CAN Primer: Creating your own Network

Summer 2009 Version 1.0

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

for the NXP LPC1700 series

Introduction:

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

A CAN controller is a sophisticated device. Nearly all the features of the CAN protocol described below are automatically handled by the controller with almost no intervention by the host processor. All you need to do is configure the controller by writing to its registers, write data to the controller and the controller then does all the housekeeping work to get your message on the bus.

The controller will also read any frames it sees on the bus and hold them in a small FIFO memory. It will notify the host processor that this data is available which you then read from the controller. The controller also contains a hardware filter mechanism that can be programmed to ignore those CAN frames you do not want passed to the processor.

INDEX:

Introduction: 1
Main Features of CAN: 2
CAN System Layout: 2
CAN Node schematic: 3
Physical Layer: the wires: 3
The CAN Frame: the fields: 4
Bus loading, Speed and Errors: 5
Bus Faults: 6
Multiple CAN frames, Types of Frames and Time-outs: 6
Transmitting and Receiving CAN Frames: 7
CAN Controllers: 8
Keil CAN demonstration software: 9
The NXP CAN Controller: 9
Keil CAN Example using the Simulator: 10
How the Keil CAN software works: 11-13
Setting up a CAN demonstration with real hardware: 14
Using the Serial Wire Viewer (SWV) with CAN: 14
SWV Exception Tracing: 15
SWV Data Read and Write Tracing: 16
Watchpoints, SWV PC Tracing and Watch and Memory Windows: 17
ETM Tracing Introduction: 18
ETM Tracing with µVision® and the Signum Systems JtagJetTrace: 19-21
How can trace help me find problems? 22
For more information: 22

 © 2009 Keil™ All rights reserved

Main Features of CAN:

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

1. 11 or 29 bit ID and from zero to 8 data bytes. **TIP:** These can be dynamically changed "on the fly".
2. Peer to Peer network. Every node can see all messages from all other nodes but 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. A 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 sometimes must endure large ground loops that can compromise signal integrity. CAN is designed using its differential pair to ignore ground voltage differences of many volts.

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.

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).

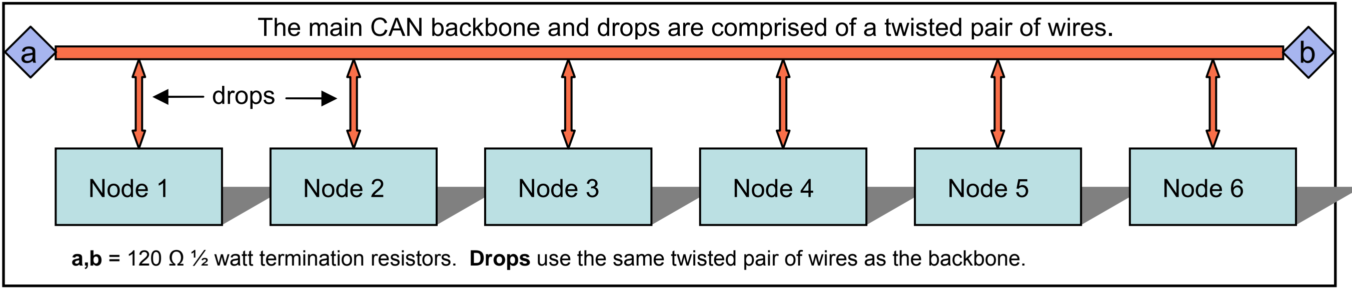
TIP: The drops should be less than 3 feet and randomly spaced to reduce standing waves. These issues all become more important at higher bus speeds. CAN is completely described in ISO 11898.

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 be 60 ohms.

CAN is a broadcast system. Any node can "broadcast" a message using a CAN frame on a bus that is in idle mode. Every node will see this message. A "message" can be considered the same as a CAN frame until you need to use more than one frame to send a long message.

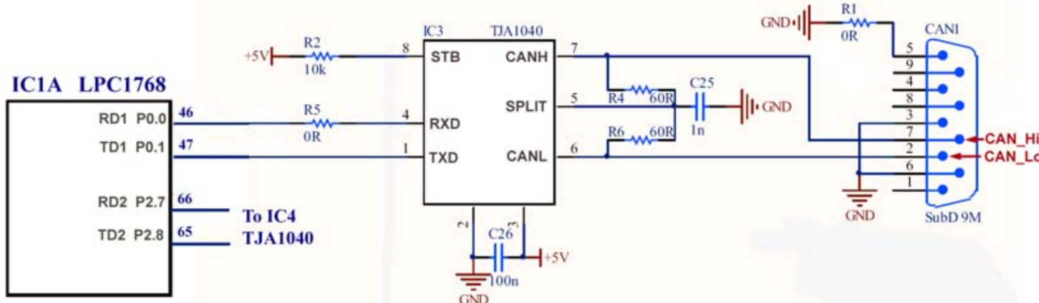
TIP: It is up to a receiving node if it must keep or ignore a frame. This can be handled in either your software or with the CAN controller acceptance filters.



