KVASER (https://www.kvaser.com/)

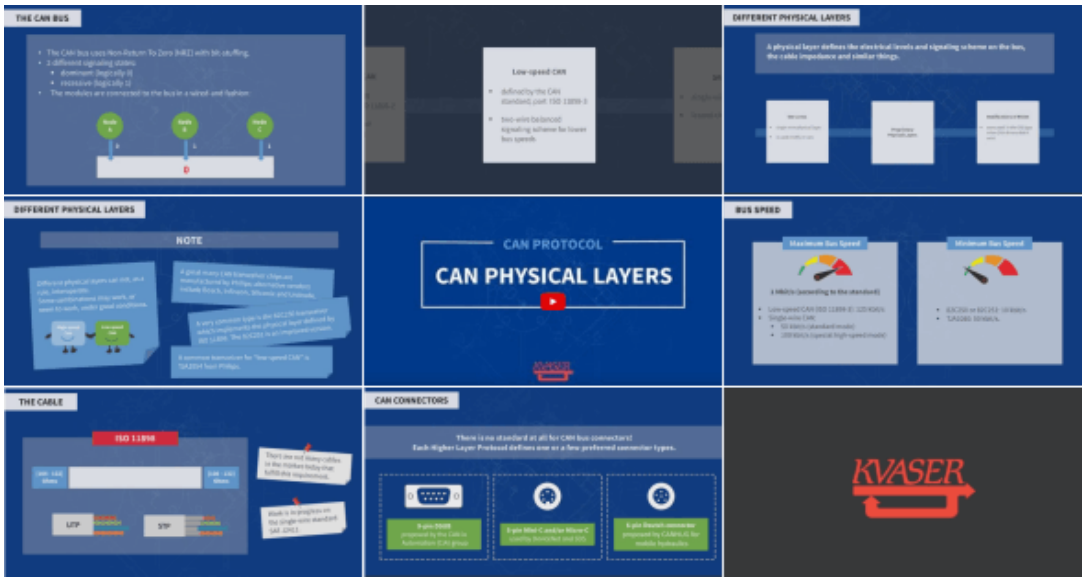
The CAN Protocol Tutorial

The CAN Protocol Tutorial gives an overview of the ISO 11898-1 and ISO 11898-2 controller area network standards. This tutorial provides a great introduction to the fundamentals of CAN (controller area network) as it is used in automotive design, industrial automation controls, and many more applications.

Feedback

Table of Contents

7. **CAN Error Handling**
8. **Higher Layer Protocols**



Take our e-Learning course!

Our updated 8-part video course sets a new standard in CAN training.

(https://www.kvaser.com/course/can-protocol-tutorial/)

Feedback

Introduction: The CAN Bus

What is CAN?

CAN is short for ‘controller area network’. Controller area network is an electronic communication bus defined by the ISO 11898 standards. Those standards define how communication happens, how wiring is configured and how messages are constructed, among other things. Collectively, this system is referred to as a CAN bus.

What We'll Cover

To get deeper into the details of CAN, the CAN bus is a broadcast type of bus. This means that all nodes can “hear” all transmissions. There is no way to send a message to just a specific node; all nodes will invariably pick up all traffic. The CAN hardware, however, provides local filtering so that each node may react only on the interesting messages. We’ll discuss this more in **Section 2, “CAN Messages” (https://www.kvaser.com/can-protocol-tutorial/##canMessages)**.

We'll also discuss how the bus uses Non-Return To Zero (NRZ) with bit-stuffing. In this system, the modules are connected to the bus in a wired-and fashion: if just one node is driving the bus to a logical 0, then the whole bus is in that state regardless of the number of nodes transmitting a logical 1.

The CAN standard defines four different message types. The messages uses a clever scheme of bit-wise arbitration to control access to the bus, and each message is tagged with a priority.

The CAN standard also defines an elaborate scheme for error handling and confinement which is described in more detail in **Section 7, "CAN Error Handling" (https://www.kvaser.com/can-protocol-tutorial/##errorHandling)**.

Bit timing and clock synchronization is discussed in **Section 6 (https://www.kvaser.com/can-protocol-tutorial/##bitTiming)** of this tutorial. Here's a **bit timing calculator (https://staging.kvaser.com/support/calculators/bit-timing-calculator/)** you can use to calculate the CAN bus parameters and register settings.

CAN bus wiring may be implemented using different physical layers (**Section 3 (https://www.kvaser.com/can-protocol-tutorial/##physicalLayers)**), some of which are described here, and there are also a fair number of CAN bus connector types (**Section 5 (https://www.kvaser.com/can-protocol-tutorial/##connectors)**) in use. We also provide a number of oscilloscope pictures (**Section 4 (https://www.kvaser.com/can-protocol-tutorial/##oscilloscopePictures)**) for those interested in the details of a message.

CAN Messages

The CAN bus is a broadcast type of bus. This means that all nodes can 'hear' all transmissions. There is no way to send a message to just a specific node; all nodes will invariably pick up all traffic. The CAN hardware, however, provides local filtering so that each node may react only on the interesting messages.

The CAN messages

CAN uses short messages – the maximum utility load is 94 bits. There is no explicit address in the messages; instead, the messages can be said to be contents-addressed, that is, their contents implicitly determines their address.

Message Types

There are four different message types (or 'frames') on a CAN bus:

1. the Data Frame
2. the Remote Frame
3. the Error Frame
4. the Overload Frame

The Data Frame

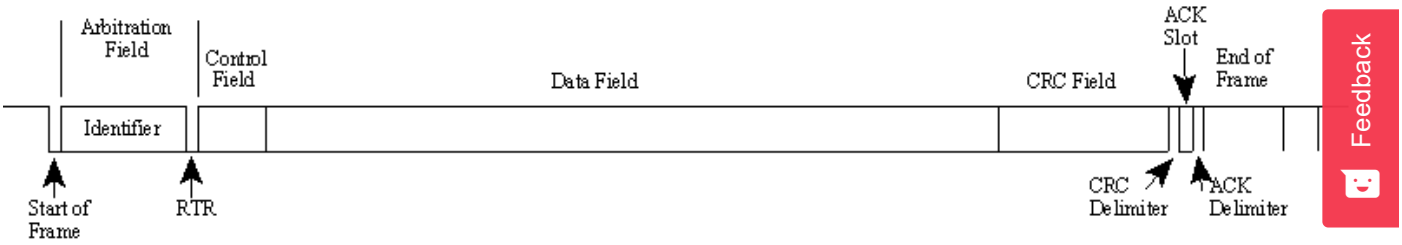
Summary: “Hello everyone, here’s some data labeled X, hope you like it!”

The Data Frame is the most common message type. It comprises the following major parts (a few details are omitted for the sake of brevity):

- the Arbitration Field, which determines the priority of the message when two or more nodes are contending for the bus. The Arbitration Field contains:
  - For CAN 2.0A, an 11-bit Identifier and one bit, the RTR bit, which is dominant for data frames.
  - For CAN 2.0B, a 29-bit Identifier (which also contains two recessive bits: SRR and IDE) and the RTR bit.
- the Data Field, which contains zero to eight bytes of data.
- the CRC Field, which contains a 15-bit checksum calculated on most parts of the message. This checksum is used for error detection.
- an Acknowledgement Slot; any CAN controller that has been able to correctly receive the message sends an Acknowledgement bit at the end of each message. The transmitter checks for the presence of the Acknowledge bit and retransmits the message if no acknowledge was detected.

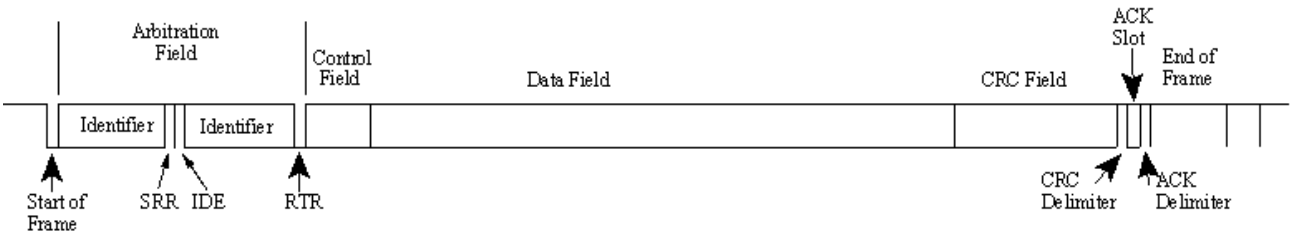
Note 1: It is worth noting that the presence of an Acknowledgement Bit on the bus does not mean that any of the intended addressees has received the message. The only thing we know is that one or more nodes on the bus has received it correctly.

Note 2: The Identifier in the Arbitration Field is not, despite of its name, necessarily identifying the contents of the message.



(https://staging.kvaser.com/wp-content/uploads/2014/01/1-can-msg-1-3.gif)

A CAN 2.0A (“standard CAN”) Data Frame.



