

# CAN Primer: Creating your own Network

Spring 2009 Version 1.57

by Robert Boys, bob.boys@arm.com San Jose, California



## Introduction:

CAN is extensively used in automobiles and trucks but has found applications everywhere. There are many “application” layers available for CAN such as ISO 15765 (cars), J1939 (trucks) and CANopen (factory automation) but it is very easy to develop your own protocol that will fit and simplify your needs. Modern CAN transceivers provide a stable and reliable CAN physical environment without the need for expensive coaxial cables. Most of the mystery of CAN has dissipated over the years. There is plenty of example CAN software available to help you quickly develop your own network.

A CAN controller is a sophisticated device. Nearly all the features of the CAN protocol described below are automatically handled by the controller with almost no intervention by the host processor. All you need to do is configure the controller by writing to its registers, write data to the controller and the controller then does all the housekeeping work to get your message on the bus. The controller will also read any frames it sees on the bus and hold them in a small FIFO memory. It will notify the host processor that this data is available which you then read from the controller. The controller also contains a hardware filter mechanism that can be programmed to ignore those CAN frames you do not want passed to the processor.

## Main Features of CAN:

For the purposes of this article; we will assume a CAN network consists of the physical layer (the voltages and the wires) and a frame consisting of an ID and a varying number of data bytes. CAN has the following general attributes:

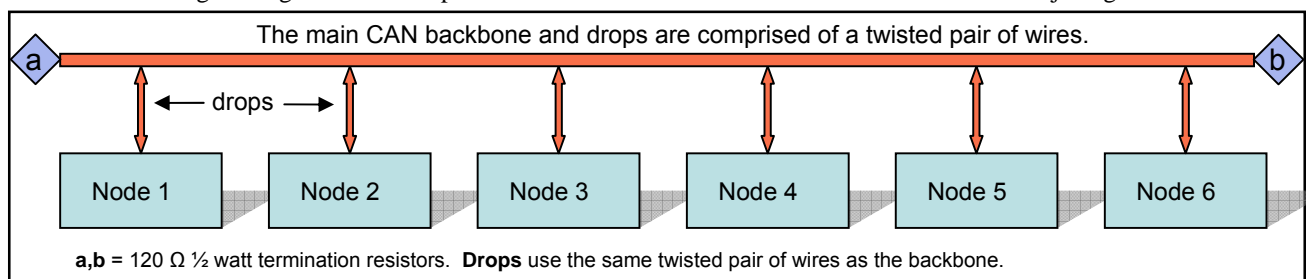
1. 11 or 29 bit ID and from zero to 8 data bytes. **TIP:** These can be dynamically changed “on the fly”.
2. Peer to Peer network. Every node can see all messages from all other nodes. A node can’t read its own messages.
3. Nodes are really easy to add. Just attach one to the network with two wires plus a ground.
4. Higher priority messages are sent first depending on the value of the ID. A lower ID has the higher priority.
5. Automatic retransmission of defective frames. A node will “buss-off” if it causes too many errors.
6. Speeds from approximately 10 Kbps to 1 Mbps. **TIP:** All nodes *must* operate at the same frequency.
7. The twisted differential pair provides excellent noise immunity and some decent bus fault protection.
8. The CAN system will work with the ground connection at different DC levels. **TIP:** Or no ground at all.

## The CAN System Layout:

A CAN network consists of at least two nodes connected together with a twisted pair of wires as shown below. A ground wire can be included with the twisted pair or separately as part of the chassis. One twist per inch (or more) will suffice and the integrity of the ground is not important for normal operation. As in any differential systems; the important signal is the voltage levels *between* the wire pair and not their values to ground. CAN is completely described in ISO 11898.

The maximum length of the network is dependent on the frequency, number of nodes and propagation speed of the wire. It is relatively easy to have a 20 node (or more), 500 Kbps system running 30 or 40 feet (or more). **TIP:** The drops should be less than 3 feet and randomly spaced to reduce standing waves. These issues all become more important at higher bus speeds.

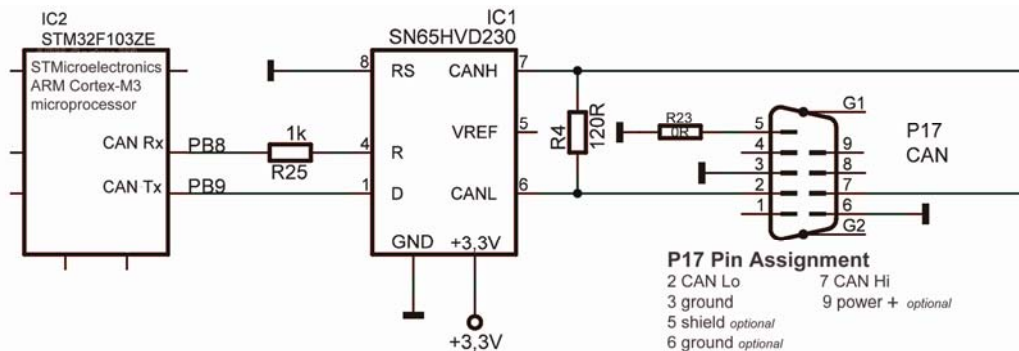
Since the twisted pair is a transmission line, 120 ohm termination resistors are needed at both ends of the backbone. Do not put any resistors at the nodes. **TIP:** Your total resistance value as measured between the two twisted wires will be 60 ohms. CAN is a broadcast system. Any node can “broadcast” a message using a CAN frame on a bus that is in idle mode. Every node will see this message. A “message” can be considered the same as a CAN frame until you need to use more than one frame to send a long message. **TIP:** It is up to the individual node if it must react to a CAN frame or just ignore it.



## A Node Schematic:

This is the schematic diagram from the Keil MCBSTM32E™ evaluation board. IC1 is a Texas Instruments CAN transceiver which performs the conversion between the single-ended CAN controller CAN Tx and CAN Rx signals to the bi-directional differential pair of the CAN bus called CANH and CANL (High and Low). This schematic is complete. The STM32 CAN I/O is TTL, CMOS and 5 volt tolerant, all at the same time making it exceptionally easy to design the interface.

This transceiver IC1 connects to the STM32 microprocessor IC2 which contains an integral CAN controller via two pins: D (Driver input) and R (Receiver output). The corresponding nomenclature on the STM32 is CAN Rx and CAN Tx. CAN Tx connects to D. CAN Rx connects to R. It is that simple. Some processors have multiple CAN controllers. These are usually used in routers, gateways or to create more receiver FIFO memory for intentionally slowed down CPUs (for EMI reasons). For general use a node normally needs only one controller. If it had at least two, it could talk to itself.



RS on IC1 (slope control) is used to adjust the rise and fall times of the output edges to limit EMI from the twisted pair.

Note R4, a 120 ohm termination resistor. This evaluation board is meant to be used with one other board as a small test network. If this board is used as a node, and is not at one of the ends, this resistor should be removed and external resistors used. P17 corresponds to a generally accepted standard for CAN on DB9 connectors. P17 Pin 7 is the CAN Hi bus line and pin 6 is CAN Lo. **TIP:** If CAN Hi and CAN Lo are reversed, the network will not operate properly. It might not work at all.

## Physical Layer: *the wires and the voltages...*

There are three physical layers used in CAN: Hi-Speed, Fault Tolerant and Single Wire. Hi-Speed is the most common and is the only one we will use in this article. Fault Tolerant offers more robustness as its name implies and is used more often in European autos. Single Wire is used by General Motors as a low speed body network along with a Hi-Speed main network.

