Pin identification



Lab 3 - Identifying pin of the Instrument Cluster

- ▶ Complete challenges “**Reference**” to “**Powering**” in the **Test bench - Instrument Cluster** category
- ▶ Pay attention on the **VOUT** and **GND** pins when connecting to the **Super Multitool 2000**



Wake-up frames

- ▶ ECUs are designed to be **power saving**, to avoid draining the car's battery
- ▶ You may need to find a **wake-up frame**
 - ▶ could be **any activity** on the CAN bus
 - ▶ or a **specific CAN frame**
- ▶ Some ECUs require a **12V input when ignition is on**, look for pin “wake signal”, “ign/ignition” or “terminal 15/KL15”
- ▶ Wake-up frames are usually sent by the **Body Control Module**, their content varying depending on the **ignition state**

Finding the wake-up frames

- ▶ **Capture** the CAN traffic of a working car and **replay it**



Beware

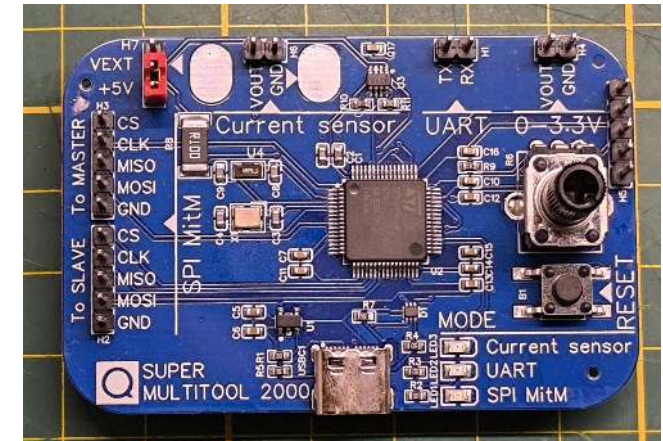
Sending data from non-enrolled ECUs may trigger anti-theft protection

- ▶ Try every Arbitration ID **incrementally**, sending **8 bytes** long message starting from “00 00 00 00 00 00 00 00” to “FF FF FF FF FF FF FF FF” and monitor the **current consumption** of your lab generator
- ▶ Having a lab generator with Ethernet/USB connectivity is a plus to automatise this search
- ▶ Otherwise, you can use a microcontroller and a current sensor to do so, like the **Super Multitool 2000**

The Super Multitool 2000



- ▶ To monitor the power consumption of an ECU, you can use the Super **Multitool 2000**
- ▶ It's based on a **STM32F4** and an **INA219** current sensor
- ▶ You can send commands to the multitool over its serial interface
- ▶ Jumper allows choosing the **power source: VEXT** when using a lab power generator, or **+5V** from the USB port.
- ▶ The device under test needs to be powered using **VOUT** and **GND** pins



Super Multitool 2000 - current sensor commands



- ▶ Sending “**m**” (measure) through the UART over USB to compute the average current consumption of the device under test
- ▶ It samples 40 times the current consumption with an interval of 10ms
- ▶ Once the measurement is done, a threshold is computed
- ▶ To arm the current trigger, send “**a**” (arm). The **Super Multitool 2000** will continuously read the current on the **INA219** and send character “!” if the threshold is reached
- ▶ This mode can be cancelled by sending “**e**” (exit) command

The Super Multitool 2000 - current sensor commands

```
user@user: ~$ screen /dev/ttyACM0 h Supported commands:
```

```
- [h]elp
```

```
- [e]xit current mode
```

```
==== Current sensor ====
```

```
- [m]easure
```

```
- [a]rm
```

```
[... redacted for readability ...]
```

```
m
```

```
Average current: 6 mA, Max current: 6 mA, Threshold: 9 mA
```

```
a
```

```
Current detection armed, threshold: 9 mA
```

```
!Current increase detected: 10 mA (Threshold: 9 mA)
```



⚗ Lab 4 - Finding the wake-up frame



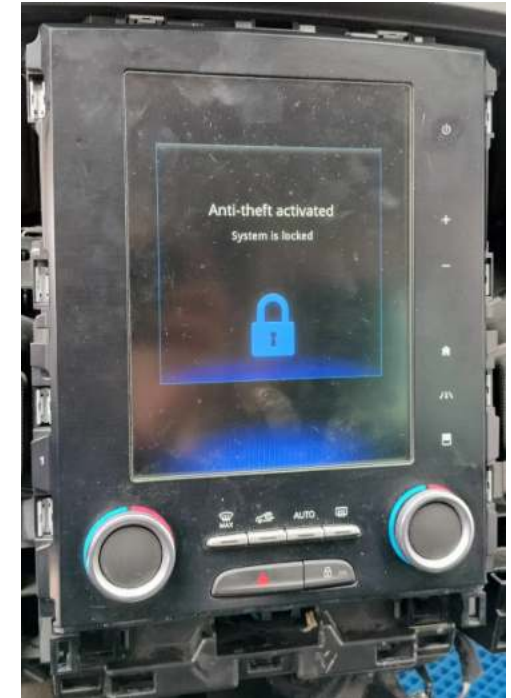
Lab 4 - Finding the wake-up frame

- ▶ Using the **Super Multitool 2000**, you'll use the current sensor to find the wake-up frame
- ▶ Complete challenges **“Wake-up frame *”** in Test bench - Instrument Cluster
- ▶ **Bonus**: find the correct byte position and its value to wake-up the Instrument Cluster

Anti-theft protection



- ▶ Some ECUs implement an **anti-theft** detection
- ▶ When active, the ECU could **work partially** or be **disabled**
- ▶ It could be triggered by an **invalid message** or a **missing CAN message**
- ▶ It is usually based on a shared **“secret”** with one or multiple ECUs being part of the enrollment
- ▶ On older **IVI** it uses a **PIN** code



🧪 Lab 5 - Bypassing the anti-theft protection 🔍

Lab 5 - Bypassing the anti-theft protection

- ▶ For this lab, the goal is to bypass the anti-theft protection, which is based on a real usage we have faced
- ▶ To do so, you'll first read the **Vehicle Identification Number** (VIN) of the vehicle
- ▶ Then analyse a **CAN bus capture** previously performed on a working vehicle using the same **Instrument Cluster**
- ▶ Complete the challenges **VIN** and **Bypassing anti-theft protection** from **Test bench - Instrument Cluster**
- ▶ **Bonus:** when reading the VIN, try to understand the **request** sent to the ECU and the **response** received

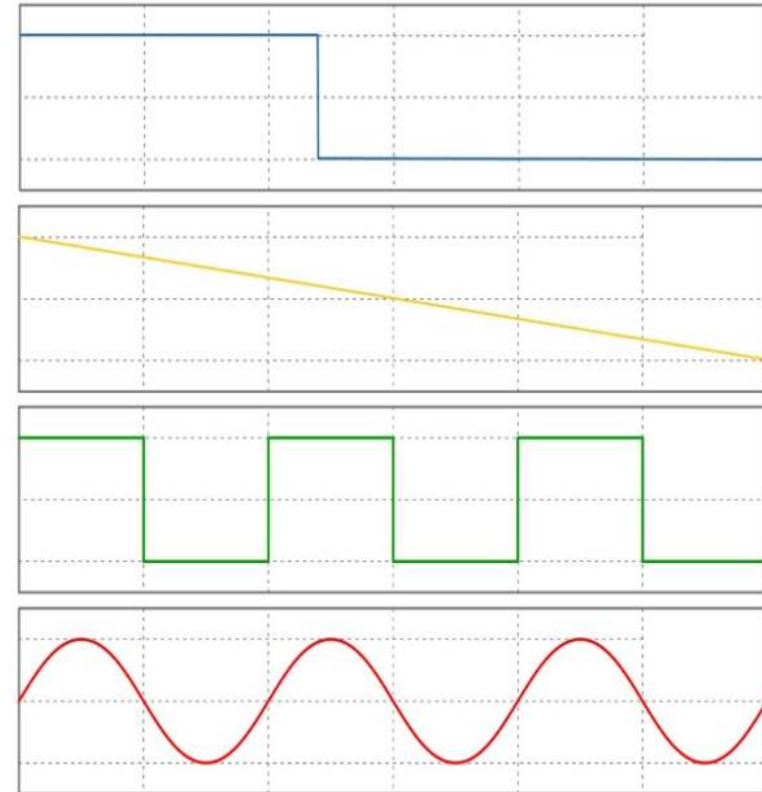


Emulating sensors

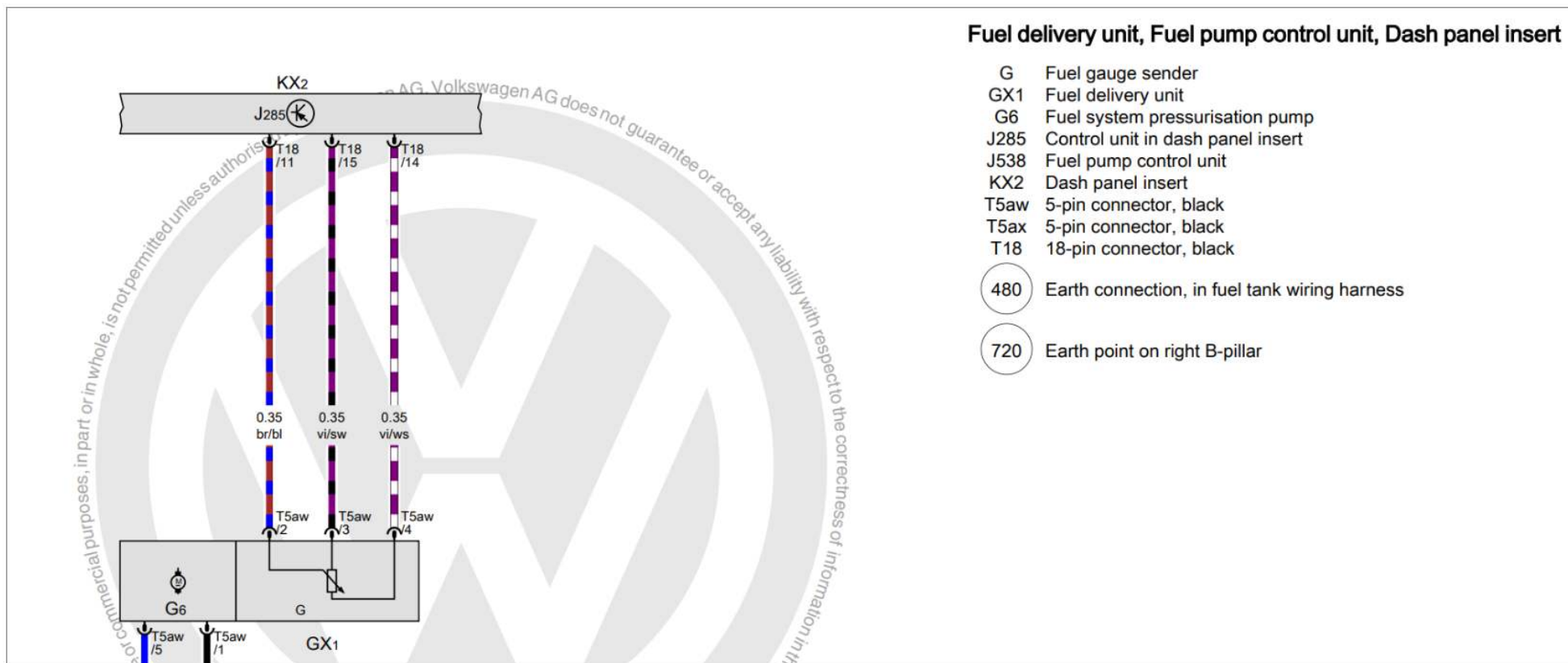


Sensors have 4 types of output:

- ▶ Relay to ground (or +12V)
Input read 1 if connected to GND (or +12V)
- ▶ Variable resistance
Mostly 5V signal going through 1-5K potentiometer
- ▶ Square signal
Also 5V signal
- ▶ Wave signal
More complex to emulate, need additional hardware



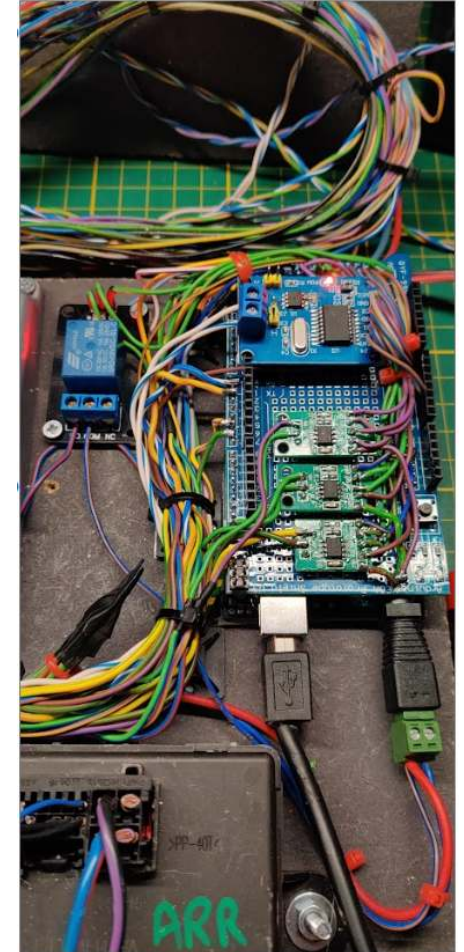
Instrument Cluster sensor diagram



Example of sensors emulation



- ▶ Switching pins mode (input/output) is used to have **signal to ground** control
- ▶ **12V output** is handled by a relay
- ▶ **Digital potentiometer** emulates temperature and fuel level sensors
- ▶ An **Arduino** board adapts values through its **UART** interface



Not all sensors are wired



- ▶ Direct **Tire Pressure Monitoring System (TPMS)** use radio-frequency
- ▶ Each sensor **sends periodically** data to the BCM
- ▶ Signals are crafted with the **Rfcat** library and supported devices based on a **CC1101 transceiver**



Diagnostic tools



- ▶ Having access to official or 3rd party **diagnostic tools** is a time saver for these tasks
- ▶ It allows to **request read value** from the ECU and evaluate **valid inputs**
- ▶ Some tools allow to modify **ECU configuration**, using under the hood UDS Service Write Data By Identifier, which can be useful to revert crash mode for example



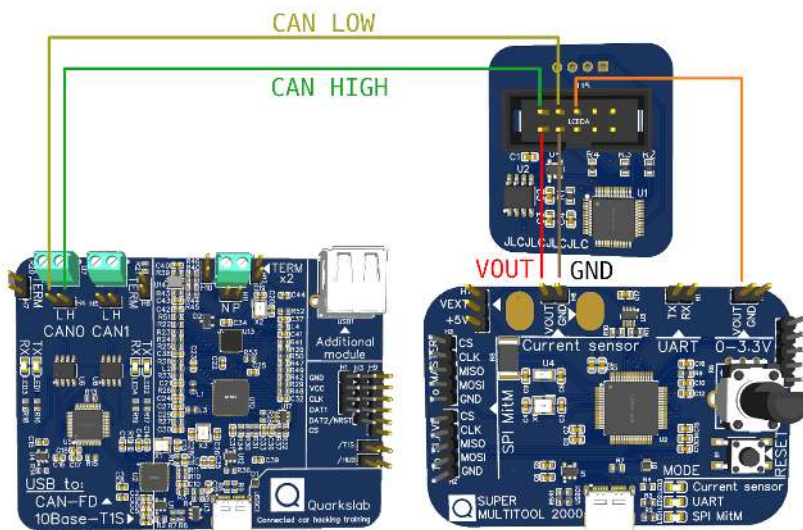
Illustration: [\[link\]](#)

⚗ Lab 6 - Sensor emulation

Lab 6 - Sensor emulation

Using the **Super Multitool 2000**, you will emulate a basic sensor on the pin 6, tied to the **Lux reader**

- ▶ Connect the **VOUT** of **0-3.3V** as shown:



⚗ Lab 6 - Sensor emulation



Lab 6 - Sensor emulation

- ▶ Complete the challenge **Probe sensor value** in **Test bench - Instrument Cluster**

Body Control Module

Body Control Module



- ▶ Next ECU will be the **Body Control Module**, which will handle the wake-up frames and provide some inputs/outputs
- ▶ It will be powered through its **USB-C plug**
- ▶ On the **Body Control Module** PCB, there is also a **Gateway** and an **OBD-II** plug for diagnosis purposes (not used in this training)
- ▶ The PCB provides pins to connect to the **two CAN buses (BODY and INFO)** and the **10BASE-T1S** network

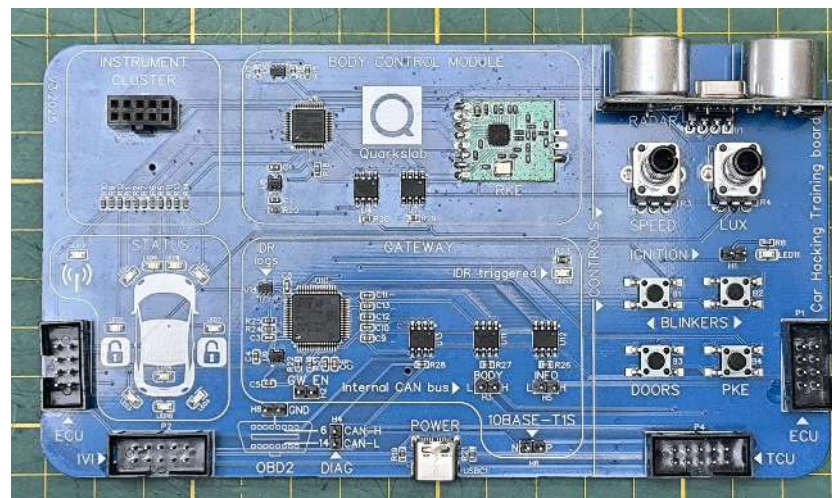


Enhancing our test bench

- ▶ Like we did with the **Instrument Cluster**, first task will be to **gather technical documentation** to identify primary pins, including
 - ▶ +12V input
 - ▶ Ground
 - ▶ CAN and Automotive Ethernet
 - ▶ Ignition output (if present)
- ▶ Connect the **Instrument Cluster** onto the **Body Control Module**
- ▶ For this ECU, we will focus on identifying **CAN payload** and how to interact with the **UDS diagnosis protocol**

Training Body Control Module

- ▶ Also integrates the **Gateway** ECU
- ▶ An OBD-II port wired to the **Gateway** and a tap to **CAN-BODY** and **CAN-INFO** CAN buses
- ▶ Dedicated connector for the **Instrument Cluster**, two 10-pin and two 8-pin connectors for additional ECUs
- ▶ A tap to the **10Base-T1S** network used by the 10-pin connectors
- ▶ Radar sensor for **ADAS**
- ▶ Two potentiometers to simulate the **accelerator pedal** and the **light sensor**
- ▶ 4 buttons for **doors**, **PKE** and **blinkers** management
- ▶ A **jumper** to simulate **ignition** status





Observing changes in CAN frames: cansniffer

- ▶ If **candump** is useful to confirm traffic on a CAN bus or to follow a specific arbitration ID, it is not the right tool to observe and understand useful payload
- ▶ **cansniffer** is an interesting option to quickly monitor CAN message and try to identify a specific payload (unlocking doors, ignition activation...)
- ▶ It displays all messages **sorted by arbitration ID** and highlight changing bytes with argument `-c`
- ▶ Filters can be applied real time, by typing `+123` or `-123` to remove arbitration ID `0x123` from the display
- ▶ Message with static payload are hidden after 1 second



Payload analysis limitation

cansniffer does not support CAN-FD payloads, it only process standard CAN frames

Examples

```
$ cansniffer -c can0
```

```
006|ms | ID | data ... < can0 # l=20 h=100 t=500 slots=8 >
00150 | 037 | 0B 54 00 00 00 00 A4 00 00 .T.....
00250 | 095 | 00 21 00 00 00 00 A4 .!....
00300 | 1E0 | AA 37 47 89 00 00 FD 62 .7G....b
99999 | 260 | 00 00 00 00 00 00 50 AF 00 .....P..
00300 | 333 | 24 00 0E 00 37 32 $....72
00677 | 430 | 00 00 13 ...
00150 | 444 | 00 00 00 00
```

```
$ cansniffer -c -b can0
```

```
69|ms | -- ID -- | data ... < can0 # l=20 h=100 t=500 slots=10 >  
00150 | ---- 037 | 000010110101010000000000000000000000000010100100000000000000  
00250 | ---- 095 | 000000000001000110000000000000000000000010100100  
00300 | ---- 333 | 1001000100000000000001110000000000011011110011111  
00677 | ---- 430 | 0000000000000000000010011  
00649 | ---- 600 | 00000100000000000000000111010000100000000  
06327 | ---- 728 | 000001010110001010000000011001010000000000010000101010101010
```

⚗ Lab 7 - Characterizing CAN messages



Lab 7 - Characterizing CAN messages

- ▶ CAN messages are sent periodically, look for the period of the anti-theft frames and complete **Frame Frequency** in **Test bench - Body Control Module**
- ▶ A CAN message can hold several information. Using cangen try to identify specific CAN message and understand its payload by completing **Wake-up byte** and **Blinkers**



- ▶ CAN bus is an **old protocol** and was not designed to be **secure**
- ▶ Any device having access to a CAN bus can send/receive messages
- ▶ **Replaying** message is trivial, there is **no native encryption** and no **authenticity check** associated to the **Arbitration ID**
- ▶ To overcome this issue, AutoSAR has released the **Secure On-board Communication (SecOC)** protocol

