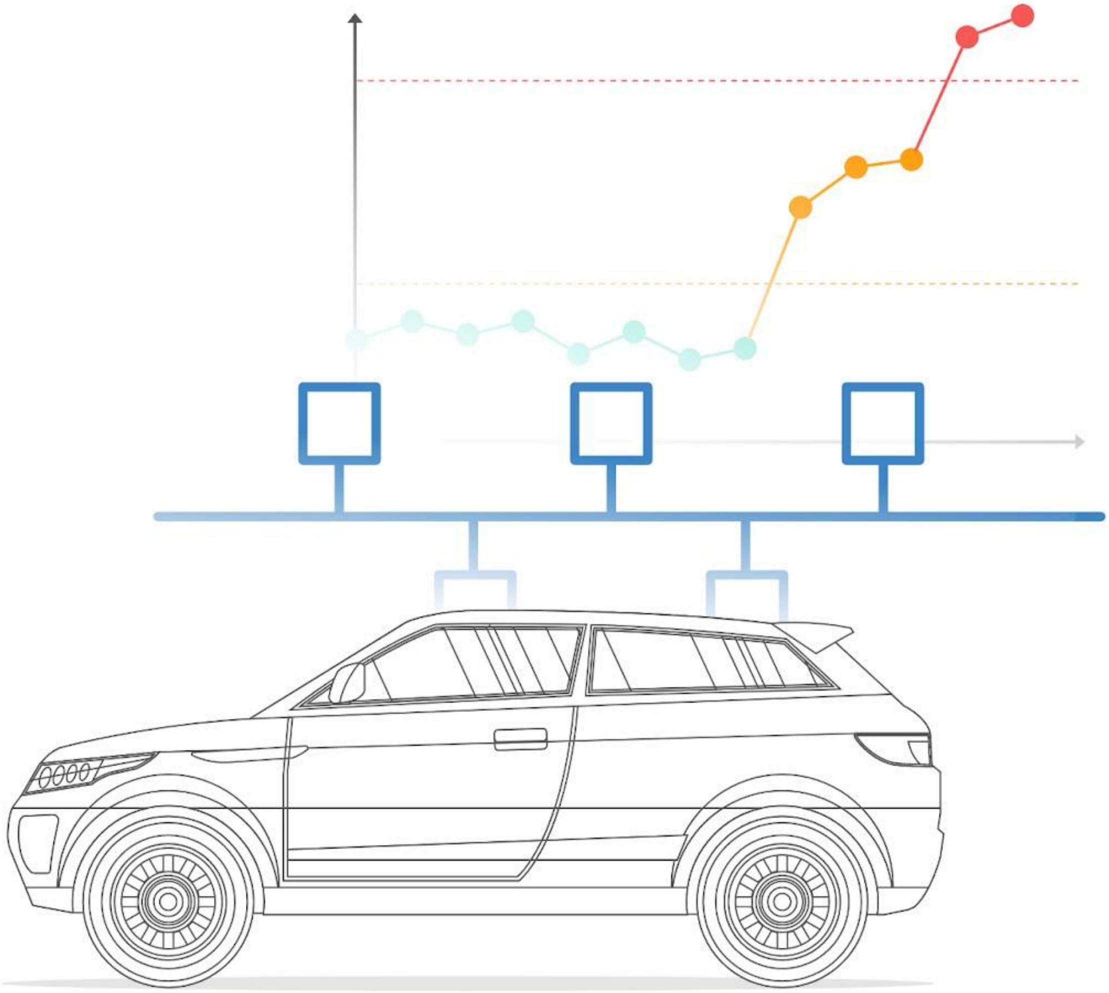
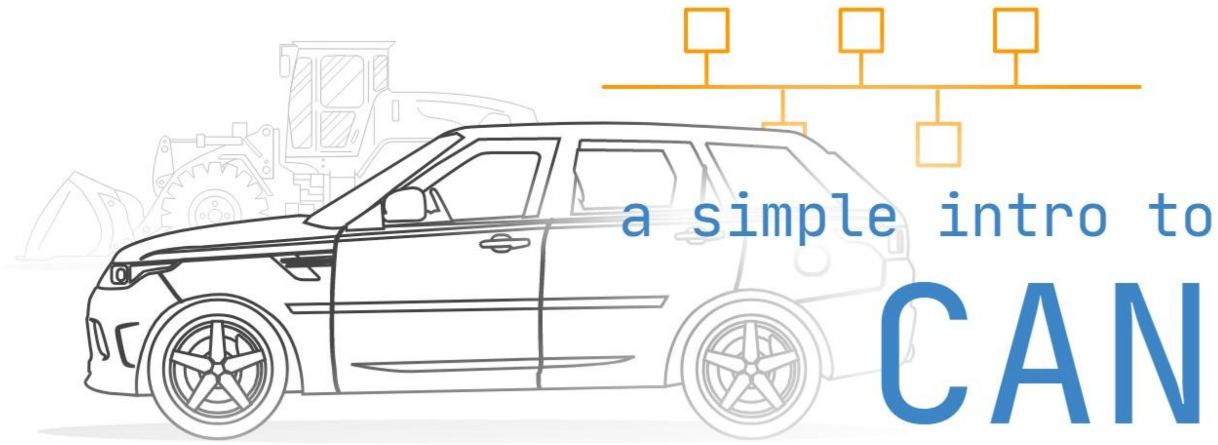