Hi-Speed in cars has a speed of 500 Kbps, trucks are 250 Kbps. CANopen runs up to 1 Mbps. Fault Tolerant is usually 125 Kbps and GM Single Wire is normally 33.33 Kbps. **TIP:** 1 Mbps in a large system is difficult to handle. 500Kbps is easier.

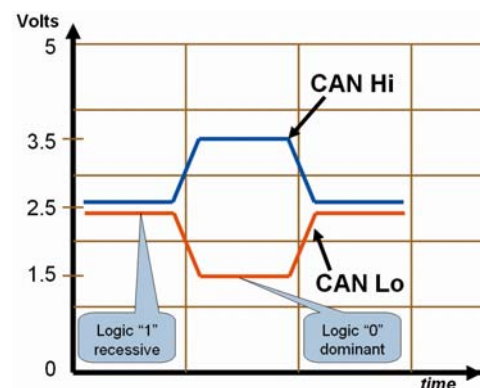
To change from one to the other requires only the transceiver chip be exchanged and probably changing the speed. These three flavors cannot be physically connected to each other as the voltage levels are different. You need to use a router or gateway to join different CAN networks together. Any CAN controller will properly service all three flavors of CAN.

The Hi-speed CAN physical layer is merely a twisted pair of wires with a 120 ohm termination resistor at each end and twisted wire drops to the individual CAN nodes. You can connect your node directly to the bus.

CAN Hi voltage with respect to ground changes between 2.5 to 4 volts nominal. CAN Lo changes from 2.5 to 1 volt. Therefore the difference between the two is either 0 volts (is logical "1") or 2 volts (is logical "0"). 0 is known as the "recessive" state and 2 volts is the "dominant" state.

These two signals, CAN Hi and CAN Lo, are 180 degrees out of phase as indicated in this diagram. Bus idle is when the voltage difference is zero.

**TIP:** How to determine the frequency of a CAN signal: This is the best and sometimes only way to determine this.

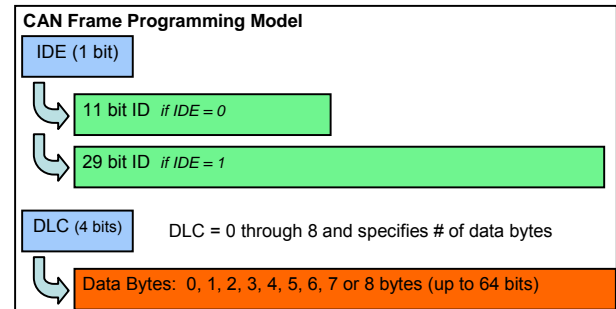


1. Connect an oscilloscope hot lead to CAN Hi and its ground to CAN Lo. The scope ground must be isolated from the CAN ground. You can go from ground to one of the CAN leads but the signals will be lower and noisier.
2. Display a trace. You might need a storage scope to see just one trace due to the non-repetitive nature of CAN.
3. Pick the smallest width signal pulse and measure its time period in seconds as accurately as you can.
4. Invert this value (divide into 1) and you have the CAN speed in bits per second.

## The CAN Frame:

The CAN frame has many fields but we can simplify this to a Programming Model as shown. These fields are those that your software must write to or read from the CAN controller registers. The CAN configuration registers are not included.

- **IDE:** Identifier Extension: 1 bit - specifies if the ID field is 11 or 29 bits  
If IDE = 0, then the ID is 11 bits.  
If IDE = 1, then the ID is 29 bits.
- **DLC:** Data Length Code: 4 bits - specifies number of data bytes in frame from 0 through 8.  
DLC = 0 through 8 and specifies # of data bytes
- **ID:** Identifier: 11 or 29 bits as set by IDE.  
This part of the CAN frame sets the priority.
- **Data Bytes:** 0 through 8 bytes. **TIP:** A CAN frame with only an ID field and no data bytes is valid and useful.



### ID: Identifier: 11 or 29 bits

The Identifier can be used for any purpose. Often it is used as a node address or to identify requests and responses. CAN does not specify what the ID should be. 11 bit is sometimes called Standard CAN and 29 bit is called Extended CAN.

1. If two or more CAN messages are put on the bus at the same time; the one with the highest priority (the lowest value) ID will immediately get through. The others will be delayed and will be resent as soon as possible.
2. An ID of 0 has the highest priority and will always get through. An 11 bit ID has priority over any 29 bit ID.
3. You can change the ID size at any time for a mix of 11 and 29 bit IDs. The controller can easily sort this out.
4. Messages tend to start transmitting at the same time. It is not permissible for a CAN node to start transmitting if another is already transmitting. This will cause a bus error. CAN controllers will not make this mistake.
5. **TIP:** Note that CAN controllers can be configured to pass only certain messages to its host processor. Choose your ID values carefully to take advantage of this if needed. This can take a large work load off a node's processor.
6. You can use the ID for data, node addressing, commands and request/response sequences. Commercial protocols use any of these in practice. You can choose any method or create your own that best suits your purpose.
7. **TIP:** Make sure two nodes will *never* send the same ID value at the same time. It is illegal but possible to do this. If two messages sent at the same time are identical, they will be seen as one. If the data bytes are different, this will result in a bus error and the frames will be resent continuously. This creates havoc on the bus until bus-off occurs.

### Data Bytes:

You can select from 0 to 8 data bytes using the 4 bit DLC field.

1. You can have any number of data bytes mixed on the CAN bus. The controller can easily sort this out.
2. **TIP:** If you always use only one number of data bytes, your software will be much simpler to write and maintain.
3. The data bytes can contain anything. It will not be prioritized like the ID is. CAN does not specify data contents.
4. Commercial protocols such as J1939 use these for data as well as control bits for multi-frame transmission schemes.

### Remote Frames:

These are not used much but are worth mentioning. A remote frame is a quick method of getting a response from another node(s). It is a request for data. The requesting node sends out a shortened CAN frame with only a user specified ID number and the number of data bytes it expects to receive (the DLC is set). No data field is sent. The responding node(s) sees this ID and DLC, recognizes that it has the desired information and sends back a standard CAN frame with the same ID, DLC and with its data bytes attached. All of this (except that the response node recognizes the ID and DLC) is implemented in the CAN controller hardware. Everything else must be configured by the user software.

**Other Bit Fields:** Only the ACK bit will be mentioned in this document:

**ACK:** Is a 1 bit field in the CAN frame created by the transmitting node but set by all the other nodes.

**TIP:** The number one reason people can't get their CAN node working is you need at least two nodes to work. When a node puts a message on the bus, it will wait for the ACK bit to be asserted by any other node that has seen the message and determines it to be valid. If so, the transmitting node finishes the message and goes into the idle state or sends its next message. If not, it will immediately resend the message forever until the ACK is inserted or the controller is RESET. This transmitting node will never go into bus-off mode. Note that a standard CAN test tool will usually act as a second node.

**TIP:** This presents an excellent opportunity to provide an easy test situation. If you can't get your network to work try this. It won't tell you if your frequency, ID or data bytes are correct, but it will tell you if you are putting out something.