A Node Schematic:

This is the schematic diagram from the Keil MCB1700™ evaluation board. IC3 is a TJA1040 NXP 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).

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.



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 hardware for this feature.

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 and this offers greater noise immunity; especially at higher CAN speeds.

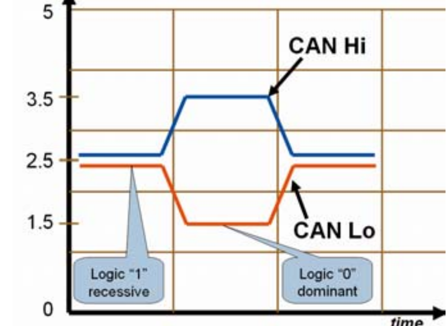
The MCB1700 evaluation board is meant to be used by itself as a small test network. It contains two separate CAN nodes and connectors. If this board is used as a node with other boards, and is not at one of the ends the resistors should be removed. Connector CAN1 corresponds to a generally accepted standard 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.

Physical Layer: *the wires and the voltages…*

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

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 and will present fewer design problems.

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



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

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

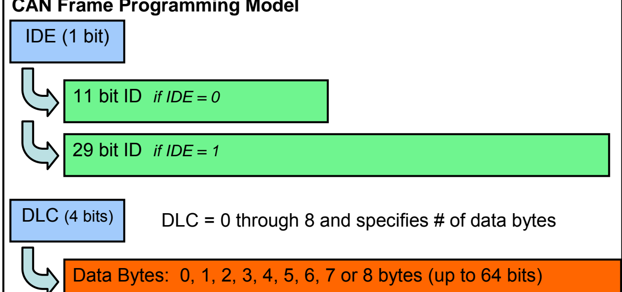
TIP: How to determine the frequency of a CAN signal: This is the best and sometimes only way to determine this.

1. Connect an oscilloscope hot lead to CAN Hi and its ground to CAN Lo. The scope ground must be isolated from the CAN ground. You can 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.

 © 2009 Keil™ All rights reserved

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

ID: Identifier: 11 or 29 bits

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

1. If two or more CAN messages are put on the bus at the same time; the one with the highest priority (the lowest value) ID will immediately get through. The others will be delayed and will be resent.
2. An ID of 0 has the highest priority and will always get through. An 11 bit ID has priority over any 29 bit ID.
3. You can 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. **TIP:** Note that CAN controllers can be configured to pass only certain messages to its host processor. Choose your ID values carefully to take advantage of this if needed. This can reduce the workload of a node's CPU.
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.
7. **TIP:** Make sure two nodes will ***never*** send the same ID value at the same time. It is illegal but possible to do this. If two messages sent at the same time are identical, they will be seen as one. If the data bytes are different, a bus error will result and the frames will be resent continuously and havoc is created on the bus.

Data Bytes:

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

1. You can have any number of data bytes mixed on the CAN bus. The controller can easily sort this out.
2. **TIP:** If you always use only one number of data bytes, your software will be much simpler to write.
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 use 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 is configured by the user software.

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

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

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

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 your frequency, ID or data bytes values, but it will tell you if you are putting out something.

1. Connect your CAN node with a transceiver connected to the CAN controller with a termination resistor.
2. Do not connect any other node or test tool. Just one node running by itself with at least one 120 ohm resistor.
3. Connect an oscilloscope hot lead to CAN Hi and ground to CAN Lo. The scope ground must be isolated from the CAN ground. You do not need a high speed scope – almost any will suffice. You can also connect a scope to the CAN controller output pin and ground for a very clean signal. But you need the controller connected. It is possible to do this test without a transceiver by shorting TXD to RXD and attaching the scope here.
4. Configure your CAN controller and write the IDE, ID, DLC and any data bytes into the appropriate register.
5. Any CAN frames will now be continuously displayed on the scope. RESET the processor to start over.

TIP: You can measure the CAN frequency with the method described in the **TIP:** under Physical Layer on Page 3.

Bus Loading:

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

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.

Bus Speed:

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

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

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. (it is usually zero)
6. IF not, it attempts to retransmit its message which has to go through the priority process with other messages.
7. All other nodes abandon reading the current frame, and adds 4 to each 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 or the transceiver chip.

TIP: Don't forget that if something happens to the integrity of your twisted pair, such as CAN Lo disconnected; it might still work but with greatly reduced noise immunity (that is what differential signals do best). If your network is in a very noisy environment, there might be 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. It can take a controller RESET or a certain number of good frames received.

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 an automatic mechanisms for this. Not all transceiver chips implement all of them. You can usually short CAN Lo to ground (ISO 11898 says can short Hi also) or open one CAN line. The ground needs to be connected for 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 work.

These will be detected as a bus error as described above. At least one node must try to transmit a frame in a bus fault condition to trigger a bus error. A bus in idle mode can't trigger a bus error. When the bus fault is removed, in many systems the network will come back 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 gets cancelled out and the 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.

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 is the 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. This is an example where one message is comprised of many CAN frames.

2) Periodic, Request/Response and Command Frames:

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

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

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 three types of traffic depending on your system's needs.