(https://staging.kvaser.com/wp-content/uploads/2014/01/2-canmsg-1-3.gif)

A CAN 2.0B (“extended CAN”) Data Frame.

The Remote Frame

Summary: “Hello everyone, can somebody please produce the data labeled X?”

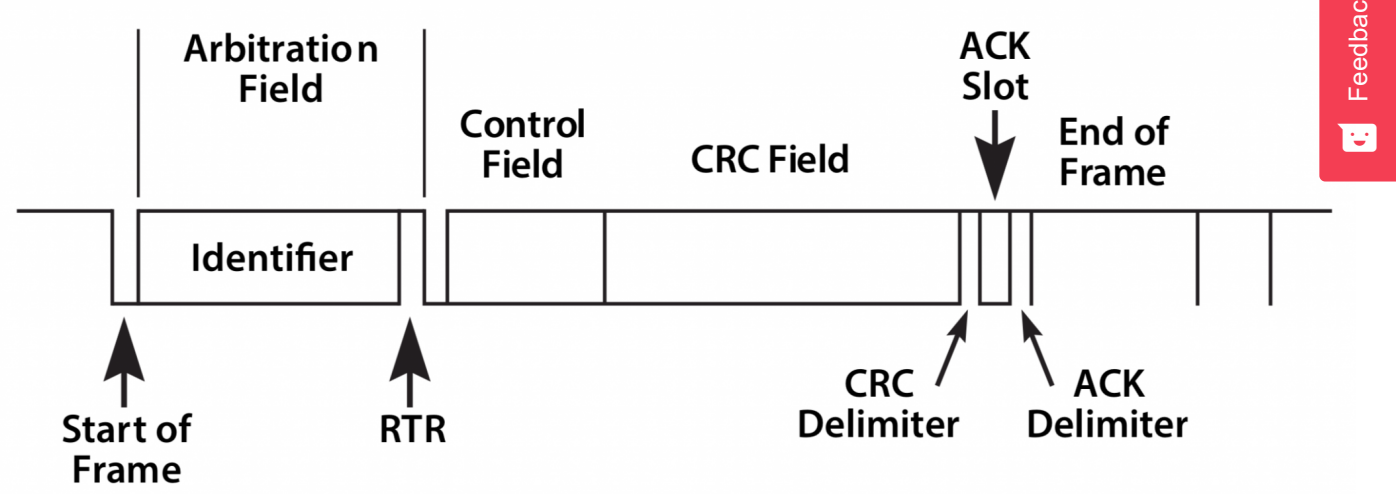
The Remote Frame is just like the Data Frame, with two important differences:

- it is explicitly marked as a Remote Frame (the RTR bit in the Arbitration Field is recessive), and
- there is no Data Field.

The intended purpose of the Remote Frame is to solicit the transmission of the corresponding Data Frame. If, say, node A transmits a Remote Frame with the Arbitration Field set to 234, then node B, if properly initialized, might respond with a Data Frame with the Arbitration Field also set to 234.

Remote Frames can be used to implement a request-response type of bus traffic management. In practice, however, the Remote Frame is little used. It is also worth noting that the CAN standard does not prescribe the behaviour outlined here. Most CAN controllers can be programmed either to automatically respond to a Remote Frame, or to notify the local CPU instead.

There’s one catch with the Remote Frame: the Data Length Code must be set to the length of the expected response message. Otherwise the arbitration will not work.



A Remote Frame (2.0A type)

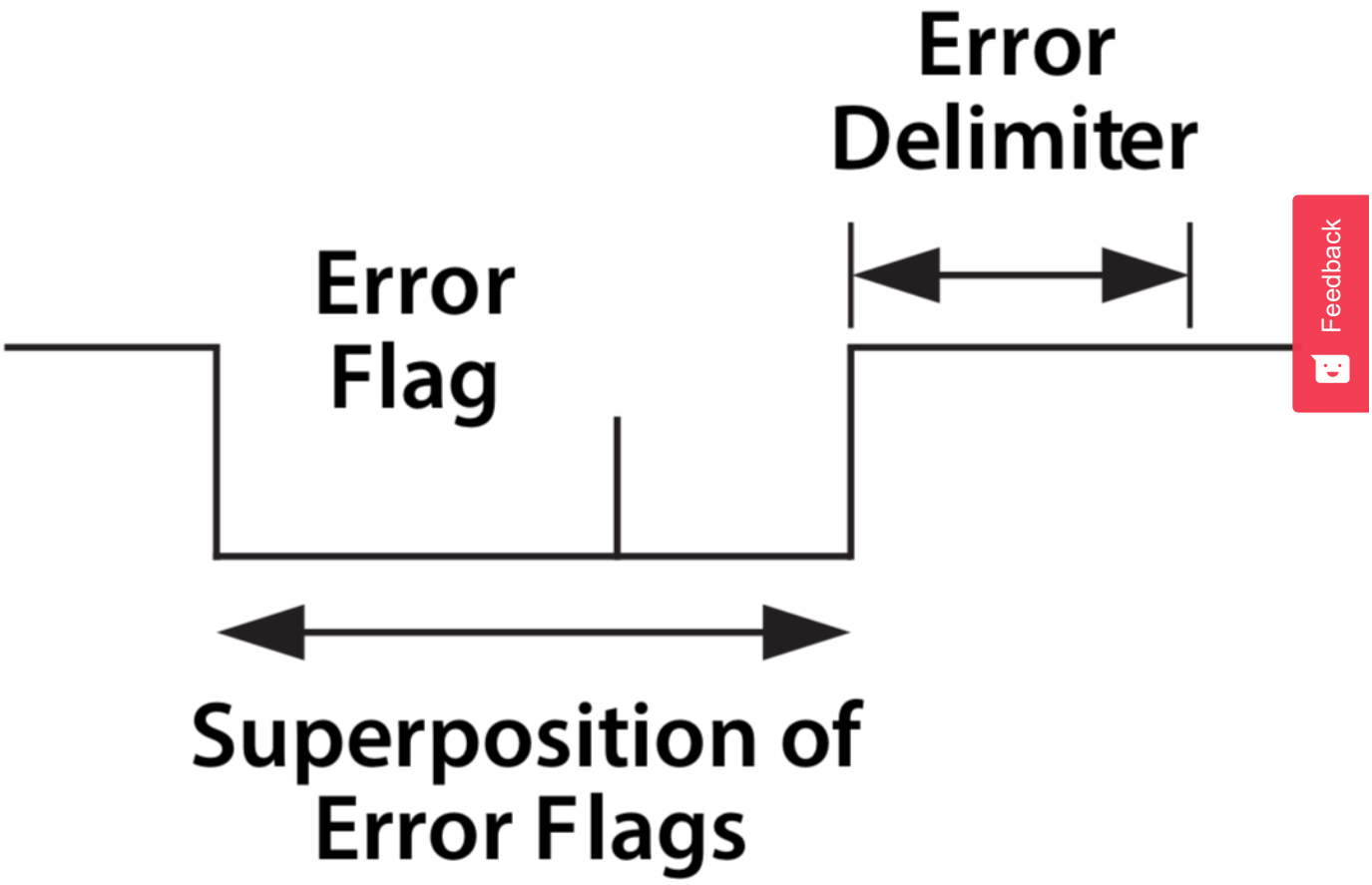
Sometimes it is claimed that the node responding to the Remote Frame is starting its transmission as soon as the identifier is recognized, thereby “filling up” the empty Remote Frame. This is not the case.

The Error Frame

Summary: (everyone, aloud) “OH DEAR, LET’S TRY AGAIN”

Simply put, the Error Frame is a special message that violates the framing rules of a CAN message. It is transmitted when a node detects a fault and will cause all other nodes to detect a fault – so they will send Error Frames, too. The transmitter will then automatically try to retransmit the message. There is an elaborate scheme of error counters that ensures that a node can’t destroy the bus traffic by repeatedly transmitting Error Frames.

The Error Frame consists of an Error Flag, which is 6 bits of the same value (thus violating the bit-stuffing rule) and an Error Delimiter, which is 8 recessive bits. The Error Delimiter provides some space in which the other nodes on the bus can send their Error Flags when they detect the first Error Flag.



An Error Frame

The Overload Frame

Summary: “I’m a very busy little 82526, could you please wait for a moment?”

The Overload Frame is mentioned here just for completeness. It is very similar to the Error Frame with regard to the format and it is transmitted by a node that becomes too busy. The Overload Frame is not used very often, as today’s CAN controllers are clever enough not to use it. In fact, the only controller that will generate Overload Frames is the now obsolete 82526.

Standard vs. Extended CAN

Originally, the CAN standard defined the length of the Identifier in the Arbitration Field to eleven (11) bits. Later on, customer demand forced an extension of the standard. The new format is often called Extended CAN and allows no less than twenty-nine (29) bits in the Identifier. To differentiate between the two frame types, a reserved bit in the Control Field was used.