1. Connect your CAN node. You must have a transceiver connected to the CAN controller with a termination resistor.
2. Do not connect any other node or test tool. Just one node running by itself with at least one 120 ohm resistor.
3. Connect an oscilloscope hot lead to CAN Hi and ground to CAN Lo. The scope ground must be isolated from the CAN ground. You do not need a high speed scope – almost any will suffice. You can also connect a scope to the CAN controller output pin and ground. If you connect from CAN Hi or Lo and ground, your signal will be smaller.
4. Configure your CAN controller and write the IDE, ID, DLC and any data bytes into the appropriate register.
5. Any CAN frames will now be continuously displayed on the scope. RESET the processor to start over.

**TIP:** You can measure the CAN frequency with the method described in the TIP: under Physical Layer.

## Bus Loading:

Many CAN networks work on a bus loading from 15 to 35 % and this is increasing. A higher bus loading can cause lower priority messages to be delayed but these messages will still get through in a timely fashion. It is quite difficult to achieve 100% bus loading although one can come quite close. Overall system performance does not drop greatly at high bus loading.

**TIP:** It is possible to get very high bus loads for a very short period of time in any CAN network. CAN does not automatically space out messages. It is possible to get a series of back-to-back messages that will equal nearly 100 % bus loading. You should be prepared for this. One solution is to select only those messages needed by a node by programming its acceptance filter. Another is to have your software space out the messages. This problem is quite hard to diagnose.

## Bus Speed:

Bus speed in a system is a balancing act between things such as propagation delays (from bus length) and EMI emissions versus necessary data throughput. Run your network as fast as possible for stable operation and with enough throughput. Do not run it faster than it needs to be, but make some room for later expansion.

**TIP:** If your network is not stable: make sure you have two good termination resistors at each end of the network. Try slowing the CAN speed down to see if this helps. Resistors can be ordinary 120 ohm ½ watt carbon type. This is not critical.

## Bus Errors:

Recall we said that all the nodes (including the transmitting node) checks each CAN frame for errors. If an error is detected, here is what happens:

1. Any or all the nodes will signify this fact by driving the bus to logical 0 (dominant state) for at least 6 CAN bits.
2. This violates the Bit Stuffing rule (never greater than 5 bits the same polarity) so every node sees this as an error.
3. This so called “Error Frame” signals to all nodes a serious error has occurred if they don't already know it.
4. The transmitting bus abandons the current frame and adds 4 to its 8 bit TEC register. (transmit error counter)
5. IF this TEC equals 0xFF, the transmitting node goes BUS OFF and takes itself off the bus. (it is zero at RESET)
6. IF not, it attempts to retransmit its message. It will still have to go through the priority process with other messages.
7. All other nodes also abandon reading the current frame, and adds 4 to each REC register. (receive error counter)
8. Any nodes that have messages queued up for transmission will transmit now. All others start listening to the bus.
9. Hopefully, this time the message(s) will be broadcast and received error free. Each time a frame is transmitted and/or received successfully, the corresponding TEC and REC registers are decremented (usually by only 1)

**Super TIP: Error Counters ?** These are two 8 bit registers in every CAN controller and you can read these with your software. This is a good idea because it gives some indication of general bus health and stability. In a good CAN network, TEC and REC will equal 0. If it starts having higher values, something has happened to your network. The usual suspect is bad hardware. The problem is usually in either the wires or the transceiver chip.

**TIP:** Don't forget that if something happens to the integrity of your twisted pair, such as CAN Lo disconnected; it might still work but with greatly reduced noise immunity (that is what differential signals do best). If your network is in a very noisy environment, there might be a lot more transient bus errors. This is very tricky to debug without knowledge of the REC and TEC register contents. Read TEC and REC with your software and report it to your diagnostic routines.

In a general sense, TEC represents a given node's errors and REC indicates the other nodes' errors.

**Bus Off:** As mentioned, if a transmitting node detects it has put too many bad frames on the bus, it will disconnect itself. It will assume that there is something very wrong with itself. To get back on the bus depends on how you configure the controller. It can take a controller RESET or a certain number of good frames received or what you configure to get back on.

## BUS Faults:

This is different (sort of) from a bus error. We normally think of a bus fault as something that has happened to the “wires” or the output transistors of the transceiver chip. Not all bus faults will result in a bus error. A bus error can be thought as the CAN controllers’ reaction to a problem on the bus such as noise, faulty node that includes a bus fault.

What happens if one of the twisted pair opens or is shorted out? CAN has an automatic mechanisms for this. Not all transceiver chips implement all of them. You can usually short CAN Lo to ground (ISO 11898 says can short Hi also) or open one CAN line. The ground needs to be connected for these to function. You can’t short both Hi and Lo together (Fault Tolerant will work) or open both up. You can cut the ground or have a large ground loop present and CAN will still work.

These will be detected as a bus error as described above. At least one node must try to transmit a frame in a bus fault condition to trigger a bus error. A bus in idle mode can’t trigger a bus error. When the bus fault is removed, in many systems the network will come back alive if so configured. CAN has excellent noise immunity because of the twisted pair. The common mode noise gets cancelled out and the CAN signal is not affected at all (because it is 180 degrees out of phase).

**The Ground:** Strictly speaking, the ground is not needed for CAN operation if the twisted pair is intact. This is readily shown with simple experiments. One experiment showed a small network still worked properly with two nodes having a 40 volts DC ground difference! However, it is a good idea to include a good ground in your system design. Some bus faults need the ground to allow the transceiver to compensate.

**TIP:** How can you create a Bus Error for testing? Easy: have a node send a message at the wrong frequency. When this frame tries to get on the bus this is certain to create a bus error condition. Some CAN controllers can send a one-shot frame.

**Bonus TIPS:** Here some items not part of the CAN specification but might prove helpful in your system:

### 1) Transmitting data sets greater than 8 bytes:

Clearly, transmitting a data set greater than 8 bytes will take multiple frames and this will require some planning. Such schemes can become very complicated as they have to deal with a wide-ranging set of contingencies. If you can focus on a narrow requirement set, design of a simpler protocol is possible.

Most current schemes use the first data byte to contain the number of total data bytes to follow plus a counter to help determine which data byte is which. The ID usually identifies the node plus whether is the request or response message. If you want to use an existing protocol see ISO 15765. This is what automobiles use. This includes OBDII diagnostics which is public information. This is a good example where one message can be comprised of many CAN frames.

### 2) Periodic, Request/Response and Command Frames:

**Periodic:** This technique sends a frame out periodically – several times a second is usual. This frame will contain data that any node can use if it wants to and is identified by its ID. Examples are speed, position, pressure and events.

**Request/Response:** A node sends out a frame requesting certain specified information. Any other nodes that have the requested information then put it on the bus. The ID identifies the Request frame and the Response by changing one bit of the Request ID. Examples are ID 0x248 is a Request frame and 0x648 is its Response. The Request frame data bytes will specify what information is requested. The Response frame will contain the requested information or an error message.

**Command:** A frame commanding some event be performed. The ID usually contains the address of the commanded node and the data bytes the actual command(s). Sometimes an Acknowledge frame is returned.

**TIP:** You might want to consider a blend of these three types of traffic depending on your system’s needs.

### 3) Time-outs:

Automotive CAN networks use time-outs and this concept is easily and effectively transferred to systems in other fields. A time-out occurs when a node fails to respond to a request in a timely fashion. Time-outs are handled completely by software. The CAN specification does not provide this mechanism. A time-out is helpful to recover from problems with the network such as severe bus errors, catastrophic bus faults, faulty nodes or intermittent connections.

