

CAN bus

the ultimate guide

CAN | J1939 | OBD2 | UDS | CANopen | CAN FD | LIN | DBC | CAN Errors

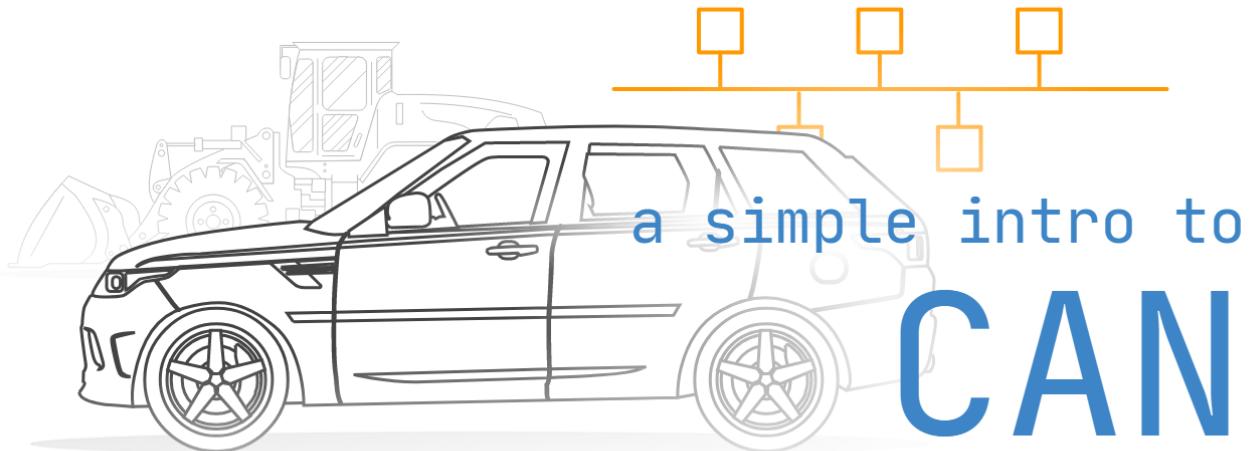
NMEA 2000 | ISOBUS | CCP/XCP on CAN

Table of Contents

CAN Bus Explained - A Simple Intro	5
What is CAN bus?	5
Top 4 benefits of CAN bus	8
The CAN bus history in short	8
The future of CAN bus	9
What is a CAN frame?	9
Logging CAN data - example use cases	10
How to log CAN bus data	10
How to decode raw CAN data to 'physical values'	11
What is the link between CAN, J1939, OBD2, CANopen ...?	13
J1939 Explained - A Simple Intro	15
What is J1939?	15
J1939 history & future	16
4 key characteristics of J1939	16
The J1939 connector (9-pin)	18
The J1939 PGN and SPN	19
J1939 truck sample data: Raw & physical values	22
J1939 request messages	23
J1939 transport protocol (TP)	24
Logging J1939 data - example use cases	27
6 practical tips for J1939 data logging	27
OBD2 Explained - A Simple Intro	29
What is OBD2?	29
The OBD2 connector	30
Does my car have OBD2?	31
Link between OBD2 and CAN bus	31
OBD2 history & future	32
OBD2 parameter IDs (PID)	34
How to log OBD2 data?	34
Raw OBD2 frame details	35
OBD2 data logging - use case examples	36
UDS Explained (Unified Diagnostic Services)	38
What is the UDS protocol?	38
UDS message structure	39
UDS vs CAN bus: Standards & OSI model	45
CAN ISO-TP - Transport Protocol (ISO 15765-2)	47
UDS vs. OBD2 vs. WWH-OBD vs. OBDDonUDS	49
FAQ: How to request/record UDS data	52
Example 1: Record single frame UDS data (Speed via WWH-OBD)	55
Example 2: Record & decode multi frame UDS data (SoC)	56
Example 3: Record the Vehicle Identification Number	59

UDS data logging - applications	62
CANopen Explained - A Simple Intro	63
What is CANopen?	63
Six core CANopen concepts	64
CANopen communication basics	65
CANopen Object Dictionary	69
SDO - configuring the CANopen network	70
PDO - operating the CANopen network	72
CANopen data logging - use case examples	73
CAN FD Explained - A Simple Intro	74
Why CAN FD?	74
What is CAN FD?	74
How does CAN FD work?	75
Overhead and data efficiency of CAN FD vs. CAN	77
Examples: CAN FD applications	79
Logging CAN FD data - use case examples	79
CAN FD - outlook	80
LIN Bus Explained - A Simple Intro	82
What is LIN bus?	82
LIN bus applications	84
How does LIN bus work?	85
Six LIN frame types	86
Advanced LIN topics	87
LIN Description File (LDF) vs. DBC files	88
LIN bus data logging - use case examples	90
Practical considerations for LIN data logging	91
CAN DBC File Explained - A Simple Intro	92
What is a CAN DBC file?	92
Example: Extract physical value of EngineSpeed signal	94
CAN DBC editor playground	96
J1939/OBD2 data & DBC samples	96
Advanced: Meta info, attributes & multiplexing	96
DBC software tools (editing & processing)	98
CAN Bus Errors Explained - A Simple Intro	100
What are CAN bus errors?	100
The CAN bus error frame	101
CAN error types	105
CAN node states & error counters	108
Examples: Generating & logging error frames	109
LIN bus errors	113
Example use cases for CAN error frame logging	114
FAQ	114
NMEA 2000 Explained - A Simple Intro	116
What is NMEA 2000?	116

NMEA 2000 vs NMEA 0183	117
NMEA history and certification	118
NMEA 2000 OSI model & standards	120
NMEA 2000 connectors & network topology	121
NMEA 2000 Fast Packet	123
N2K PGN & data fields	124
Logging NMEA 2000 maritime data	127
ISOBUS (ISO 11783) Explained - A Simple Intro	129
What is ISOBUS?	129
ISOBUS history and AEF	130
ISOBUS OSI model & standards	131
ISOBUS Functionalities	133
ISOBUS vs SAE J1939	136
The ISOBUS connectors	137
ISOBUS PGN and SPN [+ DBC]	138
Logging tractor/implement data	140
Data logging use case examples	142
CCP / XCP on CAN Explained - A Simple Intro	143
What is CCP/XCP?	143
CCP message types	146
How to record ECU data via CCP	150
Decoding CCP signal data from ECUs	155
A2L - ECU Description Files	157
Seed & key authorization	159
XCP on CAN - the basics	160
Using the CANedge for CCP/XCP data acquisition	164
CCP/XCP data logging - applications	164



CAN Bus Explained - A Simple Intro

Need a simple, practical intro to CAN bus? In this tutorial we explain the Controller Area Network (CAN bus) 'for dummies' incl. message interpretation, CAN logging - and the link to OBD2, J1939 and CANopen.

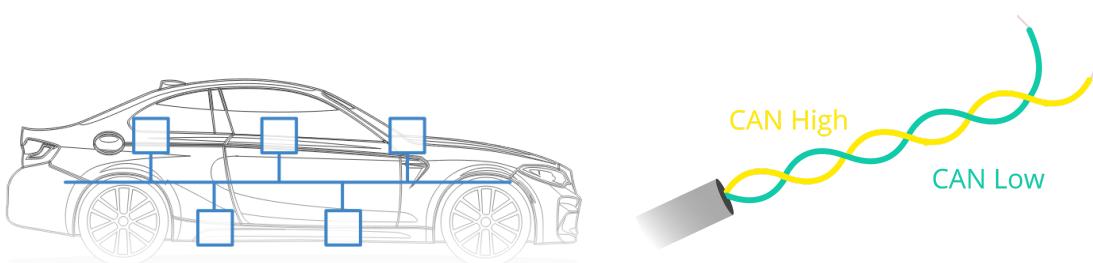
Read on to learn why this has become the #1 guide on CAN bus.

What is CAN bus?

Your car is like a human body:

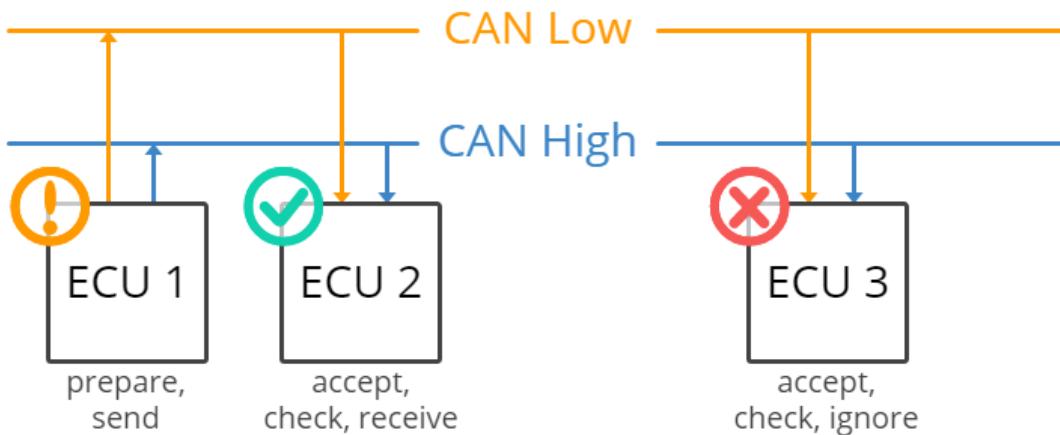
The [Controller Area Network](#) (CAN bus) is the nervous system, enabling communication.

In turn, 'nodes' or 'electronic control units' (ECUs) are like parts of the body, interconnected via the CAN bus. Information sensed by one part can be shared with another.



So what is an ECU?

In an automotive CAN bus system, ECUs can e.g. be the engine control unit, airbags, audio system etc. A modern car may have up to 70 ECUs - and each of them may have information that needs to be shared with other parts of the network.



This is where the CAN standard comes in handy:

The CAN bus system enables each ECU to communicate with all other ECUs - without complex dedicated wiring.

Specifically, an ECU can prepare and broadcast information (e.g. sensor data) via the CAN bus (consisting of two wires, CAN low and CAN high). The broadcasted data is accepted by all other ECUs on the CAN network - and each ECU can then check the data and decide whether to receive or ignore it.

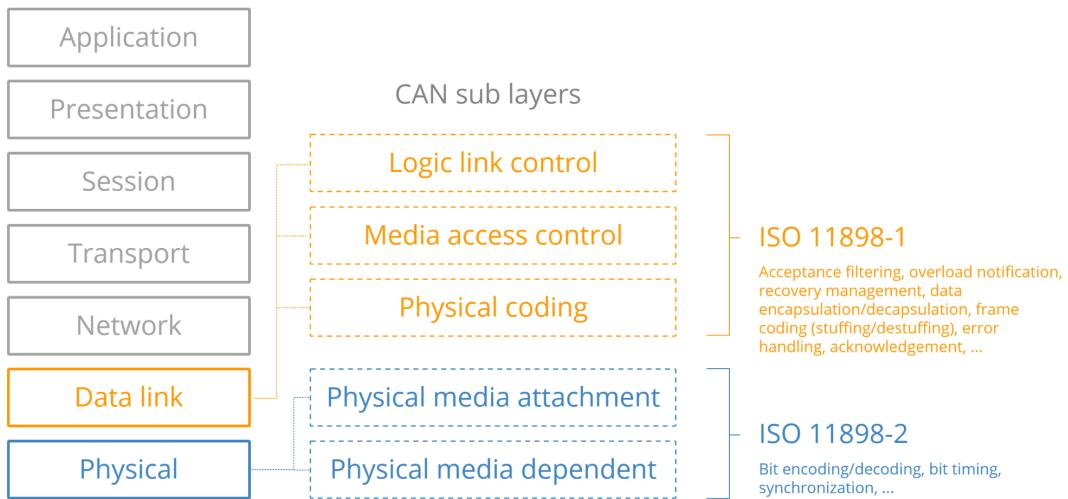
CAN bus physical & data link layer (OSI)

In more technical terms, the controller area network is described by a [data link layer](#) and physical layer. In the case of high speed CAN, ISO 11898-1 describes the data link layer, while [ISO 11898-2](#) describes the physical layer. The role of CAN is often presented in the 7 layer OSI model as per the illustration.

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

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

7 layer OSI model



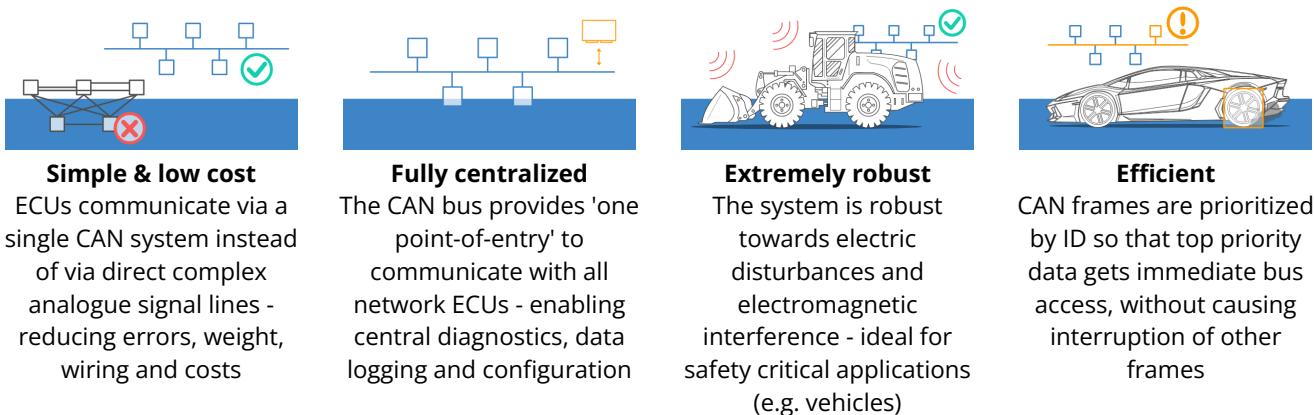
High speed CAN, low speed CAN, LIN bus, ...

In the context of automotive vehicle networks, you'll often encounter a number of different types of network types. Below we provide a very brief outline:

- **High speed CAN bus:** The focus of this article is on high speed CAN bus (ISO 11898). It is by far the most popular CAN standard for the physical layer, supporting bit rates from 40 kbit/s to 1 Mbit/s (Classical CAN). It provides simple cabling and is used in practically all automotive applications today. It also serves as the basis for several higher layer protocols such as [OBD2](#), [J1939](#), [NMEA 2000](#), [CANopen](#) etc. The second generation of CAN is referred to as [CAN FD \(CAN with Flexible Data-rate\)](#)
- **Low speed CAN bus:** This standard enables bit rates from 40 kbit/s to 125 kbit/s and allows the CAN bus communication to continue even if there is a fault on one of the two wires - hence it is also referred to as 'fault tolerant CAN'. In this system, each CAN node has its own [CAN termination](#)
- **LIN bus:** LIN bus is a lower cost supplement to CAN bus networks, with less harness and cheaper nodes. LIN bus clusters typically consist of a LIN master acting as gateway and up to 16 slave nodes. Typical use cases include e.g. non-critical vehicle functions like aircondition, door functionality etc. - for details see our [LIN bus intro](#) or [LIN bus data logger](#) article
- **Automotive ethernet:** This is increasingly being rolled out in the automotive sector to support the high bandwidth requirements of ADAS (Advanced Driver Assistance Systems), infotainment systems, cameras etc. Automotive ethernet offers much higher data transfer rates vs. CAN bus, but lacks some of the safety/performance features of Classical CAN and CAN FD. Most likely, the coming years will see both automotive ethernet, CAN FD and [CAN XL](#) being used in new automotive and industrial development

Top 4 benefits of CAN bus

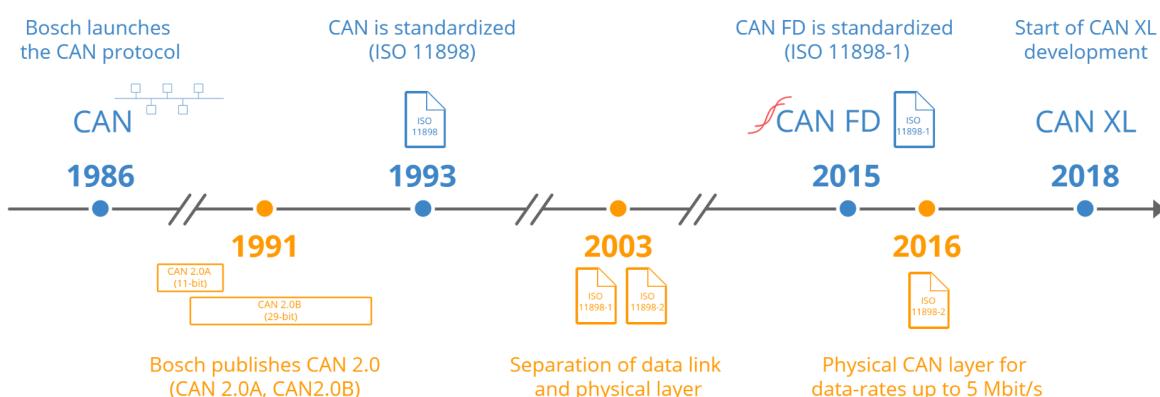
The CAN bus standard is used in practically all vehicles and many machines due to below key benefits:



The CAN bus history in short

- Pre CAN: Car ECUs relied on complex point-to-point wiring
- 1986: Bosch developed the CAN protocol as a solution
- 1991: Bosch published CAN 2.0 (CAN 2.0A: 11 bit, 2.0B: 29 bit)
- 1993: CAN is adopted as international standard (ISO 11898)
- 2003: ISO 11898 becomes a standard series
- 2012: Bosch released the CAN FD 1.0 (flexible data rate)
- 2015: The CAN FD protocol is standardized (ISO 11898-1)
- 2016: The physical CAN layer for data-rates up to 5 Mbit/s standardized in ISO 11898-2

Today, CAN is standard in automotives ([cars](#), [trucks](#), buses, tractors, ...), [ships](#), planes, [EV batteries](#), [machinery](#) and more.



The future of CAN bus

Looking ahead, the CAN bus protocol will [stay relevant](#) - though it will be impacted by major trends:

- A need for increasingly advanced vehicle functionality
- The rise of [cloud computing](#)
- Growth in [Internet of Things](#) (IoT) and connected vehicles
- The impact of [autonomous vehicles](#)

In particular, the rise in connected vehicles ([V2X](#)) and cloud will lead to a rapid growth in [vehicle telematics](#) and [IoT CAN loggers](#). In turn, bringing the CAN bus network 'online' also exposes vehicles to [security risks](#) - and may require a shift to new CAN protocols like CAN FD.

The rise of CAN FD

As vehicle functionality expands, so does the load on the CANbus. To support this, [CAN FD](#) (Flexible Data Rate) has been designed as the 'next generation' CAN bus. Specifically, CAN FD offers three benefits (vs Classical CAN):

- It enables data rates up to 8 Mbit/s (vs 1 Mbit/s)
- It allows data payloads of up to 64 bytes (vs 8 bytes)
- It enables improved security via authentication

In short, CAN FD boosts speed and efficiency - and it is therefore being rolled out in newer vehicles. This will also drive an increasing need for [IoT CAN FD data loggers](#).

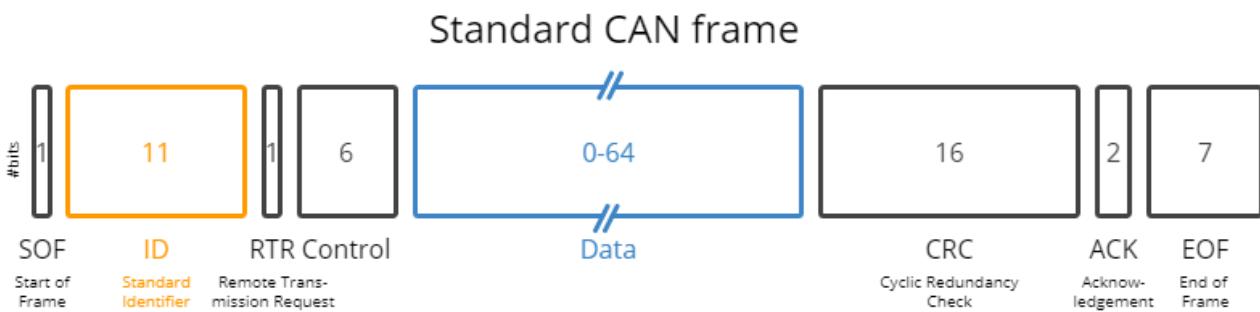
"The first cars using CAN FD will appear in 2019/2020 and CAN FD will replace step-by-step Classical CAN"

- CAN in Automation (CiA), "[CAN 2020: The Future of CAN Technology](#)"

What is a CAN frame?

Communication over the CAN bus is done via CAN frames.

Below is a standard CAN frame with 11 bits identifier (CAN 2.0A), which is the type used in most cars. The extended 29-bit identifier frame (CAN 2.0B) is identical except the longer ID. It is e.g. used in the [J1939 protocol](#) for heavy-duty vehicles. Note that the CAN ID and Data are highlighted - these are important when recording CAN bus data, as we'll see below.



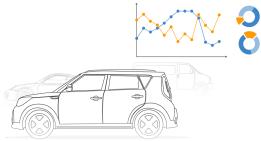
The 8 CAN bus protocol message fields

- **SOF**: The Start of Frame is a 'dominant 0' to tell the other nodes that a CAN node intends to talk
- **ID**: The ID is the frame identifier - lower values have higher priority

- **RTR:** The Remote Transmission Request indicates whether a node sends data or requests dedicated data from another node
- **Control:** The Control contains the Identifier Extension Bit (IDE) which is a 'dominant 0' for 11-bit. It also contains the 4 bit Data Length Code (DLC) that specifies the length of the data bytes to be transmitted (0 to 8 bytes)
- **Data:** The Data contains the data bytes aka payload, which includes CAN signals that can be extracted and decoded for information
- **CRC:** The Cyclic Redundancy Check is used to ensure data integrity
- **ACK:** The ACK slot indicates if the node has acknowledged and received the data correctly
- **EOF:** The EOF marks the end of the CAN frame

Logging CAN data - example use cases

There are several common use cases for recording CAN bus data frames:



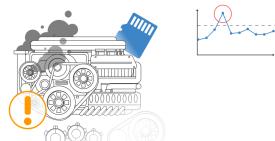
Logging/streaming data from cars
OBD2 data from cars can e.g. be used to reduce fuel costs, improve driving, test prototype parts and insurance
[learn more](#)



Heavy duty fleet telematics
J1939 data from trucks, buses, tractors etc. can be used in fleet management to reduce costs or improve safety
[learn more](#)



Predictive maintenance
Vehicles and machinery can be monitored via IoT CAN loggers in the cloud to predict and avoid breakdowns
[learn more](#)

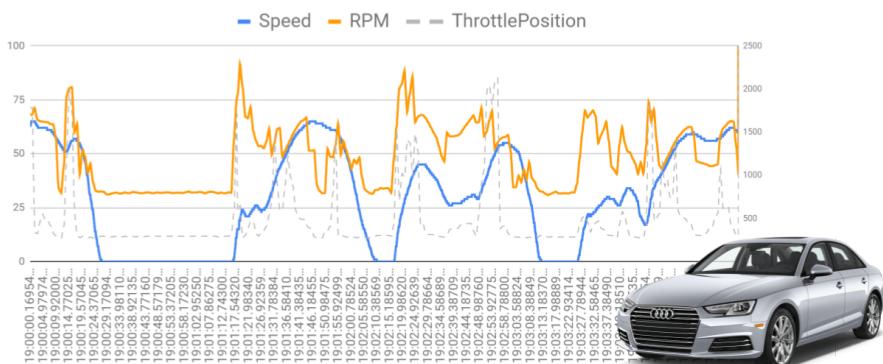


Vehicle/machine blackbox
A CAN logger can serve as a 'blackbox' for vehicles or equipment, providing data for e.g. disputes or diagnostics
[learn more](#)

How to log CAN bus data

As mentioned, two CAN fields are important for CAN logging: The CAN ID and the Data. To record CAN data you need a [CAN logger](#). This lets you log timestamped CAN data to an SD card. In some cases, you need a [CAN interface](#) to stream data to a PC - e.g. for [car hacking](#).

OBD2 Data - Speed, RPM, ThrottlePos (Audi A4, CANedge2)



Connecting to the CAN bus

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

- **Cars:** In most cars, you simply use an [OBD2 adapter](#) to connect. In most cars, this will let you log raw CAN data, as well as perform requests to log [OBD2](#) or [UDS \(Unified Diagnostic Services\)](#) data
- **Heavy duty vehicles:** To [log J1939 data](#) from trucks, excavators, tractors etc you can typically connect to the J1939 CAN bus via a standard [J1939 connector cable \(deutsch 9-pin\)](#)
- **Maritime:** Most ships/boats use the [NMEA 2000 protocol](#) and enable connection via an M12 adapter to log marine data
- **CANopen:** For [CANopen logging](#), you can often directly use the CiA 303-1 DB9 connector (i.e. the default connector for our [CAN loggers](#)), optionally with a [CAN bus extension cable](#)
- **Contactless:** If no connector is available, a typical solution is to use a [contactless CAN reader](#) - e.g. the [CANCrocodile](#). This lets you log data directly from the raw CAN twisted wiring harness, without direct connection to the CAN bus (often useful for warranty purposes)
- **Other:** In practice, countless other connectors are used and often you'll need to create a custom CAN bus adapter - here a [generic open-wire adapter](#) is useful

When you've identified the right connector and verified the pin-out, you can connect your CAN logger to start recording data. For the CANedge/CLX000, the CAN baud rate is auto-detected and the device will start logging raw CAN data immediately.

Example: Raw CAN sample data (J1939)

You can optionally download raw OBD2 and J1939 samples from the [CANedge2](#) in our [intro docs](#). You can e.g. load this data in the free CAN bus decoder software tools. Data from the CANedge is recorded in the popular binary format, [MF4](#), but can be converted to any file format via our simple [MF4 converters](#) (e.g. to CSV, ASC, TRC, ...).

Below is a CSV example of raw CAN frames logged from a heavy-duty truck using the J1939 protocol. Notice that the CAN IDs and data bytes are in hexadecimal format:

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1578922367.777150;1;14FEF131;1;8;8;0;0;0;CFFFFF300FFFF30
1578922367.777750;1;10F01A01;1;8;8;0;0;0;2448FFFFFFFFFFFF
1578922367.778300;1;CF00400;1;8;8;0;0;0;107D82BD1200F482
1578922367.778900;1;14FF0121;1;8;8;0;0;0;FFFFFFFFFFFFCFF
...
```

Example: CANedge CAN logger

The [CANedge1](#) lets you easily record data from any CAN bus to an 8-32 GB SD card. Simply connect it to e.g. a car or truck to start logging - and decode the data via [free software/APIs](#). Further, the [CANedge2](#) adds WiFi, letting you auto-transfer data to your own server - and update devices over-the-air.



How to decode raw CAN data to 'physical values'

If you review the raw CAN bus data sample above, you will probably notice something: Raw CAN bus data is not human-readable.

To interpret it, you need to decode the CAN frames into scaled engineering values aka physical values (km/h, degC, ...).

Below we show step-by-step how this works.



Extracting CAN signals from raw CAN frames

Each CAN frame on the bus contains a number of CAN signals (parameters) within the CAN databytes. For example, a CAN frame with a specific CAN ID may carry data for e.g. 2 CAN signals.



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

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

To extract a CAN signal, you 'carve out' the relevant bits, take the decimal value and perform a linear scaling:
 $\text{physical_value} = \text{offset} + \text{scale} * \text{raw_value_decimal}$

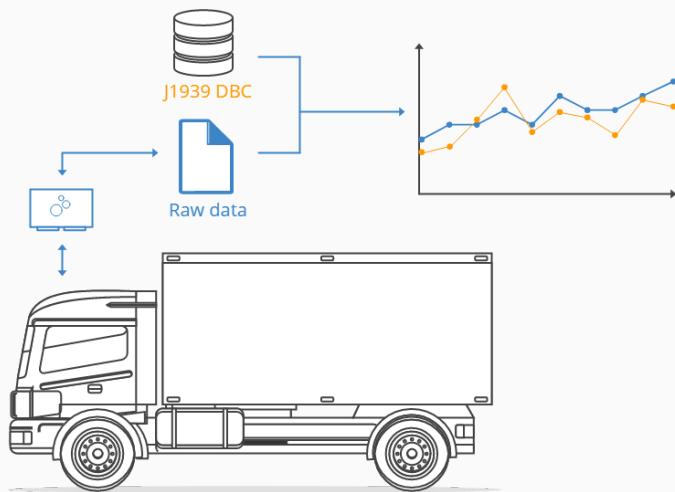
The challenge of proprietary CAN data

Most often, the CAN bus "decoding rules" are proprietary and not easily available (except to the OEM, i.e. Original Equipment Manufacturer). There are a number of solutions to this when you're not the OEM:

- **Record J1939 data:** If you're logging raw data from heavy duty vehicles (trucks, tractors, ...), you're typically recording J1939 data. This is standardized across brands - and you can use our [J1939 DBC file](#) to decode data. See also our [J1939 data logger intro](#)
- **Record OBD2/UDS data:** If you need to log data from cars, you can typically request OBD2/UDS data, which is a standardized protocol across cars. For details see our [OBD2 data logger intro](#) and our free [OBD2 DBC file](#)
- **Use public DBC files:** For cars, online databases exist where others have reverse engineered proprietary some of the CAN data. We keep a list of such databases in our [DBC file intro](#)
- **Reverse engineer data:** You can also attempt to reverse engineer data yourself by using a [CAN bus sniffer](#), though it can be time consuming and difficult
- **Use sensor modules:** In some cases you may need sensor data that is not available on the CAN bus (or which is difficult to reverse engineer). Here, sensor-to-CAN modules like the [CANmod](#) series can be used. You can integrate such modules with your CAN bus, or use them as add-ons for your CAN logger to add data such as [GNSS/IMU](#) or [temperature](#) data
- **Partner with the OEM:** In some cases the OEM will provide decoding rules as part of the CAN bus system technical specs. In other cases you may be able to get the information through e.g. a partnership

CAN database files (DBC) - J1939 example

In some cases, conversion rules are standard across manufacturers - e.g. in [the J1939 protocol for heavy-duty](#). This means that you can use the J1939 parameter conversion rules on practically any heavy-duty vehicle to convert a large share of your data. To make this practical, you need a format for storing the conversion rules. Here, the CAN database ([DBC](#)) format is the industry standard - and is supported by most CAN bus decoder software incl. the supporting tools for our CAN loggers, [asammcf and CANvas](#). We also offer a low cost J1939 DBC file, which you can purchase as a digital download. With this, you can get quickly from raw J1939 data to human-readable form. [Learn more!](#)



Example: Decoded CAN sample data (physical values)

To illustrate how you can extract CAN signals from raw CAN data frames, we include below the previous J1939 sample data - but now decoded via a [J1939 DBC file](#) using the [asammcf GUI tool](#).

As evident, the result is timeseries data with parameters like oil temperature, engine speed, GPS, fuel rate and speed:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFuelRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
...
```

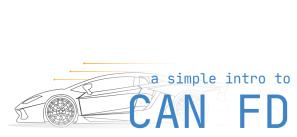
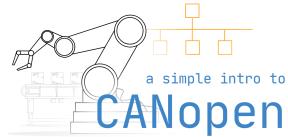
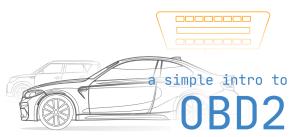
For more on logging J1939 data, see our [J1939 data logger](#) and [mining telematics](#) articles. You can also learn how to analyze/visualize your CAN data via the free [asammcf GUI tool](#) or [telematics dashboards](#).

What is the link between CAN, J1939, OBD2, CANopen ...?

The Controller Area Network provides the basis for communication - but not a lot more.

For example, the CAN standard does not specify how to handle messages larger than 8 bytes - or how to decode the raw data. Therefore a set of **standardized protocols** exist to further specify how data is communicated between CAN nodes of a given network.

Some of the most common standards include SAE J1939, OBD2 and CANopen. Further, these higher-layer protocols will increasingly be based on the 'next generation' of CAN, CAN FD (e.g. CANopen FD and J1939-17/22).



SAE J1939

J1939 is the standard in-vehicle network for heavy-duty vehicles (e.g. trucks & buses). J1939 parameters (e.g. RPM, speed, ...) are identified by a suspect parameter number (SPN), which are grouped in parameter groups identified by a PG number (PGN).

[j1939 intro](#)

[j1939 telematics](#)

OBD2

On-board diagnostics (OBD, ISO 15765) is a self-diagnostic and reporting capability that e.g. mechanics use to identify car issues. OBD2 specifies diagnostic trouble codes (DTCs) and real-time data (e.g. speed, RPM), which can be recorded via OBD2 loggers.

[obd2 intro](#)

[obd2 logging](#)

CANopen

CANopen is used widely in embedded control applications, incl. e.g. industrial automation. It is based on CAN, meaning that a [CAN bus data logger](#) is also able to log CANopen data. This is key in e.g. machine diagnostics or optimizing production.

[canopen intro](#)

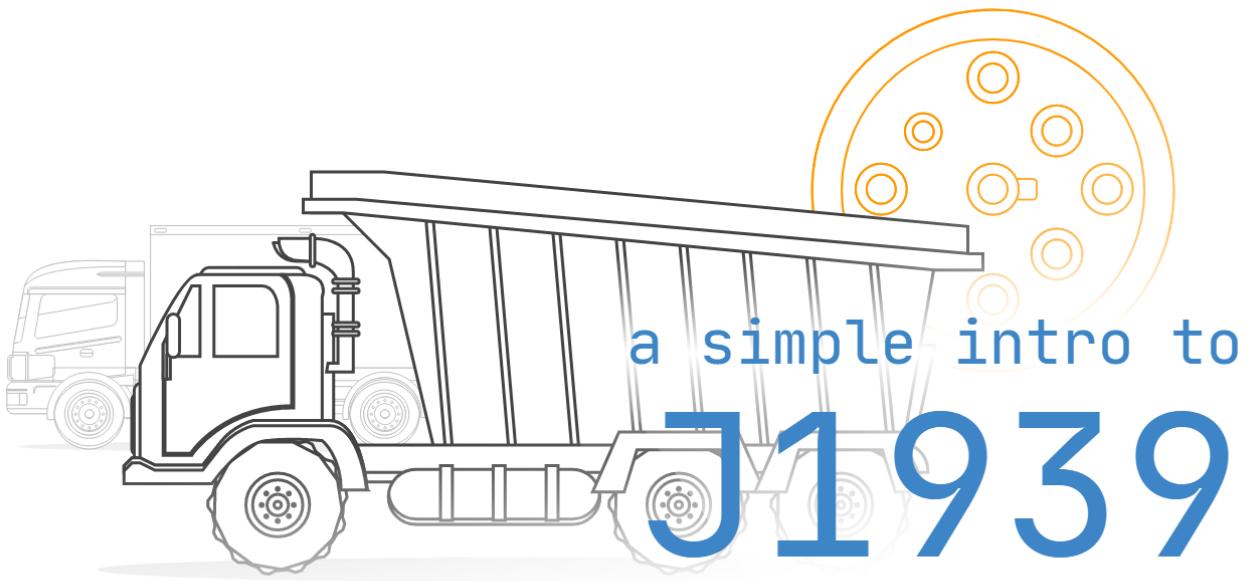
[canopen logger](#)

CAN FD

CAN bus with flexible data-rate (CAN FD) is an extension of the Classical CAN data link layer. It increases the payload from 8 to 64 bytes and allows for a higher data bit rate, dependent on the CAN transceiver. This enables increasingly data-intensive use cases like EVs.

[can fd intro](#)

[can fd logger](#)



J1939 Explained - A Simple Intro

In this guide we introduce the J1939 protocol basics incl. PGNs and SPNs. This is a practical intro so you will also learn how to decode J1939 data via DBC files, how J1939 logging works, key use cases and practical tips.

What is J1939?

In short, SAE J1939 is a set of standards that define how [ECUs](#) communicate via the [CAN bus](#) in heavy-duty vehicles. As explained in our [CAN bus intro](#), most vehicles today use the [Controller Area Network](#) (CAN) for ECU communication. However, CAN bus only provides a "basis" for communication (like a telephone) - not a "language" for conversation.

In most heavy-duty vehicles, this language is the SAE J1939 standard defined by the [Society of Automotive Engineers](#) (SAE). In more technical terms, J1939 provides a [higher layer protocol](#) (HLP) based on CAN as the "physical layer". What does that mean, though?

One standard across heavy-duty vehicles

In simple terms, J1939 offers a standardized method for communication across ECUs, or in other words: J1939 provides a common language across manufacturers. In contrast, e.g. cars use proprietary [OEM](#) specific protocols.

J1939 application examples

Heavy-duty vehicles (e.g. trucks and buses) is one of the most well-known applications. However, several other key industries leverage SAE J1939 today either directly or via derived standards (e.g. [ISO 11783](#), [MilCAN](#), [NMEA 2000](#), [FMS](#)):

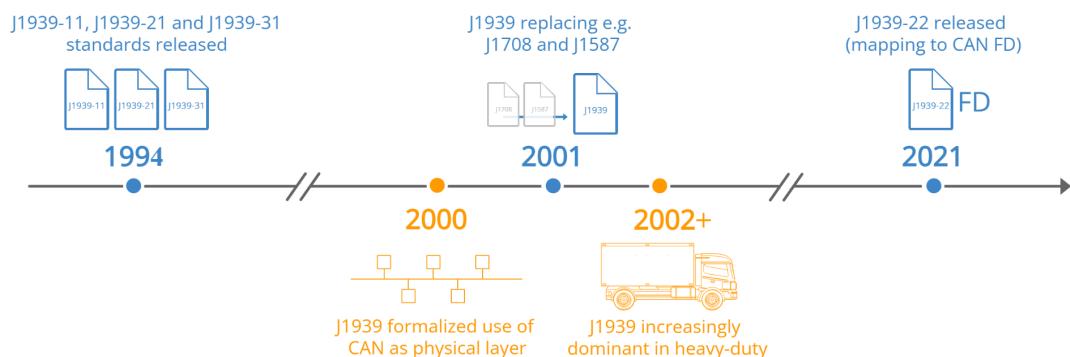
- [Forestry machines](#) (e.g. delimiters, forwarders, skidders)
- [Mining vehicles](#) (e.g. bulldozers, draglines, excavators, ...)
- [Military](#) vehicles (e.g. tanks, transport vehicles, ...)
- [Agriculture](#) (e.g. tractors, harvesters, ...)

- Construction (e.g. mobile hydraulics, cranes, ...)
- Fire & Rescue (e.g. ambulances, fire trucks, ...)
- Other (e.g. [ships](#), pumping, [e-buses](#), power generation, ...)

J1939 history & future

History

- 1994: First docs were released ([J1939-11](#), [J1939-21](#), [J1939-31](#))
- 2000: The initial top level document was published
- 2000: CAN formally included as part of J1939 standard
- 2001: J1939 starts replacing former standards SAE J1708/J1587



Future

With the rise of [heavy-duty telematics](#), J1939 will increasingly play a role in the market for connected vehicles. In turn, this will increase the need for [secure J1939 IoT loggers](#). In parallel, OEMs will increasingly shift from Classical CAN to [CAN FD](#) as part of the transition to [J1939 with flexible data-rate](#). In turn, this will increase the need for [J1939 FD data loggers](#).

"The market for in-vehicle connectivity - the hardware and services bringing all kinds of new functionality to drivers and fleet owners - is expected to reach EUR 120 billion by 2020."

- Boston Consulting Group, [Connected Vehicles and the Road to Revenue](#)

4 key characteristics of J1939

The J1939 protocol has a set of defining characteristics outlined below:



250K baud rate & 29-bit extended ID

The J1939 baud rate is typically 250K (though recently with support for 500K) - and the identifier is extended 29-bit (CAN 2.0B)

Broadcast + on-request data

Most J1939 messages are broadcast on the CAN-bus, though some data is only available by requesting the data via the CAN bus

PGN identifiers & SPN parameters

J1939 messages are identified by 18-bit Parameter Group Numbers (PGN), while J1939 signals are called Suspect Parameter Numbers (SPN)

Multibyte variables & Multi-packets

Multibyte variables are sent least significant byte first (Intel byte order). PGNs with up to 1785 bytes are supported via J1939 transport protocol

Additional J1939 characteristics

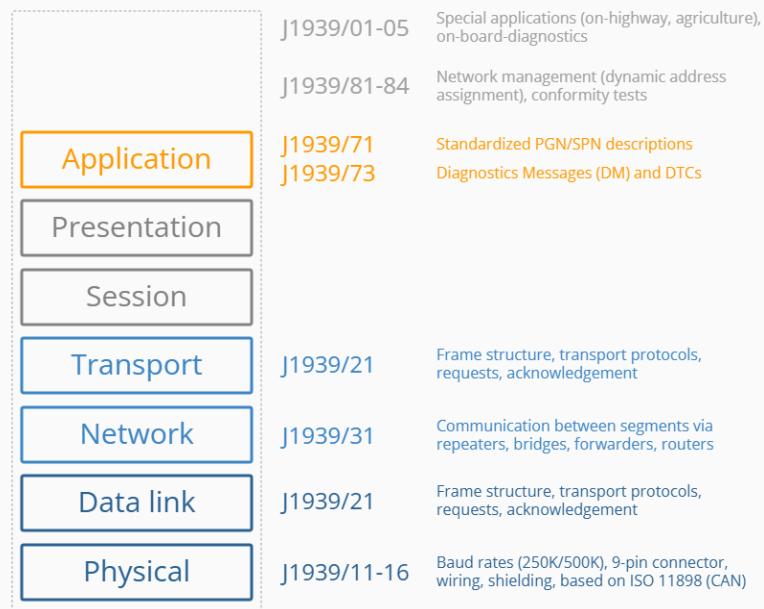
Below are a set of additional characteristics of the J1939 protocol:

- Reserved: J1939 includes a large range of standard PGNs, though PGNs 00FF00 through 00FFFF are reserved for proprietary use
- Special Values: A data byte of 0xFF (255) reflects N/A data, while 0xFE (254) reflects an error
- J1939 address claim: The SAE J1939 standard defines a procedure for assigning source addresses to J1939 ECUs after network initialization via an 8-bit address in a dynamic way

Technical: J1939 'higher layer protocol' explained

J1939 is based on CAN, which provides the basic "[physical layer](#)" and "[data link layer](#)", the lowest layers in the [OSI model](#). Basically, CAN allows the communication of small packets on the CAN bus, but not a lot more than that. Here, J1939 serves as a higher layer protocol on top, enabling more complex communication.

7 layer OSI model | J1939 standards



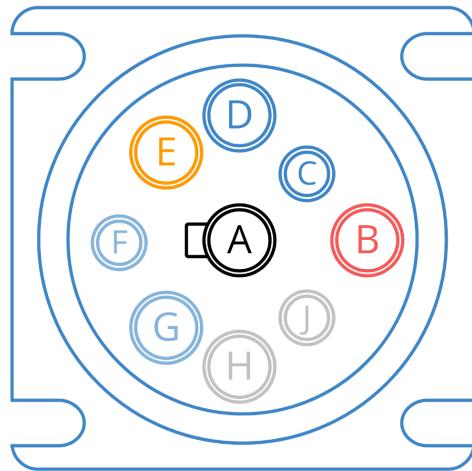
A higher layer protocol enables communication across the large complex networks of e.g. vehicle manufacturers.

For example, the SAE J1939 protocol specifies how to handle "multi-packet messages", i.e. when data larger than 8 bytes needs to be transferred. Similarly, it specifies how data is to be converted into human-readable data. It does so by providing a family of standards. For example, [J1939-71](#) is a document detailing the information required to convert a large set of cross-manufacturer standardized J1939 messages into human-readable data (more on this below). Many other CAN based higher layer protocols exist, e.g. [CANopen](#), [DeviceNet](#), [Unified Diagnostic Services](#). These typically offer some level of standardization within their respective industries - though all of them can be extended by manufacturers.

In comparison, the aforementioned passenger cars have unique standards per manufacturer. In other words, you can use the same J1939 database file to convert e.g. engine speed across two trucks from different manufacturers - but you cannot e.g. read the data from an Audi A4 using the same IDs & scaling parameters as for a Peugeot 207.

The J1939 connector (9-pin)

The J1939-13 standard specifies the 'off-board diagnostic connector' - also known as the J1939 connector or 9-pin deutsch connector. This is a standardized method for interfacing with the J1939 network of most heavy duty vehicles - see the illustration for the J1939 connector pinout.



- A** Ground
- B** Battery power
- C** CAN 1 H
- D** CAN 1 L
- E** CAN shield
- F** J1708 (+) / CAN 2 H
- G** J1708 (-) / CAN 2 L
- H** OEM specific
- J** OEM specific

Type 1 (black): CAN 1 = 250K

Type 2 (green): CAN 1 = 500K

Black type 1 vs green type 2

Note that the J1939 deutsch connector comes in two variants: The original black connector (type 1) and the newer green connector (type 2), which started getting rolled out around 2013-14.

J1939 connector (9-pin deutsch)



[J1939 type 2 female connectors](#) are physically backwards compatible, while type 1 female connectors only fit type 1 male sockets. The type 2 connector was designed for the SAE J1939-14 standard, which adds support for 500K bit rates. The purpose of "blocking" type 1 connectors is to ensure that older hardware (presumably using 250K bit rates) is not connected to type 2 500K bit rate J1939 networks. Specifically, the physical block is through a smaller hole for pin F in the type 2 male connectors. See also the example of a DB9-J1939 adapter cable (type 2).



Multiple J1939 networks

As evident, the J1939 deutsch connector provides access to the J1939 network through pins C (CAN high) and D (CAN low). This makes it easy to interface with the J1939 network across most heavy duty vehicles.

In some cases, however, you may also be able to access a secondary J1939 network through pins F and G or pins H and J (with H being CAN H and J being CAN L).

Many of today's heavy duty vehicles have 2 or more parallel CAN bus networks and in some cases at least two of these will be available through the same J1939 connector. This also means that you will not necessarily have gained access to all the available J1939 data if you've only attempted to interface through the 'standard' pins C and D.

Other heavy duty connectors

While the J1939 deutsch connector is the most common way to interface with the J1939 network of heavy duty vehicles, other connectors of course exist. Below are some examples:

- J1939 Backbone Connector: This 3-pin deutsch connector provides pins for CAN H/L a CAN shield (no power/ground)
- CAT connector: The [Caterpillar industrial connector](#) is a grey 9-pin deutsch connector. However, the pin-out differs from the J1939 connector (A: Power, B: Ground, F: CAN L, G: CAN H) and the connector physically blocks access from standard type 1 and 2 J1939 connectors
- OBD2 type B connector: The [type B OBD2 connector](#) (SAE J1962) in heavy duty vehicles sometimes provide direct access to the J1939 network through pins 6 and 14
- Volvo 2013 OBD2 connector: This variant matches the type B OBD2 connector, but also adds access to the J1939 high via pin 3 and J1939 low via pin 11

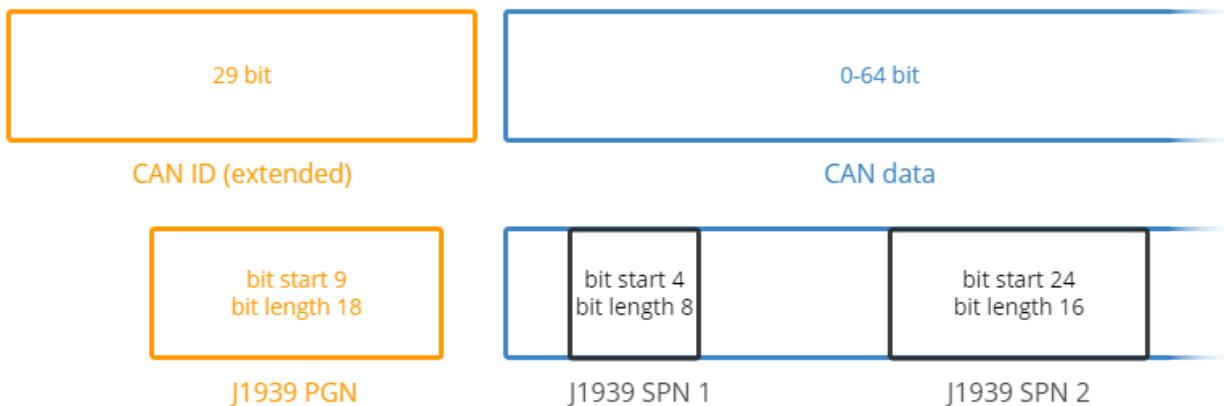
The J1939 PGN and SPN

In the following section we explain the J1939 PGNs and SPNs.

Parameter Group Number (PGN)

The J1939 PGN comprises an 18-bit subset of the 29-bit extended CAN ID. In simple terms, the PGN serves as a unique frame identifier within the J1939 standard. For example, you can look this up in the [J1939-71](#) standard documentation, which lists PGNs/SPNs.

J1939 message (PGN & SPNs)



Example: J1939 PGN 61444 (EEC1)

Assume you recorded a J1939 message with HEX ID 0CF00401. Here, the PGN starts at bit 9, with length 18 (indexed from 1). The resulting PGN is 0F004 or in decimal 61444. Looking this up in the SAE J1939-71 documentation, you will find that it is the "Electronic Engine Controller 1 - EEC1". Further, the document will have details on the PGN including priority, transmission rate and a list of the associated SPNs - cf. the illustration. For this PGN, there are seven SPNs (e.g. Engine Speed, RPM), each of which can be looked up in the J1939-71 documentation for further details.

PGN61444 - Electronic Engine Controller 1 - EEC1

Transmission Repetition Rate: Engine speed dependent

Data Length: 8 bytes

Data Page: 0

PDU Format: 240

PDU Specific: 4

Default Priority: 3

Parameter Group Number: 61444 (0x00F004)

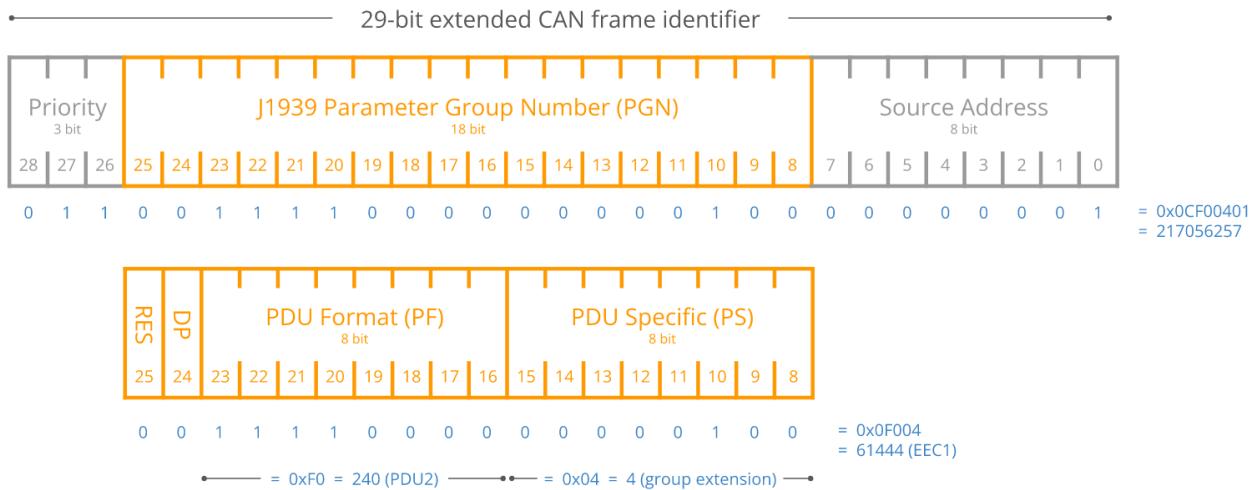
Bit Start/Byte	Length	SPN ID	SPN Description
1.1	4 bits	899	Engine Torque Mode
2	1 byte	512	Driver's Demand Engine - % Torque
3	1 byte	513	Actual Engine - Percent Torque
4-5	2 bytes	190	Engine Speed
6	1 byte	1483	SA of Controlling Device for Engine Control
7.1	4 bits	1675	Engine Starter Mode
8	1 byte	2432	Engine Demand - Percent Torque

Detailed breakdown of the J1939 PGN

Let's look at the CAN ID to PGN transition in detail. Specifically, the 29 bit CAN ID comprises the Priority (3 bits), the J1939 PGN (18 bits) and the Source Address (8 bits). In turn, the PGN can be split into the Reserved Bit (1 bit), Data Page (1 bit), PDU format (8 bit) and PDU Specific (8 bit).

The detailed PGN illustration also includes example values for each field in binary, decimal and hexadecimal form.

To learn more about the transition from 29 bit CAN ID to 18 bit J1939 PGN, see also our online [CAN ID to J1939 PGN converter](#). The converter also includes a full J1939 PGN list for PGNs included in our [J1939 DBC file](#).



blue: Example values

RES: Reserved | DP: Data Page | PDU: Protocol Data Unit (message format)

PF < 240: Message is PDU1 (addressable message, PS contains destination address)
PF >= 240: Message is PDU2 (broadcast message, PS contains group extension)

Suspect Parameter Number (SPN)

The J1939 SPN serves as the identifier for the CAN signals (parameters) contained in the databytes. SPNs are grouped by PGNs and can be described in terms of their bit start position, bit length, scale, offset and unit - information required to extract and scale the SPN data to physical values.



Example: Extracting J1939 SPN 190 (Engine Speed)

Assume you have recorded a raw J1939 frame as below:

CAN ID	Data bytes
0CF00401	FF FF FF 68 13 FF FF FF

By [converting the CAN ID to the J1939 PGN](#) you identify that this is the PGN 61444 from before. From the J1939-71 document, you observe that one of the SPNs within this PGN is Engine Speed (SPN 190) with details as in the illustration below.

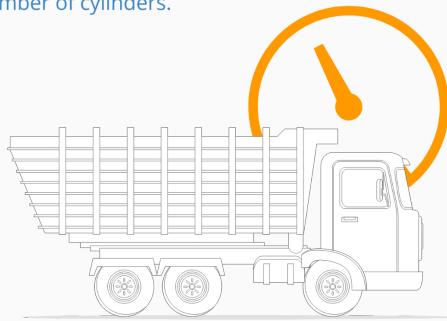
Using these details, it is possible to extract the Engine Speed physical value data e.g. for plot purposes. To do so, note from the SPN info that the relevant data is in bytes 4 and 5, i.e. the HEX data bytes 68 and 13. Taking the decimal form of the HEX value 1368 (Intel byte order), we get 4968 in decimal. To arrive at the RPM, we conduct a scaling of this value using the offset 0 and the scale 0.125 RPM/bit. The physical value (aka scaled engineering value) is 621 RPM.

Note how some data bytes in the above are FF or 255 decimal, i.e. not available. While the PGN may theoretically support SPNs in this range, the FF padding means that this particular application does not support these parameters.

J1939 SPN 190 | Engine Speed

Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.

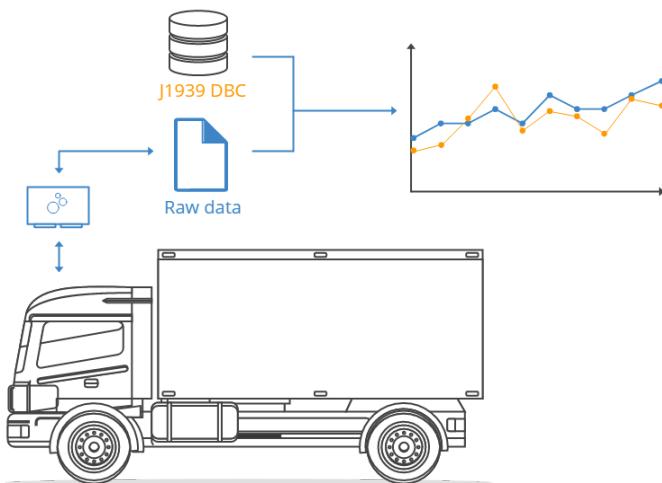
Bit start: 24
Length: 2 bytes
Scale: 0.125
Offset: 0
Unit: rpm
min-max: 0-8031.875
PGN reference: 61444



In practice, you will not 'PDF-lookup' rules for J1939 data - instead, this info is stored in a [CAN database file](#) (DBC).

Example: J1939 DBC file

A J1939 DBC file can be used to decode data across most heavy-duty vehicles. For example, raw J1939 data can be recorded with a [CAN bus data logger](#) and analyzed in a CAN software tool that supports DBC conversion (e.g. [asammcf](#)). This will typically result in a conversion of 40-60% of the vehicle data - with the rest being OEM specific proprietary data that requires [reverse engineering](#).



J1939 truck sample data: Raw & physical values

Below we illustrate what real J1939 data looks like. The 'raw' J1939 data was recorded from a heavy duty truck using a [CANedge2](#), while the 'physical values' reflect the output after decoding the raw data via the free [asammcf software](#) and the [J1939 DBC](#).