The standards are formally called

- 2.0A, with 11-bit Identifiers only,
- 2.0B, extended version with the full 29-bit Identifiers (or the 11-bit, you can mix them.) A 2.0B node can be
  - “2.0B active”, i.e. it can transmit and receive extended frames, or
  - “2.0B passive”, i.e. it will silently discard received extended frames (but see below.)
- 1.x refers to the original specification and its revisions.

New CAN controllers today are usually of the 2.0B type. A 1.x or 2.0A type controller will get very upset if it receives messages with 29 arbitration bits. A 2.0B passive type controller will tolerate them, acknowledge them if they are correct and then – discard them; a 2.0B active type controller can both transmit and receive them.

Controllers implementing 2.0B and 2.0A (and 1.x) are compatible – and may be used on the same bus – as long as the controllers implementing 2.0B refrain from sending extended frames!

Sometimes people advocate that standard CAN is “better” than Extended CAN because there is more overhead in the Extended CAN messages. This is not necessarily true. If you use the Arbitration Field for transmitting data, then Extended CAN may actually have a lower overhead than Standard CAN has.

Basic CAN vs. Full CAN

The terms “Basic CAN” and “Full CAN” originate from the childhood of CAN. Once upon a time there was the Intel 82526 CAN controller which provided a DPRAM-style interface to the programmer. Then came along Philips with the 82C200 which used a FIFO- (queue-) oriented programming model and limited filtering abilities. To distinguish between the two programming models, people for some reason termed the Intel way as “Full CAN” and the Philips way as “Basic CAN”. Today, most CAN controllers allow for both programming models, so there is no reason to use the terms “Full CAN” and “Basic CAN” – in fact, these terms can cause confusion and should be avoided.

Of course, a “Full CAN” controller can communicate with a “Basic CAN” controller and vice versa. There are no compatibility problems.

Bus Arbitration and Message Priority

The message arbitration (the process in which two or more CAN controllers agree on who is to use the bus) is of great importance for the actually available bandwidth for data transmission.

Any CAN controller may start a transmission when it has detected an idle bus. This may result in two or more controllers starting a message (almost) at the same time. The conflict is resolved in the following way. The transmitting nodes monitor the bus while they are sending. If a node detects a dominant level when it is sending a recessive level itself, it will immediately quit the arbitration process and become a receiver instead. The arbitration is performed over the whole Arbitration Field and when that field has been sent, exactly one transmitter is left on the bus. This node continues the transmission as if nothing had happened. The other potential transmitters will try to retransmit their messages when the bus becomes available next time. No time is lost in the arbitration process.

An important condition for this bit-wise arbitration to succeed is that no two nodes may transmit the same Arbitration Field. There is one exception to this rule: if the message contains no data, then any node may transmit that message.

Since the bus is wired-and and a Dominant bit is logically 0, it follows that the message with the numerically lowest Arbitration Field will win the arbitration.

Q: What happens if a node is alone on the bus and tries to transmit?
A: The node will, of course, win the arbitration and happily proceeds with the message transmission. But when the time comes for acknowledging... no node will send a dominant bit during the ACK slot, so the transmitter will sense an ACK error, send an error flag, increase its transmit error counter by 8 and start a retransmission. This will happen 16 times; then the transmitter will go error passive. By a special rule in the error confinement algorithm, the transmit error counter is not further increased if the node is error passive and the error is an ACK error. So the node will continue to transmit forever, at least until someone acknowledges the message.

Message Addressing and Identification

It is worth noting once again that there is no explicit address in the CAN messages. Each CAN controller will pick up all traffic on the bus, and using a combination of hardware filters and software, determine if the message is “interesting” or not.

In fact, there is no notion of message addresses in CAN. Instead, the contents of the messages is identified by an identifier which is present somewhere in the message. CAN messages are said to be “contents-addressed”.

A conventional message address would be used like “Here’s a message for node X”. A contents-addressed message is like “Here’s a message containing data labeled X”. The difference between these two concepts is small but significant.

The contents of the Arbitration Field is, per the Standard, used to determine the message’s priority on the bus. All CAN controllers will also use the whole (some will use just a part) of the Arbitration Field as a key in the hardware filtration process.

The Standard does not say that the Arbitration Field must be used as a message identifier. It’s nevertheless a very common usage.

A note on the Identifier Values

We said that 11 (CAN 2.0A) or 29 (CAN 2.0B) bits is available in the Identifier. This is not entirely correct. Due to compability with a certain old CAN controller (guess which?), identifiers must not have the 7 most significant bits set to all ones, so only the identifiers 0..2031 are left for the 11-bit identifiers, and the user of 29-bit identifiers can use 532676608 different values.

Note that all other CAN controllers accept the “illegal” identifiers, so in a modern CAN system identifiers 2032..2047 can be used without restrictions.

CAN Physical Layers

The CAN Bus

The CAN bus uses Non-Return To Zero (NRZ) with bit-stuffing. There are two different signaling states: dominant (logically 0) and recessive (logically 1). These correspond to certain electrical levels which depend on the physical layer used (there are several.) The modules are connected to the bus in a wired-and fashion: if just one node is driving the bus to the dominant state, then the whole bus is in that state regardless of the number of nodes transmitting a recessive state.

Different Physical Layers

A physical layer defines the electrical levels and signaling scheme on the bus, the cable impedance and similar things.

There are several different physical layers:

- The most common type is the one defined by the CAN standard, part ISO 11898-2, and it's a two-wire balanced signaling scheme. It is also sometimes known as "high-speed CAN".
- Another part of the same ISO standard, ISO 11898-3, defines another two-wire balanced signaling scheme for lower bus speeds. It is fault tolerant, so the signaling can continue even if one bus wire is cut or shorted to ground or Vbat. It is sometimes known as "low-speed CAN".
- SAE J2411 defines a single-wire (plus ground, of course) physical layer. It's used chiefly in cars – e.g. GM-LAN.
- Several proprietary physical layers do exist.
- Modifications of RS485 were used in the Old Ages when CAN drivers didn't exist.
- Go to Page 6 to view a number of oscilloscope pictures for those interested in the details of a message.

Different physical layers can not, as a rule, interoperate. Some combinations may work, or seem to work, under good conditions. For example, using both "high-speed" and "low-speed" transceivers on the same bus can work ... sometimes.

A great many CAN transceiver chips are manufactured by NXP; alternative vendors include Bosch, Infineon, Texas Instruments and Vishay Siliconix.

A very common type is the 82C250 transceiver which implements the physical layer defined by ISO 11898. The 82C251 is an improved version.

A common transceiver for "low-speed CAN" is TJA1054 from NXP.

Maximum Bus Speed

The maximum speed of a CAN bus, according to the standard, is 1 Mbit/second. Some CAN controllers will nevertheless handle higher speeds than 1Mbit/s and may be considered for special applications.

Low-speed CAN (ISO 11898-3, see above) can go up to 125 kbit/s.

Single-wire CAN can go up to around 50 kbit/s in its standard mode and, using a special high-speed mode used e.g. for ECU programming, up to around 100 kbit/s.

Minimum Bus Speed

Be aware that some bus transceivers will not allow you to go below a certain bit rate. For example, using 82C250 or 82C251 you can go down to 10 kbit/s without problems, but if you use the TJA1050 instead you can't go below around 50 kbit/s. Check the data sheet.

Maximum Cable Length

At a speed of 1 Mbit/s, a maximum cable length of about 40 meters (130 ft.) can be used. This is because the arbitration scheme requires that the wave front of the signal be able to propagate to the most remote node and back again before the bit is sampled. In other words, the cable length is restricted by the speed of light. A proposal to increase the speed of light has been considered but was turned down because of its inter-galactic consequences.

Other maximum cable lengths are (these values are approximate):

- 100 meters (330 ft) at 500 kbit/s
- 200 meters (650 ft) at 250 kbit/s
- 500 meters (1600 ft) at 125 kbit/s
- 6 kilometers (20000 ft) at 10 kbit/s

If optocouplers are used to provide galvanic isolation, the maximum bus length is decreased accordingly. Hint: use fast optocouplers, and look at the delay through the device, not at the specified maximum bit rate.

Bus Termination

An ISO 11898 CAN bus must be terminated. This is done using a resistor of 120 Ohms in each end of the bus. The termination serves two purposes:

1. Remove the signal reflections at the end of the bus.
2. Ensure the bus gets correct DC levels.

An ISO 11898 CAN bus must always be terminated regardless of its speed. I'll repeat this: an ISO 11898 CAN bus must always be terminated regardless of its speed. For laboratory work just one terminator might be enough. If your CAN bus works even though you haven't put any terminators on it, you are just lucky.

Note that other physical layers, such as “low-speed CAN”, single-wire CAN, and others, may or may not require termination. But your vanilla high-speed ISO 11898 CAN bus will always require at least one terminator.

Learn more about **CANbus termination in this article. (https://www.kvaser.com/using-termination-ensure-recessive-bit-transmission/)**

The Cable

The ISO 11898 prescribes that the cable impedance be nominally 120 Ohms, but an impedance in the interval of [108..132] Ohms is permitted.