The result is usually a limp-home mode where a node will attempt to run itself without information from the rest of the network. In some cases, a punitive limp-home mode is entered that forces the user to perform repairs.

A good example is if the transmission fails and proper shifting becomes impossible. In this case, the module will go into limp-home mode and the transmission might be put into one gear such as second to allow the vehicle to still be driven. This can be for safety reasons or to prevent further damage to the power train.

**Heart-beats and Address Claiming:** The other side to a time-out is a heart beat. Periodic messages can be sent out to determine that all nodes are on the bus and active. CANopen uses such heart-beats. J1939 has a software mechanism where each node can declare itself to be on the bus and be recognized by the other nodes. This is called “Address Claiming” and occurs during the system startup. None of these mechanisms are provided by the CAN specification but by your software.



## Sequence of Transmitting Data on the CAN Bus:

1. Any node(s), seeing the bus idle for the required minimum time, can start sending a CAN frame.
2. All other nodes start receiving it except those also starting to transmit a message. (they all start at the same time)
3. If any other node starts transmitting: the priority process starts – the node with the highest priority continues on and those with a lesser priority stops sending, immediately turns into a receiver and receives the higher priority message.
4. At this point, only one node is transmitting a message and no other will start at this time.
5. When the transmitting node has completed sending its message, it waits one bit time for the 1 bit ACK field to be pulled to a logic 0 by any other node (or usually all of them) to signify the frame was received without errors.
6. If this happens, the transmitting node assumes the message reached its recipient, sends the end-of-frame bits and goes into receive mode or starts to send its next message if it has one. The receiving nodes pass the received message to their host processors for processing unless the acceptance filtering prevents this action.
7. At this time, any node can start sending any messages or the bus goes into the idle state. Go to 1.
8. If this does not happen (ACK bit not set) then the transmitting retransmits the message at the earliest time allowed. If the ACK bit is never set, the transmitting node will send this message forever.

### Transmitting Notes:

- **How does a node know when it should transmit a message ?** Easy – you create the CAN frame you want to send by loading up the IDE, ID, DLC and any data byte registers in the CAN controller and then, in most controllers, you set a bit that triggers sending the frame as soon as legally possible. After this, the controller takes care of sending all frame bits. Until the controller signals otherwise to the processor, you can assume the message was sent.
- **What if there is an error ?** All nodes, including the transmitting node, monitor the bus for any errors. If a error condition is detected – a node or nodes signify to the other nodes there is an error by holding the bus at logical 0 for at least 6 bus cycles. At this point, all nodes take appropriate action. The message being sent (and now aborted) will be resent but only for a certain number of times. See Bus Errors, TEC and REC registers on page 4.
- **What if no node wants or uses the message ?** Nothing. The ACK bit only says that the CAN frame was transmitted without errors and at least one node saw this frame error free. Remember the transmitting frame can't ACK itself. CAN does not provide any acknowledgment mechanism that a frame was used or not by its intended recipient. If needed, you will have to provide this in your software as many systems do.

**TIP:** In a periodic system, if a node misses a message, it doesn't matter much as another copy will be along shortly.

## Sequence of Receiving data from the CAN Bus:

1. All nodes except those currently transmitting frames are in listening mode.
2. A CAN frame is sent using the procedure as described above: Sequence of Transmitting data on the CAN Bus:
3. This frame is received by all listening nodes. If deemed to be a valid CAN message with no errors – the ACK bit is set. In CAN terminology, this set to the “dominant” state as opposed to the recessive state.
4. The frame is sent through the controller's acceptance filter mechanism. If this frame is rejected it is discarded. If accepted it is sent to the controller FIFO memory. If the FIFO is full, the oldest frame is lost.
5. The host processor is alerted to the fact a valid frame is ready to be read from the FIFO. This is done either by an interrupt or a bit set in a controller register. This frame must be read as soon as possible.
6. The host processor decides what to do with this message as determined by your software.

### Receiving Notes:

- **TIP:** You must decide whether to use polling or interrupts to alert the host processor a frame is available. Polling is where the host processor “polls” or continuously tests the bit mentioned in # 5. Polling runs the risk of losing or “dropping” a frame but is sometimes easier to implement and debug. Interrupts cause the processor to jump to an interrupt handler where the frame is read from the controller. Using interrupts is the recommended method.
- **What happens if a message is “dropped” ?** This can cause some problems as CAN itself does not have a mechanism for acknowledging a CAN frame. If you want this, you must add it to your software. In the case of Periodic Messages, it doesn't normally matter much as a replacement message will be along shortly.
- **How fast do I have to read the FIFO to not drop messages ?** It depends on the CAN speed, frame size, and bus loading. It is a good idea to read these frames as soon as possible since once a frame is dropped, it will not be automatically recovered or resent by the transmitting node. It is gone forever unless you provide a suitable mechanism in your software to have it resent.

## CAN Controllers and their Errata Sheets:

As mentioned before, CAN controllers are very sophisticated modules. Many times someone is experiencing trouble getting something to work or has an unexpected crash or result and they desperately search their code for the error causing this. Sometimes the answer lies in the errata sheet and not in your software. This document that lists all known deviant behaviour from that claimed in the device datasheet. Some CAN controllers do have bugs and you should find out what they are.

Note that technical support staff statistics show that most errors are in the user software code so check this carefully.

You should get all the latest errata sheets and read them. You can potentially save an enormous amount of time. Sometimes the weirdest problems are caused by these defects. And then, of course, you might have to be prepared for the day these bugs get fixed and show up in real silicon on your board. Most issues will be in the controllers and not the simpler transceivers.

**TIP:** There are several Internet CAN newsgroups and mailing lists that can help you with your network. Remember that not all people on these groups are experts and there is some risk of getting poor information. Fortunately, these people are in the minority. See <http://groups.yahoo.com/group/CANbus> and [www.vector-informatik.com/canlist/](http://www.vector-informatik.com/canlist/).

## Test Tools:

The biggest problem in getting your first CAN network running is that in order to see some messages, you have to have both a receiving node and a transmitting node properly working *at the same time*. This can be quite the onerous job. There are two ways to help here. One is to use a working node such as an evaluation board with proven CAN examples provided. You can attempt to receive these known good CAN frames with your node. See the last section in this document for an example.

Second, you can purchase a CAN test tool. This is the best idea. These provide both sending and receiving capabilities and act as a CAN node. There are two types: simple low cost devices that provide basic creating and displaying bus traffic and those offering advanced capabilities.

Typical sources for inexpensive tools are SYS TEC ([www.phytec.com](http://www.phytec.com)), [www.kvaser.com](http://www.kvaser.com) and PEAK [www.peak-system.com](http://www.peak-system.com) which is also sold in the USA through [www.phytec.com](http://www.phytec.com). There are many other companies that sell these types of inexpensive tools. Search on the Internet to find these.

If you are developing a more capable and powerful CAN system, you might want to consider a CAN analyzer. These offer very advanced features such as triggering, filtering and best of all; a database where your ID and data bytes are displayed in words rather than raw hex numbers. This will save a lot of time and make for a better, more reliable product. Typical suppliers are Dearborn Group [www.dgtech.com](http://www.dgtech.com), Vector CANalyzer [www.vector.com](http://www.vector.com), National Instruments [www.ni.com](http://www.ni.com) and Intrepid [www.intrepidcs.com](http://www.intrepidcs.com). Do not be afraid to use an automotive type device even if your application is something else. CAN is CAN no matter where it is used and no matter what anybody says. Everything else sits on top of CAN.