CAN bus

the ultimate guide

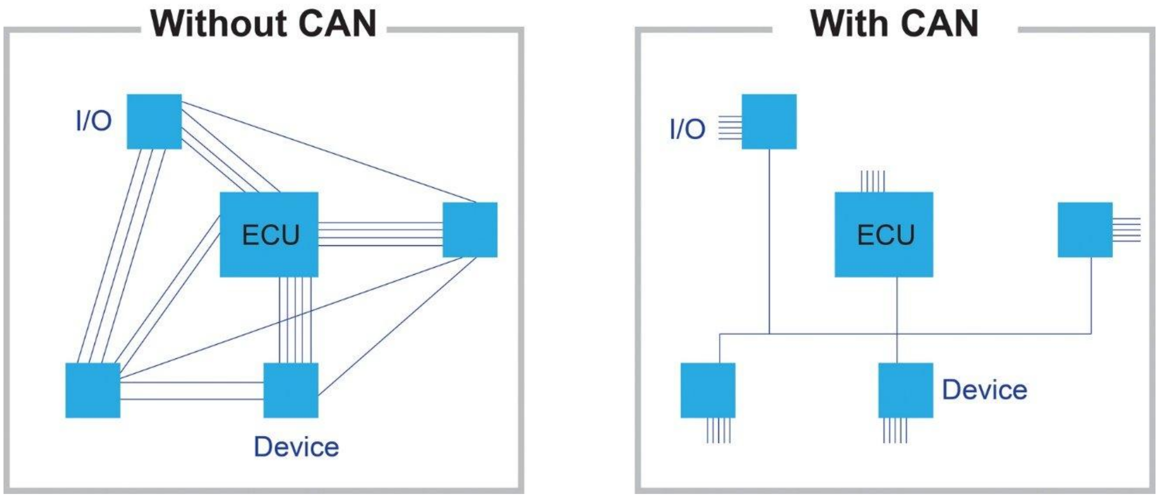
Contents



CAN Protocol Basic

What is CAN Protocol?

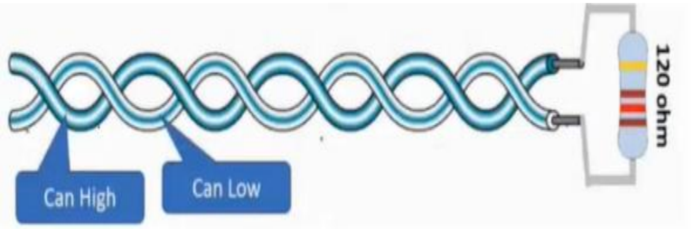
- A Controller Area Network (CAN) refers to a network of independent controller.
- It is a serial communications protocol that efficiently support distributed real-time control with a very high level of security.
- Reduces wiring complexity and cost.
- Ensures all devices receive messages.



CAN bus

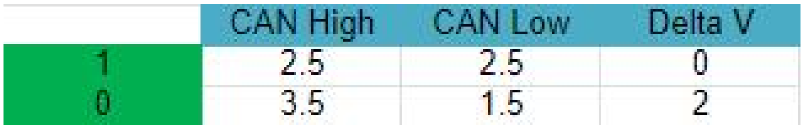
- CAN bus is a Dual twisted pair. The wires are twisted because the signals transmitted on the wires are made from measurement on both, therefore when the wires are twisted together they are both subject to the same interference and the change of discrepancy is greatly reduced.

- Combines with impedance 120 Ohm to prevent signal reflection, prevent data corruption and maintaining signal clarity.

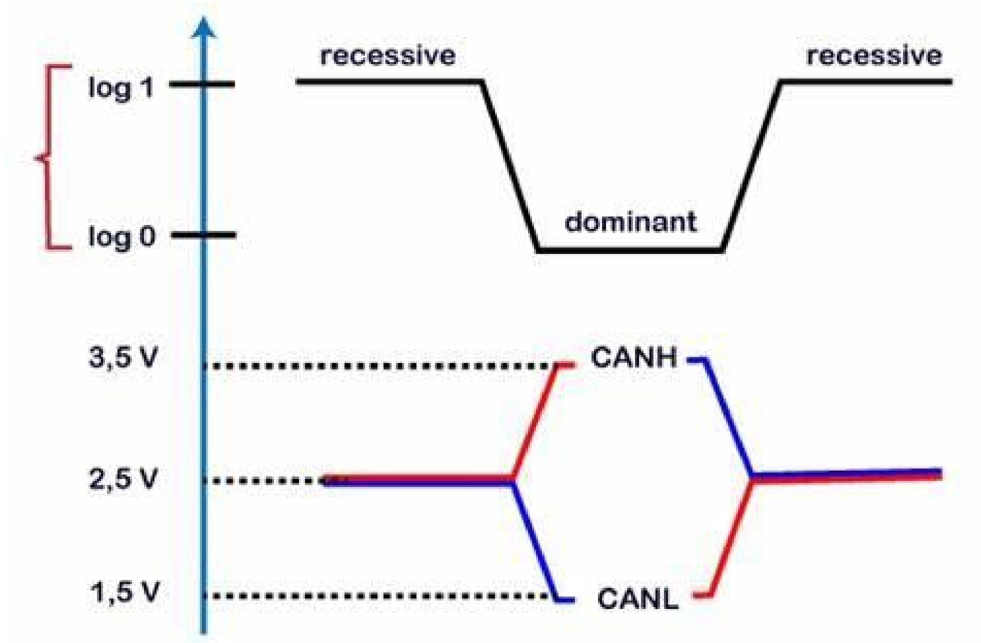


- CAN bus separate CAN_H and CAN_L so causing the different voltage.

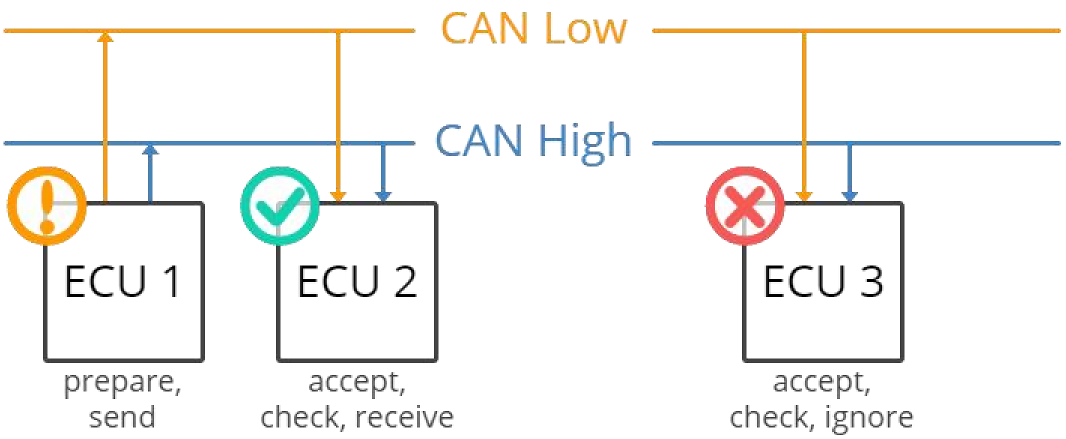
- Different voltage logic:
    - Logic 0: Dominant bit
    - Logic 1: Recessive bit