Sample: Raw J1939 truck data (CSV)

Data from the CANedge is recorded in a standardized binary format, [MDF4](#), which can be converted to any file format via our [MDF4 converters](#) (e.g. to CSV, ASC, TRC, ...). Below is a CSV version of the raw J1939 frames. Notice that the CAN IDs and data bytes are in hexadecimal format:

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1578922367.777150;1;14FEF131;1;8;8;0;0;0;CFFFFFFF300FFFFF30
1578922367.777750;1;10F01A01;1;8;8;0;0;0;2448FFFFFFFFFFFF
1578922367.778300;1;CF00400;1;8;8;0;0;0;107D82BD1200F482
1578922367.778900;1;14FF0121;1;8;8;0;0;0;FFFFFFF7FFFCCF
1578922367.779500;1;18F0000F;1;8;8;0;0;0;007DFFFF0F7DFFFF
1578922367.780050;1;18FFA03D;1;8;8;0;0;0;2228240019001AFF
1578922367.780600;1;10FCFD01;1;8;8;0;0;0;FFFFFFF1623FFFF
1578922367.781200;1;18FD9401;1;8;8;0;0;0;A835FFFFA9168F03
1578922367.781750;1;18FDA101;1;8;8;0;0;0;1224FFFFFFF00FF
1578922367.782350;1;18F00E3D;1;8;8;0;0;0;741DFFFFFFF7FFF
1578922367.782950;1;18F00F3D;1;8;8;0;0;0;B40FFFFFFF7FFF
1578922367.783500;1;10FDA301;1;8;8;0;0;0;FFFFFFF7FFF
```

...

You can optionally download full raw J1939 MDF4 samples from the [CANedge2](#) in our [intro docs](#). The sample data also includes a demo J1939 DBC so that you can replicate the conversion steps via [asammdf](#).

Sample: Decoded physical values J1939 truck data (CSV)

Once the raw J1939 data is decoded and exported, the result is timeseries data with parameters like oil temperature, engine speed, GPS, fuel rate and speed:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFuelRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.289050102+01:00,0,1527.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.299000025+01:00,0,1528.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.308300018+01:00,0,1526.86,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.309099913+01:00,0,1526.75,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.317199945+01:00,0,1526.45,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
...
```

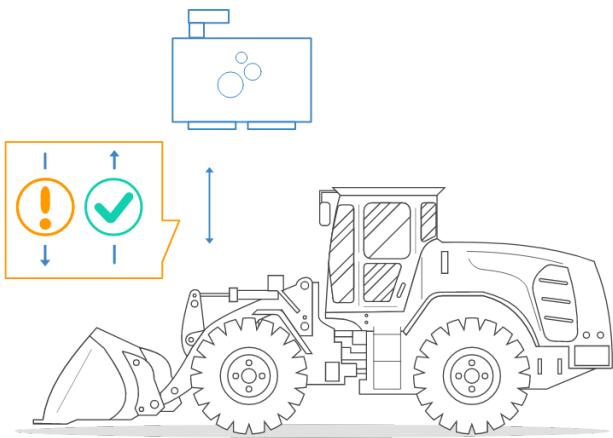
For more on logging J1939 data, see our [J1939 data logger](#) and [mining telematics](#) articles.

About the CANedge J1939 logger

The [CANedge](#) lets you easily record J1939 data to an 8-32 GB SD card. Simply connect it to e.g. a truck to start logging - and decode the data via [free software/APIs](#) and our [J1939 DBC](#). [Learn more](#).

J1939 request messages

Most J1939 messages are broadcast via the CAN bus, but some are only sent "on-request" (e.g. when polled by a [J1939 data logger](#)). On-request data often includes J1939 diagnostic trouble codes (DTCs), making it important in vehicle diagnostics. Below we briefly outline how it works:



Sending J1939 request messages

To send a J1939 request via the CAN bus, a special 'request message' is used (PGN 59904), which is the only J1939 message with only 3 bytes of data. It has priority 6, a variable transmit rate and can either be sent as a global or specific address request. The data bytes 1-3 should contain the requested PGN (Intel byte order). Examples of requested J1939 messages include the diagnostic messages (e.g. J1939 DM2).

A CAN bus data logger like the [CANedge](#) can be set up to send J1939 request messages - see e.g. our [CANedge Intro](#) for a detailed step-by-step guide.

J1939 code requests vs warranty compliance

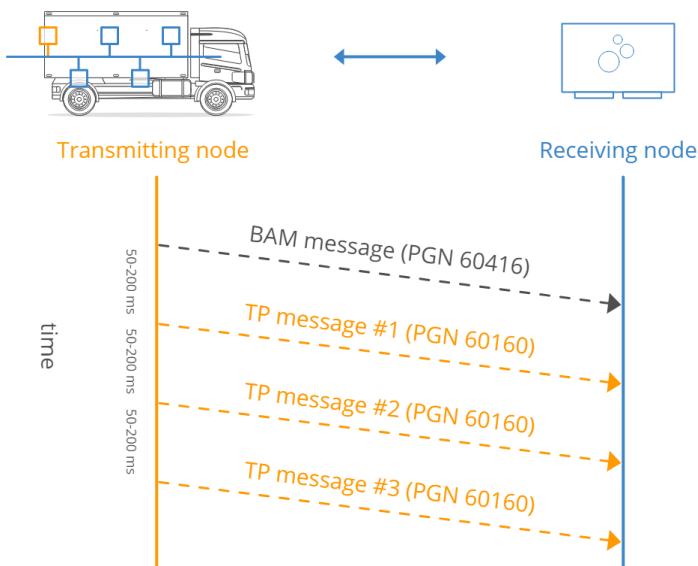
Sending request messages is typically key to requesting J1939 codes and thus J1939 diagnostics. One challenge, however, is that J1939 loggers are often required to have a contactless connection to the J1939 data link, meaning that they're unable to interact with the CAN bus and transmit J1939 request frames. The restriction is often related to warranty compliance as some vehicle manufacturers do not allow direct access by 3rd party devices via the J1939 connector.

In some cases, it is required that the J1939 analyzer is "physically" contactless, e.g. via a [CANcrocodile adapter](#). In other cases, it is sufficient that the J1939 logger operates in "configurable" silent mode. The latter makes it easier to perform ad hoc requests for J1939 fault codes, either via a manual configuration update or via an over-the-air update for the [CANedge2](#).

J1939 transport protocol (TP)

The previous PGN and SPN examples are based on J1939 messages with 8 data bytes. While these are most common, J1939 multi-frame messages also exist with >8 data bytes - sent via the J1939 transport protocol.

Below we outline how the J1939 transport protocol works, a practical J1939 TP data example and how to decode multi-frame J1939 messages via DBC files:



How does the J1939 transport protocol work?

The J1939 protocol specifies how to deconstruct, transfer and reassemble packets across multiple frames - a process referred to as the J1939 Transport Protocol (see [J1939-21](#)). Two types of the J1939 TP exist:

1. The Connection Mode (intended for a specific device)
2. The BAM (Broadcast Announce Message) which is intended for the entire network

For example, a transmitting ECU may send an initial BAM packet to set up a data transfer. The BAM specifies the PGN identifier for the multi-packet message as well as the number of data bytes and packets to be sent. It is then followed by up to 255 packets/frames of data. Each of the 255 packets use the first data byte to specify the sequence number (1 up to 255), followed by 7 bytes of data. The max number of bytes per multi-packet message is therefore 7 bytes x 255 = 1785 bytes.

The final packet contains at least one byte of data, followed by unused bytes set to FF. In the BAM type scenario, the time between messages is 50-200 ms. In post processing, a conversion software tool can reassemble the multiple entries of 7 data bytes into a single payload and handle it according to the multi-packet PGN and SPN specifications as found in e.g. a [J1939 DBC file](#).

A practical J1939 transport protocol example

Decoding J1939 multiframe data is more complex than decoding standard J1939 frames. To understand why, consider the below example of a J1939 transport protocol response, recorded with the [CANedge2](#):

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1605078459.438750;1;1CECFF00;1;8;8;0;0;0;20270006FFE3FE00
1605078459.498750;1;1CEBFF00;1;8;8;0;0;0;013011B2A041B240
1605078459.559750;1;1CEBFF00;1;8;8;0;0;0;021FB2102CB2603B
1605078459.618750;1;1CEBFF00;1;8;8;0;0;0;03B230430000D309
1605078459.678750;1;1CEBFF00;1;8;8;0;0;0;04C0441E37967DE1
1605078459.738750;1;1CEBFF00;1;8;8;0;0;0;05E02E7B02FFFF80
1605078459.799850;1;1CEBFF00;1;8;8;0;0;0;06E0FFFFFFF
```

The above sequence consists of two J1939 message types:

PGN 60416 - J1939 Transport Protocol BAM (Connection Management)

Data Length: 8 bytes
Default Priority: 7
Parameter Group Number: 60416 (0xEC00)

Byte	Description
1	Fixed at 32
2-3	Message size in bytes
4	Number of packets
5	Reserved (filled with FF)
6-8	PGN

PGN 60160 - J1939 Transport Protocol (Data Transfer)

Data Length: 8 bytes
Default Priority: 7
Parameter Group Number: 60160 (0xEB00)

Byte	Description
1	Sequence number (1 to 255)
2-8	Data payload (unused locations in the last frame are set to FF)

A J1939 BAM message with ID 1CECFF00 (PGN 60416 or EC00), which contains the response data length and J1939 PGN - and a J1939 data transfer messages with ID 1CEBFF00 (PGN 60160 or EB00). These contain the payload across multiple frames.

Below we break down the J1939 transport protocol example with focus on the data byte interpretation:

Timestamp (Epoch)	CAN ID	PGN (HEX)	PGN (DEC)	DataBytes	legend
1605078459.438750	1CECFF00	EC00	60416	20270006FFE3FE00	control byte (0x20=BAM)
1605078459.498750	1CEBFF00	EB00	60160	013011B2A041B240	#data bytes (0x0027=39)
1605078459.559750	1CEBFF00	EB00	60160	021FB2102CB2603B	#packets (0x06=6)
1605078459.618750	1CEBFF00	EB00	60160	03B230430000D309	reserved
1605078459.678750	1CEBFF00	EB00	60160	04C0441E37967DE1	J1939 PGN (0x00FEE3=65251)
1605078459.738750	1CEBFF00	EB00	60160	05E02E7B02FFFF80	sequence numbers
1605078459.799850	1CEBFF00	EB00	60160	06E0FFFFFFF	payload data (39 bytes)
					unused (3 bytes)
1605078459.438750	18FEE3FE	FEE3	65251	3011B2A041B240 ... E0FFFFFF	

Generally, a J1939 transport protocol response sequence can be processed as follows:

- Identify the BAM frame, indicating a new sequence being initiated (via the PGN 60416)
- Extract the J1939 PGN from bytes 6-8 of the BAM payload to use as the identifier of the new frame
- Construct the new data payload by concatenating bytes 2-8 of the data transfer frames (i.e. excl. the 1st byte)

Above, the last 3 bytes of the BAM equal E3FE00. When reordered, these equal the PGN FEE3 aka Engine Configuration 1 (EC1). Further, the payload is found by combining the the first 39 bytes across the 6 data transfer packets/frames.

Note: The last 3 data payload bytes in this practical example happen to be FF, yet we still include these in the payload as the BAM message specifies the data length to be 39. The final 3 FF bytes in the 6th packet are unused.

How to decode a J1939 transport protocol message

With the method explained above, we have created a 'constructed' J1939 data frame with a data length exceeding 8 bytes. This frame can be decoded using a J1939 DBC file, just like a regular J1939 data frame. For the PGN EC1, the [J1939 DBC](#) specifies a data length of 40 with signals defined for the full payload.

As such, once the J1939 software/API has reconstructed the multiframe response into a single J1939 frame, the DBC decoding can be done as usual. One minor tweak is that most J1939 DBC files expects that the raw log file of J1939 data will contain 29-bit CAN IDs (not 18-bit J1939 PGNs). As such, if the software embeds the reconstructed J1939 TP frame in the original raw data, it may need to convert the extracted J1939 PGN into a 29-bit CAN ID first. You can also see our [J1939 google sheet](#), which breaks down how a J1939 PGN can be converted to a 29-bit CAN ID.

J1939 TP data & Python API decoding

The CANedge lets you request and record J1939 transport protocol data. To decode the TP data, you can either convert the raw log files to another format (like Vector ASC), or you can use our [Python API](#). In our [api-examples](#) library on github, we provide a basic example of how J1939 transport protocol data can be reconstructed and DBC decoded, incl.

sample data. Since the CANedge [Python API](#) enables decoding of J1939 transport protocol data, J1939 signals from multiframe messages can e.g. be visualized in J1939 telematics dashboards.

Logging J1939 data - example use cases

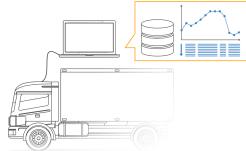
There are several common use cases for recording J1939 data:



Heavy duty fleet telematics

J1939 data from trucks, buses, tractors etc. can be used in fleet management to reduce costs or improve safety

[learn more](#)



Live stream diagnostics

By streaming decoded J1939 data to a PC, technicians can perform real-time J1939 diagnostics on vehicles

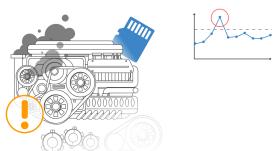
[learn more](#)



Predictive maintenance

Vehicles can be monitored via [WiFi CAN loggers](#) in the cloud to predict breakdowns based on the J1939 data

[learn more](#)



Heavy-duty vehicle blackbox

A [CAN logger](#) can serve as a 'blackbox' for heavy-duty vehicles, providing data for e.g. disputes or J1939 diagnostics

[learn more](#)

6 practical tips for J1939 data logging

Many of our end users work with J1939 logging in the field - and below we share 6 practical logging tips:

J1939 logger vs J1939 streaming interface

Standalone J1939 data loggers with SD cards are ideal for logging data from e.g. vehicle fleets over weeks or months. A WiFi J1939 logger also enables [telematics](#) use cases. In contrast, a [J1939 USB-PC interface](#) requires a PC to stream data from the CAN bus in real-time. This is e.g. useful for diagnostic purposes - or analysing physical events. The CLX000 enables both modes of operation, while the [CANedge2](#) is perfect for telematics.

Direct adapter cable vs contactless reading

To connect your CAN analyzer to a J1939 asset (e.g. a truck) you can typically use the 9-pin J1939 connector. We offer a DB9-J1939 adapter which fits the 9-pin deutsch connector found in many heavy duty vehicles. Alternatively, you may prefer to connect your CAN logger directly to the CAN bus via e.g. a CANCrocodile. This method uses induction to record data silently without cutting any CAN wiring.

WiFi vs. cellular (3G/4G) data upload

For vehicle fleet management & telematics you will typically upload the data via either WiFi or 3G/4G. The [CANedge2](#) lets you transfer data by connecting to a WiFi access point - which can both be a WLAN router or a 3G/4G hotspot. If you need data from a truck on-the-road, you can install the CANedge2 and use it to power a [3G/4G USB hotspot](#). The benefit to this is that you'll have continuous access to the device - unless it is out-of-coverage. However, in cases where

data only needs to be periodically uploaded an alternative can be to upload data via WLAN routers when the vehicles visit e.g. specific areas (garages, repair shops etc) - letting you reduce data transfer costs.

Software selection & J1939 DBC file

When logging or streaming J1939 data, software for post processing is key. In particular, the software should support DBC-based J1939 conversion to allow easy conversion to human-readable data. The free supporting [softwares/APIs](#) for our CAN loggers support this. For USB streaming, our free Wireshark plugin enables [live DBC conversion](#). Further, we offer a digital download [J1939 DBC](#) file in collaboration with SAE.

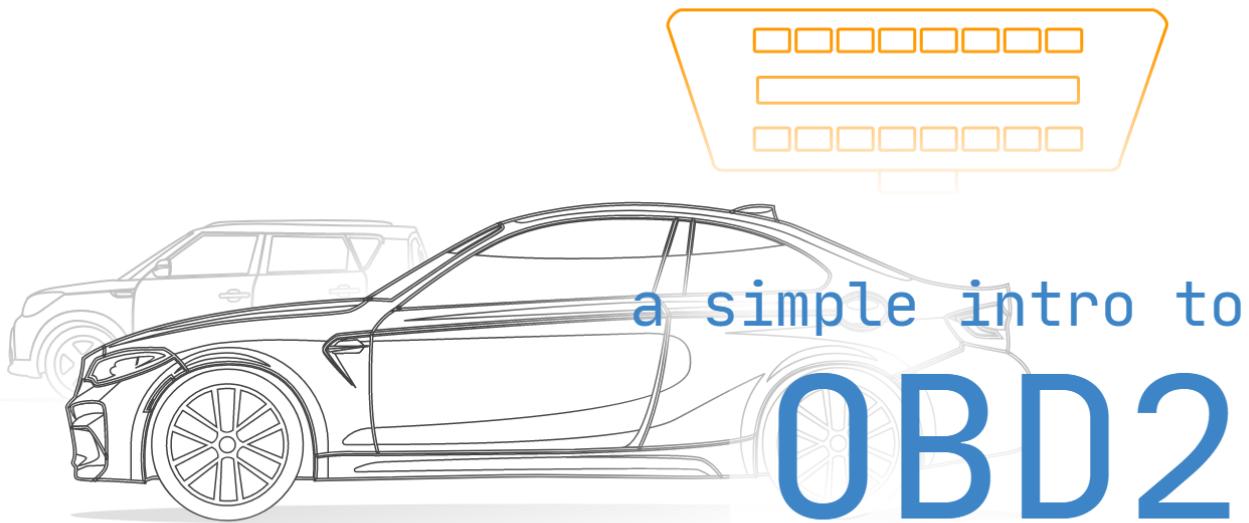
Consider the need for request PGNs

Some J1939 PGNs are only available on-request, meaning that you need to "poll" the CAN bus to log these. The CANedge and CLX000 are able to transmit custom CAN messages, which can be used to send periodic PGN requests. Note that this is not possible in "silent mode" (i.e. it is not possible if the logger is connected via e.g. a CANCrocodile).

Filter, compress and encrypt the data

To optimize your J1939 data logging, a number of advanced configurations can be helpful. In particular, the CANedge advanced filters and sampling rate options help optimize the amount of data logged - key for e.g. minimizing cellular bandwidth usage. Other options include silent mode and cyclical logging, with the latter enabling the logger to always prioritize the latest data (useful in e.g. blackbox logging).

Since J1939 is standardized, it is critical to encrypt your data 'at rest' (e.g. on an SD card) and 'in transit' (during upload). Not doing so exposes your data processing to various security risks, incl. GDPR/CCPA fines and loss of confidentiality and data integrity. For details on securing your J1939 data logging, see our [intro to secure CAN logging](#).

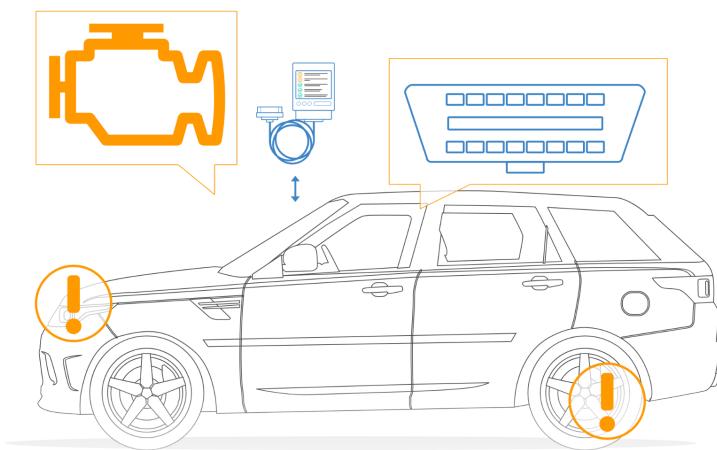


OBD2 Explained - A Simple Intro

In this guide we introduce the On Board Diagnostic (OBD2) protocol incl. the OBD2 connector, OBD2 parameter IDs (PID) and the link to CAN bus. This is a practical intro so you will also learn how to request and decode OBD2 data, key logging use cases and practical tips.

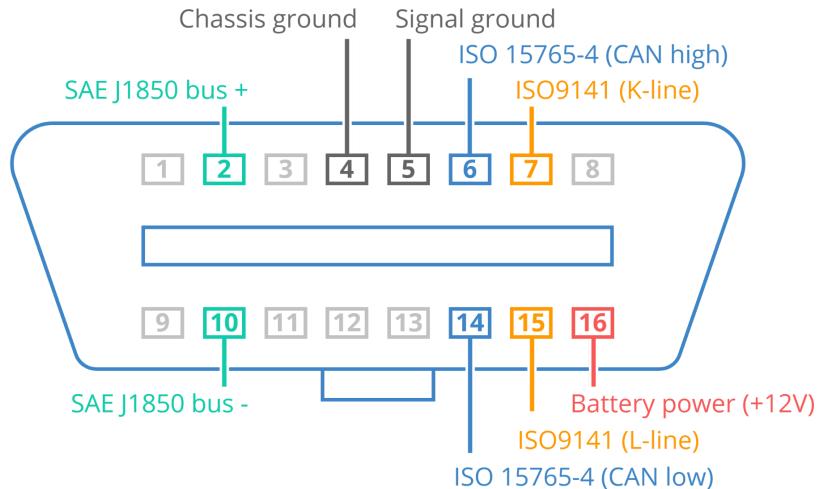
What is OBD2?

In short, OBD2 is your vehicle's built-in self-diagnostic system. You've probably encountered OBD2 already: Ever noticed the [malfunction indicator light](#) on your dashboard? That is your car telling you there is an issue. If you visit a mechanic, he will use an OBD2 scanner to diagnose the issue. To do so, he will connect the OBD2 reader to the [OBD2 16 pin connector](#) near the steering wheel. This lets him read OBD2 codes aka Diagnostic Trouble Codes (DTCs) to review and troubleshoot the issue.



The OBD2 connector

The OBD2 connector lets you access data from your car easily. The standard SAE J1962 specifies two female OBD2 16-pin connector types (A & B). In the illustration is an example of a Type A OBD2 pin connector (also sometimes referred to as the Data Link Connector, DLC).

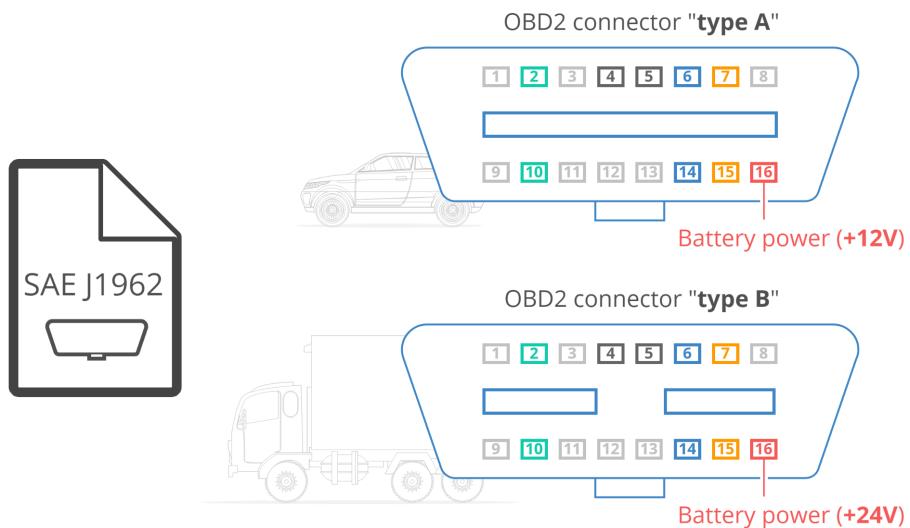


A few things to note:

- The OBD2 connector is near your steering wheel, but [may be hidden behind covers/panels](#)
- Pin 16 supplies battery power (often while the ignition is off)
- The OBD2 pinout depends on the communication protocol
- The most common protocol is CAN (via ISO 15765), meaning that pins 6 (CAN-H) and 14 (CAN-L) will typically be connected

OBD2 connector - type A vs. B

In practice, you may encounter both the type A and type B OBD2 connector. Typically, type A will be found in cars, while type B is common in medium and heavy duty vehicles.



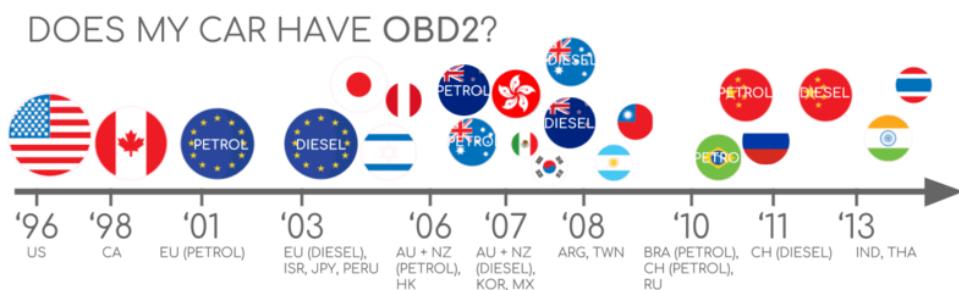
Type A & B connector differences

As evident from the illustration, the two types share similar OBD2 pinouts, but provide two different power supply outputs (12V for type A and 24V for type B). Often the baud rate will differ as well, with cars typically using 500K, while most heavy duty vehicles use 250K (more recently with support for 500K).

To help physically distinguish between the two types of OBD2 sockets, note that the type B OBD2 connector has an interrupted groove in the middle. As a result, a [type B OBD2 adapter cable](#) will be compatible with both types A and B, while a type A will not fit into a type B socket.

Does my car have OBD2?

In short: Probably! Almost all newer cars support OBD2 and most run on CAN (ISO 15765). For older cars, be aware that even if a 16 pin OBD2 connector is present, it [may still not support OBD2](#). One way to determine compliance is to identify where & when it was bought new:



Link between OBD2 and CAN bus

On board diagnostics, OBD2, is a '[higher layer protocol](#)' (like a language). CAN is a method for communication (like a phone). In particular, the OBD2 standard specifies the OBD2 connector, incl. a set of five protocols that it can run on (see below). Further, since 2008, CAN bus ([ISO 15765](#)) has been the mandatory protocol for OBD2 in all cars sold in the US.

What is the ISO 15765 standard?

ISO 15765 refers to a set of restrictions applied to the CAN standard (which is itself defined in [ISO 11898](#)). One might say that ISO 15765 is like "[CAN for cars](#)". In particular, ISO 15765-4 describes the physical, data link layer and network layers, seeking to standardize the CAN bus interface for external test equipment.

ISO 15765-2 in turn describes the transport layer (ISO TP) for sending CAN frames with payloads that exceed 8 bytes. This sub standard is also sometimes referred to as Diagnostic Communication over CAN (or DoCAN). See also the 7 layer OSI model illustration. OBD2 can also be compared to other higher layer protocols (e.g. [J1939](#), [CANopen](#)).

7 layer OSI model

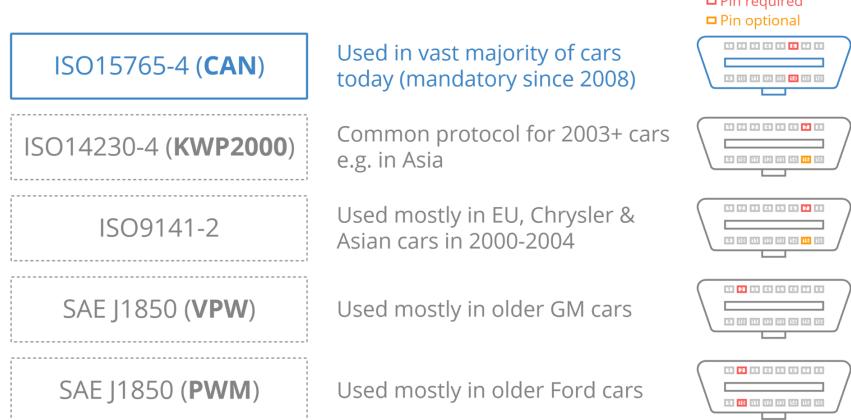
Application	ISO 14229-1 (UDS) ISO 15765-3 (UDS) SAE J1979 / ISO 15031
Presentation	
Session	ISO 15765-3 (UDS)
Transport	ISO 15765-2 (ISO-TP)
Network	ISO 15765-4 ('OBD via CAN') ISO 15765-2 (ISO-TP)
Data link	ISO 15765-4 ('OBD via CAN') ISO 11898-1 (CAN)
Physical	ISO 15765-4 ('OBD via CAN') ISO 11898-2 (CAN)

The five OBD2 protocols

As explained above, CAN bus today serves as the basis for OBD2 communication in the vast majority of cars through ISO 15765. However, if you're inspecting an older car (pre 2008), it is useful to know the other four protocols that have been used as basis for OBD2. Note also the pinouts, which can be used to determine which protocol may be used in your car.

- ISO 15765 (CAN bus): Mandatory in US cars since 2008 and is today used in the vast majority of cars
- ISO14230-4 (KWP2000): The Keyword Protocol 2000 was a common protocol for 2003+ cars in e.g. Asia
- ISO9141-2: Used in EU, Chrysler & Asian cars in 2000-04
- SAE J1850 (VPW): Used mostly in older GM cars
- SAE J1850 (PWM): Used mostly in older Ford cars

The five OBD2 compliant signal protocols

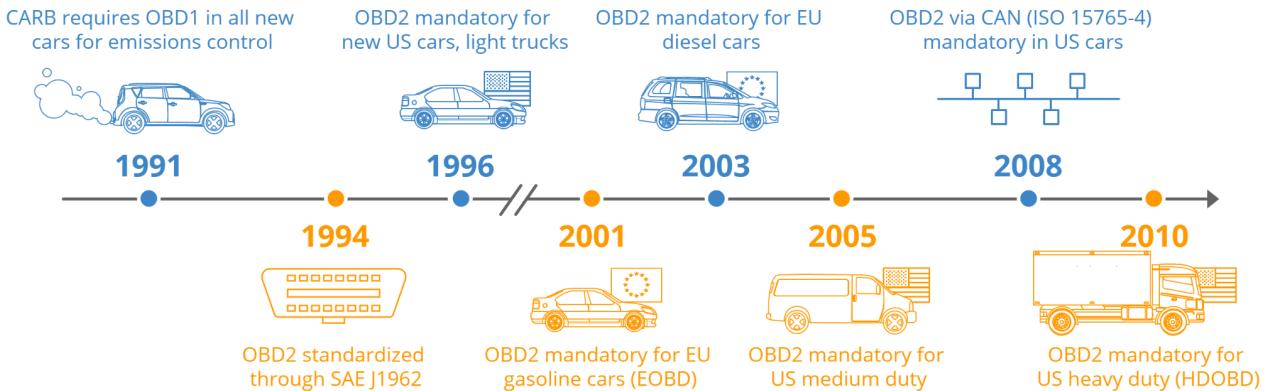


OBD2 history & future

History

OBD2 originates from California where the [California Air Resources Board](#) (CARB) required OBD in all new cars from 1991+ for emission control purposes. The OBD2 standard was recommended by the [Society of Automotive Engineers](#) (SAE) and standardized DTCs and the OBD connector across manufacturers ([SAE J1962](#)). From there, the OBD2 standard was rolled out step-by-step:

- 1996: OBD2 made mandatory in USA for cars/[light trucks](#)
- 2001: Required in EU for gasoline cars
- 2003: Required in EU also for diesel cars (EOBD)
- 2005: OBD2 was required in US for [medium duty vehicles](#)
- 2008: US cars must use [ISO 15765-4](#) (CAN) as OBD2 basis
- 2010: Finally, OBD2 was required in US heavy duty vehicles



Future

OBD2 is here to stay - but in what form? Two potential routes may radically change OBD2:

OBD3/OBD-III - wireless emission testing

In today's world of connected cars, OBD2 tests can seem cumbersome: Manually doing emission control checks is time-consuming and expensive. The solution? OBD3 - adding telematics to all cars. Basically, OBD3 adds a small radio transponder (as in e.g. bridge tolls) to all cars. Using this, the car [vehicle identification number](#) (VIN) and DTCs can be sent via WiFi to a central server for checks.

Many devices today already facilitate transfer of CAN or OBD2 data via WiFi/cellular - e.g. the [CANedge2](#) WiFi CAN logger. This saves cost and is convenient, but it is also politically a challenge due to surveillance concerns.

Eliminating 3rd party OBD2 services

The OBD2 protocol was originally designed for stationary emission controls. Yet, today OBD2 is used extensively for generating real-time data by 3rd parties - via [OBD2 dongles](#), [CAN loggers](#) etc. However, the [German car industry is looking to change this](#):

"OBD has been designed to service cars in repair shops. In no way has it been intended to allow third parties to build a form of data-driven economy on the access through this interface"

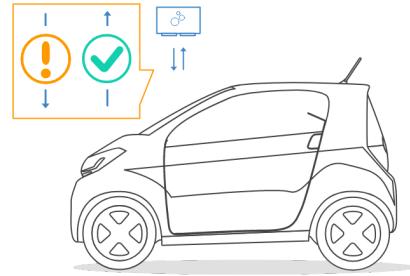
- Christoph Grote, SVP Electronics, BMW (2017)

The proposal is to "turn off" the OBD2 functionality while driving - and instead collect the data in a central server. This would effectively put the manufacturers in control of the automotive 'big data'. The argumentation is based in security (e.g. removing the [risk of car hacking](#)), though [many see it as a commercial move](#). Whether this becomes a real trend is to be seen - but it may truly disrupt the market for OBD2 3rd party services.

OBD2 parameter IDs (PID)

Why should you care about OBD2 data? Mechanics obviously care about OBD2 DTCs (maybe you do too), while regulatory entities need OBD2 to control emission. But the OBD2 protocol also supports a broad range of standard parameter IDs (PIPs) that can be logged across most cars.

This means that you can easily get [human-readable OBD2 data](#) from your car on speed, RPM, throttle position and more. In other words, OBD2 lets you analyze data from your car easily - in contrast to the OEM specific proprietary raw CAN data.



Decoding OBD2 vs CAN bus data

In principle it is simple to log the raw CAN frames from your car. If you e.g. connect a [CAN logger](#) to the OBD2 connector, you'll start logging broadcasted CAN bus data out-the-box. However, the raw CAN messages need to be decoded via a [database of conversion rules \(DBC\)](#) and a suitable CAN software that supports DBC decoding (like e.g. [asammdf](#)). The challenge is that these CAN DBC files are typically proprietary, making the raw CAN data unreadable unless you're the automotive OEM.

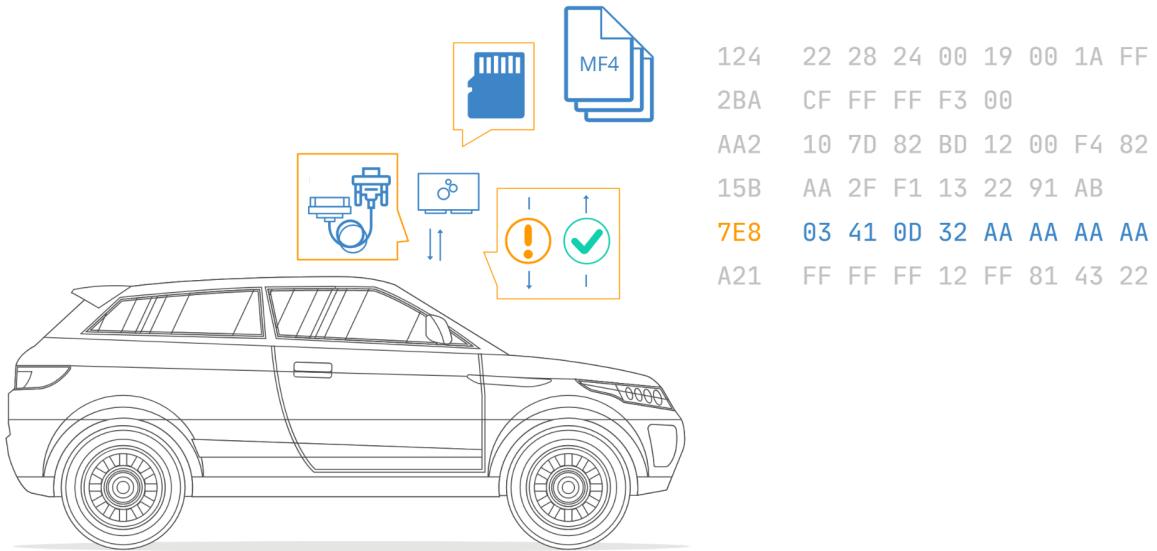
Car hackers may try to [reverse engineer](#) the rules, though this can be difficult. CAN is, however, still the only method to get "full access" to your car data - while OBD2 only provides access to a limited subset of data.

How to log OBD2 data?

OBD2 data logging works as follows:

- You connect an [OBD2 logger](#) to the OBD2 connector
- Using the tool, you send 'request frames' via CAN
- The relevant ECUs send 'response frames' via CAN
- Decode the raw OBD2 responses via e.g. an [OBD2 DBC](#)

In other words, a [CAN logger](#) that is able to transmit custom CAN frames can also be used as an OBD2 logger. Note that cars differ by model/year in what OBD2 PIDs they support. For details, see our [OBD2 data logger](#) guide.

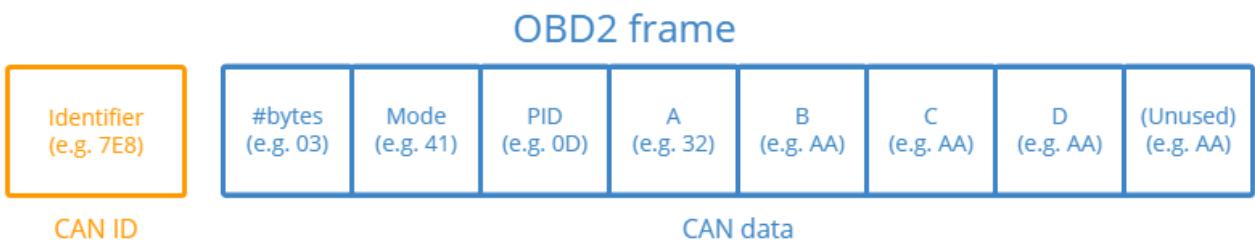


CANedge OBD2 data logger

The [CANedge](#) lets you easily log OBD2 data to an 8-32 GB SD card. Simply specify what OBD2 PIDs you wish to request, then connect it to your car via an [OBD2 adapter](#) to start logging. Process the data via [free software/APIs](#) and our [OBD2 DBC](#).

Raw OBD2 frame details

To get started recording OBD2 data, it is helpful to understand the basics of the raw OBD2 message structure. In simplified terms, an OBD2 message consists of an identifier and data. Further, the data is split in Mode, PID and data bytes (A, B, C, D) as below.



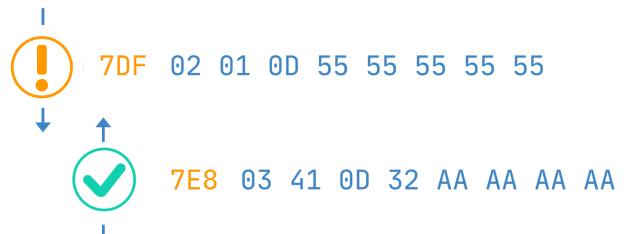
OBD2 message fields explained

- **Identifier:** For OBD2 messages, the identifier is standard 11-bit and used to distinguish between "request messages" (ID 7DF) and "response messages" (ID 7E8 to 7EF). Note that 7E8 will typically be where the main engine or ECU responds at.
- **Length:** This simply reflects the length in number of bytes of the remaining data (03 to 06). For the Vehicle Speed example, it is 02 for the request (since only 01 and 0D follow), while for the response it is 03 as both 41, 0D and 32 follow.
- **Mode:** For requests, this will be between 01-0A. For responses the 0 is replaced by 4 (i.e. 41, 42, ..., 4A). There are 10 modes as described in the SAE J1979 OBD2 standard. Mode 1 shows Current Data and is e.g. used for looking at real-time vehicle speed, RPM etc. Other modes are used to e.g. show or clear stored diagnostic trouble codes and show freeze frame data.

- **PID:** For each mode, a list of standard OBD2 PIDs exist - e.g. in Mode 01, PID 0D is Vehicle Speed. For the full list, check out our [OBD2 PID overview](#). Each PID has a description and some have a specified min/max and conversion formula. The formula for speed is e.g. simply A, meaning that the A data byte (which is in HEX) is converted to decimal to get the km/h converted value (i.e. 32 becomes 50 km/h above). For e.g. RPM (PID 0C), the formula is $(256 \times A + B) / 4$.
- **A, B, C, D:** These are the data bytes in HEX, which need to be converted to decimal form before they are used in the PID formula calculations. Note that the last data byte (after Dh) is not used.

OBD2 request/response example

An example of a request/response CAN message for the PID 'Vehicle Speed' with a value of 50 km/h can be seen in the illustration. Note in particular how the formula for the OBD2 PID 0D (Vehicle Speed) simply involves taking the 4th byte (0x32) and converting it to decimal form (50).



Extended OBD2 PID request/response

In some vehicles (e.g. vans and light/medium/heavy duty vehicles), you may find that the raw CAN data uses extended 29-bit CAN identifiers instead of 11-bit CAN identifiers.

In this case, you will typically need to modify the OBD2 PID requests to use the CAN ID 18DB33F1 instead of 7DF. The data payload structure is kept identical to the examples for 11-bit CAN IDs.

If the vehicle responds to the requests, you'll typically see responses with CAN IDs 18DAF100 to 18DAF1FF (in practice, typically 18DAF110 and 18DAF11E). The response identifier is also sometimes shown in the '[J1939 PGN](#)' form, specifically the PGN 0xDA00 (55808), which in the J1939-71 standard is marked as 'Reserved for ISO 15765-2'.

We provide an [OBD2 DBC file](#) for both the 11-bit and 29-bit responses, enabling simple decoding of the data in most CAN software tools.

The 10 OBD2 services (aka modes)

There are 10 OBD2 diagnostic services (or modes) as described in the SAE J1979 OBD2 standard. Mode 1 shows Current Data and is used for looking at real-time parameters like vehicle speed, RPM, throttle position etc. Other modes are e.g. used to show/clear diagnostic trouble codes (DTCs) and show freeze frame data. Manufacturers do not have to support all diagnostic services - and they may support modes outside these 10 services (i.e. manufacturer specific OBD2 services).

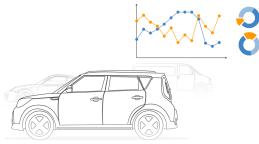
OBD2 diagnostic services/modes (SAE J1979)

- | | |
|-----------|-----------------------------------------------------------------------|
| 01 | Show current data (e.g. real-time data) |
| 02 | Show freeze frame data (as above, but at time of freeze frame) |
| 03 | Show stored Diagnostic Trouble Codes (DTCs) |
| 04 | Clear DTCs and stored values |
| 05 | Test results for oxygen sensors (non CAN only) |
| 06 | Test results for system monitoring (and oxygen sensors for CAN) |
| 07 | Show pending DTCs |
| 08 | Control operation of on-board system |
| 09 | Request vehicle information (e.g. VIN) |
| 0A | Permanent DTCs (aka cleared DTCs) |



OBD2 data logging - use case examples

OBD2 data from cars and light trucks can be used in various use cases:



Logging data from cars

OBD2 data from cars can e.g. be used to reduce fuel costs, improve driving, test prototype parts and insurance

[learn more](#)



Real-time car diagnostics

OBD2 interfaces can be used to stream human-readable OBD2 data in real-time, e.g. for diagnosing vehicle issues

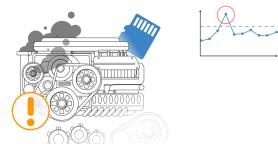
[learn more](#)



Predictive maintenance

Cars and light trucks can be monitored via IoT OBD2 loggers in the cloud to predict and avoid breakdowns

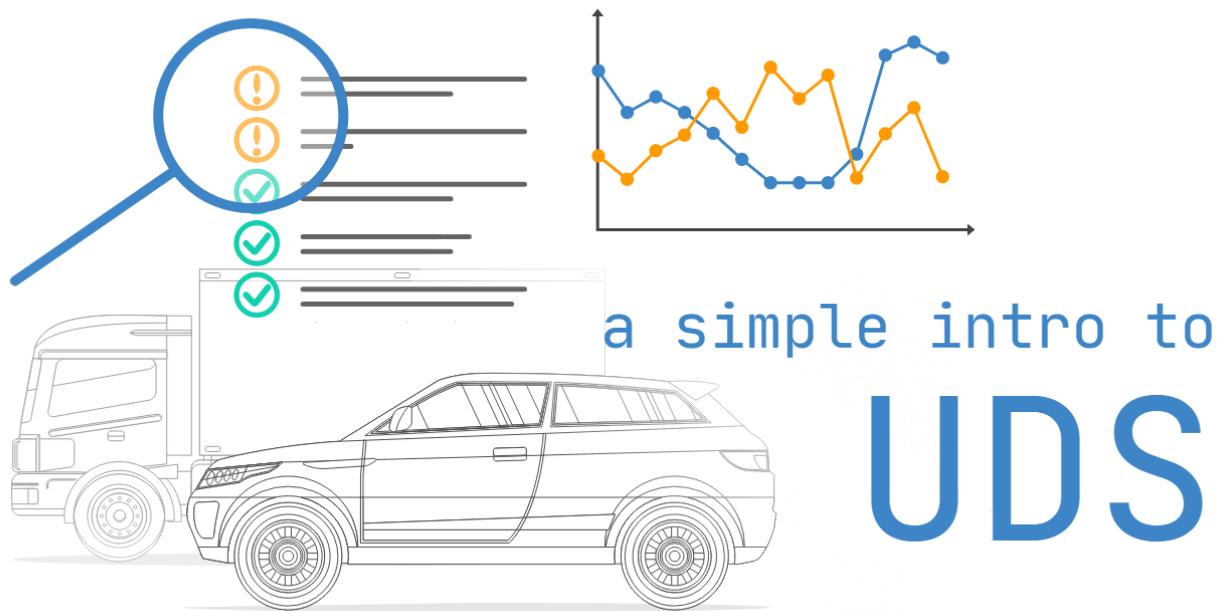
[learn more](#)



Vehicle blackbox logger

An OBD2 logger can serve as a 'blackbox' for vehicles or equipment, providing data for e.g. disputes or diagnostics

[learn more](#)



UDS Explained (Unified Diagnostic Services)

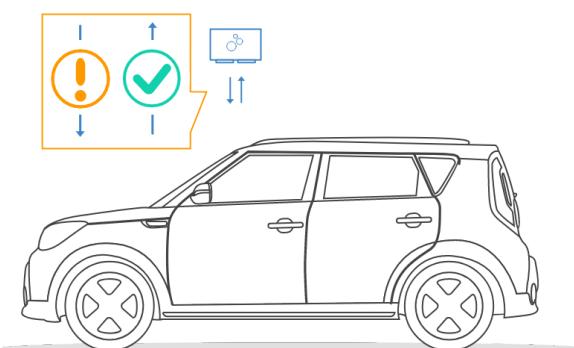
In this practical tutorial, we introduce the UDS basics with focus on UDS on CAN bus (UDSonCAN) and Diagnostics over CAN (DoCAN). We also introduce the ISO-TP protocol and explain the difference between UDS, OBD2, WWH-OBD and OBDonUDS. Finally, we'll explain how to request, record & decode UDS messages - with practical examples for logging EV State of Charge and the Vehicle Identification Number (VIN).

What is the UDS protocol?

Unified Diagnostic Services (UDS) is a communication protocol used in automotive Electronic Control Units (ECUs) to enable diagnostics, firmware updates, routine testing and more.

The UDS protocol (ISO 14229) is standardized across both manufacturers and standards (such as CAN, KWP 2000, Ethernet, LIN). Further, UDS is today used in ECUs across all tier 1 Original Equipment Manufacturers (OEMs).

In practice, UDS communication is performed in a client-server relationship - with the client being a tester-tool and the server being a vehicle ECU. For example, you can connect a [CAN bus](#) interface to the [OBD2 connector](#) of a car and send UDS requests into the vehicle. Assuming the targeted ECU supports UDS services, it will respond accordingly.



In turn, this enables various use cases:

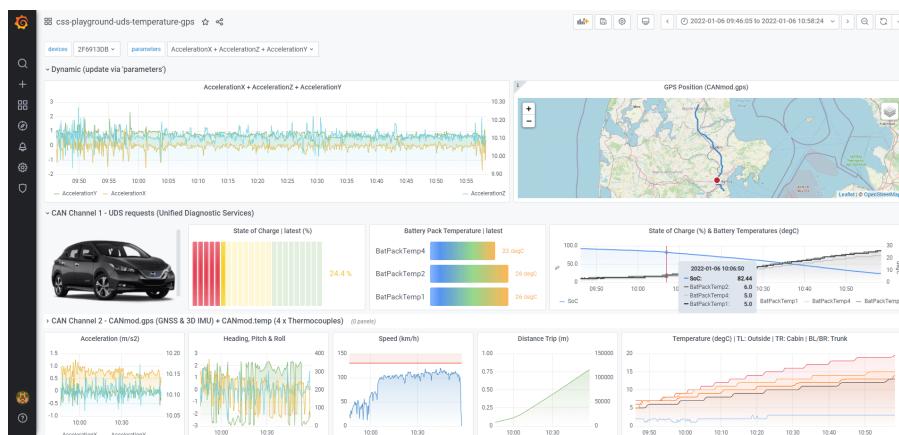
- Read/clear diagnostic trouble codes (DTC) for troubleshooting vehicle issues
- Extract parameter data values such as temperatures, state of charge, VIN etc
- Initiate diagnostic sessions to e.g. test safety-critical features
- Modify ECU behavior via resets, firmware flashing and settings modification

UDS is often referred to as 'vehicle manufacturer enhanced diagnostics' or 'enhanced diagnostics' - more on this below.

Example: Nissan Leaf SoC%

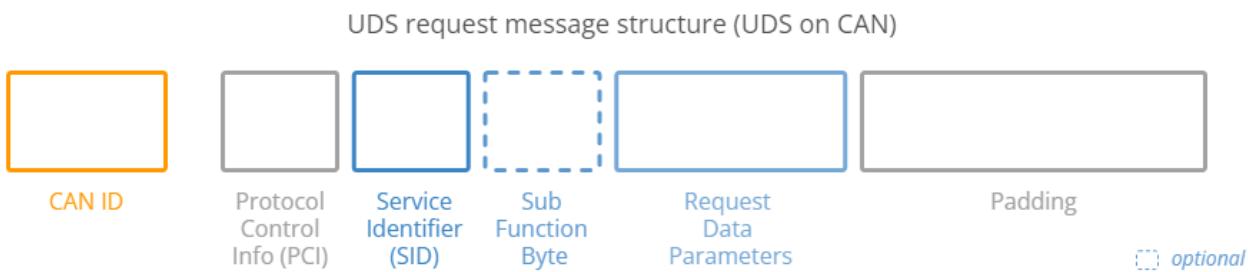
UDS and CAN ISO-TP are complex topics. As motivation, we've done a case study to show how it can be useful. Specifically, we use a [CANedge2](#) to request data on State of Charge (SoC%) and battery temperatures from a Nissan Leaf EV. In the example, we also add a [CANmod.gps](#) and [CANmod.temp](#) to add GNSS, IMU and temperature data.

The multiframe ISO-TP responses and CANmod signals are DBC decoded via our [Python API](#) and written to a database for visualization in [Grafana dashboards](#). Check it out: [playground case study](#)



UDS message structure

UDS is a request based protocol. In the illustration we've outlined an example of an UDS request frame (using CAN bus):



A diagnostic UDS request on CAN contains various fields that we detail below:

Protocol Control Information (PCI)

The PCI field is not per se related to the UDS request itself, but is required for diagnostic UDS requests made on CAN bus. In short, the PCI field can be 1-3 bytes long and contains information related to the transmission of messages that do not fit within a single CAN frame. We will detail this more in the section on the CAN bus transport protocol (ISO-TP).

UDS Service ID (SID)

The use cases outlined above relate to different UDS services. When you wish to utilize a specific UDS service, the UDS request message should contain the UDS Service Identifier (SID) in the data payload. Note that the identifiers are split between request SIDs (e.g. 0x22) and response SIDs (e.g. 0x62). As in OBD2, the response SIDs generally add 0x40 to the request SIDs. See also the overview of all standardized UDS services and SIDs. We will mainly focus on UDS service 0x22 in this article, which is used to read data (e.g. speed, SoC, temperature, VIN) from an ECU.

UDS service identifiers (SIDs)

UDS SID (request)	UDS SID (response)	Service	Details
Diagnostic and Communications Management	0x10	Diagnostic Session Control	Control which UDS services are available
	0x11	ECU Reset	Reset the ECU ("hard reset", "key off", "soft reset")
	0x27	Security Access	Enable use of security-critical services via authentication
	0x28	Communication Control	Turn sending/receiving of messages on/off in the ECU
	0x29	Authentication	Enable more advanced authentication vs. 0x27 (PKI based exchange)
	0x3E	Tester Present	Send a "heartbeat" periodically to remain in the current session
	0x83	Access Timing Parameters	View/modify timing parameters used in client/server communication
	0x84	Secured Data Transmission	Send encrypted data via ISO 15764 (Extended Data Link Security)
	0x85	Control DTC Settings	Enable/disable detection of errors (e.g. used during diagnostics)
	0x86	Response On Event	Request that an ECU processes a service request if an event happens
Data Transmission	0x87	Link Control	Set the baud rate for diagnostic access
	0x22	Read Data By Identifier	Read data from targeted ECU - e.g. VIN, sensor data values etc.
	0x23	Read Memory By Address	Read data from physical memory (e.g. to understand software behavior)
	0x24	Read Scaling Data By Identifier	Read information about how to scale data identifiers
	0x2A	Read Data By Identifier Periodic	Request ECU to broadcast sensor data at slow/medium/fast/stop rate
	0x2C	Dynamically Define Data Identifier	Define data parameter for use in 0x22 or 0x2A dynamically
	0x2E	Write Data By Identifier	Program specific variables determined by data parameters
DTCs	0x3D	Write Memory By Address	Write information to the ECU's memory
	0x14	Clear Diagnostic Information	Delete stored DTCs
	0x19	Read DTC Information	Read stored DTCs, as well as related information
	0x2F	Input Output Control By Identifier	Gain control over ECU analog/digital inputs/outputs
Upload/Download	0x31	Routine Control	Initiate/stop routines (e.g. self-testing, erasing of flash memory)
	0x34	Request Download	Start request to add software/data to ECU (incl. location/size)
	0x35	Request Upload	Start request to read software/data from ECU (incl. location/size)
	0x36	Transfer Data	Perform actual transfer of data following use of 0x74/0x75
	0x37	Request Transfer Exit	Stop the transfer of data
	0x38	Request File Transfer	Perform a file download/upload to/from the ECU
	0x7F	Negative Response	Sent with a Negative Response Code when a request cannot be handled

UDS SIDs vs other diagnostic services

The standardized UDS services shown above are in practice a subset of a larger set of diagnostic services - see below overview. Note here that the SIDs 0x00 to 0x0F are reserved for legislated OBD services (more on this later).

Diagnostic service identifiers - overview (0x00 - 0xFF)

Service Identifier (SID)	Service type	Further details
0x00 - 0x0F	OBD service requests	ISO 15031-5
0x00 - 0x3E	ISO 14229 (requests)	ISO 14229
0x3F	Not applicable	Reserved
0x40 - 0x4F	OBD service responses	ISO 15031-5
0x50 - 0x7E	ISO 14229 (responses)	ISO 14229
0x7F	Negative response SID	ISO 14229
0x80	Not applicable	ISO 14229 (reserved)
0x81 - 0x82	Not applicable	ISO 14229 (reserved)
0x83 - 0x88	ISO 14229 (requests)	ISO 14229
0x89 - 0x9F	Service requests	Reserved
0xA0 - 0xB9	Service requests	Defined by vehicle OEM
0xBA - 0xBE	Service requests	Defined by systems OEM
0xBF	Not applicable	Reserved
0xC0	Not applicable	ISO 14229 (reserved)
0xC1 - 0xC2	Not applicable	ISO 14229 (reserved)
0xC3 - 0xC8	ISO 14229 (responses)	ISO 14229
0xC9 - 0xDF	Service responses	Reserved
0xE0 - 0xF9	Service responses	Defined by vehicle OEM
0xFA - 0xFE	Service responses	Defined by systems OEM
0xFF	Not applicable	Reserved

UDS request SIDs UDS positive response SIDs UDS negative response SID

UDS security via session-control (authentication)

As evident, UDS enables extensive control over the vehicle ECUs. For security reasons, critical UDS services are therefore restricted through an authentication process. Basically, an ECU will send a 'seed' to a tester, who in turn must produce a 'key' to gain access to security-critical services. To retain this access, the tester needs to send a 'tester present' message periodically.

In practice, this authentication process enables vehicle manufactures to restrict UDS access for aftermarket users and ensure that only designated tools will be able to utilize the security-critical UDS services.

Note that the switching between authentication levels is done through diagnostic session control, which is one of the UDS services available. Vehicle manufactures can decide which sessions are supported, though they must always support the 'default session' (i.e. which does not involve any authentication). With that said, they decide what services are supported within the default session as well. If a tester tool switches to a non-default session, it must send a 'tester present' message periodically to avoid being returned to the default session.

UDS Sub Function Byte

The sub function byte is used in some UDS request frames as outlined below. Note, however, that in some UDS services, like 0x22, the sub function byte is not used. Generally, when a request is sent to an ECU, the ECU may respond positively or negatively. In case the response is positive, the tester may want to suppress the response (as it may be irrelevant). This is done by setting the 1st bit to 1 in the sub function byte. Negative responses cannot be suppressed.

The remaining 7 bits can be used to define up to 128 sub function values. For example, when reading DTC information via SID 0x19 (Read Diagnostic Information), the sub function can be used to control the report type - see also below table.

UDS services - sub function types

UDS SID (request)	UDS SID (response)	Service	Sub function types
0x10	0x50	Diagnostic Session Control	Diagnostic session type
0x11	0x51	ECU Reset	Reset type
0x27	0x67	Security Access	Security access type
0x28	0x68	Communication Control	Control type
0x3E	0x7E	Tester Present	"Zero sub function"
0x83	0xC3	Access Timing Parameters	Timing parameter access type
0x85	0xC5	Control DTC Settings	DTC setting type
0x86	0xC6	Response On Event	Event type
0x87	0xC7	Link Control	Link control type
0x2C	0x6C	Dynamically Define Data Identifier	Definition type
0x19	0x59	Read DTC Information	Report type
0x31	0x71	Routine Control	Routine control type

Example: Service 0x19 sub functions

If we look specifically at service 0x19, we can see an example of the various sub functions below:

UDS service 0x19 - sub function byte values & types

UDS SID (request)	UDS SID (response)	Report type (sub function value)	Report
0x19	0x59	0x01	Number of DTC by status mask
		0x02	DTC by status mask
		0x03	DTC snapshot identification
		0x04	DTC snapshot record by DTC number
		0x05	DTC stored data by record number
		0x06	DTC extended data record by DTC number
		0x07	Number of DTC by severity mask record
		0x08	DTC by severity mask record
		0x09	Severity information of DTC
		0x0A	Supported DTC
		0x0B	First failed DTC
		0x0C	First confirmed DTC
		0x0D	Most recent failed DTC
		0x0E	Most recent confirmed DTC
		0x0a	Mirror memory DTC by status mask
		0x10	Mirror memory DTC by DTC number
		0x11	Number of mirror memory DTC by status mask
		0x12	Number of emissions OBD DTC by status mask
		0x13	Emissions OBD DTC by status mask
		0x14	DTC fault detection counter
		0x15	DTC with permanent status
		0x16	DTC extended data record by record number
		0x17	User defined memory DTC by status mask
		0x18	User defined memory DTC snapshot by number
		0x19	User defined memory DTC record by number
		0x42	WWH-OBD DTC by status mask record
		0x55	WWH-OBD DTCs with permanent status

UDS 'Request Data Parameters' - incl. Data Identifier (DID)

In most UDS request services, various types of request data parameters are used to provide further configuration of a request beyond the SID and optional sub function byte. Here we outline two examples.

Service 0x19 (Read DTC Information) - request configuration

For example, service 0x19 lets you read DTC information. The UDS request for SID 0x19 includes a sub function byte - for example, 0x02 lets you read DTCs via a status mask. In this specific case, the sub function byte is followed by a 1-byte parameter called DTC Status Mask to provide further information regarding the request. Similarly, other types of sub functions within 0x19 have specific ways of configuring the request.

Service 0x22 (Read Data by Identifier) - Data Identifiers

Another example is service 0x22 (Read Data by Identifier). This service uses a Data Identifier (DID), which is a 2-byte value between 0 and 65535 (0xFFFF). The DID serves as a parameter identifier for both requests/responses (similar to how the parameter identifier, or PID, is used in OBD2).

For example, a request for reading data via a single DID in UDS over CAN would include the PCI field, the UDS service 0x22 and the 2-byte DID. Alternatively, one can request data for additional DIDs by adding them after the initial DID in the request. We will look further into this in the section on how to record and decode UDS communication.

Data Identifiers can be proprietary and only known by OEMs, though some DIDs are standardized. This is for example the case for the WWH-OBD DIDs (more on this later) and the Vehicle Identification Number (VIN) is 0xF190. See the separate table for a list of standardized DIDs across UDS.

UDS - standardized data identifiers (DID)

UDS DID (data identifier)	Description
0xF180	Boot software identification
0xF181	Application software identification
0xF182	Application data identification
0xF183	Boot software fingerprint
0xF184	Application software fingerprint
0xF185	Application data fingerprint
0xF186	Active diagnostic session
0xF187	Manufacturer spare part number
0xF188	Manufacturer ECU software number
0xF189	Manufacturer ECU software version
0xF18A	Identifier of system supplier
0xF18B	ECU manufacturing date
0xF18C	ECU serial number
0xF18D	Supported functional units
0xF18E	Manufacturer kit assembly part number
0xF190	Vehicle Identification Number (VIN)
0xF192	System supplier ECU hardware number
0xF193	System supplier ECU hardware version number
0xF194	System supplier ECU software number
0xF195	System supplier ECU software version number
0xF196	Exhaust regulation/type approval number
0xF197	System name / engine type
0xF198	Repair shop code / tester serial number
0xF199	Programming date
0xF19D	ECU installation date
0xF19E	ODX file

Positive vs. negative UDS responses

When an ECU responds positively to an UDS request, the response frame is structured with similar elements as the request frame. For example, a 'positive' response to a service 0x22 request will contain the response SID 0x62 (0x22 + 0x40) and the 2-byte DID, followed by the actual data payload for the requested DID. Generally, the structure of a positive UDS response message depends on the service. However, in some cases an ECU may provide a negative response to an UDS request - for example if the service is not supported. A negative response is structured as in below CAN frame example:

UDS Negative Response example (UDS on CAN)



Details + Negative Response Code table

Below we briefly detail the negative response frame with focus on the NRC:

- The 1st byte is the PCI field
- The 2nd byte is the Negative Response Code SID, 0x7F
- The 3rd byte is the SID of the rejected request
- The 4th byte is the Negative Response Code (NRC)

In the negative UDS response, the NRC provides information regarding the cause of the rejection as per the table below.

UDS SID 0x7F – Negative Response Codes (NRC)

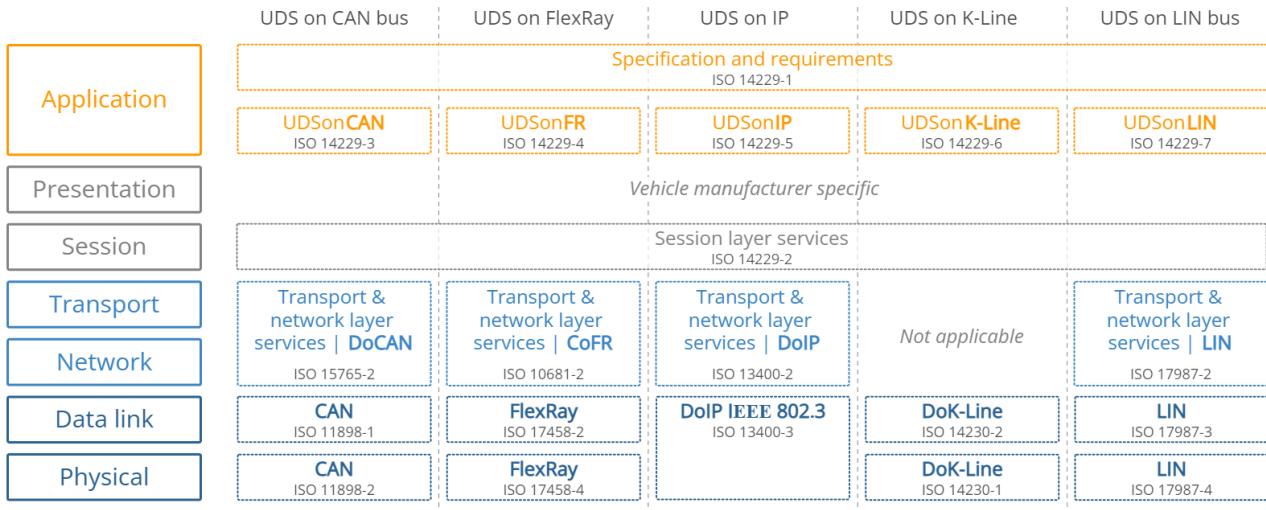
UDS NRC	Description
0x10	General reject
0x11	Service not supported
0x12	Sub-function not supported
0x13	Invalid message length/format
0x14	Response too long
0x21	Busy-repeat request
0x22	Conditions not correct
0x24	Request sequence error
0x25	No response from subnet component
0x26	Failure prevents execution of requested action
0x31	Request out of range
0x33	Security access denied
0x35	Invalid key
0x36	Exceeded number of attempts
0x37	Required time delay has not expired
0x70	Upload/download not accepted
0x71	Transfer data suspended
0x72	Programming failure
0x73	Wrong block sequence counter
0x78	Request received - response pending
0x7E	Sub function not supported in active session
0x7F	Service not supported in active session
0x81/0x82	RPM too high/low
0x83/0x84	Engine is running/not running
0x85	Engine run time too low
0x86/0x87	Temperature too high/low
0x88/0x89	Speed too high/low
0x8A/0x8B	Throttle pedal too high/low
0x8C/0x8D	Transmission range not in neutral/dear
0x8F	Brake switches not closed
0x90	Shifter lever not in park
0x91	Torque converter clutch locked
0x92/0x93	Voltage too high/low
0xF0-0xFE	Manufacturer specific conditions not correct

UDS vs CAN bus: Standards & OSI model

To better understand UDS, we will look at how it relates to CAN bus and the OSI model.

As explained in our [CAN bus tutorial](#), the Controller Area Network serves as a basis for communication. Specifically, CAN is described by a data-link layer and physical layer in the OSI model (as per ISO 11898). In contrast to CAN, UDS (ISO 14229) is a 'higher layer protocol' and utilizes both the session layer (5th) and application layer (7th) in the OSI model as shown below:

7 layer OSI model | Unified Diagnostic Services (UDS)



Overview of UDS standards & concepts

UDS refers to a large number of standards/concepts, meaning it can be a bit overwhelming. To give you an overview, we provide a high-level explanation of the most relevant ones below (with focus on CAN as the basis).

Quick overview of the UDS OSI model layers

In the following we provide a quick breakdown of each layer of the OSI model:

- Application: This is described by ISO 14229-1 (across the various serial data link layers). Further, separate ISO standards describe the UDS application layer for the various lower layer protocols - e.g. ISO 14229-3 for CAN bus (aka UDSonCAN)
- Presentation: This is vehicle manufacturer specific
- Session: This is described in ISO 14229-2
- Transport + Network: For CAN, ISO 15765-2 is used (aka ISO-TP)
- Data Link: In the case of CAN, this is described by ISO 11898-1
- Physical: In the case of CAN, this is described by ISO 11898-2

As illustrated, multiple standards other than CAN may be used as the basis for UDS communication - including FlexRay, Ethernet, LIN bus and K-line. In this tutorial we focus on CAN, which is also the most common lower layer protocol.

ISO 14229-1 (Application Layer)

The ISO 14229-1 standard describes the application layer requirements for UDS (independent of what lower layer protocol is used). In particular, it outlines the following:

- Client-server communication flows (requests, responses, ...)
- UDS services (as per the overview described previously)
- Positive responses and negative response codes (NRCs)
- Various definitions (e.g. DTCs, parameter data identifiers aka DIDs, ...)

ISO 14229-3 (Application Layer for CAN)

The purpose of 14229-3 is to enable the implementation of Unified Diagnostic Services (UDS) on Controller Area Networks (CAN) - also known as UDSONCAN. This standard describes the application layer requirements for UDSONCAN.

This standard does not describe any implementation requirements for the in-vehicle CAN bus architecture. Instead, it focuses on some additional requirements/restrictions for UDS that are specific to UDSONCAN.

Specifically, 14229-3 outlines which UDS services have CAN specific requirements. The affected UDS services are ResponseOnEvent and ReadDataByPeriodicIdentifier, for which the CAN specific requirements are detailed in 14229-3. All other UDS services are implemented as per ISO 14229-1 and ISO 14229-2.

ISO 14229-3 also describes a set of mappings between ISO 14229-2 and ISO 15765-2 (ISO-TP) and describes requirements related to 11-bit and 29-bit CAN IDs when these are used for UDS and legislated OBD as per ISO 15765-4.

ISO 14229-2 (Session Layer)

This describes the session layer in the UDS OSI model. Specifically, it outlines service request/confirmation/indication primitives. These provide an interface for the implementation of UDS (ISO 14229-1) with any of the communication protocols (e.g. CAN).

ISO 15765-2 (Transport + Network Layer for CAN)

For UDS on CAN, ISO 15765-2 describes how to communicate diagnostic requests and responses. In particular, the standard describes how to structure CAN frames to enable communication of multi-frame payloads. As this is a vital part of understanding UDS on CAN, we go into more depth in the next section.

ISO 11898 (Physical + Data Link Layer for CAN)

When UDS is based on CAN bus, the physical and data link layers are described in ISO 11898-1 and ISO 11898-2. When UDS is based on CAN, it can be compared to a higher layer protocol like [J1939](#), OBD2, [CANopen](#), [NMEA 2000](#) etc. However, in contrast to these protocols, UDS could alternatively be based on other communication protocols like FlexRay, Ethernet, [LIN](#) etc.

UDSONCAN vs DoCAN

When talking about UDS based on CAN bus, you'll often see two terms used: UDSONCAN (UDS on CAN bus) and DoCAN (Diagnostics on CAN bus). Some UDS tutorials use these terms interchangeably, which may cause confusion.

In ISO 14229-1 the terms are used as in our OSI model illustration. In other words, UDSONCAN is used to refer to ISO 14229-3, while DoCAN is used to refer to ISO 15765-2 aka ISO-TP.

However, part of the confusion may arise because ISO 14229-3 also provides an OSI model where DoCAN is both used in relation to ISO 15765-2 and as an overlay across OSI model layers 2 to 7. In ISO 14229-2, DoCAN is referred to as the

communication protocol on which UDS (ISO 14229-1) is implemented. This is in sync with the illustration from ISO 14229-3. In this context, DoCAN can be viewed as a more overarching term for the implementation of UDS on CAN, whereas UDSonCAN seems consistently to refer to ISO 14229-3 only.

ISO 15765-3 vs. ISO 14229-3

UDS on CAN bus (UDSonCAN) is sometimes referred to through ISO 15765-3. However, this standard is now obsolete and has been replaced by ISO 14229-3.

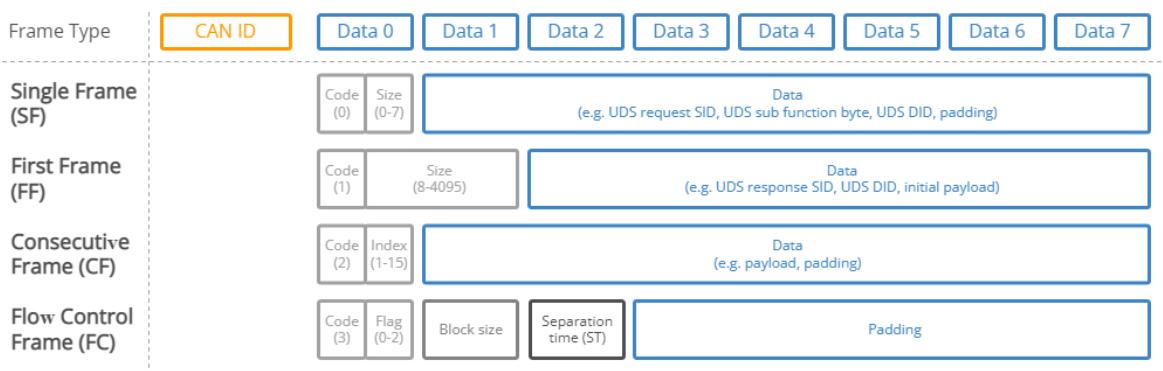
CAN ISO-TP - Transport Protocol (ISO 15765-2)

When implementing diagnostics on CAN, one challenge is the size of the CAN frame payload: For Classical CAN frames, this is limited to 8 bytes and for CAN FD the payload is limited to 64 bytes. Vehicle diagnostics often involve communication of far larger payloads.

ISO 15765-2 was established to solve the challenge of large payloads for CAN based vehicle diagnostics. The standard specifies a transport protocol and network layer services for use in CAN based vehicle networks. The most common use cases include UDS (ISO 14229-1), OBD (SAE J1979, ISO 15031-5) and world-wide harmonized OBD aka WWH-OBD (ISO 27145).

The ISO-TP standard outlines how to communicate CAN data payloads up to 4095 bytes through segmentation, flow control and reassembly. ISO-TP defines specific CAN frames for enabling this communication as shown below:

ISO TP frame types (CAN Bus Transport Protocol, ISO 15765-2)



Regarding the Flow Control frame

The flow control frame is used to 'configure' the subsequent communication. It can be constructed as below:

ISO TP Flow Control (FC) frame types (CAN Bus Transport Protocol, ISO 15765-2)



Flag
0x00 = Continue To Send
0x01 = Wait
0x02 = Overflow/abort

Block size
0x00 = remaining "frames" to be sent without flow control or delay
> 0x00 = send number of "frames" before waiting for the next flow control frame

Separation time (ST)
0x00 - 0x7F = separation time in milliseconds (0 - 127 ms)
0xF1 - 0xF9 = separation time in even multiples of 100 microseconds (0xF1 = 100 µs, 0xF2 = 200 µs, ... 0xF9 = 900 µs)

A few comments:

- In the simplest case, the FC payload can be set to 30 00 00 00 00 00 00 00 (all remaining frames to be sent without delay)
- Alternatively, one can decide to perform more granular control over the communication by e.g. alternating between the Wait and Continue commands, as well as specifying a specific separation time (in milliseconds) between frames

Other ISO-TP frame comments

- The ISO-TP frame type can be identified from the first nibble of the first byte (0x0, 0x1, 0x2, 0x3)
- The total frame size can be up to 4095 bytes (0xFFFF) as evident from the FF frame
- The CF index runs from 1 to 15 (0xF) and is then reset if more data is to be sent
- Padding (e.g. 0x00, 0xAA, ...) is used to ensure the frame payloads equal 8 bytes in length

Below we outline how the ISO-TP protocol works for single-frame and multi-frame communication:

ISO-TP: Single-frame communication

In vehicle diagnostics, communication is initiated by a tester tool sending a request. This request frame is a Single Frame (SF). In the simplest case, a tester tool sends a Single Frame to request data from an ECU. If the response can be contained in a 7-byte payload, the ECU provides a Single Frame response.

UDS Single Frame request/response example: Read Data by Identifier (UDS on CAN)



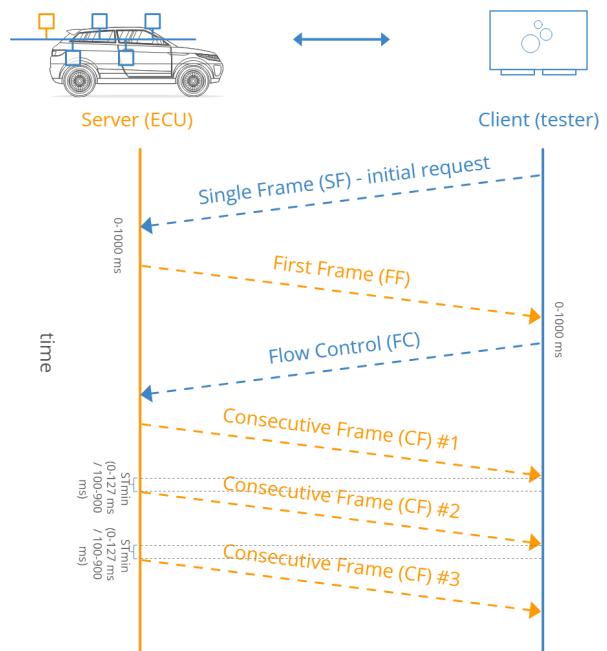
ISO-TP: Multi-frame communication

When the payload exceeds 7 bytes, it needs to be split across multiple CAN frames.

As before, a tester starts by sending a Single Frame (SF) request to an ECU (sender). However, in this case the response exceeds 7 bytes. Because of this, the ECU sends a First Frame (FF) that contains information on the total packet length (8 to 4095 bytes) as well as the initial chunk of data. When the tester receives the FF, it will send a Flow Control (FC) frame, which tells the ECU how the rest of the data transfer should be transmitted.

Following this, the ECU will send Consecutive Frames (CF) that contain the remaining data payload.

ISO-TP plays an important role in most CAN based diagnostics protocols. Before we show practical examples of such communication flows, it is useful to get an overview of the most common vehicle diagnostic protocols.



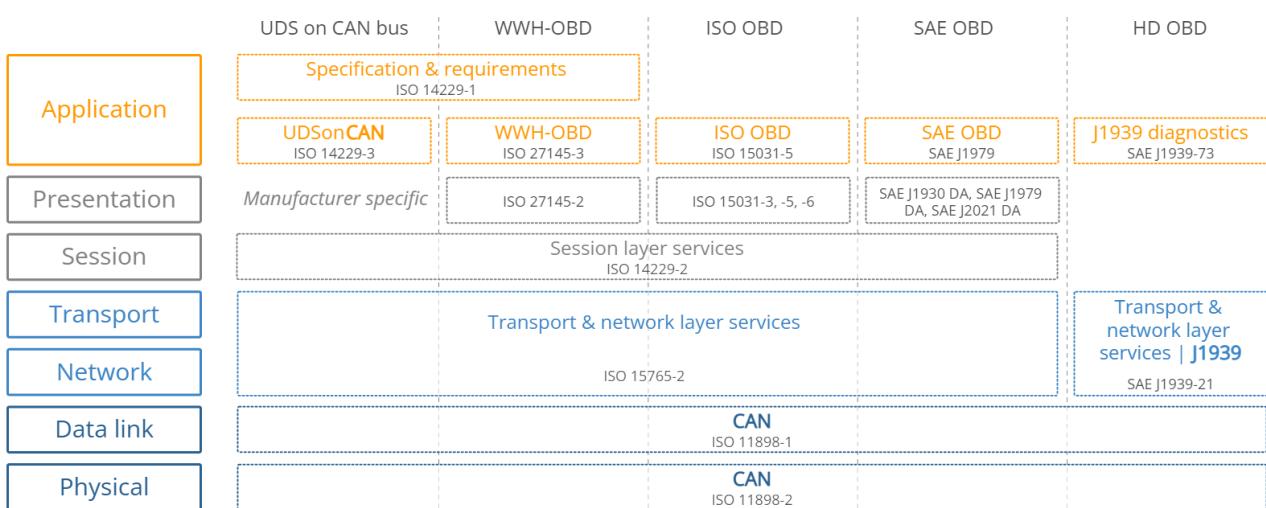
UDS vs. OBD2 vs. WWH-OBD vs. OBDonUDS

A common question is how UDS relates to On-Board Diagnostics (OBD2), World-Wide Harmonized OBD (WWH-OBD) and OBDonUDS. To understand this, it is important to first note the following:

OBD (On-Board Diagnostics) is today implemented in different ways across countries and vehicles.

This is illustrated via the below OSI model comprising CAN based vehicle diagnostic protocols in use today:

7 layer OSI model | Diagnostic Protocols (on CAN)



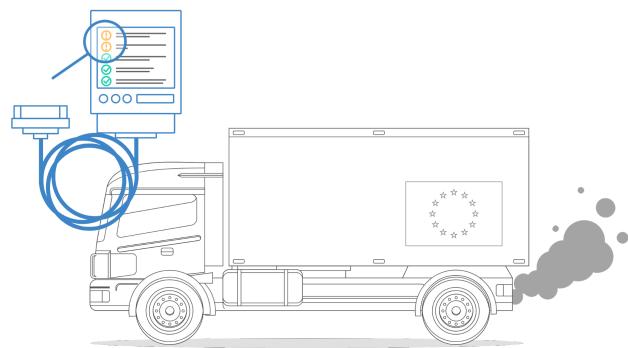
Let's look at each diagnostic protocol:

- ISO OBD (or EOBD) refers to the OBD protocol specification legislated for use in EU cars, while SAE OBD refers to the OBD protocol specification legislated for use in US. The two are technically equivalent and hence often referred to simply as OBD or OBD2
- HD OBD (SAE J1939) typically refers to heavy duty OBD and is commonly implemented through the J1939 protocol in both US and EU produced vehicles with J1939-73 specifying diagnostic messages
- UDS (ISO 14229) has been implemented by vehicle manufacturers to serve the need for richer diagnostics data/functionality - beyond the limits of the emissions-focused OBD protocols. It is implemented in most ECUs today, across markets and vehicle types - though in itself, UDS does not offer the necessary standardization required to serve as an alternative to OBD
- WWH-OBD (and/or possibly OBDonUDS) provide an updated version of OBD2 for emissions-related diagnostics - based on UDS

To understand UDS, it is useful to better understand WWH-OBD and OBDonUDS:

What is WWH-OBD (ISO 27145)?

WWH-OBD is a global standard for vehicle diagnostics, developed by the UN under the Global Technical Regulations (GTR) mandate. It aims to provide a single, future-proof alternative to the existing OBD protocols (ISO OBD, SAE OBD, HD OBD). Furthermore, WWH-OBD is based on UDS in order to suit the enhanced diagnostics functionality already deployed by most automotive OEMs today.



Advantages of WWH-OBD

Moving from OBD2 to WWH-OBD will involve a number of benefits, primarily derived from using the UDS protocol as the basis. First of all, the data richness can be increased. OBD2 parameter identifiers (PID) are limited to 1 byte, restricting the number of unique data types to 255, while the UDS data identifier (DID) is 2 bytes, enabling 65535 parameters.

For diagnostic trouble codes (DTCs), OBD2 would allow for 2-byte DTCs. Here, WWH-OBD allows for 'extended DTCs' of 3 bytes. This allows for grouping DTCs by 2-byte types and using the 3rd byte as a failure mode indicator to specify the DTC sub type. Further, WWH-OBD enables a classification of DTCs based on how severe an issue is in regards to the exhaust emissions quality.

WWH-OBD also seeks to take potential future requirements into account by allowing for the Internet Protocol (IP) to be used as an alternative to CAN, meaning that UDSONIP will also be possible in future implementations of WWH-OBD. One potential benefit from this will be the ability to one day perform remote diagnostics through the same protocol.

What is the status on WWH-OBD roll-out?

The intent of WWH-OBD is to serve as a global standard, across all markets and across all vehicle types (cars, trucks, buses, ...). Further, the aim is to potentially expand the standardized functionality beyond just emissions-control.

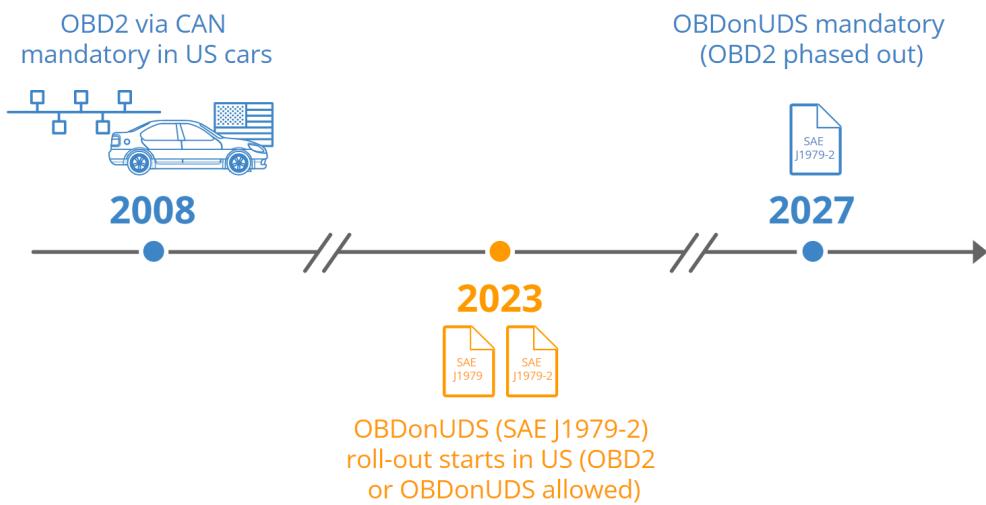
In practice, WWH-OBD has been required in EU since 2014 for newly developed heavy duty vehicles (as per the Euro-VI standards). Note in this case that HD OBD (as per J1939) remains allowed in these vehicles.

Beyond this, however, the roll-out of WWH-OBD has been limited. One challenge is that WWH-OBD is currently not accepted by EPA/CARB in USA. See e.g. [this discussion](#) for potential motivations. However, recently OBDonUDS (SAE J1979-2) is being adopted in US.

What is OBDonUDS (SAE J1979-2)?

Similar to how OBD2 has been split into 'SAE OBD' (SAE J1979) for US and 'ISO OBD' (ISO 15031) for EU, the 'next generation' of OBD2 may again be regionally split.

Specifically, WWH-OBD (ISO 21745) has been adopted in EU for heavy duty vehicles already - but not yet in the US. Instead, it has recently been decided to adopt OBD on UDS in US vehicles in the form of the SAE J1979-2 standard, which serves as an update to the SAE J1979. The new SAE J1979-2 standard is also referred to as OBDonUDS. The aim is to initiate a transition phase starting in 2023, where ECUs are allowed to support either OBD2 or OBDonUDS. From 2027, OBDonUDS will be a mandatory requirement for all vehicles produced in the US.



Looking ahead: WWH-OBD vs. OBDonUDS

To recap, WWH-OBD and OBDonUDS both serve as possible solutions for creating a 'next generation' protocol for emissions-related on-board diagnostics. It remains to be seen if the two will exist in parallel (like ISO/SAE OBD), or if one of the protocols will become the de facto standard across both US, EU and beyond.

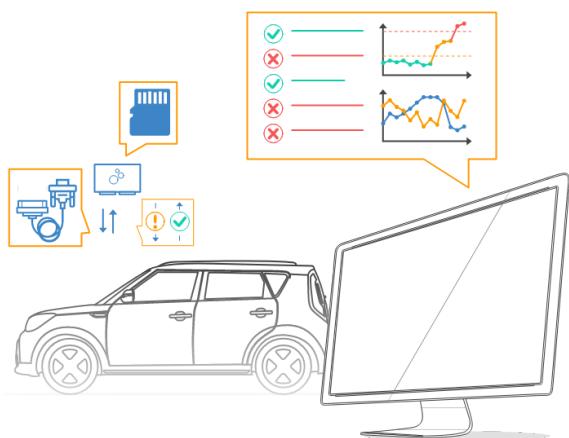
In either case, the basis for emissions-related diagnostics will be UDS, which will serve to simplify ECU programming as the emissions-related diagnostics can increasingly be implemented within the same UDS based structure as the manufacturer specific enhanced diagnostics.

FAQ: How to request/record UDS data

We have now gone through the basics of UDS and the CAN based transport protocol.

With this in place, we can provide some concrete guidance on how you can work with UDS data in practice. In particular, we will focus on how UDS can be used to log various data parameters - like state of charge (SoC) in electric vehicles.

Before the examples, we'll cover frequently asked questions on UDS data logging:

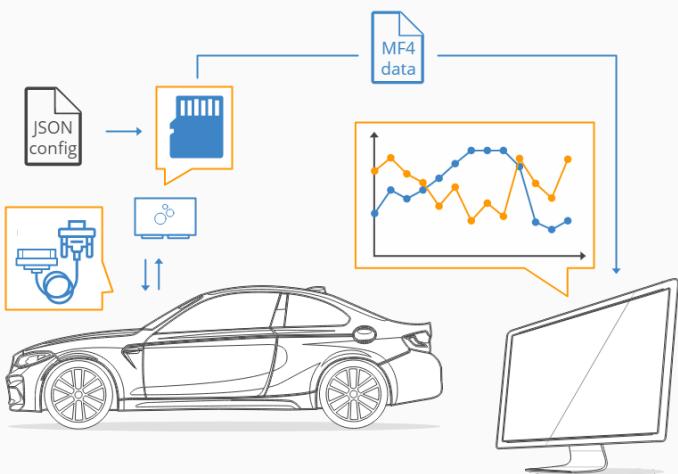


Can the CANedge record UDS data?

Yes, as we'll show further below, the CANedge can be configured to request UDS data. Effectively, the device can be configured to transmit up to 64 custom CAN frames as periodic or single shot frames. You can control the CAN ID, CAN data payload, period time and delay.

For single-frame UDS communication, you simply configure the CANedge with the request frame, which will trigger a single response frame from the ECU.

For multi-frame communication, you can again configure the CANedge with a request frame and then add the Flow Control frame as a separate frame to be transmitted X milliseconds after the request frame. By adjusting the timing, you can set this up so that the Flow Control is sent after the ECU has sent the First Frame response.



Note that the CANedge will record the UDS responses as raw CAN frames. You can then process and decode this data via your preferred software (e.g. [Vector tools](#)) or our [CAN bus Python API](#) to reassemble and decode the frames.

Note: In future firmware updates, we may enhance the transmit functionality to enable the CANedge to transmit custom CAN frames based on a trigger condition, such as receiving a specific frame. This would allow for sending the Flow Control frame with a set delay after receiving the First Frame, providing a simpler and more robust

implementation. With that said, the current functionality will serve to support most UDS communicated related to services 0x22 (Read Data by Identifier) and 0x19 (Read DTC Information).

What data can I log via UDS?

A very common use case for recording UDS data via 'standalone' data loggers will be to acquire diagnostic trouble code values, e.g. for use in diagnosing issues.

In addition to the trouble codes, UDS lets you request the 'current value' of various sensors related to each ECU. This allows e.g. vehicle fleet managers, researchers etc. to collect data of interest such as speed, RPM, state of charge, temperatures etc. from the car (assuming they know how to request/decode the data, as explained below).

Beyond the above, UDS can of course also be used for more low-level control of ECUs, e.g. resets and flashing of firmware, though such use cases are more commonly performed using a CAN bus interface - rather than a 'standalone' device.

Is UDS data proprietary - or are there public parameters?

Importantly, UDS is a manufacturer specific diagnostic protocol. In other words, while UDS provides a standardized structure for diagnostic communication, the actual 'content' of the communication remains proprietary and is in most cases only known to the manufacturer of a specific vehicle/ECU.

For example, UDS standardizes how to request parameter data from an ECU via the 'Read Data By Identifier' service (0x22). But it does not specify a list of standardized identifiers and interpretation rules. In this way, UDS differs from OBD2, where a public list of OBD2 PIDs enable almost anyone to interpret OBD2 data from their car.

With that said, vehicles that support WWH-OBD or OBDonUDS may support some of the usual OBD2 PIDs like speed, RPM etc via the usual PID values - but with a prefix of 0xF4 as shown in Example 1 below.

Generally, only the manufacturer (OEM) will know how to request proprietary parameters via UDS - and how to interpret the result. Of course, one exception to this rule is cases where companies or individuals successfully reverse engineer this information. Engaging in such reverse engineering is a very difficult task, but you can sometimes find public information and DBC files where others have done this exercise. Our [intro to DBC files](#) contain a list of public DBC/decoding databases.

Who benefits from logging UDS data?

Because of the proprietary nature of UDS communication, it is typically most relevant to automotive engineers working at OEMs. Tools like the CANedge CAN bus data logger allow these users to record raw CAN traffic from a vehicle - while at the same time requesting diagnostic trouble codes and specific parameter values via UDS.

Further, some after market users like vehicle fleet managers and even private persons can benefit from UDS assuming they are able to identify the reverse engineered information required to request and decode the UDS parameters of interest.

Logging UDS data will also become increasingly relevant assuming WWH-OBD gets rolled out as expected. Already today, WWH-OBD is used in EU heavy duty vehicles produced after 2014, meaning UDS communication will be relevant for use cases related to on-board diagnostics in these vehicles.

Central Gateway (CGW): Why log sensor data via UDS vs. CAN?

If you're looking to request UDS-based diagnostic trouble codes (DTC), you'll of course have to use UDS communication for this purpose. However, if your aim is to record current sensor values it is less clear.

Typically, data parameters of interest for e.g. vehicle telematics (speed, state of charge etc) will already be communicated between ECUs on the CAN bus in the form of raw CAN frames - without the need for a diagnostic tool requesting this information. That is because ECUs rely on communicating this information as part of their operation (as explained in our [intro to CAN bus](#)).

If you have direct access to the CAN bus, it would thus appear easier to simply log this raw CAN traffic and process it. If you are the vehicle manufacturer, you will know how to interpret this raw CAN data either way - and it'll be simpler to perform your device configuration and post processing in most cases. If you're in the aftermarket, it'll also be simpler to reverse engineer the raw CAN frames as you can focus on single frames - and avoid the request/response layer of complexity.

However, one key reason why UDS is frequently used for extracting sensor values despite the above is due to 'gateways'. Specifically, an increasing share of modern cars have started to block the access to the raw CAN bus data via the OBD2 connector. This is particularly often the case for German vehicles, as well as electric vehicles.

To record the existing CAN traffic in such a car, you would need to e.g. use a CANCrocodile adapter and 'snap' it onto the CAN low/high wiring harness. This in turn will require exposing the wiring harness by removing a panel, which is often prohibitive for many use cases. In contrast, the OBD2 connector gateways typically still allow for UDS communication - incl. sensor value communication.

A secondary - and more subtle - reason is that most reverse engineering work is done by 'OBD2 dongle manufacturers'. These develop tools that let you extract data across many different cars through the OBD2 connector. Increasingly, the only way for these dongles to get useful information through the OBD2 connector is through UDS communication, which drives a proportionally higher availability of information/databases related to UDS parameters vs. raw CAN parameters.

Does my vehicle support UDS?

Since most ECUs today support UDS communication, the answer is in "yes, most likely".

If you're the vehicle manufacturer, you will in most cases have the information required to construct UDS requests for whatever data you need - and you'll also know how to decode it.

For the specific case of recording WWH-OBD data in EU trucks, standardized DID information can be recorded by both OEMs and after-market users - similar to how you can record public OBD2 PIDs from cars/trucks.

Beyond the above, if you are in the after market and wish to record proprietary UDS information from a car/truck, it will be difficult. In this case, you would either have to contact the OEM (e.g. as a system integrator/partner) or identify the request/decoding information through reverse engineering. The latter is in practice impossible for most people.

In select cases you may be able to utilize public information shared on e.g. github to help in constructing UDS requests - and decoding the responses. Just keep in mind that public resources are based on reverse engineering efforts - and may risk being incorrect or outdated. You should therefore take all information with a significant grain of salt.

Why is UDS increasingly important?

If your use case involves recording data from cars produced between 2008 and 2018, you will most often be interested in data that can be collected via OBD2 data logging. This is because most ICE cars after 2008 support a large share of the public OBD2 parameter identifiers like speed, RPM, throttle position, fuel level etc.

However, the availability of OBD2 data is expected to decrease over time for multiple reasons.

First of all, as we explained in the previous section, WWH-OBD (based on UDS) or OBDonUDS are expected to gradually replace OBD2 as the de facto standard for vehicle diagnostics.

Second, with the rise of electric vehicles, legislated OBD2 is not necessarily supported at all. And even if an EV supports some OBD2 PIDs, you'll note from our OBD2 PID list that some of the most relevant EV parameters like State of Charge (SoC) and State of Health (SoH) are not available via OBD2. In contrast, UDS remains supported in most EVs and will provide access to a far broader range of data - although without the after-market convenience of a public list of UDS parameters (at least yet). It is expected that EV sales will overtake ICE car sales between [2030 and 2040](#) - and thus UDS communication will become increasingly relevant.

About the CANedge CAN logger

The [CANedge](#) lets you easily record CAN/UDS data to an 8-32 GB SD card. You can customize what CAN frames to send, incl. custom UDS requests and flow control frames. Data can be processed via [free software/API tools](#).



Example 1: Record single frame UDS data (Speed via WWH-OBD)

To show how UDS works in practice, we will start with a basic example. As outlined before, WWH-OBD is based on UDS - and is mandated in all EU trucks after 2014.

As part of this, many EU heavy duty trucks will let you request parameters like speed, RPM, fuel level etc in a way similar to how you'd request this information via OBD2 PID requests in a car - see our OBD2 intro and OBD2 data logger intro for details. However, under WWH-OBD (ISO 21745-2), the OBD2 PIDs are replaced by the WWH-OBD DIDs.



For service 01, WWH-OBD PIDs are equivalent to the OBD2 PIDs, except that 0xF4 is added in front. For example, the OBD2 PID Vehicle Speed is 0x0D - which becomes 0xF40D in the WWH-OBD context.

In this case, we will use a [CANedge2 CAN bus data logger](#) as our "tester" tool. This tool lets you record raw CAN bus data, as well as transmit custom CAN frames. To request WWH-OBD Vehicle Speed, we will use the UDS service 0x22 (Read Data by Identifier) and the Data Identifier 0xF40D. The request/response looks as below:

UDS on CAN - single frame communication example

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type	Legend
4.0435	18DB33F1	03 22 F4 0D AA AA AA AA	CANedge (client)	Single Frame (SF)	PCI field UDS SID (request) UDS DID
4.0468	18DAF100	04 62 F4 0D 32 AA AA AA	ECU (server)	First Frame (FF)	UDS SID (response) padding/unused payload (1 byte)

Communication flow details

Note how the request is sent with CAN ID 0x18DB33F1. This is the same 29-bit CAN ID that would be used to perform a functionally addressed OBD2 PID request in heavy duty vehicles and can be compared with the 11-bit 0x7DF CAN ID used for OBD2 requests in cars.

The response has CAN ID 0x18DAF100, an example of a physical response ID matching the IDs you'd see in regular OBD2 responses from heavy duty vehicles.

Let's break down the communication flow message payloads:

First, the CANedge2 sends a Single Frame (SF) request:

- The initial 4 bits of the PCI field equal the frame type (0x0 for SF)
- The next 4 bits of the PCI field equal the length of the request (3 bytes)
- The 2nd byte contains the Service Identifier (SID), in this case 0x22
- The 3rd and 4th bytes contain the DID for Vehicle Speed in WWH-OBD (0xF40D)
- The remaining bytes are padded

In response to this request, the truck responds with a Single Frame (SF) response:

- The 1st byte again reflects the PCI field (now with a length of 4 bytes)
- The 2nd byte is the response SID for Read Data by Identifier (0x62, i.e. 0x22 + 0x40)
- The 3rd and 4th bytes again contain the DID 0xF40D
- The 5th byte contains the value of Vehicle Speed, 0x32

Here we can use the same decoding rules as for ISO/SAE OBD2, meaning that the physical value of Vehicle Speed is simply the decimal form of 0x32 - i.e. 50 km/h. See also our OBD2 PID conversion tool.

If you are familiar with logging OBD2 PIDs, it should be evident that WWH-OBD requests are very similar, except for using the UDSonCAN payload structure for requests/responses.

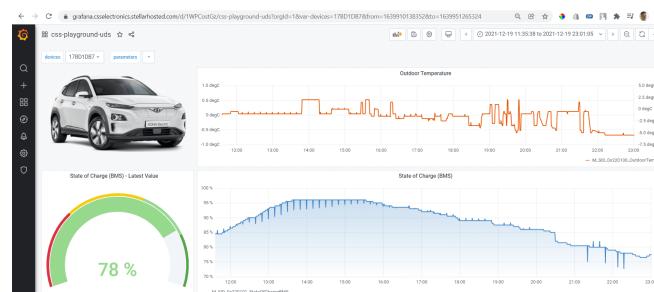
Example 2: Record & decode multi frame UDS data (SoC)

In this section, we illustrate how multi frame UDS communication works in the context of CAN ISO-TP.

Specifically, we will use the CANedge2 and the UDS service SID 0x22 (Read Data By Identifier) to request the current value of State of Charge (SoC%) from a Hyundai Kona electric vehicle.

First, the CANedge is configured to send two CAN frames:

1. A Single Frame (SF) request (period: 5000 ms, delay: 0 ms)
2. A Flow Control (FC) frame (period: 5000 ms, delay: 100 ms)



A subset of the resulting communication flow looks as below:

UDS on CAN - multiframe communication example

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type	Legend
1.0135	7E4	03 22 01 01 AA AA AA AA	CANedge (client)	Single Frame (SF)	PCI field
1.0228	7EC	10 3E 62 01 01 FF F7 E7	ECU (server)	First Frame (FF)	UDS SID (request)
1.0235	7E4	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)	UDS DID
1.0426	7EC	21 FF 96 3E 11 42 90 83	ECU (server)	Consecutive Frame (CF)	UDS SID (response)
1.0486	7EC	22 00 12 0E F3 17 16 16	ECU (server)	Consecutive Frame (CF)	padding/unused
1.0586	7EC	23 16 17 16 00 00 18 C3	ECU (server)	Consecutive Frame (CF)	FC block size, ST
1.0687	7EC	24 02 C2 0B 00 00 92 00	ECU (server)	Consecutive Frame (CF)	payload (59 bytes)
1.0787	7EC	25 01 04 50 00 01 02 0E	ECU (server)	Consecutive Frame (CF)	
1.0886	7EC	26 00 00 63 F3 00 00 60	ECU (server)	Consecutive Frame (CF)	
1.0986	7EC	27 74 00 3D A5 07 0D 01	ECU (server)	Consecutive Frame (CF)	
1.1086	7EC	28 7F 00 00 00 00 03 E8	ECU (server)	Consecutive Frame (CF)	
Reassembled UDS frame					
1.0135	7EC	62 01 01 FF F7 E7 00 00 00 ... 7F 00 00 00 00 03 E8			

Communication flow details

In the following we explore this communication between the CANedge and ECU in detail. First of all, the initial Single Frame (SF) request is constructed via the same logic as in our previous example - containing the PCI field, the SID 0x22 and the DID. In this case, we use the DID 0x0101. In response to the initial SF request, the targeted ECU sends a First Frame (FF) response:

- The initial 4 bits equal the frame type (0x1 for FF)
- The next 12 bits equal the data payload size, in this case 62 bytes (0x3E)
- The 3rd byte is the response SID for Read Data by Identifier (0x62, i.e. 0x22 + 0x40)
- The 4th and 5th bytes contain the Data Identifier (DID) 0x0101
- The remaining bytes contain the initial part of the data payload for the DID 0x0101

Following the FF, the tester tool now sends the Flow Control (FC) frame:

- The initial 4 bits equal the frame type (0x3 for FC)
- The next 4 bits specifies that the ECU should "Continue to Send" (0x0)
- The 2nd byte sets remaining frames to be sent without flow control or delay
- The 3rd byte sets the minimum consecutive frame separation time (ST) to 0

Once the ECU receives the FC, it sends the remaining Consecutive Frames (CF):

- The initial 4 bits equal the frame type (0x2 for CF)
- The next 4 bits equal the index counter, incremented from 1 up to 8 in this case
- The remaining 7 bytes of each CF contain the rest of the payload for the DID

Regarding proprietary UDS data

Part of the information used here is proprietary. In particular, it is generally not known what Data Identifier (DID) to use in order to request e.g. State of Charge from a given electric vehicle, unless you're the vehicle manufacturer (OEM). Further, as explained in the next section, it is not known how to decode the response payload.

However, various online resources exist e.g. on github, where enthusiasts create open source databases for specific parameters and certain cars (based on reverse engineering). The information we use for this specific communication is taken from one such database.

Regarding the CAN IDs used

In this case we use the CAN ID 0x7E4 to request data from a specific ECU, which in turn responds with CAN ID 0x7EC. This is known as a physically addressed request.

In contrast, functionally addressed request would use the CAN ID 0x7DF (or 0x18DB33F1 in heavy duty vehicles). Generally, request/response CAN IDs are paired (as per the table below) and you can identify the physical request ID corresponding to a specific physical response ID by subtracting the value 8 from the response ID. In other words, if an ECU responds via CAN ID 0x7EC, the physical request ID targeting that ECU would be 0x7E4 (as in our EV example).

Since you may not know what address to target initially, you can in some cases start by sending out a functional request using the CAN ID 0x7DF, in which case the relevant ECU should provide a positive First Frame response if the initial request payload is structured correctly. In some vehicles, you may be able to also send the subsequent Flow Control frame using the same CAN ID, 0x7DF, in order to trigger the ECU to send the remaining Consecutive Frames. However, some implementations may require that you instead utilize the physical addressing request ID for the Flow Control frame.

Implementing a request structure with dynamically updating CAN IDs may be difficult. If you're the manufacturer, you will of course know the relevant CAN IDs to use for sending physically addressed service requests. If not, you may perform an analysis using e.g. a CAN bus interface to identify what response CAN IDs appear when sending functionally addressed service requests - and using this information to construct your configuration.

On a separate note, ISO 15765-4 states that enhanced diagnostics requests/responses may utilize the legislated OBD2 CAN ID range as long as it does not interfere - which is what we are seeing in this specific Hyundai Kona example where the IDs 0x7EC/0x7E4 are used for proprietary data. See also the table from ISO 15765-4 for an overview of the legislated OBD CAN identifiers for use in functional and physical OBD PID requests.

Legislated OBD CAN identifiers (11-bit)

CAN ID	Description
0x7DF	Functionally addressed request sent by tester
0x7E0 to 0x7E7	Physical request from tester to ECU #1 to #8
0x7E8 to 0x7EF	Physical response from ECU #1 to #8 to tester

Legislated OBD CAN identifiers (29-bit)

CAN ID	Description
0x18DB33F1	Functionally addressed request sent by tester
0x18DAxxF1	Physical request from tester to ECU #xx
0x18DAF1xx	Physical response from ECU #xx to tester

Regarding timing parameters

In the above example, we generally focus on the sequence of CAN frames. The sequence is important: For example, if your tester tool sends the Flow Control frame before receiving the First Frame, the Flow Control frame will either be ignored (thus not triggering the Consecutive Frames) or cause an error.

However, in addition to this, certain timing thresholds will also need to be satisfied. For example, if your tester tool receives the First Frame from an ECU of a multi frame response, the ECU will 'time out' if the Flow Control frame is not sent within a set time period.

As a rule of thumb, you should configure your tester (e.g. the CANedge) so that the Flow Control frame is always sent after the First Frame response is received from the ECU (typically this happens within 10-50 ms from sending the initial request) - but in a way so that it is sent within a set time after receiving the First Frame (e.g. within 0-50 ms). For details on this, feel free to contact us.

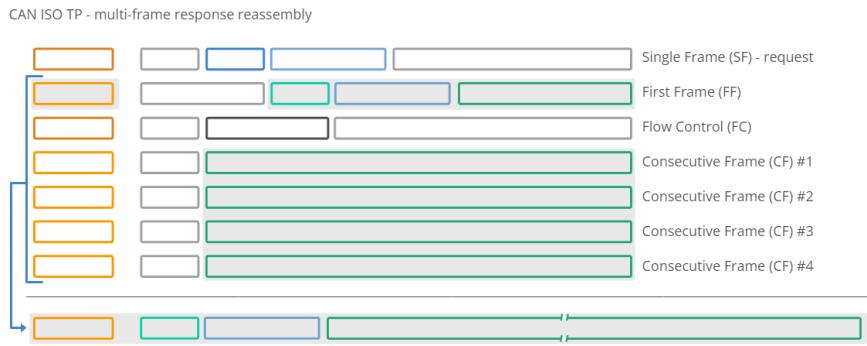
How to reassemble and decode multi-frame UDS data?

We've now shown how you can request/record a multi-frame UDS response to collect proprietary ECU sensor data. In order to extract 'physical values' like State of Charge, you need to know how to interpret the response CAN frames. As explained, the 'decoding' information is typically proprietary and only known to the OEM. However, in the specific case of the Hyundai Kona EV, we know the following about the SoC signal from [online resources](#):

- The signal is in the 8th byte of the data payload
- The signal is Unsigned
- The signal has a scale 0.5, offset 0 and unit "%"

So how do we use this knowledge to decode the signal?

First, we need to reassemble the segmented CAN frames. The result of this is shown in the previous communication example. Via reassembly, we get a "CAN frame" with ID 0x7EC and a payload exceeding 8 bytes. The payload in this case contains the SID in the 1st byte and DID in the 2nd and 3rd bytes.



You could process the reassembled CAN frame manually in e.g. Excel. However, we generally recommend to use [CAN databases \(DBC files\)](#) to store decoding rules. In this particular case, you can treat the reassembled CAN frame as a case of extended multiplexing. We provide an [example UDS DBC file](#) for the Hyundai Kona incl. State of Charge and temperature signals, which can be useful as inspiration.

Our [CAN bus Python API](#) enables reassembly & DBC decoding of multi-frame UDS responses - see our [API examples](#) repository for more details incl. the Hyundai Kona sample data.