SecOC - key concept



- ▶ It provides **integrity** and **authentication** for messages (**I-PDU**), and **freshness** against replay attacks
- ▶ Before sending an I-PDU, the ECU computes an **Authenticator (MAC)**, using a **128 bit AES CMAC** algorithm on the concatenated value of the **SecOCDataId**, the **payload** and the **Freshness Value**
- ▶ **SecOCDataId** is the full **16 bit** representation of the **Arbitration ID**
- ▶ Truncated value of the **Authenticator** (MSB) and the **Freshness** (LSB) are added to the **I-PDU** to generate a **Secured I-PDU**

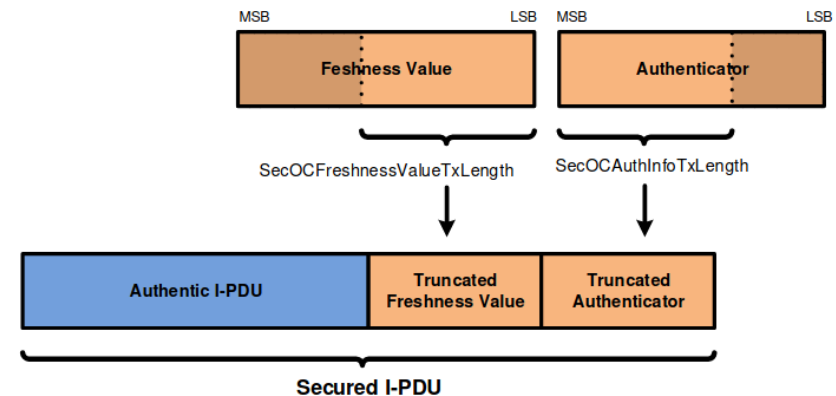
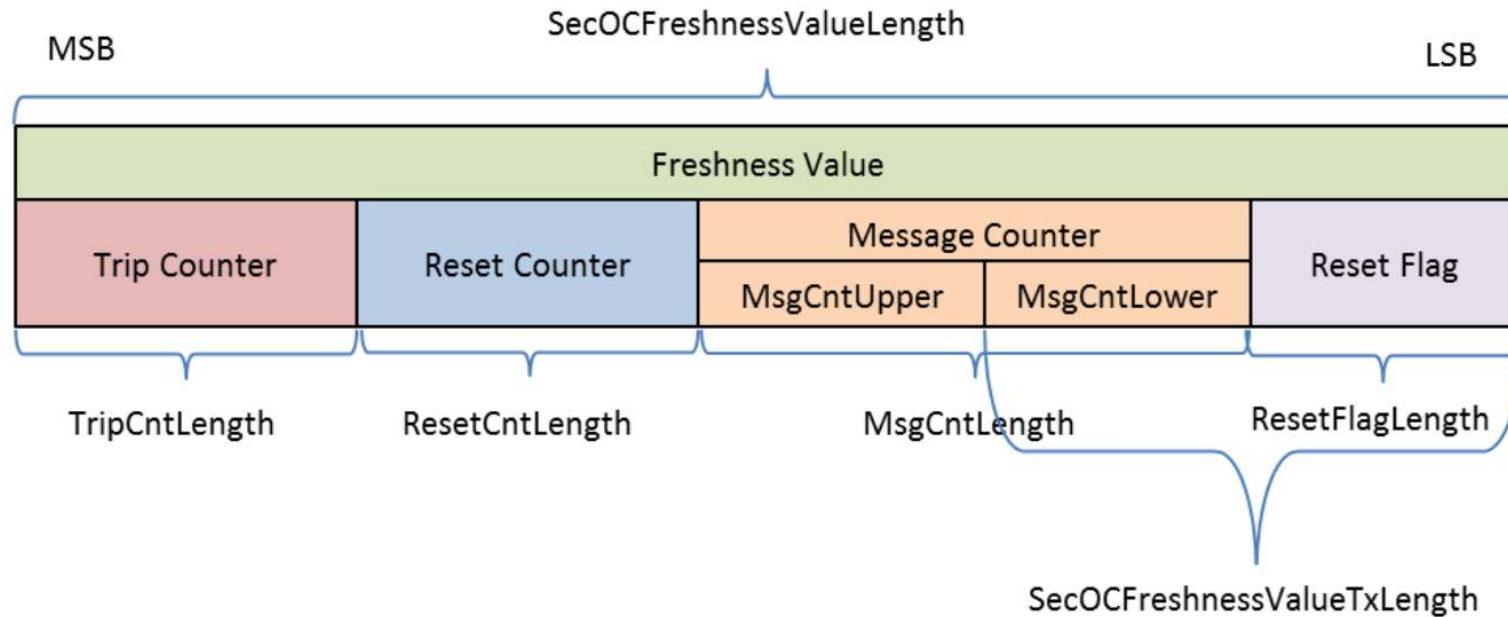


Illustration: [\[link\]](#)



- ▶ **Freshness Value** is a monotonic counter that needs to be synchronized between all the ECUs, to do so an ECU (like the gateway) acts as a **Freshness Manager**
- ▶ The **Freshness Value** is composed of several counters:
 - ▶ **TripCounter**: incremented at each ignition
 - ▶ **ResetCounter**: incremented at a fixed interval
 - ▶ **MessageCounter**: incremented each time a message is sent by a specific ECU
 - ▶ **ResetFlag**: contains lower bits of the ResetCounter to ensure truncated Freshness Value contains both MessageCounter and ResetFlag bits
- ▶ The **Freshness Manager** sends a synchronization message on each **TripCounter** and **ResetCounter** update. Synchronization messages are protected by a **MAC**

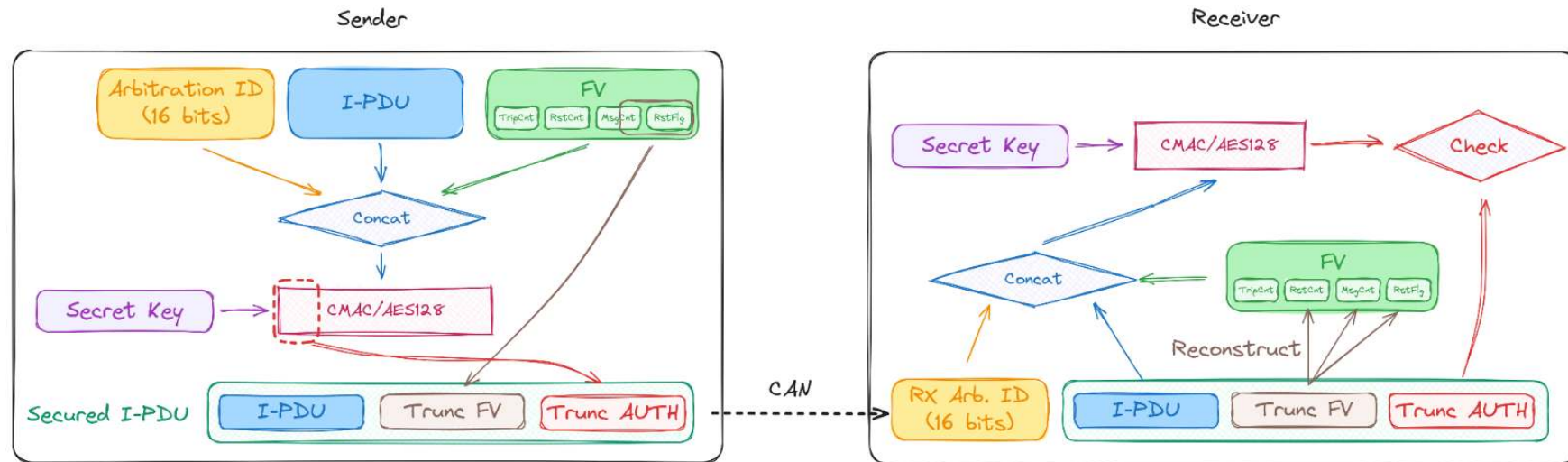
SecOC - freshness



SecOC - message verification



- As lower bits of the **Reset Counter** and the **Message Counter** are present in the Secured I-PDU, receiving ECU can recover the **Freshness Value** used to compute the **Authenticator**



- **AutoSAR** has defined several profiles for **SecOC**

Profile	Algorithm	Truncated FV	Truncated MAC
SecOC Profile 1 (<i>24Bit-CMAC-8Bit-FV</i>)	CMAC/AES-128	8bits	24 bits
SecOC Profile 2 (<i>24Bit-CMAC-No-FV</i>)	CMAC/AES-128	No FV	24 bits
SecOC Profile 3 (<i>JASPAR</i>)	CMAC/AES-128	4 bits	28 bits



Getting deeper with an ECU

- ▶ For now, we managed to:
 - ▶ **Power up** an ECU
 - ▶ Find the **wake-up frame**
 - ▶ **Emulate sensors**
 - ▶ **Analyse** CAN messages payload
- ▶ But it could be useful to get information about the ECU's **configuration** and see if we can tweak it
- ▶ Let's get into **ECU diagnosis protocol**!

How to diagnose an ECU?

- ▶ ECUs need to be **diagnosed**/tuned over the time
- ▶ In 1968, VW introduced the first **On Board Diagnostic (OBD)** computer
- ▶ Various diagnostic protocols were developed over the time
- ▶ Most recent protocols allow to fully **reprogram/flash** an ECU
- ▶ Such protocols can also generate **specific behaviour** for development or mechanic workshop purposes, which can be useful for us
- ▶ From a **security point of view**, diagnosis support is a **target of interest**



Diagnosis of an ECU with the UDS protocol

- ▶ **The Unified Diagnostic Service (UDS - ISO 14229)** is a mandatory protocol for diagnosis, tuning and update operations on ECUs
- ▶ It uses a system of **Service** and **Sub-Function**
- ▶ Queries are made by the **Tester** (client) to a **Server** (ECU)
- ▶ Each **Server** has its own **Request arbitration ID** and **Reply arbitration ID**
- ▶ For each query, the **Server** replies with a **positive response** (Service code + 0x40) or **negative response** (0x7F)
- ▶ Usual **arbitration ID** range is **0x700-0x7FF** & **0x18DA0000-0x18DAFFFF**, **0x7DF** being reserved as a broadcast request

From 8 bytes to 4095 bytes

- ▶ Standard CAN bus has an **8-byte payload** limit
- ▶ To overcome this limitation, the **ISO-TP** protocol allows sending up to **4Kb** of data
- ▶ It **fragments** the payload in multiple CAN frame
- ▶ **UDS** requests and responses are carried with the **ISO-TP protocol**



- ▶ With ISO-TP, the **high nibble** of the first byte of each frame defines its type

- ▶ 4 values are possible:

- ▶ **0: Single Frame**
- ▶ **1: First Frame**
- ▶ **2: Consecutive Frame**
- ▶ **3: Flow Control Frame**

can0	720	[8]	03	22	F1	90	CC	CC	CC	CC
can0	728	[8]	10	12	62	F1	90	51	43	41
can0	700	[8]	30	00	00	CC	CC	CC	CC	CC
can0	728	[8]	21	52	5F	31	33	37	5F	56
can0	728	[8]	22	49	4E	21	21	AA	AA	AA

- ▶ For **Single Frame** the **low nibble** of the first byte is the length of the payload
- ▶ The second byte of a **First Frame** allows to specify a length up to **0xFFFF**

UDS: a verbose protocol

- ▶ UDS being a **verbose protocol**, understanding in a black-box approach supported services and subfunctions of an ECU is simple
- ▶ When requesting a service, a **valid response** will return the **requested service + 0x40**
- ▶ If the request is incorrect, the response returned will start with **0x7F**, followed by the requested Service and the **Negative Response Code**

can0	720	[8]	02	10 02	CC CC CC CC CC	# Request DiagnosticSession 0x02
can0	728	[8]	06	50 02	00 32 00 00 AA	# Positive response
can0	720	[8]	02	10 FF	CC CC CC CC CC	# Request DiagnosticSession 0xFF
can0	728	[8]	03	7F 10 12	AA AA AA AA	# NRC: Sub-function not supported (0x12)

- ▶ UDS support several services, including:
 - ▶ Diagnostic Session Control
 - ▶ Tester Present
 - ▶ ECU Reset
 - ▶ Read Data By Identifier and Write Data By Identifier
 - ▶ Security Access and Authentication
 - ▶ Read Memory By Address and Write Memory By Address
 - ▶ Request Upload, Request Download, Request Transfer Data, Request Transfer Exit, Transfer Data
- ▶ More details on the [UDS Wikipedia page](#)

Diagnostic Session Control & Tester Present



- ▶ An ECU supports several **diagnostic sessions** using **Diagnostic Session Control (0x10)**
- ▶ **Default session**, 0x01 is when the ECU is in normal mode
- ▶ Usually, session **ExtendedDiagnosticSession** (0x03) and **Programming Session** (0x02) are supported
- ▶ **OEM** and **Tier 1** manufacturers may use custom sessions in the range of **0x40-0x5F** and **0x60-0x7E**, which may give access to leftover **development-related features...**

can0	720	[8]	02	10 03	CC	CC	CC	CC	CC
can0	728	[8]	06	50 03	00	32	03	E8	AA



Diagnostic Session Control & Tester Present

- ▶ A session has a **timeout**, and can be maintained active by using service **TesterPresent (0x3E)**
- ▶ As **TesterPresent** only supports **subfunction 0x00**, it is a good service to use to scan for the **Arbitration Id** used for the UDS protocol

can0	720	[8]	02	3E 00	CC	CC	CC	CC	CC
can0	728	[8]	02	7E 00	AA	AA	AA	AA	AA

- ▶ The service **ECU Reset (0x11)** allows several reset types of the ECU, including:
 - ▶ **Full reset**
 - ▶ **On/Off ignition**
- ▶ Improper implementation could be used to perform a very basic **DoS** of the ECU
- ▶ A **full reset** revert the internal clock to 0, which can be useful to craft an **oracle** for time related protection/function

```
can0 720  [8] 02 11 01 CC CC CC CC CC
can0 728  [8] 02 51 01 AA AA AA AA AA
```



Read and Write Data By Identifier

- ▶ ECU **state** and **configuration** can be read using **Read Data By Identifier (0x22)**
- ▶ A **Data Identifier (DID)** is 2-byte long
- ▶ Some **Data Identifier** are writable, using **Write Data By Identifier (0x2E)**
- ▶ If **OEM** or **Tier 1** sessions are available, sometimes **development functions** can be activated by writing a specific DID

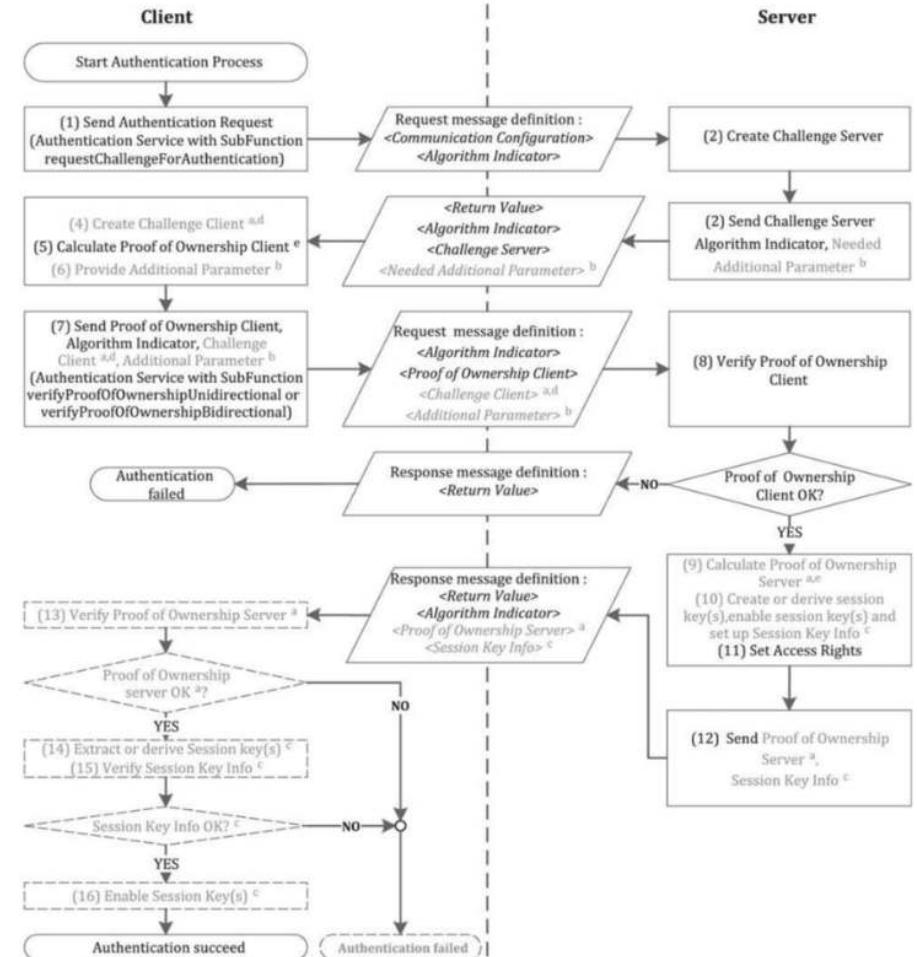
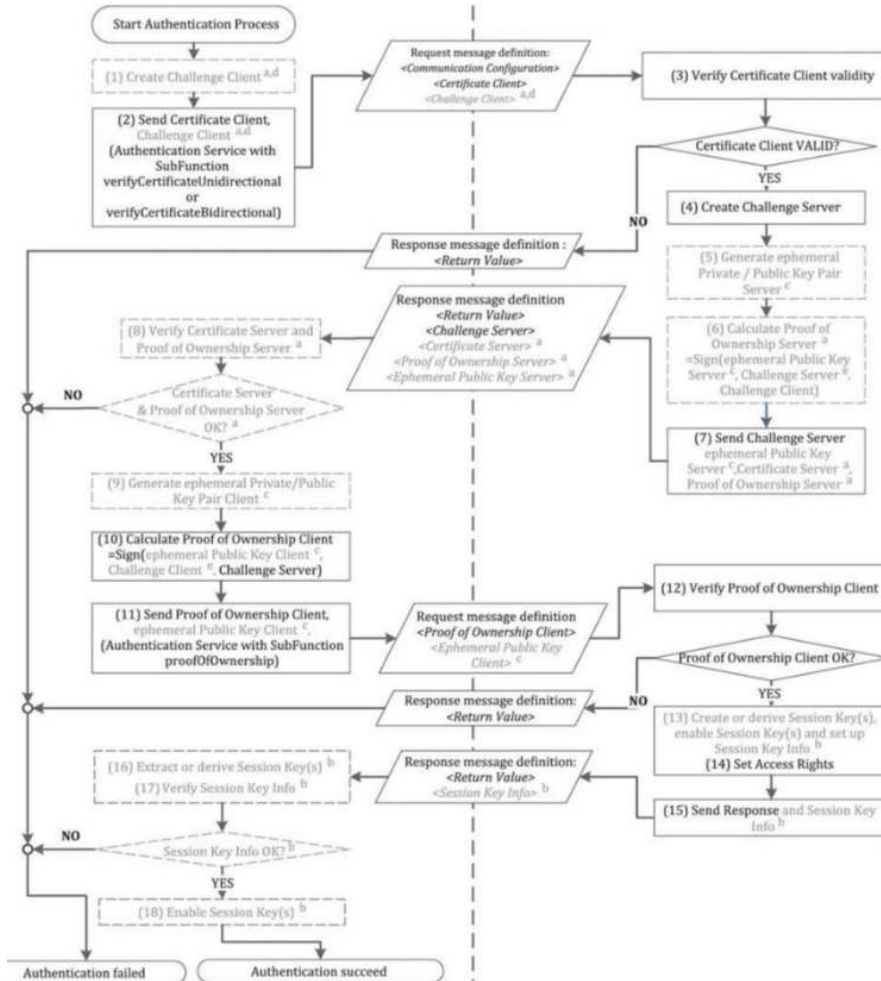
can0	720	[8]	02	22 01	23	CC	CC	CC	CC
can0	728	[8]	06	62 01 23	C0	FF	EE	AA	

Security Access and Authentication

- ▶ Critical services/sub-functions can be protected using **Security Access** or **Authentication**
- ▶ A **Security Access (0x27)** is a challenge-response mechanism, where a valid **key** must be sent after getting a **seed** from the ECU
- ▶ As the UDS protocol did not specify any algorithm for the **seed/key** challenge, many OEMs used **custom algorithms** (and some are leaked)
- ▶ **ISO 14229-1:2020** added **Authentication (0x29)**, which rely on proven **symmetric and asymmetric algorithm**, through a **challenge-response** or **certificates** verification

can0	720	[8]	02	27 01	CC	CC	CC	CC	CC
can0	728	[8]	07	67 01	FE	ED	C0	FF	EE
can0	720	[8]	02	27 02	EE	FF	C0	ED	FE
can0	728	[8]	07	67 02	AA	AA	AA	AA	AA

Security Access and Authentication



Routine Control



- ▶ **Routine Control (0x31)** is designed to perform **tests** or complex **actions** on the ECU
- ▶ It requires a **Data Identifier** and a **mode** (start, stop, read result)
- ▶ Again, it is an interesting service to investigate, as **development related Routine Control** are often left in production

```
can0 720  [8] 04 31 AA BB 01 CC CC CC
can0 728  [8] 04 71 AA BB 01 AA AA AA
```