3) Time-outs:

Automotive CAN networks use time-outs and this concept is easily and effectively transferred to systems in other fields. A time-out occurs when a node fails to respond to a request in a timely fashion. Time-outs are handled completely by software 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.

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

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

Sequence of Transmitting Data on the CAN Bus:

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

Transmitting Notes:

- **How does a node know when it should transmit a message ?** Easy – you create the CAN frame you want to send by loading up the IDE, ID, DLC and any data byte registers in the CAN controller and then, in most controllers, you set a bit that triggers sending the frame as soon as legally possible. After this, the controller takes care of sending all frame bits. Until the controller signals otherwise to the processor, you can assume the message was sent.
- **What if there is an error ?** All nodes, including the transmitting node, monitor the bus for any errors. If a error condition is detected – a node or nodes signify to the other nodes there is an error by holding the bus at logical 0 for at least 6 bus cycles. At this point, all nodes take appropriate action. The message being sent (and now aborted) will be resent but only for a certain number of times. See Bus Errors on page 5.
- **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 will be along shortly.

Sequence of Receiving data from the CAN Bus:

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

Receiving Notes:

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

CAN Controllers and their Errata Sheets:

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

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

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 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 that can save some serious cash and time.

Typical sources for inexpensive tools are SYS TEC (www.phytec.com), 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 on the Internet to find these.

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

Bit Stuffing:

The CAN protocol states that when there are 5 consecutive bits of the same polarity, one bit of opposite polarity will be inserted to 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.

Conclusion:

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

Now, on the next few pages, let us look at how we can program a real CAN controller to transmit and receive messages. There are some hands-on experiments you can try – the Keil evaluation software is free and for some exercizes no hardware is needed. For the others, you will need an evaluation board with a NXP1700 processor and a Keil ULINK2, ULINK-ME and for ETM tracing either a ULINK*Pro* or a Signum JtagJetTrace USB to JTAG adapter.

For more information relating to CAN please see http://dgtech.com/pdfs/techpapers/primer.pdf
For information regarding testing CAN networks: http://dgtech.com/pdfs/techpapers/CIA_article.pdf
For CANopen and other information see: www.can-cia.org/ Most CANopen docs are free while most others charge.

A differential twisted pair of wires with two 120 Ω termination resistors: This is the minimum network of 2 nodes. The MCB1700 requires such a cable but without resistors as they are provided.

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 Keil® Microcontroller Development Kit (MDK-ARM) for the NXP LPC1700 ARM® Cortex™-M3 microcontroller. 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 ULINK*Pro*, Signum Systems JtagJetTrace or a Segger J-Trace.

You can download the latest version of MDK-ARM at: www.keil.com/demo and select ARM.

There is no charge for this software. Please install this software on your PC. Use MDK version 3.60 or later.

Keil also provides software tools for NXP ARM7, ARM9, Cortex-M3 and Cortex-M0 processor-based devices and most 8051 processors. Many of these have CAN controllers.

NXP microprocessors have 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 including Serial Wire Viewer (SWV) and ETM Embedded Trace Macrocell. Examples are provided showing trace with these powerful debugging tools.

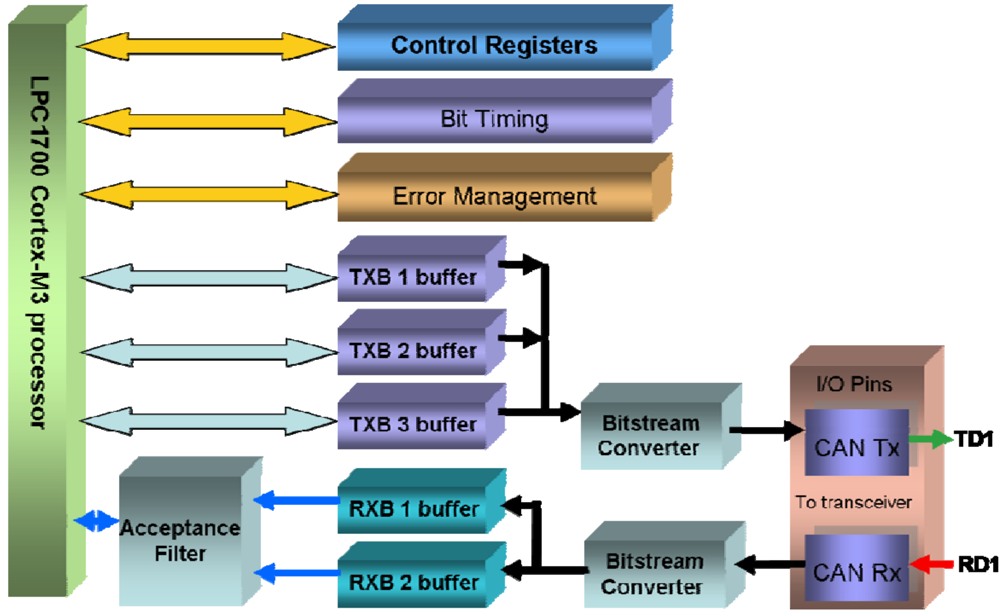
NXP CAN Controller for Cortex-M3 Processors.

