

CAN Primer: Creating Your Own Network

ARM® Keil™ MDK™ toolkit featuring Simulator, Serial Wire Viewer and ETM Trace

For the NXP LPC1700 Cortex™-M3

V 2.02 Robert Boys bob.boys@arm.com



Using the Keil Simulator or MCB1700™ Evaluation Board

Introduction:

The latest version of this document is here: www.keil.com/download/docs/385.asp

CAN is extensively used in automotive but it has found applications everywhere. There are many “application” layers available for CAN such as ISO 15765 (cars), J1939 (trucks), DeviceNET and CANopen (both are for 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. Nearly all of the mystery of CAN has dissipated over the years. There is plenty of example CAN software to help you develop your own network.

Many think CAN is just for automotive, but this is not true. CAN *has* become the standard for vehicle networks, but it has been adopted in most other fields. As you find out in these pages, there are no attributes in the Bosch CAN specification that are automotive related. It is completely generic. You can easily implement your own protocol on top of CAN.

A CAN controller is a sophisticated device. Nearly all the features of the CAN protocol described are automatically handled by the controller with almost no intervention by the host processor. All you need to do in practice is to 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. MDK provides sample CAN examples for many ARM processors which you can practice with.

The controller will 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 is then reads it 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.

Modern bus transceiver chips have made the physical CAN bus much less “finicky” and easier to construct and maintain.

The techniques discussed can be applied to many other microprocessors. We use ARM Keil MDK toolkit for the examples. There is no charge for the evaluation version: MDK-Lite™. You can use MDK-Lite for all the CAN examples described. There are many CAN examples in MDK for many boards using ARM processors.

Keil provides a CAN stack as part of MDK-Professional™. Details are on www.keil.com/rl-arm/rl-can.asp

Keil products are listed on the last page of this document.

Shown are evaluation boards using NXP processors that support CAN: There are many more such boards.



How CAN works:	
Introduction:	1
Main Features of CAN:	3
CAN System Layout:	3
CAN Node Schematic:	4
CAN Network with no Transceiver ICs:	4
Physical Layer: the wires and voltages: a real waveform with oscilloscope:	5
The CAN Frame: the Programming Model:	6
Other Bit Fields: Bit Stuffing, Bus loading, Bus Speed:	7
Bus Errors, Bus Faults:	8
Is NOT CAN but useful: Multiple CAN frames, Types of Frames and Time-outs:	9
Sequence of Transmitting CAN Data:	10
Sequence of Receiving CAN Data:	11
CAN FD: A new CAN protocol:	12
CAN Controllers and their Errata Sheets:	13
Test Tools and Software:	13
Keil CAN demonstration software:	14
NXP CAN Controller:	14
CAN Demonstration hands-on Example (using the free Keil Simulator):	15
How the Keil CAN software works:	16-18
Viewing CAN Frames Graphically, Performance Analysis, Execution Profiler:	19
Viewing Program Flow with Instruction Trace:	20
Experimenting with the CAN Software exercises:	21
Getting a CAN Network to run on a real board (Keil MCB1700™):	22
Demonstrating BUSoff and the TXERR Counter:	23
Serial Wire Viewer (SWV) and ETM, CoreSight™ Debug connections:	24
Configuring Serial Wire Viewer (SWV) with ULINK2™, ULINKpro™, J-Link:	25
SWV Exception Tracing:	26
SWV Data Write Tracing:	27
A practical debugging example with SWV Data Write Tracing:	28
Watchpoints:	28
SWV PC Tracing plus Watch and Memory Windows:	29
ETM Tracing Introduction: Embedded Trace Macrocell: ULINKpro:	30
Performance Analyzer <i>with ETM or Simulator</i> :	30
Code Coverage with ETM and Simulator:	31
ETM Trace Triggering:	31
Execution Profiling <i>with ETM or Simulator</i> :	32
In-the-Weeds example with ETM:	33
More Useful Information:	
How to Determine the CAN Frequency:	34
Four Newbie Mistakes YOU can avoid:	34
Useful Documents:	34
How can I learn more about CAN ?	34
How can trace help me find problems?	35
Serial Wire Viewer and ETM Trace Summary:	35
Keil Products and Contact Information:	36

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), a frame consisting of an ID and a varying number of data bytes all with the following general attributes:

1. 11 or 29 bit ID and from zero to 8 data bytes. **TIP:** These attributes can be dynamically changed “on the fly”.
2. Peer to Peer network. Every node can see all messages from all other nodes but it can’t see its own.
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. The lower ID has a higher priority.
5. Automatic retransmission of defective frames. A node will “bus-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 Ground:

This is a contentious issue. A CAN system, especially in vehicles, sometimes must endure large ground loops or corrosion that can compromise signal integrity. CAN is designed using its differential pair to ignore ground voltage differences of many volts. The differential pair also cancels out incoming common mode interference and cancels potential outgoing EMI.

This means that if the ground wire is cut or doesn’t exist, as long as CAN-Hi and CAN-Lo are intact, the system will perform at high performance capabilities. CAN, depending on the transceiver chip, can handle various bus problems such as cut or shorted lines. This capability is lost without the ground. Therefore, it is recommended to always include a ground in your system design. If the ground is made through a chassis connection or negative power supply rail, any shielded CAN cables must have the ground connected at one end only to minimize ground loop problems.

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 described above. As in any differential systems; the important signal is the voltage levels *between* the wire pair and not their values to ground or a voltage supply.

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).

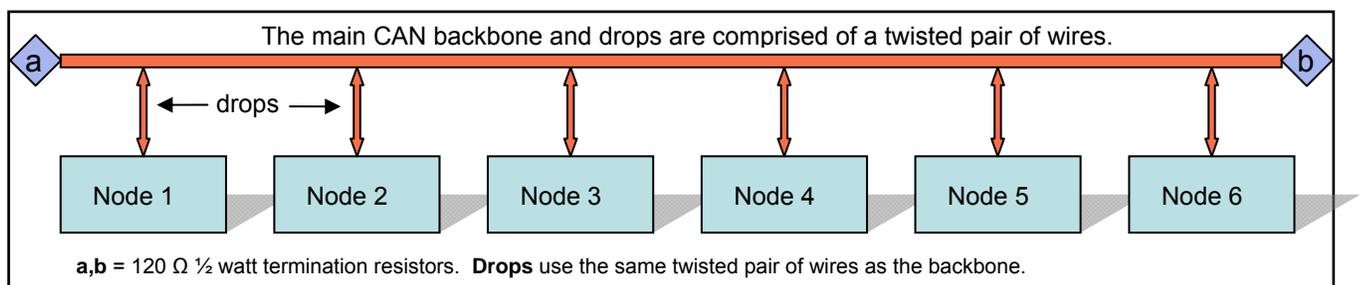
TIP: The drops should be less than 3 feet and randomly spaced to reduce standing waves. These issues all become more important at higher bus speeds. CAN is completely described in ISO 11898.

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 unless a node is at the end of the backbone. Sometimes the resistors are not at the end of a backbone but very close and this seems to work. Resistors are often installed inside an end node.

TIP: Your total resistance value as measured between the two twisted wires will therefore be 60 ohms. 10% is good enough.

CAN is a broadcast system. Any node can “broadcast” a message using a CAN frame on a bus that is in idle mode. Multiple controllers tend to start their messages at the same time. 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. In this case, you would use some sort of a multi-packet protocol.

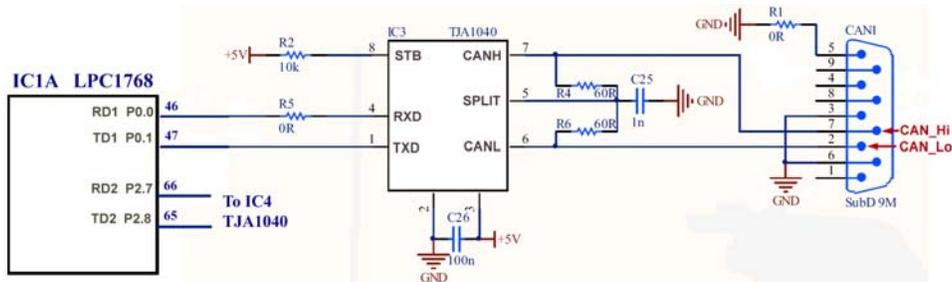
TIP: It is up to a receiving node if it must keep or ignore a frame. This can be handled in either your processor software and/or by configuring the CAN controller acceptance filters.



A Node Schematic:

This is the schematic diagram from the Keil MCB1700 evaluation board. IC3 is a NXP TJA1040 CAN transceiver which performs the conversion between the single-ended CAN controller TD1 and RD1 signals to and from the bi-directional differential pair of the CAN bus CANH and CANL (High and Low). Connecting these parts together is a simple process.

This transceiver IC3 pins TXD (Driver input) and RXD (Receiver output) connect to the NXP LPC1700 series processor IC1A which contains two integral CAN controllers. The corresponding nomenclature on the LPC1700 is TD1 and RD1. TD1 connects to TXD. RD1 connects to RXD. It is that simple. Processors with two or more controllers 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 (this is quite useful for self-diagnostics) or be a gateway between two CAN networks that could operate at different frequencies.



STB on IC3 when high disables the Transceiver. This is useful for low power modes or when the CAN controller is in Bus-off mode. This will effectively remove the node from the bus. **TIP:** You need to use external hardware to set/unset STB.

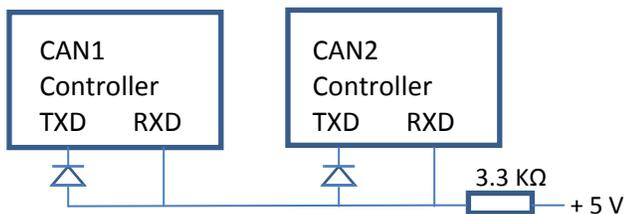
Note R4 and R6: two 60 ohm termination resistors. The TJA1040 has provisions for split termination. Two 60 ohm resistors are used in series with capacitor C25 to offer greater noise immunity; especially at higher CAN speeds.

The MCB1700 evaluation board is meant to be used by itself as a small test network. It contains two separate CAN nodes and connectors. If this board is used as a node with other boards, and is not at one of the ends; the resistors should be removed. Connectors CAN1 and CAN2 correspond to a generally accepted standard for CAN on DB9 connectors. Pin 7 is the CAN Hi bus line and pin 2 is CAN Lo. **TIP:** If CAN Hi and CAN Lo are reversed, the network will not operate.

The controller can see in and out of the bus at the same time. It is normally unable to see its own messages.

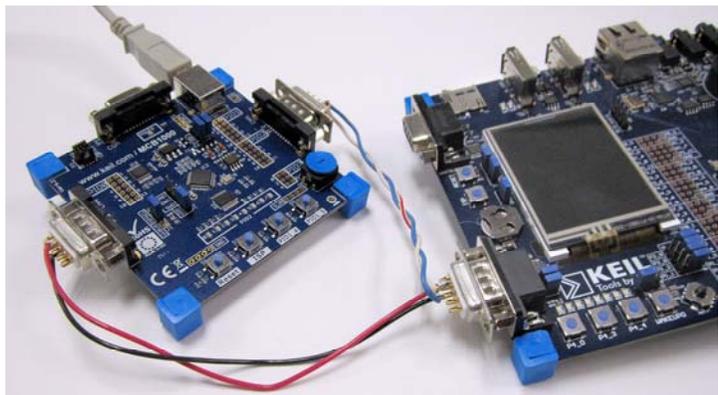
A Tiny Network without Transceiver ICs:

Sometimes you have a CAN equipped processor on a low cost board but it has no CAN transceiver chips. Here is a method that can be used to create a small demonstration network with such a board. There will be no noise immunity and you might have to lower the speed...but many experimenters have made this work satisfactorily. Use a signal diode similar to 1N914 or 1N4148. Power supply diodes usually do not have a fast enough recovery time for CAN to function.



A three node CAN network. Uses two CAN nodes on a LPC1114 Cortex-M3 (Keil MCB1114™) and another on a LPC4350 Cortex-M4/M0 (MCB4300™). The ground is provided through the power supplies. Termination is on the boards.

This network works perfectly even though the wires are a bit sloppy and are hardly a twisted pair. This is because of the network's small size and robustness of CAN in general: even at 500 Kbps.



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 and a few others as a low speed body 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 use and maintain and will present fewer design problems. In general, the longer the physical wires and more nodes, the frequency should be lowered to maintain stability and reduce bus errors.

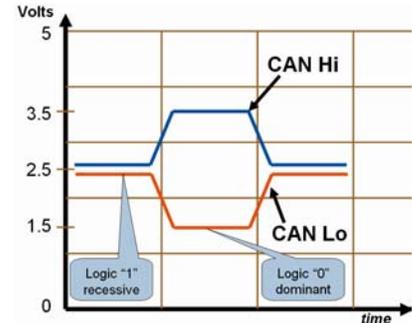
To change from one layer to the other requires only the transceiver chip need be exchanged and probably the speed changed. These three flavors of CAN cannot be physically connected to each other as the voltage levels are different. You need to use a router or gateway to join different CAN networks together.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

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's transceiver chip directly to the bus. It is possible to implement isolation techniques using appropriate devices.

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 volts 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 CAN Hi and CAN Lo voltage difference is zero or close to it.



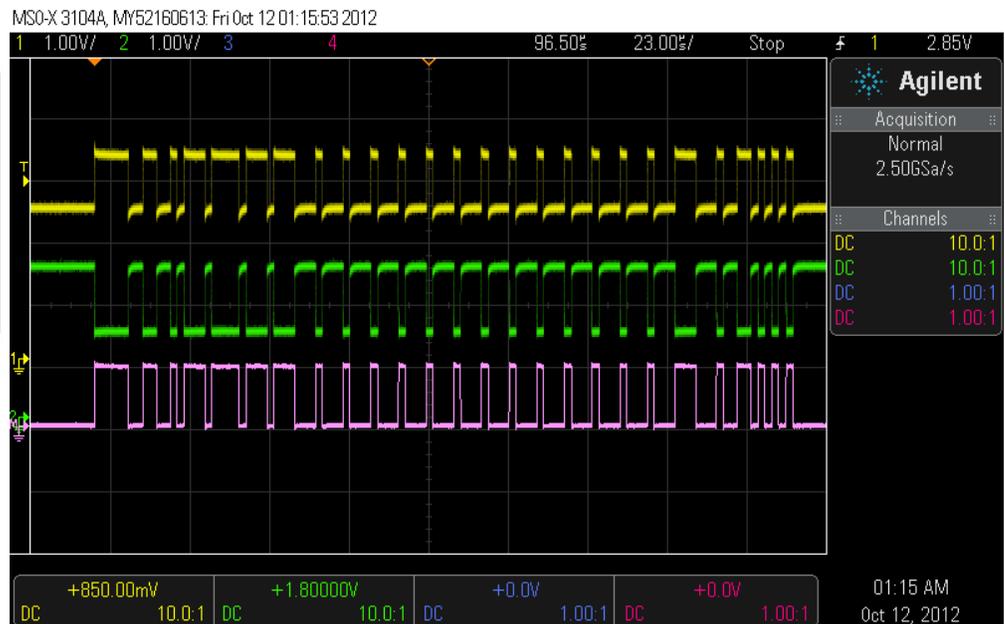
A 2 node CAN cable assembly:

Two wires and two 120 Ω resistors.
A ground connection is preferred.