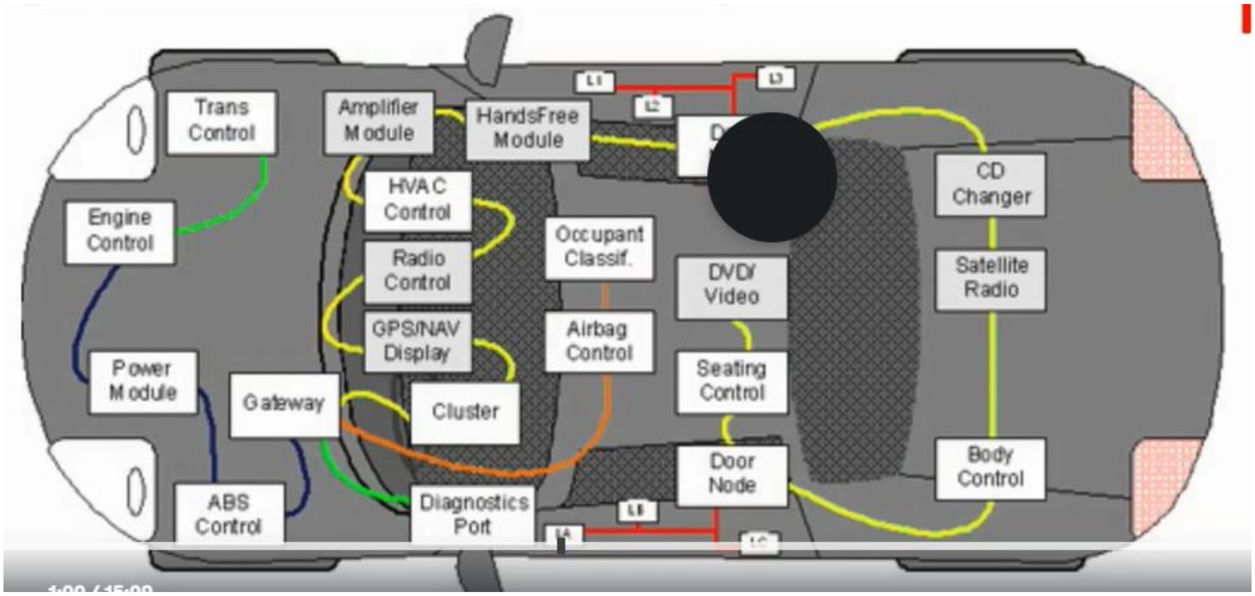
	CAN High	CAN Low	Delta V
1	2.5	2.5	0
0	3.5	1.5	2



- 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.

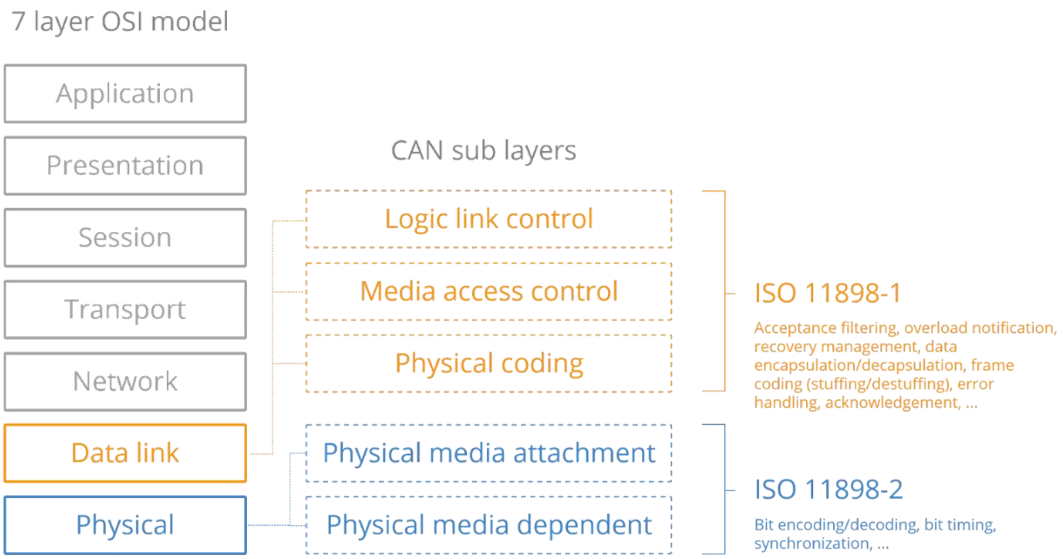


- In an automotive CAN bus system, ECUs can e.g. be the engine control unit, airbags, audio system etc. A modern car may have up to 70 ECUs - and each of them may have information that needs to be shared with other parts of the network.



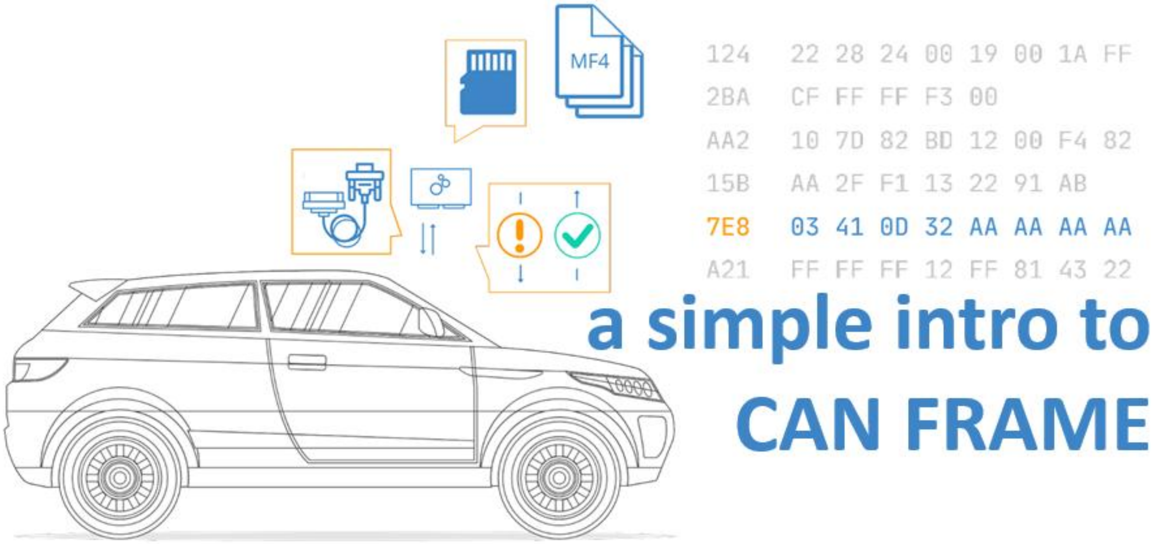
Physical & data link layer (OSI)

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.