There are not many cables in the market today that fulfill this requirement. There is a good chance that the allowed impedance interval will be broadened in the future.

ISO 11898 is defined for a twisted pair cable, shielded or unshielded. Work is in progress on the single-wire standard SAE J2411.

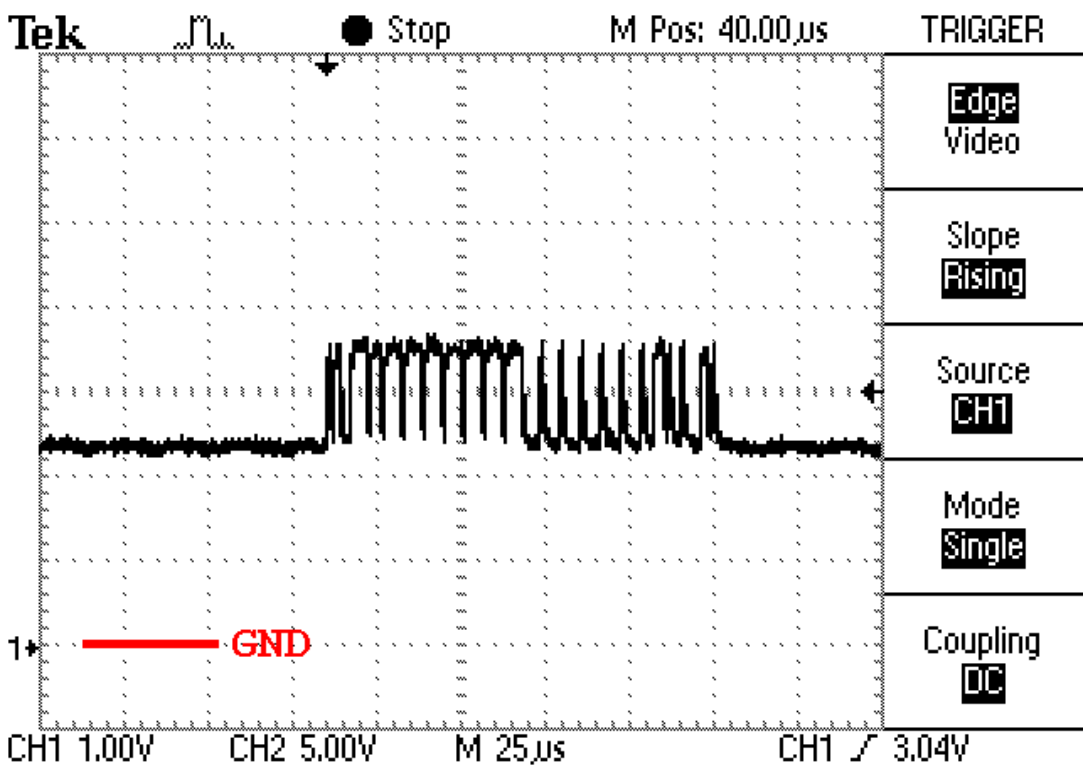
CAN connectors

There is no standard at all for CAN bus connectors! Usually, each Higher Layer Protocol(!) defines one or a few preferred CAN bus connector types. Common types include

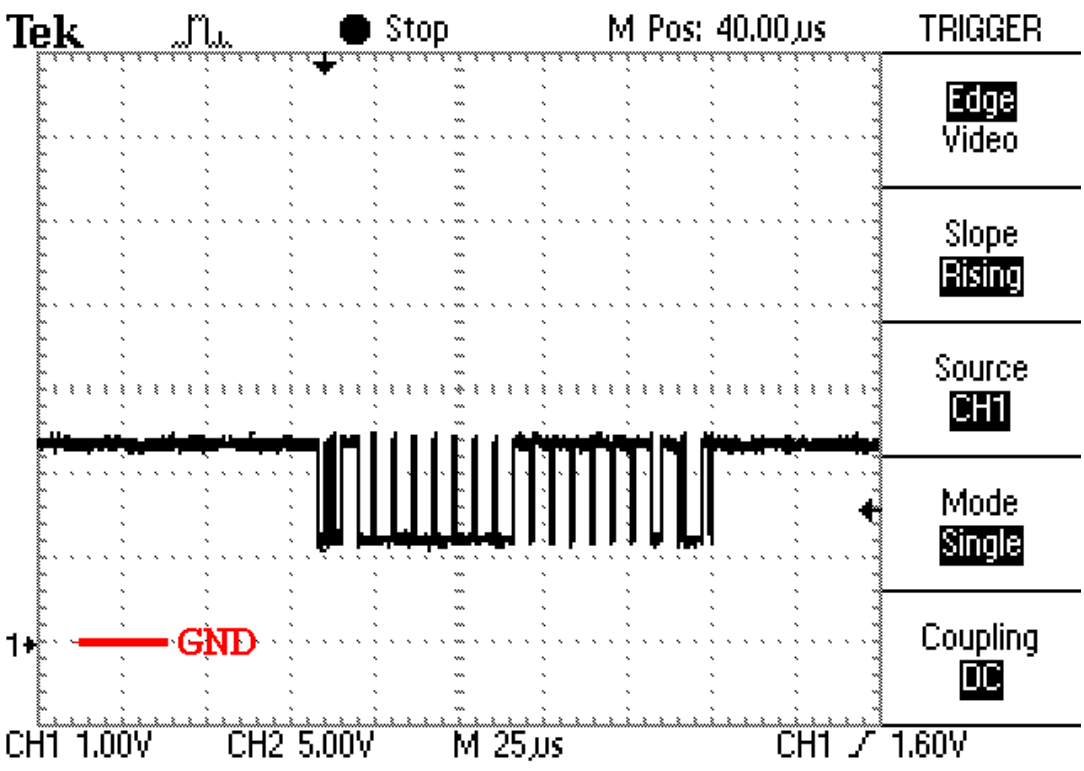
- 9-pin DSUB, proposed by CiA.
- 5-pin Mini-C and/or Micro-C, used by DeviceNet and SDS.
- 6-pin Deutch connector, proposed by CANHUG for mobile hydraulics.Go to Page 7 to view a few different connector layouts.

CAN Oscilloscope Pictures

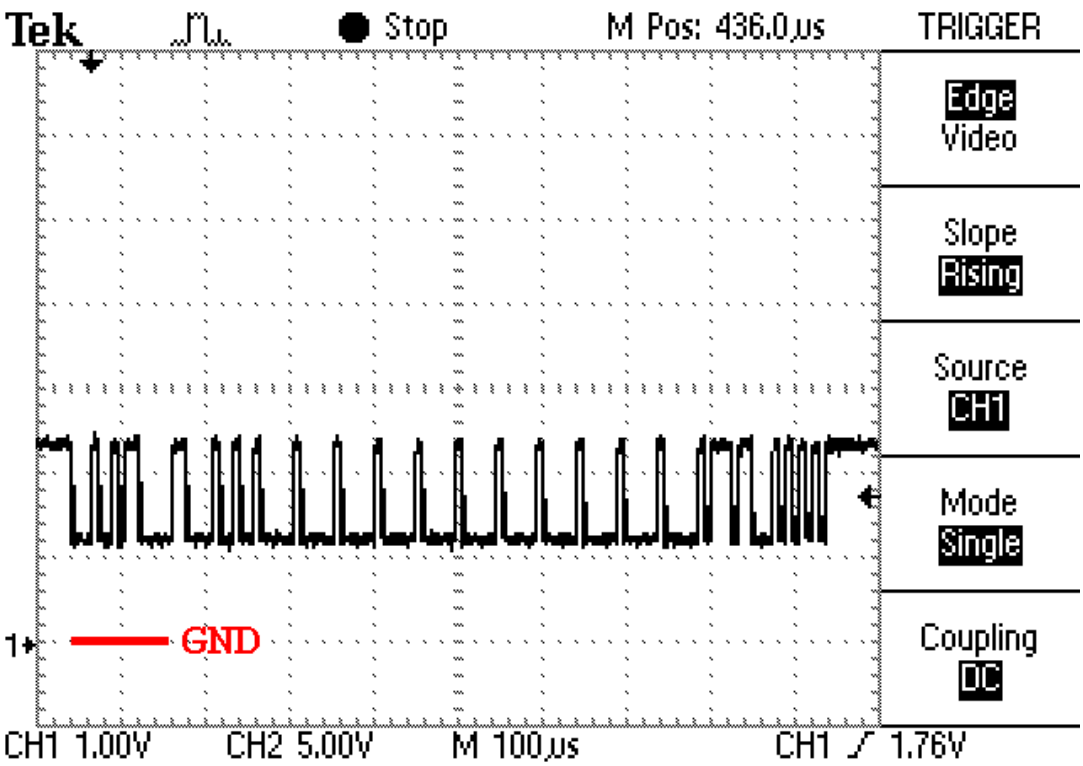
Here’s a picture from a perfectly normal ISO 11898 CAN bus, running at 1 Mbit/s. The transceiver is a 82C251; in other words, the physical layer is the one specified by ISO 11898.