Read and Write Memory By Address

- ▶ Often used during development phase, services **Read Memory By Address (0x23)** and **Write Memory By Address (0x3D)** allow a direct access to the RAM of the ECU
- ▶ It uses a **Address And Length Format Identifier** to specify the number of bytes to read/write and the address

can0	720	[8]	02	23	1	2	08	00	02	CC	CC
can0	728	[8]	07	63	12	08	00	02	BE	EF	



ECU update sequence

Typical ECU **firmware update** flow looks like:

1. **Diagnostic Session Control** for a programming session
2. **Read Data Identifier** to check version, state...
3. **Security Access**
4. **Routine Control** to clear the memory
5. **Request Download**, specifying size of the firmware
6. Multiple **Request Transfer Data**, **Transfer Data** and **Request Transfer Exit** to send the data
7. **Routine Control** to check and install the new firmware, switch memory bank
8. **Write Data Identifier** to write the current FW version
9. **ECU Reset**

A verbose protocol



UDS protocol being verbose, the **Negative Response Codes** (NRC) are useful when probing an ECU in a black-box testing or during reverse engineering

- ▶ 0x10: **General Reject**
- ▶ 0x11: **Service Not Supported**
- ▶ 0x12: **Sub-function Not Supported**
- ▶ 0x13: **Incorrect Message Length or Invalid Format**
- ▶ 0x22: **Conditions Not Correct**
- ▶ 0x24: **Request Sequence Error**
- ▶ 0x31: **Request Out Of Range**
- ▶ 0x33: **Security Access Denied**
- ▶ 0x35: **Invalid Key**
- ▶ 0x36: **Exceeded Number of Attempts**
- ▶ 0x7E: **Sub-Function not Supported in Active Session**
- ▶ 0x7F: **Service Not Supported in Active Session**

🧪 Lab 8 - Exploring UDS



Lab 8 - Exploring UDS

- ▶ In this lab, you will find the **UDS Arbitration ID** used by the Body Control Module, analyse Negative Response Code to find valid parameters and sequence to request a target service
- ▶ Complete challenge **UDS TX/RX ID** in **Test bench - Body Control Module**
- ▶ Complete challenge **UDS Read Data By Identifier**. To send a UDS request, script **uds_manager.py** is provided in folder **~/Tools**, supporting the following arguments:
 - ▶ **-session AA**: request specific session 0xAA
 - ▶ **-sa**: perform a Security Access, requesting the seed and asking you for the key
 - ▶ **-read-data AAAA**: read DataIdentifier 0xAAAA
 - ▶ **-write-data AAAA**: write DataIdentifier 0xAAAA
 - ▶ **-payload AABBCDD**: write payload 0xAABBCDD using **write-data**



In-Vehicle Infotainment

In-Vehicle Infotainment (IVI)

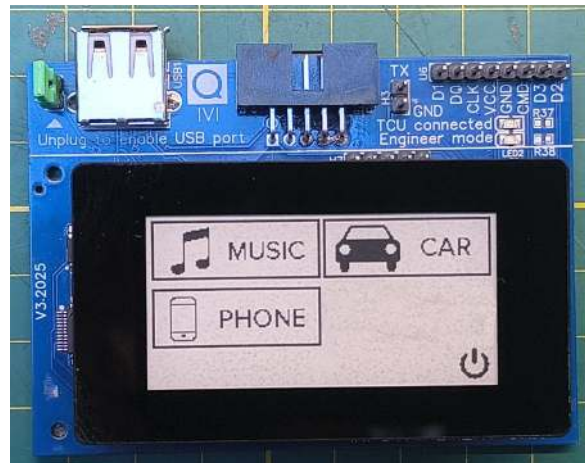


- ▶ In-Vehicle Infotainment (IVI) unit is the ECU in charge of most of the **multimedia** and **customer-oriented features** inside the vehicle
- ▶ Primary purpose is to enhance the driving experience by offering a wide range of features:
 - ▶ **Navigation**
 - ▶ **Entertainment**
 - ▶ **Connected applications**
 - ▶ **Interaction with driver/passenger smartphones**
- ▶ This ECU has the **widest range of attack surfaces**, being a target of choice



Illustration: [link]

- ▶ Can be connected to the **BCM** using a **10-pin ribbon cable**
- ▶ The USB connector is active if the jumper is removed, which **disable** its 10BASE-T1S connectivity
- ▶ The ePaper screen has a **touch capability**, to open various menus



Lab 9 - IVI pinout



Lab 9 - IVI pinout



You'll look at the **IVI pinout** from a technical documentation

- ▶ Complete challenge **Pinout** in **Test bench - Infotainment**

Automotive Ethernet - the need for speed



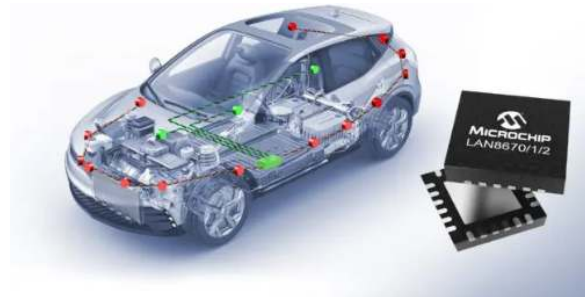
Illustration: [\[link\]](#)

- ▶ As cars are becoming more and more complex (assisted/autonomous driving), there is a growing need for:
 - ▶ **Low-latency**
 - ▶ **Robust links over** simple wires
 - ▶ **Flexible technologies** that cover multiple use-cases



Automotive Ethernet: two standards

- ▶ First implementation: **100/1000Base-T1**
 - ▶ Defined by **IEEE 802.3bw 2015**
 - ▶ “Classical” **point-to-point** network
- ▶ Evolution: **10Base-T1S**
 - ▶ Defined by **IEEE 802.3cg 2020**
 - ▶ **Multidrop** network

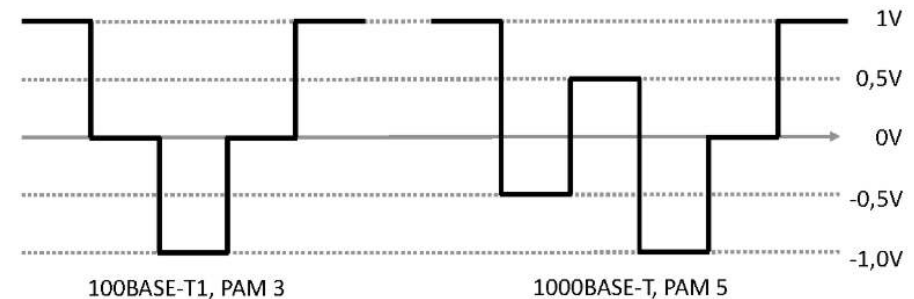
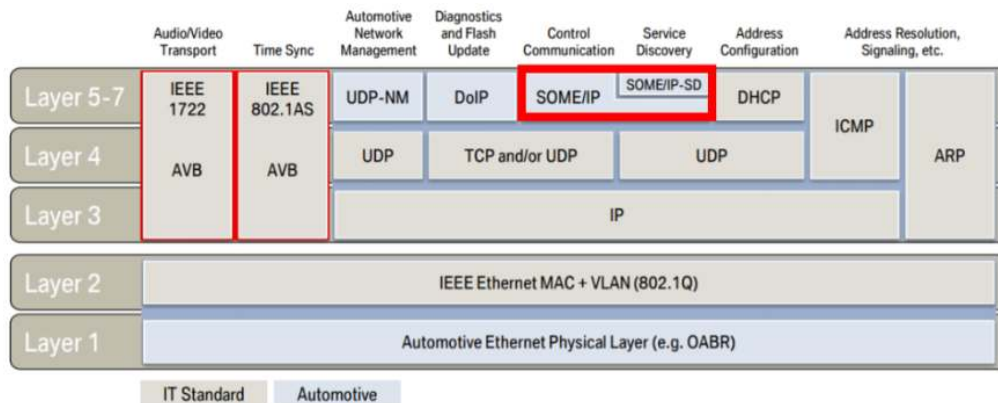


Difference between Ethernet & 100/1000Base-T1



- ▶ Only the **physical** layer differs:
 - ▶ Uses single differential **unshielded copper twisted pair**
 - ▶ Uses **PAM-3** signalling
 - ▶ Maximum length is **15 m**
 - ▶ **Connectors** are not defined (no RJ45 !)
 - ▶ A node is set as **Master**, the other as **Slave**, to handle **echo cancellation**

Illustration: [\[link\]](#), [\[link\]](#)

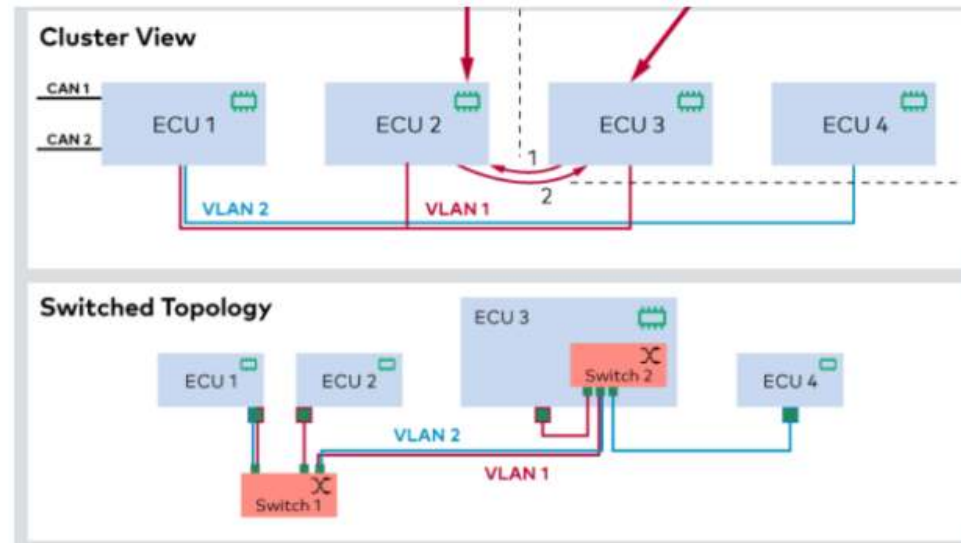


Network topology



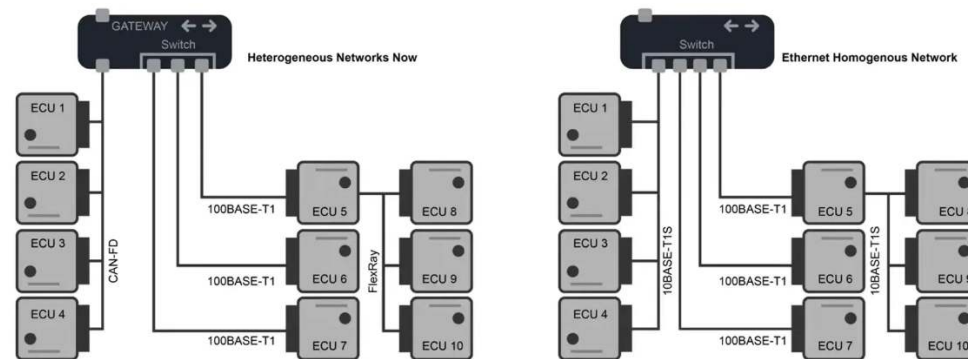
- ▶ ECUs are linked **port to port**
- ▶ An ECU can be a **switch** (like the Gateway)
- ▶ Several **VLANs** are used for **security** or to define different levels of **quality of services**