## Bit Stuffing:

The CAN protocol states that when there are 5 consecutive bits of the same polarity, one bit of opposite polarity will be inserted to maintain the counter accuracy. These bits make the CAN frame longer and are very common. These bits are inserted and removed automatically by the CAN controller and are only visible when an oscilloscope is attached to the bus.

**TIP:** When bits are added (or not) to the CAN frame as various messages are sent on the bus, the changing frame length will look like jitter on the bus. It is not jitter of course; CAN just works this way. Just something to be aware of.

## Conclusion:

You now have enough CAN theory to enable you to develop and troubleshoot a small CAN network.

Now, on the next few pages, let us look at how we can program a real CAN controller to transmit and receive messages. There are some hands-on experiments you can try – the Keil evaluation software is free and for one experiment no hardware is needed. For the other, you will need an evaluation board with a STM32 processor and a Keil ULINK2 or ULINK-ME USB to JTAG adapter.

For more information relating to CAN please see <http://dgtech.com/pdfs/techpapers/primer.pdf>

For information regarding testing CAN networks: [http://dgtech.com/pdfs/techpapers/CIA\\_article.pdf](http://dgtech.com/pdfs/techpapers/CIA_article.pdf)

**A differential twisted pair of wires with two 120  $\Omega$  termination resistors: This is the minimum network of 2 nodes.**



## CAN Demonstration Software:

In order to experiment with a CAN network it is useful to try a simulator before the real hardware. This document shows how to use the complete device simulation included in the Keil® Microcontroller Development Kit (MDK-ARM) for the STM32 ARM® Cortex™-M3 microcontroller. No hardware is needed.

You can download the latest version of MDK-ARM at: <http://www.keil.com/update/rvmdk.asp>

There is no charge for this software. Please install this software on your PC.

Keil also provides simulation software for NXP ARM processor-based devices and various 8051 processors that have CAN controllers. Luminary processors are supported but by using the Luminary evaluation boards with either the ULINK® 2 or ULINK-ME adapters or connection to the Luminary USB on-board JTAG adapter.

Complete technical information on the ST CAN module is found in the Reference Manual RM0008 available from [www.st.com/stm32](http://www.st.com/stm32). Other manufacturers have similar documentation available on their websites.

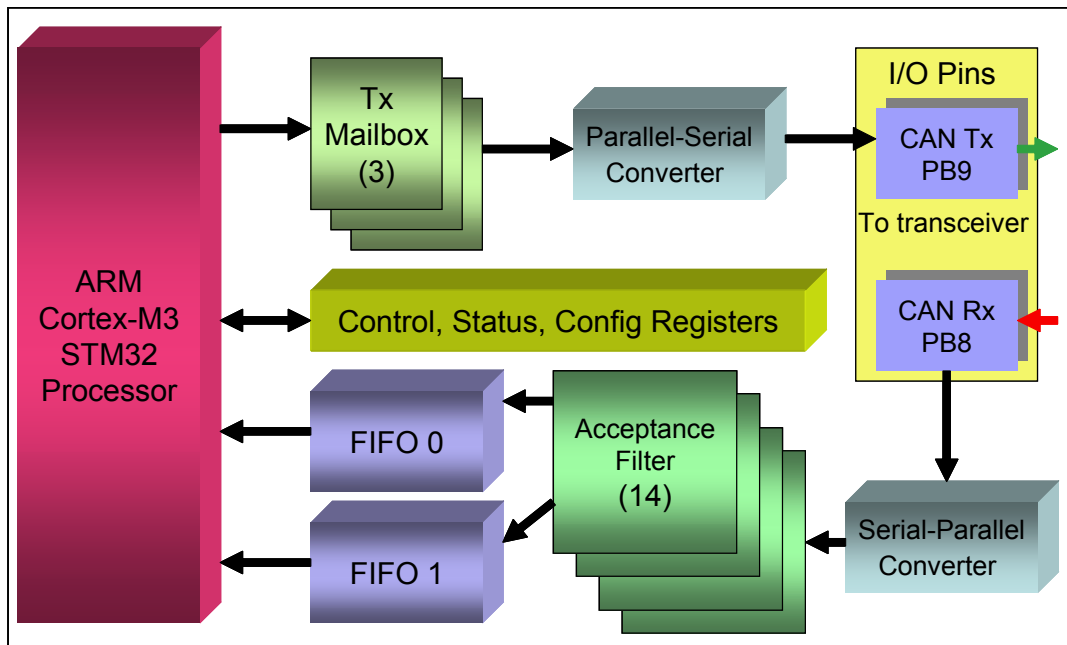
## STMicroelectronics CAN Controller for Cortex-M3 Processors.

Shown is a block diagram of the CAN controller. Here are the main points of all CAN controllers:

1. I/O Pins: These connect to the CAN transceiver chip pins R and D as already described.
2. Parallel-Serial Converters: CAN is a serial bus while the processor is parallel. Conversion happens here.
3. Tx mailbox: The messages to be transmitted are written here. ID, data (if any) and the DLC go here.
4. Acceptance Filter: This passes only specified messages to the processor via the FIFOs. By default at RESET, these filters pass all messages to the FIFOs. Your software must configure them to filter messages.
5. FIFO 0 & 1: Each Receive FIFO can hold 3 CAN messages. They provide a buffering system to the processor.
6. Control, Status, Configuration registers: Your software must configure these registers, usually at initialization. Various flags and switches are found here. Examples are set CAN speed, request transmission, manage receive messages, enable interrupts and obtain diagnostic information. Keil provides examples on how to set and use these registers.

All CAN controllers have the same basic architecture. Different controllers will have differences in the number of receive FIFO buffers, transmit buffers, size of acceptance filters and the bit mapping, addresses and definitions of the various configuration registers. All CAN controllers are licensed by Robert Bosch GmbH in Germany and therefore they are able to exert considerable control over basic CAN attributes to make them consistent with various manufacturers.

This means that all CAN controllers can communicate with other brands in a reliable and predictable manner.









## Keil Example CAN Program:

1. Start µVision® by clicking on its icon on your Desktop.





2. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\Keil\MCBSTM32\CAN\CAN.Uv2.
3. Make sure “Simulator” is selected in the Target window. 
4. There is a typo in a source file. In the file CanDemo.c, go to line 126.  
It will probably be: `delay (45000000); // Wait for initial display (~5s)`  
This will be much too long. Please change this to a lower value. 45000 works good.
5. Compile the source files by clicking on the Build icon. . They will compile with no errors or warnings.
6. Click on the Options for Target icon.  Then, select the Debug tab and confirm “Use Simulator” is checked.
7. Enter the Debug mode by clicking on the debug icon.  Select OK when the Evaluation Mode box appears.
8. Position the Toolbox, CAN: Communication and CAN: Controller windows as appropriate.
9. Click on the RUN icon.  Note: you can stop the program with the STOP icon. 
10. Note CAN messages with an ID of 0x21 will appear in the CAN: Communications window. You can see both the transmit and receive frames. The CAN controller is in a special Test Mode that allows it to see its own messages.
11. In the Toolbox window, click on the “Analog Sweep 0...3.3v” button.
12. Changing data values representing output from the A/D convertor will now appear in the CAN messages.