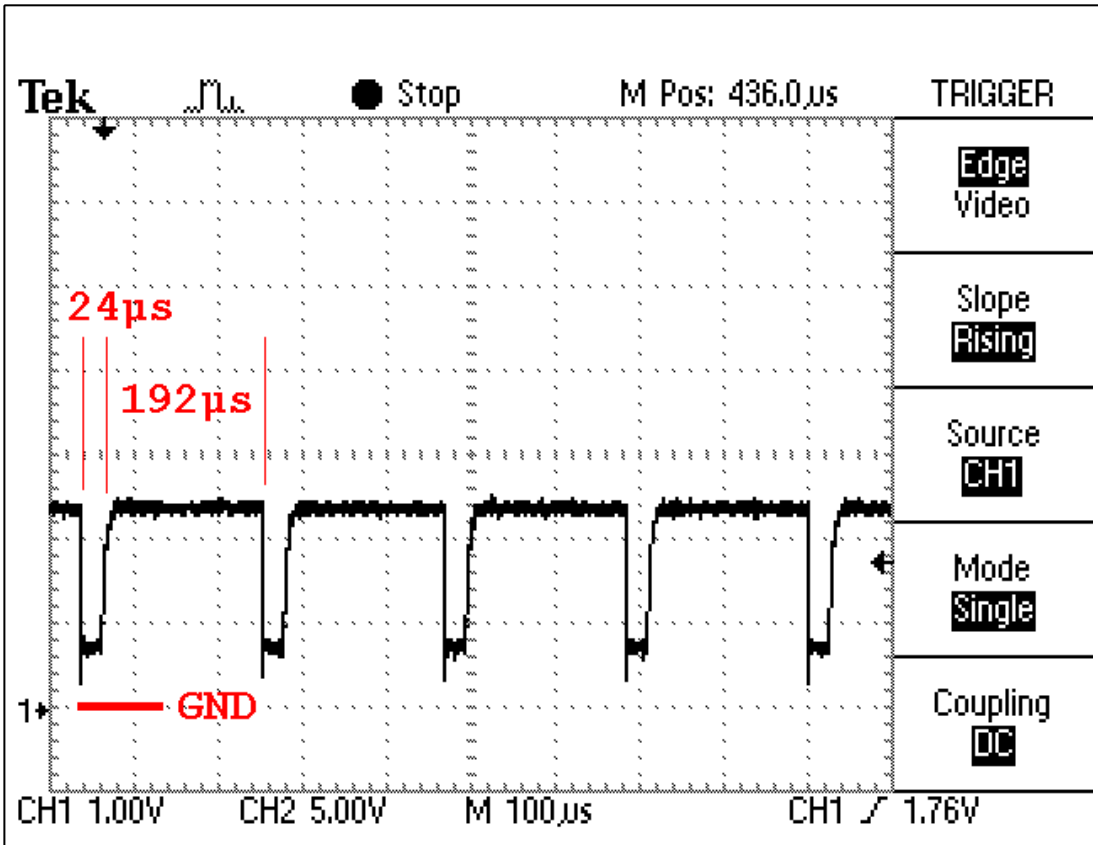
The measurement was done between CAN_H and GND. Note that the quiescent and recessive bus voltages lie around 2.5 V. When a dominant bit is transmitted the voltage rises to around 3.5V.



Now here's the same bus, but measurement is done between CAN_L and GND instead.



Here's another message, sent at 125 kbit/s. The message's (11-bit) identifier is 300, or 12c in hexadecimal. Look closely and you should be able to identify the first bits in the message.

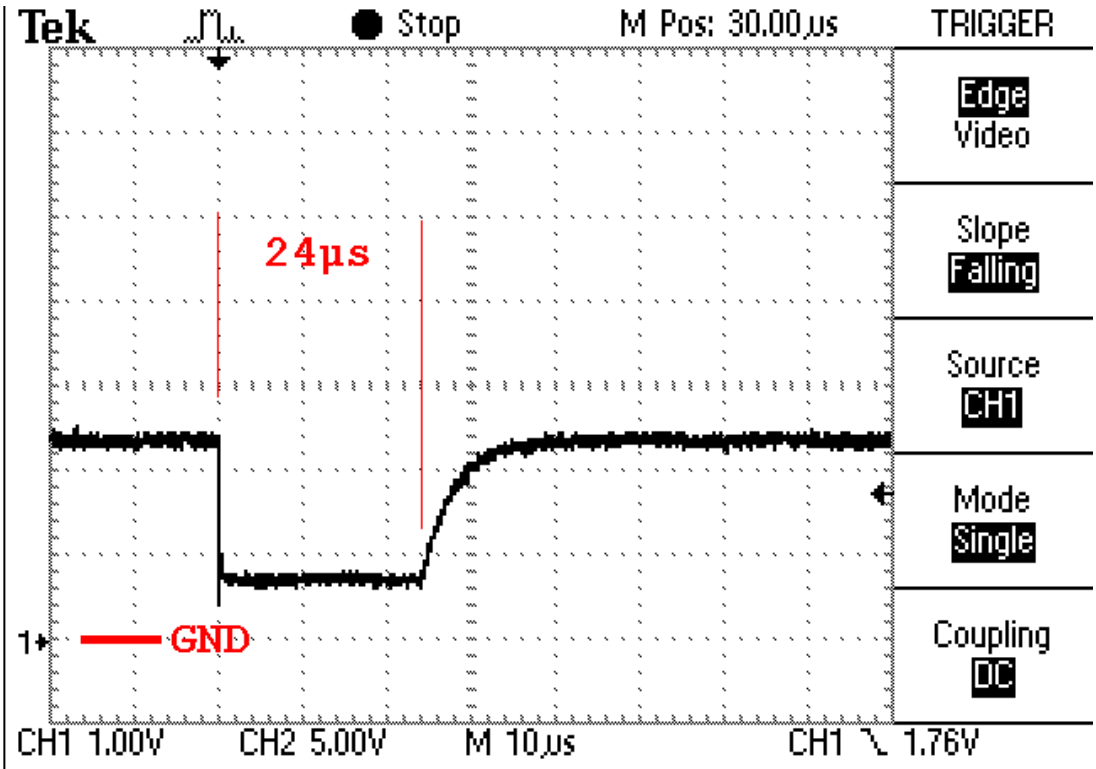


Here's a trickier picture. It shows the same message as above, still (11-bit) identifier 300 and still 125 kbit/s, but **without termination on the CAN bus**. The CAN cable was a short run of flat ribbon cable.

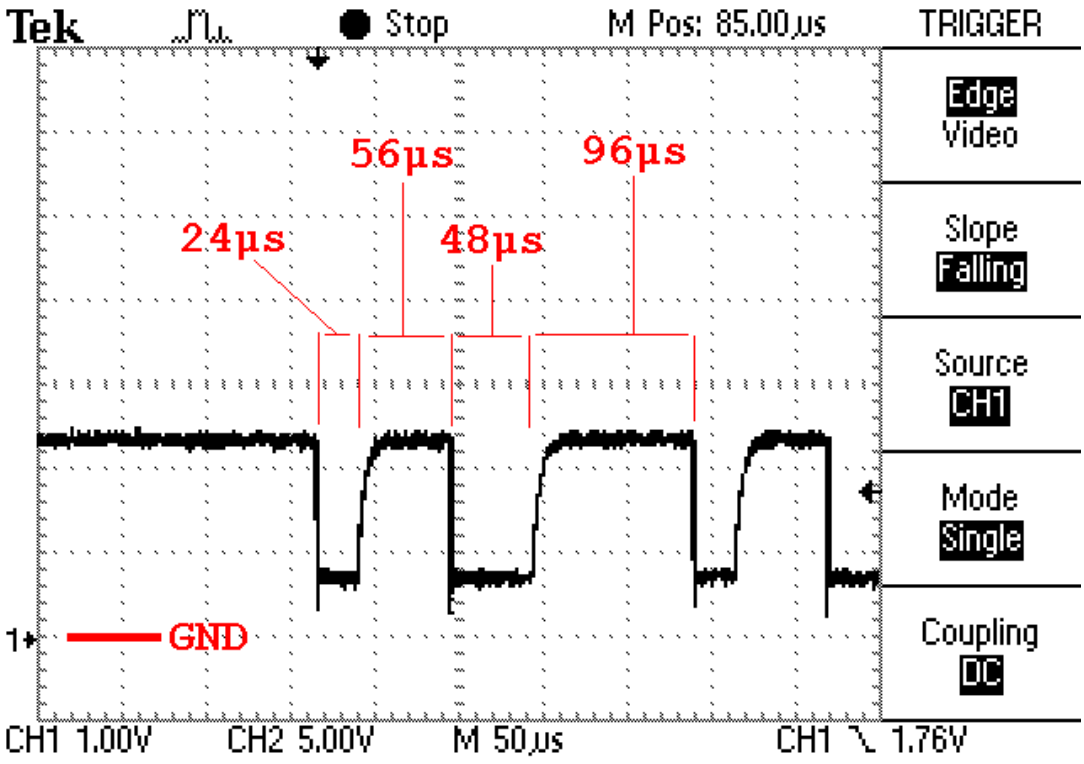
So, what's happening? This is 125 kbit/s, so one bit is 8 microseconds.