Illustration: [link]



10Base-T1S Automotive Ethernet

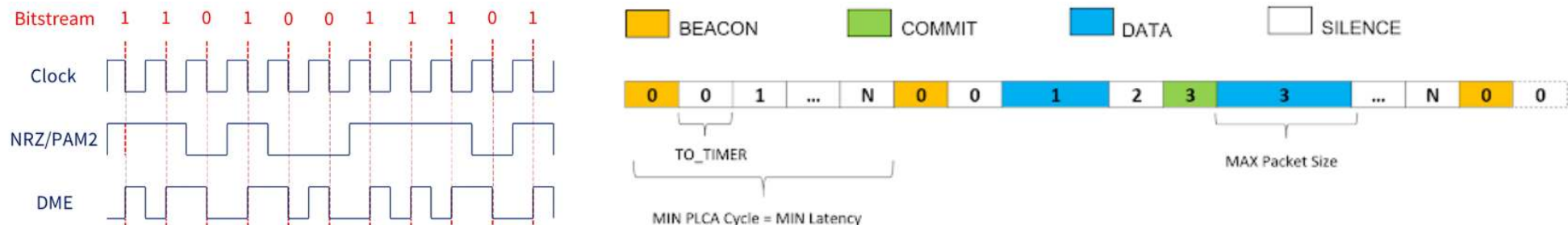
- Design to provide an **homogenous network**
- Allows **2 to 8 nodes** to communicate over a **single twisted pair**, up to 25m
- Aims to replace classical automotive networks, like CAN, having an all-Ethernet network



10Base-T1S - technical specifications

- ▶ Also uses single differential **unshielded copper twisted pair**
- ▶ But relies on **Differential Manchester Encoding (DEM)** signalling
- ▶ Bus is terminated by **100 ohms resistors**
- ▶ Each **node** has an **ID**, 0 being for the Master, for the **Physical Layer Collision Avoidance (PLCA)**
- ▶ The Master sends **periodic beacon**. Slave nodes are given a **transmit opportunity** in order of their ID
- ▶ A **silence** (20 bits) is when a **node** has **no data to transmit**. It could also send a **commit** to buy additional time to transmit data

Illustration: [link], [link]

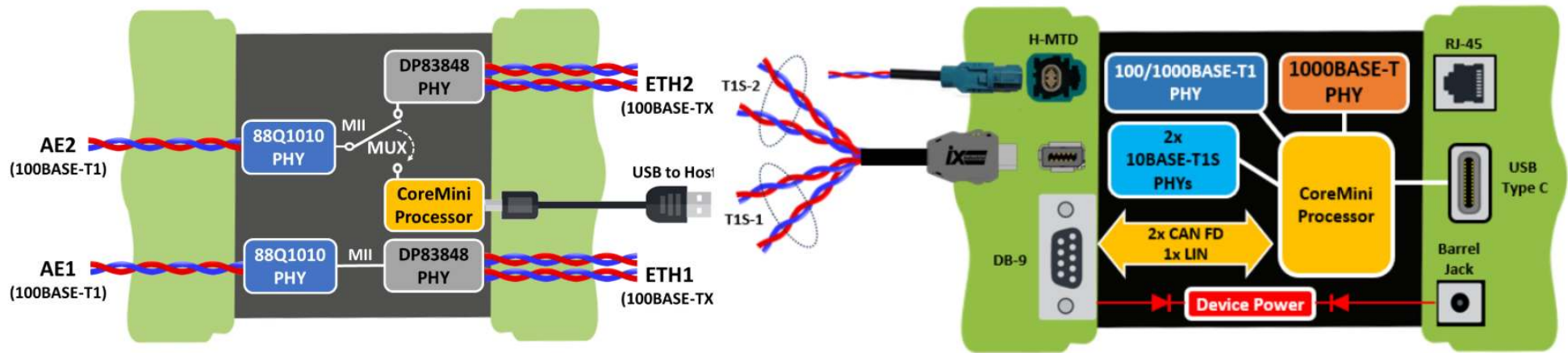


Connecting to an Automotive Ethernet network



- ▶ To connect to an automotive Ethernet network, a **Media Independent Interface (MII)** is required
- ▶ It **bridges** classical and automotive ethernet **physical layers** so you can plug an RJ-45

Illustration: [link]





Working with Automotive Ethernet network

- ▶ As only the **physical layer differs**, all **common tools** could be used to work with **Automotive Ethernet**:
 - ▶ **Wireshark**
 - ▶ **Nmap**
 - ▶ **Scapy**
 - ▶ ...
- ▶ **Automotive Ethernet** has to be **Energy Efficient**, specific **wake-up** frame could be required to activate it on an ECU

Interlude: termshark

- ▶ **Termshark** is installed on the VM, providing **Wireshark** in your terminal
- ▶ You can launch it by invoking: `sudo termshark -i [iface to listen to]`
- ▶ Some protocols are not natively detected by **termshark**, you can add decoder rules by adding: `-d udp.port==30490,someip`
- ▶ Using TAB key you can navigate between displays and access filter by pressing /
- ▶ To close termshark, type ESC then :q!

termshark v2.4.0 | demo.pcap

Filter: <Apply> <Recent>

No.	Time	Source	Dest	Proto	Length	Info	Seq
1	0.000000	192.168.1.1	192.168.1.2	TCP	78	62014 → 21 [SYN] Seq=0 Win=65535 Len=0 MSS=14 0	0
2	0.000043	192.168.1.2	192.168.1.1	TCP	74	21 → 62014 [SYN, ACK] Seq=0 Ack=1 Win=5792 Le 0	0
3	0.000167	192.168.1.1	192.168.1.2	TCP	66	62014 → 21 [ACK] Seq=1 Ack=1 Win=65535 Len=0 1	1
4	0.040555	192.168.1.2	192.168.1.1	FTP	136	Response: 220 ProFTPD 1.3.0a Server (ProFTPD 1	1
5	0.040757	192.168.1.1	192.168.1.2	TCP	66	62014 → 21 [ACK] Seq=1 Ack=71 Win=65535 Len=0 1	1
6	2.522199	192.168.1.1	192.168.1.2	FTP	76	Request: USER ftp	1
7	2.522248	192.168.1.2	192.168.1.1	TCP	66	21 → 62014 [ACK] Seq=71 Ack=11 Win=5792 Len=0 71	71
8	2.641952	192.168.1.2	192.168.1.1	FTP	142	Response: 331 Anonymous login ok, send your c 71	71
9	2.642138	192.168.1.1	192.168.1.2	TCP	66	62014 → 21 [ACK] Seq=11 Ack=147 Win=65535 Len=11	11
10	4.098171	192.168.1.1	192.168.1.2	FTP	76	Request: PASS ftp	11

[+] Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface unknown, id 0 [=]
 [+] Ethernet II, Src: Apple_2e:52:3b (00:17:f2:2e:52:3b), Dst: IntelCor_9f:04:2f (00:13:20:9f:04:2f)
 [+] Internet Protocol Version 4, Src: 192.168.1.182, Dst: 192.168.1.231
 [+] Transmission Control Protocol, Src Port: 62014, Dst Port: 21, Seq: 0, Len: 0

0000 00 13 20 9f 04 2f 00 17 f2 2e 52 3b 08 00 45 00 R. . .E.
 0010 00 40 1c 60 40 00 40 06 99 6a c0 a8 01 b6 c0 a8 .@. @. .j.
 0020 01 e7 f2 3e 00 15 04 84 6e d9 00 00 00 b0 02 n.
 0030 ff ff e5 df 00 00 02 04 05 b4 01 03 03 00 01 01 G.
 0040 08 0a 1e 8d 47 f6 00 00 00 00 04 02 00 00

🧪 Lab 10 - Exploring Ethernet Automotive



Lab 10 - Exploring Ethernet Automotive

In this lab you'll scan the **10Base-T1S Ethernet Automotive** connectivity of the **Infotainment Unit**



- ▶ Connect pins **N** and **P** of the **USB to 10Base-T1S** adapter to **N** and **P** of the **10BASE-T1S** pins on the **Body Control Module**
- ▶ Find the IP address of the **IVI** and analyse network traffic by completing challenges **IP** and **Port**



- ▶ An ECU can support two types of diagnostic, the **Diagnostic over CAN (DoCAN)** and the **Diagnostic over IP (DoIP)**
- ▶ In both cases the diagnostic protocol used is **UDS**; only the **transport layer** differs
- ▶ **DoIP** is a protocol defined by **ISO 13400**
- ▶ It uses both **TCP** and **UDP** and usually exposes port **13400**