A CAN frame:

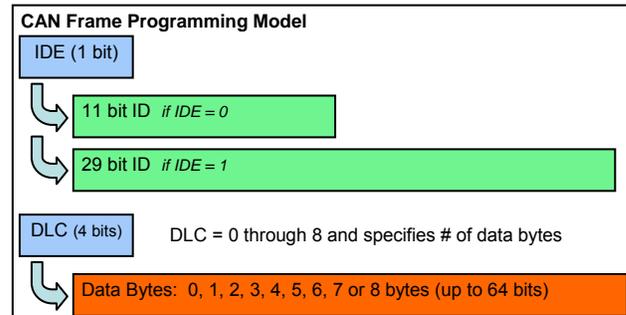
The top two traces are CAN_Hi and CAN_Lo respectively. Note they are 180 degrees out of phase. The bottom trace is the algebraic sum of the top two. This is from the Keil CAN example program.



The CAN Frame:

The CAN frame has many fields but we can simplify this to a Programming Model as shown. These fields are accessed by your software through the CAN controller registers. The configuration registers are not included here.

- **IDE:** Identifier Extension: 1 bit - specifies if the ID field to be transmitted is 11 or 29 bits:
If IDE = 0, then the ID is 11 bits.
If IDE = 1, then the ID is 29 bits.
- **ID:** Identifier: 11 or 29 bits as set by the IDE field.
This part of the CAN frame sets the priority.
- **DLC:** Data Length Code: 4 bits - specifies number of data bytes in the frame from 0 through 8.
- **Data Bytes:** 0 through 8 bytes.



TIP: A CAN frame with an ID field of either 11 or 29 bits *and* with zero data bytes is valid and useful.

TIP: You tell the transmitter size of ID and number of data bytes. The receiver tells you these values when it gets a frame.

ID: Identifier: 11 or 29 bits

The Identifier can be used for any purpose. It is often used as a node address or to identify requests and responses. CAN does not specify any ID values. 11 bit is often called Standard CAN and 29 bit is often 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 at the next bus idle time.
2. An ID of 0 has the highest priority and will always get through. When the first 11 bits of an 11 and 29 bit ID are the same, the 11 bit ID has priority over the 29 bit ID. This is because the IDE bit = 0 for 11 bits and wins arbitration.
3. You can have any mixture of 11 and 29 bit IDs on the bus. The controller can easily sort this out.
4. Messages tend to start transmitting at the same time. A CAN node is not allowed to start transmitting a frame in the middle of another node's frame. This will cause a bus error. Can controllers will not make this mistake.
5. CAN controllers can be configured to pass only certain received messages to its host processor. Choose your ID values carefully to take advantage of this if needed. This will reduce the workload of a node's CPU. This is why some systems use 29 bit IDs even though they do not need that many addresses. This facilitates grouping of IDs for easier filtering since acceptance filters usually do not usually have a very fine granularity.
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 create your own method if that best suits your purpose.

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 completely identical, they will be seen on the bus as one. If the data bytes are different, a bus error will result and the frames will be resent continuously and havoc is created on the bus for a short time until a bus-off.

Data Bytes:

You can select from 0 to 8 data bytes using the 4 bit DLC field. A valid DLC is returned when a frame is received.

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

Remote Frames:

These are not used much anymore 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 frame, recognizes that it has the desired information and sends back a standard CAN frame with the same ID, DLC and with 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: This is a one bit field in the CAN frame created by the transmitting node but set by only by all the other nodes.

TIP: The number 1 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 determined 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 asserted or the controller is RESET. This transmitting node will never go into bus-off mode while re-transmitting this message. Note that a standard CAN test tool will usually act as a second node by providing the ACK signal unless its controller has the ACK feature disabled.

TIP: This presents an excellent opportunity to provide an easy test situation to confirm you are sending out CAN frames. It won't tell you the frequency, ID or data bytes values, but it will tell you if you are putting out something.

1. Connect a termination resistor to your single CAN node with a transceiver connected to the CAN controller. If you do not have a transceiver connected, join the TXD and RXD of the controller together. No resistor is then needed.
2. Do not connect any other node or test tool. Just one node running by itself.
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 for a very clean signal.
4. Configure your CAN controller and write the IDE, ID, DLC and any data bytes into the appropriate registers.
5. A CAN frame 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 at the bottom of this page.

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 prevent sync loss. 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.

Bus Loading:

Many CAN networks work on a bus loading from 15 to 35 % and this is increasing in some applications. 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 but one can come close. System performance does not drop greatly at high bus loading. Recall the message with the highest priority (lowest ID #) goes straight through with no time delay.

TIP: It is possible to get very high bus loads in a short period of time in a 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 as it is transitory.

TIP: It is possible that on a highly loaded bus, low priority messages will never get transmitted or be delayed by higher ones.

Bus Speed:

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

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 120 ohm ½ watt and their value is not critical. Try adjusting the BTR0 and BTR1 timing registers to obtain the most stable operating parameters.

TIP: How to determine the bus 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 connect from ground to one of the CAN leads but the signals will be lower.
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. i.e. 2 µsec = 500 Kbps.

Bus Errors:

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

1. 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 > 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. (TEC is usually zero)
6. IF not, it attempts to retransmit its message which has to go through the priority process again with other messages.
7. All other nodes abandon reading the current frame, and add 4 to their own REC register. (Receive Error Counter)
8. Any nodes that have messages queued up will transmit them now. All others start listening to the bus.
9. Hopefully, this time the message 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, the transceiver chip or the termination resistors.

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 more transient bus errors. This is very tricky to debug without knowledge of the REC and TEC 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. This can require a controller RESET or a certain number of good frames received. See your controller docs.

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 bus fault such as noise, a faulty node or other errors.

What happens if one of the twisted pair opens or is shorted out ? CAN has automatic mechanisms for this. Not all transceiver chips implement all of them. You can usually short CAN Lo to ground (ISO 11898 says you can short Hi also) or open one CAN line. The ground needs to be connected for this case. 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.

Serious bus faults may be reported 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 to life if so configured. CAN has excellent noise immunity because of the twisted pair that are 180 degrees out of phase. The common mode noise is cancelled out and the differential CAN signal is not affected.

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. This is good engineering practice. Watch out for ground loops !

For a practical demonstration of BUS faults: see the section on getting a real system working.

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 require 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 it is a request or response message. If you want to use an existing protocol see ISO 15765. This is what automobiles use. OBDII diagnostics on automobiles use this protocol. OBDII is an example where one message can be comprised of many CAN frames. Diagnostics are common.

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. Messages that are lost (usually because the processor fails to empty the controller input queue fast enough) are replaced quickly.

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 and the Response frames by changing one bit of the ID. An example is that ID 0x248 is a Request frame and 0x648 is its Response frame. The Request frame data bytes indicate what information is requested. The Response frame will contain the requested information or an error indication.

Command: A frame commanding some event to 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 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 and not by the CAN specification. A time-out is helpful to recover from problems with the network such as severe bus errors, catastrophic bus faults, faulty nodes, intermittent connections or a user abort.

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. Another result is to revert back to normal operation. This is common when a user stops making inputs for a long time. You do not want a system to sit in a configuration mode forever. You must control your processes.

A time-out consequence can be a system RESET or less likely, a shutdown. In any case, notification to an operator is a very good idea. In extreme circumstances, such as a remote system not accessible: a mode to download new firmware is effective.

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. Another example of a time-out is when the setting of the clock is started but stopped midway. After the time-out, the clock will revert to its normal operating mode.

Another good example is the MARS Rovers. If communication stopped for a certain period of time: it will be assumed that a catastrophic event has occurred. Hopefully this will be a software bug and not a hardware failure. The Rover will go into a special mode where it listens for a software update or if really bad, a complete replacement to be sent from Earth.

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 rather by your software or a suitable specification or protocol.

Sequence of Transmitting Data on the CAN Bus:

1. You give the transmitter the ID, the size of the ID (IDE), the number of data bytes (DLC) and the data if any.
2. You then set a bit to tell the transmitter in the CAN controller to send this frame.
3. Any node(s), seeing the bus idle for the required minimum time, can start sending a CAN frame.
4. All other nodes start receiving it except those also starting to transmit a message at the same time.
5. If any other node starts transmitting: the arbitration process starts – the node with the highest priority (lower ID value) continues and lesser priority nodes stop sending and immediately turn into receivers and receive the priority message.
Note: The losing node “knows” the beginning of the other message. CAN arbitration is non-destructive.
6. At this point, only one node is transmitting a message and no other will start during this time else a bus error happens.
7. 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 (normally all of them) to signify the frame was received without errors.
8. 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.
9. At this time, any node can start sending any message or the bus goes into the idle state if none do. Go to 1.
10. If # 7 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 its initial message forever.

Transmitting Notes:

- **How do I transmit my 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. Unless the controller signals otherwise to the processor, you can assume the message was sent.
- **How does a node know when it should transmit a message ?** The CAN controller continuously monitors the bus. When it sees \geq required number of idle bits (11 after ACK bit), it starts transmitting. It is quite possible for other node(s) to start transmitting at the same time. Arbitration decides which message is actually transmitted.
- **What if there is an error ?** All nodes, including the transmitting node, monitor the bus for any errors. If an error condition is detected – a node or nodes signal 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 note this error event and take appropriate action. The message being sent (and now aborted) will be resent but only for a certain number of times. See Bus Errors on page 8.
- **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 a copy frame will be along shortly.
- **How do I add a node ?** Just attach CAN-Hi and CAN-Lo signals to the existing network. If your CAN controller is not initialized, nothing will happen. If it is initialized to the correct frequency, it will start listening to the bus and set the ACK bit if appropriate. Until your software reads messages out of the buffer, messages will be lost.
- **What happens if I “hot plug” a node on the bus ?** Usually any disturbances will be taken care of by the CAN error detection schemes. A message that is sufficiently corrupted will result in a bus error and subsequent retransmission. If the TEC register is high or the disturbances are of a long duration, a bus-off might result.

Arbitration Notes:

1. Arbitration is performed bit-wise. That is, bit by bit on the ID (11 or 29 bit) and the IDE bit. No other bits are used.
2. The node that wins arbitration is not slowed or delayed by this process. CAN arbitration is non-destructive.
3. The losing nodes, if there are any, will attempt to retransmit at the next idle bus time.
4. CAN is not deterministic. This means you are usually never sure when a CAN message will appear on the bus.
5. If you need determinism, try Time Triggered CAN. TTCAN is described in ISO 11898-4. It is a software layer that sits on top of regular CAN. It places frames into specified time slots.

Sequence of Receiving data from the CAN Bus:

1. All nodes except those currently transmitting frames and those in bus-off mode are in listening mode.
2. A CAN frame is sent using the procedure as described previously: Sequence of Transmitting Data on the CAN Bus:
3. This sent frame is received by all listening nodes. If deemed to be a valid CAN message with no errors – the ACK bit is set by all listeners. 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 it is enabled. 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. You do not tell the receiver what the ID size is or the number of data bytes to be received. When the receiver gets a valid frame, it deciphers this information and provides you with the appropriate IDE and DLC registers.
7. The host processor decides what to do with this message as determined by your software.

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. Using interrupts is the recommended method and causes the CPU to jump to a handler to read the frame from the controller. Make sure your processor can handle 100% bus load bursts.

Receiving Notes:

1. 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. This appears to be designed into CAN to handle dropped messages.

2. 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 cannot be recovered or automatically resent by the transmitting node. It is gone forever unless you provide a suitable mechanism in your software to have it resent.

It is possible to have a burst of CAN traffic approaching 100% bus loading when the controller dumps all its data on the bus. Your system must be prepared for this event. The CAN specification does not space out frames.

3. How do the CAN controllers stay in sync when there is only the bus idle voltage and no transistions:

The CAN controller depends on its internal clock to stay as close to the design frequency as possible. Upon receipt of the start bit, an internal counter starts counting the “time quanta” (TQ) that further divide the bit time. The number of TQs in a bit period is set in the controller by the user. The controller will automatically add or subtract TQs to adjust its effective operation frequency.

CAN bit time transitions are used to calculate the correct number of TQs needed to keep the receivers in sync.

See the data sheet for your CAN controller for details.

4. I have a high bus loading factor. How can I reduce the pressure on my CAN controller ?

There are many ways and here are several:

- Use the acceptance filters to ignore any messages your processor does not need. Ignored messages will never be sent to the processor. They are discarded very quickly by the CAN controller.
- Use more than one CAN controller in your processor to receive CAN frames. Set the acceptance filters to divide the messages by ID (or even 1st data byte) to each of the controllers. Your processor must still have the ability to process the messages. For transmit: use mailboxes to alternatively transmit the CAN frames.
- Space the messages sent by the nodes.
- Use sub-nodes. Sometimes using a different protocol than CAN is appropriate.
- Use a faster ARM processor.

5. Where do I get the values for the timing registers BTR0 and BTR1 ?

The controller data sheet will provide the formulae to configure these timing registers. Suggested values for various bus frequencies are usually provided. It is especially important to get the sampling point correctly set.

CAN FD: CAN with Flexible Data Rate:

CAN FD is a new extension to the standard CAN 2.0 protocol. For more information search the internet for the files [can_fd_spec.pdf](#) and [can_fd.pdf](#). CAN FD was created by Robert Bosch GmbH.

The CAN frequency and bit overhead added to information carrying bits (ID + data bytes) (also called payload) are limitations to the effective maximum data rate transmission. CAN FD is one solution to this.

In a CAN system, once the frequency is chosen and implemented, it is difficult to change. Changes are easier if you know exactly what nodes are in a system such as in a passenger vehicle. In systems where the nodes can be supplied from different manufacturers depending on customer options or added by after-market users, the change problem is usually a problem.

This is certainly true for SAE J1939. J1939 is used in heavy duty trucks, buses, marine and in construction and farm equipment. It uses CAN at 250 Kbps. This speed is not fast enough for larger, more bus intensive J1939 systems. CAN FD might be a good solution as systems can slowly migrate from CAN to CAN FD.

CAN FD provides:

1. Up to 64 bytes of data bytes. Remember regular CAN has from 0 to 8 data bytes.
2. It is possible to increase the bus speed during the transmission of the DLC, data and CRC fields and before ACK bit. This time period occurs just after arbitration is complete.

Features of CAN FD:

1. A CAN FD controller will be different than regular CAN. It can also transmit and receive regular CAN frames.
2. Uses the same physical layer as regular CAN.
3. Can use the same transceiver although specialized ones might be made available.
4. CAN FD can be used for specific applications such as programming, large data transfers or general use.
5. When CAN FD is transmitting, regular CAN must be in standby. CAN and FD collisions will result in Bus errors.
6. Regular CAN uses bits 0000 through 1000 of the 4 bits of DLC (Data Length Code). CAN FD adds 1001 through 1111 to extend the data field.
7. Reserve bit R0 (in 11 bit) or R1 (in 29 bit) are used to signify the frame is CAN FD. This is the EDL bit.
8. Three new bits are added: They are added just before the DLC field.
 - a. EDL: (Extended Data Length): specifies a CAN or CAN FD frame (EDL was called R0 or R1).
 - b. BRS: (Bit Rate Switch): switches the bit rate after arbitration and before ACK bit.
 - c. ESI: (Error State Indicator): denotes if the node is in error-active or error-passive mode.

Selection of Data Bytes:

The DLC has 4 bits: b0000 through b1000 are used by regular CAN and CAN FD to signify from 0 to 8 data bytes.

CAN FD uses b1001 through b1111. Each of these bits represents not one byte as in regular CAN, but map into a table. This makes a total of up 64 data bytes. See this table to the right: With this scheme it is not possible to select some byte numbers.