1. First the transmitter sends a start bit. This is a logical '0', i.e. a dominant level.
2. Then the identifier is transmitted. 300 decimal is 12c in hex, or 001 0010 1100 in binary. The first two zeroes are transmitted just fine. This explains the 24 microseconds of dominant level as seen in the picture.
3. Then a '1' should be transmitted, but as the bus isn't terminated the rising slope is not what is should have been. The transmitting node will now *think* it saw a '0' on the bus.
4. Since this happens during the arbitration phase, the transmitter will stop transmitting – it thinks that some other node is transmitting. The bus will now be left recessive, because no-one is in fact transmitting.
5. After 6 recessive bits, both the transmitter and the receivers will detect a *stuff error* and the error handling starts. At this point, 80 microseconds has gone (one start bit, two '0', one misinterpreted bit, and six recessive bits – a total of 10 bits = 80 microseconds).
6. All nodes detecting the stuff error will now start to transmit an error frame. In this case the error frame is *passive* because a number of errors was generated before the above picture was captured, so the transmitter is *error passive*. A passive error frame is just as an active error frame, but it's transmitted with a recessive level and so isn't visible on the bus.
7. The passive error frame prevails for 6 bit times.
8. Then all nodes are waiting for a period of 8 recessive bits, called the *error delimiter*.
9. Then all nodes are waiting for a period of 3 recessive bits, called the *intermission*.
10. Summing up the numbers above, we arrive at 1+6+6+8+3 = 24 recessive bits = 192 microseconds (see the picture!).

Moral: Always terminate the CAN bus! The reflections will not necessarily hurt, but the bad shape of the edges will kill the communication.



Here's the same CAN bus in another time scale. The CAN bus was around 2 decimeters (8 in.) long. The undershoot and ringing is visible but in this case clearly not important. This time the slow rising edge is the culprit.



Here's the same setup, but this time both the transmitter and the receiver are error active.

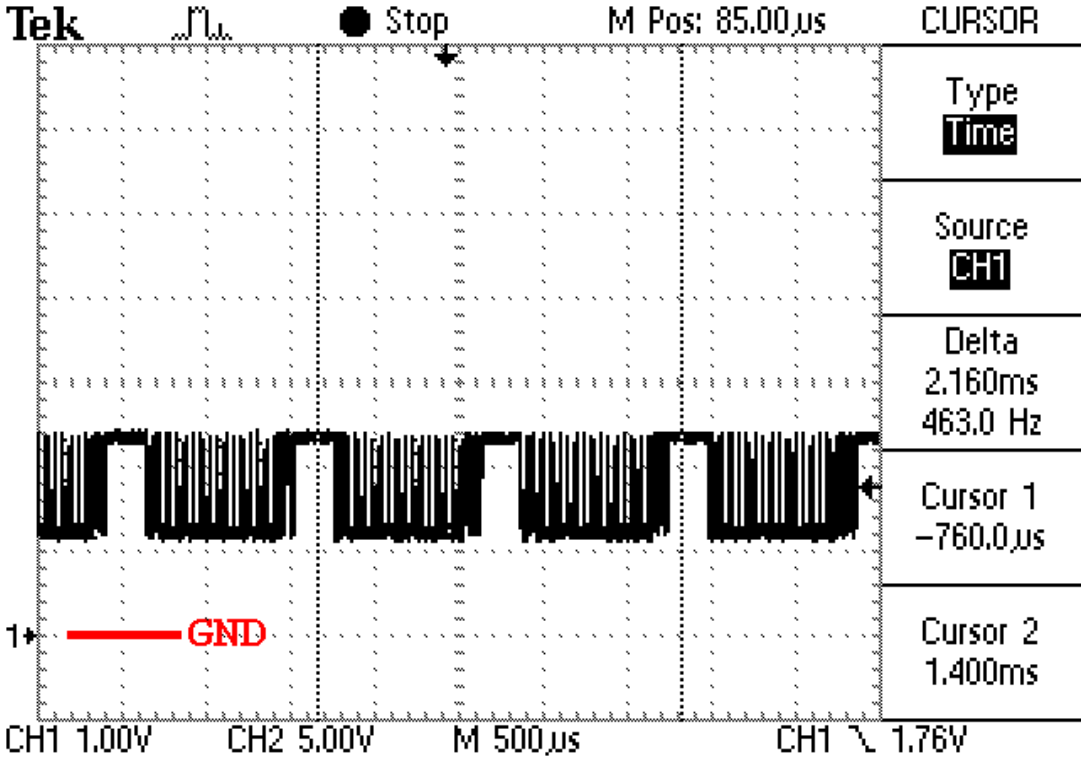
What's happening?

1. Just as in the figure above, three '0' are transmitted (takes 24 microseconds,) and the next bit is misinterpreted so the transmitter thinks it has lost the arbitration.
2. The transmitter waits for 6 bits and then detects a stuff error. The misinterpreted bit and the 6 bits takes 56 microseconds.
3. The transmitter and the receiver now start to transmit an error frame. It's 6 dominant bits (48 microseconds.)
4. The nodes that transmitted the error frame now waits for 8 recessive bits but as the rising slope is bad, the first bit is misinterpreted. The nodes will think this is another node transmitting an error frame and will ignore it.
5. When the bus is back at the recessive level all nodes wait for 8 bits.
6. Then the intermission of 3 recessive bits comes.
7. 3+9 = 12 bits = 96 microseconds, as seen in the picture.
8. Then the transmitter tries again with the same result. After a while the transmitter goes error passive and will behave as described earlier.

Here is yet another picture. In this setup there is only a single node on the (properly terminated) CAN bus. It's trying to transmit a message, but no one is listening.

So what's happening?

1. First, the transmitter sends the whole message.
2. The transmitter expects a dominant level in the ACK slot, but as no one is listening, no ACK arrives, so the transmitter detects an Acknowledgement error.
3. The transmitter then transmits a passive error flag (passive because in the picture above it has been trying to send for a few seconds, so it's no longer error active.)
4. The passive error flag is followed by an error delimiter and the intermission.
5. Since this node tried to send a message but failed, it has to wait for another 8 bits before initiating a new transmission. This is called 'suspend transmission' in the CAN spec.
6. The transmitting node also has to increase its tx error counter by 8, but by a special case in the CAN spec, this happens only as long as the transmitter is error active. When the transmitter goes error passive it will not (in this case) increase its tx error counter, and consequently the transmission is retried forever.



So what you see in the picture above is a message being transmitted, followed by a small pause which is the sum of the error flag, the error delimiter, the intermission and the suspend transmission. The message is then retransmitted and retransmitted and ...

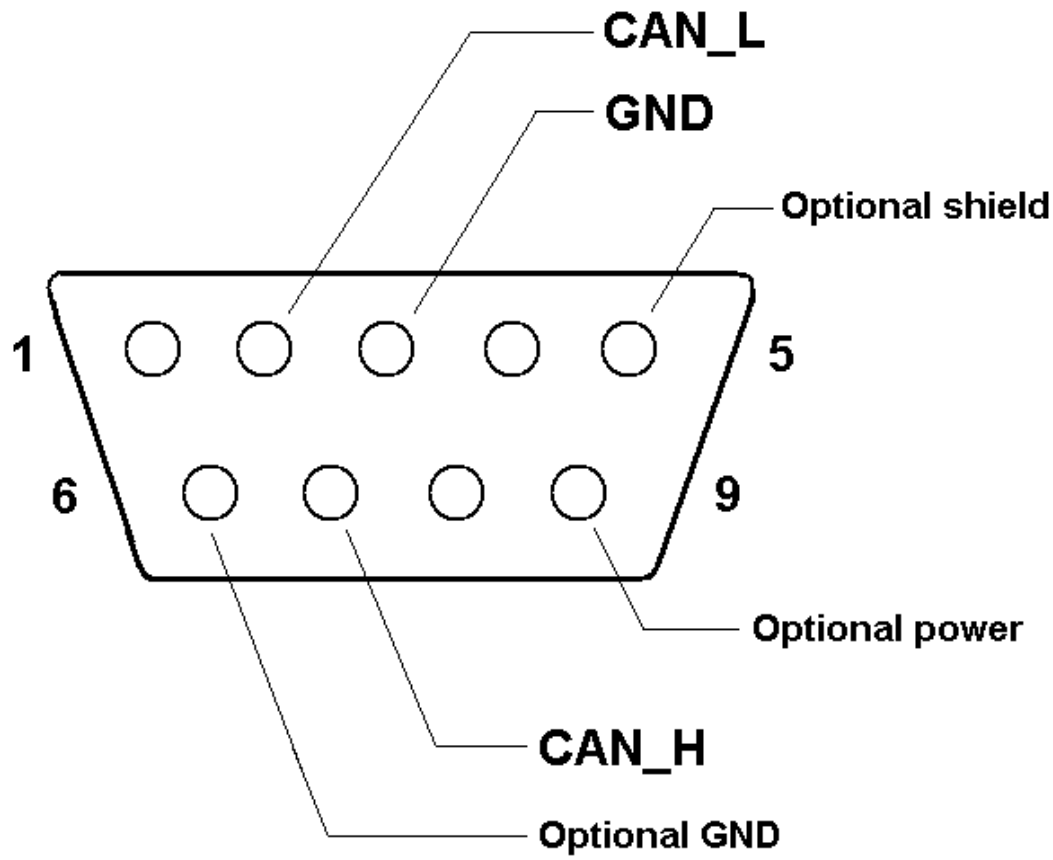
CAN Connectors

9-pin DSUB

This connector layout is recommended by **CiA (https://www.kvaser.com/can-automation-cia/)** and is pretty much *the* industrial standard.