7 layer OSI model
Application
Presentation
Session
Transport
Network
Data link
Physical
CAN sub layers
Logic link control
Media access control
Physical coding
Physical media attachment
Physical media dependent
ISO 11898-1
Acceptance filtering, overload notification, recovery management, data encapsulation/decapsulation, frame coding (stuffing/destuffing), error handling, acknowledgement, ...
ISO 11898-2
Bit encoding/decoding, bit timing, synchronization, ...

CAN Properties

- CAN is a Wired Protocol – Bus Topology
- CAN is a Serial Bus Protocol with MSB going first
- CAN is Asynchronous Communication Protocol
- CAN Protocol is a Multicast (Similar Broadcast) type
- Message Filtering
- CAN has an acknowledgement method


MF4
124 22 28 24 00 19 00 1A FF
2BA CF FF FF F3 00
AA2 10 7D 82 BD 12 00 F4 82
15B AA 2F F1 13 22 91 AB
7E8 03 41 0D 32 AA AA AA AA
A21 FF FF FF 12 FF 81 43 22
a simple intro to
CAN FRAME

CAN Frame

CAN Frame Types

1. Data Frame

- Frame carrying the data from transmitter node to all receiver nodes
- Serves the main purpose of CAN protocol
- Two types of Data Frame:
  - Standard Frame: has 12bit length for Arbitration Field
  - Extended Frame: has 32bit length for Arbitration Field
- In CAN when we are normally talking of frames, we are referring Data frames only

Bus idle	SOF	Arbitration Field	Control Field	Data Field	CRC Sequence	DEL	ACK	DEL	EOF	IFS	Bus idle

2. Remote Frame

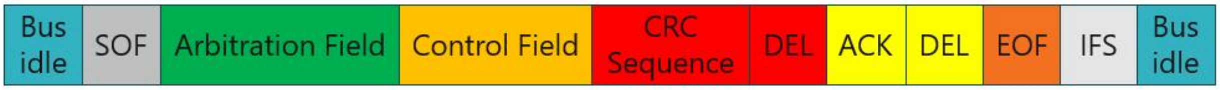
- Frame Requesting for a Data Frame

- A node wants a particular Data frame, so it sends the Remote frame with same Message ID on the bus

- This remote frame is followed by the Data frame on the bus transmitter by the node who owns that data frame

- Hence Remote frame sending node got the data it requested for

- Now a days Remote frame is obsolete and not used in CAN protocol



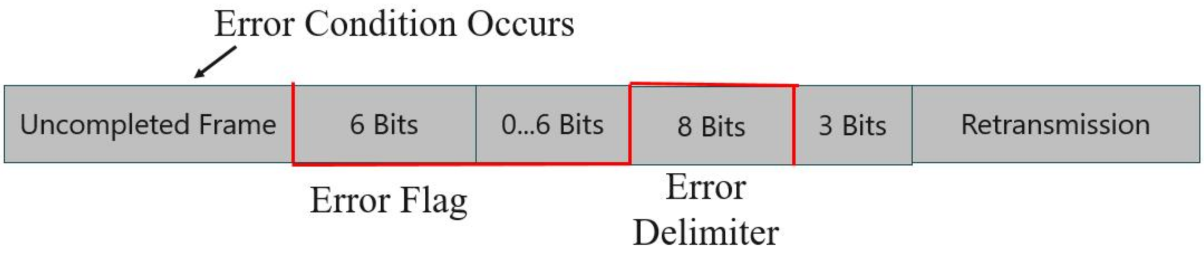
3. Error Frame

- Error Frame signals an error condition in the data frame being transmitted currently

- If a node detects an error in the data frame being transmitted on the bus currently, then it destroys that data frame and signals all other nodes by transmitting an error frame

- The error can be detected by any of the receiver nodes or the transmitter node

- Error Frame can be 14...20 Bits



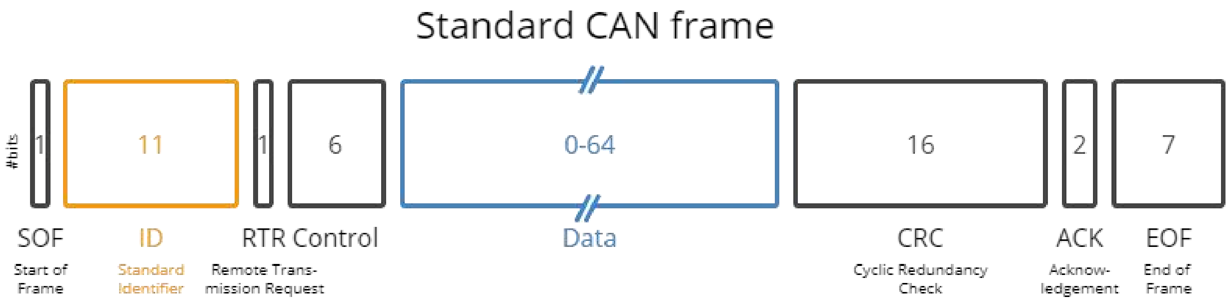
4. Overload Frame

- Overload Frame is transmitted by a node when it is overload and needs some time to process the data frame received previously