Diagnostic over IP

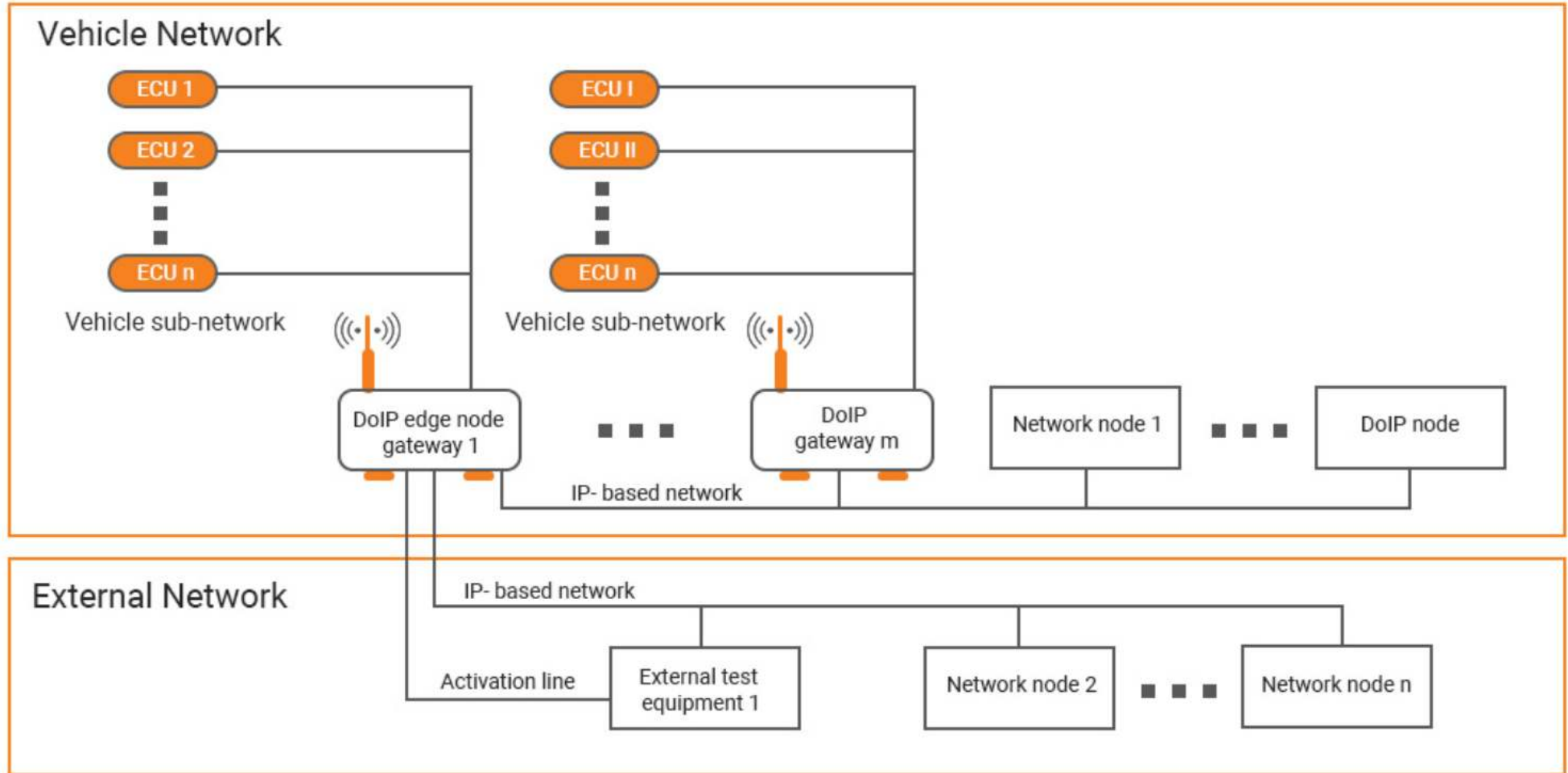
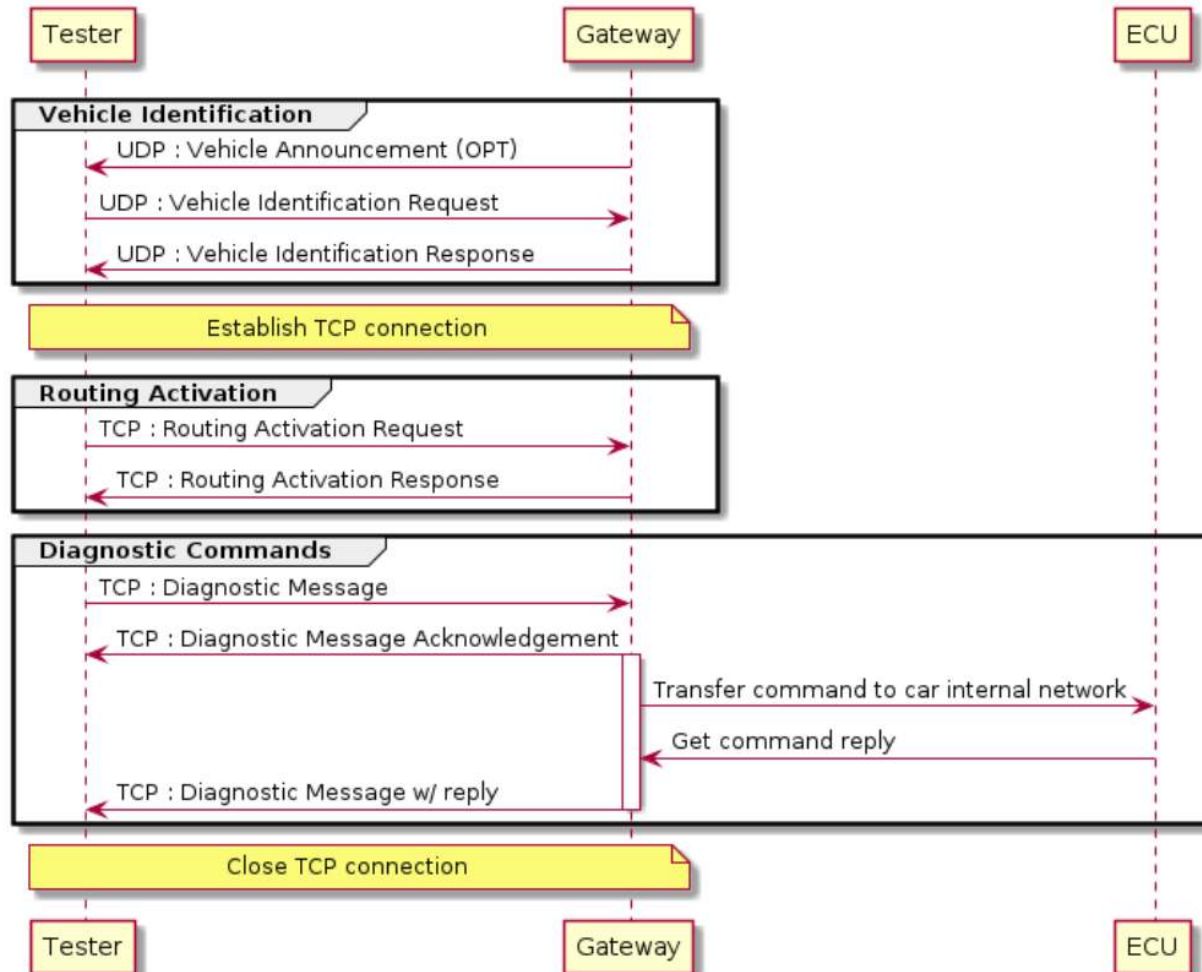


Illustration: [\[link\]](#)

Diagnostic over IP - Flowchart



Diagnostic over IP - Packet



8 bits	8 bits	8 bits	8 bits
Protocol Version (0x02)	Inverse Protocol Version (0xFD)	Payload Type (0x8001)	
Payload Length			
Payload			

- ▶ DoIP supports several payload types, including:
 - ▶ **0x0001** : Vehicle Identification Request Message
 - ▶ **0x0002** : Vehicle Identification Request Message with EID
 - ▶ **0x0003** : Vehicle Identification Request Message with VIN
 - ▶ **0x0004** : Vehicle Announcement Message/Vehicle Identification Response
 - ▶ **0x8001** : Diagnostic Message
 - ▶ **0x8002** : Diagnostic Message Positive Acknowledgement
 - ▶ **0x8003** : Diagnostic Message Negative Acknowledgement
- ▶ Each ECU accessible to the **DoIP gateway** will have a 2-byte **logical address**
- ▶ The tester also have a logical address
- ▶ At boot, a DoIP server will send **Vehicle Announcement Message**, and may continue periodically

🧪 Lab 11 - Diagnostic over IP



Lab 11 - Diagnostic over IP

In this lab you will explore the DoIP protocol, sending few requests and analysing corresponding network traffic to understand how this protocol works

- ▶ You can send **DoIP diagnostic message** using **doip_requests.py** in **~Tools**
- ▶ Arguments are almost the same as **uds_requests.py**, plus two arguments:
 - ▶ **-ip**: IP address of the DoIP ECU
 - ▶ **-logical-addr**: logical address of the target ECU
- ▶ Complete all remaining **DoIP** challenges in **Test bench - Infotainment**



Telematic Control Unit

Telematic Control Unit (TCU) is the key element in a car's connectivity, offering:

- ▶ **Backend connectivity**
 - ▶ OEM monitoring
 - ▶ Mobile application interaction (door lock/unlock, positioning, charging status, HVAC...)
 - ▶ Firmware updates
- ▶ **Emergency/Assistance call (xCall)**
- ▶ **Internet connectivity**
 - ▶ On-board navigation
 - ▶ 3rd application connectivity

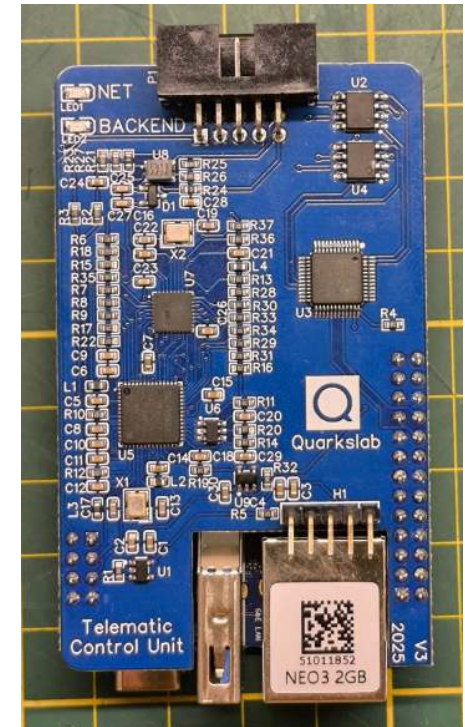
Role of IoT in connected vehicle technology



Training TCU



- ▶ Can be connected to the **BCM** using a **10-pin ribbon cable**
- ▶ Runs **srs-us** to emulate a 4G user equipment with card SIM support
- ▶ Emulation of the LTE packets is performed with the **ZMQ protocol**
- ▶ Establishes connection to a **training backend** using LTE emulated connectivity otherwise uses the **Infotainment unit** as a router



🧪 Lab 12 - TCU discovery



Lab 12 - TCU discovery

As we did for the other ECUs, a first step is to identify to **which network of the car** the ECU is connected to

- ▶ The training **TCU** is connected to the two **CAN** buses and the **10BASE-T1S** network
- ▶ Look for the IP address of the TCU and complete challenge **IP** in **Test bench - Telematic**
- ▶ Search for the UDS Arbitration ID, complete challenge **UDS**
- ▶ The **TCU** and the **IVI** exchange data, analyse Automotive Ethernet network traffic and complete challenge **Ports**



- ▶ **SOME/IP** is a lightweight **service-oriented** automotive protocol
- ▶ Was developed by the **BMW** group in 2011
- ▶ Offers various middleware features like:
 - ▶ **Serialization**
 - ▶ **Remote Procedure Call**
 - ▶ **Event Notification**
- ▶ Uses port **30490** and **30500+**



Signal-based vs Service-oriented

- ▶ **Signal-based** protocols like CAN, LIN, Flexray are not flexible:
 - ▶ **Messages** are sent even if the receiver does **not need** them
- ▶ With **Service-oriented** communication
 - ▶ The receiver **requests** a message **on demand**
 - ▶ Receiver **subscribes** to a message to be **notified** on each new event or updated value