The rest of the story- The Details:

There are many details covered in the Bosch and other documents. When this was written, no CAN FD controllers are yet available but NXP might be first. For details visit <http://can-newsletter.org/> and search for **FD TechDay**.

Other useful network protocols:

Here are some other protocols that might prove useful to complement CAN:

FlexRay: High speed dual channel Time Triggered network. You can use one or two channels. www.flexray.com It is used for redundancy in safety critical applications.

LIN: Single wire network using a common UART. Nearly any controller can be used to implement LIN. LIN is a very low cost network. It is often used as a sub-network to CAN. See www.lin-subbus.org

Safe-by-Wire: A very reliable network used to activate vehicle airbags. Search the web for [safebywire.pdf](#).

Ethernet: It is very common to use CAN for a small, local network (such as on one machine) and then use a gateway to ethernet for the long distances or to connect to the other machines. It can handle the traffic load of multiple CAN networks.

Wireless: WiFi, ZigBee, Bluetooth and NFC (Near Field Communication) are all useful in small networks attached to CAN.

DLC	Number of Data Bytes
1001	12
1010	16
1011	20
1100	24
1101	32
1110	48
1111	64

CAN Controllers and their Errata Sheets:

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 lists all known deviant behaviour from that claimed in the device datasheet. Some CAN controllers 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 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. Then you have to be prepared for the day these bugs get fixed and show up in silicon on your board. Most issues will be in the controllers and not the rather 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 self-proclaimed experts are in the minority. See <http://tech.groups.yahoo.com/group/CANbus/> and www.canlist.org.

For CANopen and other information see: www.can-cia.org/ Most CANopen docs are free while most others charge.

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 an onerous task. There are two ways to help here. One is to use a working node such as an evaluation board with some proven CAN examples provided. You can attempt to receive these known good CAN frames with your node.

Second, you can purchase a CAN test tool. This is the best idea. These provide both sending and receiving capabilities and usually (optionally) 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 such as translation that can save some serious cash and time.

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

Oscilloscopes are quite useful in making sure the CAN waveforms are not distorted. This can disclose the causes of some very strange network behavior. For a combination CAN analyzer and oscilloscope see www.phytec.com/pcan-diag.html or search the web for Phytec PCAN-Diag. Standard scopes also work well and of these, memory scopes work best.

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. Normally, you can construct your own database to convert numbers embedded in the CAN frames to your own custom descriptive words.

Typical suppliers are Dearborn Group www.dgtech.com, National Instruments www.ni.com, Intrepid www.intrepidcs.com and Vector CANalyzer www.vector.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. Even so, it would be good to check if an analyzer is sufficiently adaptable for your needs.

As with all tools, buy the best analyzer you can afford ! You are rarely disappointed with fine products...only cheap ones...

CAN Documents:

CAN documents are available for ISO (ISO 11898) and SAE (J1939). They are not free. www.iso.org and www.sae.org.

The original Bosch 2.0 document is free: Search the web for [can2spec.pdf](#).

Keil provided Software:

Keil provides CAN middleware for many ARM Cortex-M processors. Visit www.keil.com/rl-arm/rl-can.asp

The Keil RTX RTOS is now available free with a BSD type license. Ports are available for Keil MDK, IAR and GCC.

See www.arm.com/cmsis for more information.

What's ahead:

Now, on the next few pages, we will 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 most exercises no hardware is needed. We will use the Keil simulator which is part of MDK. No license is needed. For the others, you will need an evaluation board with a NXP1700 processor and a Keil ULINK2, ULINK-ME™ or a Segger J-Link (black case V6 or later). For ETM instruction tracing, a ULINKpro is used. You can easily substitute other boards. Keil has many CAN examples.

Keil 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 simulator included in the ARM Keil™ MDK toolkit for the NXP LPC1700 ARM® Cortex™-M3 processor. No hardware is needed. MDK will also connect to any ARM processor hardware through either the JTAG or Serial Wire Debug (SWD) ports using a Keil ULINK2™, ULINK-ME™, a Segger J-Link and for ETM trace: a ULINKpro™. Keil provides CAN examples for many ARM processors. Look for directories \CAN in C:\Keil\ARM\boards.

Keil MDK Software: You can download the latest version of MDK-ARM at: www.keil.com/demo and select MDK-ARM. There is no charge for this software. Please install this software on your PC. Use MDK version 4.60 or later.

Keil also provides software tools for NXP ARM7™, ARM9™, Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4 and the LPC4300 Cortex-M0/M4 processors and most 8051 processors. Many of these processors have CAN controllers.

Documents: NXP has 2 main pieces of documentation: 1) Datasheet (Overview and HW specification) and 2) Users Manual (detailed peripheral information). Technical details on the NXP CAN module are found in the Users Manual. See www.nxp.com/microcontrollers. The LPC1700 CAN controller is the same one used in the LPC23xx.

NXP LPC1700 series processors incorporate ARM Serial Wire Debug (SWD or SW), Serial Wire Viewer (SWV) and ETM Trace. Examples are provided in this document showing trace with these powerful debugging tools.

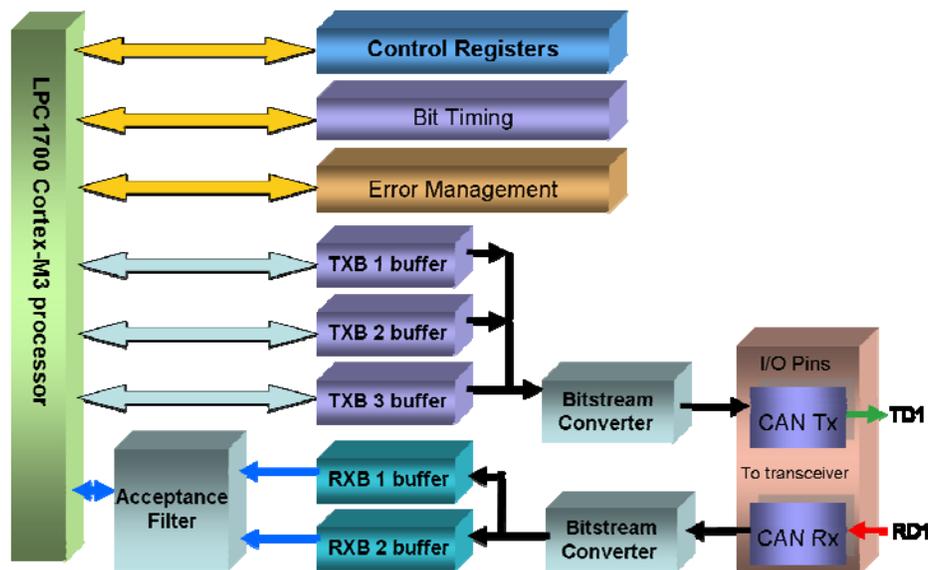
NXP CAN Controller for Cortex-M3 Processors.

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

1. I/O Pins: These connect to the CAN transceiver chip pins RD1 and TD1 as already described.
2. Bitstream Converters: CAN is a serial bus while the processor is parallel. Conversion happens here.
3. TXB mailbox: The messages to be transmitted are written here. ID, IDE, DLC and data (if any) 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 and enable them to filter messages.
5. RXB Receive Buffer: Each buffer holds 1 CAN message. They provide a buffering system to the processor.
6. Control, Status, Bit Timing and Error management 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: (using the Keil simulator in μ Vision 4)

1. Obtain Keil MDK for ARM at: www.keil.com/arm and install it in the default directory. No license is needed.

2. Start μ Vision by clicking on its icon on your Desktop.



3. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\Keil\MCB1700\CAN\CAN.uvproj.

4. Select “Simulator” in the Target window.



5. Compile the source files by clicking on the Build icon.



6. Click on the Target Options icon. Select Debug tab and confirm “Use Simulator” is selected. Click OK.



7. Enter the Debug mode by clicking on the debug icon. Select OK if the Evaluation Mode box appears.



8. Position the Toolbox and the “CAN: Communication” windows as appropriate.

9. Click on the RUN icon. **Note:** you stop the program with the STOP icon.



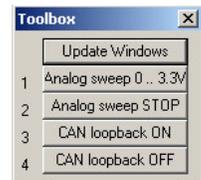
10. Note CAN messages with an ID of 0x21 and one data byte 0x00 will appear in the CAN: Communications window. Only Xmit CAN frames will be visible from controller 2 (CAN1).

11. Click CAN loopback ON to see both transmit and receive frames as shown below.

Note the ID, Len (length) and Data columns. You should now know what these are. # represents which CAN controller the frame represents: 1 or 2 relating to CAN0 and CAN1 respectively.

12. Controller CAN2 is used to transmit frames and CAN1 to receive them in this example. This is our minimum two node CAN network.

13. In the Toolbox window, click on the “Analog Sweep 0...3.3v”.



14. Changing data values representing output from the simulated A/D convertor will now appear in the CAN data field in the CAN Communication window as shown below.

15. If you open Watch 1 (View/Watch/Watch 1) you can see the send and receive variables update with the CAN data.

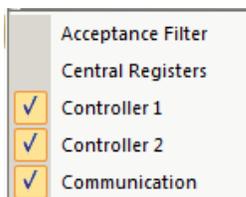
TIP: To see how we simulated the data values and the Loopback feature: open Debug/Function Editor.

TIP: If you don't see any values changing, make sure View/Periodic Window Update is selected.

The Keil simulator provides an excellent way to develop your programs before and after you have your hardware.

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. The processor must still be able to handle the CAN traffic.

The CAN Controller windows can be opened if not already in Peripherals/CAN as shown below: Select CAN Controller 1 and/or CAN Controller 2. These contain the registers used to configure each CAN controller by your software.



TIP: To clear the CAN Communication window: just double-click anywhere in it.

Number	States	#	ID (Hex)	Dir	Len	Data (Hex)
490	16542208	2	021	Xmit	1	80
491	16542208	1	021	Rec	1	80
492	16593094	2	021	Xmit	1	80
493	16593094	1	021	Rec	1	80
494	16643980	2	021	Xmit	1	8D
495	16643980	1	021	Rec	1	8D
496	16694866	2	021	Xmit	1	8D
497	16694866	1	021	Rec	1	8D
498	16745752	2	021	Xmit	1	99
499	16745752	1	021	Rec	1	99
500	16796638	2	021	Xmit	1	99
501	16796638	1	021	Rec	1	99
502	16847524	2	021	Xmit	1	A6
503	16847524	1	021	Rec	1	A6
504	16898410	2	021	Xmit	1	A6
505	16898410	1	021	Rec	1	A6
506	16949296	2	021	Xmit	1	B2
507	16949296	1	021	Rec	1	B2
508	17000182	2	021	Xmit	1	B2

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 on the previous page. You can view and edit the source files whether in debug mode or not, but to compile them you must be in edit mode and not debug mode. This example uses almost no assembly code as it is nearly entirely written in C. Any source file can be opened in μ Vision if not already visible by clicking on File/Open. 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 arrays: one with two transmit message objects and one with two receive message objects for the two CAN controllers.

Can.c: This C code initializes the CAN controller, writes and transmits a message, receives a message, configures the Acceptance Filters and provide the receive interrupt handlers which also sets the frame transmitted flags.

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.

TIP: The receive controller is interrupt driven and the transmit controller is periodically driven in this example. Normally, we recommend interrupt driven functions for both receive and transmit to reduce the chances of dropped messages.

1) 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. Can.h is used. Examples are found in the function **CAN_setup** (lines 78 to 100).

1. Enable and set the clocks and initialize both CAN controllers.

TIP: The clock must be stable for CAN. No R-C oscillators allowed! Good stable clocks are necessary for CAN.

2. Configure GPIO ports P0.0, P0.1, P2.7 and P2.8 for the transmit and receive lines to the transceiver chip.
3. Enable the interrupt for the receive functions. (recall transmit is periodically driven).
4. Transmit frame and receive frame functions plus enable interrupt.
5. Configure Acceptance Filters. This is optional. By default, all frames are passed through to the CPU.

Set CAN_BTR: This is a 32 bit CAN controller register where items such as bit timing, bus frequency and the sample point are set. In our example, the baudrate is set to 500 Kbps (bits/sec) in the function `CAN_cfgBaudrate`.

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 optimize these settings for the most robust bus performance in your particular system.

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 and for the calculations see the LPC1700 Users Manual or the documents for your processor.

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

TIP: CAN engineers use sensitivity testing to verify their networks are robust and reliable. They change many variables to see when the network fails and execute hundreds or thousands of runs. This is a very popular strategy.

Super TIP: If you are experiencing strange problems that you are unable to fix, please read the CAN controller errata sheet. CAN controllers are complex devices and bugs are not unknown.

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 you 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 release it to be sent to the LPC1700 processor.
- **CAN_wrFilter:** Configure the acceptance filter. This is not discussed in this article. Source code to program the acceptance filter is in the LPC1700 CAN example.
- **CAN_IRQHandler:** The receive interrupt handler as well as setting frame transmitted flags `CAN_TxRdy[]`.

These six functions are called by the `main()` function in CanDemo.c.

2) Can.h

The CAN Structure CAN_Msg: (lines 25-31, 41 & 42)

Shown is the structure declaration in Can.h. You should now be able to recognize each of these members. You enter either an 11 or 29 bit identifier. Two instances of CAN.msg are defined for each CAN controller and are shown below as the arrays CAN_TxMsg and CAN_RxMsg. CanDemo.c writes the data to these to create the CAN messages.

The definition for the data structure used in Can.c are listed in Can.h in lines 25 to 31. These are visible in µVision:

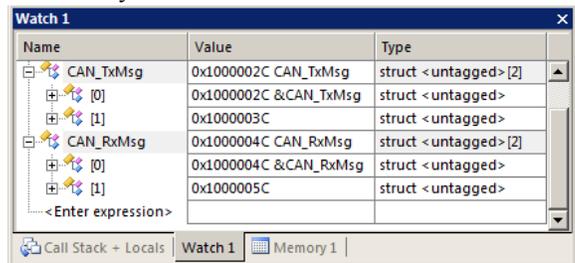
```
25 typedef struct {
26     unsigned int id;           // 29 bit identifier
27     unsigned char data[8];    // Data field
28     unsigned char len;       // Length of data field in bytes
29     unsigned char format;    // 0 - STANDARD, 1- EXTENDED IDENTIFIER
30     unsigned char type;      // 0 - DATA FRAME, 1 - REMOTE FRAME
31 } CAN_msg;
...
41 extern CAN_msg CAN_TxMsg[2]; // CAN message for sending
42 extern CAN_msg CAN_RxMsg[2]; // CAN message for receiving
```

How to view these CAN structures in the Watch window:

1. **Find an instance of CAN_TxMsg in Can.h or CAN_Demo.c.** Right click on it and select “Add Can_TxMsg to ...” and select Watch 1. Repeat for Can_RxMsg. Watch 1 will now display all four structure elements:

TIP: You can also click on <Enter Expression> in the Watch window and manually enter the structure names.

2. Expand the various entries and you can view the various structure elements and their location and contents.
3. Look at CAN_TxMsg[1]. Recall this is to transmit frames.
4. Expand the Data member data. If the program is running you can see Data [0] changing as shown below:
5. Look at CAN_RxMsg[0]. Recall this is to receive frames.
6. Expand down to Data [0]. If the CAN Loopback button was pressed while the program is running, this element will update as it is received from the CAN transmit controller CAN1.