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, data (if any) and the DLC go here.
4. Acceptance Filter: This passes only specified messages to the processor via the FIFOs. By default at RESET, these filters pass all messages to the FIFOs. Your software must configure them to filter messages.
5. 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.



 © 2009 Keil™ All rights reserved

Keil Example CAN Program: (using the Keil simulator in µVision4)

1. Start µVision4 by clicking on its icon on your Desktop. (See the note below concerning µVision3).
2. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\Keil\MCB1700\CAN\CAN.Uv2 (or .uvproj).
3. Make sure "Simulator" is selected in the Target window.
4. Compile the source files by clicking on the Build icon. . They will compile with no errors and 1 warning. The one warning is a placeholder for some optional user code. Ignore this warning as it will not affect you.
5. Click on the Options for Target icon. Select Debug tab and confirm "Use Simulator" is selected. Click OK.
6. Enter the Debug mode by clicking on the debug icon. Select OK if the Evaluation Mode box appears.
7. Position the Toolbox, "CAN: Communication" and "CAN: Controller" windows as appropriate. The CAN: Controller windows can be opened if not already in Peripherals/CAN as CAN Controllers 1 & 2.
8. Click on the RUN icon. Note: you stop the program with the STOP icon.
9. Note CAN messages with an ID of 0x21 will appear in the CAN: Communications window. You can see only the transmit frames if CAN loopback OFF is active. Click CAN loopback ON to see both transmit and receive frames as shown below. You should be familiar with most of the columns in CAN communication. **Important Note:** the first time you run this program there will be a delay of more than 10 seconds as the LCD routine runs. See the **Speed TIP:** below to speed things up.
10. Controller CAN1 is used to transmit frames and CAN2 to receive them. This is our minimum two node network.
11. In the Toolbox window, click on the "Analog Sweep 0…3.3v".
12. Changing data values representing output from the A/D convertor will now appear in the CAN data field in the CAN Communication window as shown below.

Speed TIP: To speed the program up: In the function main() in the files CanDemo.c, comment out every line starting with GLCD (10 of them) and the line val_display (); and rebuild the project. It will run much faster.

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.

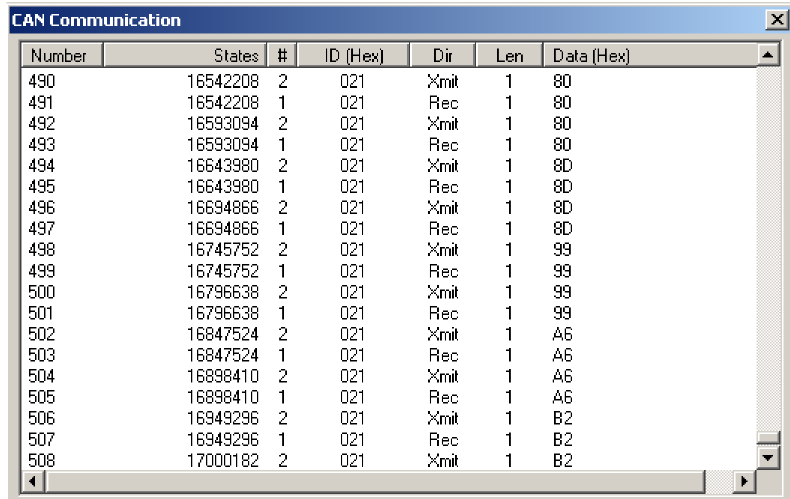
TIP: If you have more than one CAN controller in your processor you can operate these as parallel receivers. Divide the messages up with the Acceptance Filters. This will help capture all the messages on a very busy bus without losing any. Each CAN controller will handle its share of the messages. This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

µVision4: At the time of this writing, in Summer 2009, Keil is migrating to µVision4 which this document uses in the examples. At this time, the CAN project is distributed in MDK in µVision3 format.

µVision4 will import µVision 3 projects (.Uv2) files and save them as .uvproj. It will add the .bak extension to the original .Uv2 files. µVision4 can also export files in the µVision3 format.

A µVision4 add-on for older versions of Keil MDK is available at www.keil.com. MDK ships only µVision4 starting with MDK 4.0.

Do not confuse MDK 4.0 with µVision4. MDK is the entire tool chain of which µVision is one component.



CAN Communication

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

1) Can.h

The CAN Structure CAN_Msg: (*lines 23-29, 42 & 43*)

Shown is the structure declaration in Can.h. You should now be able to recognize each of these elements. You can enter either an 11 or 29 bit identifier. Two instances of CAN.msg are invoked 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 prototypes for functions used in Can.c are listed in Can.h in lines 29 to 35. These are visible in µVision:

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

2) Can.c

Configuring the CAN Controller: (Can.C)

There are several things that must be done to properly configure the CAN controller. These are done in Can.C by functions that are called by CanDemo.c. Examples are found in the function **CAN_setup** (lines 86 to 109).