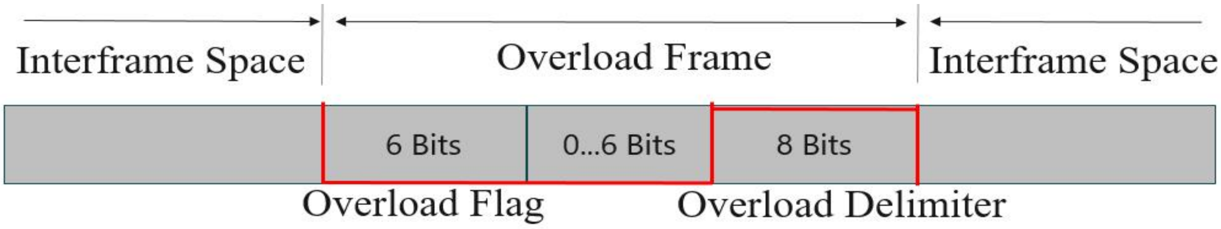
- Overload frame format is same as that of an error frame

- Maximum of 3 consecutive overload frames can be transmitted by each node after a data frame

Standard Data Frame Format



- SOF (1 Bit): Start of Frame (Dominant bit)
- Message ID (11 Bits):
  - 11 Bits ID
  - Has 512 different CAN frames in a single can network
  - Range from Hex Decimal 000 to Hex decimal 7FF
- RTR (1 Bit): Remote Transmit Request
  - RTR = 1: Remote Frame
  - RTR = 0: Data Frame
- IDE (1 Bit): Identifier extension bit, Dominant Bit
- Reserve bits (1 Bit): R0
- DLC (4 Bits): Data Length Code, number of bytes of data (0 – 8 bytes)
- Data Field (0-8 bytes): Data to by transmitted (length in bytes dictated by DCL)
- CRC Sequence (15 Bits): Cyclic redundancy check
- DEL (1 Bit): CRC Delimiter, must be recessive

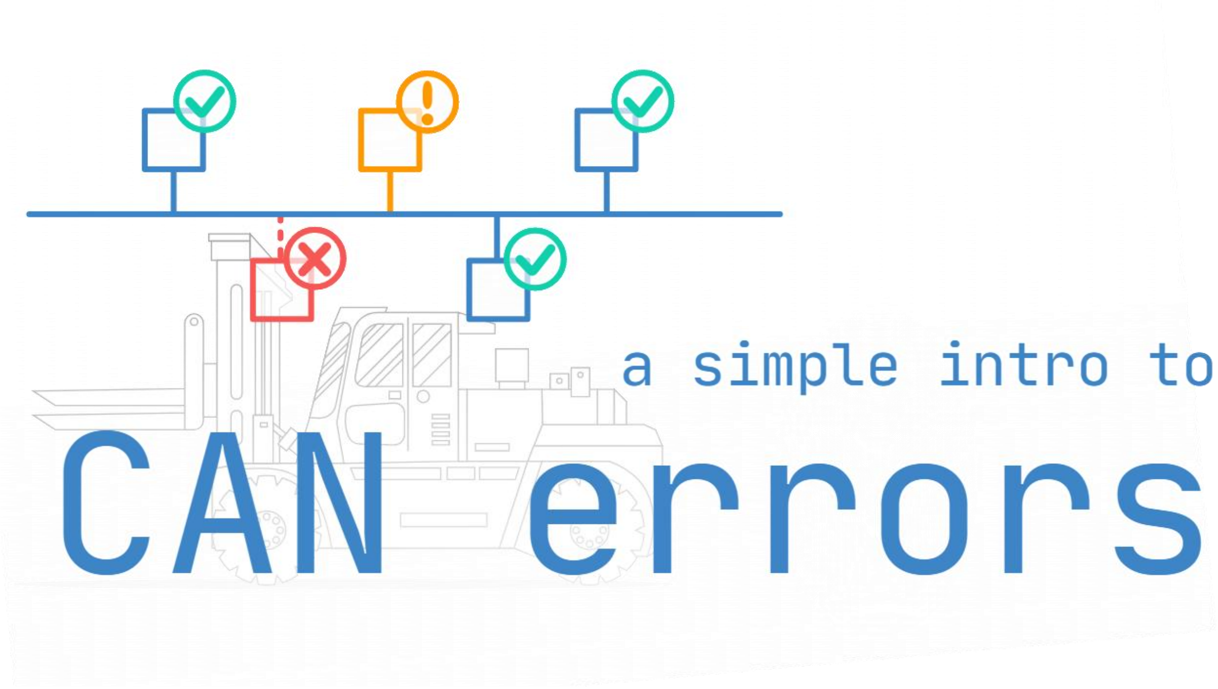


- ACK (1 Bit): Acknowledgement, transmitter sends recessive and any receiver can assert a dominant

- DEL (1 Bit): ACK Delimiter, must be recessive

- EOF (7 Bits): End Of Frame, must be recessive

- IFS (3 Bits): Interframe Spacing, must be recessive



CAN Bus Errors

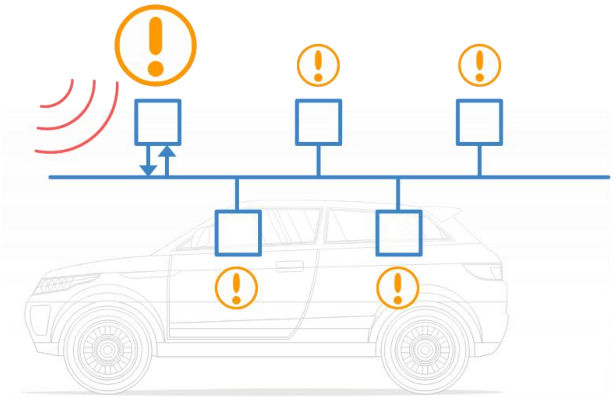
What are CAN bus errors?

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.

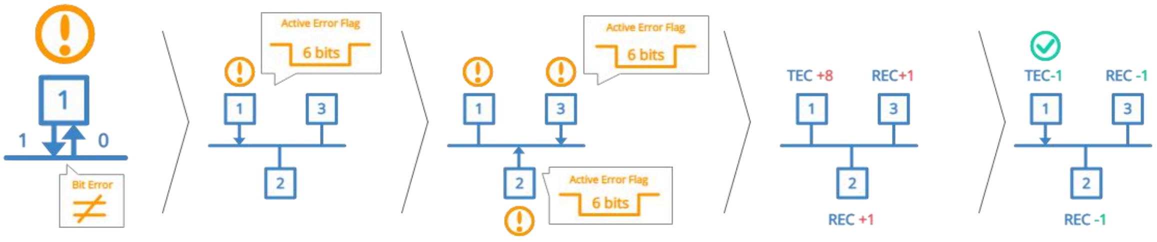
In particular, error handling identifies and rejects erroneous messages, enabling a sender to re-transmit the message.