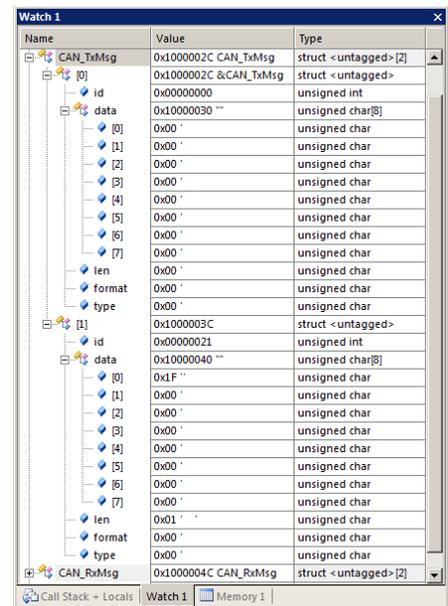
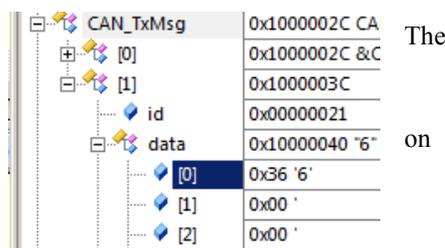


TIP: You can see these update in real-time on a real board with any ULINK.

On a real board, ARM CoreSight™ technology is used to read and write memory locations without stealing CPU cycles.

Note the CAN_TxMsg structure starts at address 0x1000002C. CAN_RxMsg structure starts at address 0x1000004C.

This address could change depending on the specific compiler optimization settings you are using.



3) CanDemo.c

This contains the main() function and contains the example program that reads the voltage from 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:

Note: CAN_TxMsg[0] in the box below should be CAN_TxMsg[1].

Lines 115 to 119 loads default values into the structure CAN_TxMsg. (Except for the data byte from the A/D converter.)

```
115 CAN_TxMsg[1].id = 33; /* initialize message to send */
116 for (i = 0; i < 8; i++) CAN_TxMsg[0].data[i] = 0;
117 CAN_TxMsg[1].len = 1;
118 CAN_TxMsg[1].format = STANDARD_FORMAT;
119 CAN_TxMsg[1].type = DATA_FRAME;
```

This CAN message will send one data byte. For example, if you change the value in the member CAN_TxMsg[1].len to “3”, three data bytes will be sent on the bus. What data will be in them depends on the contents of CAN_TxMsg[1].data.

TIP: If you send more data bytes than you have data, it is a good practice to fill the empty data bytes with either 00 or FF.

Line 128 sees if the IRQ Handler has determined the last CAN message was sent and clears this flag. Line 130 loads the A/D value val_Tx into the data member CAN_TxMsg[0].data in data byte 0 and line 131 transmits it by calling the function CAN_wrMsg which resides in module Can.c.

```
127 if (CAN_TxRdy[1]) { /* tx msg on CAN Ctrl #2 */
128     CAN_TxRdy[1] = 0;
130     CAN_TxMsg[1].data[0] = val_Tx; /* data[0] field = ADC value */
131     CAN_wrMsg (2, &CAN_TxMsg[1]); /* transmit message */
```

Receiving a CAN Message:

Lines 136 to 139 are used to retrieve a CAN message from the controller. But something more must be going on here. Line 142 shows that the data byte received and inserted in the array[0] is sent to be displayed on the LCD by val_display.

How exactly does the CAN data byte get into the member array CAN_RxMsg.data[0]? This is done in CAN_RdMsg (in CAN.c) and was triggered by the CAN interrupt 25.

```
136 if (CAN_RxRdy[0]) {
137     CAN_RxRdy[0] = 0;
139     val_Rx = CAN_RxMsg[0].data[0]; }
142     val_display (); /* display TX and RX values */
```

In CanDemo.c:

1. If flag CAN_RxRdy[0] = 1, then set it to 0. This means the controller activated interrupt 25.
2. Line 139: Get received data byte from array CAN_RxMsg[0] and put in val_Rx.
3. val_Rx is displayed on LCD by function val_display if enabled. Go to 136 to get another received message.

In Can.c:

1. If CAN_IRQHandler sees the CAN controller has a received message ready, it calls function CAN_rdMsg.
2. CAN_RdMsg reads data byte (among other things) and then puts this data byte into array CAN_RxMsg[0].
3. CAN_IRQHandler sets flag CAN_RxRdy[0] = 1 which will be detected by main() in CanDemo.c. Go to 1.

Interrupt Handler: void CAN_IRQHandler (void) (Can.c)

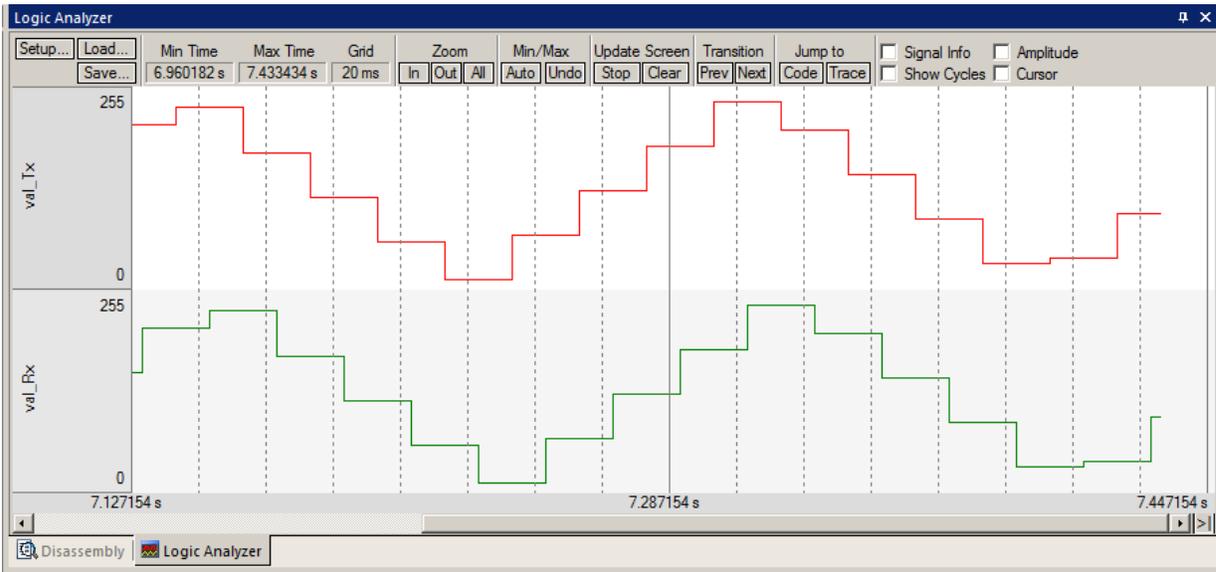
The LPC1700 has one common interrupt for both CAN controllers. This is ExtIRQ 25 also known as Number 41.

Lines Lines 299 to 330 determine what caused the interrupt. We can see which parts of the handler have been executed after you have run the program using Keil Code Coverage. Green blocks mean the line/instruction was executed and grey means not executed. You should therefore notice that CAN 1 only received frames and CAN 2 only transmitted them.

Viewing the CAN Frames with the Logic Analyzer (LA):

µVision has a Logic Analyzer (LA). This feature is available using Serial Wire Viewer on a real processor or the simulator.

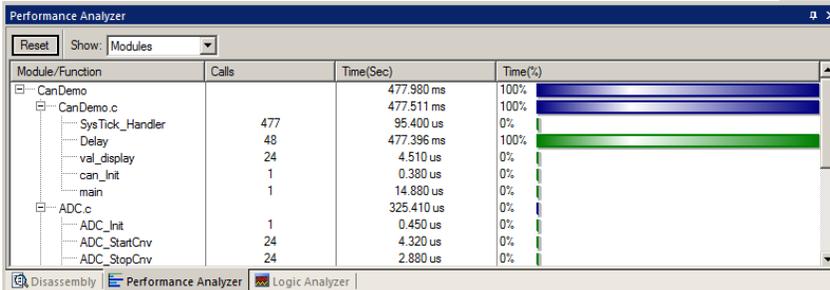
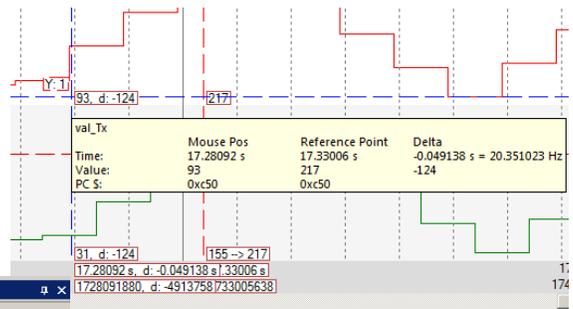
1. Run the CAN_Demo program. See page 15. You can configure the LA while the program is running.
2. CAN_Demo.c has two global variables val_Tx and val_Rx. Find where these are declared near line 28.
3. Right click on val_Tx and select Add val_Tx to... and select Logic Analyzer. This will also open the LA up.
4. Repeat for val_Rx. Both variables will now be visible in the LA as shown below.
5. **Adjust the range:** Click on Setup... and highlight each variable in turn and set Display Range: MAX: to 0xFF.
6. Click on OK to return to the main µVision window.



7. Click on Zoom All and/or IN and Out to get an appropriate screen as shown above.
8. The values of the CAN Data byte 1 are displayed as they change.
9. You can select Signal Info, Show Cycles, Amplitude and Cursor.
10. Click on Stop in Update Screen if desired. You can also stop the program if you prefer.
11. Select Signal Info, Show Cycles, Amplitude and Cursor.
12. Move your cursor over the waveform to see the effects and the information about timings provided.

Performance Analyzer and Execution Profiler:

13. Open View/Analysis Windows/ Performance Analyzer. Expand on some of the module names.
14. Note the information provided as shown below:
15. This is also provided with ETM trace and a ULINKpro.
16. If you select Debug/Execution Profiling you can see information in times executed or time in the right corner here:



Abstract.txt	CanDemo.c
122	5.850 us ADC_
123	15.600 us val_
124	5.850 us ADC_
125	

Viewing program Flow with the Instruction Trace: With Simulator or ETM:

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_rdmMsg at lines 147 to 163. Examining it, clearly it is here that CAN ID and data byte(s) are read. But how does this function get called? It is not called from CanDemo.c.

We can use the Trace function of μ Vision in the Simulator or ETM mode to figure out how CAN_rdmMsg is called.

The Trace Data window contains a record of all the instructions executed. Instruction trace is extremely useful to figure out in what order functions and instructions were executed and for other program flow problems.

Trace works in the Keil simulator and with ETM with a ULINK pro . Most NXP Cortex processors have ETM trace.

1. Use the same Blinky program as before. Be in Debug mode. 
2. Set a breakpoint in line 147 at the start of CAN_rdmMsg in Can.c.
3. Click on the arrow beside the Trace icon and select Enable Trace Recording.  Or open View/Trace.
4. Click on the arrow beside the Trace icon again and select Trace Data.
5. Run the program to the breakpoint.

TIP: If the program doesn't stop click on ToolBox CAN Loopback ON.

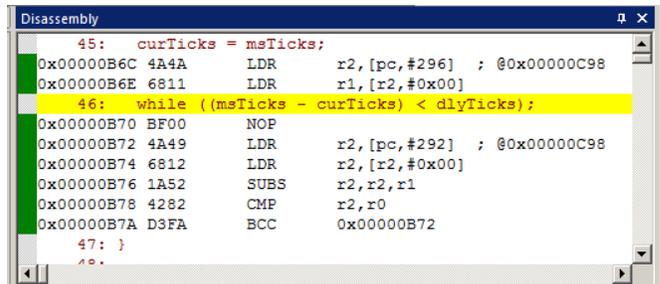
6. View the Trace Data window: Here are some interesting facts:
7. The last instruction to be executed was 0xAD8 BL.W: a branch to function CAN_rdmMsg at 0x760. If you double-click on the line at Nr. 32,767 this line will be displayed in the Disassembly and C source windows.
8. The next instruction to be executed is 0x760 which is the start of the function CAN_rdmMsg. See the PC value in the Disassembly or Register windows. This instruction is not in the trace as it has not been executed yet.



Nr.	Time	Address	Opcode	Instruction	Src Code
32,754	17.640 503 557 s	0x00000B76	1A52	SUBS r2,r2,r1	
32,755	17.640 503 567 s	0x00000B78	4282	CMP r2,r0	
32,756	17.640 503 577 s	0x00000B7A	D3FA	BCC 0x00000B72	
32,757	17.640 503 607 s	0x00000B72	4A49	LDR r2,[pc,#292] ; @0x00000C98	
32,758	17.640 503 627 s	0x00000B74	6812	LDR r2,[r2,#0x00]	
32,759	17.640 503 647 s	0x00000B76	1A52	SUBS r2,r2,r1	
32,760	17.640 503 757 s	0x00000AC8	B510	PUSH {r4,lr}	void CAN_IRQHandler (void) {
32,761	17.640 503 787 s	0x00000ACA	4820	LDR r0,[pc,#128] ; @0x00000B4C	icr = LPC_CAN1->ICR; /* clear interrupts */
32,762	17.640 503 807 s	0x00000ACC	68C4	LDR r4,[r0,#0x0C]	
32,763	17.640 503 827 s	0x00000ACE	F0140F01	TST r4,#0x01	if (icr & (1 << 0)) { /* CAN Controller #1 message is received */
32,764	17.640 503 837 s	0x00000AD2	D009	BEQ 0x00000AEB	
32,765	17.640 503 847 s	0x00000AD4	491E	LDR r1,[pc,#120] ; @0x00000B50	CAN_rdmMsg (1, &CAN_RxMsg[0]); /* read the message */
32,766	17.640 503 867 s	0x00000AD6	2001	MOVS r0,#0x01	
32,767	17.640 503 877 s	0x00000AD8	F7FFFE42	BL.W CAN_rdmMsg (0x00000760)	

CAN_IRQHandler:

1. Note Nr. 32,760. This instruction (PUSH) and the rest are part of the IRQ handler.
2. The instructions before this are in the Delay function source line 46: while ((msTicks - curTicks) < dlyTicks);
3. Double-click on Nr. 32,75 through 32,759 to confirm they are from these instructions:
4. What happened is the CAN controller generated an interrupt which jumped to CAN_IRQHandler at 0x0AC8. Examine the source code to confirm. The interrupt was caused by the CAN controller receiving a CAN message. Therefore the function CAN_rdmMsg was executed to retrieve this message from CAN controller 0.



Experimenting with the CAN software:

The default MCB1700 CAN example produces a CAN frame with an ID of 0x21 and one data byte. We will make some modifications to CANDemo.c to show how easy it is to modify it for your own uses. This works with either the Simulator or with a real processor. You can use any ULINK or a Segger J-Link (black case V 6 or later) with a target board.

Have the CAN project loaded as before. Close the Data Trace window. Exit Debug mode.

We will modify these settings first: they are located in CANDemo.c near the lines as indicated:

```

115     CAN_TxMsg[1].id = 33;                               /* initialize msg to send */
116     for (i = 0; i < 8; i++) CAN_TxMsg[0].data[i] = 0;
117     CAN_TxMsg[1].len = 1;
118     CAN_TxMsg[1].format = STANDARD_FORMAT;
119
120     CAN_TxMsg[1].type = DATA_FRAME;

```

1) How to change the ID from 11 to 29 bit and also change the ID value:

1. Modify line 115 from `CAN_TxMsg[1].id = 33;` to this: `CAN_TxMsg[1].id = 0x1234567;`
2. Change the value in line 118 from `STANDARD_FORMAT` to `EXTENDED_FORMAT`. You can also use a 1.
3. Click on Build.  Enter Debug mode.  Click on the RUN icon. 
4. The CAN Communication window will display the new ID:

This was really easy to do !