```
BO_ 2028 Battery: 62 Vector XXX
SG_ M_SID_0x220101_StateofChargeBMS m257 : 56|8@1+ (0.5,0) [0|0] "%" Vector XXX
SG_ response m98M : 15|16@0+ (1,0) [0|0] "unit" Vector XXX
SG_ service M : 7|8@0+ (1,0) [0|0] "" Vector XXX

BO_ 1979 Temperature: 54 Vector XXX
SG_ M_SID_0x220100_IndoorTemp m256 : 64|8@1+ (0.5,-40) [0|0] "degC" Vector XXX
SG_ response m98 : 15|16@0+ (1,0) [0|0] "unit" Vector XXX
SG_ service M : 7|8@0+ (1,0) [0|0] "" Vector XXX
SG_ M_SID_0x220100_OutdoorTemp : 79|8@0+ (0.5,-40) [0|0] "" Vector XXX
SG_ M_SID_0x220100_Speed : 255|8@0+ (0.5,0) [0|0] "" Vector XXX

BA_DEF_ BO_ "VFrameFormat" ENUM  "StandardCAN","ExtendedCAN","StandardCAN_FD","ExtendedCAN_FD","J1939PG";
BA_DEF_ "ProtocolType" STRING ;
BA_DEF_DEF_ "VFrameFormat" "";
BA_DEF_DEF_ "ProtocolType" "";
BA_ "ProtocolType" "";
BA_ "VFrameFormat" BO_ 2028 0;