SOME/IP - key concepts



- ▶ An ECU **produces** a Service, acting as a **Server**
- ▶ An ECU **consumes** a Service, acting as a **Client**
- ▶ A Service is a combination of:
 - ▶ **Methods**: method, function, subroutine invocables
 - ▶ **Events**: subscribable callbacks (only group of events are subscribable)
 - ▶ **Fields**: status with a value, could have getter, setter and notifier
- ▶ A Service is identified by a unique **2 bytes Service ID**
- ▶ Each **method/event** of a Service has a unique **15 bits Method ID/Event ID**

SOME/IP example



TCU

IP: 192.168.222.1

Port: 30500

Provided Service:

Backend service (0x2233)

Event Groups

Connectivity status (0x700)

Backend command (0x702)

Method

Vehicle_state_setter (0x601)

Modem_IMEI_getter (0x501)

Modem_IMSI_getter (0x502)

Modem_ICCID_getter (0x503)

TCU_connectivity_getter (0x504)

IVI

IP: 192.168.222.2

Port: -

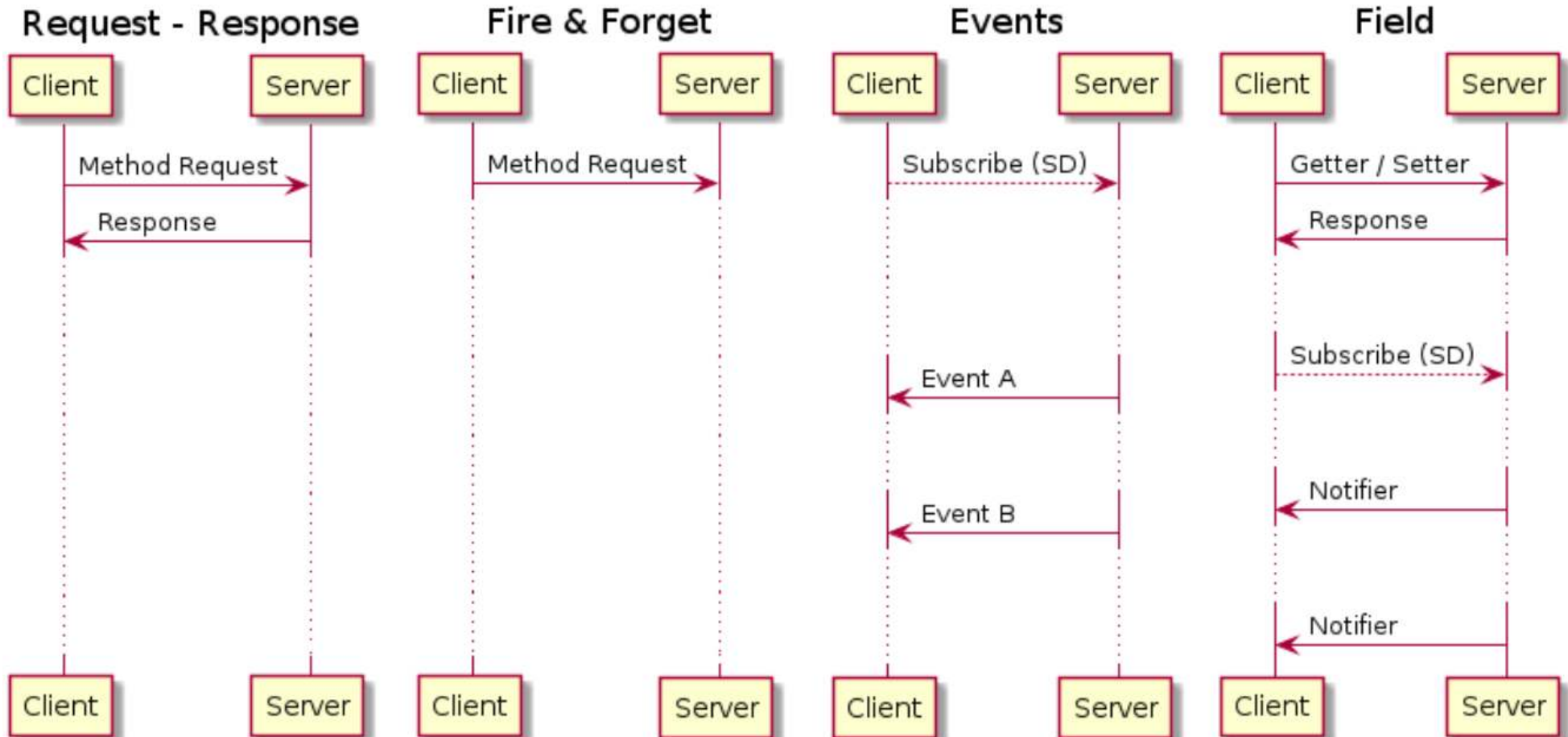
Provided Service:

None

Subscribed Service:

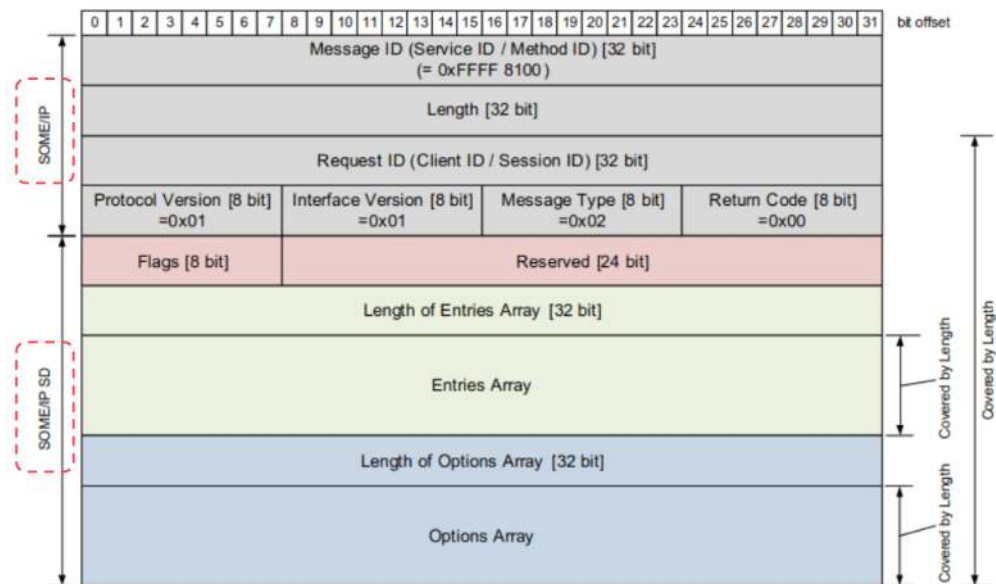
Backend service

SOME/IP - features

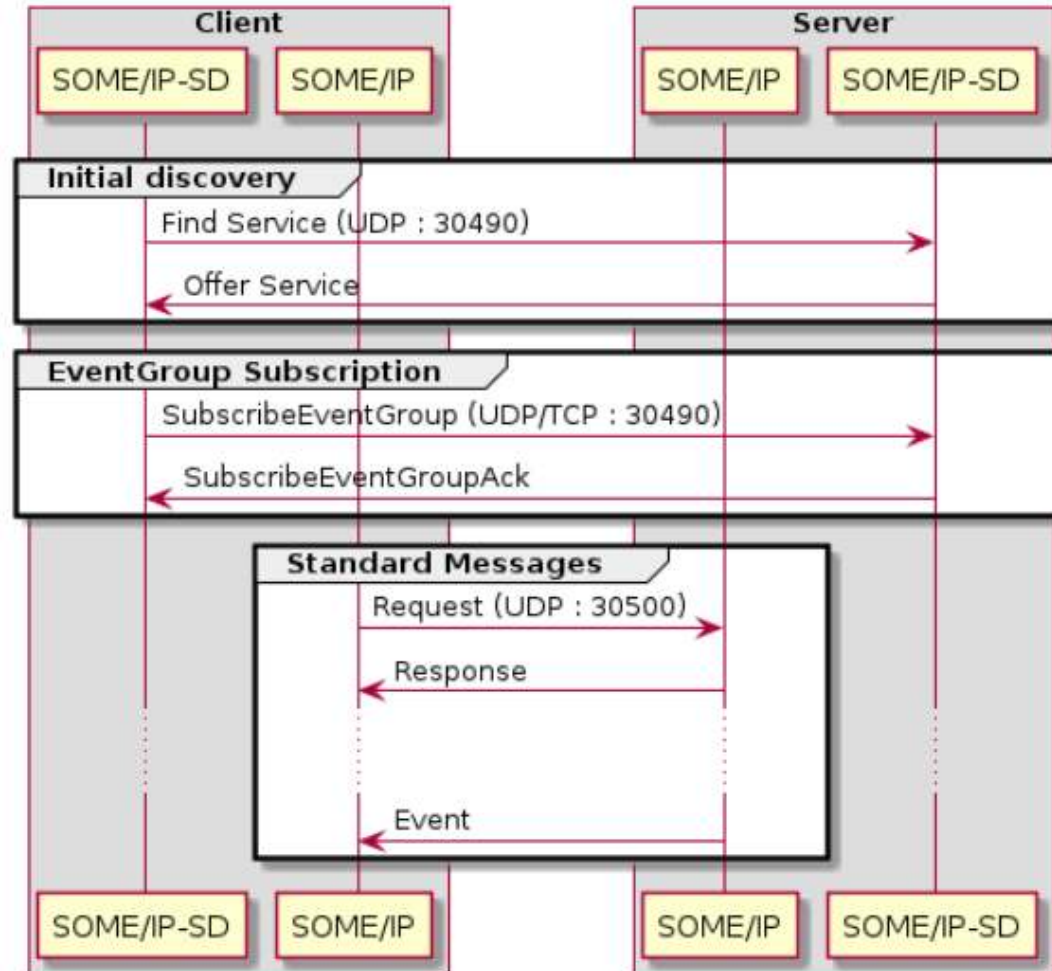


SOME/IP - Service Discovery

- For **discovering** and **subscribing** to an **EventGroup**, a client will use **SOME/IP-SD** messages
- SOME/IP-SD message is **encapsulated** into standard **SOME/IP** message



SOME/IP and SOME/IP-SD relationship



Lab 13 - Exploring SOME/IP



Lab 13 - Exploring SOME/IP



Let's discover the SOME/IP protocol, you'll analyse a SOME/IP SD packet to find information about a Service, craft a request and dig into notifications

- ▶ Complete challenges **SOME-IP** in **Test-bench - Telematic**

Thank you

Contact information

Email:

contact@quarkslab.com

Phone:

+33 1 58 30 81 51

Website:

<https://www.quarkslab.com>