Number	States	#	ID (Hex)	Dir	Len	Data (Hex)
0	22650	2	01234567	Xmit	1	00
1	2022078	2	01234567	Xmit	1	00
2	4022073	2	01234567	Xmit	1	00
3	6022078	2	01234567	Xmit	1	00

2) Create more than one Data Byte in a message:

1. Stop the program STOP icon.  and exit Debug mode. 
2. Modify line 117 from `CAN_TxMsg[1].len = 1;` to: `CAN_TxMsg[1].len = 8;`
3. Click on Build.  Enter Debug mode. 
4. Select Analog Sweep and CAN Loopback ON. Click on the RUN icon. 
5. Note the eight data bytes but only the first one has a value:

This was also really easy to do !

Number	States	#	ID (Hex)	Dir	Len	Data (Hex)
0	33851	2	01234567	Xmit	8	00 00 00 00 00 00 00 00
1	33851	1	01234567	Rec	8	00 00 00 00 00 00 00 00
2	2033278	2	01234567	Xmit	8	3E 00 00 00 00 00 00 00
3	2033278	1	01234567	Rec	8	3E 00 00 00 00 00 00 00

3) Fill in the Seven Data Bytes:

- 1) Stop the program STOP icon.  and exit Debug mode. 
- 2) Just after line 130: `CAN_TxMsg[1].data[0] = val_Tx;` `/* data[0] = ADC value */`
 ADD: `for (i = 1; i < 8; i++) CAN_TxMsg[1].data[i] = 0x77;`
- 3) Click on Build.  Enter Debug mode. 
- 4) Select Analog Sweep and CAN Loopback ON. Click on the RUN icon. 
- 5) Now you can see 0x77 added to the seven data bytes. You should have no trouble to put your own data in.

4) Acceptance Filter: If you look in the data members for `CAN_RxMsg[0]` in the Watch 1 window, you will not see the data changing. The Acceptance Filter (setup in CAN.c) is turned on blocking everything except ID 0x21. However, the CAN Communication window can see IDs other than 0x21. On a real CAN board (see the next page), Rx will not display any data.

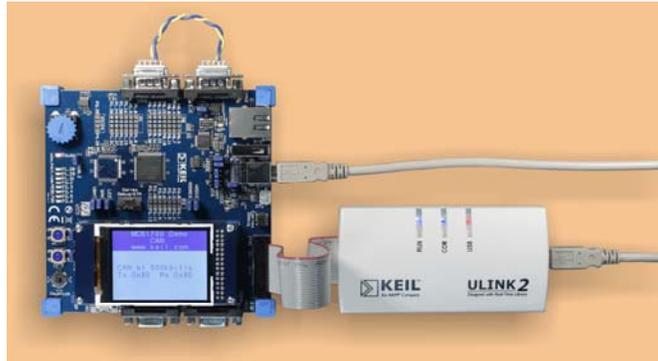
Number	States	#	ID (Hex)	Dir	Len	Data (Hex)
4	8033358	2	01234567	Xmit	8	2E 77 77 77 77 77 77 77
5	8033358	1	01234567	Rec	8	2E 77 77 77 77 77 77 77
6	10033358	2	01234567	Xmit	8	6C 77 77 77 77 77 77 77
7	10033358	1	01234567	Rec	8	6C 77 77 77 77 77 77 77

Getting a CAN Network to work on a real board: *with Serial Wire Viewer...*

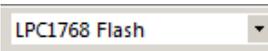
Below is a real two node CAN network using a Keil MCB1700 evaluation board using the same example code. This board contains two separate CAN nodes. We will have one node talk to the other.

This lab uses the ULINK2 or ULINK-ME exclusively for the exercises. You can use other board(s) for this example.

This shows the Keil ULINK2 USB debug adapter connected to a Keil MCB1700 evaluation board. Serial Wire Viewer is supported. You can also use the ULINK-ME, ULINK*pro* or Segger J-Link (black case V6 or later). These are all supported with μ Vision. All provide JTAG, Serial Wire Debug (SWD) and Serial Wire Viewer (SWV). The ULINK*pro*, provides ETM (Embedded Trace Macrocell) support. ETM is discussed presently.



Keil Example CAN Program: (using the Keil MCB1700 board and a ULINK adapter)

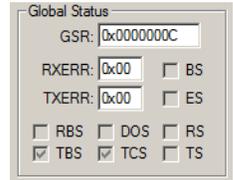
1. Connect the MCB1700 as shown. The CAN1 and CAN2 connectors have pin 2 connected on one to pin 2 on the other and then pins 7 connected in the same fashion. They are not crossed but connected from one to the other.
2. Start μ Vision by clicking on its icon on your Desktop. 
3. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\Keil\MCB1700\CAN\CAN.uvproj.
4. Make sure "LPC1768 Flash" is selected in the Target window. 
5. Set the ID back to 0x21 (dec 33) and length to 11 bits. Recall these were changed on the previous page. See steps 1 and 2 under: **1) How to change the ID from 11 to 29 bit and also change the ID value:**
OR comment out this line in CANDemo.c near line 75: CAN_wrFilter (1, 33, STANDARD_FORMAT);
The Tx: value on the LCD will change but the Rx: value will not when the pot is changed if this is not done.
6. Compile the source files by clicking on the Build icon.  They will compile with 0 errors and 0 warnings.
7. Click the "Target Options" icon.  Select Debug tab and set to "ULINK2/ME Cortex debugger". Click OK.
8. Program the LPC1700 Flash by clicking on the Load icon. 
9. Enter the Debug mode by clicking on the debug icon.  Select OK if the Evaluation Mode box appears.
10. Click on the RUN icon.  Note: you stop the program with the STOP icon. 
11. Note CAN messages with Tx and Rx values will appear on the MCB1700 LCD as shown in the photo above. Turning the board's potentiometer will change the values displayed. This is the CAN data byte displayed which reflects the value of the pot position and is captured by the A/D converter in the LPC1700.
12. This is our minimum two nodes in operation. You could connect a CAN analyzer to the CAN_Hi and CAN_LO wires on the DB9 connectors to monitor the CAN messages or use it to insert values on to the bus.

TIP: If Tx changes and Rx doesn't – the most likely cause is the CAN ports are not wired together correctly. Or you are powering the board using only a ULINK-ME. The ULINK-ME V4 supplies only 3.3 volts to the board through a Shottky diode D1 and not the 5 volts needed by the CAN transceivers. Use a second USB cable as pictured to provide this 5 volts. Or see Step 5 on this page. The Acceptance Filter blocks any ID other than 0x21 and any ID that is 29 bits.

Demonstrating Busoff and the TXERR Counter:

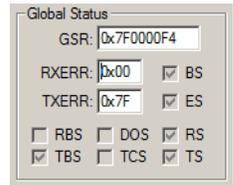
CAN errors was discussed on page 7. Now, we will demonstrate a BUS Fault and how the CAN controller handles this.

1. With the CAN example still running, confirm both Tx: and Rx: change on the LCD when the pot is rotated.
2. From the main μ Vision menu, Open Peripherals/CAN and select Controller 1. Focus on the Global Status area:
GSR: the entire 32 bit memory location.
RXERR: the receive error counter. An 8 bit value.
3. TXERR: The transmit error counter. . An 8 bit value.
BS: Bus Status. Indicates if controller is BUSoff or not.
ES: Error Status: Indicates one or both controllers error is above Warning Level (set to 0x60)
4. In the Memory window, enter the address 0x4004 8008. This is the address of the CAN2GSR register. Both TXerr and RXerr as well as other flags for CAN 2 are at this location. Your program can read these locations. CAN1GSR is at 0x4004 4008. Refer to your controller documentation for details.



Create BUS errors:

5. Short the CAN_Hi and CAN_Lo together. This causes plenty of Bus errors.
6. Note the Rx: filed on LCD stops changing and the Global Status is modified as shown here:
7. TXERR: 0x7F. This was 0xFF but at BUSoff, the controller sets this to 0x7F and RXERR to 0. This is to facilitate the BUSoff recovery process. See the LPC1700 User Manual.
8. BS being set indicates CAN2 is in BUSoff mode.
9. ES being set indicates TXERR is above 0x60 Warning Level.
10. There are many details in this process. Please refer to the LPC1700 User Manual from NXP for more information.

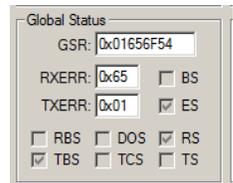


To recover from BUSoff:

1. Press RESET button on the MCB1700 board. Click RUN if necessary. Or exit and re-enter Debug mode.

TIP: When you enter Debug mode you might notice the Global Status register shows some strange numbers as shown here. This is because when μ Vision encounters main() and stops the program, the CAN controllers have not yet been initialized.

2. Set a breakpoint near line 122 on the ADC_StartCnv(); function call in CanDemo.c.
3. Click on RUN and you will now see the GSR value initialized to something appropriate.
4. Remove the breakpoint.



Serial Wire Viewer (SWV) and Embedded Trace Macrocell (ETM):

All NXP Cortex-M3 and Cortex-M4 LPC processors possess significant debugging capabilities including Serial Wire Viewer (SWV). Most also have ETM trace. These are supported by Keil μ Vision using the Keil debug adapters ULINK2, ULINK-ME for SWV and ULINK pro for both SWV and ETM trace. Keil also supports the Segger J-Link (black case V6 and later).

SWV can display PC samples (program counter), data reads and writes, interrupts and CPU counters all with a timestamp. ETM adds all instructions executed. These are provided in real-time with no CPU cycle stealing. The ITM provides printf type debugging which is slightly intrusive. The RTX RTOS Event Viewer uses SWV for the Event Viewer. Examples on NXP hardware are shown using both ETM and SWV.

SWV is not provided by the simulator. SWV are not needed as the simulator by definition has access to all the processor internal nodes. Some of the interesting and useful simulator features such the Performance Analyzer, Code Coverage, Instruction Execution and Instruction Trace are also available with ETM and a ULINK pro on a real target processor.

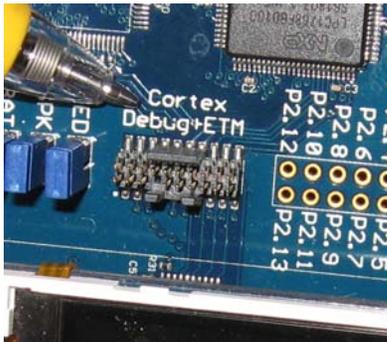
Debugging connections to the MCB1700 and other ARM Cortex boards:

In addition to the standard legacy 20 pin JTAG connector; the MCB1700 has two additional connectors. One is the 20 pin Cortex ETM connector shown on the left. The other is the new 10 pin Cortex Debug connector pictured on the right:

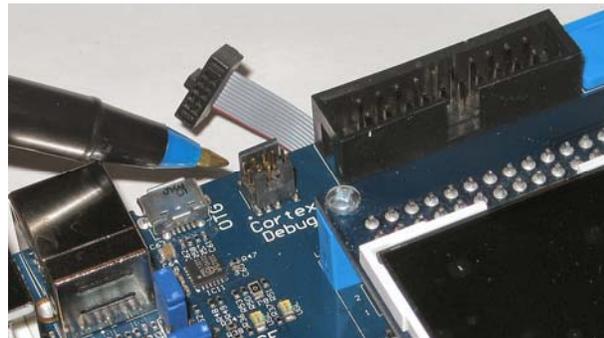
All necessary signals for debugging are on the new 20 and 10 pin Cortex connectors. This includes both JTAG and SWD.

Connector Summary:

1. Legacy JTAG: JTAG, SWD and SWV (out SWO/TDO pin). No ETM.
2. ARM 10 pin CoreSight Cortex Debug: JTAG, SWD and SWV (sent out SWO pin). No ETM.
3. ARM 20 pin CoreSight Cortex Debug+ETM: JTAG, SWD, SWV and ETM (both are sent out on 4 bit Trace Port)



20 Pin Cortex Debug+ETM connector



10 Pin Cortex Debug beside the legacy 20 pin JTAG
Note the 10 pin cable end in the background.

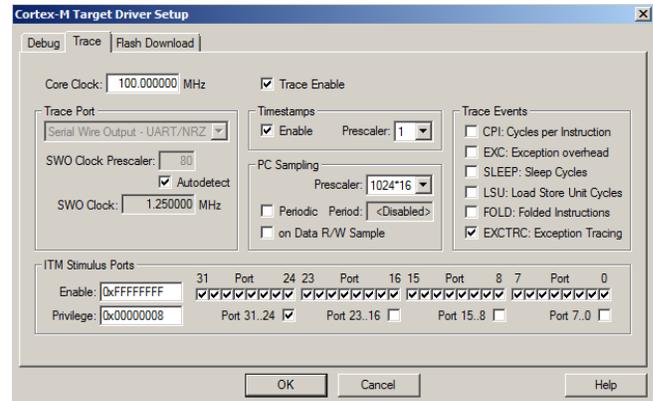
Part numbers for these connectors are discussed here: www.keil.com/support/docs/3506.htm

TIP: If the controllers are not connected together, the sending controller will attempt to put its first frame on the bus. Since there is no other node to provide the ACK bit, the transmitter will resend this first message forever. The fact that Tx in the LCD changes is a bit misleading because without the ACK bit – the Tx value displayed should not change past the first value.

We changed this in the source code to make it easier to visualize things and get the two nodes working together faster. At least you can see that half it is working and the program is not hung up somewhere.

Configuring SWV with ULINK2, ULINK-ME or J-Link: *For Reference:*

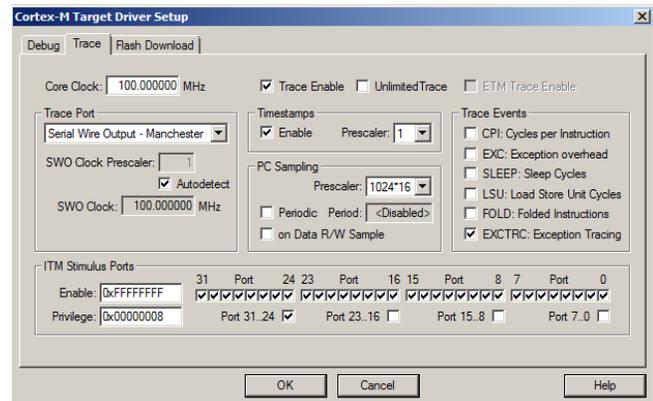
1. Stop  and exit debug mode .
2. Click on the Target Options icon.  Select the Debug tab. Select ULINK2/ME Cortex debugger.
3. Click Settings on the right side of the window.
4. Make sure the SWJ box is checked and SW is entered in the “Port:” box.
5. Click on the Trace tab.
6. Enter 100 for Core Clock: and check the Trace Enable and EXCTRC boxes. Uncheck the Periodic box. This window will be visible:
7. Click on OK twice to return to Edit mode.
8. The Serial Wire Viewer is now configured.