SG_MUL_VAL_ 2028 M_SID_0x220101_StateofChargeBMS response 257-257;
SG_MUL_VAL_ 2028 response service 98-98;
```

Example 3: Record the Vehicle Identification Number

The Vehicle Identification Number (aka VIN, chassis number, frame number) is a unique identifier code used for road vehicles. The number has been standardized and legally required since the 1980s - for details see the [VIN page on Wikipedia](#).

VIN - Vehicle Identification Number



A VIN consists of 17 ASCII characters and can be extracted on-request from a vehicle. This is useful in e.g. data logging or telematics use cases where a unique identifier is required for association with e.g. CAN bus log files.

CAN data bytes can be converted from HEX to ASCII via tables, online [HEX to ASCII converters](#), [Python packages](#) etc. For example, the byte 0x47 corresponds to the letter "G". Since a VIN is 17 bytes (17 ASCII characters) long, it does not fit into a single CAN frame, but has to be extracted via a multi frame diagnostic request/response as in Example 2. Further, the VIN is extracted differently depending on the protocol used.

Below we provide three examples on how to record the VIN.

3.1: How to record the VIN via OBD2 (SAE J1979)

To extract the Vehicle Identification Number from e.g. a passenger car using OBD2 requests, you use Service 0x09 and the PID 0x02:

VIN - Vehicle Identification Number request/response (OBD2)					
Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type	Legend
1.0135	7E0	02 09 02 AA AA AA AA AA	CANedge (client)	Single Frame (SF)	PCI field
1.0228	7E8	10 14 49 02 01 32 54 33	ECU (server)	First Frame (FF)	OBD service (request)
1.0235	7E0	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)	OBD PID
1.0426	7E8	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)	OBD service (response)
1.0486	7E8	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)	NODI padding/unused
Reassembled OBD2 frame					
1.0135	7E8	49 02 01 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37			FC block size, ST payload (17 bytes)
VIN = 2T3RFREV7DW108177					

Communication flow details

The logic of the frame structure is identical to Example 2, with the tester tool sending a Single Frame request with the PCI field (0x02), request service identifier (0x09) and data identifier (0x02).

The vehicle responds with a First Frame containing the PCI, length (0x014 = 20 bytes), response SID (0x49, i.e. 0x09 + 0x40) and data identifier (0x02). Following the data identifier is the byte 0x01 which is the Number Of Data Items (NODI), in this case 1 (see SAE J1979 or ISO 15031-5 for details).

The remaining 17 bytes equal the VIN and can be translated from HEX to ASC via the methods previously discussed.

3.2: How to record the VIN via UDS (ISO 14229-2)

To read the Vehicle Identification Number via UDS, you can use the UDS SID 0x22 and the DID 0xF190:

VIN - Vehicle Identification Number request/response (UDS on CAN)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	7E0	03 22 F1 90 AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	7E8	10 14 62 F1 90 32 54 33	ECU (server)	First Frame (FF)
1.0235	7E0	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	7E8	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)
1.0486	7E8	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)

Legend

- PCI field
- UDS SID (request)
- UDS DID
- UDS SID (response)
- padding/unused
- FC block size, ST payload (17 bytes)

Reassembled UDS frame

1.0135 7E8 62 F1 90 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37

VIN = 2T3RFREV7DW108177

Communication flow details

As evident, the request/response communication flow looks similar to the OBD2 case above. The main changes relate to the use of the UDS service 0x22 instead of the OBD2 service 0x09 - and the use of the 2-byte UDS DID 0xF190 instead of the 1 byte OBD2 PID 0x02. Further, the UDS response frame does not include the Number of Data Items (NODI) field after the DID, in contrast to what we saw in the OBD2 case.

3.3: How to record the VIN via WWH-OBD (ISO 21745-3)

If you need to request the Vehicle Identification Number from an EU truck after 2014, you can use the WWH-OBD protocol. The structure is identical to the UDS example, except that WWH-OBD specifies the use of the DID 0xF802 for the VIN.

VIN - Vehicle Identification Number request/response (WWH-OBD on CAN)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	18DB33F1	03 22 F8 02 AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	18DAF100	10 14 62 F8 02 32 54 33	ECU (server)	First Frame (FF)
1.0235	18DB33F1	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	18DAF100	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)
1.0486	18DAF100	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)

Legend

- PCI field
- UDS SID (request)
- UDS DID
- UDS SID (response)
- padding/unused
- FC block size, ST payload (17 bytes)

Reassembled WWH-OBD frame

1.0135 18DAF100 49 F8 02 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37

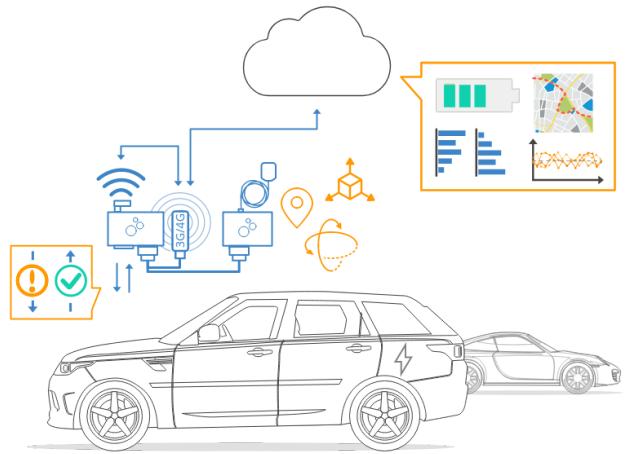
VIN = 2T3RFREV7DW108177

UDS data logging - applications

In this section, we outline example use cases for recording UDS data.

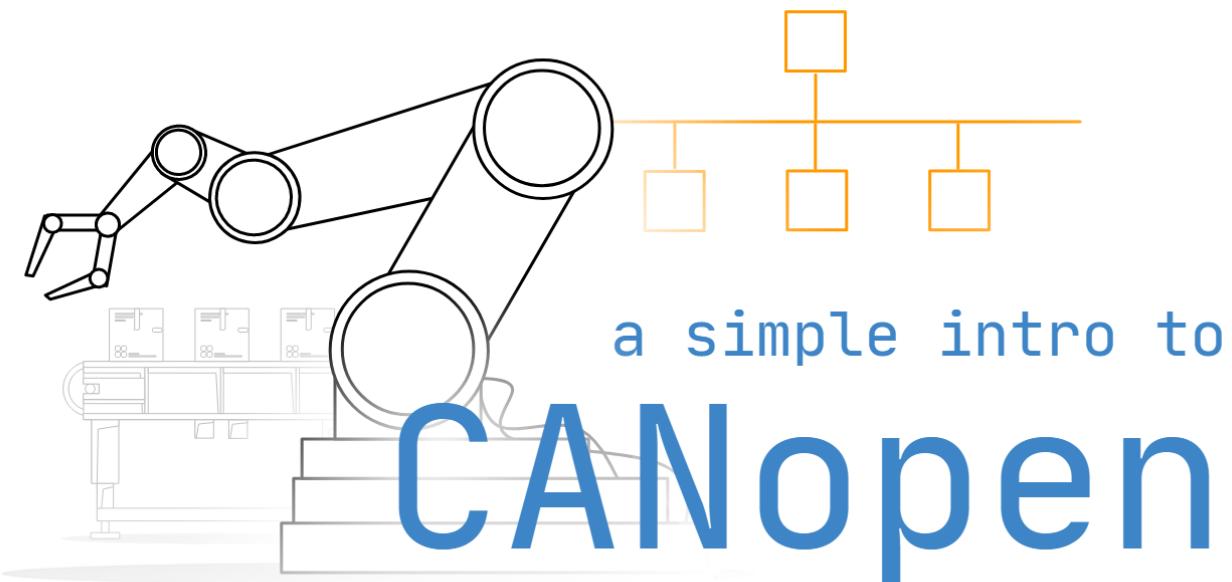
UDS telematics for prototype electric vehicles

As an OEM, you may need to get data on various sensor parameters from prototype EVs while they are operating in the field. Here, the CANedge2 can be deployed to request data on e.g. state of charge, state of health, temperatures and more by transmitting UDS request frames and flow control frames periodically. The data can e.g. be combined with GNSS/IMU data from a CANmod.gps and sent via a 3G/4G access point to your own cloud server for analysis via Vector tools, Python or MATLAB.



Training a predictive maintenance model

If you're looking to implement predictive maintenance across fleets of heavy duty vehicles, the first step is typically to "train your model". This requires large amounts of training data to be collected, including both sensor data (speed, RPM, throttle position, tire pressures etc) and "classification results" (fault / no fault). One way to obtain the latter is by periodically requesting diagnostic trouble codes from the vehicle, providing you with log files that combine both types of data over time. You can use the CANedge1 to collect this data offline onto SD cards, or the CANedge2 to automatically offload the data - e.g. when the vehicles return to stationary WiFi routers in garages.



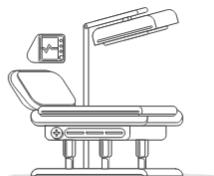
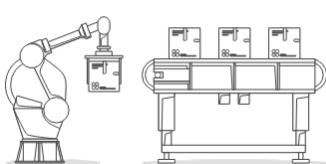
CANopen Explained - A Simple Intro

In this guide we introduce the CANopen protocol basics incl. the object dictionary, services, SDO, PDO and master/slave nodes. CANopen can seem complex - so this tutorial is a visual intro in layman's terms.

What is CANopen?

CANopen is a CAN based communication protocol.

The CANopen standard is useful as it enables off-the-shelf interoperability between devices (nodes) in e.g. industrial machinery. Further, it provides standard methods for configuring devices - also after installation. CANopen was originally designed for motion-oriented machine control systems. Today, CANopen is extensively used in motor control (stepper/servomotors) - but also a wide range of other applications:



Robotics

Automated robotics, conveyor belts & other [industrial machinery](#)

Medical

X-ray generators, injectors, patient tables & dialysis devices

Automotive

Agriculture, railway, trailers, [heavy duty](#), [mining](#), [marine](#) & more

CANopen - higher layer protocol

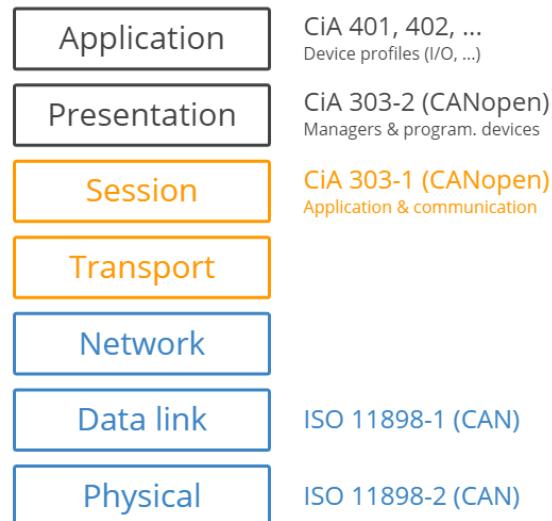
The following is important to understand:

CANopen is a "higher layer protocol" based on CAN bus.

This means that CAN bus (ISO 11898) serves as the 'transport vehicle' (like a truck) for CANopen messages (like containers).

You can view CANopen from a 7-layer [OSI model](#).

7 layer OSI model



CANopen in OSI model context

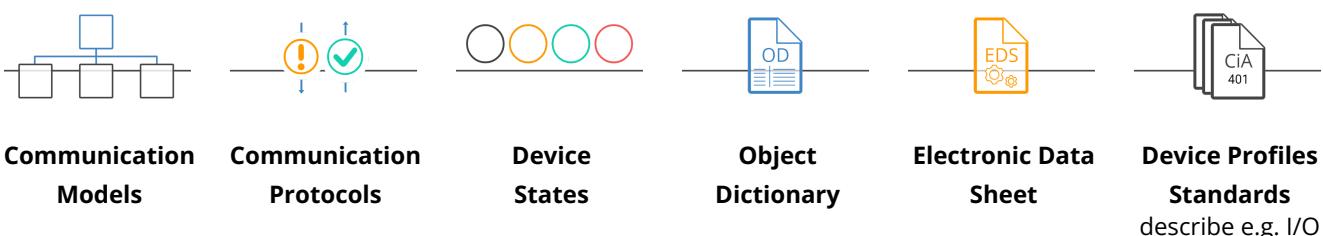
The OSI model is a conceptual model standardizing communication functions across diverse communication technologies. Lower layers describe basic communication (e.g. raw bit streams), while higher layers describe things like segmentation of long messages and services like initiating, indicating, responding, and confirming of messages. CAN bus represents the two lowest layers (1: Physical, 2: Data Link). This means that CAN simply enables the transmission of frames with an 11 bit CAN ID, a remote transmission (RTR) bit and 64 data bits (fields relevant to higher-layer protocols). In other words, CAN bus plays the same role in CANopen as it does in e.g. the [J1939 protocol](#). As evident above, CANopen implements the 7th layer of the OSI model (Application) via a set of standards. As part of this, it adds several important concepts that we detail below. It's worth noting, that CANopen could also be adapted to other data link layer protocols than CAN (e.g. EtherCAT, Modbus, Powerlink). To fully understand CAN bus vs. CANopen, see also our [CAN bus intro tutorial](#).

CANopen FD

In this article we primarily focus on CANopen based on Classical CAN. However, it is worth noting that as [CAN FD](#) is being rolled out, CANopen FD may play an increasingly important role as the next generation of CANopen. For details, see the overview by CAN in Automation on [CANopen FD](#).

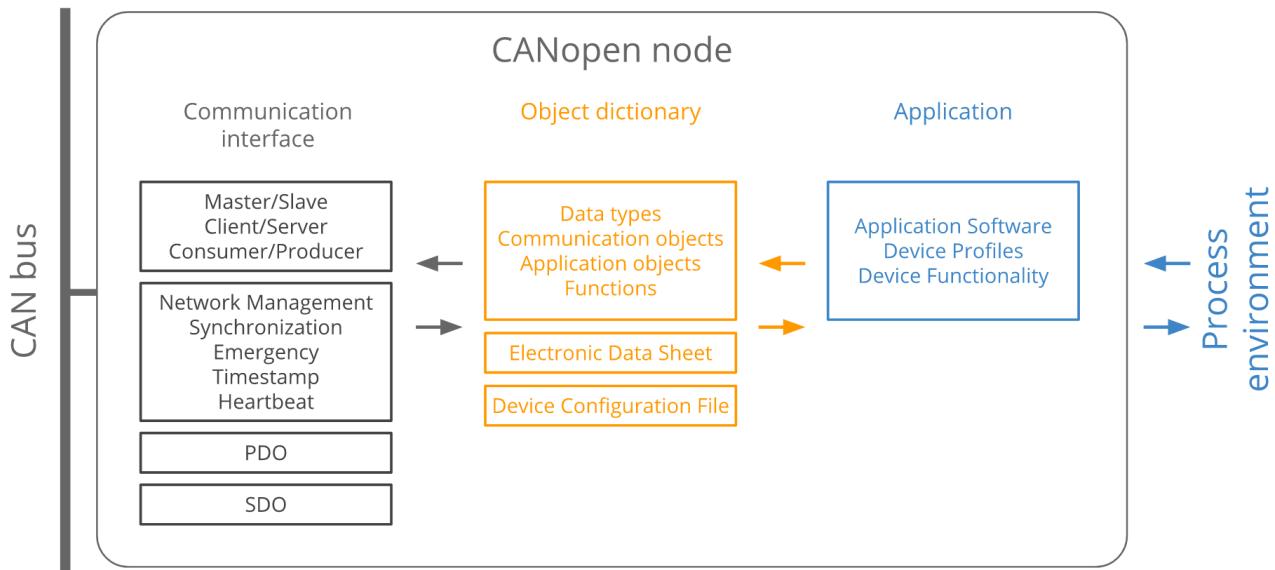
Six core CANopen concepts

Even if you're familiar with CAN bus and e.g. J1939, CANopen adds a range of important new concepts:



There are 3 models for device/node communication: Master/slave, client/server and producer/consumer	Protocols are used in communication, e.g. configuring nodes (SDOs) or transmitting real-time data (PDOs)	A device supports different states. A 'master' node can change state of a 'slave' node - e.g. resetting it	Each device has an OD with entries that specify e.g. the device config. It can be accessed via SDOs	The EDS is a standard file format for OD entries - allowing e.g. service tools to update devices	modules (CiA 401) and motion-control (CiA 402) for vendor independence
--------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------	------------------------------------------------------------------------

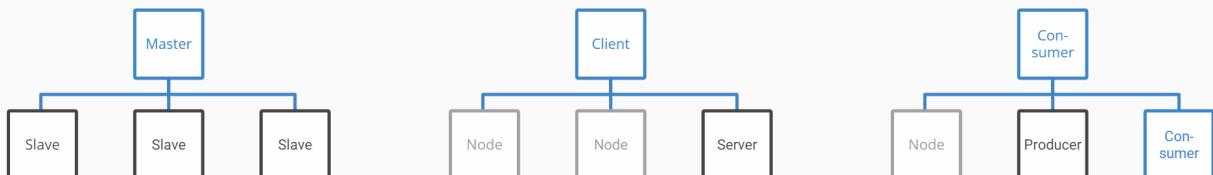
The below illustration shows how the CANopen concepts link together - and we will detail each below:



CANopen communication basics

In a CANopen network, several devices need to communicate. For example, in an industrial automation setup you may have a robot arm with multiple servomotor nodes and a control interface/PC node. To facilitate communication, three models exist within CANopen - each closely linked to the CANopen protocols that we look at shortly. See below for a brief introduction:

CANopen communication models



Master/Slave

One node (e.g. the control interface) acts as application master or host controller. It sends/requests data from the slaves (e.g. the servo motors). This is used in e.g. diagnostics or state management.

Client/Server

A client sends a data request to a server, which replies with the requested data. Used e.g. when an application master needs data from the OD of a slave. A read from a server is an "upload", while a write is

Consumer/Producer

Here, the producer node broadcasts data to the network, which is consumed by the consumer node. The producer either sends this data on request (pull model) or without a specific request (push model).

There can be 0-127 slaves in standard applications. Note that in a single CANopen network, there can be different host controllers sharing the same data link layer.
Service example: [NMT](#)

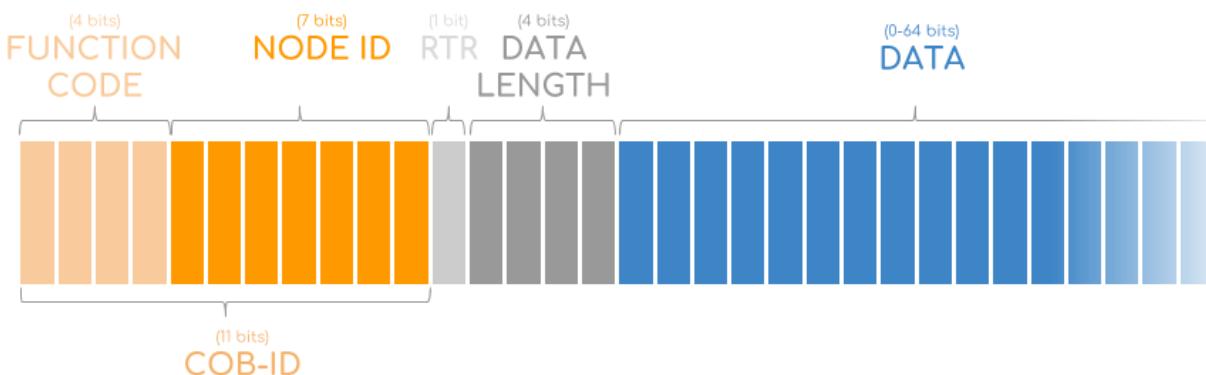
a "download" (the terminology takes a "server side" perspective).
Service example: [SDO](#)

Service example: [Heartbeat](#)

As evident, the models are practically identical, but we distinguish between them for terminology consistency.

The CANopen frame

To understand CANopen communication, it is necessary to break down the CANopen CAN frame:



The 11-bit CAN ID is referred to as the Communication Object Identifier (COB-ID) and is split in two parts: By default, the first 4 bits equal a function code and the next 7 bits contain the node ID.

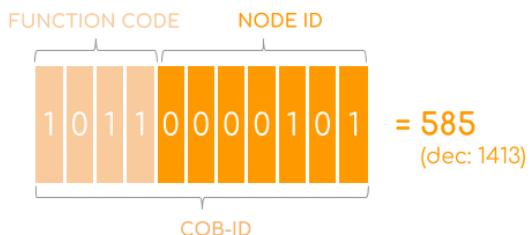
To understand how the COB-ID works, let's take outset in the pre-defined allocation of identifiers used in simple CANopen networks (see the table). Note: We'll refer to COB-IDs and Node IDs in HEX below. As evident, the COB-IDs (e.g. 381, 581, ...) are linked to the communication services (transmit PDO 3, transmit SDO, ...). As such, the COB-ID details which node is sending/receiving data - and what service is used.

COMMUNICATION OBJECT	FUNCTION CODE (4 bit, bin)	NODE IDs (7 bit, bin)	COB-IDs (hex)	COB-IDs (dec)	#
1 NMT	0000	0000000	0	0	1
2 SYNC	0001	0000000	80	128	1
3 EMCY	0001	0000001-1111111	81 - FF	129 - 255	127
4 TIME	0010	0000000	100	256	1
5 Transmit PDO 1	0011	0000001-1111111	181 - 1FF	385 - 511	127
Receive PDO 1	0100	0000001-1111111	201 - 27F	513 - 639	127
Transmit PDO 2	0101	0000001-1111111	281 - 2FF	641 - 767	127
Receive PDO 2	0110	0000001-1111111	301 - 37F	769 - 895	127
Transmit PDO 3	0111	0000001-1111111	381 - 3FF	897 - 1023	127
Receive PDO 3	1000	0000001-1111111	401 - 47F	1025 - 1151	127
Transmit PDO 4	1001	0000001-1111111	481 - 4FF	1153 - 1279	127
Receive PDO 4	1010	0000001-1111111	501 - 57F	1281 - 1407	127
6 Transmit SDO	1011	0000001-1111111	581 - 5FF	1409 - 1535	127
Receive SDO	1100	0000001-1111111	601 - 67F	1537 - 1693	127
7 HEARTBEAT	1110	0000001-1111111	701 - 77F	1793 - 1919	127

Example

A CANopen device with node ID 5 would transmit an SDO via the 11-bit CAN ID 585.

This corresponds to a binary function code of 1011 and a node ID of 5 (binary: 0000101) - see the illustration.



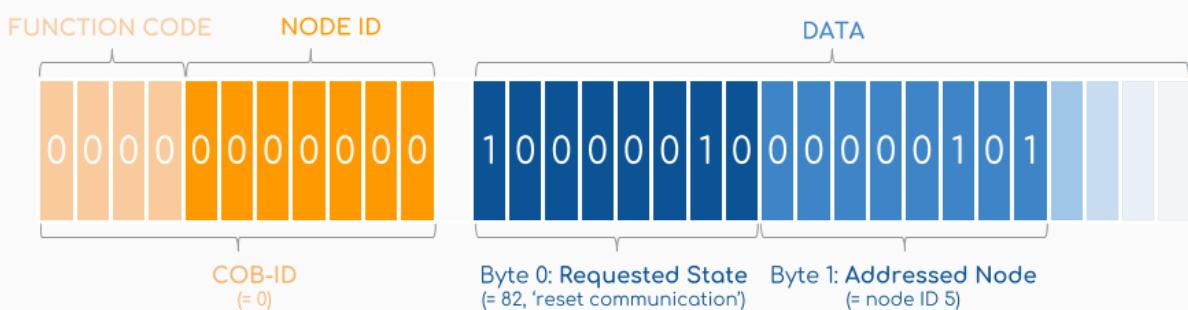
CANopen communication protocols/services

Below we briefly outline the 7 service types mentioned, incl. how they utilize the 8 CAN frame data bytes.

#1 Network Management (NMT)

What is it? The NMT service is used for controlling the state of CANopen devices (e.g. pre-operational, operational, stopped) by means of NMT commands (e.g. start, stop, reset).

How does it work? To change state, the NMT master sends a 2-byte message with CAN ID 0 (i.e. function code 0 and node ID 0). All slave nodes process this message. The 1st CAN data byte contains the requested state - while the 2nd CAN data byte contains the node ID of the targeted node. The node ID 0 indicates a broadcast command.



Possible commands include transition to operational (state 01), to stopped (state 02), pre-operational (state 80) as well as reset application (81) and reset communication (82).

#2 Synchronization (SYNC)

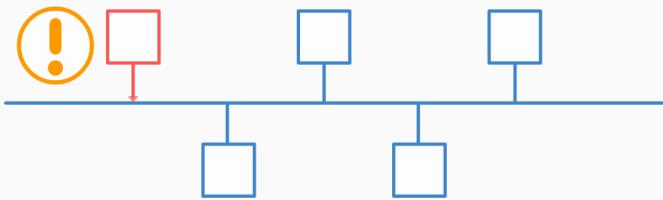
What is it? The SYNC message is used e.g. to synchronize the sensing of inputs and actuation of several CANopen devices - typically triggered by the application master.

How does it work? The application master sends the SYNC message (COB ID 080) to the CANopen network (with or without SYNC counter). Multiple slave nodes may be configured to react to the SYNC and respond by transmitting input data captured at the very same time or by setting the output at the very same time as the nodes participating in the synchronous operation. Using the SYNC counter several groups of synchronously operating devices can be configured.

#3 Emergency (EMCY)

What is it? The emergency service is used in case a device experiences a fatal error (e.g. a sensor failure), allowing it to indicate this to the rest of the network.

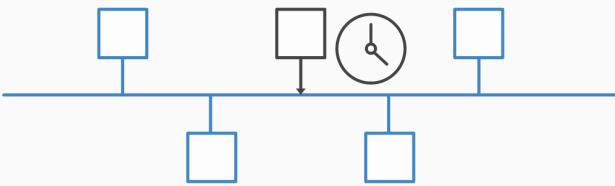
How does it work? The affected node sends a single EMCY message (e.g. with COB-ID 085 for node 5) to the network with high priority. The data bytes contain information about the error, which can be looked up for details.



#4 Timestamp (TIME) [PDO]

What is it? With this communication service a global network time can be distributed. The TIME service contains a 6-byte date & time information.

How does it work? An application master sends out the TIME message with CAN ID 100, where the initial 4 data bytes contain time in ms after midnight and the next 2 bytes contain the number of days since January 1, 1984.



#5 Process Data Object [PDO]

What is it? The PDO service is used to transmit real-time data between devices - e.g. measured data such as position or command data such as torque requests. In this respect it is similar to e.g. broadcasted data parameters in J1939.

How does it work? We'll deep dive on this [further below](#).

#6 Service Data Object [SDO]

What is it? The SDO services are used to access/change values in the object dictionary of a CANopen device - e.g. when an application master needs to change certain configurations of a CANopen device.

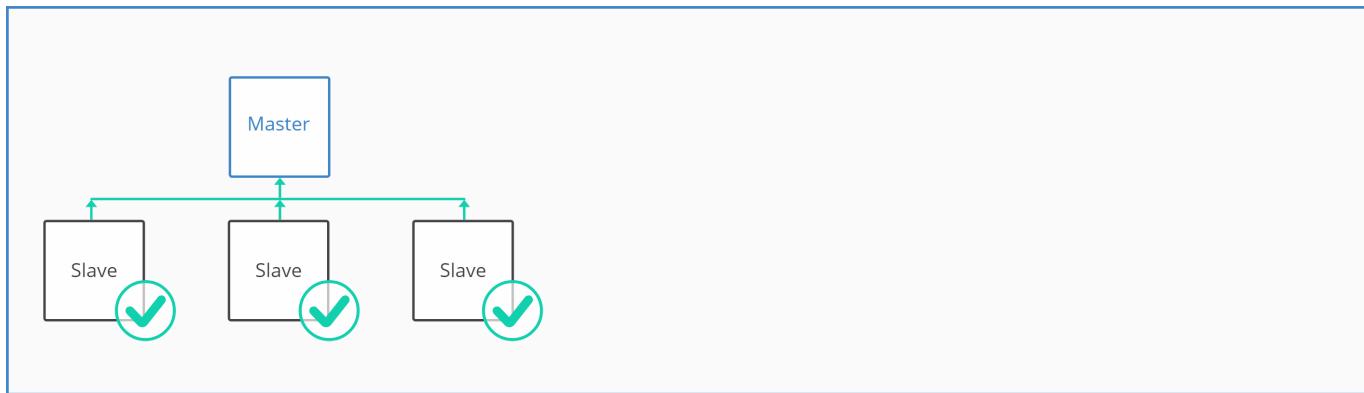
How does it work? We'll deep dive on this [further below](#).

#7 Node monitoring (Heartbeat) [SDO]

What is it? The Heartbeat service has two purposes: To provide an 'alive' message and to confirm the NMT command.

How does it work? An NMT slave device periodically sends (e.g. every 100ms) the Heartbeat message (e.g. with CAN ID 705 for node 5) with the node's "state" in the 1st data byte

The "consumer" of the Heartbeat message (e.g. the NMT master and optionally any other device) then reacts if no message is received in a certain time limit.



The PDO and SDO services are particularly important as they form the basis for most CANopen communication. Below we deep-dive on each of these - but first we need to introduce a core concept of CANopen: The object dictionary.

CANopen Object Dictionary

All CANopen nodes must have an object dictionary (OD) - but what is it? The object dictionary is a standardized structure containing all parameters describing the behavior of a CANopen node. OD entries are looked up via a 16-bit index and 8-bit subindex. For example, index 1008 (subindex 0) of a CANopen-compliant node OD contains the node *device name*.

Specifically, an entry in the object dictionary is defined by attributes:

- Index: 16-bit base address of the object
- Object name: Manufacturer device name
- Object code: Array, variable, or record
- Data type: E.g. VISIBLE_STRING, or UNSIGNED32 or Record Name
- Access: rw (read/write), ro (read-only), wo (write-only)
- Category: Indicates if this parameter is mandatory/optional (M/O)

OD standardized sections

The object dictionary is split into standardized sections where some entries are mandatory and others are fully customizable. Importantly, OD entries of a device (e.g. a slave) can be accessed by another device (e.g. a master) via CAN using e.g. SDOs. For example, this might let an application master change whether a slave node logs data via a specific input sensor - or how often the slave sends a heartbeat.

OD INDEX (16 bits, hex)	DESCRIPTION
0000	Reserved
0001 - 025F	Data types
0260 - 0FFF	Reserved
1000 - 1FFF	Communication object area
2000 - 5FFF	Manufacturer specific area
6000 - 9FFF	Device profile specific area
A000 - BFFF	Interface profile specific area
C000 - FFFF	Reserved

Link to Electronic Data Sheet and Device Configuration File

To understand the OD, it is helpful to look at the 'human-readable form': The Electronic Data Sheet and Device Configuration File.



The Electronic Data Sheet (EDS)

In practice, configuring/managing complex CANopen networks will be done using adequate software tools. To simplify this, the CiA 306 standard defines a human-readable (and machine friendly) INI file format, acting as a "template" for the OD of a device - e.g. the "ServoMotor3000". This EDS is typically provided by the vendor and contains info on all device objects (but not values).

Device Configuration File (DCF)

Assume a factory has bought a ServoMotor3000 to integrate into their conveyor belt. In doing so, the operator edits the device EDS and adds specific parameter values and/or changes the names of each object described in the EDS. In doing so, the operator effectively creates what is known as a Device Configuration File (DCF). With this in place, the ServoMotor3000 is ready for integration into the specific CANopen network on-site.

EDS and DCF example

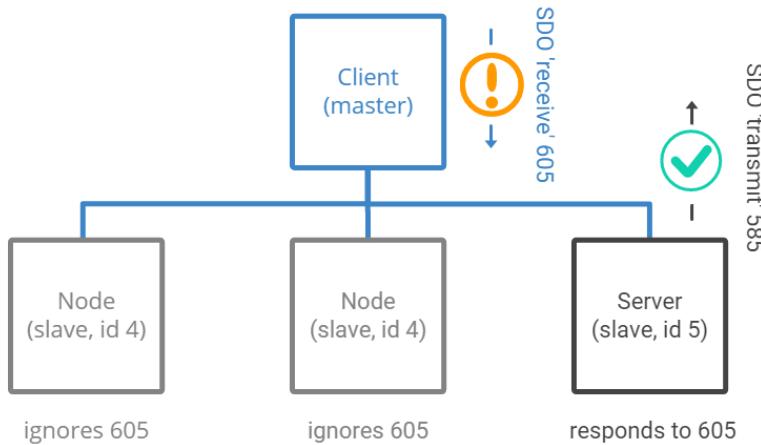
Reviewing real EDS/DCF examples is one of the best ways to really understand the object dictionary of CANopen - see e.g. the difference between an EDS and DCF object entry below. We recommend checking out the [CiA 306](#) standard to gain a deeper understanding of the OD, EDS and DCF with practical examples.

EDS OBJECT 1006	Comments	DCF OBJECT 1006
<pre>; value for object 1006h [1006] SubNumber=0 ParameterName=ParaName ObjectType=0x7 DataType=0x0007 LowLimit=1000 HighLimit=100000 DefaultValue=20000 AccessType=ro PDOMapping=0</pre>	<i>Comment field</i> <i>16 bit object index</i> <i>Number of sub indices</i> <i>The object's name</i> <i>Object type (7 = VAR)</i> <i>OD index of data type</i> <i>Minimum (if applicable)</i> <i>Maximum (if applicable)</i> <i>Used if no value given</i> <i>Access: ro = read only</i> <i>PDO mappable? (0 = no)</i> <i>Specific parameter value</i>	<pre>; value for object 1006h [1006] SubNumber=0 ParameterName=ParaName ObjectType=0x7 DataType=0x0007 LowLimit=1000 HighLimit=100000 DefaultValue=20000 AccessType=ro PDOMapping=0 ParameterValue=15000</pre>

As mentioned, the DCF is typically created upon device integration. However, often it will be necessary to read and/or change the object values of a node after initial configuration - this is where the CANopen SDO service comes into play.

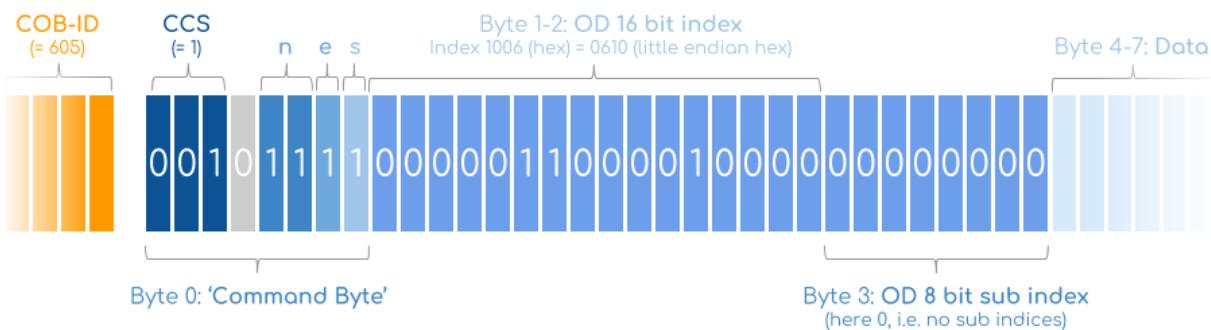
SDO - configuring the CANopen network

What is the SDO service? The SDO service allows a CANopen node to read/edit values of another node's object dictionary over the CAN network. As mentioned under 'communication models', the CANopen SDO services utilize a "client/server" behavior. Specifically, an SDO "client" initiates the communication with one dedicated SDO "server". The purpose can be to update an OD entry (called an "SDO download") or read an entry ("SDO upload"). In simple master/slave networks, the node with NMT master functionality acts as the client for all NMT slave nodes reading or writing to their ODs.



Example: Client node SDO download

The client node can initiate an SDO download to node 5 by broadcasting below CAN frame - which will trigger node 5 (and be ignored by other nodes, see above illustration). The SDO 'receive' (i.e. request) CAN frame looks as below:



SDO message variables explained

- First, COB-ID 605 reflects the use of an 'SDO receive' (COB-ID 600 + node ID).
- The CCS (client command specifier) is the transfer type (e.g. 1: Download, 2: Upload)
- n is the #bytes in data bytes 4-7 that do not contain data (valid if e & s are set)
- If set, e indicates an "expedited transfer" (all data is in a single CAN frame)
- If set, s indicates that data size is shown in n
- Index (16 bits) and subindex (8 bits) reflect the OD address to be accessed
- Finally, bytes 4-7 contain the data to be downloaded to node 5

CANopen SDO example comments

Once the CAN frame is sent by the master (client), the slave node 5 (server) responds via an 'SDO transmit' with COB-ID 585. The response contains the index/subindex and 4 empty data bytes. Naturally, if the client node requested an upload instead (i.e. reading data from the node 5 OD), node 5 would respond with the relevant data contained in bytes 4-7. A few comments:

- As evident, each SDO uses 2 identifiers, creating an "SDO channel"
- The example is simplified as it's "expedited" (data is contained in the 4 bytes)
- For larger data scenarios, SDO [segmentation/block transfers](#) can be used

SDOs are flexible, but carry a lot of overhead - making them less ideal for real-time operational data. This is where the PDO comes in.

PDO - operating the CANopen network

First of all: What is the CANopen PDO service? The CANopen PDO service is used for effectively sharing real-time operational data across CANopen nodes. For example, the PDO would carry pressure data from a pressure transducer - or temperature data from a temperature sensor. But wait: Can't the SDO service just do this?

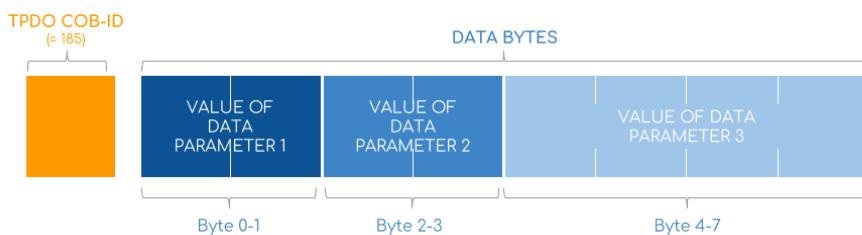
Yes, in principle the SDO service could be used for this. However, a single SDO response can only carry 4 data bytes due to overhead (command byte and OD addresses). Further, let's say a master node needs two parameter values (e.g. "SensTemp2" and "Torque5") from Node 5 - to get this via SDO, it would require 4 full CAN frames (2 requests, 2 responses).



In contrast, a PDO message can contain 8 full bytes of data - and it can contain multiple object parameter values within a single frame. Thus, what would require at least 4 frames with SDO could potentially be done with 1 frame in the PDO service. The PDO is often seen as the most important CANopen protocol as it carries the bulk of information.

How does the CANopen PDO service work?

For PDOs, the consumer/producer terminology is used. Thus, a producer 'produces data', which it transmits to a 'consumer' (master) using a transmit PDO (TPDO). Conversely, it may receive data from the consumer via a receive PDO (RPDO). Producer nodes may e.g. be configured to respond to a SYNC trigger broadcasted by the consumer every 100 ms. Node 5 may then e.g. broadcast below transmit PDO with COB-ID 185:



Note how the data bytes are packed with 3 parameter values. These values reflect real-time data of specific OD entries of node 5. The nodes that use this information (the consumers) of course need to know how to interpret the PDO data bytes.

PDO service vs. J1939 PGNs and SPNs

Isn't the PDO service a bit similar to J1939 PGNs & SPNs? Yes, to some extent this is similar to how a specific [J1939 parameter group \(PG\)](#) will contain multiple SPNs/signals (aka data parameters) in the 8 data bytes. The J1939 CAN frame does not need to waste data bytes on "decoding" information because this is known by the relevant nodes (and by external tools via e.g. [J1939 DBC files](#) or the J1939 PDF standards). The wrinkle is that in CANopen, these 'PDO mappings' are often configurable and can be changed during the creation of the DCF and/or via the SDO service. For more details on PDOs, see [this article](#) (p. 5).

CANopen data logging - use case examples

Since CANopen is a CAN-based protocol, it is possible to record raw CANopen frames using a CAN bus data logger. As an example, the [CANedge](#) lets you record CANopen data to an 8-32 GB SD card.

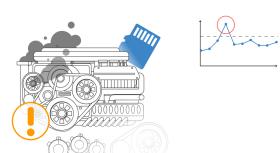
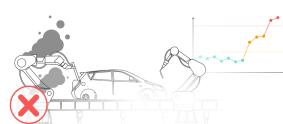
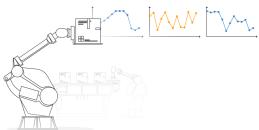
Simply connect it to your application to start logging - and process the data via [free software/APIs](#). [Learn more!](#)



Raw data decoding via CANopen DBC files

As CANopen is based on CAN bus, you can store your CANopen decoding rules in the standardized CAN database format, a [CAN DBC file](#). With this, you can directly decode your raw CANopen data in open source software/API tools for the CANedge, including the [asammdf GUI](#), [telematics dashboards](#) and [Python API tools](#). Note that you may not have a CANopen DBC file readily available, but rather your CANopen frame decoding rules may be stored in your EDS/DCF or in a PDF. In such a case you can take outset in our DBC intro to learn how to construct a CANopen DBC file from scratch using various free editor tools.

Solutions like the CANedge enable several CANopen logging use cases:



Logging CANopen node data

Generally, logging CANopen data can be used to e.g. analyze operational data. [WiFi CAN loggers](#) can also be used for e.g. over-the-air SDOs [learn more](#)

Warehouse fleet management

CANopen is often used in EV forklifts/AGVs in warehouses, where monitoring e.g. SoC helps reduce breakdowns and improve battery life [learn more](#)

Predictive maintenance

Industrial machinery can be monitored via [IIoT CAN loggers](#) in the cloud to predict and avoid breakdowns based on the CANopen data [learn more](#)

Machinery diagnostic blackbox

A [CAN logger](#) can serve as a 'blackbox' for industrial machinery, providing data for e.g. disputes or rare issue diagnostics [learn more](#)



CAN FD Explained - A Simple Intro

In this guide we introduce CAN FD (CAN Flexible Data-rate) - incl. CAN FD frames, overhead & efficiency, example applications and logging use cases.

Why CAN FD?

The CAN protocol has been around since 1986 and it's popular: Practically any machine that moves utilizes CAN today - whether it's cars, trucks, boats, planes or robots.

But with the rise of modern technology, the "["Classical" CAN protocol](#)" (official term used in ISO 11898-1:2015) is pressured:

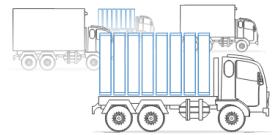
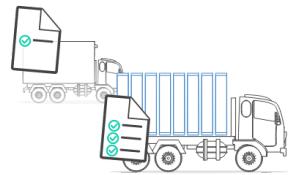
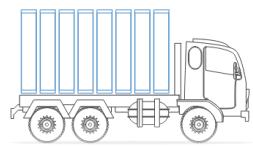
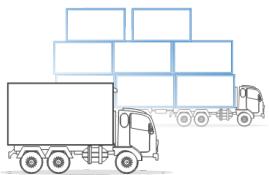
- A rise in vehicle functionality is driving an explosion in data
- Networks are increasingly limited by the 1-Mbit/s bandwidth
- To cope, OEMs create complex & costly workarounds

Specifically, Classical CAN struggles with substantial overhead (>50%) as each CAN data frame can only contain 8 data bytes. Further, the network speed is limited to 1 Mbit/s, restricting the implementation of data-producing features. CAN FD resolves these issues - making it future-proof.

What is CAN FD?

The CAN FD protocol was pre-developed by [Bosch](#) (with industry experts) and released in 2012. It was improved through standardization and is now in ISO 11898-1:2015. The original Bosch CAN FD version (non-ISO CAN FD) is incompatible with ISO CAN FD.

CAN FD offers four major benefits:



#1 Increased length

CAN FD supports up to 64 data bytes per data frame vs. 8 data bytes for Classical CAN. This reduces the protocol overhead and leads to an improved protocol efficiency.

#2 Increased speed

CAN FD supports dual bit rates: The nominal (arbitration) bit-rate limited to 1 Mbit/s as in Classical CAN - and the data bit-rate, which depends on the network topology/transceivers. In practice, data bit-rates up to 5 Mbit/s are achievable.

#3 Better reliability

CAN FD uses an improved cyclic redundancy check (CRC) and the "protected stuff-bit counter", which lower the risk of undetected errors. This is e.g. vital in safety-critical applications like vehicles and industrial automation.

#4 Smooth transition

CAN FD and Classical CAN only ECUs can be mixed under certain conditions. This allows for a gradual introduction of CAN FD nodes, greatly reducing costs and complexity for OEMs.

In practice, CAN FD can improve network bandwidth by [3-8x](#) vs Classical CAN, creating a simple solution to the rise in data.

How does CAN FD work?

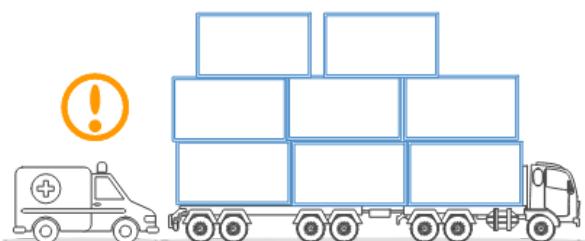
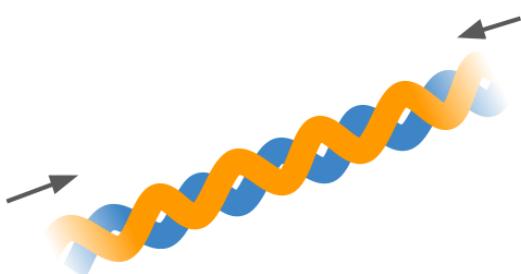
So CAN FD seems pretty simple: Speed up the data transmission and pack more data into each message, right? In practice, however, it's not that straightforward. Below we outline the main challenges that the CAN FD solution had to solve.

Two key challenges

Before looking at the CAN FD data frame, it's key to understand two core parts of Classical CAN that we want to maintain:

#1 Avoid critical message delays

Why not simply pack Classical CAN frames with 64 bytes of data? Doing so would reduce overhead and simplify message interpretation. However, if the bit-rate is unchanged, this would also block the CAN bus for longer, potentially delaying mission-critical higher-priority data frames.



#2 Maintain practical CAN wire lengths

So, more speed is needed to send more data per message. But why not speed up the entire CAN message (rather than just the data phase)? This is due to "arbitration": If 2+ nodes transmit data simultaneously, arbitration determines which node takes priority. The "winner" continues sending (without delay), while the other nodes "back off" during the data transmission.

Regarding bit time

During arbitration, a "bit time" provides sufficient delay between each bit to allow every node on the network to react. To be certain that every node is reached within the bit time, a CAN network running at 1 Mbit/s needs to have a maximum length of 40 metres (in practice 25 metres). Speeding up the arbitration process would reduce the maximum length to unsuitable levels.

On the other hand, after arbitration there's an "empty highway" - enabling high speed during the data transmission (when there is just one node driving the bus-lines). Before the ACK slot - when multiple nodes acknowledge the correct reception of the data frame - the speed needs to be reduced to the nominal bit-rate.

In short, it is necessary to find a way to only increase the speed during the data transmission.

Solution: The CAN FD frame

The CAN FD protocol introduces an adjusted CAN data frame to enable the extra data bytes and flexible bit-rates. Below we compare an 11-bit Classical CAN frame vs. an 11-bit CAN FD frame (29-bit is also supported):



Below we go through the differences step-by-step:

CAN FD frame fields explained

RTR vs. RRS: The Remote Transmission Request (RTR) is used in Classical CAN to identify data frames and corresponding [remote frames](#). In CAN FD, remote frames are not supported at all - the Remote Request Substitution (RRS) is always dominant (0).

r0 vs. FDF: In Classical CAN, r0 is reserved and dominant (0). In CAN FD, it's named FDF and recessive (1). After the r0/FDF bit, the CAN FD protocol adds "3 new bits". Note that nodes that are not CAN FD capable produce an error frame after the FDF bit.

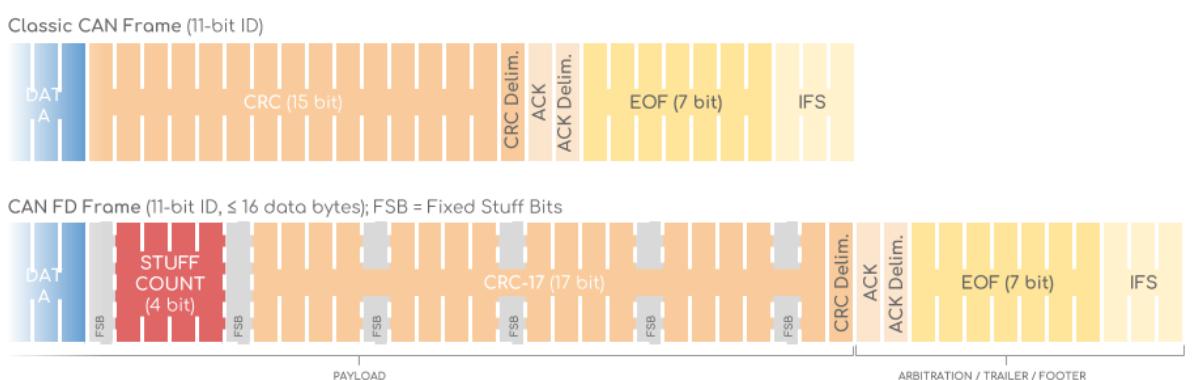
res: This new reserved bit plays the same role as r0 - i.e. it may in the future be set to recessive (1) to denote a new protocol.

BRS: The Bit Rate Switch (BRS) can be dominant (0), meaning that the CAN FD data frame is sent at the arbitration rate (i.e. up to max 1 Mbit/s). Setting it to recessive (1) means that the remaining part of the data frame is sent at a higher bit rate (up to [5 Mbit/s](#)).

ESI: The Error Status Indicator (ESI) bit is by default dominant (0), i.e.'error active'. If the transmitter becomes 'error passive' it'll be recessive (1) to indicate it is in error passive mode.

DLC (bin)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
DLC (dec)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Classic CAN	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8
CAN FD	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

DLC: Like in Classical CAN, the CAN FD DLC is 4 bits and denotes the number of data bytes in the frame. The above table shows how the two protocols use the DLC consistently up to 8 data bytes. To maintain a 4-bit DLC, CAN FD uses the remaining 7 values from 9 to 15 to denote the number of data bytes used (12, 16, 20, 24, 32, 48, 64).



SBC: The Stuff Bit Count (SBC) precedes the CRC and consists of 3 [gray-coded](#) bits and a parity bit. The following Fixed Stuff-Bit can be regarded as a second parity bit. The SBC is added to improve communication reliability.

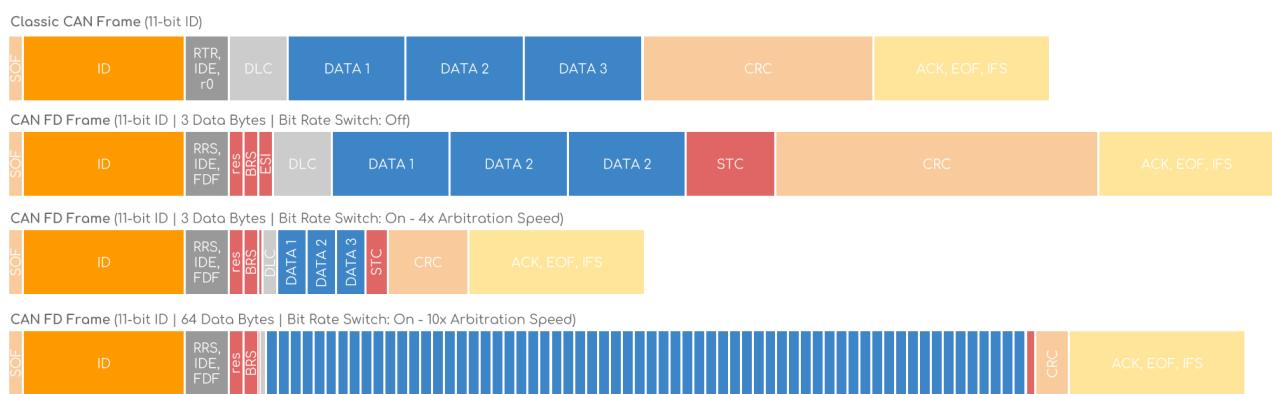
CRC: The Cyclic Redundancy Check (CRC) is 15 bit in the Classical CAN, while in CAN FD it's 17 bits (for up to 16 data bytes) or 21 bits (for 20-64 data bytes). In Classical CAN, there can be 0 to 3 stuff bits in the CRC, while in CAN FD there are always four fixed stuff bits to improve communication reliability.

ACK: The data phase (aka payload) of the CAN FD data frame stops at the ACK bit, which also marks the end of the potentially increased bit rate.

Overhead and data efficiency of CAN FD vs. CAN

As evident, the added functionality of CAN FD adds a lot of extra bits vs. Classical CAN - how can this lead to less overhead?

The answer is that it doesn't - see the below visualization of Classical CAN vs. CAN FD for 3 data bytes. In fact, the efficiency of CAN FD does not exceed Classical CAN until crossing 8 data bytes. However, by moving towards 64 data bytes, the efficiency can go from ~50% up towards ~90% (more on this below).



Need for speed: Turning on bit rate switching

As mentioned, sending 64 data bytes at regular speed would block the CAN bus, reducing the real-time performance. To solve this, bit rate switching can be enabled to allow the payload to be sent at a higher rate vs the arbitration rate (e.g. 5 Mbit/s vs 1 Mbit/s). Above we illustratively visualize the effect for the 3 data byte and 64 data byte scenarios. Note that the higher speed applies to the data frame section starting in the BRS bit and ending in the CRC delimiter. Further, most vehicles today use 0.25-0.5 Mbit/s, meaning that with 5 Mbit/s CAN FD would 10x the speed of the payload transmission.

Mixing Classical CAN and CAN FD nodes

As mentioned earlier, Classical CAN and CAN FD nodes can be mixed under certain conditions. This allows for a step-by-step migration towards CAN FD, rather than having to switch every ECU in one go. Two scenarios exist:

100% CAN FD system: Here, the CAN FD controllers can freely mix Classical CAN and CAN FD data frames.

Some nodes are legacy Classical CAN: Here, the CAN FD controllers can switch to Classical CAN communication to avoid error frame reactions from the Classical CAN nodes. Also, during e.g. ECU flashing, the Classical CAN nodes may be turned off to allow a temporary shift to CAN FD communication.

What is the max bit rate of CAN FD?

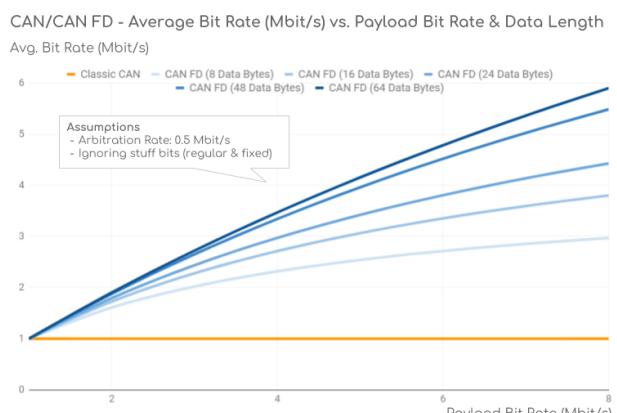
A confusing aspect of CAN FD is the max bit rate during the payload phase - as different articles mention different levels. Some state that practical applications enable up to 8 Mbit/s and theoretically 15 Mbit/s. Others state up to [12 Mbit/s](#). Further, Daimler state that [beyond 5 Mbit/s is doubtful](#) - both as there's no standard for this and because low-cost automotive Ethernet (10 BASE-T1) is expected to limit the demand for CAN FD at higher speed. So what is correct?

It depends. Looking at ISO 11898-2 (the transceiver chip standard), it specifies two symmetry parameter sets. It is recommended to use those with improved symmetry parameters, often advertised as 5 Mbit/s transceivers. The achievable data phase bit-rate depends on many things. One of the most important is the desired temperature range. Flashing of an ECU does not require the support of low temperatures. This implies that for ECU flashing, it is possible to go up to 12 Mbit/s. Another important bit-rate limitation is caused by the chosen topology. Bus-line topology with very short stubs allows significantly higher bit-rates versus hybrid topologies with long stubs or even stars. Most multi-drop bus-line networks are limited to 2 Mbit/s for a temperature range of -40 degC to +125 degC. CiA provides appropriate rules of thumb in the CiA 601-3 network design recommendation. This includes recommendations to set the sample-points in the data-phase.

CAN FD calculator: Efficiency & baud rate

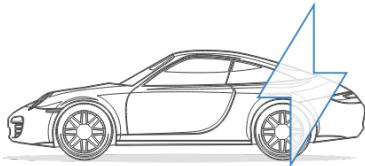
For a detailed understanding of CAN FD efficiency and bit rates, see our [CAN FD calculator](#).

The calculator compares Classical CAN vs. CAN FD in regards to frames and visualizes the frame structures in response to the settings you provide.



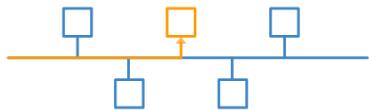
Examples: CAN FD applications

In short, CAN FD enables more data at faster rates. This is key for multiple increasingly relevant use cases:



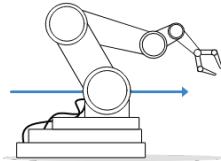
Electric vehicles

EVs and hybrids use new powertrain concepts that require far higher bit rates. Added complexity comes from new control units related to the DC/DC inverter, battery, charger, range extender etc. By 2025 it is expected that the required bit rate exceeds CAN - and with the [explosive rise in EVs](#), this may be the spearhead of the CAN FD roll-out.



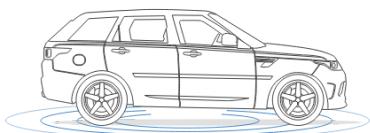
ECU flashing

Vehicle software is becoming increasingly complex. As such, performing ECU updates via e.g. the OBD2 port can take hours today. With CAN FD, such processes can [be made >4x faster](#). This use case has been one of the original drivers behind the demand for CAN FD by automotive OEMs.



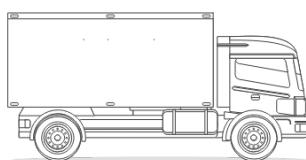
Robotics

Several applications rely on time-synced behaviour - e.g. robot arms with multiple axles. Such devices often use [CANopen](#) and require multiple CAN frames (PDOs) to be sent by each controller time-synced (without interruption from higher-priority frames). By shifting to CAN FD, the data can be sent in a single frame for efficiency.



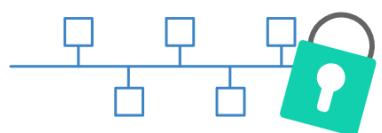
ADAS & safe driving

Increasingly, advanced driver assistance systems (ADAS) are being introduced in passenger cars and commercial vehicles. This pressures the bus load of Classical CAN, yet ADAS is key to improving safety. Here, CAN FD will be key to enhancing safe driving in the near future.



Trucks & buses

Trucks & buses utilize long CAN buses (10-20 meters). As a result, they rely on slow bit rates (250 kbit/s or 500 kbit/s as per [J1939-14](#)). Here, the upcoming J1939 FD protocol is expected to enable a significant improvement in commercial vehicle features, incl. e.g. ADAS.

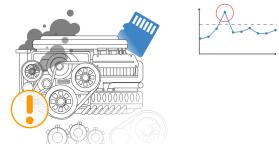
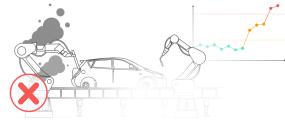


Secure CAN bus

As shown in [recent CAN hacker attacks](#), Classical CAN is vulnerable. If hackers gain access to the CAN bus (e.g. [over-the-air](#)), they could e.g. [turn off critical functions](#). CAN FD authentication via the Secure Onboard Communication (SecOC) module may be a key roll-out driver.

Logging CAN FD data - use case examples

With the rise of CAN FD there will be several use cases for logging CAN FD data:



Logging data from cars

As CAN FD gets rolled out in new cars, [CAN FD data loggers](#) will be key for OEM R&D and diagnostics

[learn more](#)

Heavy duty fleet telematics

[IoT CAN FD loggers](#) compatible with J1939 FD (flexible data-rate) will be key to future heavy duty telematics

[learn more](#)

Predictive maintenance

As CANopen FD rolls out, new industrial machinery will need CAN FD IoT loggers to help predict and avoid breakdowns

[learn more](#)

Vehicle/machine blackbox

A [CAN FD logger](#) can serve as a 'blackbox' for e.g. new prototype vehicles, providing data for diagnostics and R&D

[learn more](#)

CAN FD logging - practical considerations

When logging CAN FD data the following considerations are relevant:

#1 ISO CAN FD vs. non-ISO CAN FD

Before the ISO 11898-1:2015 update, the CAN FD standard had a [weakness related to error checking](#). Controllers adhering to the updated standard are sometimes referred to as "ISO CAN FD". When recording data from an early prototype CAN FD system you may therefore need to turn on "NON-ISO CAN FD" mode if your device supports this.

#2 Bit rate switch - on/off?

By default, your CAN FD data logger will be able to handle both Classical CAN and CAN FD data messages - without having to be pre-configured between them. Similarly, you won't have to pre-specify whether bit rate switching is on/off for pure logging purposes. However, when you transmit data to the CAN bus, you'll need to specify whether to use bit rate switching or not. If enabled, your data payloads are transmitted at the system's 2nd bit rate, which will typically be 2 or 4 Mbit/s.

#3 Conversion & CAN FD DBC files

CAN FD minimizes the need for handling multi-packet messages. This can greatly simplify the software development for converting raw CAN FD data to human-readable form, to the benefit of end users of CAN FD analyzers. Further, the standard CAN database format, [DBC](#), also supports CAN FD conversion rules. As such, it's always recommended that you collect your scaling rules in a DBC file to enable easy transition between various CAN software like e.g. [asammfd](#) etc.

CAN FD - outlook

CAN FD is predicted to take over Classical CAN in the coming years:

- The first CAN FD capable cars are expected [to be sold in 2019/20](#)
- Initial roll-out will likely use 2 Mbit/s, gradually transitioning to 5 Mbit/s
- [CANopen FD](#) has been adapted via CiA 1301 1.0
- [J1939-22](#) uses CAN FD data frames

- CAN is still a growing technology, but recently [mainly due to CAN FD](#)
- It's anticipated that in the future, [CAN FD will be used in most new development](#)

CAN FD vs. other protocols

Of course, legacy applications with no bandwidth and payload requirements will still use Classical CAN. Further, the CAN community is already developing the next generation of a CAN data link layer supporting payloads up to 2048 bytes. This approach can be regarded as an alternative to 10 Mbit/s Ethernet. As such, it is still to be determined exactly what role CAN FD will play in the future - but it will definitely be on the rise.



LIN Bus Explained - A Simple Intro

In this guide we introduce the Local Interconnect Network (LIN) protocol basics incl. LIN vs. CAN, use cases, how LIN works and the six LIN frame types.

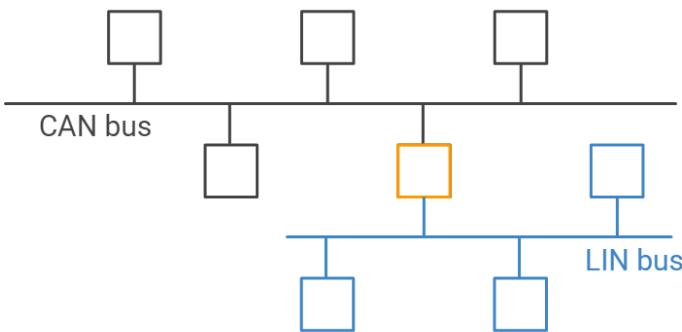
What is LIN bus?

LIN bus is a supplement to [CAN bus](#). It offers lower performance and reliability - but also drastically lower costs. Below we provide a quick overview of LIN bus and a comparison of LIN bus vs. CAN bus.

- Low cost option (if speed/fault tolerance are not critical)
- Often used in vehicles for windows, wipers, air condition etc..
- LIN clusters consist of 1 master and up to [16 slave nodes](#)
- Single wire (+ground) with 1-20 kbit/s at max 40 m bus length
- Time triggered scheduling with guaranteed latency time
- Variable data length (2, 4, 8 bytes)
- LIN supports error detection, checksums & configuration
- Operating voltage of [12V](#)
- Physical layer based on ISO 9141 (K-line)
- Sleep mode & wakeup support
- Most newer vehicles have [10+ LIN nodes](#)

LIN bus vs CAN bus

- LIN is lower cost (less harness, no license fee, cheap nodes)
- CAN uses twisted shielded dual wires 5V vs LIN single wire 12V
- A LIN master typically serves as gateway to the CAN bus
- LIN is deterministic, not event driven (i.e. no bus [arbitration](#))
- LIN clusters have a single master - CAN can have multiple
- CAN uses 11 or 29 bit identifiers vs 6 bit identifiers in LIN
- CAN offers up to 1 Mbit/s vs. LIN at max 20 kbit/s



LIN bus history

Below we briefly recap the history of the LIN protocol:

- 1999: LIN 1.0 released by the [LIN Consortium](#) (BMW, VW, Audi, Volvo, Mercedes-Benz, Volcano & Motorola)
- 2000: The LIN protocol was updated (LIN 1.1, LIN 1.2)
- 2002: LIN 1.3 released, mainly changing the physical layer
- 2003: LIN 2.0 released, adding major changes (widely used)
- 2006: [LIN 2.1](#) specification released
- 2010: [LIN 2.2A](#) released, now [widely implemented](#) versions
- 2010-12: [SAE](#) standardized LIN as SAE J2602, based on LIN 2.0
- 2016: CAN in Automation standardized LIN (ISO 17987:2016)

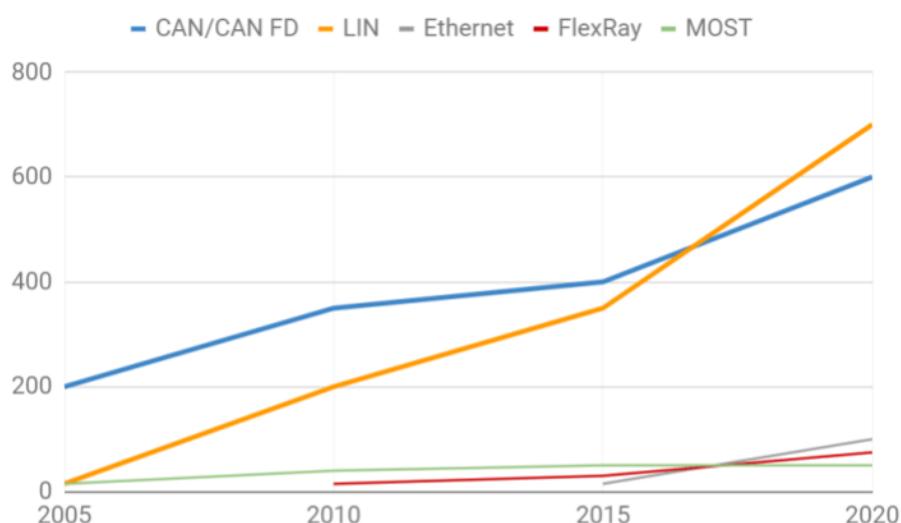


LIN bus future

The LIN protocol serves an increasingly important role in providing low cost feature expansion in modern vehicles. As such, LIN bus has exploded in popularity in the last decade with >700 million nodes expected in automotives by 2020 vs ~200 million in 2010.

#Nodes in Automotives by Technology (2005-2020)

Source: Strategic Analytics, 2013



Cybersecurity & new protocols

However, with the rise of LIN also comes increased scrutiny in regards to cyber security. [LIN faces similar risk exposures](#) as CAN - and since LIN plays a role in e.g. the seats and steering wheel, a resolution to these risks may be necessary. The future automotive vehicle networks are seeing a rise in CAN FD, FlexRay and automotive Ethernet. While there's uncertainty regarding the role each of these systems will play in future automotives, it's expected that LIN bus clusters will remain vital as the low cost solution for an ever increasing demand for features in modern vehicles.

Master/slave vs. commander/responder

As part of designing a more inclusive wording for LIN bus, the CiA/ISO/SAE agreed wording will transition to commander/responder. As such, this will be the de facto standard wording used in most guidelines and LIN bus specifications going forward.

LIN bus applications

Today, LIN bus is a de facto standard in practically all modern vehicles - with examples of automotive use cases below:

- Steering wheel: Cruise control, wiper, climate control, radio
- Comfort: Sensors for temperature, sun roof, light, humidity
- Powertrain: Sensors for position, speed, pressure
- Engine: Small motors, cooling fan motors
- Air condition: Motors, control panel (AC is often complex)
- Door: Side mirrors, windows, seat control, locks
- Seats: Position motors, pressure sensors
- Other: Window wipers, rain sensors, headlights, airflow

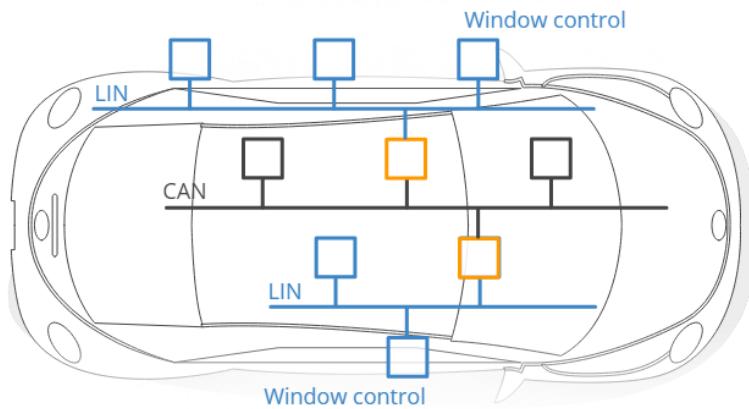
Further, LIN bus is also being used in [other industries](#):

- Home appliances: Washing machines, refrigerators, stoves
- Automation: Manufacturing equipment, metal working

Example: LIN vs CAN window control

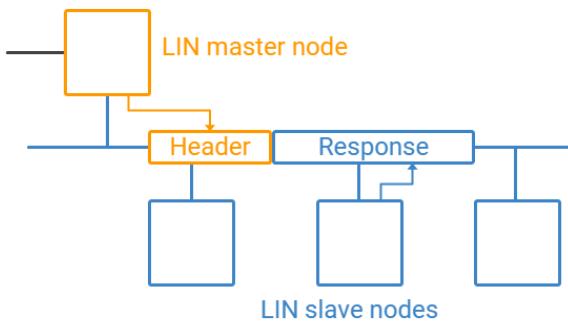
LIN nodes are typically bundled in clusters, each with a master that interfaces with the backbone CAN bus.

Example: In a car's right seat you can roll down the left seat window. To do so, you press a button to send a message via one LIN cluster to another LIN cluster via the CAN bus. This triggers the second LIN cluster to roll down the left seat window.



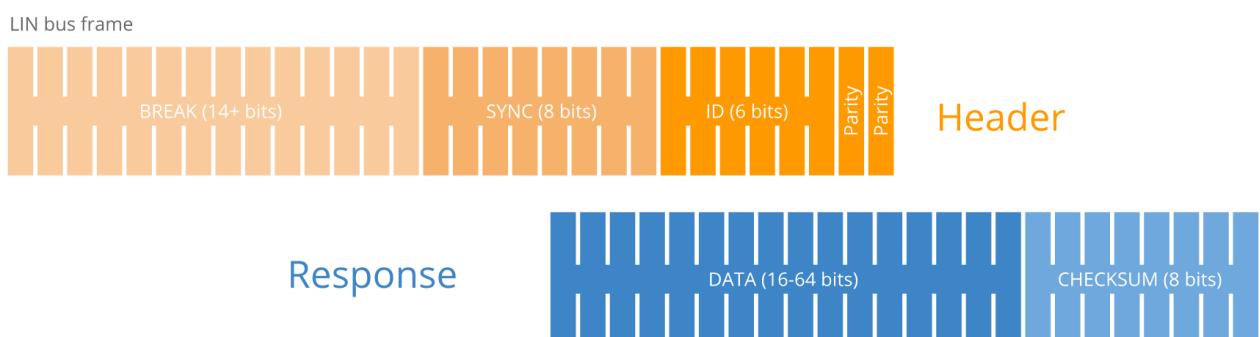
How does LIN bus work?

LIN communication at its core is relatively simple: A master node loops through each of the slave nodes, sending a request for information - and each slave responds with data when polled. The data bytes contain LIN bus signals (in raw form). However, with each specification update, new features have been added to the LIN specification - making it more complex. Below we cover the basics: The LIN frame & six frame types.



The LIN frame format

In simple terms, the LIN bus message frame consists of a header and a response. Typically, the LIN master transmits a header to the LIN bus. This triggers a slave, which sends up to 8 data bytes in response. This overall LIN frame format can be illustrated as below:



LIN frame fields

Break: The Sync Break Field (SBF) aka Break is minimum 13 + 1 bits long (and in practice most often 18 + 2 bits). The Break field acts as a "start of frame" notice to all LIN nodes on the bus.

Sync: The 8 bit Sync field has a predefined value of 0x55 (in binary, 01010101). This structure allows the LIN nodes to determine the time between rising/falling edges and thus the baud rate used by the master node. This lets each of them stay in sync.

Identifier: The Identifier is 6 bits, followed by 2 parity bits. The ID acts as an identifier for each LIN message sent and which nodes react to the header. Slaves determine the validity of the ID field (based on the parity bits) and act via below:

1. Ignore the subsequent data transmission
2. Listen to the data transmitted from another node
3. Publish data in response to the header

Typically, one slave is polled for information at a time - meaning zero collision risk (and hence no need for arbitration). Note that the 6 bits allow for 64 IDs, of which ID 60-61 are used for diagnostics (more below) and 62-63 are reserved.

Data: When a LIN slave is polled by the master, it can respond by transmitting 2, 4 or 8 bytes of data. The data length can be customized, but it is typically linked to the ID range (ID 0-31: 2 bytes, 32-47: 4 bytes, 48-63: 8 bytes). The data bytes contain the actual information being communicated in the form of LIN signals. The LIN signals are packed within the data bytes and may be e.g. just 1 bit long or multiple bytes.

Checksum: As in CAN, a checksum field ensures the validity of the LIN frame. The classic 8 bit checksum is based on summing the data bytes only (LIN 1.3), while the enhanced checksum algorithm also includes the identifier field (LIN 2.0).

Logging LIN bus data

The [CANedge](#) lets you easily log LIN bus data to an 8-32 GB SD card. Simply connect it to your LIN application to start logging - and process the data via [free software/APIs](#).

For example, the free [asammdf GUI/API](#) lets you DBC decode your LIN data to physical values and e.g. plot your LIN signals. [Learn more](#)



Six LIN frame types

Multiple types of LIN frames exist, though in practice the vast majority of communication is done via "unconditional frames".

Note also that each of the below follow the same basic LIN frame structure - and only differ by timing or content of the data bytes. Below we briefly outline each LIN frame type:

ID (dec)	ID (hex)	LIN Frame Type
0-59	00-3B	Unconditional
0-59	00-3B	Event Triggered
0-59	00-3B	Sporadic
60-61	3C-3D	Diagnostic
62	3E	User Defined
63	3F	Reserved

					
Unconditional Frames The default form of communication where the master sends a header, requesting information from a specific slave. The relevant slave reacts accordingly	Event Trigger Frames The master polls multiple slaves. A slave responds if its data has been updated, with its protected ID in the 1st data byte. If multiple respond, a collision occurs and the master defaults to unconditional frames	Sporadic Frames Only sent by the master if it knows a specific slave has updated data. The master "acts as a slave" and provides the response to its own header - letting it provide slave nodes with "dynamic" info	Diagnostic Frames Since LIN 2.0, IDs 60-61 are used for reading diagnostics from master or slaves. Frames always contain 8 data bytes. ID 60 is used for the master request, 61 for the slave response	User Defined Frames ID 62 is a user-defined frame which may contain any type of information	Reserved Frames Reserved frames have ID 63 and must not be used in LIN 2.0 conforming LIN networks

Advanced LIN topics

Below we include two advanced topics - click to expand.

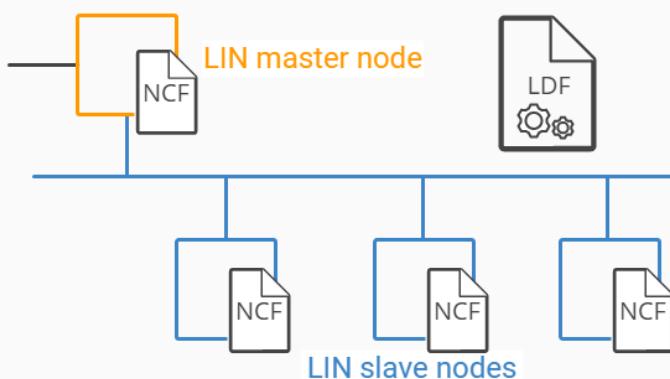
The LIN Node Configuration File (NCF) and LIN Description File (LDF)

To quickly set up LIN bus networks, off-the-shelf LIN nodes come with Node Configuration Files (NCF). The NCF details the LIN node capabilities and is a key part of the LIN topology.

An OEM will then combine these node NCFs into a cluster file, referred to as a LIN Description File (LDF). The master then sets up and manages the LIN cluster based on this LDF - e.g. the time schedule for headers.

Note that the LIN bus nodes can be re-configured by using the diagnostic frames described earlier. This type of configuration could be done during production - or e.g. everytime the network is started up. For example, this can be used to change node message IDs.

If you're familiar with [CANopen](#), you may see parallels to the [Device Configuration File](#) used to pre-configure CANopen nodes - and the role of Service Data Objects in updating these configurations.

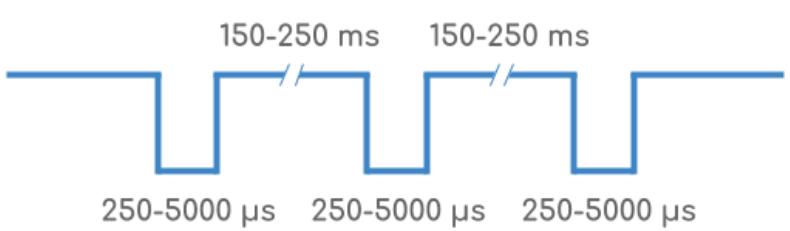


LIN Sleep & Wakeup

A key aspect of LIN is not only saving costs, but also power.

To achieve this, each LIN slave can be forced into sleep mode by the master sending a diagnostic request (ID 60) with the first byte equal to 0. Each slave also automatically sleeps after 4 seconds of bus inactivity.

The slaves can be woken up by either the master or slave nodes sending a wake up request. This is done by forcing the bus to be dominant for 250-5000 microseconds, followed by a pause for 150-250 ms. This is repeated up to 3 times if no header is sent by the master. After this, a pause of 1.5 seconds is required before sending a 4th wake up request. Typically nodes wake up after 1-2 pulses.

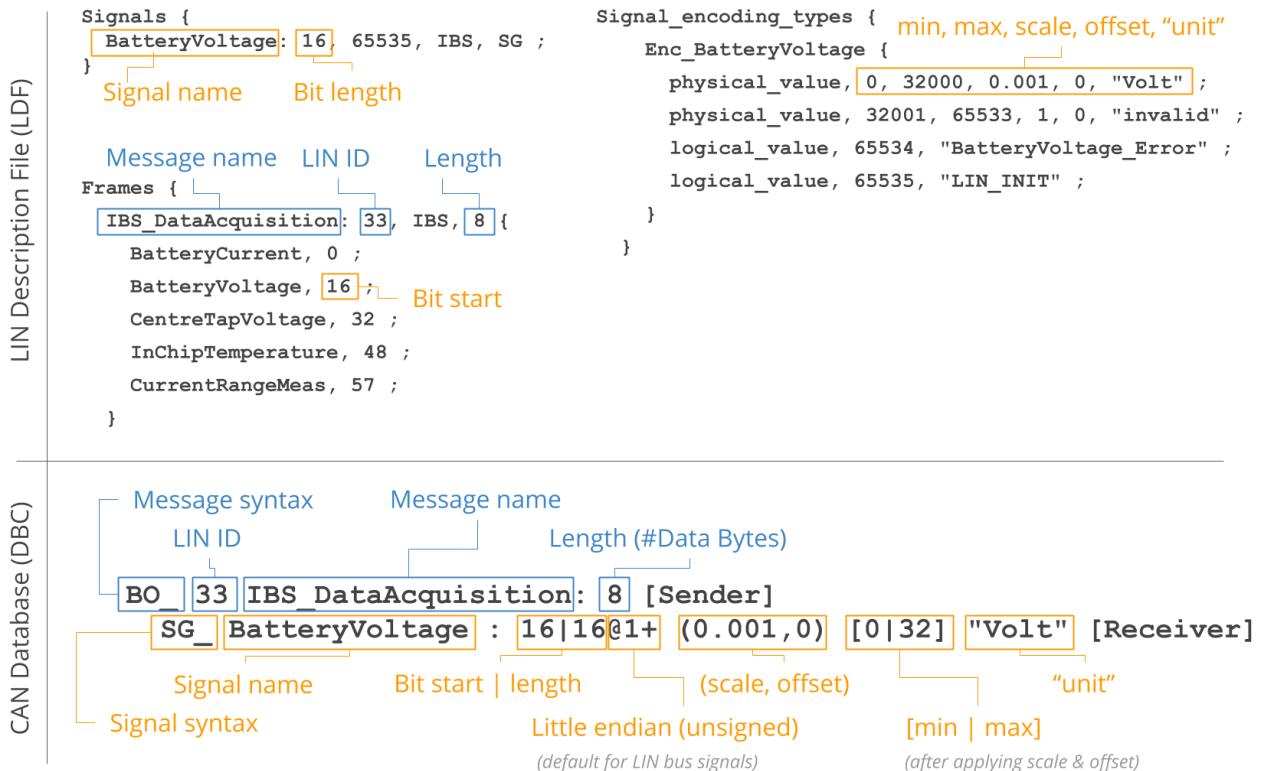


LIN Description File (LDF) vs. DBC files

As part of your LIN data logger workflow, you may need to decode your raw LIN bus data to physical values. Specifically, this involves extracting LIN signals from the LIN frame payload and decoding these to human-readable form.

This process of LIN bus decoding is similar to CAN bus decoding and requires the same information:

- ID: Which LIN frame ID contains the LIN bus signal
- Name: The LIN signal name should be known
- Start bit: Start position of the LIN signal in the payload
- Length: Length of the LIN bus signal
- Endianness: LIN signals are little endian (Intel byte order)
- Scale: How to multiply the decimal value of the LIN signal bits
- Offset: By what constant should the LIN signal value be offset
- Unit/Min/Max: Additional supporting information (optional)



This information is typically available as part of the LIN Description File (LDF) for a local interconnect network. However, since many software tools do not natively support the LDF format, we explain below how to use DBC files as an alternative.

LIN Description File (LDF) vs. DBC files

As evident from our [CAN bus intro](#) and [DBC file intro](#), the above entries are equivalent to the information stored in a CAN DBC file. This means that a simple method for storing LIN bus decoding rules is to use the DBC file format, which is supported by many software and API tools (incl. the CANedge [software tools](#) like asammcf). For example, you can load a LIN DBC file and your raw LIN bus data from the [CANedge](#) in [asammcf](#) to extract LIN bus signals from the data, which you can then plot, analyze or export.

In many cases, you may not have a LIN DBC file directly available, but instead you may have a LIN description file (LDF). Below we therefore focus on how you can convert the relevant LIN signal information into the DBC format.

Note: The LDF typically contains various other information relevant to the operation of the LIN bus, which we do not focus on here. For a full deep-dive on the LIN protocol and the a detailed description of the LDF specification, see the [LIN protocol PDF standard](#).

How to convert a LIN Description File (LDF) to DBC [incl. Examples]

Below we provide an example to showcase how you can extract LIN signal information from an LDF and enter it into a DBC file. We use a very simplified LIN description file (with only one signal and excluding some sections).

You can expand the below examples to see the LIN signal, BatteryVoltage, in the LDF format and in the DBC format.

You can also download a raw [LIN bus log file \(M4\)](#) from the [CANedge2](#) with data for this signal, which you can open and DBC decode in asammcf.

Example: BatteryVoltage signal (LDF)

[download .lrf](#) | [download .dbc](#)

Guide to LDF to DBC conversion

In short, to convert an LDF file to DBC, you'll go through the following steps for each LIN signal:

- Get the LIN signal name and length from the Signals section
- Get the LIN signal message name, ID and length from the Frames section
- Get the LIN signal bit start from the Frames section
- Go to the LDF Signal_encoding_types section and find "Enc_[signal_name]"
- Get remaining info via the syntax: 'physical_value, [min], [max], [scale], [offset], "[unit]" ;'

If you're looking to create your own LIN DBC file, we suggest you review our [DBC file introduction](#) for details on the syntax, as well as DBC editor tools.

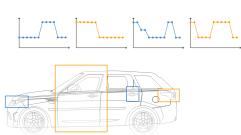
Minor pitfalls

The conversion from LDF to DBC is not entirely 1-to-1. In particular, note how the LIN signal BatteryVoltage has 2 entries for the physical value, one for the decimal range 0 to 32000 and one for 32001 to 65533. In this specific case, only the data in the first range are valid (the unit is "invalid" for the 2nd range). However, in some cases there can be multiple ranges that require separate scaling factors - something which is not possible to handle in the DBC file format. In this case, you will need to choose one of the ranges and e.g. treat results outside this range as invalid.

This is also the simplest way to handle the LIN signal 'logical_value' entries in the Signal_encoding_types section. These typically reflect how specific values of the LIN signal should be treated (e.g. as errors). One way of treating these entries would be to ignore them and possibly exclude them as part of your data post processing - similar to how FF byte values in CAN bus are often excluded as they represent invalid or N/A data.

LIN bus data logging - use case examples

LIN bus data logging is relevant across various use cases:



Vehicle CAN/LIN development

Logging CAN/LIN data via a hybrid logger is key to OEM vehicle development and can be used in optimization or diagnostics

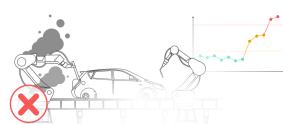
[learn more](#)



Field prototype telematics

CAN/LIN data from automotive prototype equipment can be collected at scale using IoT CAN/LIN hybrid loggers to speed up R&D

[learn more](#)



Predictive maintenance

Industrial machinery can be monitored via [IoT CAN/LIN loggers](#) in the cloud to predict and avoid breakdowns via prediction models

[learn more](#)



Rare LIN issue diagnostics

A [LIN bus logger](#) can serve as a 'blackbox' for industrial machinery, providing data for e.g. disputes or rare issue diagnostics

[learn more](#)

Practical considerations for LIN data logging

Below we list key considerations for your LIN bus data logging:

LIN logger vs. LIN interface

To record LIN bus data, you'll need a [LIN bus data logger](#) and/or interface. A LIN bus data logger with SD card has the advantage of letting you record data in standalone mode - i.e. during actual usage of the vehicle. An interface, on the other hand, is helpful during e.g. dyno testing of the vehicle functionality.

For standalone LIN loggers, it's key that the device is plug & play, compact and low cost - so as to allow it to be used in scale applications across e.g. vehicle fleets.

CAN vs. LIN support

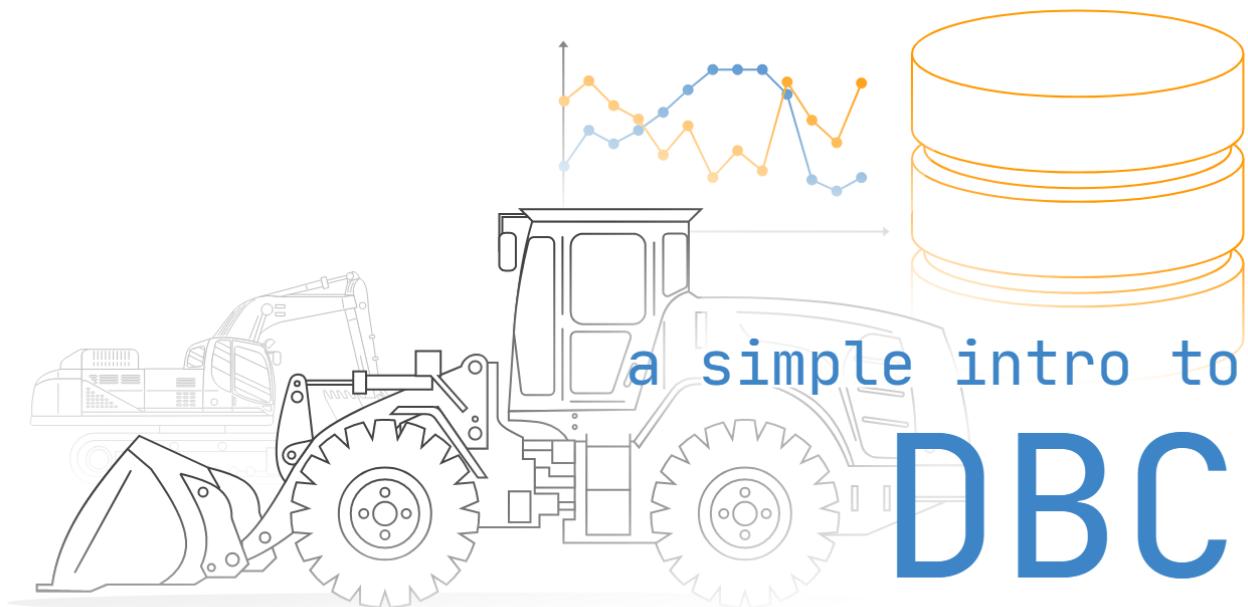
Often, you'll want to combine LIN data with CAN data to get a holistic perspective of the vehicle in use - for example:

- How is driving behavior correlated with use of various LIN bus features?
- Do issues arise in the interaction between LIN masters and the CAN bus?
- Are LIN related issues correlated with certain CAN based events?

To combine this data, you'll want a hybrid CAN/LIN logger with multiple channels. Further, [CAN FD](#) support is also key as it's expected to increasingly be rolled out in new vehicles.

WiFi

Collecting logged LIN bus data can be a hassle if you need to physically extract the data from e.g. large vehicle test fleets. Here, a WiFi enabled CAN/LIN logger can be a powerful solution. You simply specify a WiFi hotspot that the vehicle will get in range of from time-to-time - and the data will then be uploaded automatically from the SD card when in range. It's also possible to add a cellular hotspot within the vehicle for near real-time data transfer.



CAN DBC File Explained - A Simple Intro

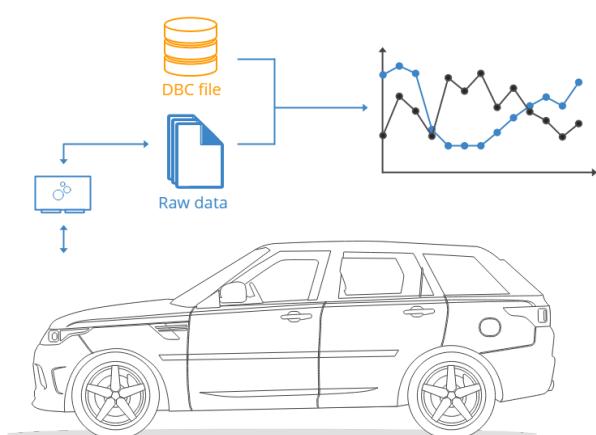
In this tutorial we explain DBC files ([CAN bus](#) databases), incl. structure, syntax, examples - and a cool [DBC editor playground](#). To get practical, we also include real J1939/OBD2 data and DBC files - which you can load in free open source CAN tools.

What is a CAN DBC file?

A CAN DBC file (CAN database) is a text file that contains information for decoding raw CAN bus data to 'physical values'.

To understand what 'raw CAN data' looks like, see the below example CAN frame from a truck:

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



If you have a CAN DBC that contains decoding rules for the CAN ID, you can 'extract' parameters (signals) from the data bytes. One such signal could be EngineSpeed:

Message	Signal	Value	Unit
EEC1	EngineSpeed	621	rpm

To understand how DBC decoding works, we will explain the DBC syntax and provide step-by-step decoding examples.

DBC message & signal syntax

Let us start with a real CAN DBC file example. Below is a demo [J1939 DBC file](#) that contains decoding rules for speed (km/h) and engine speed (rpm). You can copy this into a text file, rename it as e.g. j1939dbc and use it to extract speed/RPM from trucks, tractors or other heavy-duty vehicles.

J1939 DBC demo example