1. Enable and set the clock for the CAN controller. **TIP:** The clock must be stable for CAN. No R-C oscillators !
2. Initialize both CAN controllers.
3. Configure GPIO ports P0.0, P0.1, P2.7 and P2.8 for the transmit and receive lines to the transceiver chip.
4. Enable the interrupt for the receive functions. (recall transmit is periodically driven).

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 adjust these settings for the most robust bus performance.

All CAN controllers have the same general settings for bit timing because of the licensing agreements with Robert Bosch GmbH. For a detailed explanation of CAN bit timing see www.port.de/pdf/CAN_Bit_Timing.pdf and for the calculations see the LPC1700 Users Manual.

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.

Other Functions in Can.c:

- CAN_start: Starts the CAN controller by ending the initialization sequence.
- CAN_waitReady: Waits until transmit mailbox is ready – then can add another message to be transmitted.
- CAN_wrMsg: Write a message to the CAN controller and transmit it.
- CAN_rdMsg: Read a message from the CAN controller and release it to be sent to the LPC1700 processor.
- CAN_wrFilter: Configure the acceptance filter. This is not discussed in this article.
- 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.

3) CanDemo.c

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

Transmitting a CAN Message:

Lines 130 to 134 puts values into the structure CAN_TxMsg. (Except for the data byte from the A/D converter.)

```
130  CAN_TxMsg[1].id = 33;                            /* initialise message to send */
131  for (i = 0; i < 8; i++) CAN_TxMsg[0].data[i] = 0;
132  CAN_TxMsg[1].len = 1;
133  CAN_TxMsg[1].format = STANDARD_FORMAT;
134  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 idea to fill the empty data bytes with either 0 or FF.

Lines 140 puts the A/D value into the data member CAN_TxMsg[0].data in data byte 0 and line 141 transmits it by calling the function CAN_wrMsg which resides in module Can.c.

```
140  CAN_TxMsg[1].data[0] = val_Tx;              /* data[0] field = ADC value */
141  CAN_wrMsg (2, &CAN_TxMsg[1]);               /* transmit message */
```

Receiving a CAN Message:

Lines 147 to 148 indicate when a CAN message is received. But something more must be going on here. Line 150 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] ?

```
147   if (CAN_RxRdy[0]) {
148      CAN_RxRdy[0] = 0;
150      val_Rx = CAN_RxMsg[0].data[0];
153      val_display ();                          /* display TX and RX values */
```

In CanDemo.c:

1. If flag CAN_RxRdy[0] = 1, then set it to 0.
2. Get data byte from array CAN_RxMsg[0] and put in val_Rx.
3. Val_Rx is displayed on LCD by function val_display. Goto to 1.

In Can.c:

1. If CAN_IRQHandler sees CAN controller has a received message ready 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. Goto 1.

Interrupt Handler: (Can.c)

In lines Lines 138 to 143 in CanDemo.c (reproduced below) we can see where the CAN frame is transmitted out CAN2. Clearly, this IF statement is executed periodically and CAN_wrMsg is called from here.

```
138   if (CAN_TxRdy[1]) {                              /* tx message on CAN Controller #2 */
139         CAN_TxRdy[1] = 0;
141         CAN_TxMsg[1].data[0] = val_Tx;             /* data[0] field = ADC value */
142         CAN_wrMsg (2, &CAN_TxMsg[1]);              /* transmit message */
143       }
```

Recall we said before that the function to read the CAN data was located in Can.c. If we look in Can.c, we find the function CAN_rdMsg at lines 155 to 171. 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 µVision that is available in Simulator mode to figure out how CAN_rdMsg is called.

1. Set a breakpoint in line 155 at the start of CAN_rdMsg in Can.c. Enter Debug mode.
2. Click on the arrow beside the Trace icon and select Enable Trace Recording. Or open View/Trace.
3. Run the program to the breakpoint. **TIP:** If the program doesn't stop click on ToolBox CAN Loopback ON.
4. Click on the arrow beside the Trace icon and select Show Records in Disassembly or press Ctrl-T. View the Disassembly window that opens up as shown below.

The yellow arrow points to the start of the function CAN_rdMsg. This where the program counter is pointing to.

The grey area shows a recording of the instructions that were executed and those in the white area are unexecuted. The small red block indicates the breakpoint. The green block (behind the breakpoint), the orange block (must scroll down for an example) and the grey blocks are for Code Coverage. Green is executed, orange is partial execution (usually for branches with all options not taken) and gray is not executed.

307: is the start of the handler CAN_IRQHandler which is found in Can.c starting at line 307. **311:** is the statement that calls CAN_rdMsg. You can see the assembly addresses jump from 0x07DE to 0x0450 as CAN_rdMsg is called.

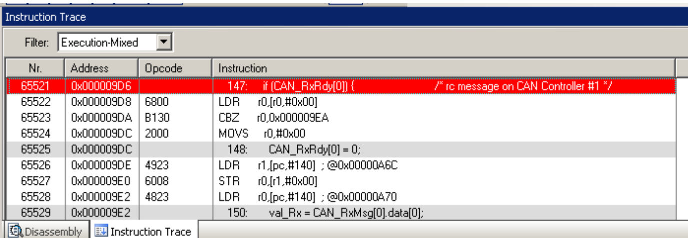
So, this interrupt handler called the function that reads the CAN frame from the CAN controller. You can scroll down as see the rest of the unexecuted statements and instructions in the interrupt handler.

If you click on any line in the disassembly window, it will be highlighted in a yellow bar and a cyan arrow will point to the corresponding source line in the source file. You can use the up and down arrows to move through the code. The register contents will change to reflect their values when the highlighted instruction was executed.