SWV with ULINK2/ME or J-Link:

Configuring SWV with ULINKpro:

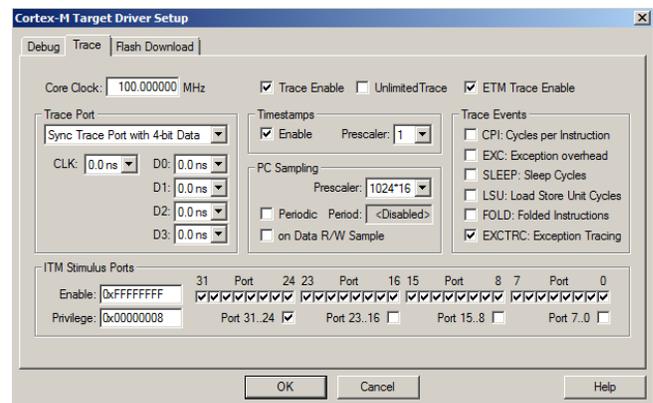
1. Click on the Target Options icon.  Select the Debug tab. Select ULINKPro Cortex debugger.
2. Click Settings on the right side of the window.
3. Make sure the SWJ box is checked. You can select either JTAG or SW in the “Port:” box.
4. Click on the Trace tab.
5. Select Serial Wire Output: Manchester.
6. Enter 100 for Core Clock: and check the Trace Enable and EXCTRC boxes. Uncheck the Periodic box. The middle window will be visible:
7. Click on OK twice to return to Edit mode.
8. The Serial Wire Viewer is now configured.



SWV with ULINKpro:

Configuring SWV & ETM with ULINKpro:

1. Click on the Target Options icon.  Select the Debug tab. Select ULINKPro Cortex debugger.
2. In the Initialization File: box enter the initialization file LPC17xx_TPIU.ini. It is located at C:\Keil\ARM\Boards\Keil\MCB1700\Blinky_Ulp Use the Browse icon to locate and select it.
3. Click Settings on the right side of the window.
4. Make sure the SWJ box is checked. Select SW in the “Port:” box. You can use JTAG with the Trace Port.
5. Click on the Trace tab. Select Trace Enable. (do not select ETM Trace Enable just yet).
6. Enter 100 for Core Clock: and check the Trace Enable and EXCTRC boxes. Uncheck the Periodic box.
7. Select Sync Trace Port with 4-bit Data. This is what allows the use of JTAG or SWD (SW) with ULINKpro.
8. Select ETM Trace Enable. Unselect Unlimited Trace. Click on OK twice.
9. The Serial Wire Viewer and ETM Trace are now configured.



SWV and ETM with ULINKpro:

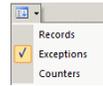
Note: This does not configure the Flash programmer. Click on the Target Options icon.  Select the Utilities tab. The CAN project for the MCB1700 board is pre-configured to use the ULINK2 or ULINK-ME.

Exception Tracing with ULINK2, ULINK-ME or ULINKpro:

1. Enter the Debug mode by clicking on the debug icon.  Select OK if the Evaluation Mode box appears.

2. Click on the RUN icon. 

3. Click on the arrow on the Trace icon and select Exceptions:

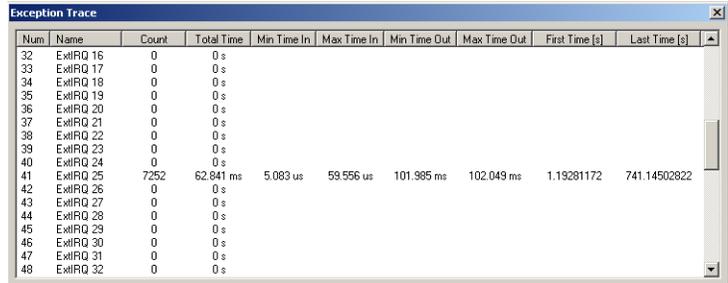


4. The window below appears. Note all exceptions (RESET, NMI etc.) are listed.

5. Note Exception 15 SysTick timer is activated. Scroll down to Num: 41 ExtIRQ 25:

6. Displayed is the LPC1700 CAN exception. CAN_IRQHandler in Can.c is being triggered. This is shown here:

7. This window shows how many times this exception has occurred with various times. This is a very useful feature to debug IRQs and determine how often they are triggered or if not at all because of a bug in the program.



Num	Name	Count	Total Time	Min Time In	Max Time In	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
32	ExtIRQ 16	0	0 s						
33	ExtIRQ 17	0	0 s						
34	ExtIRQ 18	0	0 s						
35	ExtIRQ 19	0	0 s						
36	ExtIRQ 20	0	0 s						
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	7252	62.841 ms	5.083 us	59.556 us	101.985 ms	102.049 ms	1.19281172	741.14502822
42	ExtIRQ 26	0	0 s						
43	ExtIRQ 27	0	0 s						
44	ExtIRQ 28	0	0 s						
45	ExtIRQ 29	0	0 s						
46	ExtIRQ 30	0	0 s						
47	ExtIRQ 31	0	0 s						
48	ExtIRQ 32	0	0 s						

8. Double-click in it to clear it.

TIP: Num is the exception number: RESET is 1. External exceptions start at Num 16. 41 is the CAN IRQ as described in the LPC17xx Users Manual. Num 41 is also known as 41-16 = External IRQ 25.

9. Open View/Trace and select Trace Records. Note the window is filled with SysTick exceptions. We can't see CAN.

To Turn off the SysTick timer:

10. In CanDemo.c, comment out this line near 43: `while ((mstTicks - curTicks) < dlyTicks).`

11. Add this after line 43: `for (curTicks = 0; curTicks < (dlyTicks * 0x1000); curTicks++) __NOP();`

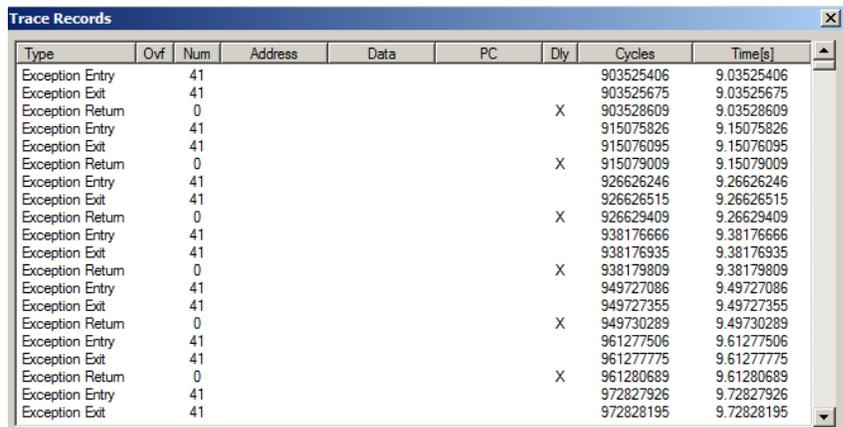
12. Comment out this line near 113: `sysTick_Config(SystemCoreClock / 1000);`

13. Exit Debug mode  and rebuild the project . Program the Flash 

13. Enter Debug mode  and click on RUN 

14. The Trace Records window will look like this:

15. Note from one Exception Entry to the next is about 14.8 ms which is indicated in the Exceptions window as Min & Max Time OUT.



Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Exception Entry		41					903525406	9.03525406
Exception Exit		41					903525675	9.03525675
Exception Return	0					X	903528609	9.03528609
Exception Entry		41					915075826	9.15075826
Exception Exit		41					915076095	9.15076095
Exception Return	0					X	915079009	9.15079009
Exception Entry		41					926626246	9.26626246
Exception Exit		41					926626515	9.26626515
Exception Return	0					X	926629409	9.26629409
Exception Entry		41					938176666	9.38176666
Exception Exit		41					938176935	9.38176935
Exception Return	0					X	938179809	9.38179809
Exception Entry		41					949727086	9.49727086
Exception Exit		41					949727355	9.49727355
Exception Return	0					X	949730289	9.49730289
Exception Entry		41					961277506	9.61277506
Exception Exit		41					961277775	9.61277775
Exception Return	0					X	961280689	9.61280689
Exception Entry		41					972827926	9.72827926
Exception Exit		41					972828195	9.72828195

TIP: ULINKpro and J-Link have a different Trace Records window.

TIP: Double-click inside any trace window to clear it.

TIP: All of these features and more are included with the Keil MDK. There is nothing more to purchase. SWV works with the Keil ULINK2, ULINK-ME, ULINKpro and J-Link (V 6 and later black case).

TIP: - Exception Entry is when the exception occurs.

- Exception Exit is when the exception returns.

- Exception Return is when the ALL exceptions return.

The "x" in Dly (delay) means the timestamp was delayed and might not be accurate due to overloading of the SWO pin.

An "x" in the Ovf (overflow) column means at least one frame was lost for probably the same reason.

Data Tracing: displaying the data writes...

The ARM CoreSight technology with the LPC1700 family also includes data tracing – of course it is also in real-time.

1. Open Debug/Debug Settings and select Trace. Unselect EXCTRC and select “on Data R/W sample”. Click OK.
2. Select OK when a window opens asking to stop the program and take new values.
3. Click on the Logic Analyzer icon  Click on Setup then click the insert icon. .
4. Enter the global variable val_Tx. Set the Display Range Max: to 0xFF. Click on Close.

TIP: You can drag and drop a variable from a source window or manually enter a variable or right click on it and add to LA.

TIP: This action also causes the variable val_Tx to be displayed in the Trace Records window. You can add four variables.

5. A Logic Analyzer window like the one shown below will open. Click on **Out** or **In** until **Range:** equals 1 s or so.
6. Click Run.  Rotate the pot P7. Shown here is an example display that results.
7. Open View/Trace and select Trace Records. You can select this from the Trace icon too. .
8. What we are seeing in the Trace Records columns:

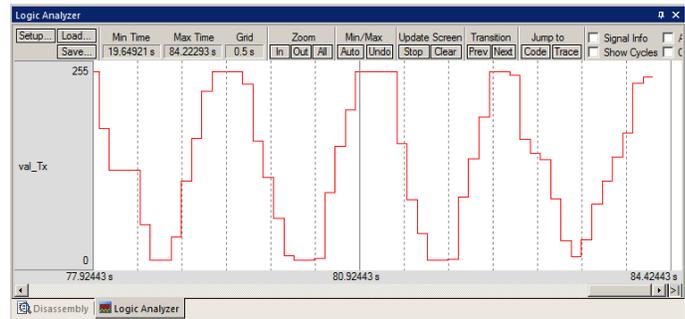
Address: The address of the variable val_Tx.

Data: The data values being read or written.

PC: the address of the instruction responsible for the read or write cycle. This was activated by the setting “on Data R/W Sample”.

Cycles and Time: when the operations occurred.

Time(s) comes from the box t1: in the main µVision window. The X in DLY indicates a delay in the timestamp.



Eliminate the Dly “x”:

1. Open Debug/Debug Settings and click on the Trace tab. Unselect on Data R/W Sample.
2. Click RUN. Double-click to clear the Trace Records window.
3. Note the Dly column is probably clear now. The SWO pin is not so heavily loaded now.

Display the CAN exception:

4. Open Debug/Debug Settings and click on the Trace tab. Select EXCTRC. You will have to click Run again.
5. Double-click in the Trace Records window and note that Exceptions are also displayed.

TIP: It is not possible to send everything out the SWV. Overflows are common and normal with SWV and it is good practice to limit what information you ask to be sent out. Just limit how many boxes you check to necessities. Overflows are indicated in the Ovf column in the Trace Records window. µVision recovers gracefully from overflows.

TIP: Right click in the Trace Records window and you can filter out those types of records you do not want to see.

Note: Currently only Data Write frames are displayed. Data Read frames are not displayed to reduce the traffic on the SWO pin.

This might be added soon to µVision.

TIP: If SWV stops and µVision stops and the ULINK2 or ME must be disconnected to recover: your laptop may have a USB bug. Use a PCMCIA USB adapter or try another computer. T1: will stop incremented. Desktop computers and Windows 7 is not affected.

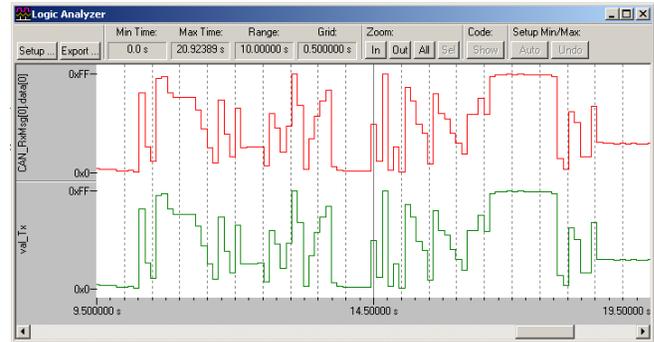
Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			10000228H	00000052H	0000183AH	X	2000778281	20.00778281
Data Write			10000228H	00000052H	0000183AH	X	2012328821	20.12328821
Data Write			10000228H	00000052H	0000183AH	X	2023879361	20.23879361
Data Write			10000228H	00000052H	0000183AH	X	2035429901	20.35429901
Data Write			10000228H	00000052H	0000183AH	X	2046980441	20.46980441
Data Write			10000228H	00000052H	0000183AH	X	2058530981	20.58530981
Data Write			10000228H	00000052H	0000183AH	X	2070081521	20.70081521
Data Write			10000228H	00000052H	0000183AH	X	2081632061	20.81632061
Data Write			10000228H	00000052H	0000183AH	X	2093182601	20.93182601
Data Write			10000228H	00000052H	0000183AH	X	2104733141	21.04733141
Data Write			10000228H	00000052H	0000183AH	X	2116283681	21.16283681
Data Write			10000228H	00000052H	0000183AH	X	2127834221	21.27834221
Data Write			10000228H	00000052H	0000183AH	X	2139384761	21.39384761
Data Write			10000228H	00000052H	0000183AH	X	2150935301	21.50935301
Data Write			10000228H	00000052H	0000183AH	X	2162485841	21.62485841
Data Write			10000228H	00000052H	0000183AH	X	2174036381	21.74036381
Data Write			10000228H	00000052H	0000183AH	X	2185586921	21.85586921
Data Write			10000228H	00000052H	0000183AH	X	2197137461	21.97137461
Data Write			10000228H	00000052H	0000183AH	X	2208688005	22.08688005
Data Write			10000228H	00000052H	0000183AH	X	2220238585	22.20238585

Data Tracing: a practical debugging example...

1. This is an example ran on a particular MCB1700 board. Val_Tx is displayed in the Logic Analyzer (LA).
2. The variable CAN_RxMsg[0].data[0] is also displayed in the Logic Analyzer as shown here:
3. With the program running, the potentiometer was varied. The screen displayed the value of val_Tx in real-time as shown here in the Logic Analyzer:

If you think this graph doesn't reflect what you think a pot rotated should look like you are correct ! The pot on this particular board is defective (it has many bad spots) and I have saved it for demonstrations like this. Turning the pot and watching the LCD seems to correctly display the digits.

However, watching the data writes in the Trace Records reveals what is really happening. See in the Data column as I rotate from 77 to C1 there is a short drop down to the 30s – this in one of the bad spots and this is why the graph has the sharp jumps. This was confirmed with a CAN analyzer attached to the bus and an oscilloscope on the pot.



Data Read	1000001CH	0000000FH	×	148247611168	2058.93459956
Data Read	1000001CH	0000000FH		148247650589	2058.99514707
Data Write	1000001CH	00000012H		148254915750	2059.09605208
Data Read	1000001CH	00000012H	×	148254916938	2059.09606858

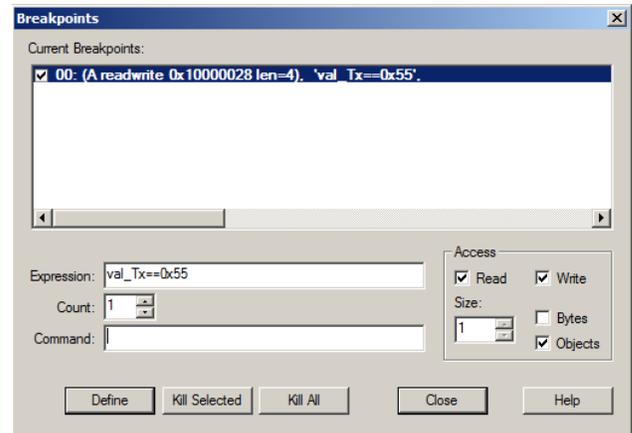
This is one example of the usefulness of Serial Wire Viewer. It could be otherwise very difficult to find this with without Serial Wire Viewer. Another good

use is finding interrupts that fire too often using up CPU time and slowing your program down. Viewing variables that do not exist in the real world and only in formulae (and therefore cannot be measured with instruments) is another.