```
VERSION ""

NS_ :
  CM_
  BA_DEF_
  BA_
  BA_DEF_DEF_

BS_:

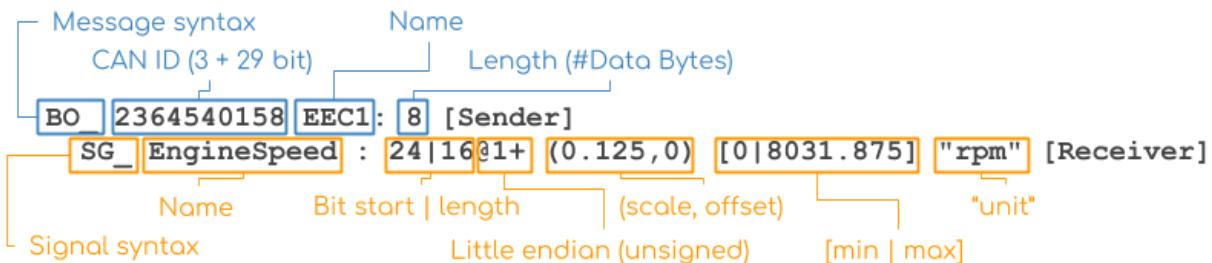
BU_:

B0_ 2364540158 EEC1: 8 Vector_XXX
SG_ EngineSpeed : 24|16@1+ (0.125,0) [0|8031.875] "rpm" Vector_XXX

B0_ 2566844926 CCVS1: 8 Vector_XXX
SG_ WheelBasedVehicleSpeed : 8|16@1+ (0.00390625,0) [0|250.996] "km/h" Vector_XXX

CM_ B0_ 2364540158 "Electronic Engine Controller 1";
CM_ SG_ 2364540158 EngineSpeed "Actual engine speed which is calculated over a minimum crankshaft angle
of 720 degrees divided by the number of cylinders....";
CM_ B0_ 2566844926 "Cruise Control/Vehicle Speed 1";
CM_ SG_ 2566844926 WheelBasedVehicleSpeed "Wheel-Based Vehicle Speed: Speed of the vehicle as calculated
from wheel or tailshaft speed.";
BA_DEF_ SG_ "SPN" INT 0 524287;
BA_DEF_ B0_ "VFrameFormat" ENUM "StandardCAN", "ExtendedCAN", "reserved", "J1939PG";
BA_DEF_ "BusType" STRING ;
BA_DEF_ "ProtocolType" STRING ;
BA_DEF_DEF_ "SPN" 0;
BA_DEF_DEF_ "VFrameFormat" "J1939PG";
BA_DEF_DEF_ "BusType" "";
BA_DEF_DEF_ "ProtocolType" "";
BA_ "ProtocolType" "J1939";
BA_ "BusType" "CAN";
BA_ "VFrameFormat" B0_ 2364540158 3;
BA_ "VFrameFormat" B0_ 2566844926 3;
BA_ "SPN" SG_ 2364540158 EngineSpeed 190;
BA_ "SPN" SG_ 2566844926 WheelBasedVehicleSpeed 84;
```

At the heart of a DBC file are the rules that describe how to decode CAN messages and signals:



DBC message syntax explained

- A message starts with BO_ and the ID must be unique and in decimal (not hexadecimal)
- The DBC ID adds 3 extra bits for 29 bit CAN IDs to serve as an 'extended ID' flag
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The length (DLC) must be an integer between 0 and 1785
- The sender is the name of the transmitting node, or Vector__XXX if no name is available

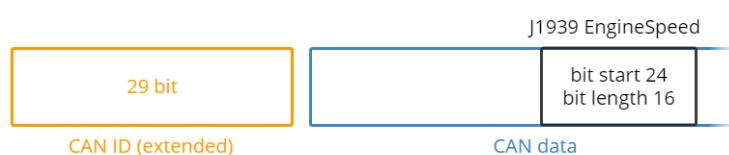
DBC signal syntax explained

- Each message contains 1+ signals that start with SG_
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The bit start counts from 0 and marks the start of the signal in the data payload
- The bit length is the signal length
- The @1 specifies that the byte order is little-endian/Intel (vs @0 for big-endian/Motorola)
- The + informs that the value type is unsigned (vs - for signed signals)
- The (scale,offset) values are used in the physical value linear equation (more below)
- The [min|max] and unit are optional meta information (they can e.g. be set to [0|0] and "")
- The receiver is the name of the receiving node (again, Vector__XXX is used as default)

Example: Extract physical value of EngineSpeed signal

To understand how DBC decoding works, we'll show how to extract the signal EngineSpeed from the CAN frame in the intro:

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



1# Match the CAN ID vs DBC ID

In practice, most raw CAN data log files contain 20-80 unique CAN IDs. As such, the first step is to map each CAN ID to the relevant conversion rules in the DBC. For regular 11-bit CAN IDs, this can simply be done by mapping the decimal value of the CAN ID to the DBC CAN IDs. For extended 29-bit CAN IDs, a mask (0x1FFFFFFF) needs to be applied to the 32-bit DBC ID to get the 29-bit CAN ID - which can then be mapped against the log file.

To easily convert between the IDs, see the 'DBC ID vs CAN ID' sheet in our [DBC editor playground](#).

Regarding J1939 PGN conversion

In this example we directly map the 29-bit CAN ID to the 'masked' DBC ID. In practice, J1939 conversion is often done by extracting the 18-bit J1939 PGN from the CAN ID and the DBC ID and then comparing the PGNs.

The method depends on your software. In the CANedge tools like [asammdf](#) and our [Python API tools](#), loading a J1939 DBC will automatically make the tools use J1939 PGN matching.

J1939 message (PGN & SPNs)



2# Extract the signal bits

Next, use the DBC bit start, length and endianness to extract the relevant bits from the CAN frame data payload. In this example, the start bit is 24 (meaning byte 3 when counting from 0) and the bit length is 16 (2 bytes):

FF FF FF 68 13 FF FF FF

Big endian vs. little-endian (advanced)

In the example, we use little-endian, meaning that the 'bit start' in the DBC file reflects the position of least-significant bit (LSB). If you add a DBC signal in a DBC file viewer (e.g. Vector CANDB++), the LSB is also used as the bit start in the DBC editor.

If you instead add a big-endian signal in a DBC editor GUI, you'll still see the LSB as the bit start - but when you save the DBC, the bit start is set to the most significant bit (MSB) in the signal. This approach makes GUI editing more intuitive, but can be a confusing if you switch between a GUI and text editor. To understand this fully, check out our [CAN DBC editor playground](#).

Bit indices							
7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16
31	30	29	28	27	26	25	24
39	38	37	36	35	34	33	32
47	46	45	44	43	42	41	40
55	54	53	52	51	50	49	48
63	62	61	60	59	58	57	56

3# Scale the extracted bits

The EngineSpeed signal is little-endian (@1) and we therefore need to reorder the byte sequence from 6813 to 1368. Next, we convert the HEX string to decimal and apply the linear conversion:

```
physical_value = offset + scale * raw_value_decimal  
621 rpm = 0 + 0.125 * 4968
```

In short, the EngineSpeed physical value (aka scaled engineering value) is 621 rpm.

4# DBC decode your full dataset

Steps 1-3 take outset in a single CAN frame. In practice, you'll decode raw data from e.g. vehicles, machines or boats with millions of CAN frames, timestamps, several unique CAN IDs and sometimes hundreds of signals.

To handle this complexity, specialized software is used to decode raw CAN data into human-readable form by loading the data log files and related DBC file(s). To illustrate what the output may look like, we've added a snippet of DBC decoded J1939 data logged with a [CANedge](#). The data has been converted via asammdf, filtered to include a set of relevant signals and exported as CSV:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFuelRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.289050102+01:00,0,1527.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.299000025+01:00,0,1528.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.308300018+01:00,0,1526.86,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.309099913+01:00,0,1526.75,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.317199945+01:00,0,1526.45,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.319149971+01:00,0,1526.38,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.320349932+01:00,0,1526.15,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.326800108+01:00,0,1524.93,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.25
...
...
```

CAN DBC editor playground

Adding or editing signals in your DBC file can be confusing. To help you better understand how this works, we've created a very basic online '[CAN DBC editor playground](#)'. You can open it and edit the yellow cells - or make a copy for personal use. Feel free to bookmark the editor and share it!

J1939/OBD2 data & DBC samples

The best way to learn more about DBC conversion of raw CAN bus data is to try it out. Below you can download raw J1939/OBD2 data from the [CANedge2](#). The zip also includes an OBD2 DBC and a demo J1939 DBC.

Via the link, you can also download the free [asammdf GUI](#) to open the data & DBC files. Try plotting EngineSpeed as an example. [Download the sample data & dbc files](#).

Advanced: Meta info, attributes & multiplexing

In this section we briefly outline some of the more advanced topics of CAN DBC files, incl. meta info, attributes, value tables and multiplexing. If you're new to DBC files you can optionally skip this section.

Comments (message/signal names, ...)

The DBC file can contain various extra information and below we outline the most common types. This information is stored in the DBC file after all the messages. The comment attribute lets you map a 1-255 character comment to a message ID or signal name, e.g. to provide more information. Example for EEC1 message:

```
CM_ BO_ 2364540158 "Electronic Engine Controller 1"
```

Example for EngineSpeed signal:

```
CM_ SG_ 2364540158 EngineSpeed "Actual engine speed which is calculated over  
a minimum crankshaft angle of 720 degrees divided by the number of cylinders....";
```

Attributes: Frame format, SPN IDs, ...

Custom attributes can be added to messages and signals in a similar way as the comment syntax. A typical attribute is the VFrameFormat, which can be used to describe the message frame type. It is initialized as below:

```
BA_DEF_BO_ "VFrameFormat" ENUM "StandardCAN","ExtendedCAN","reserved","J1939PG";
```

Once initialized, a message can be mapped as follows (indexing from 0):

```
BA_ "VFrameFormat" BO_ 2364540158 3;
```

In this case, we inform that the message EEC1 is of the J1939 PGN type, which may result in specific display or handling in various DBC editor GUI tools, as well as data processing tools.

Similarly, you can add J1939 SPN IDs as an attribute as below:

```
BA_DEF_SG_ "SPN" INT 0 524287;  
BA_ "SPN" SG_ 2364540158 EngineSpeed 190;
```

Here, EngineSpeed is assigned the SPN 190. You might find it more natural to do this in the opposite way - i.e. use the SPN IDs in the message/signal section, then map the SPN names via attributes. While you can definitely do this, it is not the most common convention. Further, the first character in a CAN DBC signal name cannot be a number - so you'd need to write e.g. _190 or SPN_190.

Value tables

Some CAN DBC signals are 'state variables', such as gear-shift, diagnostic trouble codes, status codes etc. These take discrete values and will require a mapping table to enable interpretation.

The CAN DBC format enables this via value tables, which let you assign a state description to the physical decimal value of each state of a signal. The states should be in descending order.

Example:

```
VAL_ 2297441534 MaterialDropActiveStatus 3 "NotAvailable" 2 "Error" 1 "On" 0 "Off" ;
```

Multiplexed signals (OBD2 DBC example)

Multiplexing is sometimes used in CAN bus communication, with perhaps the most known example being within OBD2 communication. Consider for example the below OBD2 response frames:

```
7E8 03 41 11 30 FF FF FF FF  
7E8 03 41 0D 37 FF FF FF FF
```

Here, both response frames carry 1 byte of data in byte 3, yet despite the CAN ID being identical, the interpretation of the data differs between the two frames. This is because byte 2 serves as a multiplexer, specifying which [OBD2 PID](#) the data is coming from. To handle this in a DBC file context, the below syntax can be applied:

```

B0_ 2024 OBD2: 8 Vector__XXX
SG_ S1_PID_0D_VehicleSpeed m13 : 31|8@0+ (1,0) [0|255] "km/h" Vector__XXX
SG_ S1_PID_11_ThrottlePosition m17 : 31|8@0+ (0.39216,0) [0|100] "%" Vector__XXX
SG_ S1 m1M : 23|8@0+ (1,0) [0|255] "" Vector__XXX
SG_ Service M : 11|4@0+ (1,0) [0|15] "" Vector__XXX
...
SG_MUL_VAL_ 2024 S1_PID_0D_VehicleSpeed S1 13-13;
SG_MUL_VAL_ 2024 S1_PID_11_ThrottlePosition S1 17-17;
SG_MUL_VAL_ 2024 S1 Service 1-1;

```

Here, the M in the Service signal serves as the 'multiplexor signal'. In this case, it toggles which OBD2 service mode is used (mode 01, 02, ...). The signal S1 is multiplexed by Service, which is evident from the SG_MUL_VAL_field where the two are grouped. As evident, the signal S1 has the value m1 after the signal name, which means that if the Service signal takes the value 1, the data reflects the OBD2 service mode 01.

The above is referred to as simple multiplexing. But CAN DBC files also support extended multiplexing, where a multiplexed signal (in this case S1) can also be a multiplexor and thus control the interpretation of other parts of the data payload. To see this, note that S1 takes the same role as Service in the syntax, adding an M after m1 and being grouped with the two OBD2 PIDs, speed and throttle position.

Extended multiplexing works as before: If S1 takes the value 13 (HEX 0D), it means that only signals that are A) part of the S1 group and B) have a multiplexer switch value of 13 should be taken into account. In this example, it means that byte 4 reflects data for vehicle speed. If byte 3 equals 17 (HEX 11), byte 4 reflects data for the throttle position instead.

DBC multiplexing and extended multiplexing is an advanced topic and not supported by all data processing tools. However, you can use e.g. the CANDB++ DBC editor to more easily view and understand DBC files with multiplexing, like the OBD2 DBC: [obd2 dbc file](#)

The OBD2 DBC file can be used together with our [CANedge CAN loggers](#) for the purpose of decoding OBD2 frames in e.g. [asammcf](#) or our [Python API modules](#). For more on this topic, see Vector's guide to extended multiplexing in DBC files [here](#).

DBC software tools (editing & processing)

Software that uses CAN DBC files can be split in two groups: Editing & data processing.

DBC editor tools

- **Vector CANDB++:** The free demo version of Vector's CANalyzer includes a useful version of CANDB++, the Vector DBC editor. It offers the most extensive functionality available, including quick "consistency checks" of DBC files
- **Kvaser Database Editor:** Kvaser provides a free and easy-to-use CAN DBC editor, which includes the ability to append DBC files and visualize the signal construction
- **canmatrix:** This open source Python DBC module lets you load DBC files, edit them and export them to other formats. It is used in e.g. asammcf and our Python API

CAN data processing tools

Most CAN data processing tools support DBC files - including below:

- **asammcf GUI:** The [asammcf GUI](#) lets you load raw MDF4 data and DBC convert it to human-readable form - as well as create quick plots, analyses and exports
- **Python API:** Our [Python API](#) lets you automate the DBC conversion of your data e.g. for large-scale data analysis or as part of [telematics dashboard](#) integrations
- **Other tools:** Our [MDF4 converters](#) let you quickly convert your CANedge [MF4](#) data to e.g. Vector ASC, PEAK TRC and more

How to create a DBC file?

Getting started: If you need to construct a new DBC file, we recommend using one of the DBC editors above. For beginners, we recommend the Kvaser DBC editor. When creating a new DBC file, you can typically select a DBC template (e.g. a [CAN FD DBC](#), [J1939 DBC](#), [NMEA 2000 DBC](#) etc). Next, start by adding a single message and a single signal to the DBC and save it.

To ensure your DBC looks OK, we recommend to open the DBC in a text editor. This way you can verify that the basic DBC syntax looks as you'd expect - and you can use this version as a benchmark for comparison. It's a good idea to maintain git revisioning on any changes to the DBC from here.

Test your DBC: As a second step, we recommend to test the DBC file using a CAN bus decoder software tool. For example, if you're using a [CANedge](#) CAN bus data logger to record raw CAN data from your application, you can use the free CAN decoder software, asammmdf, to load your raw data and your DBC file. This way you can quickly extract the signal you added in the DBC - and verify via a visual plot that the construction looks OK.

Expand your DBC: Next, you can add any remaining CAN messages and signals, as well as comments/descriptions, value tables etc. We recommend to do regular checks as before to ensure the construction is OK.

Check consistency: Finally, we recommend doing a full consistency check both in Vector's DBC editor, CANDB++, and Kvaser's DBC editor. In CANDB++ select 'File/Consistency Check' and keep an eye out for critical errors (though note that your DBC may be sufficiently valid for most tools, even if some issues are reported). In the Kvaser database editor, you can select each message to quickly spot signals with invalid fields (they'll be highlighted in yellow). Once you are done, we always recommend doing a visual analysis of your scaled CAN data to ensure that you do not have e.g. endianness, scale factor or offset errors.



a simple intro to

CAN errors

CAN Bus Errors Explained - A Simple Intro

Need a practical intro to CAN bus errors?

In this tutorial you'll learn about the basics of CAN error handling, the 5 CAN bus error types, the CAN error frame and CAN node error states.

To get practical, we'll also generate & record CAN errors in 6 experiments.

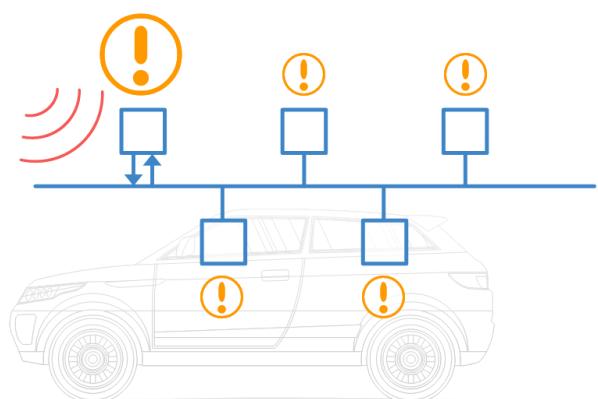
What are CAN bus errors?

As explained in our simple intro to CAN bus, the Controller Area Network is today the de facto standard across automotives and industrial automation systems.

A core benefit is the robustness of CAN, making it ideal for safety critical applications. Here, it is worth noting:

Error handling is vital to the robustness of CAN.

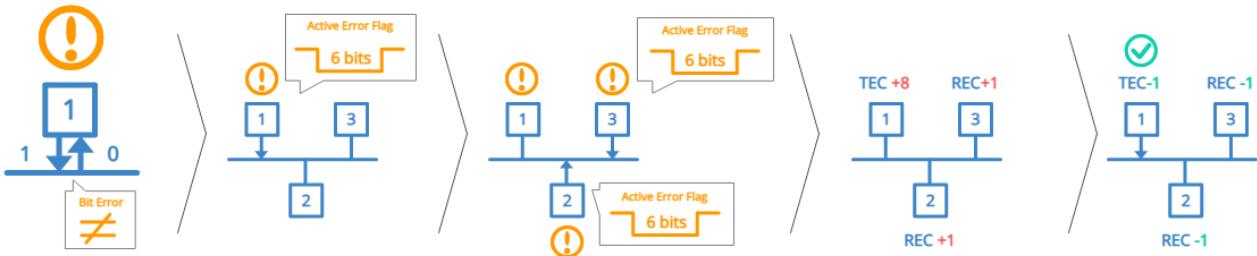
CAN bus errors can occur for several reasons - faulty cables, noise, incorrect termination, malfunctioning CAN nodes etc. Identifying, classifying and resolving such CAN errors is key to ensuring the continued performance of the overall CAN system.



In particular, error handling identifies and rejects erroneous messages, enabling a sender to re-transmit the message. Further, the process helps identify and disconnect CAN nodes that consistently transmit erroneous messages.

How does CAN error handling work?

Error handling is a built-in part of the CAN standard and every CAN controller. In other words, every CAN node handles fault identification and confinement identically. Below we've made a simple illustrative example:



Example step-by-step

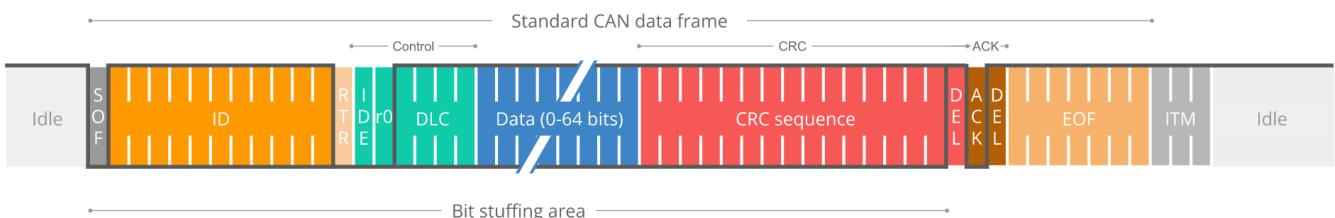
1. CAN node 1 transmits a message onto the CAN bus - and reads every bit it sends
2. In doing so, it discovers that one bit that was sent dominant was read recessive
3. This is a 'Bit Error' and node 1 raises an Active Error Flag to inform other nodes
4. In practice, this means that node 1 sends a sequence of 6 dominant bits onto the bus
5. In turn, the 6 dominant bits are seen as a 'Bit Stuffing Error' by other nodes
6. In response, nodes 2 and 3 simultaneously raise an Active Error Flag
7. This sequence of raised error flags comprise part of a 'CAN error frame'
8. CAN node 1, the transmitter, increases its 'Transmit Error Counter' (TEC) by 8
9. CAN nodes 2 and 3 increase their 'Receive Error Counter' (REC) by 1
10. CAN node 1 automatically re-transmits the message - and now succeeds
11. As a result, node 1 reduces its TEC by 1 and nodes 2 and 3 reduce their REC by 1

The example references a number of concepts that we will detail shortly: Error frames, error types, counters and states.

The CAN bus error frame

In the illustrative example, the CAN nodes 'raise Active Error Flags', thus creating an 'error frame' in response to detecting a CAN error.

To understand how this works, let us first look at a "normal" CAN frame (without errors):

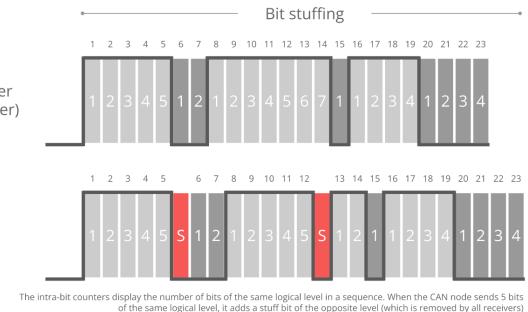


CAN bus bit stuffing

Notice that we highlighted 'bit stuffing' across the CAN frame.

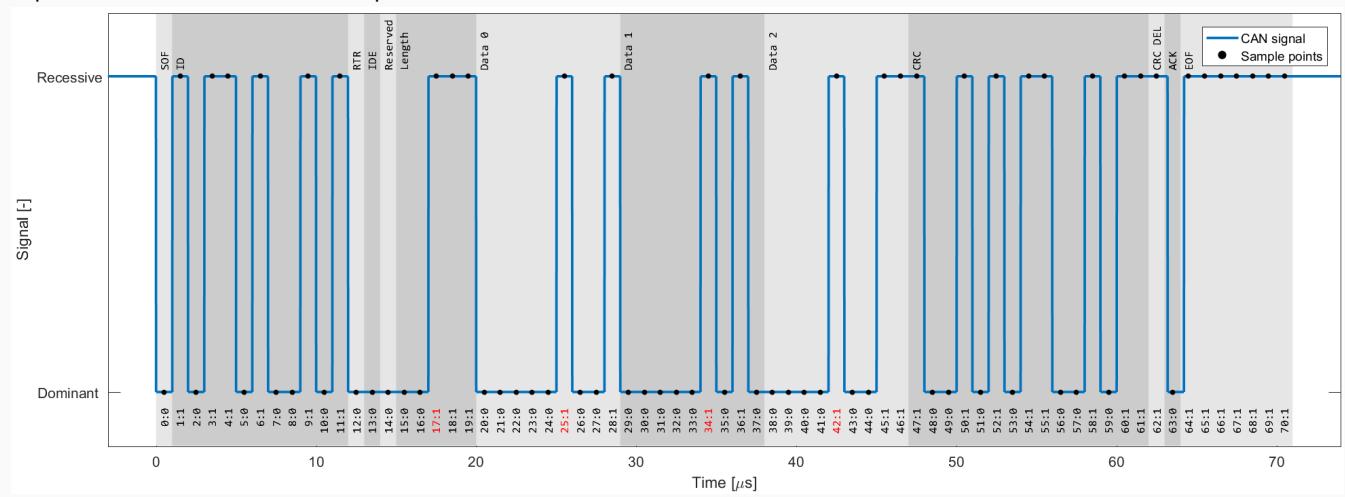
Bit stuffing is a subtle, but vital part of the CAN standard. Basically it states that whenever a CAN node sends five bits of the same logic level (dominant or recessive), it must send one bit of the opposite level. This extra bit is automatically removed by the receiving CAN nodes. This process helps ensure continuous synchronisation of the network.

As per the previous example, when CAN node 1 detects an error during the transmission of a CAN message, it immediately transmits a sequence of 6 bits of the same logic level - also referred to as raising an Active Error Flag.



Oscilloscope example

If we measure the transmission of a CAN frame via an oscilloscope and digitize the result, we can also see the stuff bits in practice (see the red timestamp marks):



Active Error Flags

As we just learned, such a sequence is a violation of the bit stuffing rule - aka a 'Bit Stuffing Error'. Further, this error is visible to all CAN nodes on the network (in contrast to the 'Bit Error' that resulted in this error flag being raised). Thus, the raising of error flags can be seen as a way of "globalizing" the discovery of an error, ensuring that every CAN node is informed. Note that the other CAN nodes will see the Active Error Flag as a Bit Stuffing Error. In response they also raise an Active Error Flag.

As we'll explain shortly, it is important to distinguish between the error flags. In particular, the first error flag (from the 'discovering' node) is often referred to as a 'primary' Active Error Flag, while the error flags of subsequent 'reacting' nodes are referred to as the 'secondary' Active Error Flag(s).

3 CAN error frame examples

Let's look at three example scenarios:

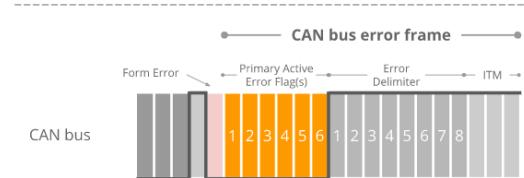
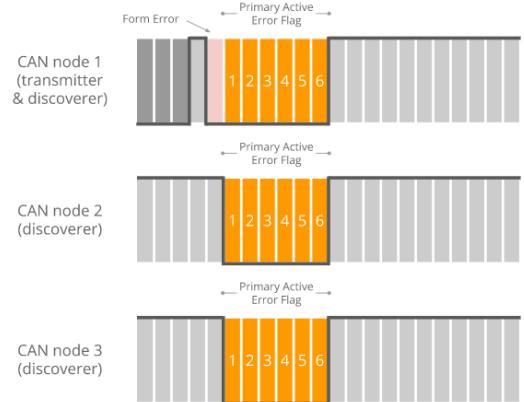
Example 1: 6 bits of error flags

Here, all CAN nodes simultaneously discover that an error exists in a CAN message and raise their error flags at the same time.

The result is that the error flags all overlap and the total sequence of dominant bits lasts for 6 bits in total. All CAN nodes will in this case consider themselves the 'discovering' CAN nodes.

This type of simultaneous discovery is less common in practice. However, it could e.g. happen as a result of Form Errors (such as a CRC delimiter being dominant instead of recessive), or if a CAN transmitter experiences a bit error during the writing of a CRC field.

Example 1: 'Active' CAN bus error frame (6 bits of Active Error Flags)

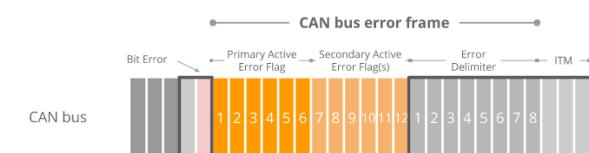


Example 2: 12 bits of error flags

Here, CAN node 1 transmits a dominant bit, but reads it as recessive - meaning that it discovers a Bit Error. It immediately transmits a sequence of 6 dominant bits.

The other nodes only discover the Bit Stuffing Error after the full 6 bits have been read, after which they simultaneously raise their error flags, resulting in a subsequent sequence of 6 dominant bits - i.e. 12 in total.

Example 2: 'Active' CAN bus error frame (12 bits of Active Error Flags)

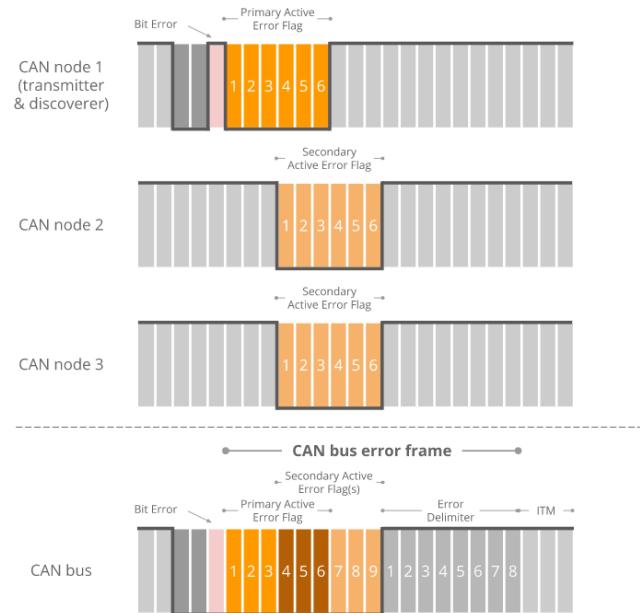


Example 3: 9 bits of error flags

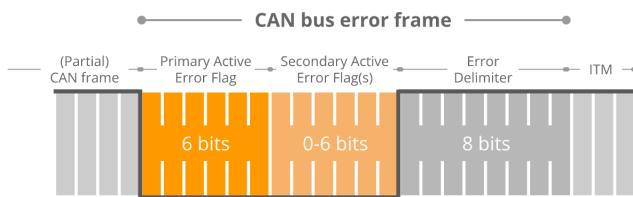
Here, CAN node 1 has already transmitted a sequence of 3 dominant bits when it discovers a Bit Error and begins sending 6 dominant bits.