If power is supplied, it shall be in the range +7..+13 V, 100 mA. Modules provide a male connector and have to connect pin 3 and 6 internally.

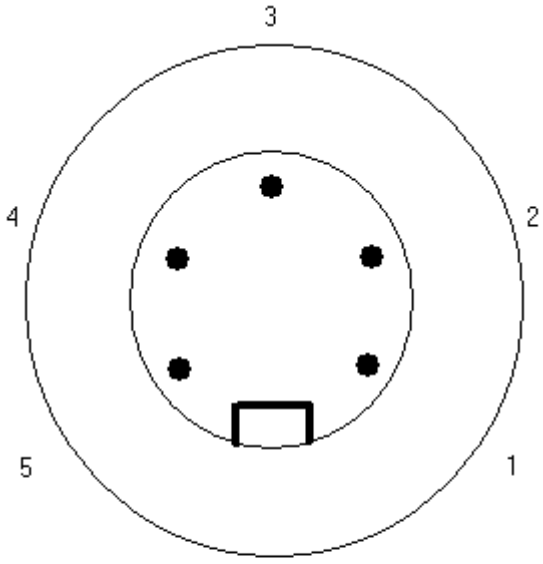
The pin numbering is valid for a male connector, viewed from the connector side, or for a female connector viewed from the soldering side. – To memorize the pinning, note that CAN_LOW has a LOW pin number and CAN_HIGH has a HIGH pin number.



This is a male connector viewed from the connector side, or a female connector viewed from the soldering side.

1	-	Reserved
2	CAN_L	CAN_L bus line (dominant low)
3	CAN_GND	CAN Ground
4	-	Reserved
5	(CAN_SHLD)	**Optional** CAN shield
6	(GND)	**Optional** CAN ground
7	CAN_H	CAN_H bus line (dominant high)
8	-	Reserved (error line)
9	CAN_V+	**Optional** power

5-pin Mini-C



Male (pins)

Pin	Function	DeviceNet Color
1	Drain	Bare
2	V+	Red
3	V-	Black
4	CAN_H	White
5	CAN_L	Blue

Pin	Function	Recommended cable colour
1	Power negative	Black
2	CAN_H	White
3	Optional Signal GND	Yellow
4	Optional Initiate	Gray
5	Power positive	Red
6	CAN_L	Blue

Feedback

Used by both **DeviceNet (https://www.kvaser.com/about-can/higher-layer-protocols/devicenet/)** and **SDS (http://www.honeywell.com/sensing/prodinfo/sds)**and happens to be compatible between these two protocols.

The modules have male connectors. The supplied power is 24V +- 1%.

Note: in the DeviceNet specification version 1.x, the female connector in figure 9.13 has the numbers in wrong order. Specification 2.0 and later versions has got it right.

6-pin Deutsch DT04-6P

Recommended by CANHUG for use in mobile hydraulics applications.

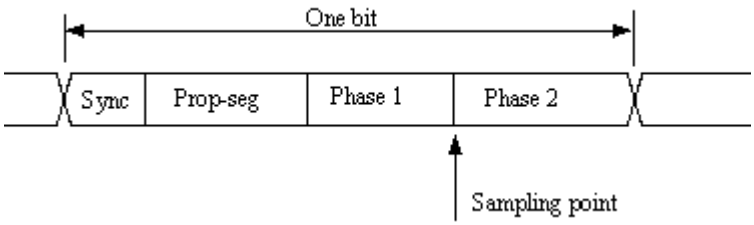
Module side male, bus side female. There is currently no recommendation as to the supplied power.

CAN Bit Timing

Layout of a Bit

Each bit on the CAN bus is, for timing purposes, divided into at least 4 quanta. The quanta are logically divided into four groups or segments:

- the Synchronization Segment
- the Propagation Segment
- the Phase Segment 1
- the Phase Segment 2



Here is a picture of a CAN data bit.

The Synchronization Segment, which always is one quantum long, is used for synchronization of the clocks. A bit edge is expected to take place here when the data changes on the bus.

The Propagation Segment is needed to compensate for the delay in the bus lines.

(https://www.kvaser.com/support/calculators/bit-timing-calculator/)

Helpful Tools

Bit Timing Calculator

(https://www.kvaser.com/support/calculators/bit-timing-calculator/)

Calculate all possible sets of CAN bus parameters for a given input frequency and a given bus speed.

(https://www.kvaser.com/support/calculators/bit-timing-calculator/)

The Phase Segments may be shortened (Phase Segment 1) or lengthened (Phase Segment 2) if necessary to keep the clocks in sync. The bus levels are sampled at the border between Phase Segment 1 and Phase Segment 2.

Most CAN controllers also provide an option to sample three times during a bit. In this case, the sampling occurs on the borders of the two quanta that precedes the sampling point, and the result is subject to majority decoding (at least this is the case for the 82527).

Clock Synchronization

In order to adjust the on-chip bus clock, the CAN controller may shorten or prolong the length of a bit by an integral number of quanta. The maximum value of these bit time adjustments are termed the Synchronization Jump Width, SJW.

Hard synchronization occurs on the recessive-to-dominant transition of the start bit. The bit time is restarted from that edge.

Resynchronization occurs when a bit edge doesn't occur within the Synchronization Segment in a message. One of the Phase Segments are shortened or lengthened with an amount that depends on the phase error in the signal; the maximum amount that may be used is determined by the Synchronization Jump Width parameter.

Bit Timing Register Calculation

Most CAN controllers allows the programmer to set the bit timing using the following parameters:

- A clock prescaler value
- The number of quanta before the sampling point
- The number of quanta after the sampling point

- The number of quanta in the *Synchronization Jump Width*, SJW

Usually two registers are provided for this purpose: btr0 and btr1. Things tend to vary slightly between different controllers, however, so read your data sheets carefully.

On the 82c200 and SJA1000, both from NXP (previously Philips), the register layout is like this:

	7	6	5	4	3	2	1	0
btr0	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
btr1	SAM	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10

- BRP0..BRP5 sets the clock prescaler value
- SJW0..SJW1 sets the length of the SJW
- TSEG10..TSEG13 sets the number of quanta before the sampling point (the start bit is not included)
- TSEG20..TSEG22 sets the number of quanta after the sampling point.
- SAM is set to 1 if three samples is to be obtained and to 0 if one sample is enough.

Note: the actual value of these parameters is one more than the value written into the register.

Example: if the oscillator signal fed to the SJA1000 is 16 MHz, and we want a bit rate of 250 kbit/s, with a sampling point close to 62% of the whole bit, and a SJW of 2 quanta, we can set:

BRP = 4, which gives a quantum length of 2 * 4 / 16000000 s = 500 ns, and
TSEG1 = 5, which gives 5 quanta before the sampling point, and
TSEG2 = 3, which gives 3 quanta after the sampling point.

- BRP0..BRP5 sets the clock prescaler value
- SJW0..SJW1 sets the length of the SJW
- TSEG10..TSEG13 sets the number of quanta before the sampling point (the start bit is not included)
- TSEG20..TSEG22 sets the number of quanta after the sampling point.
- SAM is set to 1 if three samples is to be obtained and to 0 if one sample is enough.

Note: the actual value of these parameters is one more than the value written into the register.

Example: if the oscillator signal fed to the SJA1000 is 16 MHz, and we want a bit rate of 250 kbit/s, with a sampling point close to 62% of the whole bit, and a SJW of 2 quanta, we can set:

BRP = 4, which gives a quantum length of 2 * 4 / 16000000 s = 500 ns, and
TSEG1 = 5, which gives 5 quanta before the sampling point, and
TSEG2 = 3, which gives 3 quanta after the sampling point.