```
Disassembly
0x00000876 D1FC      BNE      0x00000872
0x00000872 1E40      SUBS     r0,r0,#1
0x00000874 2800      CMP      r0,#0x00
0x00000876 D1FC      BNE      0x00000872
   307: void CAN_IRQHandler (void)  {
0x000007CE B510      PUSH     {r4,lr}
   310:   if (CAN1->GSR & (1 << 0)) {                     /* CAN Controller #1 meassage is received */
0x000007D0 4822      LDR      r0,[pc,#136]  ; @0x0000085C
0x000007D2 6880      LDR      r0,[r0,#0x08]
0x000007D4 F0100F01  TST      r0,#0x01
0x000007D8 D009      BEQ      0x000007EE
   311:           CAN_rdMsg (1, &CAN_RxMsg[0]);                /*  read the message */
0x000007DA 4921      LDR      r1,[pc,#132]  ; @0x00000860
0x000007DC 2001      MOVS     r0,#0x01
0x000007DE F7FFFE37  BL.W     CAN_rdMsg (0x00000450)
   155: void CAN_rdMsg (uint32_t ctrl, CAN_msg *msg)  {
0x00000450 B530      PUSH     {r4-r5,lr}
   156:   CAN_TypeDef *pCAN = (ctrl == 1) ? CAN1 : CAN2; /* select CAN controller */
   157:   uint32_t CANData;
   158:
```

Trace Window: Click on the Trace icon to open this window up or the arrow beside it and select Instruction Trace:

On top of the disassembly window will be a new µVision4 trace window. If you double-click on any line you will be taken to the assembly line in the Disassembly window and C source in the source file. You can use the up and down arrow keys to move forwards or backwards in the program flow.


Instruction Trace

Filter: Execution-Mixed

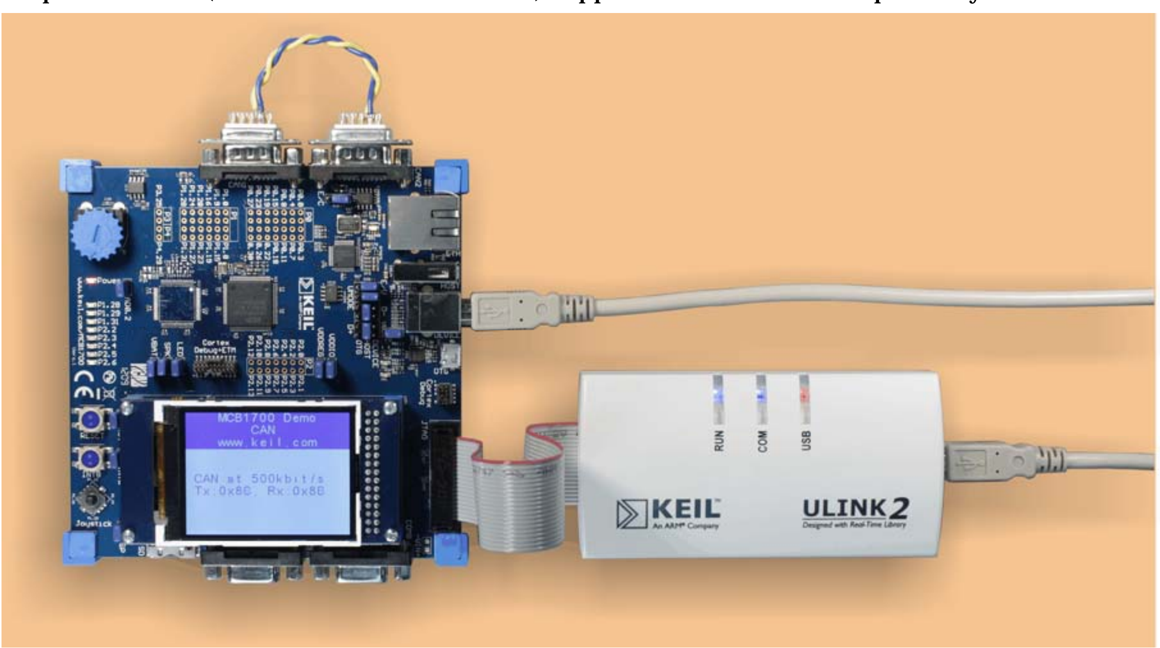
Nr.	Address	Opcode	Instruction
65521	0x000009D6		147: if (CAN_RxRdy[0]) { /* rc message on CAN Controller #1 */
65522	0x000009D8	6800	LDR r0,[r0,#0x00]
65523	0x000009DA	B130	CBZ r0,0x000009EA
65524	0x000009DC	2000	MOVS r0,#0x00
65525	0x000009DC		148: CAN_RxRdy[0] = 0;
65526	0x000009DE	4923	LDR r1,[pc,#140] ; @0x00000A6C
65527	0x000009E0	6008	STR r0,[r1,#0x00]
65528	0x000009E2	4823	LDR r0,[pc,#140] ; @0x00000A70
65529	0x000009E2		150: val_Rx = CAN_RxMsg[0].data[0];

 © 2009 Keil™ All rights reserved

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 discussed in this article. This board contains two separate CAN nodes. We will have one node talk to the other.

This shows the Keil ULINK2 USB adapter with Serial Wire Viewer. You can also use the ULINK-ME, ULINK*Pro*, Segger J-Link, J-Trace and the Signum Systems JTAGJet or JTAGJetTrace. These, and more, are all supported with µVision. All provide JTAG, Serial Wire Debug (SWD) and Serial Wire Viewer (SWV). The ULINK*Pro,* JTAGJetTrace and J-Trace provide ETM (Embedded Trace Macrocell) support. ETM is discussed presently.



Exception, PC and Data Tracing:

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 one to the other.
2. Start µVision by clicking on its icon on your Desktop.
3. Select Project/Open Project. Open the file C:\Keil\ARM\Boards\Keil\MCB1700\CAN\CAN.Uv2 (or uvproj).
4. Make sure "MCB1700" is selected in the Target window. MCB1700
5. Compile the source files by clicking on the Build icon. . They will compile with no errors and 1 warning.
6. Click the "Options for Target" icon. Select Debug tab and set to "Use ULINK Cortex debugger". Click OK.
7. Program the LPC1700 Flash by clicking on the Load icon. LOAD