Once halfway through the primary Active Error Flag, nodes 2 and 3 recognize the Bit Stuffing Error (due to the 3 initial dominant bits being followed by another 3 dominant bits) and they begin raising their error flags. The result is that the sequence of dominant bits from error flags becomes 9 bit long.

Example 3: 'Active' CAN bus error frame (9 bits of Active Error Flags)



The above logic of raising error flags is reflected in what we call an 'active' CAN error frame.



Note in particular how the secondary error flags raised by various nodes overlap each other - and how the primary and secondary flags may overlap as well. The result is that the dominant bit sequence from raised error flags may be 6 to 12 bits long.

This sequence is always terminated by a sequence of 8 recessive bits, marking the end of the error frame.

In practice, the active error frame may "begin" at different places in the erroneous CAN frame, depending on when the error is discovered. The result, however, will be the same: All nodes discard the erroneous CAN frame and the transmitting node can attempt to re-transmit the failed message.

Passive Error Flags

If a CAN node has moved from its default 'active' state to a 'passive' state (more on this shortly), it will only be able to raise so-called 'Passive Error Flags'. A Passive Error Flag is a sequence of 6 recessive bits as seen below.

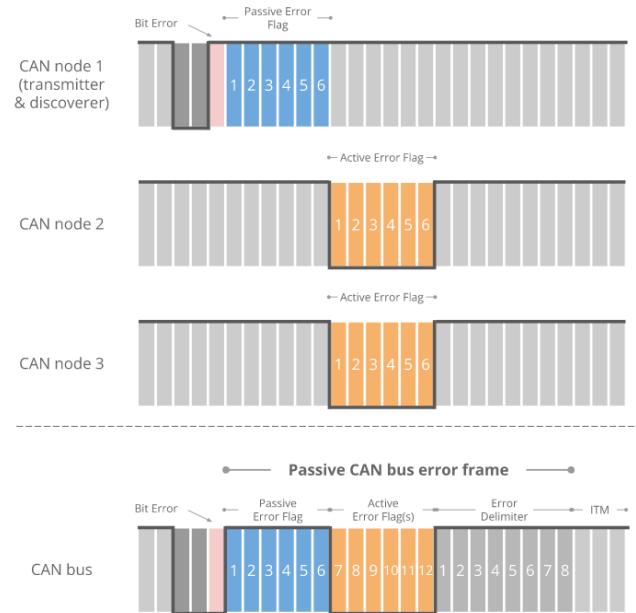
In this case it's relevant to distinguish between a Passive Error Flag raised by a transmitting node and a receiving node.

Example 4: Transmitter is Error Passive

As shown in the illustration (Example 4), if a transmitter (such as CAN node 1 in our example) raises a Passive Error Flag (e.g. in response to a Bit Error), this will correspond to a consecutive sequence of 6 recessive bits.

This is in turn detected as a Bit Stuffing Error by all CAN nodes. Assuming the other CAN nodes are still in their Error Active state, they will raise Active Error Flags of 6 dominant bits. In other words, a passive transmitter can still "communicate" that a CAN frame is erroneous.

Example 4: 'Passive' CAN bus error frame (transmitter in Error Passive mode)

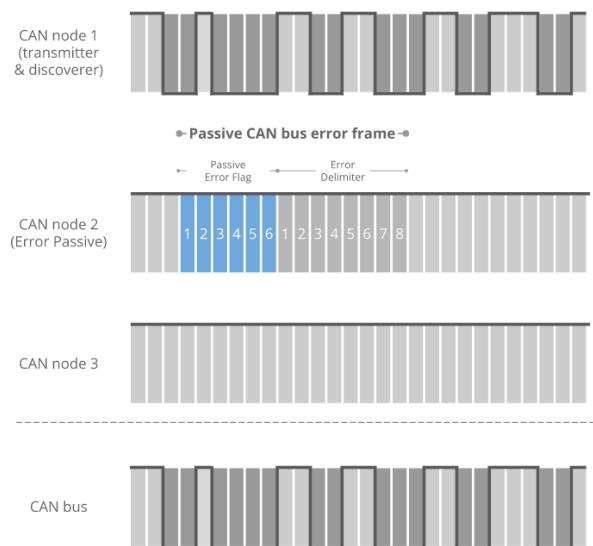


Example 5: Receiver is Error Passive

In contrast, if a receiver raises a Passive Error Flag this is in practice "invisible" to all other CAN nodes on the bus (as any dominant bits win over the sequence of recessive bits) - see also Example 5.

Effectively, this means that an Error Passive receiver no longer has the ability to destroy frames transmitted by other CAN nodes.

Example 5: 'Passive' CAN bus error frame (receiver in Error Passive mode)



CAN error types

Next, let us look at what errors may cause CAN nodes to raise error flags.

The CAN bus protocol specifies 5 CAN error types:

1. Bit Error [Transmitter]
2. Bit Stuffing Error [Receiver]
3. Form Error [Receiver]
4. ACK Error (Acknowledgement) [Transmitter]
5. CRC Error (Cyclic Redundancy Check) [Receiver]

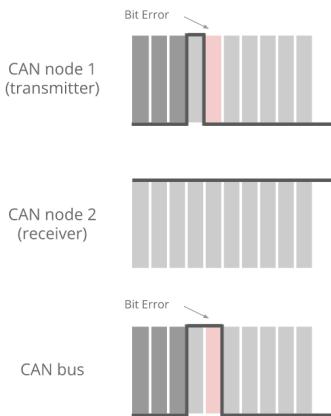
We've already looked at Bit Errors and Bit Stuffing Errors briefly, both of which are evaluated at the bit level. The remaining three CAN error types are evaluated at the message level.

Below we detail each error type:

CAN bus error types

1	Bit Error	Node transmits a dominant/recessive bit, but reads back the opposite logical level
2	Bit Stuffing Error	Node detects a sequence of 6 bits of the same logical level between the SOF and CRC
3	Form Error	Node detects a bit of an invalid logical level in the SOF/EOF fields or ACK/CRC delimiters
4	ACK Error	Node transmits a CAN message, but the ACK slot is not made dominant by receiver(s)
5	CRC Error	Node calculates a CAN message CRC that differs from the transmitted CRC field value

Example: CAN Bus Bit Error

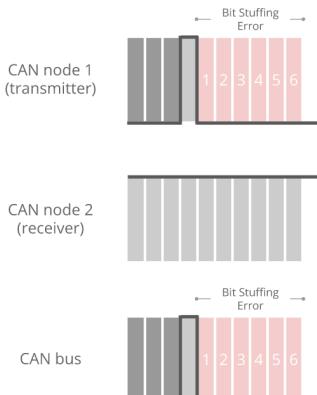


#1 Bit Error

Every CAN node on the CAN bus will monitor the signal level at any given time - which means that a transmitting CAN node also "reads back" every bit it transmits. If the transmitter reads a different data bit level vs. what it transmitted, the transmitter detects this as a Bit Error.

If a bit mismatch occurs during the arbitration process (i.e. when sending the CAN ID), it is not interpreted as a Bit Error. Similarly, a mismatch in the acknowledgement slot (ACK field) does not cause a Bit Error as the ACK field specifically requires a recessive bit from the transmitter to be overwritten by a dominant bit from a receiver.

Example: CAN Bus Bit Stuffing Error

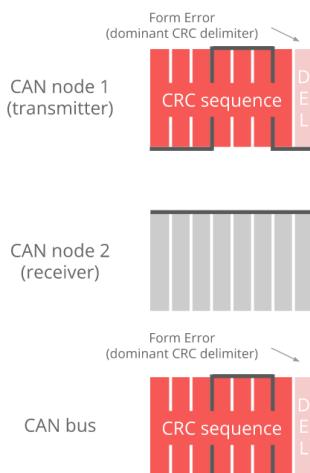


#2 Bit Stuffing Error

As explained, bit stuffing is part of the CAN standard. It dictates that after every 5 consecutive bits of the same logical level, the 6th bit must be a complement. This is required to ensure the on-going synchronization of the network by providing rising edges. Further, it ensures that a stream of bits are not mis-interpreted as an error frame or as the interframe space (7 bit recessive sequence) that marks the end of a message. All CAN nodes automatically remove the extra bits.

If a sequence of 6 bits of the same logical level is observed on the bus within a CAN message (between the SOF and CRC field), the receiver detects this as a Bit Stuffing Error aka Stuff Error.

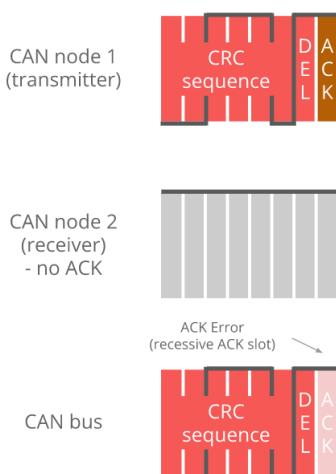
Example: CAN Bus Form Error



#3 Form Error

This message-level check utilises the fact that certain fields/bits in the CAN message must always be of a certain logical level. Specifically the 1-bit SOF must be dominant, while the entire 8-bit EOF field must be recessive. Further, the ACK and CRC delimiters must be recessive. If a receiver finds that any of these are bits are of an invalid logical level, the receiver detects this as a Form Error.

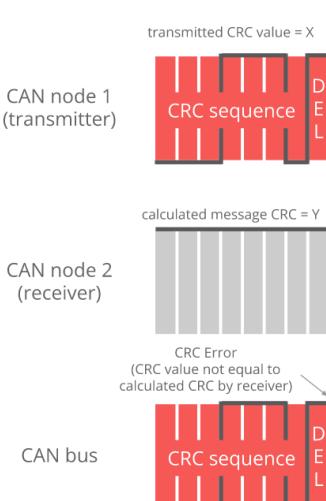
Example: CAN Bus ACK Error



#4 ACK Error (Acknowledgement)

When a transmitter sends a CAN message, it will contain the ACK field (Acknowledgement), in which the transmitter will transmit a recessive bit. All listening CAN nodes are expected to send a dominant bit in this field to verify the reception of the message (regardless of whether the nodes are interested in the message or not). If the transmitter does not read a dominant bit in the ACK slot, the transmitter detects this as an ACK Error.

Example: CAN Bus CRC Error



#5 CRC Error (Cyclic Redundancy Check)

Every CAN message contains a Cyclic Redundancy Checksum field of 15 bits. Here, the transmitter has calculated the CRC value and added it to the message. Every receiving node will also calculate the CRC on their own. If the receiver's CRC calculation does not match the transmitter's CRC, the receiver detects this as a CRC Error.

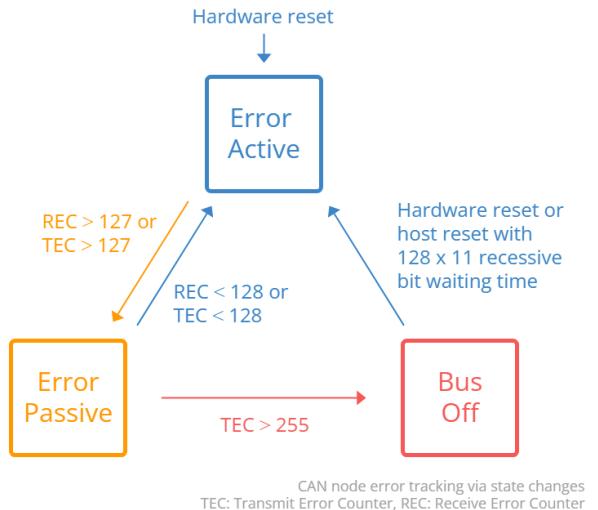
CAN node states & error counters

As evident, CAN error handling helps destroy erroneous messages - and enables CAN nodes to retry the transmission of erroneous messages.

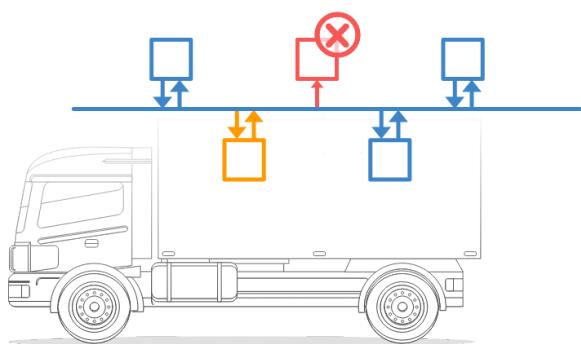
This ensures that short-lived local disturbances (e.g. from noise) will not result in invalid/lost data. Instead, the transmitter attempts to re-send the message. If it wins arbitration (and there are no errors), the message is successfully sent.

However, what if errors are due to a systematic malfunction in a transmitting node? This could trigger an endless loop of sending/destroying the same message - jamming the CAN bus.

This is where CAN node states and error counters come in.



CAN node error tracking via state changes
TEC: Transmit Error Counter, REC: Receive Error Counter



In short, the purpose of CAN error tracking is to confine errors by gracefully reducing the privileges of problematic CAN nodes.

Specifically, let's look at the three possible states:

1. **Error Active**: This is the default state of every CAN node, in which it is able to transmit data and raise 'Active Error Flags' when detecting errors
2. **Error Passive**: In this state, the CAN node is still able to transmit data, but it now raises 'Passive Error Flags' when detecting errors. Further, the CAN node now has to wait for an extra 8 bits (aka Suspend Transmission Time) in addition to the 3 bit intermission time before it can resume data transmission (to allow other CAN nodes to take control of the bus)
3. **Bus Off**: In this state, the CAN node disconnects itself from the CAN bus and can no longer transmit data or raise error flags

Every CAN controller keeps track of its own state and acts accordingly. CAN nodes shift state depending on the value of their error counters. Specifically, every CAN node keeps track on a Transmit Error Counter (TEC) and Receive Error Counter (REC):

- A CAN node enters the Error Passive state if the REC or TEC exceed 127
- A CAN node enters the Bus Off state if the TEC exceeds 255

How do the error counters change?

Before we get into the logic of how error counters are increased/reduced, let us revisit the CAN error frame as well as the primary/secondary error flags.

As evident from the CAN error frame illustration, a CAN node that observes a dominant bit after its own sequence of 6 dominant bits will know that it raised a primary error flag. In this case, we can call this CAN node the 'discoverer' of the error.

At first, it might sound positive to have a CAN node that repeatedly discovers errors and reacts promptly by raising an error flag before other nodes. However, in practice, the discoverer is typically also the culprit causing errors - and hence it is punished more severely as per the overview.

CAN bus error counter logic

TEC +8	Transmitter raises primary error flag
REC +8	Receiver raises primary error flag
REC +1	Receiver raises secondary error flag
REC -1	Receiver successfully receives message
TEC -1	Transmitter successfully sends message

TEC: Transmit Error Counter, REC: Receive Error Counter

Details on TEC/REC counters

There are some additions/exceptions to the above rules, see e.g. this overview.

Most are pretty straight-forward based on our previous illustrative example. For example, it seems clear that CAN node 1 would increase the TEC by 8 as it discovers the Bit Error and raises an error flag. The other nodes in this case increase their REC by 1.

This has the intuitive consequence that the transmitting node will quickly reach the Error Passive and eventually Bus Off states if it continuously produces faulty CAN messages - whereas the receiving nodes do not change state.

The case where a receiver raises the primary error flag may seem counter-intuitive. However, this could for example be the case if a receiver CAN node is malfunctioning in a way that causes it to incorrectly detect errors in valid CAN messages. In such a case, the receiver would raise the primary error flag, effectively causing an error. Alternatively, it can happen in cases where all CAN nodes simultaneously raise error flags.



CAN/LIN data & error logger

The [CANedge1](#) lets you easily record data from 2 x CAN/LIN buses to an 8-32 GB SD card - incl. support for logging CAN/LIN errors. Simply connect it to e.g. a car or truck to start logging - and decode the data via [free software/APIs](#). Further, the [CANedge2](#) adds WiFi, letting you auto-transfer data to your own server - and update devices over-the-air.

Examples: Generating & logging error frames

We have now covered the theoretical basics of CAN errors and CAN error handling. Next, let us look at generating and logging errors in practice. For this we will use a couple of CANedge devices - and for some tests a PCAN-USB device.

Test #1: No CAN bus errors

As a benchmark, we start with a test involving no CAN bus errors. Here, a CANedge2 'transmitter' sends data to another CANedge2 'receiver' - and both log CAN bus errors.

By loading the MF4 log file in the asammcf GUI we verify that no CAN errors occurred during this test, which is to be expected.



Test #2: Removing the CAN bus terminal resistor

In this test, we remove the CAN termination in the middle of a log session. This effectively corresponds to immediately setting the bit level to dominant. As a result, the CANedge2 transmitter immediately starts logging **Bit Errors** (which occur when it attempts to transmit a recessive bit, but reads a dominant bit). The CANedge2 Receiver logs **Bit Stuffing Errors** as it detects 6 consecutive dominant bits. These errors are recorded until the termination is added again.

[see the online article for screenshots of the errors in asammcf]

How relevant is this in practice?

Lack of termination is rarely a practical issue if you're recording data from a vehicle, machine etc. However, it's a common issue when working with 'test bench' setups. Here, the lack of termination may cause confusion as it can be difficult to distinguish from an inactive CAN bus. If in doubt, enabling error frame logging on the CANedge can be useful in troubleshooting.

Test #3: Setting an incorrect baud rate

In this test we configure the CANedge receiver node to have a baud rate of 493.827K vs. the baud rate of the transmitter of 500K. This is a fairly extreme difference and results in **ACK Errors** for the transmitter and **Bit Stuffing Errors** for the receiver.

In more realistic scenarios, smaller differences in the baud rate configuration of various nodes may cause intermittent error frames and thus message loss.

[see the online article for screenshots of the errors in asammcf]



How relevant is this in practice?

This example is rather extreme. However, in practice we have sometimes seen CAN buses that use standard bit rates (250K, 500K, ...), but with specific bit timing settings that differ from the ones that are typically recommended (and hence used by the CANedge). This will not lead to a complete shut-down of the communication, but rather periodic frame loss of a few percentages. To resolve this, you can construct an 'advanced bit rate' in the CANedge configuration, essentially setting up the bit-timing to better match the CAN bus you're logging from.

Test #4: Removing the acknowledging CAN node

In this test, we use three CANedge units configured as follows:

- CANedge1: Configured to acknowledge data
- CANedge2 A: Configured in 'silent mode' (no acknowledgement)
- CANedge2 B: Configured to transmit a CAN frame every 500 ms

In the default setup, data is transmitted by the CANedge2 B onto the CAN bus and recorded with no errors. However, if we remove the CANedge1 from the bus there are no longer any CAN nodes to acknowledge the frames sent by the transmitter.

As a result, the transmitter detects **ACK Errors**. In response, it increases its Transmit Error Counter and raises Active Error Flags onto the CAN bus. These are in turn recorded by CANedge2 A (which silently monitors the bus) as **Form Errors**.

[see the online article for screenshots of the errors in asammcf]



The CANedge records Form Errors due to the fact that the transmitter raises them upon identifying the lack of a dominant bit in the ACK slot. As soon as a dominant bit is observed by the receiver in the subsequent EOF field (which should be recessive), a Form Error is detected.

As evident, the transmitter broadcasts 16 Active Error Flags as its TEC is increased from 0 to $16 \times 8 = 128$. The transmitter has now exceeded the threshold of a TEC of 127 and enters Error Passive mode. As a result, the transmitter still experiences ACK Errors, but now only raises Passive Error

Flags (not visible to the receiver). At this point, the transmitter keeps attempting to transmit the same frame - and the receiver keeps recording this retransmission sequence.

How relevant is this in practice?

This type of error is one we often encounter in our support tickets. Specifically, users may be trying to use our CAN loggers to record data from a single CAN node (such as a sensor-to-CAN module like our CANmod). If they decide to enable 'silent mode' on the CANedge in such an installation, no CAN nodes will acknowledge the single CAN node broadcasting data - and the result will either be empty log files, or log files filled with retransmissions of the same CAN frame (typically at very high frequency).

Test #5: CAN frame collisions (no retransmission)

When setting up a CAN bus, it is key to avoid overlapping CAN IDs. Failing to do so can result in frame collisions as two CAN nodes may both believe they've won the arbitration - and hence start transmitting their frames at the same time.

To simulate this, we use the same setup as in test #4. In addition, we connect a PCAN-USB device as a secondary transmitter.

The CANedge2 transmitter is now configured to output a single CAN frame every 10 ms with CAN ID 1 and a payload of eight 0xFF bytes. Further, we configure the CANedge2 to disable retransmission of frames that were disrupted by errors. The PCAN-USB outputs an identical CAN frame every 2 ms with the 1st byte of the payload changed to 0xFE. The PCAN device has retransmissions enabled.

This setup quickly creates a frame collision, resulting in the CANedge and PCAN transmitters detecting a **Bit Error**. In response to this, both raise an Active Error Flag, which is detected as a **Bit Stuffing Error** by the CANedge receiver. The PCAN device immediately attempts a retransmission and succeeds, while the CANedge waits with further transmission until the next message is to be sent.

[see the online article for screenshots of the errors in asammcf]



How relevant is this in practice?

This type of error should of course never happen in e.g. a car, since the design and test processes will ensure that all CAN nodes communicate via globally unique CAN identifiers. However, this problem can easily occur if you install a 3rd party device (e.g. a sensor-to-CAN module) to inject data into an existing CAN bus. If you do not ensure the global uniqueness of the CAN IDs of external CAN nodes, you may cause frame collisions and hence errors on the CAN bus. This is particularly important if your external CAN node broadcasts data with high priority CAN IDs as you may then affect safety critical CAN nodes.

Test #6: CAN frame collisions (incl. retransmission)

In this test, we use the same setup as before, but we now enable retransmissions on the CANedge2 transmitter.

In this case, the frame collision results in a sequence of subsequent frame collisions as both the CANedge2 and the PCAN-USB device attempt to re-transmit their disrupted messages.

Due to the resulting Bit Errors, both raise a total of 16 Active Error Flags, which are detected as Bit Stuffing Errors by the silent CANedge2 receiver. Both transmitters then enter Error Passive mode and stop raising Active Error Flags, meaning none of them can destroy CAN frames on the bus. As a result, one of the transmitters will succeed in transmitting a full message, thus ending the retransmission frenzy - and enabling both devices to resume transmission. However, this only lasts for a few seconds before another collision occurs.

How relevant is this in practice?

The collision handling is a good example of how effective the CAN error handling is at 'shutting down' potentially problematic sequences and enabling CAN nodes to resume communication. If a frame collision occurs, it is likely that both CAN nodes will be set up to attempt retransmission, which would cause a jam if not for the error handling and confinement.

LIN bus errors

Similar to CAN bus errors, the LIN protocol also specifies a set of four error types, which we outline briefly below. The CANedge supports both CAN/LIN error frame logging.

#1 LIN Checksum Error

As for the CAN CRC Error, this error type implies that a LIN node has calculated a different checksum vs. the one embedded in the LIN bus frame by the transmitter. If you're using the CANedge as a LIN Subscriber, this error may indicate that you've configured the device 'frame table' with incorrect identifiers for some of the LIN frames on the bus.

This can in turn be used to 'reverse engineer' the correct lengths and IDs of proprietary LIN frames via a step-by-step procedure. See the CANedge Docs for details.

#2 LIN Receive Error

These occur if a specific part of the LIN message does not match the expected value, or if there is a mismatch between what is transmitted vs. read on the LIN bus.

#3 LIN Synchronization Error

This error indicates an invalid synchronization field in the start of the LIN frame. It can also indicate a large deviation between the configured bit rate for a LIN node vs. the bit rate detected from the synchronization field.

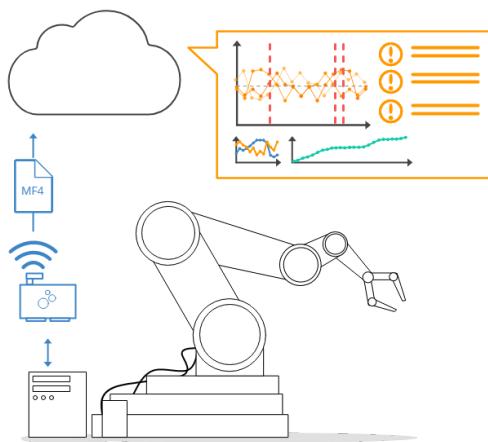
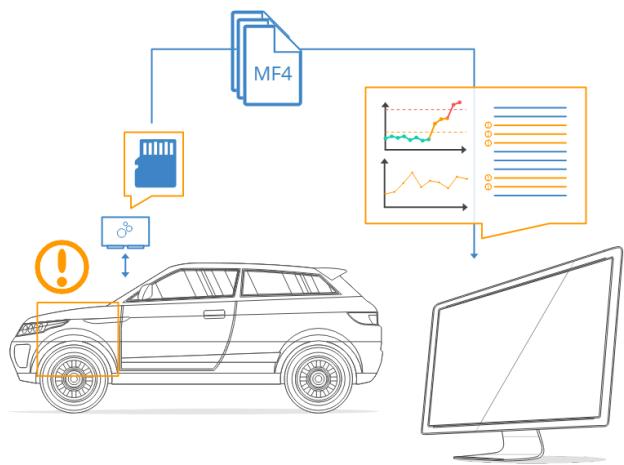
#4 LIN Transmission Error

Transmission errors can occur for LIN identifiers registered as SUBSCRIBER messages. If there are no nodes responding to a SUBSCRIBER message, a transmission error is logged.

Example use cases for CAN error frame logging

CAN bus diagnostics in OEM vehicles

An automotive OEM may have the need to record CAN error frames in the field during late stage prototype testing. By deploying a CANedge, the OEM engineering team will both be able to troubleshoot issues based on the actual CAN signals (speed, RPM, temperatures) - as well as issues related with the lower layer CAN communication in their prototype systems. This is particularly vital if the issues of interest are intermittent and e.g. only happen once or twice per month. In such scenarios, CAN bus interfaces are not well suited - and it becomes increasingly relevant to have a cost-effective device to enable scalable deployments for faster troubleshooting.



Remotely troubleshooting CAN errors in machinery

An OEM or aftermarket user may need to capture rare CAN error events in their machines. To do so, they deploy a CANedge2 to record the CAN data and related error frames - and automatically upload the data via WiFi to their own cloud server. Here, errors are automatically identified and an alert is sent to the engineering team to immediately allow for diagnosing and resolving the issue.

FAQ

Does the CANedge support all CAN/LIN error types?

Yes, the CANedge is able to record all CAN/LIN error types. It does, however, not currently record its own error counter status as this is deemed less relevant from a logging perspective.

Will a CANedge raise error flags?

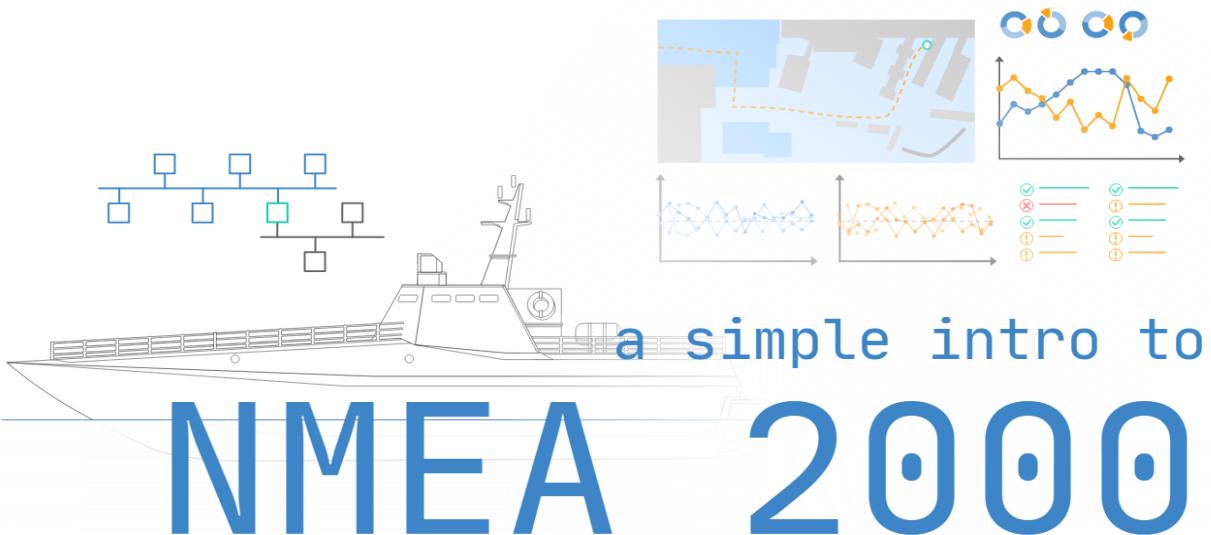
The CANedge is only able to raise error flags onto the CAN bus if it is configured in its 'normal' mode, in which it is also able to transmit messages. If in 'restricted' mode it can listen to CAN frames and acknowledge CAN frames - but not raise Active Error Flags onto the bus. In 'monitoring' mode (aka 'silent mode') it can listen to the CAN bus traffic, but not acknowledge messages nor raise Active Error Flags. The CANedge will always record internal CAN/LIN error frames.

What information can be recorded regarding a CAN/LIN error?

If a CAN frame is erroneous, resulting in an error frame, the CANedge generally only records the error type - without any data related to the erroneous frame (beyond the timestamp). One exception to this rule is for acknowledgement errors, where the CANedge will still record unacknowledged CAN frames (incl. from retransmission attempts).

Is error handling a cybersecurity risk?

Some researchers have pointed out the risk that 'bad actors' could utilize the CAN bus error handling functionality to enforce remote 'bus off' events for safety-critical ECUs. This is a good example of why CAN bus data loggers & interfaces like the [CANedge2](#) with remote over-the-air data transfer and updates need to be highly secure (see also our [intro to CAN cybersecurity](#)). For a nice overview of a remote bus off attack, see [this intro](#) by Adrian Colyer.



NMEA 2000 Explained - A Simple Intro

Need a simple, practical intro to NMEA 2000?

In this guide we introduce the N2K protocol (IEC 61162-3) used in maritime vessels such as boats, yachts, tugs and other ships. In particular, we'll cover the NMEA 2000 standards, key concepts, links to other protocols (NMEA 0183, CAN/J1939), topology and connectors. We also cover NMEA 2000 PGNs and Fast Packets.

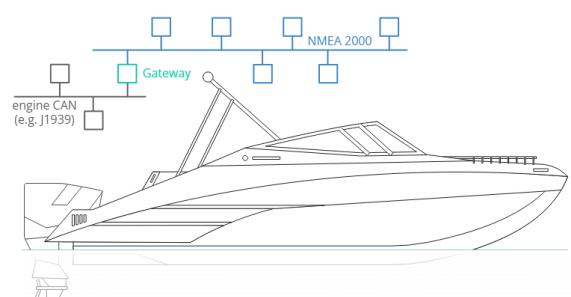
To ensure this stays practical, we also explain how to record and decode NMEA 2000 data - with practical use cases and examples. Learn more below!

What is NMEA 2000?

NMEA 2000 (IEC 61162-3) is a communication standard used in the maritime industry for connecting e.g. engines, instruments and sensors on boats. It is based on the [Controller Area Network \(CAN\)](#) and allows data to be sent/received between devices over a single network 'backbone' cable.

The higher layer functionality of NMEA 2000 (aka N2K) is based on [SAE J1939](#) and [ISOBUS](#) (ISO 11783), commonly used in heavy-duty vehicles and agriculture/forestry machinery, respectively.

NMEA 2000 succeeds the older NMEA 0183 and is today used in most modern maritime vessels like boats, ships, yachts and related equipment/sensors.



Multiple boat networks

Importantly, a maritime vessel may use multiple networks.

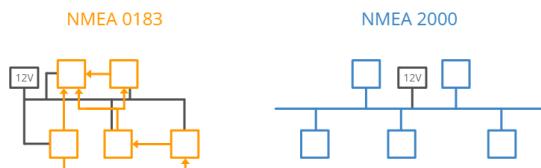
For example, a twin engine speed boat may comprise an NMEA 2000 network connecting both outboard engines to the helm accessories (control head, keys, displays). This network will typically be constructed by the boat OEM. The engines themselves may internally communicate via e.g. the J1939 protocol with a gateway/filter module feeding engine related data to the NMEA 2000 network.

In addition, various sensors measuring e.g. GPS position, wind, water depth etc. may be added to same NMEA 2000 network. However, cases also exist where such sensors are retrofitted by the boat operator onto a separate NMEA 2000 network.

NMEA 2000 vs NMEA 0183

NMEA 0183 (IEC 61162-1) is the predecessor to NMEA 2000 (IEC 61162-3). The NMEA organization published the NMEA 0183 protocol with the purpose of standardizing the communication between different electronic equipment in maritime vessels.

The protocol replaced NMEA 0180 and NMEA 0182. The NMEA 0183 protocol remains in active use today, although NMEA 2000 has become the industry standard.



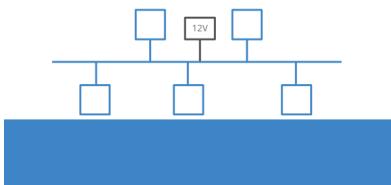
Details on NMEA 0183

The NMEA 0183 protocol uses the electrical standard RS232/RS422 for communication instead of CAN bus. In this setup, one 'talker' (e.g. a GPS module) can communicate with multiple 'listener' nodes (e.g. an autopilot and a chartplotter). However, it is not possible to have multiple 'talkers' on the same network (in contrast to CAN). As a result, a separate NMEA 0183 network is required for every 'talker' node - which quickly becomes complex when the networks are interconnected.

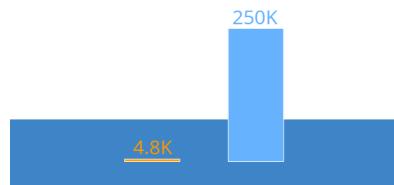
Data is communicated in an ASCII format as standardized 'sentences' defined in the application layer. The data is transmitted as printable ASCII characters in the range 0x20 to 0x7E and reflect 'physical values' (e.g. degrees), rather than 'raw data' as in NMEA 2000. For a detailed overview of the message/sentence structure, see [this table](#).

Key benefits of NMEA 2000 vs. NMEA 0183

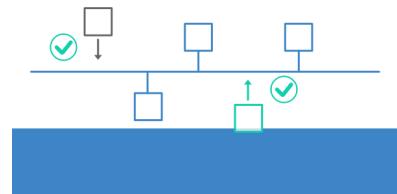
Overall, NMEA 2000 is a far more modern and flexible communication protocol compared to NMEA 0183, and is therefore today widely used in marine electronic systems. Below we outline the key advantages:



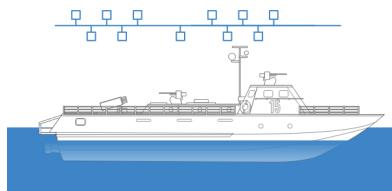
Simpler & lower cost network
NMEA 2000 allows all nodes to communicate with each other (multiple talkers) and the power supply is directly integrated in the backbone, reducing wiring costs and simplifying the network vs. NMEA 0183. The standardized T-connectors also simplify integration



Faster data transmission
NMEA 2000 enables a higher data transmission rate than NMEA 0183 (250K vs. 4.8K), meaning it can transmit more data in a given amount of time. This is useful when real-time data is required, such as in navigation or engine monitoring applications



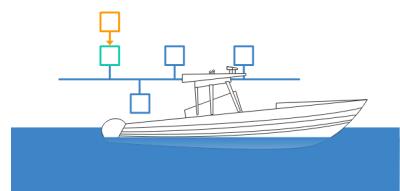
Better device compatibility
NMEA 2000 is designed to allow different devices from different manufacturers to be connected to the same network and to exchange data in a standardized way. This makes it easier to mix and match devices on the same network, and reduces the risk of compatibility issues



Larger network size
NMEA 2000 networks can be larger than NMEA 0183 networks, as they are designed to support more devices (up to 50 nodes) and transmit data over longer distances. This can be useful in larger vessels or systems where a greater number of devices need to be connected



More robust communication
NMEA 2000 uses the Controller Area Network (CAN) bus, which is a digital communication protocol that is less susceptible to noise and interference than the serial communication protocol used by NMEA 0183. This makes NMEA 2000 networks more reliable and resilient



Backwards compatibility
NMEA 2000 enables integration of NMEA 0183 devices via low cost gateways, enabling the integration of existing electronics into new networks. The reverse is, however, not true - i.e. NMEA 2000 nodes cannot be added to NMEA 0183 networks

NMEA history and certification

The [National Marine Electronics Association \(NMEA\)](#) is a non-profit organization founded in 1957. It represents manufacturers, distributors, and dealers in the maritime industry and helps ensure device compatibility across brands/manufacturers through the promotion of data communication standardization, education and certification.

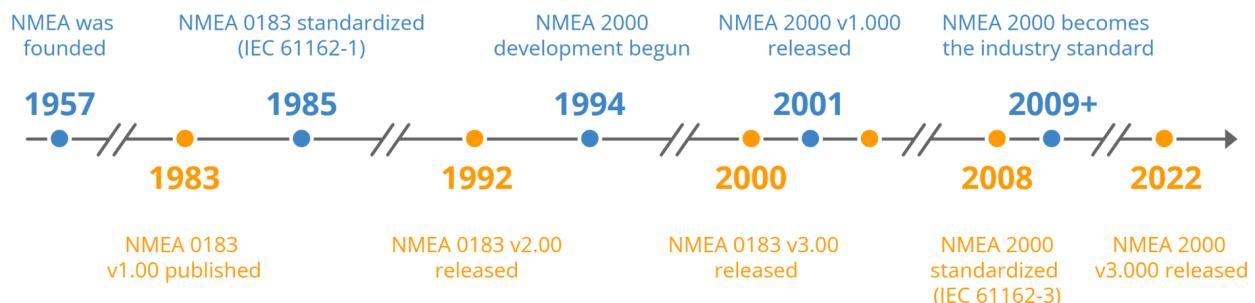
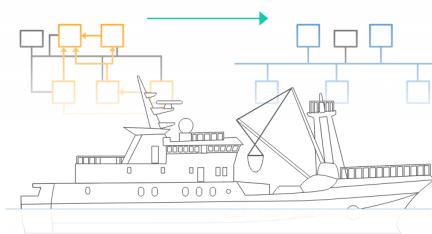


NMEA and AEF

To some extent, the role of NMEA can be compared to that of the Agricultural Industry Electronics Foundation (AEF) in the agriculture/forestry industry. Both NMEA 2000 and ISOBUS share the need for strict protocol conformance requirements in order to deliver on the promise of plug & play integration of products across manufacturers. Here, the organisations play a key role in facilitating a standardised process for product certification - as well as in providing a database for identifying conforming manufacturers/products.

NMEA history

- 1957: National Marine Electronics Association founded
- 1983: NMEA 0183 v1.00 published by NMEA
- 1985: NMEA 0183 standardized as IEC 61162-1
- 1992: NMEA 0183 v2.00 released
- 1994: N2K development begun by Standards Committee
- 2000: NMEA 0183 v3.00 released
- 2001: NMEA 2000 v1.000 released
- 2008: NMEA 2000 standardized as IEC 61162-3
- 2008: NMEA 0183 v4.00 released
- ~2009: NMEA 2000 becomes the industry standard
- 2022: NMEA 2000 v3.000 released

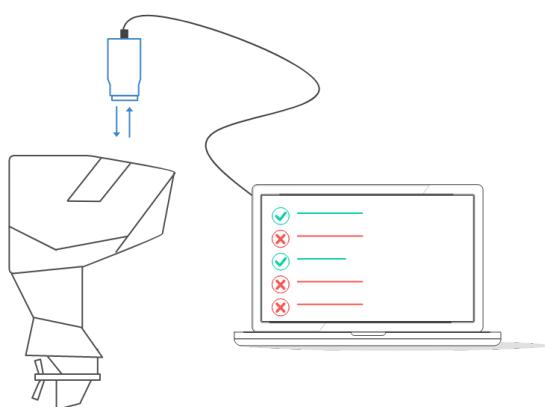


NMEA 2000 conformance certification

To ensure interoperability of N2K nodes, NMEA facilitates a certification process through which a product can become 'NMEA 2000 Certified' and add a certification logo to the product. In practice, the certification is done via a self-certification process using mandated test equipment (the NMEA 2000 Certification Tool, i.e. a CAN interface/software) and validation by NMEA. Certification does not imply that a specific product must produce certain data. Rather, it ensures that a product is compatible with other NMEA 2000 certified products - enabling them to coexist on the same network without causing interference.

NMEA certification classes and levels

The NMEA 2000 product certification can be granted as both a Class 1 and Class 2, with a Level A and B in each. Class 1 devices rely on a single NMEA 2000 interface for communication, while Class 2 devices provide two interfaces for the purpose of redundancy. For each class, Levels A and B allow for distinguishing between sophisticated devices (with full communication support incl. e.g. implementation of ISO TP communication) and more simplistic devices with a smaller scope of required communication functionality.



NMEA 2000 approved vs. certified

Some third-party products may use alternative classifications from the 'NMEA 2000 Certified':

1. NMEA 2000 Approved: These products are able to implement NMEA 2000 and have passed the physical hardware tests of the N2K certification program
2. NMEA 2000 Certified for use with [Approved Product Name]: These products may be used in combination with a specific NMEA 2000 Approved product, together creating a fully compatible product
3. NMEA 2000 Certified for use with [Certified Gateway Product Name]: These products may be used in combination with an NMEA 2000 Certified gateway product.

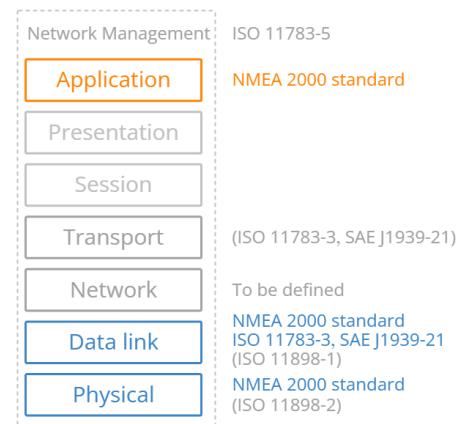
Finally, another classification is 'NMEA 2000 Approved Diagnostic Tool - Not for permanent connection to a NMEA 2000 backplane' - intended for tools that may e.g. disrupt safety-critical functionality in the NMEA 2000 network if used incorrectly.

NMEA 2000 OSI model & standards

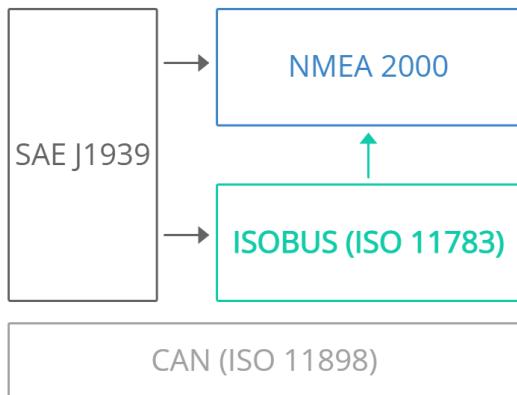
Below we briefly outline the link between each OSI model layer and the NMEA 2000 standard (as well as referenced standards).

- Application: This layer is fully defined by the NMEA 2000 standard and defines approved PGNs/signals incl. a provision for manufacturer-specific proprietary messages. NMEA Appendix A defines PGNs, while Appendix B provides a full list of the standardized NMEA 2000 PGNs and signals
- Transport: In NMEA 2000, the transport layer is responsible for establishing and maintaining connections between devices, as well as fragmenting and reassembling messages as needed. This can be done via the transport protocol used in ISOBUS/J1939 or via NMEA Fast Packets (see also the Data Link layer)
- Network: This layer is to be further defined in the future in the NMEA 2000 standard
- Data Link: This layer is responsible for transmitting data across the communication channel and ensuring that the data is delivered correctly. The NMEA 2000 standard uses the CAN protocol at the data link layer, which includes error detection and correction mechanisms to ensure reliable transmission. Note that this layer references ISO 11783-3 (ISOBUS), SAE J1939-21 and 11898-1 (CAN). The NMEA 2000 standard defines additional requirements/functionality, including the NMEA Fast Packet for multiframe communication
- Physical: This layer is fully defined by the NMEA 2000 standard, including signal voltages, cables and connectors. Note also the inclusion of 11898-2 (CAN) to reflect the role of CAN bus as the basis for NMEA 2000. This also specifies the standard baud rate of 250K for NMEA 2000 and details on cables and connectors

7 layer OSI model | **NMEA 2000**



Further, the 'Network Management' is defined by ISO 11783-5 (ISOBUS). Note in particular that all NMEA 2000 compliant devices must support dynamic address claiming



NMEA 2000 vs. J1939 vs. ISOBUS

As evident, NMEA 2000 is closely related to J1939 and ISOBUS. Here we briefly clarify the link between these protocols. First, let's do a very quick history recap:

In 1994, the Society of Automotive Engineers (SAE) published the J1939 protocol for use in heavy-duty vehicles. Next, ISOBUS was derived from J1939 for agriculture/forestry machinery, formally standardized in 2007 (ISO 11783).

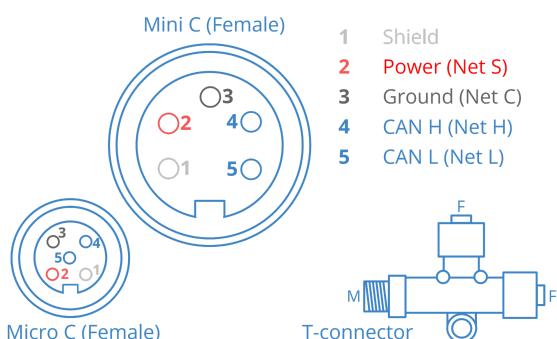
NMEA 2000 was released in 2001 and references parts of SAE J1939 and ISO 11783 almost interchangeably. ISO 11783-3 is referenced for the data link layer, but in turn references J1939-21. Similarly, NMEA 2000 references ISO 11783-5 for the network management, which is virtually identical to J1939-81. In other words, NMEA 2000 is based on ISO 11783 standards that are in turn based on J1939 standards.

Practical implications

In practical applications, NMEA 2000 is often seen in combination with J1939 or ISOBUS. For example, some boat engines use J1939 internally and deploy a gateway/filter to parse certain information onto standardized NMEA 2000 PGNs. In some cases, NMEA 2000 and J1939 messages even co-exist on the same CAN bus. This is possible as none of the NMEA 2000 PGNs overlap with J1939 PGNs. Similarly, NMEA 2000 messages are often broadcast onto an ISOBUS network since many GPS modules are used in both maritime and agricultural use cases. See our [ISOBUS intro](#) for details.

NMEA 2000 connectors & network topology

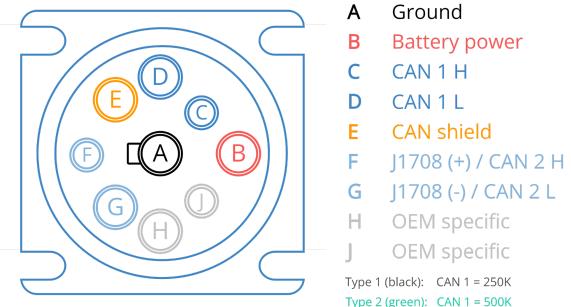
Below we describe the most common connectors encountered in maritime vessels:



#1 T connector (M12 5-pin)

The T-connector (aka T-piece) is used in constructing the NMEA 2000 CAN bus 'backbone' in a linear fashion and for connecting equipment. It is a three-way connector consisting of two female connectors and one male connector. The connectors are standardised DeviceNet 5-pin A-coded M12 connectors. The physical connector and pinout differ between mini type (for backbones) and micro type (for small backbones and for drop cables). The micro M12 connector is also used in industrial applications like CANopen and enable easy connection of CAN based equipment - e.g. CAN bus data loggers.

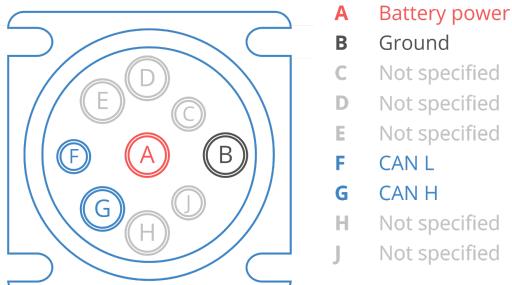
[m12 adapter cable](#)



#2 Engine connector (J1939 9-pin)

Many maritime engines (e.g. from Cummins) deploy the J1939 protocol for their internal communication. Some of this information may be broadcast onto the NMEA 2000 network directly or through a gateway that converts the PGN information to the corresponding NMEA 2000 encoding. Many engines come with separate diagnostic connectors that allow direct access to the raw J1939 data - and the most common connector for these would be the J1939 deutsch 9-pin connector.

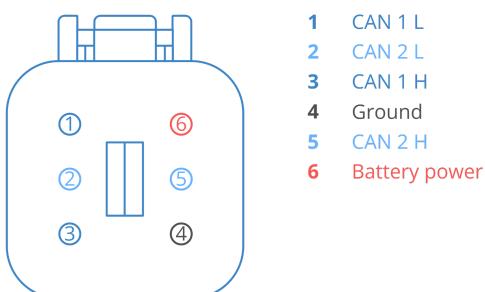
[j1939 adapter cable](#)



#3 Engine connector (CAT 9-pin)

Some maritime engines (e.g. from Caterpillar) use a deutsch 9-pin CAT adapter. This looks almost identical to the J1939 9-pin connector, but uses a different pinout.

[cat adapter cable](#)

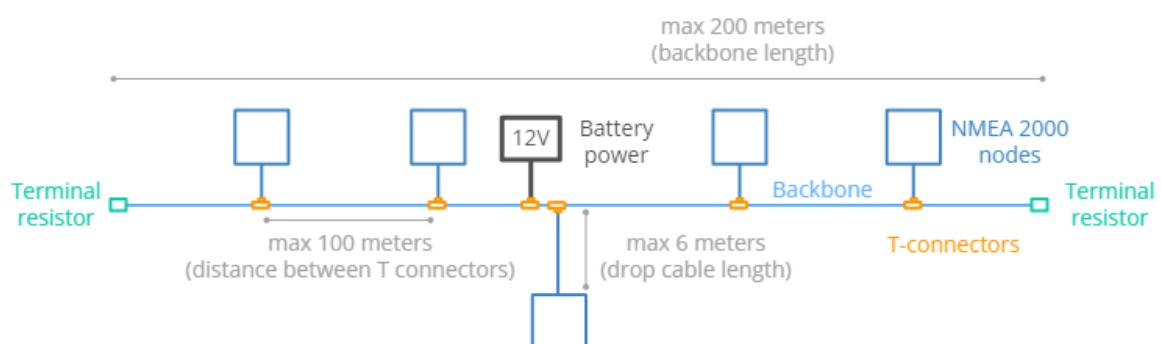


#4 Engine connector (Deutsch DT06-6P)

In some maritime engines (e.g. from Volvo Penta), alternative diagnostic connectors are used such as the DT06-6P (male 6-pin) and DT06-6S (female 6-pin). You'll often be able to convert from these connectors to other popular connectors (like OBD2) via off-the-shelf adapters.

NMEA 2000 physical layer requirements

The NMEA 2000 standard also specifies requirements in terms of the physical layer as shown below.



Details on the NMEA 2000 topology

- The standard bit rate is 250 kb/s
- The CAN bus should be constructed as a linear 'backbone'
- The distance between two T-connectors must not exceed 100m
- The maximum length of the network is 200m
- The network supports up to 50 connected devices
- The max 'drop cable' length (between a T-connector and device) is 6m
- The max cumulative drop cable length is 78m
- Cables must adhere to [DeviceNet](#) (ODVA) standards
- The backbone must include a grounded power supply in the 9-16 V DC range (e.g. from the vessel 12 V DC power supply or from an isolated 15 V DC supply)
- Three cable types: Mini/thick (for backbones requiring 4-8 amps per leg), mid (for backbones) and micro/thin (for drop cables and small backbones)

NMEA 2000 Fast Packet

A key part of the NMEA 2000 standard is the 'Fast Packet'.

As per SAE J1939-21 and ISO 11783-3, the NMEA 2000 standard supports multi-packet communication of up to 1785 bytes via the ISO 15765-2 (ISO TP) standard.

However, NMEA 2000 often requires frame payloads that exceed 8 bytes, but far less than 1785 bytes. Thus, the Fast Packet was introduced as a more efficient transport protocol for medium-size payloads up to 223 bytes without any transfer protocol delays. See the table overview for a comparison.

NMEA 2000 - Single Frame vs. ISO TP vs. Fast Packets

Single Frame	ISO TP (ISO 11783 Multi Packet)	NMEA 2000 Fast Packet
Up to 8 bytes	Up to 1785 bytes	Up to 223 bytes
Destination defined by PGN	Send any PGN to specific device	Destination defined by PGN
No handshaking	Optional handshaking	No handshaking
No transfer protocol delay	Transfer protocol delays	No transfer protocol delay
PGN is part of CAN ID	PGN is part of data payload	PGN is part of CAN ID

Frame identification

In the NMEA 2000 Fast Packet methodology, every frame retains the original PGN identifier. This means that the multi-frame message can be uniquely identified at the ID level, in contrast to the ISO TP implementation in J1939, ISOBUS and **UDS**, in which message identification requires extraction of information from the 1st CAN frame payload data.

Fast Packet trace example

Below we show an NMEA 2000 Fast Packet example. As evident, all the transmitted frames contain a 'sequence counter' in the 1st nibble and a 'frame counter' in the 2nd nibble of the 1st byte. Further, the first frame transmitted contains the total payload length in the 2nd byte. To enable decoding, the frame needs to be reassembled as shown.

Timestamp (Epoch)	CAN ID	PGN (HEX)	PGN (DEC)	DataBytes	legend
1620692704.079450	11F80F00	1F80F	129039	001B127CEAD5123D	NMEA 2000 PGN (0x1F80F = AIS Class B Position Report)
1620692704.080000	11F80F00	1F80F	129039	0131F3D0ACF2231A	#data bytes (0x1B=27)
1620692704.080600	11F80F00	1F80F	129039	0203FFFF00000000	sequence counter
1620692704.081200	11F80F00	1F80F	129039	0320FFFF0070FEFF	frame counter
...					payload data (27 bytes)
1620692705.079500	11F80F00	1F80F	129039	201B127CEAD5123D	
1620692705.080050	11F80F00	1F80F	129039	2131F3D0ACF2231A	
1620692705.080650	11F80F00	1F80F	129039	2207FFFF00000000	
1620692705.081200	11F80F00	1F80F	129039	2320FFFF0070FEFF	
1620692704.079450	11F80F00	1F80F	129039	127CEAD5123D ... 20FFFF0070FEFF	
1620692705.079500	11F80F00	1F80F	129039	127CEAD5123D ... 20FFFF0070FEFF	

The [CANedge Python API](#) enables automatic reassembly of Fast Packets into single with the full data payload (stripped of the sequence counters and payload length). This in turn enables easy DBC decoding of the data using the [NMEA 2000 DBC file](#). See our [api-examples repository](#) for script examples.



N2K PGN & data fields

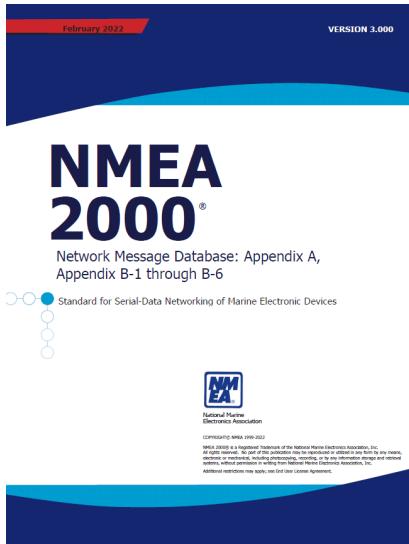
NMEA 2000 uses the concept of Parameter Group Numbers (PGN) from the J1939 protocol. This means that messages are identified based on 18-bit PGNs, which can be extracted from the full 29-bit CAN identifiers.

Signals aka parameters will be 'packaged' in the data payload of each message. Note that in contrast to J1939, signals are not referred to as Suspect Parameter Numbers (SPN), but instead as Data Fields (DF) or simply parameters.

We do not cover the PGN structure here as it is already covered in our [SAE J1939 intro](#). However, we will provide a few practical comments on NMEA 2000 PGNs and parameters.

NMEA 2000 message (PGN and Data Fields)





NMEA 2000 PGN database

While NMEA 2000 was originally based on J1939, the standard J1939 PGNs (from J1939-71) do not sufficiently cover the maritime requirements of the NMEA standard.

To solve this, the NMEA 2000 application layer defines a number of PGNs unique to NMEA 2000, which are described in the NMEA 2000 standard, Appendix B. The appendix lists a number of PGNs and the data fields contained within each of them. Each data field is described through a 'Data Dictionary'. This can be compared to how J1939-71 describes each PGN and their underlying SPNs.

As evident from the PGN extract, the Appendix B PDF contains information for decoding parameters like Wind Speed. However, extracting this manually would be time-consuming as the PDF contains 170+ PGNs and 1500+ parameters. Instead, we recommend using the [NMEA 2000 DBC file](#), which includes the PGN and parameter information in a structured way, ready for use in decoding raw NMEA 2000 data.