Further, the process helps identify and disconnect CAN nodes that consistently transmit erroneous messages.



How does CAN errors 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

CAN Error Types

The CAN bus protocol specifies 5 CAN error types:

1. Bit Error [Transmitter]

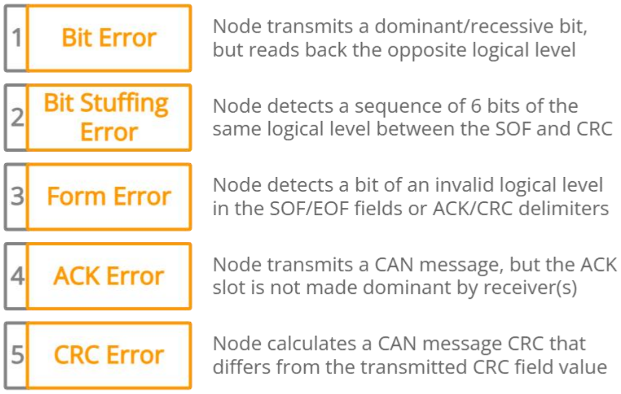
2. Bit Stuffing Error [Receiver]

3. Form Error [Receiver]

4. ACK Error (Acknowledgement) [Transmitter]

5. CRC Error (Cyclic Redundancy Check) [Receiver]

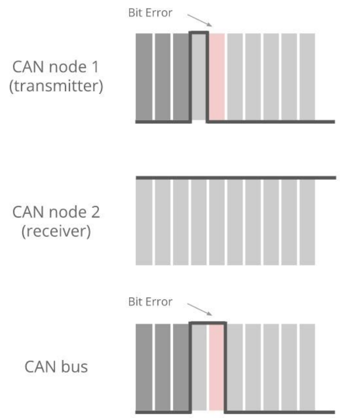
CAN bus error types



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

1. Bit Error

Example: CAN Bus Bit Error

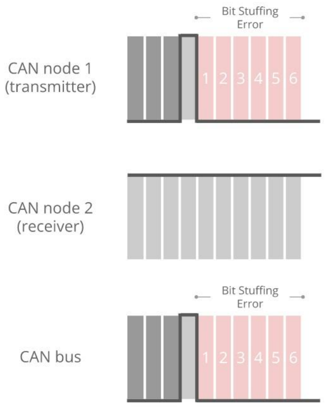


Every CAN node on the CAN bus will monitor the signal level at any given time - which means that a transmitting CAN node also "reads back" every bit it transmits. If the transmitter reads a different data bit level vs. what it transmitted, the transmitter detects this as a Bit Error.

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.

2. Bit Stuffing Error

Example: CAN Bus Bit Stuffing Error

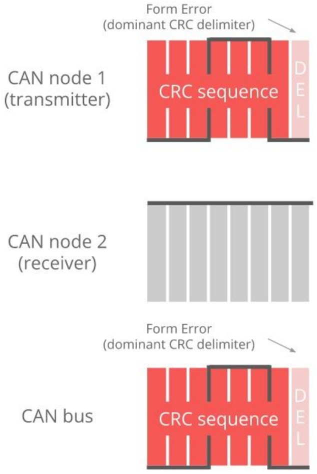


Bit stuffing is part of the CAN standard. It dictates that after every 5 consecutive bits of the same logical level, the 6th bit must be a complement. This is required to ensure the on-going synchronization of the network by providing rising edges. Further, it ensures that a stream of bits are not mis-interpreted 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.

3. Form Error

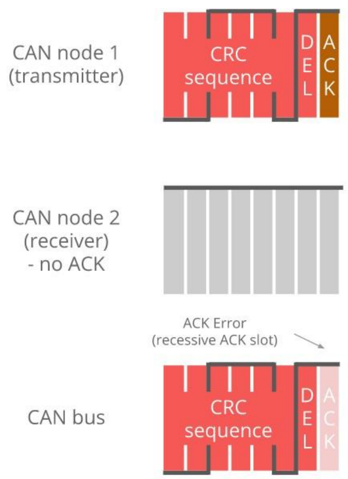
Example: CAN Bus Form Error



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.

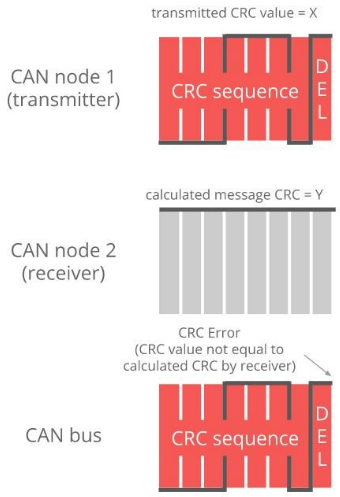
4. ACK Error (Acknowledgement)

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.

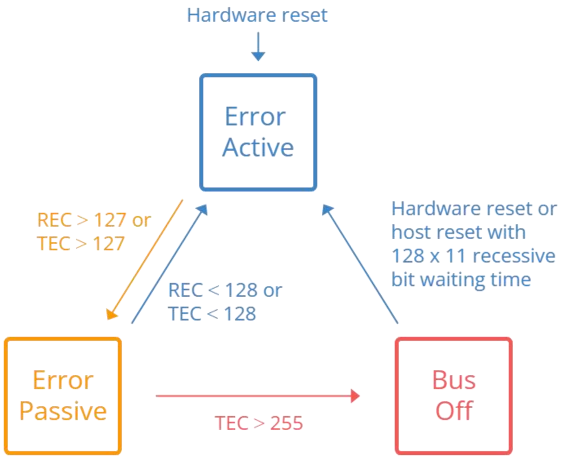

Example: CAN Bus ACK Error

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.