## The Keil CAN Demonstration Software: How it works...

Keil provides a working CAN example with their development tools. You have already compiled and ran this example. You can view and edit the C source files whether in debug mode or not, but to compile them you must not be in debug mode. This example uses almost no assembly code as it is (nearly) entirely written in C. Any source file can be opened in µVision if not already visible by clicking on File/Open and selecting it. There are three source files we will look at:

**Can.h:** This file defines a structure to contain the information used to construct the CAN frame and create two instances of it.

**Can.c:** This C code initializes the CAN controller, writes and transmits a message, receives a message, configures the Acceptance Filters and provide the transmit and receive interrupt handlers.

**CanDemo.c:** The main function is located in this file. CanDemo.c is the heart of the demonstration program and calls the functions in Can.C.

### 1) Can.h

#### The CAN Structure CAN\_Msg: (lines 23-29, 42 & 43)

Shown is the structure declaration in Can.h. You should now be able to recognize each of these elements. You can enter either an 11 or 29 bit identifier. Two instances of CAN.msg are invoked and are shown below: CAN\_TxMsg and CAN\_RxMsg. CanDemo.c writes to these to create the CAN messages with data.

The prototypes for functions used in Can.c are listed in Can.h in lines 32 to 40. These are visible in µVision.

```

23  typedef struct  {
24      unsigned int  id;                // 29 bit identifier
25      unsigned char data[8];          // Data field
26      unsigned char len;              // Length of data field in bytes
27      unsigned char format;           // 0 - STANDARD, 1- EXTENDED IDENTIFIER
28      unsigned char type;             // 0 - DATA FRAME, 1 - REMOTE FRAME
29  } CAN_msg;

42  extern CAN_msg    CAN_TxMsg;        // CAN message for sending
43  extern CAN_msg    CAN_RxMsg;        // CAN message for receiving

```

### 2) Can.c

#### Configuring the CAN Controller: (Can.C)

There are several things that must be done to properly configure the CAN controller. These are done in Can.C by functions that are called by CanDemo.c. Examples are found in the function CAN\_setup (lines 28 to 58) as shown in µVision:

1. Enable and set the clock for the CAN controller. **TIP:** The clock must be stable for CAN. No R-C oscillators here.
2. Configure GPIO ports PB8 and PB9 for the transmit and receive lines to the transceiver chip.
3. Enable the interrupts for the transmit and receive functions.
4. Set **CAN\_BTR**: This is a 32 CAN controller register where things such as bit timing, bus frequency, sample point and silent and loop back modes are set. In the Keil example, the baudrate is set to 500 Kbps (bits per second).

**TIP:** Sometimes timing settings can cause strange problems. If you experience some unusual problems you might want to study CAN timing in greater detail. For small systems, the default settings or those suggested by the processor manufacturer will work satisfactorily. You can adjust these settings for the most robust bus performance.

All CAN controllers have the same general settings for bit timing because of the licensing agreements with Robert Bosch GmbH. For a detailed explanation of CAN bit timing see [www.port.de/pdf/CAN\\_Bit\\_Timing.pdf](http://www.port.de/pdf/CAN_Bit_Timing.pdf) and for the calculations see page 505 of the ST Reference Manual RM0008.

**TIP:** All CAN controllers on a network should have consistent BTR values for stable operation.

### Other Functions in Can.c:

- CAN\_start: Starts the CAN controller by ending the initialization sequence.
- CAN\_waitReady: Waits until transmit mailbox is ready – then can add another message to be transmitted.
- CAN\_wrMsg: Write a message to the CAN controller and transmit it.
- CAN\_rdMsg: Read a message from the CAN controller and releases it to be sent to the STM32 processor.
- CAN\_wrFilter: Configure the acceptance filter. This is not discussed in this article.
- USB\_HP\_CAN\_TX\_IRQHandler: The transmit interrupt handler.
- USB\_LP\_CAN\_RX0\_IRQHandler: The receive interrupt handler.

These functions are called by CanDemo.c and in the main function.

### 3) CanDemo.c

This contains the main function and contains the example program that reads the voltage on the A/D converter and sends its value as a CAN data byte with an 11 bit ID of 0x21. CanDemo.c contains functions to configure and read the A/D converter, display the A/D values on the LCD and call the functions that initialize the CAN controller.

#### Transmitting a CAN Message:

Lines 131 to 135 puts the frame values into the structure CAN\_TxMsg. (Except for the data byte from the A/D converter.)

```
131     CAN_TxMsg.id = 33;                      // initialise message to send
132     for (i = 0; i < 8; i++) CAN_TxMsg.data[i] = 0;
133     CAN_TxMsg.len = 1;
134     CAN_TxMsg.format = STANDARD_FORMAT;
135     CAN_TxMsg.type = DATA_FRAME;
```

This CAN message will send one data byte. For example, if you change the value in the member CAN\_TxMsg.len to “3”, three data bytes will be sent on the bus. What data will be in them depends on the contents of the array CAN\_TxMsg.data.

**TIP:** If you send more data bytes than you have data, it is a good idea to fill the empty data bytes with either 0 or 0xFF.

Lines 141 puts the A/D value into the data member CAN\_TxMsg.data in data byte 0 and line 142 transmits it.

```
141     CAN_TxMsg.data[0] = adc_Get ();          // data[0] field = ADC value
142     CAN_wrMsg (&CAN_TxMsg);                 // transmit message
143     val_Tx = CAN_TxMsg.data[0];              // send to LCD screen
```

## Receiving a CAN Message:



Lines 148 to 151 indicate when a CAN message is received. But something more must be going on here. Line 151 shows that the data byte received and inserted in the array[0] is sent to be displayed on the LCD. How exactly does the CAN data byte get into the member array CAN\_RxMsg.data[0] ?

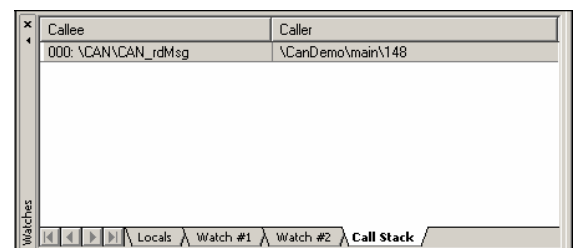
```
148  if (CAN_RxRdy) {
149      CAN_RxRdy = 0;
151      val_Rx = CAN_RxMsg.data[0];
```

Recall we said before that the function to read the CAN data was located in Can.c. If we look in Can.c, we find the function CAN\_rdMsg at lines 130 to 159. Examining it, clearly it is here that the array[0] is indeed loaded here at line 148. But how does this function get called ? It is not called from CanDemo.c.

If we set a breakpoint on Can.c line 132 (the first assembly instruction of the function CAN\_rdMsg) by double-clicking on the left side of Line 132, we can check the Call Stack window to see where it was called from. This is shown in the screen shot below. This would indicate that Line 148 in the main function called CAN\_rdMsg. Examining the assembly code at Line 148 shows this can't be true.

We can use the Trace function of µVision that is available in Simulator mode to figure out how CAN\_rdMsg is called.

1. Click on Enable Trace Recording icon. 
2. Run the program to the breakpoint set previously at Line 132.
3. Click on View Trace Records.  View the Disassembly window that opens up as shown below.