8. Enter the Debug mode by clicking on the debug icon. Select OK if the Evaluation Mode box appears.
9. Click on the RUN icon. Note: you stop the program with the STOP icon.
10. 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.
11. This is our minimum two nodes in operation. You could connect a CAN analyzer to the DB9 connectors to monitor the CAN messages or 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.

TIP: If you can't see any images on the LCD make sure you uncomment any of the GLCD and val_display (); lines as described in the simulator demonstration on page 10 and rebuild the project.

Exception Tracing:

The NXP LPC1700 series Cortex-M3 processor possesses significant debugging capabilities including Serial Wire Viewer (SWV) and ETM trace. These are supported by Keil µVision using the Keil USB-JTAG adapters ULINK2, ULINK-ME for SWV and the Signum Systems JtagJetTrace and the new Keil ULINK*Pro* for both SWV and ETM trace. Keil also supports the Segger J-Link and J-Trace units. You can use the SWV or ETM or both.

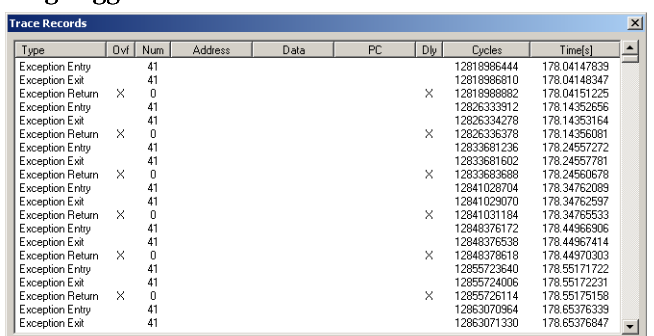
SWV can display PC (program counter) samples, data reads and writes, interrupts and CPU counters all with a timestamp. ETM provides all PCs, interrupts and a time stamp. These are provided in real-time with no CPU cycle stealing. The ITM provides printf type debugging. See www.keil.com for information on the Serial Wire Viewer. Examples on hardware are shown with both ETM and SWV. SWV and ETM do not work in the simulator. SWV and ETM are not needed as the simulator by definition has access to all the processor internal nodes.

Recall that Can.c contains an interrupt handler for receiving messages that calls the function CAN_RdMsg. These can be displayed in real-time (no CPU cycles are stolen) in the Trace Records window as shown below. This works only with a real target Cortex-M3 connected to µVision with a Keil ULINK2, ULINLK-ME, Segger JLink or Signum JtagJet.

To show our example using the Serial Wire Viewer and a LPC1768 plus a ULINK2, ULINK-ME or ULINK*Pro*:

1. You will have the example from the previous page running. Stop it and enter edit mode by clicking .
2. Click on the Options for Target icon. Select Debug tab. Select ULINK Cortex debugger. Click Settings.
3. Make sure the SWJ box is checked and SW is entered in the "Port:" box.
4. Click on Trace tab and enter 72 for Core Clock: and check the Trace Enable and EXCTRC boxes. Uncheck the Periodic box and click on OK twice to return to Edit mode.
5. Enter the Debug mode by clicking on the debug icon. Select OK if the Evaluation Mode box appears.
6. Click on the RUN icon.
7. Open View/Trace and select Record or click on the arrow on the Trace icon and select Records.
8. Displayed is the exception CAN_IRQHandler in Can.c being triggered. This is shown below:

Records
Exceptions
Counters



Trace Records

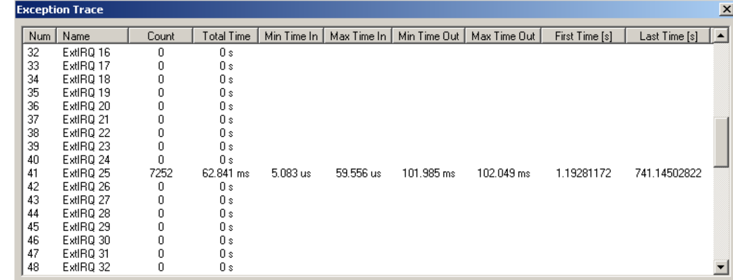
Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Exception Entry		41					12818986444	178.04147839
Exception Exit		41					12818986810	178.04148347
Exception Return	X	0				X	12818988882	178.04151225
Exception Entry		41					12826333912	178.14352656