Wind Data		PGN: 130306 hex: 1FD02	
Direction and speed of Wind. True wind can be referenced to the vessel or to the ground. The Apparent Wind is what is felt.		The apparent wind is the wind as it is experienced by the vessel. It is the vector sum of the true wind and the drift caused by the motion of the vessel through the water. The apparent wind angle is the angle between the direction of the true wind and the direction of the apparent wind.	
Single Frame: Yes	Priority Default: 2	Default Update Rate: Command Support: Optional	100 milliseconds ACK Retime: None Frequency: 10 cycles per second
Destination: Global	Query Support: Optional		
Field #	Field Name		
1	Sequence ID	Byte Field Size: 1 Bit Field Size:	Request Parameter: Command Parameter: Optional Optional
DD056 Sequence ID		An 8-bit counter number that tracks information transmitted in two or more PGNs from a single source address. Identical SBD values within two or more different PGN transmissions identifies those PGN transmissions as a single sequence.	
DF53	Integer, 8 bit unsigned	uint8 Range: 0 to 252	Resolution: 1 bit Unit-less number
2	Wind Speed	Byte Field Size: 2 Bit Field Size:	Request Parameter: Command Parameter: Optional Optional
DD044 Generic Speed			
0.00	0.00	0.00	

Regarding proprietary NMEA 2000 PGNs

The NMEA 2000 standard allows manufacturers to create their own proprietary PGNs, though with three conditions:

1. The proprietary PGN contains device specific information (e.g. a unique calibration)
 2. The data is being used for test purposes
 3. A suitable PGN is not available in the current NMEA 2000 PGN database

In general, manufacturers are encouraged to leverage the standardized PGNs (or migrate to these when they become available). Further, if proprietary PGNs are used in a device, NMEA encourages manufacturers to publish the decoding information.

Examples of NMEA 2000 signals in Appendix B

The NMEA 2000 appendix B contains 1500+ parameters aka Data Fields, encoded in NMEA 2000 specific PGNs. Below are some examples of the maritime-oriented signals included:

- Alerts
 - GNSS position
 - Vessel speed
 - Heading
 - Rudder angle

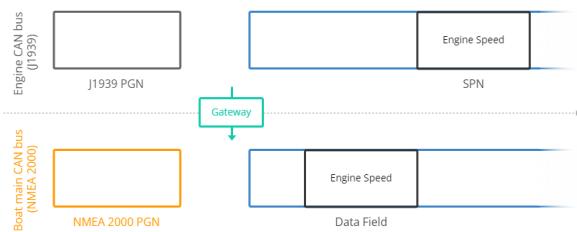
- Rate of turn
- Yaw, pitch, roll
- Engine speed
- Temperatures
- Fuel pressure
- Fuel rate
- Engine load
- Transmission
- Battery status
- Water depth
- Wind speed
- Air temperature
- ... and more

NMEA vs J1939 PGNs [+DBC files]

In maritime vessels, J1939 is often used 'within the engines' (e.g. outboard engines) to communicate various engine-related sensor data like engine speed, temperatures, etc.

In such cases, a 'gateway' will typically convert a subset of the J1939 PGN information into NMEA 2000 PGNs, thus allowing e.g. displays to visualize the engine information.

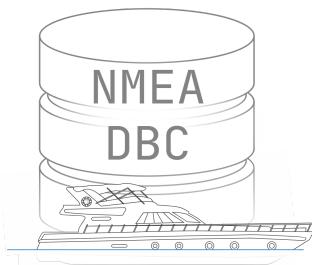
For example, the J1939 PGN 0xF004 (61444) contains Engine Speed - while it is encoded in the N2K PGN 0x1F200 (127488).



Original J1939 PGN transmission

As such, if you're recording data only from the NMEA 2000 network, you'll typically get the 'pure' NMEA 2000 PGN information plus the 'filtered' J1939 PGN information (encoded as NMEA 2000 PGNs). In some cases, however, the 'original' J1939 PGNs will be directly output onto the NMEA 2000 network. This is possible because the protocols are 'harmonized', i.e. they use the same bit rate of 250K and none of the CAN IDs overlap.

In short, to maximize the vessel data you can decode, you should consider the following DBC files:



NMEA 2000 data (vessel, sensors)

If you connect to the NMEA 2000 network (e.g. via an M12 connector), you will generally record the broadcast NMEA 2000 data. To decode this, you can use the NMEA 2000 DBC, which will allow you to extract information on the standardized PGNs from the network. This may include 'filtered' engine information as well.

[learn more](#)

J1939 data (engines)



In some cases, the NMEA 2000 network may also contain the original J1939 PGNs from the engine. Alternatively, you may be able to access the original J1939 information by connecting directly to the engine via a separate diagnostic connector (typically a deutsch 9-pin J1939 connector). In either case, you can use the J1939 DBC file to decode the standardized J1939 PGNs.

[learn more](#)



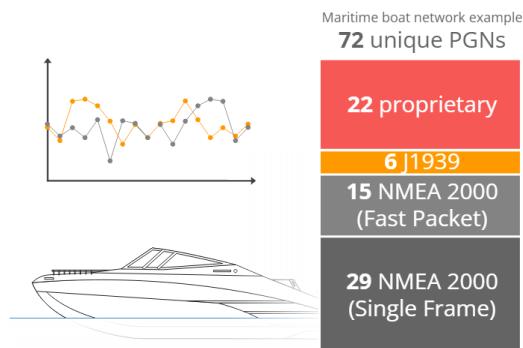
OEM data (proprietary)

There will almost always be some NMEA 2000 PGNs that are proprietary. These are in principle only known to the OEM, but since the NMEA organization encourages OEMs to publish their encoding rules, it is likely that you will find the relevant information in technical manuals or on-request. If so, you can add them to your existing DBC file by modifying it via a DBC editor (see our DBC intro for details).

Example: Decoding data from a boat

To showcase the breakdown of PGNs you'll often see in maritime CAN data, consider a log file from a speed boat. In this case, a [CANedge1](#) has been used to record data from the NMEA 2000 backbone, which also includes (at least some of) the original J1939 data from the outboard engine. The log file contains **161 unique CAN IDs**, corresponding to **72 unique PGNs**.

As evident, the majority of the PGNs are decoded by the NMEA 2000 DBC (as Single Frames or Fast Packets). Further, the J1939 DBC is able to decode some of the J1939 PGNs, while the rest are OEM specific. The proprietary PGNs may originate from the J1939 engine network or some of the NMEA 2000 network equipment.



Note regarding J1939 PGNs

Note here that the connection of the CANedge1 was made on the NMEA 2000 network - not directly onto the J1939 network (i.e. the engine's diagnostic connector). As a result, it is not known in this case whether more J1939 PGNs would be available via a direct connection to the engine. Further, in most NMEA 2000 networks, the engine J1939 PGNs are not directly broadcast onto the NMEA 2000 network, hence you'd normally need to record these separately. This is often done with the CANedge, connecting the 1st port to the NMEA 2000 network and the 2nd port to the J1939 engine diagnostic connector.

Logging NMEA 2000 maritime data

To log maritime CAN bus data from an NMEA 2000 network or J1939 engine, we recommend to check out our [maritime telematics](#) intro. Below we will briefly recap the most relevant steps to consider:

#1 How to choose a CAN logger

If you need to log N2K/J1939 data to an SD card we recommend the [CANedge1](#). The device can be left in the vessel to record data for weeks or months. To collect the data, simply extract the SD. Alternatively, the [CANedge2](#) lets you auto-upload data to your own server via WiFi. The device logs data to the SD card and uploads it when the vessel is near-shore with connectivity.



#2 How to connect to the N2K network

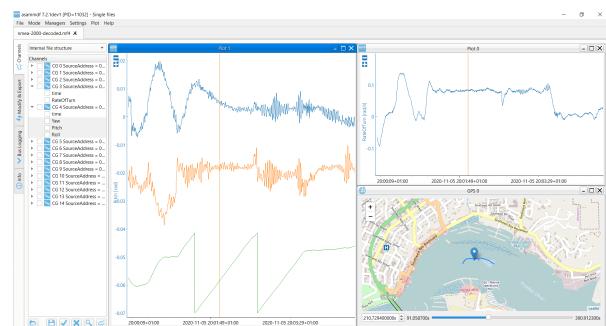
Next, you'll need to identify how to connect to the CAN bus. The simplest way to connect to the NMEA 2000 network is to use a [DB9-M12 adapter](#) to connect via a T-connector. This gives you access to all the raw NMEA 2000 data from that network. Optionally, you can use the 2nd CAN channel of the CANedge to connect to another NMEA 2000 network (if one exists) or directly to the J1939 engine network. In the latter case, you can use one of the engine diagnostic connectors listed previously.

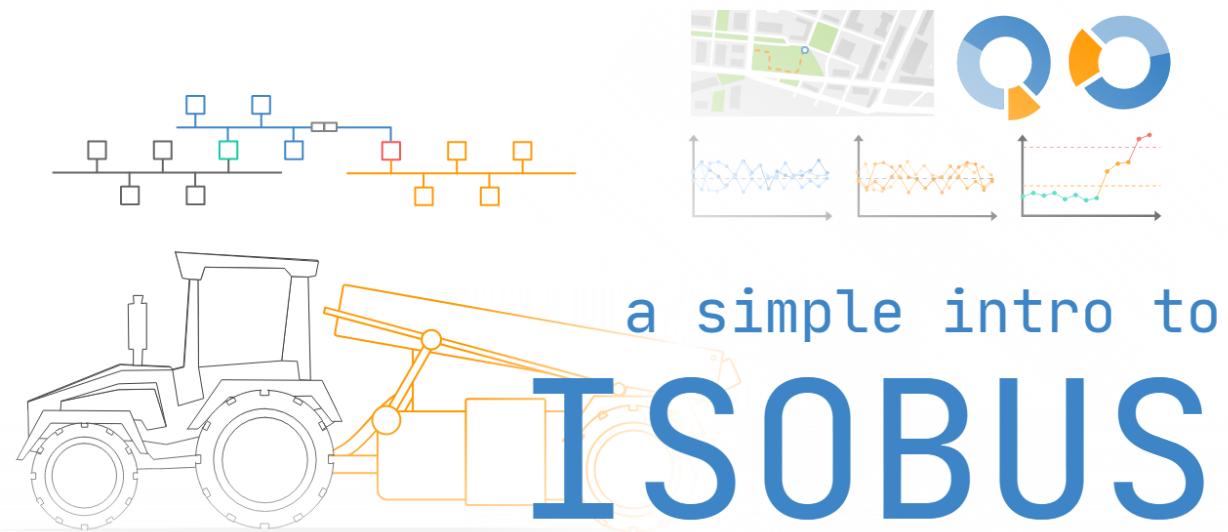


#3 How to process the CAN data

To process the raw NMEA 2000 and/or J1939 vessel data, you need a CAN software tool. In this, you'll load the log files and related DBC files to convert the raw data to physical values.

The CANedge software tools enable different forms of analysis. For example, you can use the [MF4 converters](#) to transform the raw log files to popular file formats (e.g. ASC, TRC, CSV), for use in tools like Vector CANalyzer, Warwick X-Analysier 3 or PEAK PCAN-Explorer. Or, you can use the [asammdf GUI](#) to DBC decode the data and create visual plots incl. GPS maps of your vessel routes. Further, you can deploy automated data processing via the [Python API](#) if you need to perform statistical analysis, automated reporting or database integration. The API can also be used to integrate the data with [Grafana dashboards](#).





ISOBUS (ISO 11783) Explained - A Simple Intro

Need a simple, practical intro to ISOBUS (ISO 11783)?

In this guide we introduce the ISOBUS protocol used in agricultural vehicles (like tractors) and implements (like sprayers, seeders). In particular, we'll cover the ISO 11783 standards, history, the link to J1939 and connectors.

To ensure this stays practical, we also explain how to record and decode ISOBUS data - with practical use cases and [dashboard playground examples](#).

Learn more below!

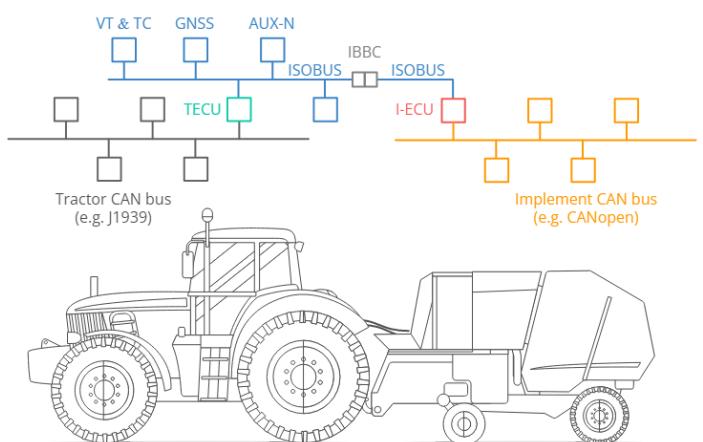
What is ISOBUS?

ISOBUS (ISO 11783) is a standardised communication protocol used in agriculture and forestry machinery. The main purpose is to enable plug & play integration between vehicles (e.g. tractors) and implements (e.g. sprayers) across manufacturers.

The protocol is based on the [Controller Area Network](#) (CAN bus). Specifically, it has been derived from the [SAE J1939](#) protocol used in most heavy-duty vehicles (including most tractors).

Today, ISOBUS is implemented in most modern tractors and implements (like balers, sprayers, fertilizers, seed drills) and enables standardised communication between them.

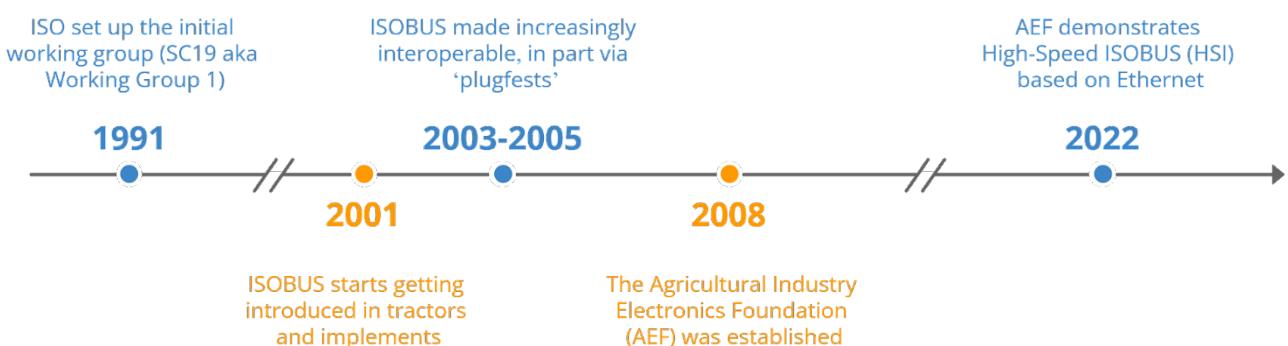
For example, a tractor may use J1939 internally. The Tractor ECU (TECU) then serves as a gateway to the tractor's ISOBUS network. Similarly, a sprayer may use e.g. J1939/[CANopen](#) internally, while an ISOBUS gateway facilitates the communication with the tractor. Assuming ISOBUS is properly implemented in both tractor and implement, the integration is seamless.



With certified ISOBUS solutions, end users avoid cluttering cabins with implement-specific terminals - and instead control any implement from a single universal terminal.

ISOBUS history and AEF

- 1991: ISO set up the initial working group (SC19 aka Working Group 1)
- 2001: ISOBUS started getting introduced in tractors and implements
- 2003-2005: ISOBUS made increasingly interoperable, in part via 'plugfests'
- 2008: The [Agricultural Industry Electronics Foundation](#) (AEF) was established
- 2022: AEF demonstrates High-Speed ISOBUS (HSI) based on Ethernet



AEF and ISOBUS certification

When ISOBUS solutions initially started rolling out in the early nineties, most were early-stage solutions. From 2001, ISOBUS experienced an adoption 'breakthrough' with tens of thousands of new ISOBUS tractors/implements sold globally.

However, in practice, the standard was complex and interpreted differently by manufacturers - causing incompatibility issues in the field (and a general loss of confidence in the ISOBUS promise). In response, AEF was established in 2008.



AEF members

AEF originally comprised seven agriculture equipment manufacturers (John Deere, CNH, Claas, AGCO, Kverneland Group, Grimme, Pottinger) and two associations (Association of Equipment Manufacturers, AEM, and German Engineering Federation, VDMA). Today, FederUnacoma serves as the 3rd core member association, while there are now 8 manufacturer core members including KUHN. In total, more than 200 companies/institutions are now members of AEF.



AEF has a set of core objectives

- Define guidelines for implementations of ISOBUS
- Coordinate improvements to ISOBUS incl. certification tests
- Coordinate collaboration in the agriculture industry
- Organize certification support, training and consulting

Further, AEF has developed a number of tools and products:

- AEF Conformance Test: Used by AEF accredited test labs to certify ISOBUS components
- AEF Database: Used by dealers/users globally to assess compatibility and conformance of ISOBUS components

Practical relevance

The conformance testing and related database are vital in ensuring a consistent, standardized implementation of ISOBUS - allowing end users to lookup components across brands and manufacturers to ensure compatibility prior to purchase. This ensures that the original promise of a 'plug & play' cross-brand experience with ISOBUS vehicles and implements can be realized in practice.

ISOBUS OSI model & standards

Next, we will briefly consider the 7 layer OSI model for ISOBUS including the link to the 14 parts of ISO 11783 standard.

Note how we also reference CAN bus (ISO 11898) for the physical layer and data link layers, to reflect the role CAN plays as the basis for ISOBUS.

While not shown in the illustration, SAE J1939 also plays a role as reference for some of the layers/standard parts - for example, J1939-71 provides the definition for most powertrain messages as described in ISO 11783-8.

Below we briefly recap the 14 parts of the ISO 11783 standard.

7 layer OSI model | ISOBUS (ISO 11783)

Application	ISO 11783-7/8/9/10/12/13/14
Presentation	ISO 11783-6/7/11
Session	ISO 11783-6/7/10/14
Transport	ISO 11783-3/5
Network	ISO 11783-4/5
Data link	ISO 11783-3 (ISO 11898-1)
Physical	ISO 11783-2 (ISO 11898-2)

ISO 11783-1: General standard for mobile data communication

This serves as an overview of the entire ISO 11783 standard collection, providing the basis for a standardized serial communication network between forestry/agriculture tractors and implements. It describes the OSI model layers and references the other standards.

ISO 11783-2: Physical layer

This standard specifies the physical layer of the network. Since ISOBUS is based on CAN, this standard is closely linked to ISO 11898-2 describing the CAN bus physical layer. In addition, ISO 11783-2 describes ISOBUS specific connectors like the ISOBUS Breakaway Connector (IBBC).

ISO 11783-3: Data link layer

This standard describes the ISOBUS message formats. Specifically, it dictates the use of extended CAN IDs and the use of J1939-style Parameter Group Numbers (PGN). The standard also specifies a transport protocol for the communication of multiframe ISOBUS messages, i.e. data that exceeds the 8 bytes of Classical CAN frames.

ISO 11783-4: Network layer

Here, the use of 'network interconnect units' are described for connecting two separate networks with differing architectures. This gateway ensures both electrical and message isolation of each network.

ISO 11783-5: Network management

This defines the process for determining source addresses (SA) and resolving address conflicts. Source addresses can be preset or dynamically claimed by each controller during power up.

ISO 11783-6: Virtual terminal

This standard describes the virtual terminal as an ECU that provides the tractor operator with an interface for controlling the tractor and/or implement(s). The control should be exerted through the use of the standardized ISO 11783 messages. The VT is a core feature of ISOBUS.

ISO 11783-7: Implement messages application layer

This standard describes standardized application messages for the ISOBUS implement. This also serves as the basis for most of the messages in our ISOBUS DBC file. To some extent, this can be compared to J1939-71, which serves as the basis for most of the J1939 PGN and SPN decoding information. In the case of ISOBUS, messages described include e.g. wheel/ground based speed and direction, key switch state, aux valves, lights and more.

ISO 11783-8: Powertrain messages

This describes powertrain messages used in the tractor network implementation. Effectively, it references the SAE J1939-71 standard, stating that any overlapping messages should be implemented as per J1939-71.

ISO 11783-9: Tractor ECU

When an implement is connected to a tractor, it is critical to ensure consistent cross-communication. Here, the tractor ECU should allow the implement to request information from ECUs on the tractor network in accordance with ISO 11783-7, as well as respond to implement commands. This also describes details on safe mode operation.

ISO 11783-10: Task controller and management information system data interchange

A 'farm management computer' may be available on the implement network. This enables the operator (typically a farmer) to define tasks/actions for the implement to perform. The 'task controller' receives these tasks via an interface from the computer. The tasks are scheduled by the controller, which ensures execution in the implement through

control functions. The controller also receives status information from the implement ECUs, which is parsed back to the farm management computer. The standard defines the operation of the task controller, including the standardized messages used for the control functions.

ISO 11783-11: Mobile data element dictionary

This is closely linked to 11783-10, providing a data dictionary to define the data elements used in the task controller messages.

ISO 11783-12: Diagnostics services

This specifies messages used for diagnostics to identify faults and issues on the network. It can be compared to J1939-73

ISO 11783-13: File server

This defines an ECU that provides data storage on the network (e.g. for multimedia), as well as a set of commands for use by read/write data to the file server.

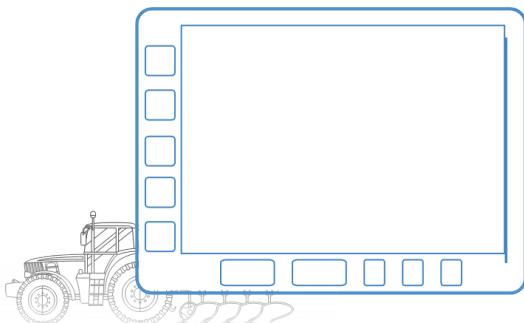
ISO 11783-14: Sequence control

This more recent standard (first issued in 2013) details how to perform sequence control within the ISO 11783 network. Here, an operator can record sequences of control functions for later on-demand replay.

ISOBUS Functionalities

AEF refers to a number of ISO 11783 based 'products' as ISOBUS Functionalities. Effectively, these can be sold to end users as a separate modules on the ISOBUS. In this section we briefly detail each of the AEF ISOBUS Functionalities:

#1 Virtual Terminal (VT) / Universal Terminal (UT)

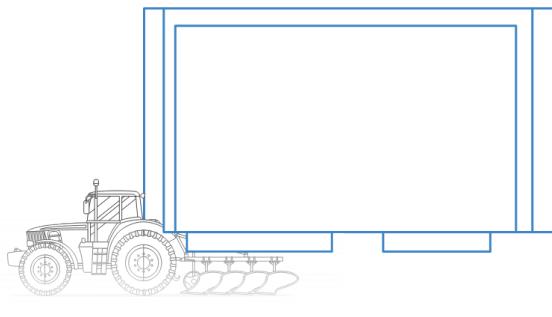


This refers to the Virtual Terminal specified in ISO 11783-6. The VT is sometimes referred to as an Universal Terminal (UT) because of the capability it provides: To allow the end user to operate different implements (even across brands/manufacturers) via a single terminal. We'll refer to it as the VT throughout.

The VT is a core part of what makes ISOBUS visually attractive: Instead of a messy tractor cabin containing a multitude of implement-specific displays, everything can now be handled through a single display - across all ISOBUS compatible implements.

In practice, the VT is initialized through a sequence of CAN messages being sent between the VT and the implement. Here, the VT informs the implement of

its specifications, allowing the implement to send its user interface to the VT for graphical visualization. Once initialized, the operator can perform various implement control tasks through the VT.

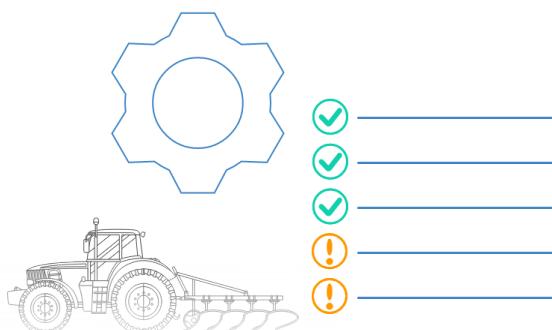


#2 Tractor ECU (TECU)

The Tractor ECU (TECU) serves as a gateway between the tractor CAN bus (typically J1939) and the ISOBUS. The TECU provides selective info to the implement like speed, RPM etc. from the tractor network via the ISOBUS.

In more recent versions of Tractor ECUs, the implement can take over full control of the tractor - incl. steering, speed, braking, transmission etc. This enables increasingly autonomous solutions where tractor and implement operate seamlessly as a single system.

Note also that the implement often has a similar ECU, sometimes referred to as the Implement ECU (I-ECU). This serves as a gateway between the implement's internal CAN bus and the ISOBUS.

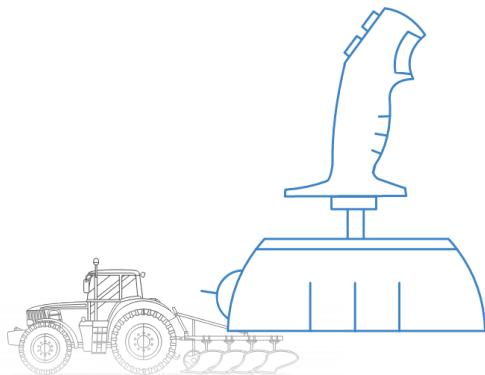


#3 Task Controller (TC)

The Task Controller is a piece of software that records and provides information on operations and assists in planning. It takes inputs from the operator via an interface and serves to schedule tasks for the implement, executing them through control functions. In particular, it allows for automated and thus more fine-grained control over the implement - key in e.g. precision farming.

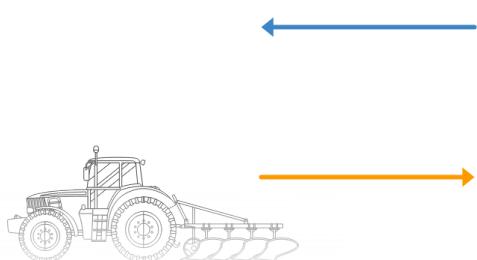
The Task Controller consists of three sub protocols:

1. Basic (TC-BAS): This is used for tracking data totals (e.g. how much product has been applied), with info provided by the implement. This makes it easy to e.g. extract documentation of the work done via the operator interface
2. Section (TC-SEC): This enables automated switching of sections in e.g. a sprayer based on for example GPS position, allowing for more precise use of implement functionality
3. Geo (TC-GEO): This is used for GPS-based task automation, such as executing geo-based seeding control. For example, this is useful if certain areas in a field need customized handling



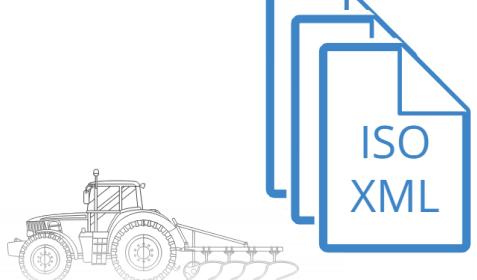
#4 Auxiliary Control (AUX)

This ISOBUS Functionality often reflects a joystick or similar control element, which enables more fine-grained control over implement functionality. Note that both a new/old version exists which are not cross-compatible (AUX-N/AUX-O).



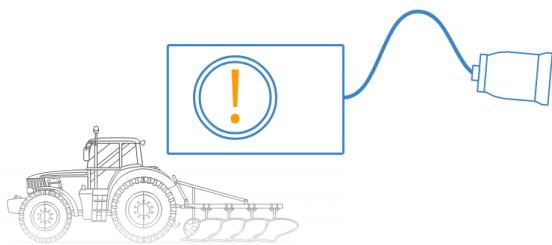
#5 Tractor Implement Management (TIM)

As mentioned in the TECU section, there is increasingly a demand for bi-directional control between tractor and implement, meaning in particular that the implement should eventually be able to control the tractor in some use cases. This would improve productivity and reduce operator fatigue.



#6 Logging of device values (LOG)

This is focused on logging data from the tractor/implement, independently of the task being executed. To some extent, it can be compared to retrofitting a CAN data logger to record all messages being communicated on the CAN buses. However, in the case of LOG, the aim is to allow for the data to be exported in the ISO XML format (similar to what is supported for Task Controller data).



#7 ISOBUS Shortcut Button (ISB)

This enables deactivation of selective implement functions. This can be relevant in use cases where multiple implements are being controlled from a single tractor Virtual Terminal. Here, one implement might be in the foreground, while the user needs to quickly deactivate functionality in another implement.

ISOBUS vs SAE J1939

A common question is how ISO 11783 (ISOBUS) relates to the SAE J1939 protocol.

In short, ISOBUS was originally derived from J1939 and a core goal of ISO 11783 is to remain compatible with J1939 through close alignment with SAE. In a sense, one can say that ISOBUS is today harmonized with J1939. This is useful as many tractor networks will be based on SAE J1939, meaning that Tractor ECUs (TECU) often need to serve as gateways between ISOBUS and J1939.

J1939 vs ISOBUS (ISO 11783)

Topic	J1939	ISOBUS
Application layer	PGNs and SPNs	Virtual Terminal, Task Controller, File Server ...
Address claiming	Optional	Mandatory
Fast packet TP	N/A	Used for GNSS (NMEA 2000)
Extended TP	N/A	Used for e.g. VT, File Server
Physical layer/connector	See J1939-1X	See ISO 11783-2 (different)
Conformance test	Optional	Mandatory in practice via AEF

Further, various J1939 standards serve as direct references for the ISOBUS standards. For example, J1939-71 defines powertrain messages and serves as a reference for ISO 11783-8. Both ISOBUS and J1939 also rely on CAN bus (CAN 2.0B) as the lower layer for communication and both leverage the concept of 18 bit PGNs to identify messages.

However, J1939 is mainly used in 'closed systems' and therefore ISOBUS has been extended beyond J1939 to serve the need for plug & play implements. It is therefore important to note that ISOBUS is not equal to J1939. We outline a number of key differences in the table. For details see below:

ISO 11783 vs. J1939 - details

In particular, note that the ISOBUS application layer includes a number of additional components, such as the Virtual Terminal and Task Controller.

ISOBUS also enables alternative transport protocol methods like Fast Packets e.g. for GNSS data (also known from [NMEA 2000](#)) and Extended Transport Protocol (e.g. for sending data to the VT or File Server). ISOBUS also introduces a number of Data Dictionary Identifiers, which are similar to SPNs - though they are generally transmitted with 1 SPN per message, rather than multiple. Due to the need for implement interoperability, dynamic address claiming is mandatory in ISOBUS, whereas it's often skipped in J1939 (due to the closed system logic). Further, ISOBUS introduces a number of alternative physical connectors vs. J1939 as we'll detail shortly.

The ISOBUS connectors

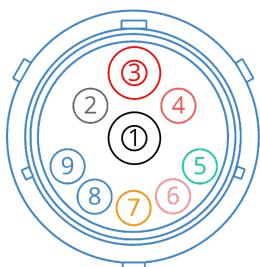
ISO 11783-2 describes a number of physical connectors used in tractors and implements.



- 1 TBC power
- 2 CAN H
- 3 TBC return path
- 4 CAN L

#1 Bus extension connector

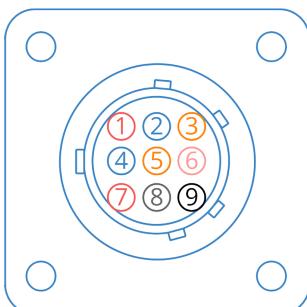
This connector enables the extension of the ISOBUS signal lines from the implement within the tractor. This is e.g. used for adding additional devices such as Virtual Terminals. The connector will be located in the tractor cabin on the right side of the operator's seat.



- 1 Ground (chassis)
- 2 ECU ground
- 3 Power
- 4 ECU power
- 5 TBC connect/remove
- 6 TBC power
- 7 TBC return path
- 8 CAN H
- 9 CAN L

#2 Implement Bus Breakaway Connector

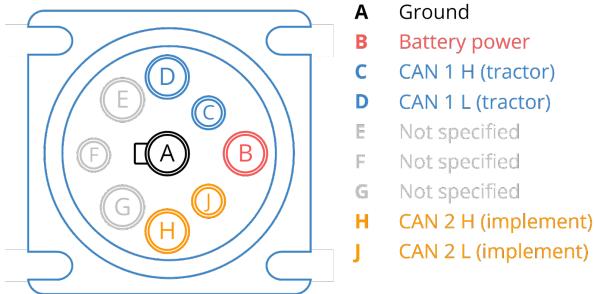
The IBBC enables the connection of implements to the tractor. The connector is located at the rear of the tractor. In some cases an optional IBBC may also be installed on the front of the tractor for use with implements that are front mounted. The IBBC must have a cap for covering it against dust and the weather when no implement is connected.



- 1 ECU power
- 2 CAN 1 L (input)
- 3 CAN 2 L (output)
- 4 CAN 1 H (input)
- 5 CAN 2 H (output)
- 6 TBC power
- 7 ECU power
- 8 TBC ground
- 9 ECU ground

#3 In-cab connector (LBS)

The tractor cabin may include an optional in-cab connector for establishing a connection to the ISO 11783 network. This is sometimes referred to as LBS.



#4 Diagnostic connector

The diagnostic connector is located in the tractor cabin in a location that is easily accessible. As we'll detail shortly, the diagnostic connector can be a good option for connecting external data logging devices for diagnostics - in particular if you need access to both the J1939 and ISOBUS networks in parallel.

ISOBUS PGN and SPN [+ DBC]

ISO 11783 uses the concept of Parameter Group Numbers (PGN) and Suspect Parameter Numbers (SPN) from the SAE J1939 protocol. This means that messages are identified based on 18-bit PGNs, which can be extracted from the full 29-bit CAN identifiers. In turn, signals or parameters are referred to as SPNs and will be 'packaged' in the data payload of each message.

We will not cover the PGN structure in detail as it is already covered in our SAE J1939 intro. However, we will provide a few practical comments on ISOBUS PGNs and SPNs.

ISOBUS PGN and SPN database

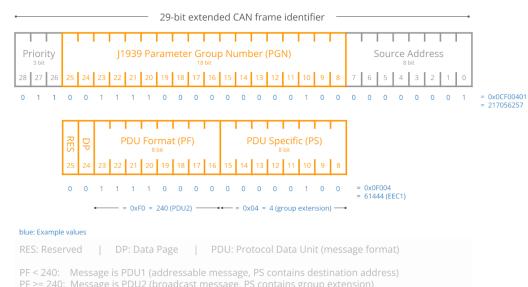
The ISO 11783-7 standard specifies PGN and SPN details unique to ISOBUS. The decoding details are publicly available from the [VDMA ISOBUS Data Dictionary](#). Signals include:

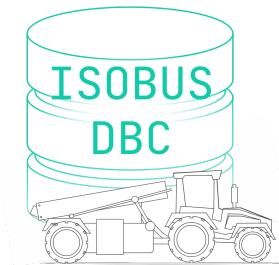
- Front/rear hitch pitch/roll/yaw angle
- Machine selected speed
- PTO information - rear shaft speed
- Engine speed, wheel based speed
- Tractor lights

The public database includes decoding rules for ~800 signals, of which ~650 can be directly transformed to a format useful in e.g. telematics and data logging. As explained below, we provide a DBC version of these ~650 ISO 11783 message decoding rules. Note that the ISOBUS network may also contain OEM proprietary data that cannot be decoded using the public information.

ISOBUS vs. J1939 vs. NMEA 2000 [+ DBC files]

If you record data from a tractor/implement, multiple protocols (and hence [DBC files](#)) may be relevant.





ISOBUS data (implement/tractor)

Naturally, the ISOBUS network will contain data related to the communication between tractor and implement. This information will often be decodable using the ISO 11783-7 decoding rules mentioned above. To enable quick decoding of the standard ISOBUS data, you can use our [ISOBUS DBC file](#).

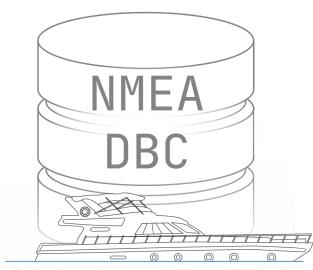
[learn more](#)



J1939 data (tractor)

If you record data directly from the tractor's CAN bus, it will most likely be J1939 data. Similarly, if you log data from the ISOBUS network, some of the data from the Tractor ECU will typically be J1939. In either case, the J1939 data requires the use of decoding rules from the J1939 Digital Annex. For this purpose, you can use our [J1939 DBC file](#).

[learn more](#)



NMEA 2000 data (GNSS)

Often, the ISOBUS network contains NMEA 2000 encoded GNSS information. The NMEA 2000 data may be broadcast as a combination of single-frames and NMEA 2000 Fast Packet TP frames. To decode the data, you can use our [NMEA 2000 DBC file](#).

[learn more](#)



OEM proprietary data (tractor/implement)

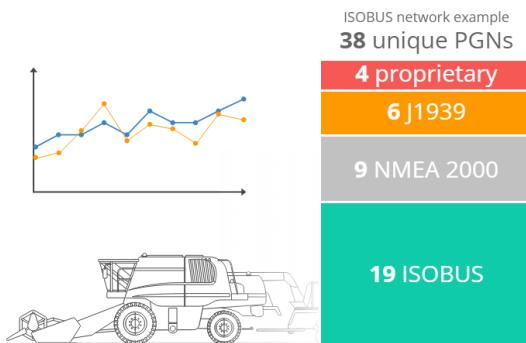
Of course, there will almost always be some messages that are proprietary regardless of the protocol. These are only known to the OEM (Original Equipment Manufacturer) and cannot be decoded using the standard DBC files. Sometimes these messages follow one of the above standards, though in other cases they may use another standard entirely (e.g. CANopen).

Example: Decoding data from an ISOBUS network

To exemplify the division that you'll often see in agriculture CAN bus data, consider a log file from a Fendt tractor with an implement from another manufacturer. In this case, a CANedge2 has been used to record data from the ISOBUS network. Through this, 337 unique CAN IDs have been recorded, corresponding to 38 unique PGNs.

The PGNs can be split by protocol as follows:

- ISOBUS DBC: 19 PGNs (~50%)
- NMEA 2000 DBC: 9 PGNs (~24%)
- J1939 DBC: 6 PGNs (~16%)
- Proprietary: 4 PGNs (~10%)



To get this ISOBUS sample data, get our [J1939 data pack](#).

Note on J1939 PGNs

Note here that the connection of the CANedge2 was made on the ISOBUS network - not directly onto the J1939 network. As a result, there are very few decodable J1939 PGNs, since they only reflect the J1939 PGNs repeated by the Tractor ECU (TECU) onto the ISOBUS. Had the CANedge2 been connected to the J1939 network directly, there would be a significantly higher amount of CAN IDs overall - including many more J1939 PGNs.

Logging tractor/implement data

In this section we outline how you can record and process data from tractors/implements.

#1 How to choose a CAN bus data logger

If you need to record your data offline, you can use the standalone [CANedge1](#) to log 2 x CAN to an SD card. The device can be left in the tractor to record data for weeks or months. To collect the data, simply extract the SD card.

Alternatively, you can use the [CANedge2](#). This adds WiFi transfer, allowing it to automatically upload data to your own server (self hosted or cloud). This enables e.g. large-scale data collection, automated reporting or predictive maintenance.



#2 How to connect to the CAN bus

Once you've selected a CAN bus data logger, you'll need to identify how to interface with the CAN networks. We generally recommend to use the diagnostic connector if possible. As outlined previously, this enables access to both the J1939 network of the tractor - and the ISOBUS network.

You can use our [J1939-DB9/DB9 \(H + I\)](#) adapter cable, which perfectly matches the diagnostic connector pinout and enables you to log both the J1939 and ISOBUS networks separately into the same log file on the CANedge.



Further details

Alternatively, you can also use our regular J1939-DB9 adapter if you only need to access the J1939 network. Often many of the messages will be repeated across networks, so for many use cases this may suffice.

Note that if you aim to connect directly to an implement diagnostic connector (rather than through the tractor cabin), the CAN 1 pins (C + D) will typically give access to the implement's internal CAN bus - while H + J will still give access to the ISOBUS network.

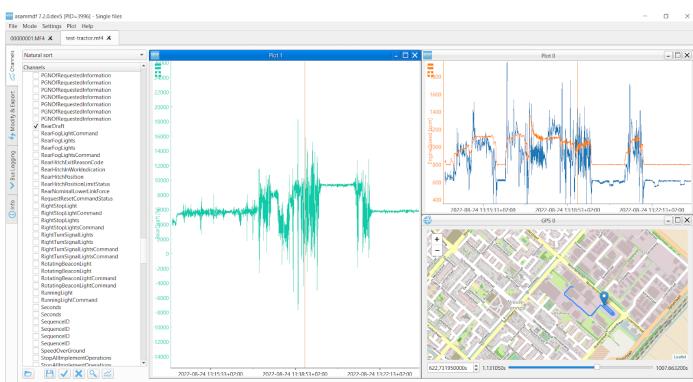
#3 How to process the CAN data

To process the raw tractor/implement data, you'll need a suitable CAN software tool. Specifically, you'll load the log files and related DBC files to extract the physical values.

The CANedge [software tools](#) enable various forms of analysis. For example, you can use the [MF4 converters](#) to quickly convert the log files to popular file formats, for use in software like Vector CANalyzer or PEAK PCAN-Explorer. Or, you can use the [asammdf GUI](#) to DBC decode the data and create visual plots.

You can also perform advanced data processing via the [Python API](#) - ideal if you need to perform statistical analysis, automated reporting or database integration.

The Python API can also be used to integrate the data with [dashboards](#) - more below.



Example: Tractor dashboard with ISOBUS, J1939 and NMEA 2000 data

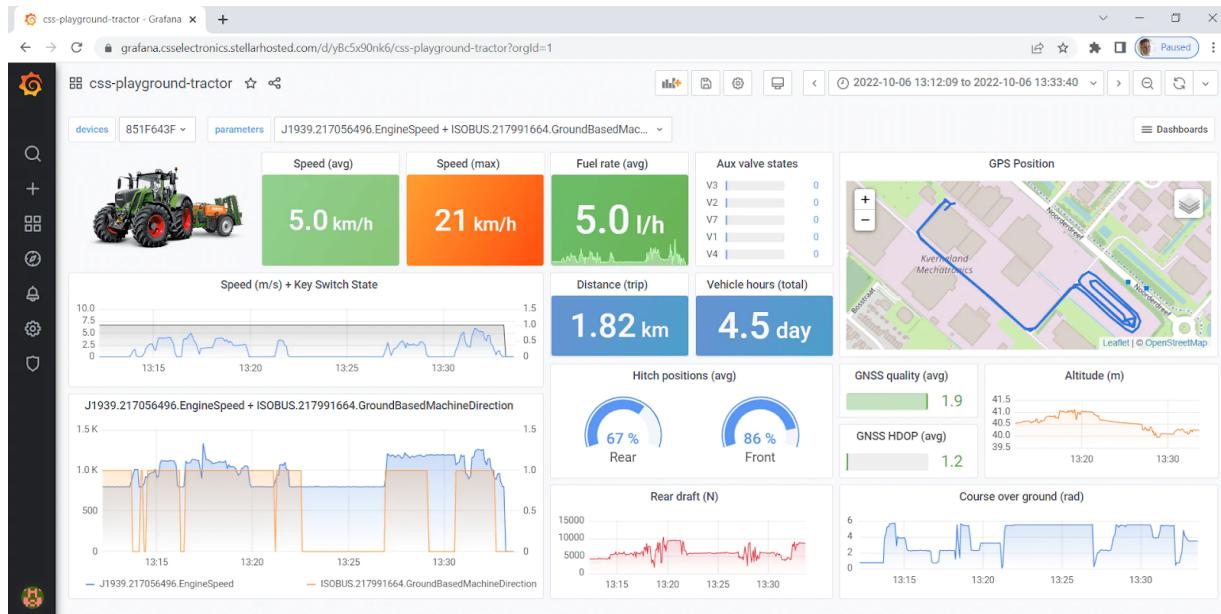
To illustrate the role of ISOBUS, J1939 and NMEA 2000 in tractor/implement data logging, check out our Grafana dashboard playground. Here, each signal is prefixed with the protocol from which it originates.

See also our [telematics dashboard intro](#) to learn more about Grafana dashboards.

Details on the recorded data

The log file was recorded using a CANedge2 connected to the implement diagnostic connector. Here, the CANedge2 specifically recorded data from the ISOBUS network. In other words, the log file does not include the tractor's internal J1939 data, except for the signals that are repeated onto the ISOBUS network.

To decode the data, we used both the J1939 DBC, ISOBUS DBC and NMEA 2000 DBC. We also leveraged the ISO TP functionality of the CANedge Python API to extract some of the NMEA 2000 Fast Packet data. Note that the proprietary OEM specific data is not included in the playground as this is not part of the standardized DBC files.

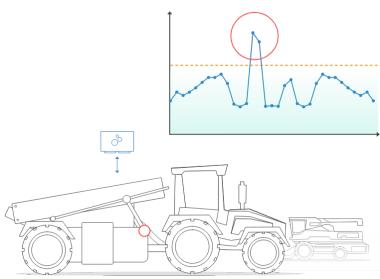


[open playground](#)

[get sample data](#)

Data logging use case examples

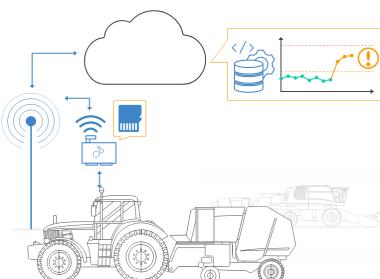
Here we outline a number of classic data logging use cases within agriculture vehicles.



Tractor black box data logger

Need to capture intermittent issues - or handle warranty disputes?

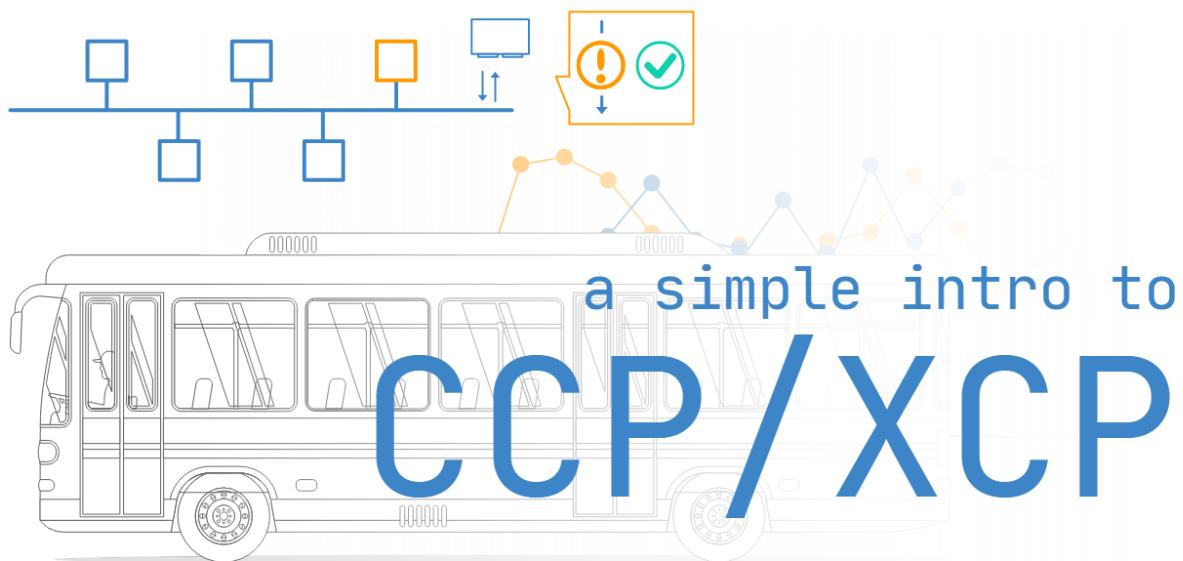
By deploying a CAN bus data logger permanently in your tractors, you will gain access to a rolling window of insight into the full scope of your CAN bus networks. This makes it an ideal solution for capturing rare issues to speed up diagnostics - or to handle cases of warranty disputes between the OEM and end user.



Monitoring operational agriculture data remotely

Need to remotely collect operational data for post analysis?

As an operator or systems integrator, you may need to collect data across a fleet of agriculture vehicles in the field. Here, the CANedge2 can be deployed to enable logging of the raw CAN data. The data can be uploaded when the vehicles return to a stationary WiFi network - or you can deploy each CANedge2 with a 3G/4G hotspot for continuous data collection from the field. Using the CANedge Python API, it's easy to perform big data processing, advanced statistical analyses and reports.



CCP / XCP on CAN Explained - A Simple Intro

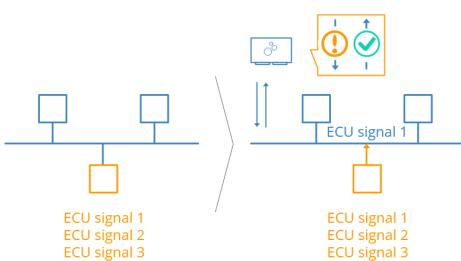
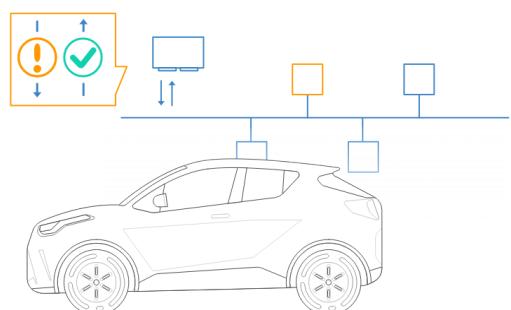
Need a simple intro to CCP/XCP on CAN bus?

In this practical tutorial, we introduce the basics of the CAN Calibration Protocol (CCP) and the Universal Measurement and Calibration Protocol (XCP) on CAN. In particular, we'll focus on the CCP/XCP frame structures, trace examples and A2L files. We also cover practical ECU data logging via polling/DAQ - and how to decode the data.

What is CCP/XCP?

The CAN Calibration Protocol (CCP) is an interface that enables read/write access to an Electronic Control Unit (ECU). It enables calibration, data measurement, flashing and more. The Universal Measurement and Calibration Protocol (XCP) is the successor to CCP with various improvements - including support for more transport layers such as Ethernet, FlexRay and SxL.

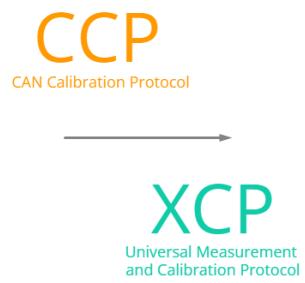
The CCP/XCP protocols have extensive overlaps, but also important differences. To avoid confusion, we will first focus on covering the CCP protocol - and subsequently go through XCP on CAN with explicit clarification on important differences.



To understand the motivation for CCP/XCP, let's revisit our [simple intro to CAN bus](#). As explained here, CAN enables communication of data between different ECUs in a vehicle/machine. Inputs and outputs of every ECU will be broadcast on the CAN bus. However, the inner workings of an ECU is a black box. Here, CCP/XCP provides direct access to the inner workings of an ECU. This lets you request high-frequency parameter data that may otherwise only be known to the ECU. Further, it also lets you modify the ECU algorithms and variables, making it easy to test and calibrate ECUs. Importantly, CCP/XCP enables these interfaces in a standardized way across ECU manufacturers.

History of CCP/XCP

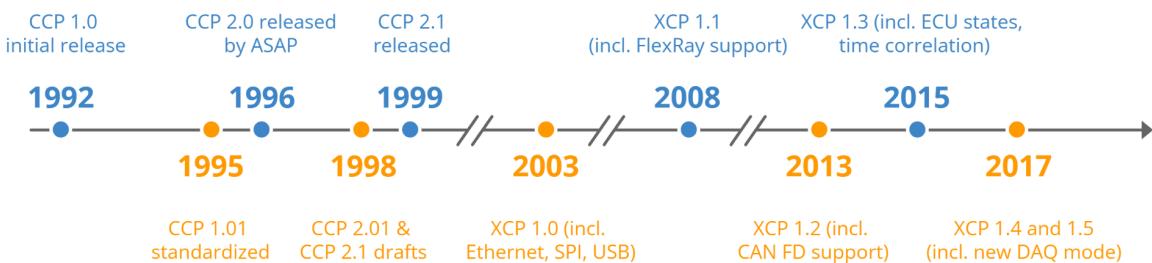
The CAN Calibration Protocol (CCP) was originally developed by a calibration systems company, Helmut Kleinknecht. Within a few years, it was improved by a working group, ASAP (Arbeitskreis zur Standardisierung von Applikationssystemen) that included Audi, BMW, VW and others. Later ASAP was renamed to [ASAM](#) (Association for Standardization of Automation and Measuring Systems).



Below are the key milestones:

- 1992: CCP 1.0 initial release by Helmut Kleinknecht
- 1995: CCP 1.01 standardized by ASAP
- 1996: CCP 2.0 released by ASAP
- 1998: Drafts of CCP 2.01 and CCP 2.1 were prepared
- 1999: CCP 2.1 was released in February
- 2003: XCP 1.0 incl. support for CAN, Ethernet, SPI, USB
- 2008: XCP 1.1 incl. support for FlexRay
- 2013: XCP 1.2 incl. ECU description file updates + CAN FD
- 2015: XCP 1.3 incl. ECU states, bypass handling, time correlation
- 2017: XCP 1.4 incl. Improvements and new DAQ mode
- 2017: XCP 1.5 incl. software debugging without a debug adapter

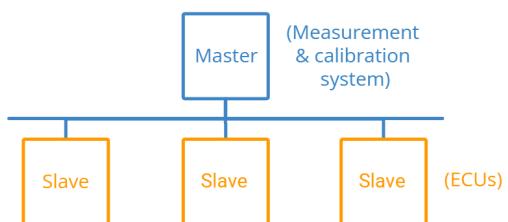
Today, XCP (aka ASAM MCD-1 XCP) is the successor to CCP (aka ASAM MCD-1 CCP). However, in practice many ECUs still use CCP, which makes it relevant to understand both protocols and key differences.



Master-slave architecture

The CCP/XCP protocol is based on a single-master/multi-slave concept. An external measurement & calibration tool (e.g. a PC/device/data logger) serves as the master and is able to read/write from one or more ECUs aka slaves.

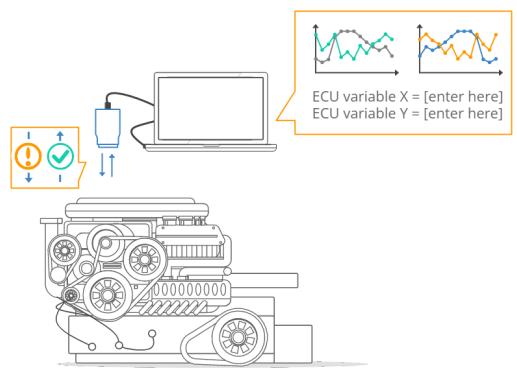
The interface between the master and slave is called ASAP1 or ASAM MCD-1. The CCP/XCP standard also describes the ASAP2 or ASAM MCD-2 MC interface between the master and an ECU description file. In practice, this database describes all relevant information about an ECU in a standardized file format called A2L (ECU Description Files) - which we will cover shortly.



CCP/XCP use cases

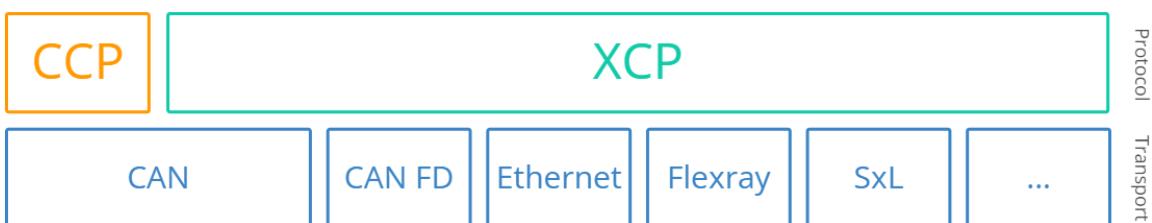
The CCP/XCP protocol enables multiple use cases:

- Plug & play ECU measurement and calibration
- Recording of ECU data at microsecond resolution
- Access to data internal to the ECU (not broadcast on CAN)
- Measurement via polling or based on events (time, triggers)
- Real-time calibration/adjustment of ECU algorithm variables
- Flash programming of ECUs
- Optional authentication for secure access
- Mainly for OEM development - rarely used post production



Major changes in XCP vs. CCP

- XCP adds support for CAN FD, Ethernet, FlexRay, SxL and more
- Less interpretation - more consistent implementations
- New 'stimulation' (STIM) mode for bypassing ECU calculations
- Predefined/dynamic DAQ lists for efficient communication
- Support for synchronous use of different DAQ modes
- ECU auto detection (master can poll slaves for information)
- More precise data acquisition by measuring ECU timestamps
- More data throughput via new optional commands



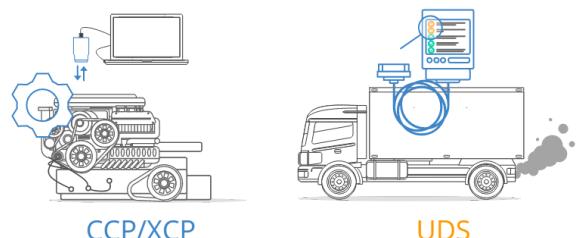
CCP/XCP vs. UDS

Before we deep-dive further on CCP/XCP, it can be useful to understand the role of these communication protocols vs. a slightly similar protocol, [Unified Diagnostic Services \(UDS\)](#).

As evident from the illustration, CCP/XCP is designed specifically for pre-launch measurement and calibration by the OEMs. Typically, the CCP/XCP access to ECUs is disabled once vehicles are ready for launch. In contrast, UDS is typically not available in early stage prototype development and only later added - also being available for communication post launch.

UDS focuses on diagnostics, whereas the diagnostics capabilities of CCP/XCP are light-weight. UDS may also be used by e.g. field technicians and in OEM specific scan tools.

The two protocols do share similarities, though - e.g. in their support for polling/cyclic data acquisition, ECU flashing and read/write access by address (CCP/XCP) or ID (UDS).