Example: CAN Bus CRC Error

CAN node states & error counter



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

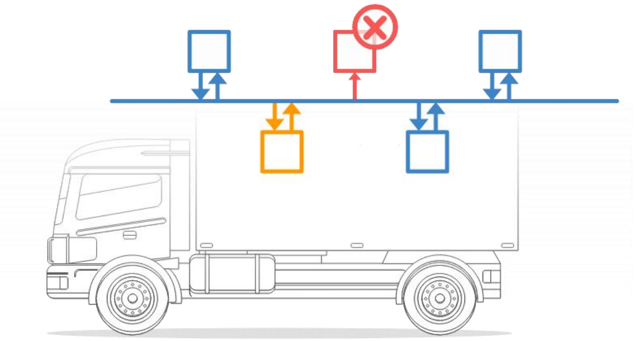
As evident, CAN error handling helps destroy erroneous messages - and enables CAN nodes to retry the transmission of erroneous messages.

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.

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:



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

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)

Bus Off : In this state, the CAN node disconnects itself from the CAN bus and can no longer transmit data or raise error flags.

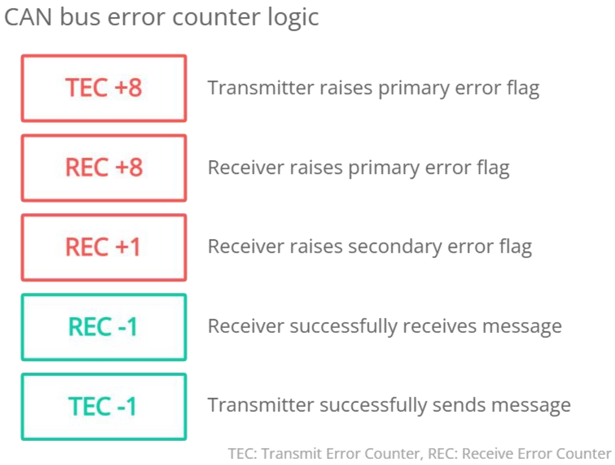
Every CAN controller keeps track of its own state and acts accordingly. CAN nodes shift state depending on the value of their error counters. Specifically, every CAN node keeps track on a Transmit Error Counter (TEC) and Receive Error Counter (REC):

- 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?

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.

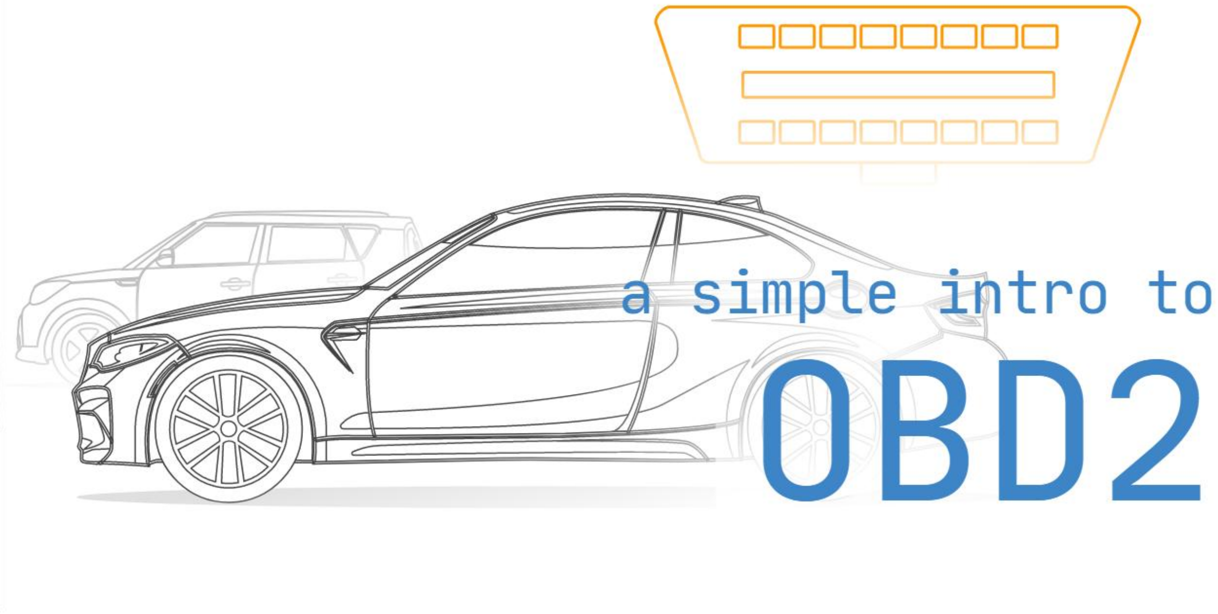
At first, it might sound positive to have a CAN node that repeatedly discovers errors and reacts promptly by raising an error flag before other nodes. However, in practice, the discoverer is typically also the culprit causing errors - and hence it is punished more severely as per the overview.



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

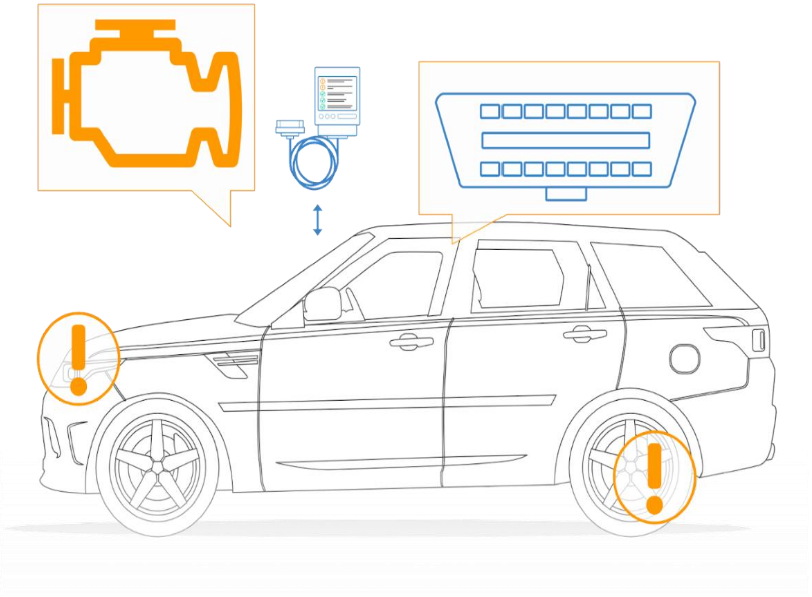
TEC: Transmit Error Counter, REC: Receive Error Counter