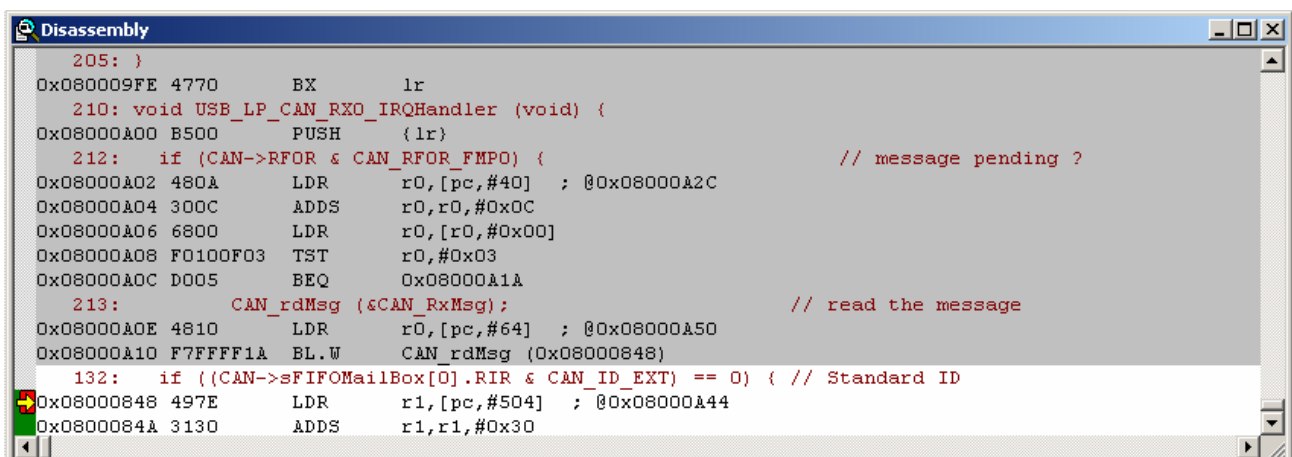
The yellow arrow points to the start of the function CAN\_rdMsg. The arrow represents the program counter.

The grey area shows a recording of the instructions that were executed. The white area displays unexecuted instructions.

Just before Line 132 is the source line **213**: that calls CAN\_rdMsg. This is actually the assembly instruction at address 0x8000A10 BL.W CAN\_rdMsg. Reading higher to **210**:, we can see this source line comes from the function USB\_LP\_CAN\_RX0\_IRQHandler which is the Receive Interrupt Handler in Can.c.

So, this interrupt handler called the function that reads the CAN frame from the CAN controller and inserts it into the structure.

If you right click on Line 210 here and select Show Source at Current Line, this will be displayed in the source file.



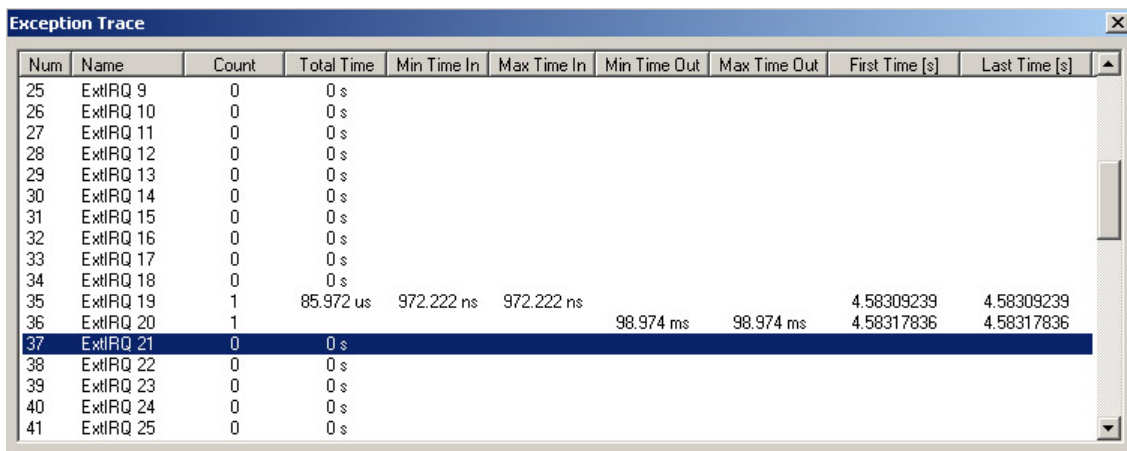
**TIP:** If you have more than one CAN controller in your processor you can operate these as parallel receivers. Divide the messages up with the Acceptance Filters. This will help capture all the messages on a very busy bus without losing any. Each CAN controller will handle its share of the messages. This effectively multiplies the number of FIFO buffer memories which is an excellent method of capturing all the CAN frames.

## Exception, PC and Data Tracing:

The ST Cortex-M3 processor possesses significant debugging capabilities. These features are grouped under the Serial Wire Viewer and are supported by Keil  $\mu$ Vision and the USB-JTAG adapter ULINK2 and ULINK-ME. Recall that Can.c contains two interrupt handlers: for transmit and receive. These can be displayed in real-time (no CPU cycles are stolen) in the Trace Records window as shown below. This works only with a real target Cortex-M3 connected to  $\mu$ Vision with a ULINK2 or ULINK-ME USB to JTAG adapter.

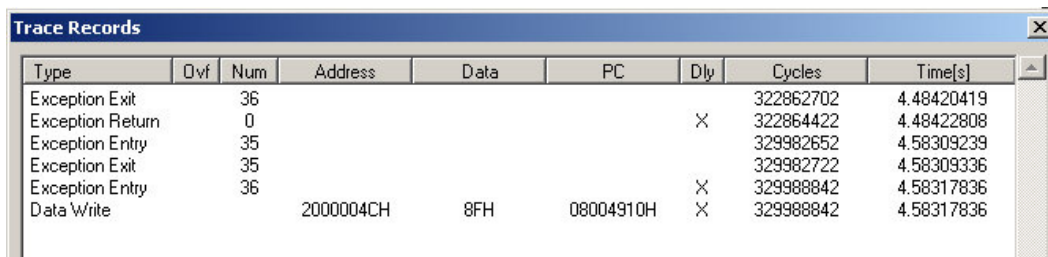
The Serial Wire Viewer can be used to display PC samples, data read and write cycles, exceptions and certain other events. The ITM is a *printf* type instrumentation output. To show our example using the Serial Wire Viewer and a STM32 chip:

1. Connect a STM32 processor to  $\mu$ Vision in and run the same CAN program using the ULINK Cortex debugger.
2. Activate the Serial Wire Viewer as described in the appendix of [www.keil.com/download/files/labst.pdf](http://www.keil.com/download/files/labst.pdf).
3. Configure the Logic Analyzer to display CAN\_RxMsg.data[0]. This will also display it in the Trace Records.
4. Set a breakpoint at the instruction at Line 148 in Can.c. This is the first instruction after the write to the array `CAN_RxMsg.data[0]`. This is needed because the instruction the breakpoint is set on is *not* executed. We want to see and record this write so we must execute this source line.
5. Run the program to the breakpoint. The resulting Exception Trace window is displayed as below.