Watchpoints: (also known as Access Breaks in Keil products)

Watchpoints provide breaks on data and address values. There are four Watchpoints in the LPC1700 series.

1. Enter Debug mode. Do not run the program yet.
2. Open Debug/Breakpoints and an empty window below opens up. You can also use Ctrl-B.
3. Enter in the field Expression: val_tx == 0x55. Don't hit Enter yet !
4. Check the Read box in Access and click on Define. The expression will move into Current Breakpoints:
5. Click on Close.
6. Open Debug/Debug Settings and click on the Trace tab. Select on Data R/W sample and unselect EXTRC.
7. RUN the program.
8. Adjust the pot and when Rx equals 0x55 the program will stop. This is how a Watchpoint works.
9. The last entry in the Trace Records will be the write of 0x55 as shown here:
10. Remove the Watchpoint using the Kill All button.



TIP: When the Breakpoints window is open you can modify a Watchpoint by double-clicking on it and it will move below into the Expression: box. When you are finished editing it click on Define again. Note that the old Watchpoint will still be listed. Click on it to highlight it and click on Kill Selected to remove it.

TIP: Click on Help while in the Breakpoints window and information will be displayed on other types of expressions.

Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			10000028H	00000059H	0000183AH	X	38742427042	387.42427042
Data Write			10000028H	00000059H	0000183AH	X	38753977614	387.53977614
Data Write			10000028H	00000059H	0000183AH	X	38765528182	387.65528182
Data Write			10000028H	00000059H	0000183AH	X	38777078754	387.77078754
Data Write			10000028H	00000059H	0000183AH	X	38788629322	387.88629322
Data Write			10000028H	00000059H	0000183AH	X	38800179894	388.00179894
Data Write			10000028H	0000005FH	0000183AH	X	38811730462	388.11730462
Data Write			10000028H	00000061H	0000183AH	X	38823280990	388.23280990
Data Write			10000028H	00000061H	0000183AH	X	38834831526	388.34831526
Data Write			10000028H	00000061H	0000183AH	X	38846382086	388.46382086
Data Write			10000028H	00000061H	0000183AH	X	38857932618	388.57932618
Data Write			10000028H	00000061H	0000183AH	X	38869483174	388.69483174
Data Write			10000028H	00000061H	0000183AH	X	38881033726	388.81033726
Data Write			10000028H	0000005DH	0000183AH	X	38892584258	388.92584258
Data Write			10000028H	0000005AH	0000183AH	X	38904134818	389.04134818
Data Write			10000028H	00000057H	0000183AH	X	38915685354	389.15685354
Data Write			10000028H	00000057H	0000183AH	X	38927235878	389.27235878
Data Write			10000028H	00000054H	0000183AH	X	38938786434	389.38786434
Data Write			10000028H	00000054H	0000183AH	X	38950336982	389.50336982
Data Write			10000028H	00000059H	0000183AH	X	38961887532	389.61887532

PC Tracing: Program Counter samples...

1. Open Debug/Debug Settings and click on the Trace tab. Unselect EXCTRC.
2. Select PC Samples. Click OK and click Run. Double-click in the Trace Records window to clear it.
3. Program samples will now be displayed with a timestamp in both CPU cycles and the time in seconds.

TIP: In this example every 16,384th PC is displayed. PC Samples are useful as they can give good indication where your CPU is spending its time. Use ETM trace with a ULINK pro if you need to see every instruction executed for advanced program flow debugging, Code Coverage, Execution Profiling or Performance Analysis.

Type	Ovt	Num	Address	Data	PC	Dly	Cycles	Time[s]
PC Sample					00001154H		38967131117	389.67131117
PC Sample					0000114AH		38967147501	389.67147501
PC Sample					00001156H		38967163885	389.67163885
PC Sample					00001154H		38967180269	389.67180269
PC Sample					0000115EH		38967196653	389.67196653
PC Sample					00001154H		38967213037	389.67213037
PC Sample					00001188H		38967229421	389.67229421
PC Sample					00001156H		38967245805	389.67245805
PC Sample					00001154H		38967262189	389.67262189
PC Sample					00001152H		38967278573	389.67278573
PC Sample					00001154H		38967294957	389.67294957
PC Sample					00001152H		38967311341	389.67311341
PC Sample					0000115EH		38967327725	389.67327725
PC Sample					00001156H		38967344109	389.67344109
PC Sample					00001152H		38967360493	389.67360493
PC Sample					00001154H		38967376877	389.67376877
PC Sample					00001156H		38967393261	389.67393261
PC Sample					00001154H		38967409645	389.67409645
PC Sample					0000115EH		38967426029	389.67426029
PC Sample					00001154H		38967442413	389.67442413

Watch and Memory windows: Updated in real-time...

The NXP LPC1700 is an ARM Cortex-M3 processor which incorporates CoreSight debugging technology. CoreSight provides a means to update the μ Vision Watch and Memory windows without stealing CPU cycles to do so.

TIP: In addition, you are able, in real-time, to insert values into the Memory window. Just click slowly on the element you want to change, enter the new value and press Enter. Try this on one of the data elements declared but not used.

Recall we have a structure `can_msg` that contains the CAN frame information. We have two instances of this: `CAN_TxMsg[2]` and `CAN_RxMsg[2]` which are arrays for each of the two CAN controllers.

1. Run the example program and confirm changing the pot changes the Rx and Tx values on the LCD display.
2. In a Watch window, double-click on <Enter expression> and enter `CAN_TxMsg`. Repeat with `CAN_RxMsg`.
3. In `CAN_RxMsg`, open to display `[0]` and `data[0]`. Note this value changes when the pot is rotated.
4. In `CAN_TxMsg`, open to display `[1]` and `data[0]`. Note this value changes when the pot is rotated.
5. In a memory window, enter `&CAN_TxMsg[1].data` and press Enter. You will see the appropriate memory locations change value as the pot is turned.

TIP: Note you can enter these variables when the program is still running. You can also highlight, drag and drop or right click on a variable and select Add *varname* to...

TIP: With CoreSight debug technology with the Cortex-M3, you can set/unset hardware breakpoints while the program is running.

This finishes a partial demonstration of the Serial Wire Viewer trace feature of Cortex-M processors. Visit www.keil.com for more information concerning the Serial Wire Viewer interface in Cortex-M processors.

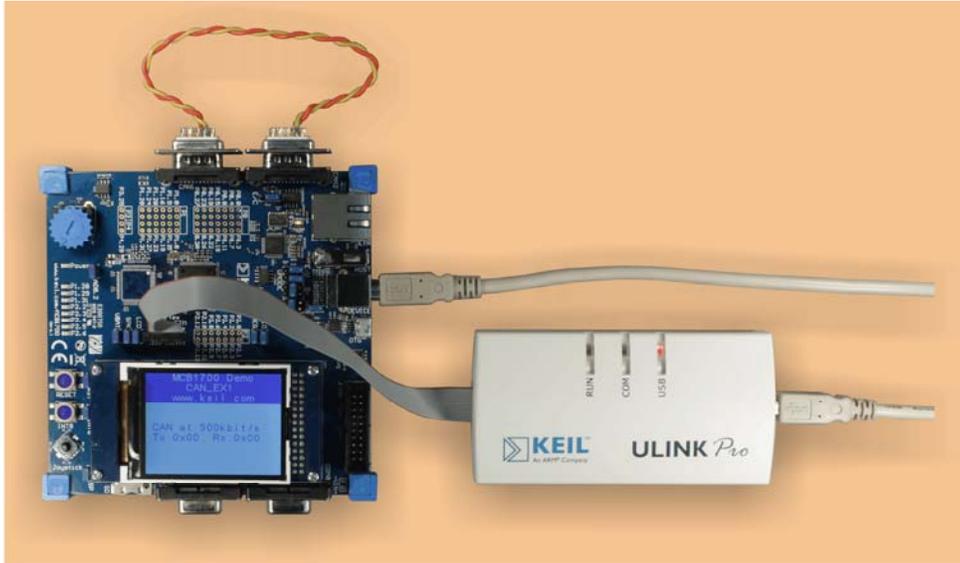
Address	Value
0x10000048	77777754 77777777 00000008 0000004D
0x10000058	77777754 77777777 00000008 00000000
0x10000068	00000000 00000000 00000000 34357830
0x10000078	00000000 00000000 00000000 00000000
0x10000088	00000000 00000000 00000000 00000000
0x10000098	00000000 00000000 00000000 00000000
0x100000A8	00000000 00000000 00000000 00000000
0x100000B8	00000000 00000000 00000000 00000000

Name	Value	Type
CAN_TxMsg	0x10000034 CAN_TxMsg	struct <untagged> [2]
[0]	0x10000034 &CAN_TxMsg	struct <untagged>
[1]	0x10000044	struct <untagged>
id	0x0000004D	unsigned int
data	0x10000048 "rwwwwwww"	unsigned char[8]
[0]	0x72 'r'	unsigned char
[1]	0x77 'w'	unsigned char
[2]	0x77 'w'	unsigned char
[3]	0x77 'w'	unsigned char
[4]	0x77 'w'	unsigned char
[5]	0x77 'w'	unsigned char
[6]	0x77 'w'	unsigned char
[7]	0x77 'w'	unsigned char
len	0x08	unsigned char
format	0x00	unsigned char
type	0x00	unsigned char
CAN_RxMsg	0x10000054 CAN_RxMsg	struct <untagged> [2]
[0]	0x10000054 &CAN_RxMsg	struct <untagged>
[1]	0x1000004D	struct <untagged>
id	0x0000004D	unsigned int
data	0x10000058 "rwwwwwww"	unsigned char[8]
[0]	0x72 'r'	unsigned char
[1]	0x77 'w'	unsigned char
[2]	0x77 'w'	unsigned char

ETM Tracing: Embedded Trace Macrocell

The NXP LPC1700 family has 4 bit ETM trace. Recall the Serial Wire Viewer provides PC Samples and exception events. Not all instructions executed are captured with PC Samples. ETM captures all the instructions executed with a timestamp. Data read and write operations are provided using the Serial Wire Viewer. ETM trace is associated with triggers and filters. Keil MDK provides ETM support with μ Vision using a ULINKpro. See www.keil.com/coresight.

The ETM is connected via a new ARM specified 20 pin connector. It is labeled Cortex Debug + ETM on the MCB1700. The ULINKpro is shown connected to it. A close-up of this connector is shown on page 24.

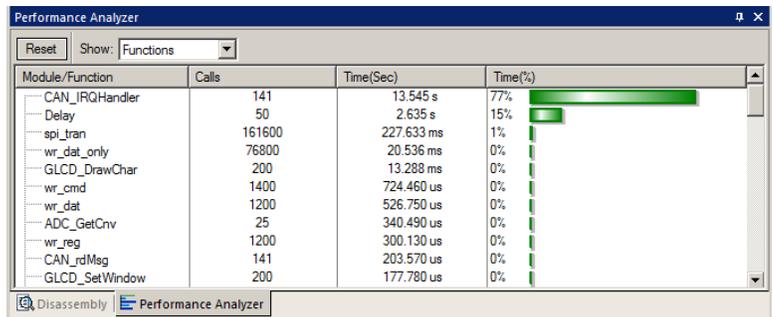
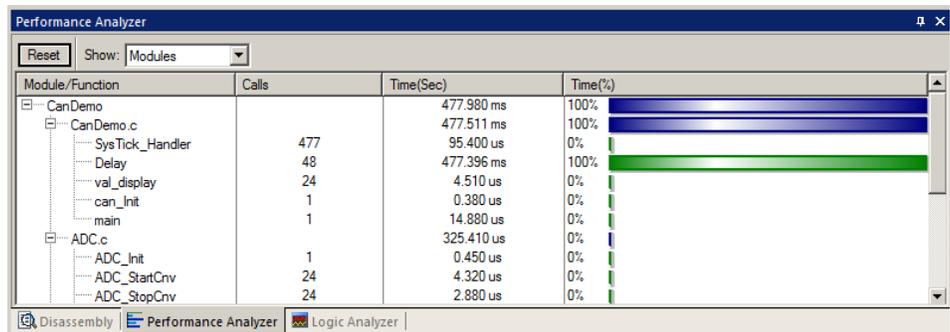


Performance Analyzer: *(this works with the Keil Simulator too)*

ETM records all the instructions executed. This information can be used to create an effective Performance Analyzer.

In the screen below, you can see the various modules in the CAN demo program. Displayed are how many times they have been called and the total time spent in each. You can also view functions as shown in the second window.

This window updates in real-time as the program is running.



Code Coverage: *(this works with the Keil Simulator too)*

The Disassembly window shows those instructions that were executed or not and whether a branch was taken or not. A green block means this instruction was executed, a darker grey means it was not executed. The orange block is a branch that was never taken and a cyan block (not shown) is a branch always taken. A branch where both conditions were satisfied is green.

Note that at 0x130A (orange) is a BNE to 0x1310 that was always taken. Therefore, 0x1310 was never executed.

Code Coverage is very important for certain approvals needed such as for the FDA. An instruction never executed has never been tested and therefore is a potential (and maybe dangerous) bug. A test case must be designed to test all code in a project.

The screenshot shows two windows from the Keil IDE. The left window is the Disassembly window, displaying the assembly code for the function `void CAN_rdmMsg (uint32_t ctrl, CAN_msg *msg) {`. The code is color-coded: green for instructions that were executed, orange for branches that were not taken, and grey for instructions that were not executed. The right window is the Trace Data window, showing a list of instructions executed during the run-time of the program. The columns are Time, Address / Port, Instruction / Data, and Src Code / Trigger Addr. The trace starts at address 0x1304 and ends at 0x1308, with a trace run between 0x1304 and 0x1308. The trace data shows the execution of instructions such as `PUSH {r4-r5,lr}`, `MOV r3,r0`, `LPC_CAN_TypeDef *pCAN = (ctrl == 1) ? LPC_CAN1 : LPC_CAN0;`, `CMP r3,#0x01`, `BNE 0x00001310`, `LDR r4,[pc,#688] ; @0x000015C0`, `B 0x00001312`, `LDR r4,[pc,#688] ; @0x000015C4`, `MOV r0,r4`, `LDR r2,[r0,#0x20]`, `LDR r2,[r0,#0x20]`, `LSRS r4,r2,#31`, `STRB r4,[r1,#0x0D]`, `LSRS r4,r2,#31`, `STRB r4,[r1,#0x0D]`, `UBFX r4,r2,#30,#1`, `STRB r4,[r1,#0x0E]`, `UBFX r4,r2,#16,#8`, `AND r4,r4,#0x0F`, `STRB r4,[r1,#0x0C]`, `STR r4,[r1,#0x00]`, `LDRB r4,[r1,#0x0E]`, `*CBNZ r4,0x0000133A`, `LDR r4,[r0,#0x28]`, `STR r4,[r1,#0x04]`, `LDR r5,[r0,#0x2C]`, `Exception Entry - ExtIRQ 25`, `Exception Exit - ExtIRQ 25`, and `Exception Return`.