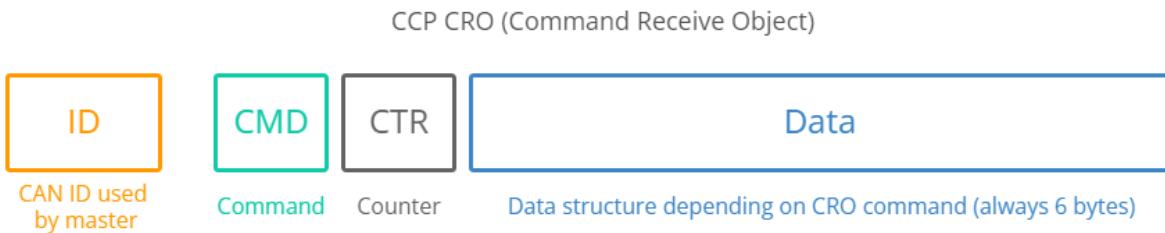
- | | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> - Mainly used by OEMs pre-launch - Measurement & calibration - Read/write by ECU address - Efficient & advanced DAQ - OEM specific/proprietary - A2L files are core | <ul style="list-style-type: none"> - Also used in field - Diagnostics/field service - Read/write by identifier - Diagnostic trouble codes - OEM specific/proprietary - Basis for e.g. WWH-OBD |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

CCP message types

To understand how CCP communication works we will first review the CCP message types. Overall, CCP communication is done through a request/response logic - similar to what is used in [OBD2](#), [UDS](#) etc. This communication consists of two types of messages: The Command Receive Object (CRO) and the Data Transmission Object (DTO).

CRO - Command Receive Object

The Command Receive Object (CRO) is a CAN frame sent by the master to an ECU. The 1st byte of the data payload is the command byte (CMD). This controls what command is being issued by the master to the ECU as per the table overview.



To track the commands issued, the 2nd byte is a counter (CTR). It is incremented by +1 for every command sent by the master and is mirrored in the response from the slave ECU. The remaining 6 bytes depend on the command.

Overview of CRO CMD values

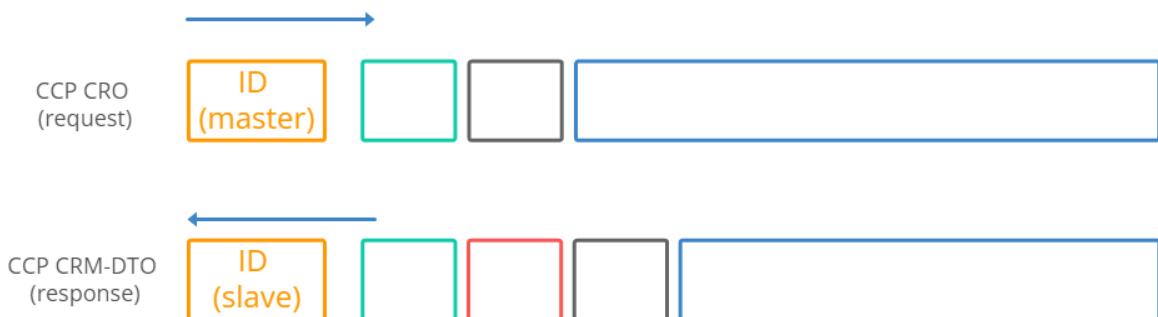
For a list of CRO commands used in CCP, see below table.

CCP CRO (Command Receive Object) - CMD (Command) table

CMD code	Command	Type	Optional?
0x01	CONNECT	Initialization	
0x1B	GET_CCP_VERSION	Initialization	
0x17	EXCHANGE_ID	Initialization	
0x12	GET_SEED	Authorization	Optional
0x13	UNLOCK	Authorization	Optional
0x02	SET_MTA		
0x03	DNLOAD		
0x23	DNLOAD_6		Optional
0x04	UPLOAD		
0x0F	SHORT_UP	Polling measurement	Optional
0x11	SELECT_CAL_PAGE		Optional
0x14	GET_DAQ_SIZE	DAQ measurement	
0x15	SET_DAQ_PTR	DAQ measurement	
0x16	WRITE_DAQ	DAQ measurement	
0x06	START_STOP	DAQ measurement	
0x07	DISCONNECT	DAQ measurement	
0x08	START_STOP_ALL	DAQ measurement	Optional
0x0C	SET_S_STATUS		Optional
0x0D	GET_S_STATUS		Optional
0x0E	BUILD_CHECKSUM		Optional
0x10	CLEAR_MEMORY		Optional
0x18	PROGRAM		Optional
0x22	PROGRAM_6		Optional
0x19	MOVE		Optional
0x05	TEST		Optional
0x09	GET_ACTIVE_CAL_PAGE		Optional
0x20	DIAG_SERVICE		Optional
0x21	ACTION_SERVICE		Optional

CCP CAN frame identifiers

In CCP, two CAN identifiers are used: One for messages sent by the master (e.g. 0x701) and one for messages sent by the slave (e.g. 0x702). These identifiers will be specified as part of the A2L (aka ECU Description File), meaning that the master will acquire this information prior to initiating the connection.



Typically low priority IDs are used for CCP to avoid disturbing safety critical info on the bus.

Importantly, this means that the master does not target specific ECUs through various CAN IDs - but rather through a connection sequence as shown below.

Example: CCP CRO CONNECT message

Below is an example of a CRO used for initializing communication with a specific ECU:

Trace example: CCP initialization (CONNECT CMD)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	01 01 39 01 AA AA AA AA	master	CRO (CONNECT CMD)

Trace details

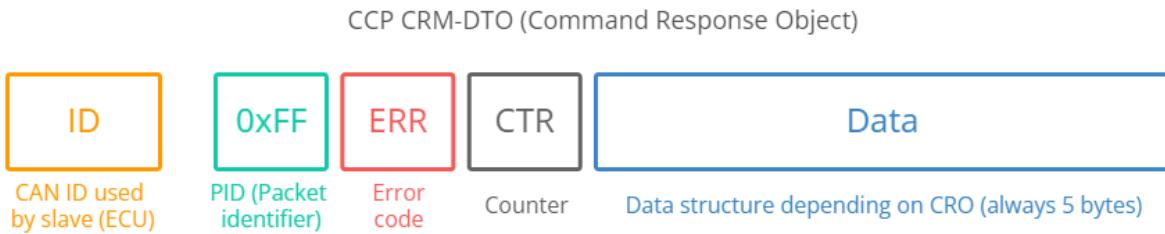
Here, the CAN ID reflects the ID used by the CCP master for communication. The 1st byte is 0x01, corresponding to the CONNECT command as evident from the previous table. The 2nd byte is the CTR value. The 3rd and 4th bytes are specific to the CONNECT command and correspond to the target ECU's station address in INTEL byte order. In other words, the above CRO is used by the master to establish a connection with ECU 0x0139. All subsequent communication will in this case be with this specific ECU until the master terminates the connection or connects to another ECU.

DTO - Data Transmission Object

The Data Transmission Object (DTO) is a CAN frame sent by the ECU to the master. Three types of DTOs exist as outlined below:

#1 CRM-DTO: Command Response Message

The CRM-DTO is sent by the ECU in response to a CRO from the master. Here, the CAN ID reflects the CAN ID used by the ECU (e.g. 0x702).



The data payload for the CRM can be broken down as follows:

- The 1st byte is the Packet Identifier (PID). For the CRM-DTO, the PID always equals 0xFF
- The 2nd byte is the error code (ERR), which can be used to e.g. inform the master of an invalid request.
- The 3rd byte is the counter (CTR), which will match the CTR value from the master's CRO
- The structure of the remaining 5 bytes depend on the original request made in the CRO

Below is the previous connection example trace, including a positive response CRM.

Trace example: CCP initialization (CONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	01 01 39 01 AA AA AA AA	master	CRO (CONNECT CMD)
1.04	702	FF 00 01 AA AA AA AA	slave	CRM (OK)

Trace details

The trace shows a connection to the ECU with station address 0x0139, which responds to the initialization command sent by the master. As evident, the 1st byte is 0xFF (as the message is a CRM). The 2nd byte shows that no errors occurred, while the 3rd byte matches the CTR of the CRO. With this sequence, the connection between the master and ECU 0x0139 is established.

Overview of CRM-DTO/EV-DTO ERR values

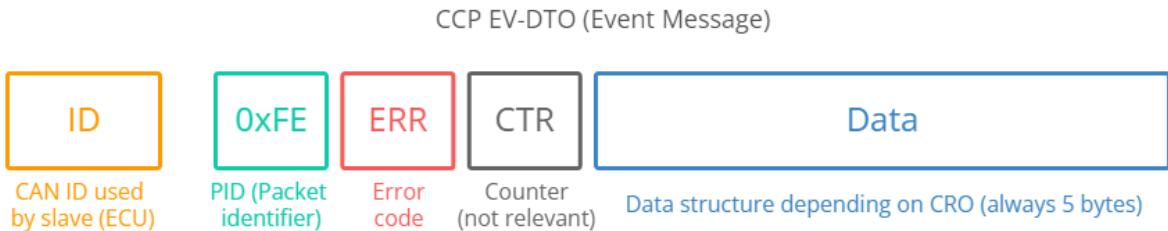
In the table below we show a set of CRM-DTO/EV-DTO error code values. Note that error codes can be sent in both CRM-DTOS (if they happen in direct response to a CRO from the master) or in EV-DTOS (if they happen asynchronously to the master's CRO commands).

CCP CRM-DTO/EV-DTO ERR code table

ERR code	Description	Error category	State transition
0x00	Acknowledge (no error)		
0x01	DAQ processor overload	C0	
0x10	Command processor busy	C1	None (wait)
0x11	DAQ processor busy	C1	None (wait)
0x12	Internal timeout	C1	None (wait)
0x18	Key request	C1	NONE
0x19	Session status request	C1	NONE
0x20	Cold start request	C2	COLD START
0x21	Cal. data init. request	C2	Cal. data init
0x22	DAQ list init. request	C2	DAQ list init
0x23	Code update request	C2	(COLD START)
0x30	Unknown command	C3	(FAULT)
0x31	Command syntax	C3	FAULT
0x32	Parameter(s) out of range	C3	FAULT
0x33	Access denied	C3	FAULT
0x34	Overload	C3	FAULT
0x35	Access locked	C3	FAULT
0x36	Resource/func. not available	C3	FAULT

#2 EV-DTO: Event Message

The EV-DTO is sent by the ECU in response to an internal event causing a status change in the ECU. This can be used to inform the master of errors that occurred since the last CRO.



The structure of the EV-DTO is identical to the CRM-DTO from before, with the EV PID always equal to 0xFE. Note that the CTR does not have relevance for EV-DTOS.

#3 DAQ-DTO: Data Acquisition Message

The DAQ-DTO is used by the ECU to automatically send data to the master in response to a specific event (e.g. a cyclic counter, a button or similar).

In DAQ communication, the master starts out by 'configuring' the ECU for a specific measurement sequence. Once initiated, the ECU will output DAQ-DTOS without further CROs from the master.

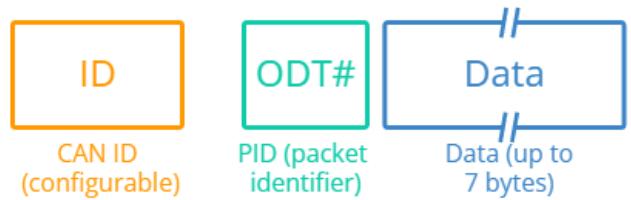
In the DAQ-DTO the PID is a reference to an 'Object Descriptor Table' (ODT), while up to 7 bytes carry data related to the ODT. More on this shortly.

Below is an example of a DAQ-DTO message sent by an ECU to a master.

Trace example: CCP DAQ-DTO frame with 5 byte signal payload

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
12.55	712	25 A1 42 34 AF 22	slave	DAQ-DTO

CCP DAQ-DTO (Data Acquisition Message)



Trace explained

The trace shows the DAQ-DTO communication from a slave following an initial configuration sequence. The data relates to ODT list #37 (0x25) with a payload of 5 bytes. Note how we do not pad the unused bytes (as this is not required for DAQ-DTOS). Note also how a new CAN ID is introduced for this DAQ response (more on this later).

General note on CCP messages

CRO, CRM-DTO and EV-DTO messages must have a payload length of 8. Any unused bytes are padded with arbitrary values (we use 0xAA in this intro). DAQ-DTO messages may use the actual payload size.

To recap: A master tool uses CRO messages to send various commands to a slave ECU. The slave ECU can in turn use DTO messages to respond to the master. Next we'll show how data measurement can be done via these messages.

How to record ECU data via CCP

Let's imagine that you're an engineer working at an automotive OEM. You've been tasked to extract the value of a specific parameter from an ECU over an extended period.

How do you do it? In the following sections we outline two methods: **Polling** and **DAQ**.

Data acquisition via CCP polling

The simplest solution would be to use a CCP measurement concept called polling.

Below we illustrate how such a communication flow may look:

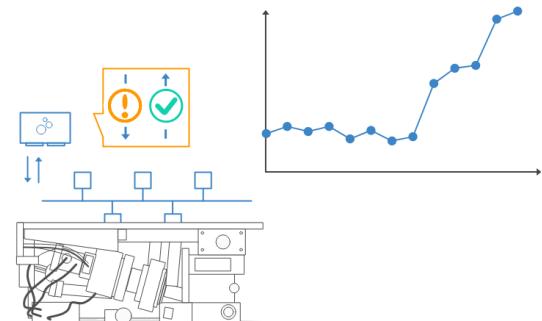
Trace example: CCP polling via SHORT_UP command

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)

Here, the master sends a request to the slave for 0x04 bytes of data stored at the 'source address' 0x12345678. The 4th byte is 0x00, which reflects the 'address extension' for this ECU source address, which can e.g. reflect a specific memory segment.

The slave responds with the 4 bytes of data, 0xF12A712F, for this particular source address. This above flow can be used to e.g. extract real-time parameter data, such as an RPM signal. It's somewhat similar to how you can request RPM via an OBD2 request/response flow in most cars.

Effectively, 'polling' is simply a sequence of CRO/CRM-DTOs with the master using a command called SHORT_UP (short upload).



Pros & cons of CCP polling

Polling is simple to set up - simply establish an ECU connection and send the correct request message every X ms. For many practical use cases this can be a perfectly viable solution. However, polling has two significant downsides.

#1 Polling is inefficient

Polling requires a request message for every response. If you e.g. need to measure a 2-byte signal at 100 Hz, that increases the busload by 200 frames/second. The inefficiency is both due to the need for requesting every single response - but also the fact that response messages are forced to be 8 bytes long, with 3 bytes spent on overhead.

#2 Polling is asynchronous

When polling for multiple signals, the request/responses need to be sent sequentially with a small delay between each request. As a result, the signal observations are not time synchronized. This makes data analysis and post processing more difficult and less precise.

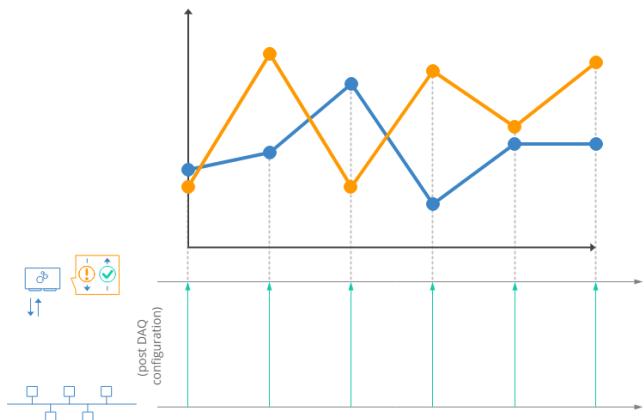


Data acquisition via CCP DAQ

CCP offers an alternative data measurement technique called DAQ (Synchronous Data Acquisition). DAQ is a bit more complex to initiate, but solves both the downsides of polling.

In simple terms, DAQ works as follows:

The master specifies what measurements to record from the slave, as well as what event should trigger the communication of data. For example, the master could request that signals A and B are broadcast every 10 ms, while signal C may be broadcast every 100 ms. The master can even configure slaves to broadcast signals based on events such as a button push or angle change. Once the DAQ configuration is complete, the master 'starts' the sequence and the slave now autonomously broadcasts the requested signals using DAQ-DTO messages.



This eliminates the request messages from the master. In addition, the signal data is packaged more efficiently in the DAQ-DTOs vs. the CRM-DTOs, reducing the busload. Further, with DAQ it is possible to package related signals in the same DAQ-DTO frames - ensuring that the requested signals are measured in a time synchronous manner - improving the quality of the data for analysis.

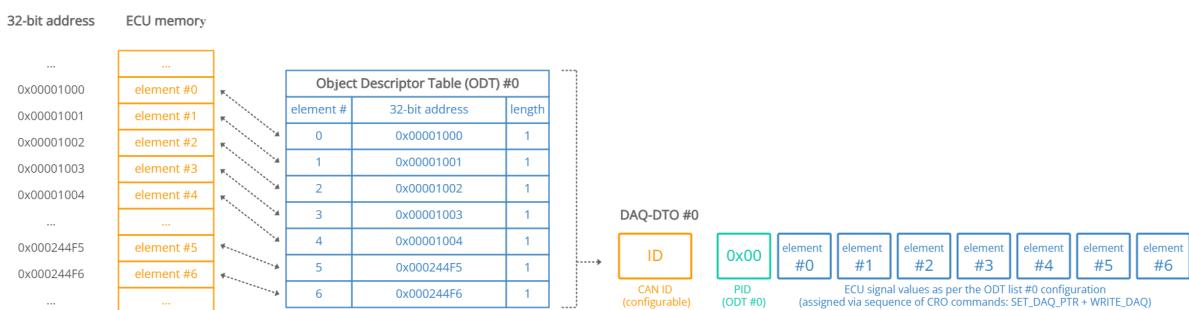
How to configure a DAQ sequence

In order to use the DAQ mode for data measurement, the master has to first configure the ECU accordingly. In practice, this process is typically handled by sophisticated GUI tools (more on this later). However, to fully understand what is going on we'll review what happens 'under the hood'.

Two concepts are key in DAQ: **Object Descriptor Tables (ODT)** and **DAQ lists**.

Object Descriptor Table

The Object Descriptor Table (ODT) is a list of references to data elements from the ECU memory. If we take the 'short upload' example used in CCP polling, the master sends a request specifying three elements: The data length (in bytes), address extension and source address. With this info, the ECU finds the data and sends it to the master. The concept of an ODT is similar: An ODT is simply a list of element references. Each entry in the ODT reflects an ECU source address - and optionally also an address extension and data length. In other words, an ODT entry contains the same info that we used to poll a signal.



ODT example & details

Let's take an example:

We wish to record 7 signals, each of them with a length of 1 byte. To do so, we define a new ODT #0 (PID 0x00) with 7 element entries. Element 1 refers to signal 1 with a specific ECU source address. Element 2 refers to signal 2 with another source address etc.

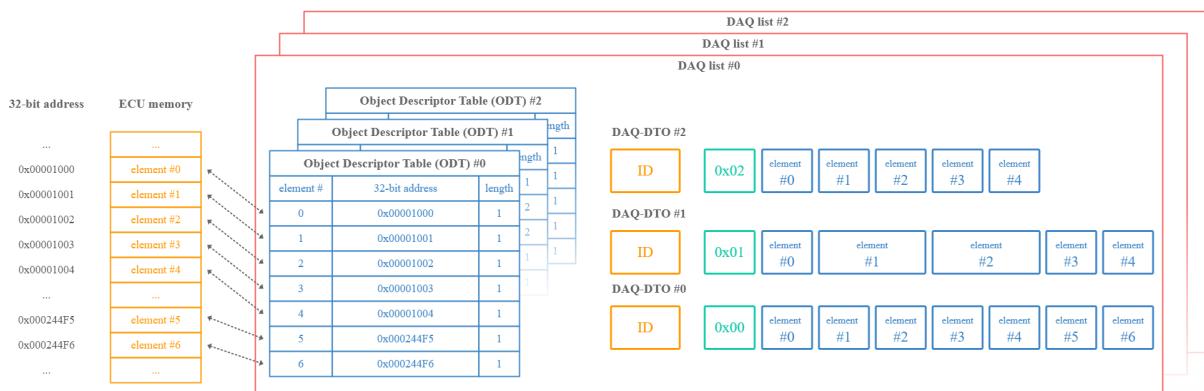
When we use the DAQ mode for data measurement, we can now refer to ODT #0 in order to get the ECU to provide us with time synced data on all 7 signals - within a single DAQ-DTO CAN frame. In other words: The ODT we defined describes the structure of a DAQ-DTO message, meaning that the ECU will now know how to package the 7 signal bytes in a single CAN frame - and the master knows how to extract the 7 signal bytes from that CAN frame.

Notice the impact on the busload: If we were to poll all 7 signals, it would require 14 CRO/CRM-DTO CAN frames per cycle - now it only requires 1 DAQ-DTO CAN frame.

Further, we can easily plot these 7 signals together as they share the same CAN frame timestamp - making it much easier to perform analysis. For the same reason, ODT lists are typically defined so as to group related signals together. In practice, a master will often define multiple ODT lists and assign PIDs for each of them in the range of 0x00 to 0xFD. Each ODT then defines the structure of a separate DAQ-DTO.

DAQ lists

When working with multiple ODT lists, it's useful to group them together. These groups of ODT lists are referred to as DAQ lists. For example DAQ list #0 may contain ODT #0, ODT #1 and ODT #2.



A DAQ list is characterized by the way in which the data is sampled:

- DAQ list #0 may be set up so that all DAQ-DTOs within it are sampled every 10 ms
- DAQ list #1 might be configured so that DAQ-DTOs in it are sampled when a button is pressed
- ...

In other words, a separate DAQ list is required for each unique sampling logic requested for the data measurement. We will describe the DAQ and ODT initialization shortly, but first consider below example trace for an already initiated DAQ sequence:

Trace example: CCP DAQ messages (multiple DAQ and ODT lists)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
2.44	712	00 AF 32 4A 00 00 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.44	712	01 22 01 00 A0 BB 09 F1	slave	DAQ-DTO (DAQ #1 ODT #1)
2.44	712	02 01 00 FF FF FF 1A 20	slave	DAQ-DTO (DAQ #1 ODT #2)
2.45	712	00 AF 32 4B 00 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.46	712	00 AF 32 4B 01 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.47	712	00 AF 33 4A 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
...
2.54	712	00 AF 33 4C 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.54	712	01 23 01 11 A1 BC 08 F4	slave	DAQ-DTO (DAQ #1 ODT #1)
2.54	712	02 02 01 FE FE FF 1B 20	slave	DAQ-DTO (DAQ #1 ODT #2)
2.55	712	00 AF 34 4C 02 0C 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.56	712	00 AF 34 4D 02 0C 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.57	712	00 AF 34 4E 02 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
...
2.64	712	00 AF 31 4D 01 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.64	712	01 24 05 01 A0 BC 09 F6	slave	DAQ-DTO (DAQ #1 ODT #1)
2.64	712	02 01 00 FF FF FF 1C 21	slave	DAQ-DTO (DAQ #1 ODT #2)
2.65	712	00 AF 31 4D 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
...

Trace explained

In this example, the target ECU is broadcasting data across three ODT lists in total, as evident from the 1st bytes spanning from 0x00 to 0x02. While it's not explicitly clear from the trace, the three ODTs are split into two DAQ lists: One DAQ list contains ODT #0 with a sampling frequency of 10 ms - while DAQ list #1 contains the remaining ODT lists #1 to #2 with a sampling frequency of 100 ms. This is why the DAQ-DTO with PID 0x00 is observed more frequently in the trace vs. the other DAQ-DTOs.

How to initialize a DAQ sequence

We've now looked at both the ODT and DAQ lists - but how do we define these over CAN?

In simple terms, the configuration of DAQ lists is done through a (potentially long) sequence of CRO/CRM-DTO frames. Here, the master essentially specifies the entire DAQ/ODT/element structure element-by-element. To define a new signal (aka element), the master specifies the address, length and address extension. Next, the master informs the ECU how to package the element in DAQ-DTOs by linking the element to an ODT# and DAQ#.

To understand this in detail, see below trace example and subsequent explanation:

Trace example: CCP DAQ messages (multiple DAQ and ODT lists)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
2.44	712	00 AF 32 4A 00 00 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.44	712	01 22 01 00 A0 BB 09 F1	slave	DAQ-DTO (DAQ #1 ODT #1)
2.44	712	02 01 00 FF FF FF 1A 20	slave	DAQ-DTO (DAQ #1 ODT #2)
2.45	712	00 AF 32 4B 00 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.46	712	00 AF 32 4B 01 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.47	712	00 AF 33 4A 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
...
2.54	712	00 AF 33 4C 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.54	712	01 23 01 11 A1 BC 08 F4	slave	DAQ-DTO (DAQ #1 ODT #1)
2.54	712	02 02 01 FE FE FF 1B 20	slave	DAQ-DTO (DAQ #1 ODT #2)
2.55	712	00 AF 34 4C 02 0C 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.56	712	00 AF 34 4D 02 0C 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.57	712	00 AF 34 4E 02 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
...
2.64	712	00 AF 31 4D 01 0A 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)
2.64	712	01 24 05 01 A0 BC 09 F6	slave	DAQ-DTO (DAQ #1 ODT #1)
2.64	712	02 01 00 FF FF FF 1C 21	slave	DAQ-DTO (DAQ #1 ODT #2)
2.65	712	00 AF 31 4D 01 0B 00 F1	slave	DAQ-DTO (DAQ #0 ODT #0)

Trace explained

So we have quite a bit going on here, but let's break it down:

We start by using the CRO command GET_DAQ_SIZE. This has two purposes: It informs us about the size of the specified DAQ list in terms of ODT lists - and it clears the current list. Essentially, this works like a 'reset' command for the DAQ list, in this case DAQ list #5.

As part of the GET_DAQ_SIZE command we also provide the 11-bit CAN ID 0x712 in the last 4 bytes. This informs the ECU that it should use CAN ID 0x712 to broadcast the subsequent DAQ-DTOs. This is subtle, but an important aspect: Essentially, the master is able to specify the CAN ID for every DAQ list, which in turn also enables the master to initiate DAQ measurement across multiple ECUs in parallel.

The ECU confirms the CRO from the master and informs us that the DAQ list #5 has 3 ODTs with the first PID being 0x07. Next, the master starts populating DAQ list #5 and ODT list #7 with the source address information for 7 x 1-byte signals. This is done through a simple loop consisting of 7 repetitions of two commands: SET_DAQ_PTR and WRITE_DAQ.

First, the master uses SET_DAQ_PTR to "select" DAQ list #5, ODT list #7 and element #0 of the ODT list. Next, the master uses WRITE_DAQ to write the contents of this element reference: A length of 1, an address extension of 0 and a source address of 0x00001000.

Basically, we have now told the ECU to "package" the value of our 1st signal (which is stored in the specified source address) into the 2nd byte of the DAQ-DTO that corresponds to DAQ list #5 and ODT list #7 (the 1st byte of the DAQ-DTO being the PID 0x07).

After this, we repeat the process for the remaining 6 signals until the ODT #7 has been completed.

In this simplistic case we only care about this particular DAQ and ODT list. Therefore, the final step is to start the DAQ measurement via the START_STOP command. Here, we specify that we wish to start DAQ list #5 and ODT list #7. As part of this, we specify the timing parameters, namely that event channel 0x03 should be used with a prescaler of 10. The ECU specific event channel details are specified in the ECU description file.

As evident, the ECU will now start broadcasting data from DAQ list #5 and ODT list #7 at the specified frequency. The CAN frame payloads include the ODT PID 0x05 in the 1st byte and the 7 'element' signal values in the remaining payload.

Note also that you can alternatively use the START_STOP command to 'prepare' multiple DAQ lists and ODT lists for measurement and then use the START_STOP_ALL command to simultaneously start or stop all of them.

How to disconnect from an ECU

Once the data acquisition has completed, the master may disconnect from the ECU via the DISCONNECT command.

Let's look at an example trace:

Trace example: CCP disconnection (DISCONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
24.01	701	07 88 01 AA 02 07 AA AA	master	CRO (DISCONNECT)
24.04	702	FF 00 88 AA AA AA AA AA	slave	CRM (OK)

Trace explained

Here, the 3rd byte 0x01 means that we end the session entirely (in contrast to a temporary disconnection). This effectively resets the ECU including our previous configuration. To target the specific ECU of interest, we specify the station address (Intel format) of the ECU, i.e. 0x702. As evident, DAQ is more convoluted to set up - but enables efficient time synced communication of ECU data.

Decoding CCP signal data from ECUs

In CCP polling/DAQ, you are in practice recording raw CAN frames with specific signal encoding structures. In order to make sense of the recorded data, you need to decode it to human-readable form aka physical values. This is the same concept as we've explained in several other CAN-related intros, incl. our intro to CAN bus, [J1939](#), [OBD2](#) and [DBC files](#). However, CCP decoding involves some extra complexities that we will explain below.

Decoding CCP polling data

Let's review an extended version of the CCP polling request/response from before:

Trace example: CCP polling via SHORT_UP command (two signals)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)
1.55	701	0F 03 04 00 AB CD EF 00	master	CRO (SHORT_UP CMD)
1.57	702	FF 00 03 00 37 1A 81 AA	slave	CRM (OK + DATA)

This reflects a decoding challenge: The ECU sends different signals with the same CAN ID - with no way to identify them in the payload.

Detailed explanation

In the above example, the master is requesting two different 4-byte signal values from the ECU, namely from source addresses 0x12345678 and 0xABCDFOO. In both cases, the ECU responds with CAN ID 0x702.

In most CAN bus decoding scenarios, you would be able to simply lookup the response CAN ID and find the 'start bit' and 'bit length' of a given signal to extract the raw data from the payload.

However, that is not possible here as the ECU uses the same CAN ID across two different signals. In other words, the CAN ID is not sufficient to distinguish between the signal coming from 0x12345678 and the one coming from 0xABCDFOO.

To some extent, this is similar to how OBD2 responses from an ECU use the same CAN ID (typically 0x7E8) across different signals like speed and RPM. In the case of OBD2 we can, however, easily solve this because the OBD2 PID (Parameter Identifier) is included in the payload. Together with the CAN ID, this serves as a unique identifier within the response frame - enabling us to view the response as a case of multiplexing.

We cannot directly do this in CCP polling because the response payload does not include the ECU source address. In other words: In use cases with multiple CCP polling signals we need to combine information from the request & response message.

In simplistic terms, we can solve this by "repackaging" the payloads as follows:

Trace example: CCP polling - decoding via repackaging of payload

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)
1.55	701	0F 03 04 00 AB CD EF 00	master	CRO (SHORT_UP CMD)
1.57	702	FF 00 03 00 37 1A 81 AA	slave	CRM (OK + DATA)
1.53	702	0F 02 04 00 12 34 56 78 FF 00 02 F1 2A 71 2F AA		CRO + CRM
1.57	702	0F 03 04 00 AB CD EF 00 FF 00 03 00 37 1A 81 AA		CRO + CRM
		$\overbrace{\hspace{10em}}$ "multiplexor"		

By doing so, we can process this using multiplexing logic - similar to how we handle [OBD2](#) and [UDS decoding](#). Here, the CAN ID to look up is 0x702 and the multiplexor is found in the 5th to 8th byte, while the signal value is found in the 12th to 15th byte. Two separate decoding rules can now be specified, with the relevant one dependent on the value of the 4-byte multiplexor.

In practice such a reconstruction of CAN frames can be done in e.g. a [Python script](#), assuming the CAN trace includes both the request and response data. Once the CAN frames are reconstructed, it is possible to use DBC files for decoding the data via multiplexing. Alternatively, the data can be [loaded into a software tool](#) supporting CCP/XCP decoding directly.

Decoding CCP DAQ data

Let's briefly review a snippet of the post-initialization DAQ trace from before:

Trace example: CCP DAQ messages

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
2.44	712	07 AF 32 4A 00 00 00 F1	slave	DAQ-DTO
2.54	712	07 AB 22 4C 01 00 00 F1	slave	DAQ-DTO
2.64	712	07 AD 24 4B 02 0F 00 F2	slave	DAQ-DTO
2.74	712	07 AB 25 4C 03 0E 00 F1	slave	DAQ-DTO
...

As evident, the DAQ-DTO messages have a pre-specified CAN identifier (0x712 above) and a payload in which the 1st byte equals the ODT list identifier aka ODT PID (0x07 above). As a result, it is not necessary to "combine" the DAQ-DTO response frames to uniquely identify which frames contain which signals - this can directly be identified through a combination of the CAN ID and the ODT PID.

This fact makes the decoding of DAQ-DTO messages simpler than the polling messages. In fact, you can directly create a DBC file with multiplexing as long as you know the signal encoding in the remaining bytes of each ODT list.

Will the CCP DAQ decoding rules be known?

In most practical applications, it is reasonable to assume that you will know this signal encoding. This is because the master itself controls (via the initialization sequence) how to package each signal into the DAQ and ODT lists, as explained in the previous section.

For example, we know that the DAQ-DTOs with CAN ID 0x712 and ODT PID 0x07 contain the signal with source address 0x00001000 in the 2nd byte - because that is how we packaged it during the initialization. From the ECU description file we can then review how to interpret this 1-byte signal value. The description may state that this signal is a temperature measured in degC and it should be multiplied by a factor of 0.8 and offset by 20. If you're familiar with DBC files, you'll note that such information can easily be entered into a DBC, allowing for quick decoding of the trace data in most CAN software tools.

Naturally, if the master changes the initialization to package the data differently on another test run, the implication will be that the DBC file must be updated accordingly.

The above should make it clear that CCP polling/DAQ can be treated within the normal logic of CAN bus frame decoding - though both involve some tweaks. DBC files can be used to process the data - but this format is not the most common choice for working with CCP data. Below we explore another file format that is generally used, the A2L or ASAP2 format.

A2L - ECU Description Files

In the prior sections, we've occasionally referred to "ECU description files". An ECU description file contains everything a master tool needs to know to communicate with an ECU. From a data measurement perspective, this includes information on CAN identifiers, available signals, signal decoding rules, DAQ/ODT information etc.

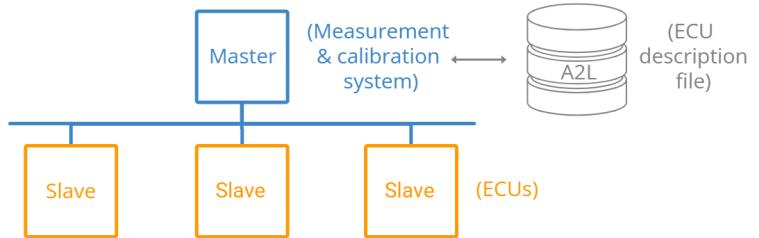
In practice, the ASAP2 description format (*.A2L) is used to structure this information. The ASAP2 data definition was standardized by ASAM in the ASAM MCD-2 MC, with the first version 1.3.1 being released June 15 1999, i.e. the same year that the CCP 2.1 standard was released. The latest version of the standard, 1.7.0, was released in 2015.

You can view an example A2L file [here](#).

In practice, the A2L file is used to "configure" the master device, enabling communication with the ECU. For example, in our previous examples we used the CAN IDs 0x701 and 0x702 for the CRO.DTO communication between the master and ECU. These IDs would be specified in the A2L file. The A2L file also describes how signals are stored in the ECU and how to decode them. See e.g. below snippet from the A2L example file:



```
/begin MEASUREMENT
/* Name */ Airflow
/* LongIdentifier */ "Final value after Linear Sensor Characterization"
/* Datatype */ FLOAT32_IEEE
/* Conversion */ CompuMethod_6
/* Resolution */ 1
/* Accuracy */ 0
/* LowerLimit */ -1E+17
/* UpperLimit */ 1E+17
ECU_ADDRESS 0x40000144
/end MEASUREMENT
```



A2L snippet explained

As before, we can compare this to the logic of signal encoding in other CAN protocols and in DBC files. The 'MEASUREMENT' tag is used to envelop the description of a measurement signal, in this case Airflow. It has a detailed description and a min/max defined by the lower/upper limit.

The signal also has a 4-byte source address, 0x40000144. If we recall the CCP polling outline, this source address could be used in a SHORT_UP command polling sequence to request the raw value for this particular ECU signal. Similarly, in the context of DAQ measurement, this source address would be used in the initialization process. In other words, an

engineer can lookup the signal of interest (Airflow) in a A2L file or A2L GUI tool - and configure his master tool to request this.

Assuming we've now recorded a trace of raw values, we'd need to decode the information. As explained before, the first step would be to extract the raw bytes from the response payload - and the method for doing this will depend on whether we're using polling or DAQ measurement. Further, the byte ordering is not specified by the CCP protocol - but will be specified within the A2L file at a global or signal level.

Once we've extracted the signal bytes and converted them to decimal form, we need to know how to convert them.

In the DBC file context, this would always be done in the form of a 'linear' equation, i.e. $y = bx + c$, where b equals a scale factor and c equals an offset - both specified in the DBC file for each signal.

In contrast to DBC files, the A2L files allow for more complex conversion methods. As evident from the signal description, it refers to a conversion rule defined as "CompuMethod_6". We can look this up in another section of the A2L file:



```
/begin COMPU_METHOD
/* Name           */ CompuMethod_6
/* LongIdentifier */ ""
/* ConversionType */ RAT_FUNC
/* Format         */ "%6.2"
/* Unit           */ "kg/s"
COEFFS 0 1 0 0 0 1
/end COMPU_METHOD
```

A2L snippet explained

As outlined, this conversion rule uses a conversion type called RAT_FUNC. This is one of multiple supported conversion types and is defined as below:

$$y = (ax^5 + bx^4 + cx^3 + dx^2 + ex + f) / (gx^5 + hx^4 + ix^3 + jx^2 + kx + l)$$

As evident, the rational function is a more sophisticated version of the linear function used in DBC files. In particular, it supports 5 coefficients, which need to be specified in the computation method. Above, they are set to 0 1 0 0 0 1, meaning the function simplifies to below:

$$y = x$$

In this simple case, the rational function simplifies to a linear function, similar to the one used in DBC files. A2L files in fact also support another conversion type, LINEAR, which is simply $y = ax + b$. In the case of the Airflow signal, this could have been used instead. Finally, the unit is also specified in the measurement details, meaning we can now plot the Airflow physical value as measured in kg/s. The Format field is %6.2, which should be read as %Length.Layout, with Length reflecting the overall length of the decoded signal and Layout reflecting the number of decimal places.

Practical use of A2L files

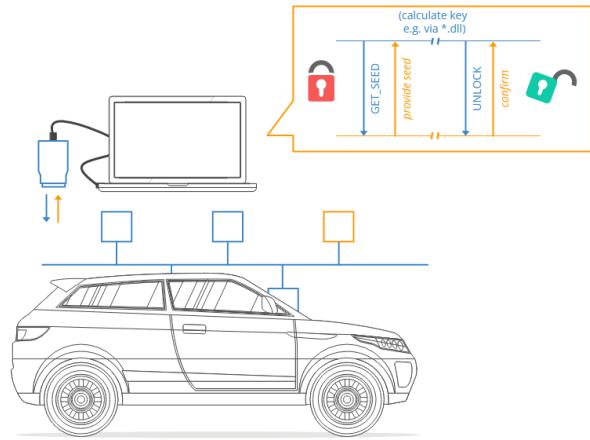
As is probably clear, one does not normally look through the A2L file via a text editor. Rather, the A2L file is simply loaded in a compatible tool (e.g. a PC GUI tool with a connected CAN interface, or a CAN data logger). This then allows the engineer to use the tool to perform the relevant measurement from the ECU without having to understand the A2L file structure and syntax.

This particular signal is simplistic enough to be described within the restrictive DBC file format - but A2L files clearly allow for more complex decoding rules. For more details on A2L file format, see the [ASAM MCD-2 MC standard](#), as well as a [practical overview](#).

Seed & key authorization

As explained previously, CCP communication is typically used by automotive OEM engineers to facilitate communication with ECUs in pre-production applications.

In some cases all it takes to communicate with an ECU is the A2L file and a suitable CAN tool. As evident, one can also simply extract a subset of the A2L file to configure a device e.g. for measurement of specific signals. However, without the A2L file, it's practically impossible to perform communication and hence many OEMs do not require any additional security measures beyond keeping the A2L file under lock.



However, some use cases warrant additional security. CCP supports this via a concept called 'seed & key' authorization. Here, the master requests a random seed from the ECU as part of the initialization of the ECU communication. The master receives the seed from the ECU and uses it as input for an internal security algorithm to calculate a key. The master then sends the key to the ECU via CAN - and if it matches the key calculated internally by the ECU, authorization is provided for further communication.

Below is an example trace for such a communication:

Trace example: CCP seed & key authorization sequence

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	12 05 02 AA AA AA AA AA	master	CRO (GET_SEED)
1.53	702	FF 00 05 01 14 15 16 17	slave	CRM (OK + SEED)
1.55	701	13 06 A3 12 FF B0 AA AA	master	CRO (UNLOCK + KEY)
1.57	702	FF 00 06 02 AA AA AA AA	slave	CRM (OK + RESOURCE MASK)

Trace explained

In this example, the master requests the seed via the GET_SEED command 0x12, specifying a 'resource mask' of 0x02 in the 3rd byte. The resource mask references the level of access requested - with 0x02 being DAQ access only (see the CCP 2.1 standard for details).

The ECU responds with a frame in which the 4th byte is 0x01, implying that DAQ access is protected (in contrast to 0x00 which would imply that authentication would not be required). The ECU provides the requested seed in the remaining 4 bytes. Based on the seed, the ECU calculates the key and provides it via the UNLOCK command, 0x13. The key is in this case 4 bytes long. The ECU responds positively including the requested resource mask 0x02, meaning that DAQ access is now unlocked.

Regarding security algorithm implementation

The most common implementation of the security algorithm is via a *.dll file, since the master device is frequently a Windows PC with a GUI tool and a CAN interface. The use of a *.dll file also enables standardization across CCP tools, while eliminating the need for the master tool to know the underlying algorithm used. It is, however, challenging to use

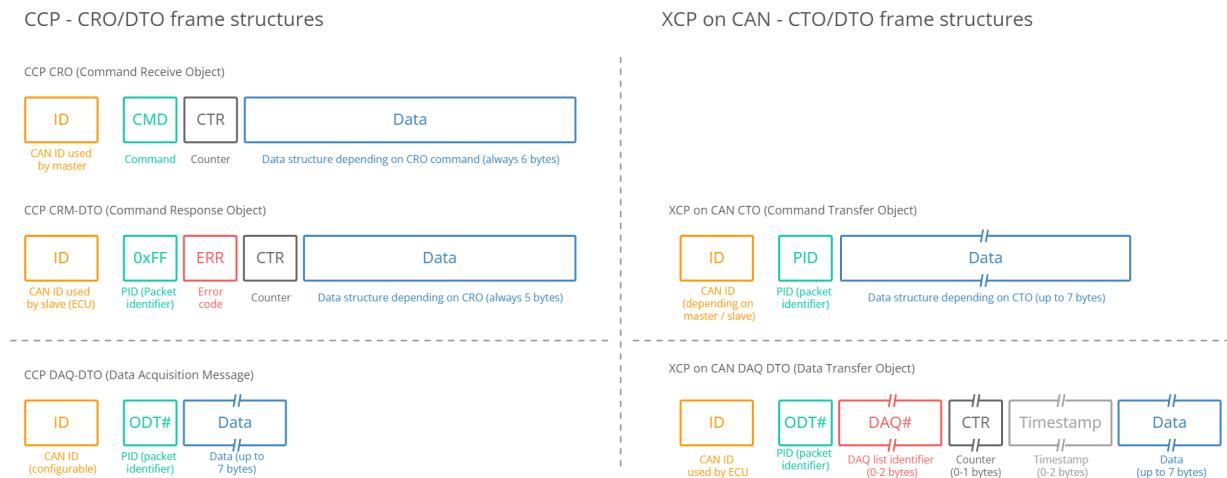
*.dll files in standalone CAN bus data loggers and telematics devices (as they do not run Windows), hence some companies use alternative solutions - e.g. proprietary file formats better suited for embedded devices.

XCP on CAN - the basics

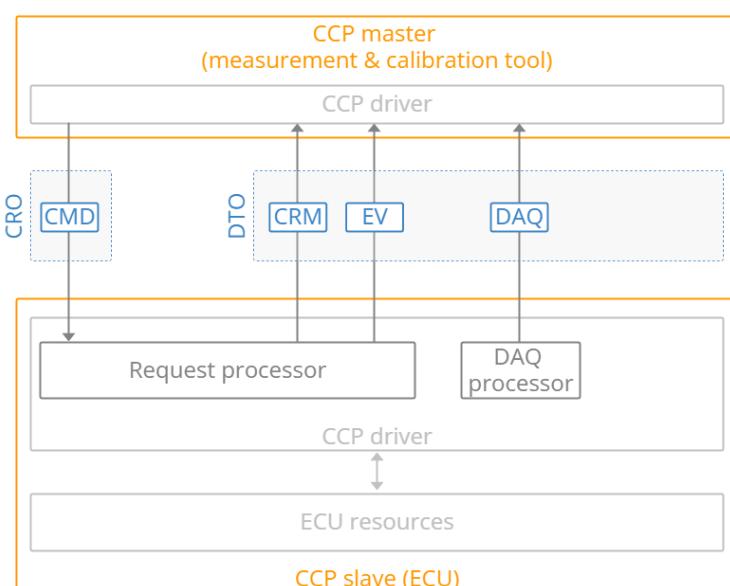
We have now gone through the basics of CCP. Next, we'll consider XCP on CAN with focus on frame structures.

The XCP packet

To understand the structural changes in XCP on CAN vs. CCP, consider below comparison of the CCP CRO/DTO and the XCP CTO/DTO. As evident, XCP packets include the **XCP CTO** (with a similar role as the CCP CRO and CRM-DTO) and the **XCP DTO** (with the XCP DAQ-DTO playing a similar role as the CCP DAQ-DTO).



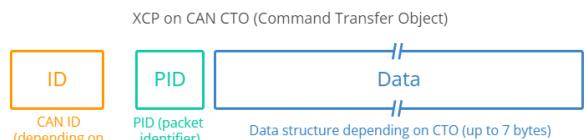
An alternative way to illustrate the role of the CCP vs. XCP messages is via below architecture comparison:



We will cover the XCP on CAN message types in more detail below.

XCP CTO - Command Transfer Object

In XCP on CAN, the CTO is a CAN frame for transferring control commands, incl. commands (CMD), command responses (RES), errors (ERR), events (EV) and service requests (SERV). In contrast to the CRO in CCP, the CTO is used by both the master and ECU. Note also that a CMD packet from the master must be answered with a RES or ERR packet from the ECU, while the other packet types are sent asynchronously.



If we look at the CTO payload, the 1st byte reflects a Packet Identifier (PID). The remaining 7 bytes in an XCP on CAN CTO payload consists of data, specific to the type of CTO packet. If we compare this structure to the CRO of CCP, it's similar, except for the exclusion of the command counter byte in XCP.

CAN bus identifiers in XCP

XCP communication requires at least two CAN bus identifiers: One for the master (e.g. 0x551) and one for the ECU (e.g. 0x552). If the master needs to communicate via XCP with more than one ECU, an additional set of identifiers will be required.

The actual values of the CMD byte across XCP and CCP differ as well. For example, CCP uses 0x01 for the CONNECT command, while XCP uses 0xFF. See also the table overview for the assignment of CTO PIDs depending on whether messages are sent from the master or slave.

XCP PIDs (master to slave)		XCP PIDs (slave to master)	
0xFF		0xFF	RES
0xFE		0xFE	ERR
0xFD		0xFD	EV
0xFC		0xFC	SERV
...			
0xC0			

See also below comparison of CCP vs. XCP CMD codes:

Select CCP vs. XCP CMD values

The below table compares a subset of the command codes between CCP and XCP:

CCP vs. XCP command code comparison (examples)

CCP CMD	XCP CMD	Command
0x01	0xFF	CONNECT
0x12	0xF8	GET_SEED
0x13	0xF7	UNLOCK
0x02	0xF6	SET_MTA
0x04	0xF5	UPLOAD
0x0F	0xF4	SHORT_UP
0x15	0xE2	SET_DAQ_PTR
0x16	0xE1	WRITE_DAQ
0x07	0xFE	DISCONNECT
0x0E	0xF3	BUILD_CHECKSUM

Example: XCP CTO CONNECT message

As in CCP, let's look at how a CONNECT sequence may look in XCP on CAN:

Trace example: XCP on CAN initialization (CONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	FF 00	master	CTO (CONNECT)
1.04	702	FF 04 80 08 08 00 01 01	slave	CTO (OK + INFO)

Trace explained

Note that the example does not use the 0xAA padding, as this is optional in XCP on CAN.

In the trace, the master sends a CONNECT CMD (0xFF) to the ECU with the communication mode set to normal (0x00). Note that, in contrast to CCP, the master does not need to specify a station address in the payload of the CONNECT frame. This is because the CAN identifiers already uniquely identify which ECU the master is communicating with.

The ECU responds with a CAN frame in which the 1st byte equals the positive response PID (RES). The 2nd byte is the resource availability, similar to the seed & key CCP communication with 0x04 e.g. meaning that DAQ is available. The 3rd byte relates to the 'communication mode' (here 'optional'), while the 4th byte is the maximum CTO size (here 8 bytes). The 5th and 6th byte equal the maximum DTO size (here 8 bytes), while the 7th and 8th bytes equal the XCP protocol layer version and transport layer version respectively (both 1 in this case).

Example: XCP Polling

With a connection established, the master can e.g. initiate polling. As in CCP, this can be done via a SHORT_UPLOAD command:

Trace example: XCP on CAN polling via SHORT_UPLOAD command

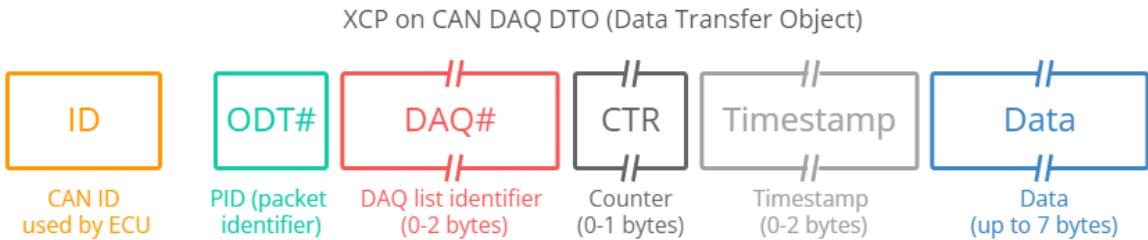
Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	551	F4 02 00 00 78 56 34 12	master	CTO (SHORT_UPLOAD)
1.53	552	FF F1 2A	slave	CTO (OK + DATA)

Trace explained

Here, the master sends a SHORT_UPLOAD CMD (PID 0xF4) for 2 bytes of data. The 3rd byte is reserved, while the 4th byte of the CTO is the address extension (in this case 0). The remaining 4 bytes equal the source address 0x12345678 (INTEL byte order). In response to the command CTO, the ECU sends a response CTO with the value of the 2 bytes of data.

XCP DTO: Data Transfer Object

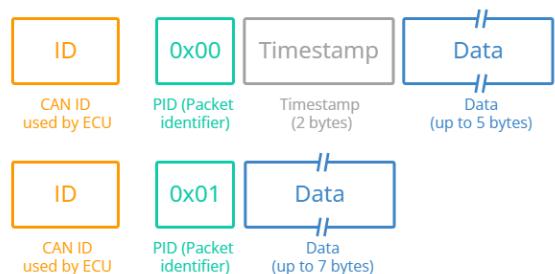
The XCP DTO is used for sending synchronous data. In particular, the DTO is used in DAQ measurement (similar to the role of the DAQ-DTO in CCP). In XCP, it also enables the transfer of 'stimulation' (STIM) data that can be used in bypassing the normal algorithm within an ECU. Stimulation and bypassing are topics that we will not cover here.



We will also not go into detail on the XCP DAQ measurement as it's similar in concept to CCP DAQ.

The XCP DTO timestamp

One important thing to note about XCP DAQ measurement, however, is that it enables the ECU to write the measurement time in the XCP DTO packets. The implication is that the master is able to correctly sync ECU data split across multiple frames by utilising the ECU measurement timestamp, rather than the master's own internal timestamp. The XCP DTO timestamp is optional and is implemented as an incrementing counter, with the incrementation logic specified in the A2L file. If a timestamp is to be included for a specific DAQ list, it will be written into the 1st ODT list (but not subsequent ones if more ODT lists exist within the same DAQ list).



DAQ/ODT identification

It is also worth noting that multiple methods exist for packing the DAQ list # and ODT list # information in the DAQ DTOs. The simplest case is 'absolute ODT numbers', where the ODT list # is unique across all DAQ lists. Here, the PID (i.e. ODT #) is sufficient for globally identifying an ODT list. Another method exists, however, where relative ODT list numbers are used across DAQ lists. Thus, you could e.g. have two ODT lists with PID = 0x00, meaning that they cannot be uniquely identified as the CAN ID is identical across the two DTOs. Here, a 1 or 2 byte DAQ list identifier can be added in the 2nd to 3rd bytes of the CAN frame payload. With this, one can again uniquely identify a specific ODT list. The A2L file and ECU will provide information on what method is used.

Optional XCP DTO counter field

A 1-byte counter field may optionally be used in the communication of XCP DTOs. If a CTR field is used in XCP DTO DAQ packets, the slave will only insert the value of the CTR in the 1st frame of a DAQ list, i.e. in the 1st ODT list.

Links for further reading on CCP/XCP

CCP/XCP of course contain numerous more topics that we have not taken the time to discuss here. Deliberately, we've focused on the basic topics and concepts related to data logging. However, below you'll find useful links for further reading.

- [Vector's XCP Book v1.5](#): A detailed overview of the XCP 1.5 protocol
- [XCP \(Wikipedia\)](#): The basics on the XCP protocol
- [ASAM MCD-1 XCP overview](#): Also provides a detailed overview of the XCP protocol

- [ASAM ASAM MCD-2 MC overview](#): Provides an overview of the ECU description file standard
- [Seed & key details \(Vector\)](#): A useful intro to the basics on authorization

Using the CANedge for CCP/XCP data acquisition

The [CANedge](#) is a series of low cost, compact 2 x CAN/LIN bus data loggers. The CANedge is commonly used by automotive OEMs - and for this reason a frequently asked question is whether the device supports CCP/XCP on CAN communication. The answer to this is 'yes, depending on your use case'. For details, see below. If you wish to use the CANedge for CCP/XCP on CAN, contact us.



CANedge + CCP/XCP

As we've seen in the previous sections, all it takes to facilitate CCP/XCP polling and CCP/XCP DAQ measurement is to be able to transmit a specific sequence of CAN frames, based on information that can be identified from an A2L file. As long as you know how to construct the request frames, you can use the CANedge transmit functionality to set up a list of single-shot or periodic custom CAN frames to e.g. achieve the CCP/XCP initialization and CCP/XCP polling/DAQ measurement. The recorded [MF4](#) log files with measurement data can then be processed in e.g. Vector tools or via our free open source software solutions. In the former case, you could e.g. use our [MF4 converters](#) to get the data into Vector tools and use A2L files for decoding the data. In the latter case, you could process the data via our [Python API](#) (for polling) and/or via [asammfdf](#). These tools would currently require creating a DBC file with the decoding rules as explained previously.

In other words, you can use the CANedge to measure signal data via CCP/XCP on CAN in standalone field deployments, which offers a low cost method for collecting this information at scale. In contrast, standalone CAN bus data loggers are not ideally suited for CCP/XCP calibration, except for very specific use cases such as remote calibration of variables via over-the-air updates.

CANedge seed & key authentication

The CANedge does not currently support seed & key authentication, so OEMs intending to use it for CCP communication would need to temporarily disable the authentication or to unlock it using a separate device. We are, however, potentially going to add support for this later on. If you have use cases that involve this functionality, feel free to contact us for further discussion.

CCP/XCP data logging - applications

In this section we provide brief examples of how CCP/XCP data logging may be useful in practice.

CCP/XCP telematics for prototype vehicle fleet

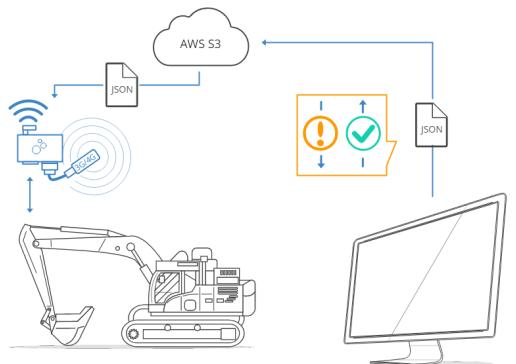
As an OEM, you may need to collect e.g. CAN, CAN FD or LIN bus data from prototype vehicles in the field. In addition, you may need to collect specific data internal to an ECU, which can only be measured via CCP or XCP on CAN. Here, a CAN logger like the CANedge can be configured with a



custom 'transmit list' consisting of single shot and/or periodic CAN frames. When deployed in the vehicle, the CANedge will then automatically transmit the pre-defined frames, allowing it to e.g. perform CCP/XCP polling of ECU data - or initialize DAQ measurement. The collected CCP/XCP data can be combined with the general CAN data in MF4 log files - and sent via WiFi/3G/4G to the OEM's own cloud server.

Remote CCP/XCP calibration via OTA updates

As a more sophisticated use case, an OEM could utilize a [CANedge2](#) to perform over-the-air updates to an ECU. For example, a CANedge2 may be deployed in a prototype vehicle in which an OEM engineer would like to re-calibrate certain variables in the ECU. To achieve this, the engineer constructs the relevant sequence of single-shot CAN frames in a new configuration file for the CANedge. Next, the engineer deploys the new configuration file on the CANedge2 S3 server, thus triggering an OTA update - allowing the CANedge2 to transmit the sequence into the vehicle remotely.



Thank you for reading our guide - we hope you found it useful!

CSS Electronics | DK36711949 | Soeren Frichs Vej 38K, 8230 Aabyhoej, Denmark

www.csselectronics.com | contact@csselectronics.com | +45 91 25 25 63 | [LinkedIn](#)

[Products](#) | [Software](#) | [Guides](#) | [Case studies](#)