Num	Name	Count	Total Time	Min Time In	Max Time In	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
25	ExtIRQ 9	0	0 s						
26	ExtIRQ 10	0	0 s						
27	ExtIRQ 11	0	0 s						
28	ExtIRQ 12	0	0 s						
29	ExtIRQ 13	0	0 s						
30	ExtIRQ 14	0	0 s						
31	ExtIRQ 15	0	0 s						
32	ExtIRQ 16	0	0 s						
33	ExtIRQ 17	0	0 s						
34	ExtIRQ 18	0	0 s						
35	ExtIRQ 19	1	85.972 us	972.222 ns	972.222 ns			4.58309239	4.58309239
36	ExtIRQ 20	1				98.974 ms	98.974 ms	4.58317836	4.58317836
37	ExtIRQ 21	0	0 s						
38	ExtIRQ 22	0	0 s						
39	ExtIRQ 23	0	0 s						
40	ExtIRQ 24	0	0 s						
41	ExtIRQ 25	0	0 s						

Shown are two IRQ events; Note an IRQ is a subset of Exceptions. Referring to the ST RM0008 reference manual, IRQ 19 is the CAN Transmit IRQ and IRQ 20 is the Receive IRQ. Below is the Trace Records window that will also be displayed.



Type	Opf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Exception Exit		36					322862702	4.48420419
Exception Return		0				×	322864422	4.48422808
Exception Entry		35					329982652	4.58309239
Exception Exit		35					329982722	4.58309336
Exception Entry		36				×	329988842	4.58317836
Data Write			2000004CH	8FH	08004910H	×	329988842	4.58317836

We can see the following lines:

- 3<sup>rd</sup> and 4<sup>th</sup> Line: **Exception Entry and Exit: 35**: This is the transmit IRQ. Translate the 35 into IRQ19 in the Exception Trace window above.
- 5<sup>th</sup> Line: **Exception Entry: 36**: This is the entry of the receive IRQ 20 – this is the Receive IRQ Handler.
- 6<sup>th</sup> Line: **Data Write**: The value 0x8F is written to address 0c2000004C by the instruction at 0x8004910. You can confirm that these values represent an assembly instruction that is part of source Line 148 and that the address written to is that of `CAN_RxMsg.data[0]`.

That finishes a partial demonstration of the Serial Wire Viewer trace feature of the Cortex-M3 processor. Visit [www.keil.com](http://www.keil.com) for more information concerning the Serial Wire Viewer interface in Cortex-M3 processors.

## How can I learn more about these CAN examples ?

**Easy !** With a hardware board you can generate and receive real CAN messages and connect to other nodes or a CAN test analyzer. The lab for the Keil MCBSTM32™ board has instructions on how to do this. This document has some interesting CAN examples. For instance, you can use the Cortex-M3 Serial Wire Viewer to see the CAN messages and interrupts displayed in real time. You can compile these examples with the evaluation version of the software. Please see [www.keil.com/download/files/labst.pdf](http://www.keil.com/download/files/labst.pdf).

**TIP:** STMicroelectronics supplies a complete software library for all their peripherals in the STM32 family using the Cortex-M3 processor. This includes CAN support. Search [www.st.com/stm32](http://www.st.com/stm32) for STM32F10xFWLib. A zip file UM0427.zip and an application note UM0427.pdf are available. Keil µVision project and source files are included. There is no charge for these libraries. Keil use portions of these libraries in their own examples programs. These are identified by the letters STLIB in the Keil µVision project filename.

Keil makes NXP evaluation boards with CAN examples. Luminary Micro also offers many with CAN controllers that are supported by Luminary and Keil examples.

You now know how CAN works and are familiar with the Keil software and will have no problem getting a real CAN system operating. You have already ran an accurate simulation of a CAN network. If you obtain a real target hardware such as the MCBSTM32 you can connect up to any CAN network and communicate with it.

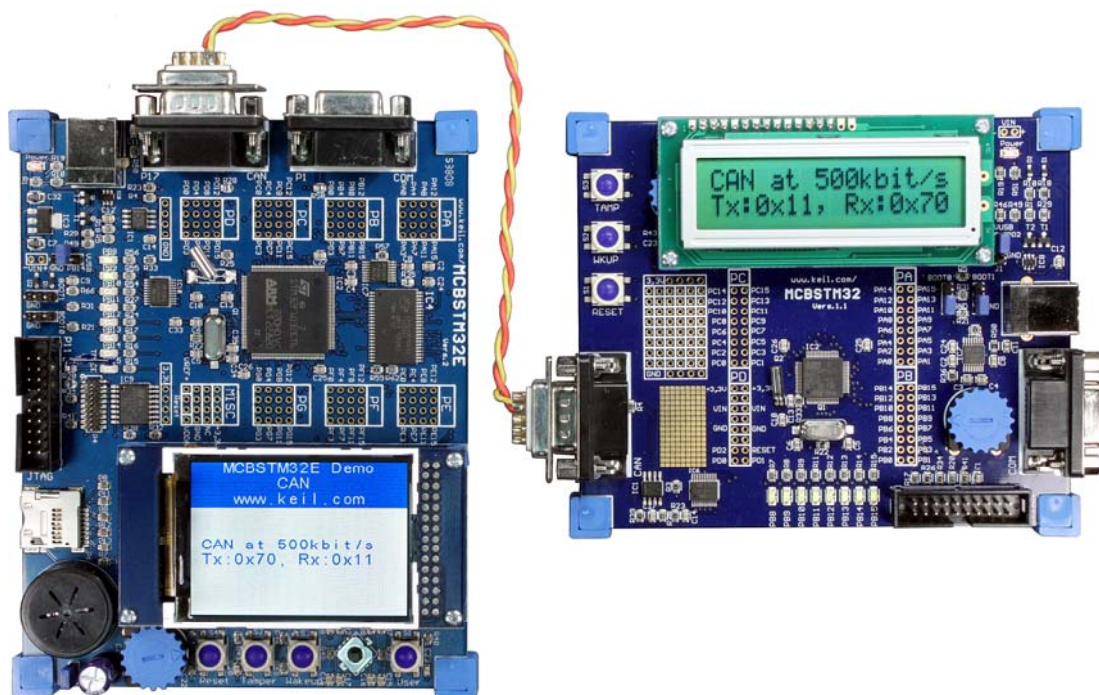
Keil offers a complete CAN stack for all ARM7™, ARM9™ and Cortex-M3/M1 processors. This comes as part of RL-ARM™. Please visit [www.keil.com/rl-arm/](http://www.keil.com/rl-arm/) for more information. This comprehensive package contains the RTX™ RTOS source (the actual RTOS is already included free with the MDK toolset), USB, TCP/IP networking and the CAN interface.

## An Example CAN Network:

Below is a real two node CAN network using Keil MCBSTM32 (right) and the MCBSTM32E evaluation boards using the same example code discussed in this article. The Tx of one node is transmitted to the Rx of the other node and this is clearly seen on the LCDs.

Note the twisted pair of wires CAN Hi and CAN Lo. Note that no ground wire is used in this small network.

This setup is part of the Keil lab available for the STM32: [www.keil.com/download/files/labst.pdf](http://www.keil.com/download/files/labst.pdf) Full details are in the lab.



## For more information

Please contact Keil Sales or Technical Support. In USA: [sales.us@keil.com](mailto:sales.us@keil.com) or [support.us@keil.com](mailto:support.us@keil.com). Outside of the US: [sales.intl@keil.com](mailto:sales.intl@keil.com) or [support.intl@keil.com](mailto:support.intl@keil.com). For comments please email [bob.boys@arm.com](mailto:bob.boys@arm.com).

Comments and constructive criticism will probably result in the creation of documents similar to this one.