On Board Diagnostic (OBD2) Protocol

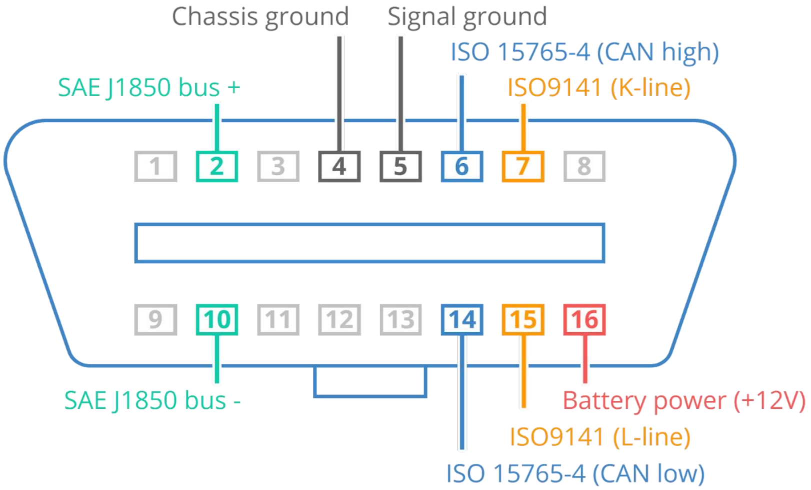
What is OBD2?

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

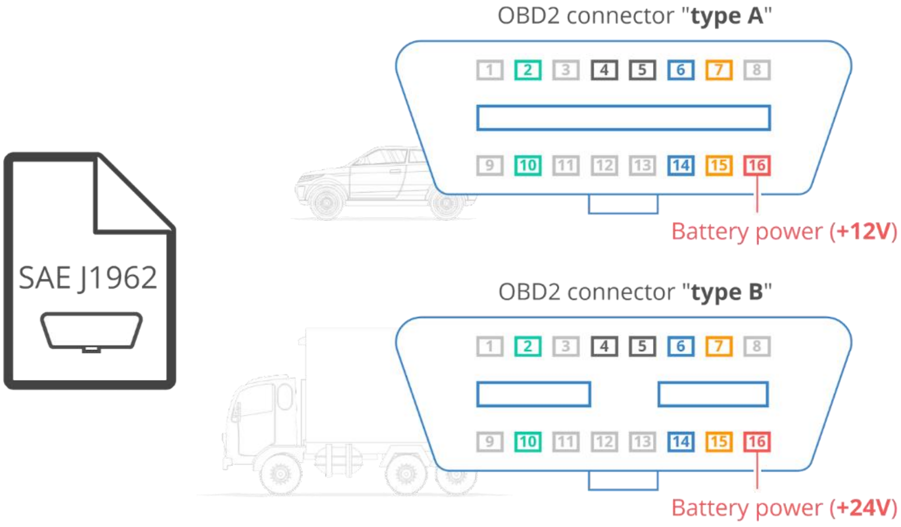
The OBD2 connector lets you access data from your car easily. The standard SAE J1962 specifies two female OBD2 16-pin connector types (A & B). In the illustration is an example of a Type A OBD2 pin connector (also sometimes referred to as the Data Link Connector, DLC).



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



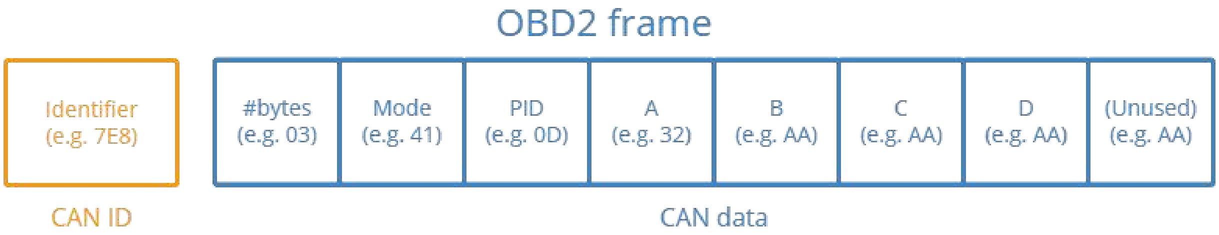
connector type A vs. B

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.

OBD2 Request

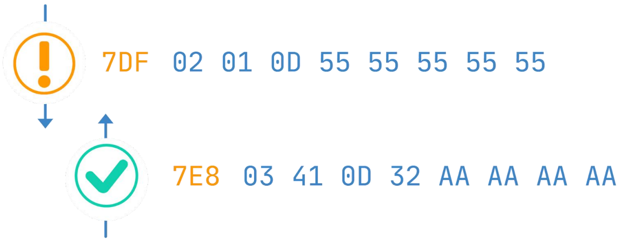
1. Raw OBD2 frame details



- **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, 0 D 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 OBD2 PID wiki . 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*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.

2. OBD2 request/response example

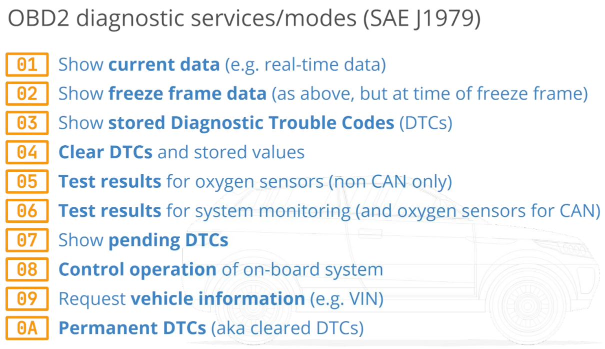
An example of a request/response CAN message for the PID 'Vehicle Speed' with a value of 50 km/h can be seen in the illustration. Note in particular how the formula for the OBD2 PID 0D (Vehicle Speed) simply involves taking the 4 th byte (0x32) and



converting it to decimal form (50).

3. 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)

4. 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.