Each bit will then comprise 5 + 3 = 8 quanta, which results in the desired bit rate of 1 / (8 * 500 ns) = 250 kbit/s. The register values should then be as is shown in the example here.

The sampling point is at 5/8 = 62.5% of a bit.

btr0 =	(SJW – 1) * 64 + (BRP -1) = (2-1)*64 + (4-1) = 67 = **0×43**
btr1 =	SAM * 128 + (TSEG2 – 1)* 16 + (TSEG1 – 1) = 0*128 + (3-1)*16 + (4-1) = *(“4” because the start bit isn’t included)* 35 = **0×23**

CAN Error Handling

How CAN Handles Errors

Error handling is built into in the CAN protocol and is of great importance for the performance of a CAN system. The error handling aims at detecting errors in messages appearing on the CAN bus, so that the transmitter can retransmit an erroneous message. Every CAN controller along a bus will try to detect errors within a message. If an error is found, the discovering node will transmit an Error Flag, thus destroying the bus traffic. The other nodes will detect the error caused by the Error Flag (if they haven’t already detected the original error) and take appropriate action, i.e. discard the current message.

Each node maintains two error counters: the Transmit Error Counter and the Receive Error Counter. There are several rules governing how these counters are incremented and/or decremented. In essence, a transmitter detecting a fault increments its Transmit Error Counter faster than the listening nodes will increment their Receive Error Counter. This is because there is a good chance that it is the transmitter who is at fault! When any Error Counter raises over a certain value, the node will first become “error passive”, that is, it will not actively destroy the bus traffic when it detects an error, and then “bus off”, which means that the node doesn’t participate in the bus traffic at all.

Using the error counters, a CAN node can not only detect faults but also perform error confinement.

Error Detection Mechanisms

The CAN protocol defines no less than five different ways of detecting errors. Two of these works at the bit level, and the other three at the message level.

1. Bit Monitoring.
2. Bit Stuffing.
3. Frame Check.
4. Acknowledgement Check.
5. Cyclic Redundancy Check.

Bit Monitoring
Each transmitter on the CAN bus monitors (i.e. reads back) the transmitted signal level. If the bit level actually read differs from the one transmitted, a Bit Error is signaled. (No bit error is raised during the arbitration process.)

Bit Stuffing
When five consecutive bits of the same level have been transmitted by a node, it will add a sixth bit of the opposite level to the outgoing bit stream. The receivers will remove this extra bit. This is done to avoid excessive DC components on the bus, but it also gives the receivers an extra opportunity to detect errors: if more than five consecutive bits of the same level occurs on the bus, a Stuff Error is signaled.

Frame check
Some parts of the CAN message have a fixed format, i.e. the standard defines exactly what levels must occur and when. (Those parts are the CRC Delimiter, ACK Delimiter, End of Frame, and also the Intermission, but there are some extra special error checking rules for that.) If a CAN controller detects an invalid value in one of these fixed fields, a Form Error is signaled.

Acknowledgement Check
All nodes on the bus that correctly receives a message (regardless of their being “interested” of its contents or not) are expected to send a dominant level in the so-called Acknowledgement Slot in the message. The transmitter will transmit a recessive level here. If the transmitter can’t detect a dominant level in the ACK slot, an Acknowledgement Error is signaled.

Cyclic Redundancy Check
Each message features a 15-bit Cyclic Redundancy Checksum (CRC), and any node that detects a different CRC in the message than what it has calculated itself will signal an *CRC Error*.

Error Confinement Mechanisms

Every CAN controller along a bus will try to detect the errors outlined above within each message. If an error is found, the discovering node will transmit an Error Flag, thus destroying the bus traffic. The other nodes will detect the error caused by the Error Flag (if they haven't already detected the original error) and take appropriate action, i.e. discard the current message.

Each node maintains two error counters: the Transmit Error Counter and the Receive Error Counter. There are several rules governing how these counters are incremented and/or decremented. In essence, a transmitter detecting a fault increments its Transmit Error Counter faster than the listening nodes will increment their Receive Error Counter. This is because there is a good chance that it is the transmitter who is at fault!

A node starts out in Error Active mode. When any one of the two Error Counters raises above 127, the node will enter a state known as Error Passive and when the Transmit Error Counter raises above 255, the node will enter the Bus Off state.

- An Error Active node will transmit Active Error Flags when it detects errors.
- An Error Passive node will transmit Passive Error Flags when it detects errors.
- A node which is Bus Off will not transmit anything on the bus at all.

The rules for increasing and decreasing the error counters are somewhat complex, but the principle is simple: transmit errors give 8 error points, and receive errors give 1 error point. Correctly transmitted and/or received messages causes the counter(s) to decrease.

Example (slightly simplified): Let's assume that node A on a bus has a bad day. Whenever A tries to transmit a message, it fails (for whatever reason). Each time this happens, it increases its Transmit Error Counter by 8 and transmits an Active Error Flag. Then it will attempt to retransmit the message.. and the same thing happens.

When the Transmit Error Counter raises above 127 (i.e. after 16 attempts), node A goes Error Passive. The difference is that it will now transmit Passive Error Flags on the bus. A Passive Error Flag comprises 6 recessive bits, and will not destroy other bus traffic – so the other nodes will not hear A complaining about bus errors. However, A continues to increase its Transmit Error Counter. When it raises above 255, node A finally gives in and goes Bus Off.

What does the other nodes think about node A? – For every active error flag that A transmitted, the other nodes will increase their Receive Error Counters by 1. By the time that A goes Bus Off, the other nodes will have a count in their Receive Error Counters that is well below the limit for Error Passive, i.e. 127. This count will decrease by one for every correctly received message. However, node A will stay bus off.

Most CAN controllers will provide status bits (and corresponding interrupts) for two states:

- "Error Warning" – one or both error counters are above 96
- Bus Off, as described above.

Some – but not all! – controllers also provide a bit for the Error Passive state. A few controllers also provide direct access to the error counters.

The CAN controller's habit of automatically retransmitting messages when errors have occurred can be annoying at times. There is at least one controller on the market (the SJA1000 from Philips) that allows for full manual control of the error handling.

Bus Failure Modes

The ISO 11898 standard enumerates several failure modes of the CAN bus cable:

1. CAN_H interrupted
2. CAN_L interrupted
3. CAN_H shorted to battery voltage
4. CAN_L shorted to ground
5. CAN_H shorted to ground
6. CAN_L shorted to battery voltage
7. CAN_L shorted to CAN_H wire
8. CAN_H and CAN_L interrupted at the same location
9. Loss of connection to termination network

For failures 1-6 and 9, it is “recommended” that the bus survives with a reduced S/N ratio, and in case of failure 8, that the resulting subsystem survives. For failure 7, it is “optional” to survive with a reduced S/N ratio.

In practice, a CAN system using 82C250-type transceivers will not survive failures 1-7, and may or may not survive failures 8-9.

There are “fault-tolerant” drivers, like the TJA1053, that can handle all failures though. Normally you pay for this fault tolerance with a restricted maximum speed; for the TJA1053 it is 125 kbit/s.

Higher Layer Protocols

The CAN standard defines the hardware (“the physical layer” – there are several) and the communication on a basic level (“the data link layer”). The CAN protocol itself just specifies how to transport small packets of data from point A to point B using a shared communications medium. But in order to manage the communication within a system, a higher layer protocol (HLP) is required.

Higher Layer Protocols include common standards like J1939, CANopen, CCP/XCP, and more.

Visit our section on Higher Layer Protocols.

Get Started (https://www.kvaser.com/about-can/higher-layer-protocols/)

PHONE

+46 31 886344 (tel:+46 31 886344)

EMAIL

sales@kvaser.com (mailto:sales@kvaser.com)

HOW CAN WE HELP?

Technical Support (https://www.kvaser.com/support/)
Developer (https://www.kvaser.com/developer/)
Contact Us (https://www.kvaser.com/contact-us/)
Where To Buy (https://www.kvaser.com/products-services/where-to-buy/)

ABOUT US

About Us (https://www.kvaser.com/about-us/)
News (https://www.kvaser.com/about-us/news/)
Career (https://career.kvaser.com/)
University Sponsorship (https://www.kvaser.com/about-us/university-sponsorships/)
Technical Associates (https://www.kvaser.com/products-services/technical-associates/)

POLICIES

Warranty (US) (https://www.kvaser.com/about-us/policies/general-conditions-for-us-market/)
Warranty (Global) (https://www.kvaser.com/about-us/policies/general-conditions-for-rest-of-world/)
NFND / EOL (https://www.kvaser.com/about-us/policies/product-nfnd-eol-policy/)
Recycling (https://www.kvaser.com/about-us/policies/recycling-policy/)
Privacy (https://www.iubenda.com/privacy-policy/48266154)
Prop 65 WARNING (https://www.kvaser.com/prop-65-warning/)
General Compliance (https://www.kvaser.com/general-compliance-page)

SOCIAL

Click here (https://www.kvaser.com/account/?newsletter) to Subscribe to our Newsletter