Trace Triggering: *(Triggering currently is not supported in the Simulator)*

It is possible to start and stop the trace collection assembly or source code. The disassembly window above shows the function `CAN_rdmMsg` in `CAN.c`. On the right is the trace Data window showing in what order these instructions were executed.

The disassembly window shows the C code as it was written and the assembly instructions as the compilation result. The Trace Data window is a record of how it actually happened during run-time of the program.

The trace is set to start at the beginning of `CAN_rdmMsg` (0x1304) and stop on 0x1308. This is clearly visible in the Trace Data window. You can see the trace turn on at the beginning of `CAN_rdmMsg` and turn off at the end. You can also see the BNE at 130A not taking the branch to 0x1310 but rather the LDR at 0x130E is executed.

Note the three exception entries. In CoreSight, the instruction trace (ETM) comes from a different source than the SWV data of which exceptions is part of. In this case, we can tell that exception 25 happened but we do not know exactly how it relates to the assembly instructions. In this case, we know the when this exception happens, the exception handler in `CAN.c` is executed. We do not see this happening because the trace is only turned on when `CAN_rdmMsg` function is being executed.

We do know that if the exception happened during `CAN_rdmMsg`, the handler would also be recorded in the trace.

There are many other interesting and useful things to do with trace.

TIP: The Exception frames are displayed because the `TraceRun` and `TraceSuspend` affect only the instruction trace. These exceptions are part of the data trace. To stop the collection of both instruction and data frames, use `TraceHalt` instead of `TraceSuspend`:

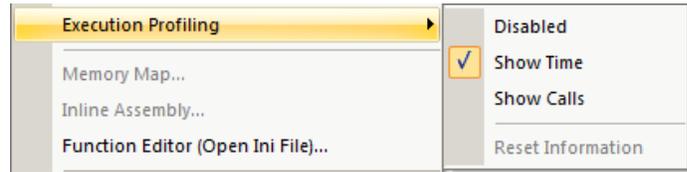


The ability to use the various trace features provides increased developer productivity. `µVision` trace configuration are constantly being improved and added. Check back often to see the latest improvements.

Execution Profiling: *(this works with the Keil Simulator too)*

Execution Profiling is used to display how much time a C source line took to execute and how many times it was called. This information is provided by the ETM trace. The μ Vision simulator also provides Execution Profiling.

1. Enter Debug mode.
2. Select Debug/Execution Profiling/Show Time.
3. In the left margin of the disassembly and C source windows will display various time values.
4. Click on RUN.
5. The times will start to fill up as shown below right:
6. Click inside the yellow margin of Blinky.c to refresh it.
7. This is done in real-time and without stealing CPU cycles.
8. Hover the cursor over a time and and more information appears as in the yellow box shown here:



Time:	Calls:	Average:
19.599 s	139910257 *	0.140 μ s

9. Recall you can also select Show Calls and this information rather than the execution times will be displayed in the margin.

```

207 0.050  $\mu$ s GLCD_SetTextColor(Blue);
208 #endif // __USE_LCD
209
210 0.033  $\mu$ s while (1) {
211 1.479 s AD_value = AD_last;
212 2.157 s if (AD_value != AD_last)
213 0.033  $\mu$ s AD_value = AD_last;
214
215 1.414 s if (AD_value != AD_print) {
216 #ifdef __USE_LCD
217 7.967  $\mu$ s GLCD_SetTextColor(Red);
218 10.817  $\mu$ s GLCD_Bargraph(9 * __FONT_W
219 6.117  $\mu$ s GLCD_SetTextColor(White);
220 #endif // __USE_LCD
221
222 3.175  $\mu$ s AD_print = AD_value;
223 6.350  $\mu$ s AD_dbg = AD_value;
224 }
225
226 /* Printf message with AD val
227 2.297 s if (clock_1s) {
228 0.500  $\mu$ s clock_1s = 0;
229 0.667  $\mu$ s sprintf(text, "AD value = 0
230
231
232 0.667  $\mu$ s Time: 0.667  $\mu$ s Calls: 9 * Average: 0.074  $\mu$ s);
233 0.875  $\mu$ s GLCD_DisplayString(5, 0, __
234 #endif // __USE_LCD
235 0.483  $\mu$ s printf("%s\r\n", text);
236 }
237
  
```

In-the-Weeds Example: *(only works with ETM and the Keil ULINKpro)*

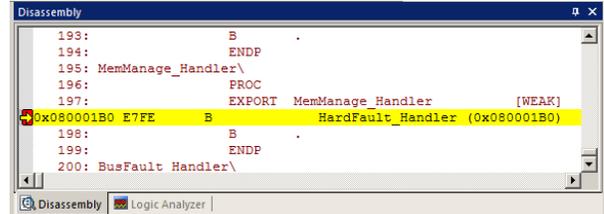
This example was not run on an MCB1700. If you have a ULINKpro you can replicate this on a MCB1700.

Some of the hardest problems to solve are those when a crash has occurred and you have no clue what caused this. You only know that it happened and the stack is corrupted or provides no useful clues. Modern programs tend to be asynchronous with interrupts and RTOS task switching plus unexpected and spurious events. Having a recording of the program flow is useful especially when a problem occurs and the consequences are not immediately visible. Another problem is detecting race conditions and determining how to fix them. ETM trace handles these problems and others quite easily. It is not hard to use.

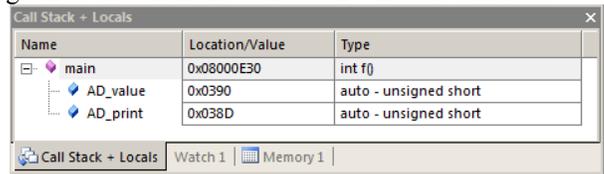
If a Hard Fault occurs, the CPU will end up at the address specified in the Hard Fault vector located at 0x00 000C. This address points to the Hard Fault handler. This is usually a branch to itself and this Branch instruction will run forever. The trace buffer will save millions of the same branch instructions. This is not useful. We need to stop the CPU at this point.

This exception vector is found in the file startup_LPC17xx.s. If we set a breakpoint by clicking on the Hard Fault handler and run the program: at the next Hard Fault event the CPU will jump to the Hard Fault handler (in this case located at 0x0800 01B0 as shown to the right) and stop.

The CPU and also the trace collection will stop. The trace buffer will be visible and extremely useful to investigate and determine the cause of the crash.



1. Open the Blinky_Ulp example, rebuild, program the Flash and enter Debug mode. Open the Data Trace window.
2. Locate the Hard fault vector near line 207 in the disassembly window or in startup_LPC17xx.s.
3. Set a breakpoint at this point. A red block will appear as shown above.
4. Run the Blinky example for a few seconds and click on STOP.
5. Click on the Step_Out icon  to go back to the main() program as shown in the Call Stack + Locals window:
6. In the Disassembly window, scroll down until you find a POP instruction. I found one at 0x0800 1256 as shown below in the third window:
7. Right click on the POP instruction (or at the MOV at 0x0800 124E as shown below) and select Set Program Counter. This will now be the next instruction executed.
8. Click on RUN and immediately the program will stop on the Hard Fault exception branch instruction.
9. Examine the Data Trace window and you find this POP plus everything else that was previously executed. In the bottom screen are the four MOV instructions plus the offending POP.
10. How would you ever find this cause without instruction trace ?
11. Note the Branch at the Hard Fault does not show in the trace window because a hardware breakpoint does execute the instruction it is set to therefore it is not recorded in the trace buffer.
12. Click Step One Line (single step) and these Branch instructions will be recorded in the Trace Data window as they are executed.



```

0x08001248 F1A40401 SUB     r4,r4,#0x01
0x0800124C DCFD   BGT     0x0800120E
0x0800124E 4648   MOV     r0,r9
0x08001250 4631   MOV     r1,r6
0x08001252 462A   MOV     r2,r5
0x08001254 4643   MOV     r3,r8
0x08001256 E8BD9FF0 POP     {r4-r12,pc}
0x0800125A 0000   MOVS   r0,r0
    
```

Time	Address / Port	Instruction / Data	Src Code / Trigger Addr
	X: 0x0800DD4	LDRB r1,[r0,#0x00]	
	X: 0x0800DD6	CBZ r1,0x0800DE0	
1.215 414 190 s	X: 0x0800DE0	MOV r0,#0xFFFFFFFF	return -1; /* Conv. in pro...
1.215 414 210 s	X: 0x0800DE4	BX lr	
	X: 0x0800124E	MOV r0,r9	
	X: 0x08001250	MOV r1,r6	
	X: 0x08001252	MOV r2,r5	
	X: 0x08001254	MOV r3,r8	
1.215 420 300 s	X: 0x08001256	POP {r4-r12,pc}	

The frames above the POP are a record of all previous instructions executed and tells you the complete program flow.

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 connect from ground to one of the CAN leads but the signals will be lower. You can connect to the Tx output on the controller or Tx input to the transceiver.
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 as accurately as you can.
4. Invert this value (divide into 1) and you have the CAN speed in bits per second. i.e. $2 \mu\text{sec} = 500 \text{ Kbps}$.

Four Newbie mistakes you can avoid:

1. You need at least two CAN nodes in order to get past the first initial message. *This is a very common mistake.*
2. There is a bug in CAN that was turned into a feature. What happens is some noise causes a transmitter to think its first message was corrupted and it sends out a copy. However, the other nodes think the first message was valid. Therefore they see two identical and valid messages. Therefore do not increment or toggle values or states. Send the actual value you want to nodes to receive. This situation occurs rarely but can be a problem.
3. The CAN controller will add (or not) a bit to the bitstream to ensure there are never more than five unchanging bits. This changes the CAN message length. When such messages are viewed on an oscilloscope – they look like jitter. It is not: this is how CAN works. Do not chase problems in your network on a false assumption you have jitter.
4. Avoid doing tricky and complicated things like changing the CAN frequency. Just keep it simple and stable.
5. *One more tip just for luck:* Do not try and design your own CAN controller. There are secrets that will stop you.

Useful Documents:

1. **The Definitive Guide to the ARM Cortex-M3** by Joseph Yiu. (he has one for the Cortex-M0) Search the web.
2. **MDK-ARM Compiler Optimizations: Appnote 202:** www.keil.com/appnotes/files/apnt202.pdf
3. **Lazy Stacking and Context Switching Appnote 298:** www.arm.com and search for 298
4. **A list of resources is located at:** <http://www.arm.com/products/processors/cortex-m/index.php>
Click on the Resources tab. Or search for “Cortex-M3” on www.arm.com and click on the Resources tab.
5. **ARM Infocenter:** <http://infocenter.arm.com> Look for Application Notes and Cortex-M listings.

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. You can use the Cortex-M 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.

Keil has evaluation boards for the LPC2300, LPC1700, LPC1100, LKPC1800 and LPC4300 families with CAN examples. See www.keil.com/txp for more information.

Keil completely supports the new NXP Cortex-M0 and Cortex-M0+ processors as well as many NXP 8051 devices with the same μ Vision. Please visit www.nxp.com/microcontrollers/ for information on these processors.

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 MCB1700 you can connect up to any CAN network and communicate with it.

Keil offers a complete CAN stack for all ARM7TM, ARM9TM and Cortex-M4/M3/M1/M0/M0+ processors. This comes as part of MDK Professional. It was previously known as RL-ARMTM. Please visit www.keil.com/rl-arm/ for details.

This middleware package contains USB, TCP/IP networking, Flash file system and the CAN interface.

The Keil RTXTM RTOS now comes with a BSD type license. Source is provided. See www.arm.com/cmsis.



How can trace help me find problems ?

Trace, either SWV or ETM, adds significant power to debugging efforts. Tells you where the program has been, how it got there, how long it took, when did the interrupts fire and all about data reads and writes.

- With RTOS and interrupt driven events – many programs are now asynchronous. Trace helps sort this out and provides for the dynamic analysis of running code.
- Putting test or printf code in your project sometimes changes or erases the problem. Trace is non-intrusive.
- Trace can often find nasty problems very quickly. Weeks or months can be replaced by minutes. *Really!*
- ...especially where the bug occurs a long time before the consequences are seen.
- Or where the state of the system disappears with a change in scope(s).
- Plus – you don't have to stop the program to see the trace. This is crucial to some applications.
- No trace availability is responsible for unsolved bugs – some of these problems are too hard to find without it.
- Pointer problems. Is your pointer really reading or writing what you think it is? Where is it pointing to?
- Illegal instructions and data aborts (such as misaligned writes).
- Code overwrites – writes to Flash, unexpected writes to peripheral registers.
- Profile Analyzer. Where is the CPU spending its time? Where should I start to optimize my program?
- Code Coverage. Can be a certification requirement. Was this instruction executed?

Serial Wire Viewer and ETM Trace Summary:

Serial Wire Viewer can see:

- Global variables.
- Static variables.
- Structures.
- Peripheral registers – just read or write to them.
- Can't see local variables. (just make them global or static).
- Can't see DMA transfers – DMA bypasses CPU and CoreSight by definition.

Serial Wire Viewer displays in various ways:

- PC Samples.
- Data reads and writes.
- Exception and interrupt events.
- CPU counters.
- Timestamps for these.

These are the types of problems that can be found with a quality ETM trace:

- Pointer problems.
- Illegal instructions and data aborts (such as misaligned writes).
- Code overwrites – writes to Flash, unexpected writes to peripheral registers (SFRs), a corrupted stack.
How did I get here?
- Out of bounds data. Uninitialized variables and arrays.
- Stack overflows. What causes the stack to grow bigger than it should?
- Runaway programs: your program has gone off into the weeds and you need to know what instruction caused this. Is very tough to find these problems without a trace. ETM trace is best for this.
- Communication protocol and timing issues. System timing problems.

Keil Products:

Keil Microcontroller Development Kit (MDK-ARM™)

- MDK-Lite™ (Evaluation version) \$0
- MDK-Basic™ (256K Compiler Limit, No debug Limit) - \$2,695
- MDK-Standard™ (unlimited compile and debug code and data size) - \$4,895
- MDK-Professional™ (Includes Flash File, TCP/IP, CAN and USB driver libraries) \$9,995

For special promotional pricing and offers, please contact Keil Sales for details.

All versions, including MDK-Lite, includes Keil RTX RTOS *with source code* !

Call Keil Sales for more details on current pricing. All products are available.

All products include Technical Support for 1 year. This can easily be renewed.

Call Keil Sales for special university pricing.

For the ARM University program: go to www.arm.com and search for university.

USB-JTAG adapter (for Flash programming too)

- ULINK2™ - \$395 (*ULINK2 and ME - SWV only – no ETM*)
- ULINK-ME™ – sold only with a board by Keil or OEM.
- ULINKpro™ - \$1,395 – Cortex-Mx SWV & ETM trace

Note: USA prices. Contact sales.intl@keil.com for pricing in other countries.

Prices are for reference only and are subject to change without notice.

For the entire Keil catalog see www.keil.com, contact Keil Sales or your local distributor.



For more information:

Keil products can be purchased directly from ARM or through various distributors.

Keil Distributors: See www.keil.com/distis/ or www.embeddedsoftwarestore.com

Keil Direct Sales In USA: sales.us@keil.com or 800-348-8051. **Outside the US:** sales.intl@keil.com

Keil Technical Support in USA: support.us@keil.com or 800-348-8051. **Outside the US:** support.intl@keil.com.

For comments or corrections please email bob.boys@arm.com.

For the latest version of this document, see www.keil.com/nxp