Exception Exit		41					12826334278	178.14353164
Exception Return	X	0				X	12826336378	178.14356081
Exception Entry		41					12833681236	178.24557272
Exception Exit		41					12833681602	178.24557781
Exception Return	X	0				X	12833683688	178.24560678
Exception Entry		41					12841028704	178.34762089
Exception Exit		41					12841029070	178.34762597
Exception Return	X	0				X	12841031184	178.34765533
Exception Entry		41					12848376172	178.44966906
Exception Exit		41					12848376538	178.44967414
Exception Return	X	0				X	12848378618	178.44970303
Exception Entry		41					12855723640	178.55171722
Exception Exit		41					12855724006	178.55172231
Exception Return	X	0				X	12855726114	178.55175158
Exception Entry		41					12863070964	178.65376339
Exception Exit		41					12863071330	178.65376847

TIP: - Exception Entry is when the exception occurs.
- Exception Exit is when the exception returns.
- Exception Return is when the ALL exceptions return.

TIP: The timestamp will tell you when they occurred.

TIP: Num is the exception number: RESET is 1. External exceptions start at Num 16. For LPC1768, 41 is CAN IRQ. This is found in the LPC17xx Users Manual.
Num 41 is also known as 41-16 = External IRQ 25.

9. Open View/Trace and select Exceptions.
10. The window to the right appears. Note all exceptions (RESET, NMI etc.) are listed.
11. Scroll down and watch 41 being triggered.
12. Note IRQ 41 occurs about every 5 ms or so.
13. 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.



Exception Trace

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						

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 standard Keil ULINK2, ULINK-ME, Signum JtagJetTrace and Segger J-Link adapters.

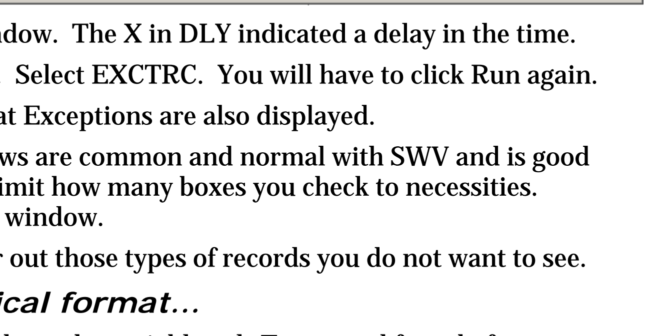
 © 2009 Keil™ All rights reserved

Data Tracing: displaying the data reads and 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 click 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.
   TIP: This action causes the variable val_Tx to be displayed in the Trace Records window.
5. A Logic Analyzer window like the one shown below will open. Click on **Out** or **In** until **Range:** equals 10 s.
6. Open View/Trace and select Trace Records. You can select this from the Trace icon too.
7. Click Run. Rotate the pot P7. Shown here is an example display that results.



Trace Records

Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			1000001CH	00000026H	000009ACH	X	185652170423	2578.50236699
Data Read			1000001CH	00000026H	000009BCH	X	185652177858	2578.50247025
Data Read			1000001CH	00000026H	000008C2H	X	185652211090	2578.50293181
Data Write			1000001CH	00000026H	000009ACH	X	185659476207	2578.60383621
Data Read			1000001CH	00000026H	000009BCH	X	185659483628	2578.60393928
Data Read			1000001CH	00000026H	000008C2H	X	185659516874	2578.60440103
Data Write			1000001CH	00000027H	000009ACH	X	185666781991	2578.70530543
Data Read			1000001CH	00000027H	000009BCH	X	185666789398	2578.70540831
Data Read			1000001CH	00000027H	000008C2H	X	185666822658	2578.70587025
Data Write			1000001CH	00000025H	000009ACH	X	185674087955	2578.80677715
Data Read			1000001CH	00000025H	000009BCH	X	185674095354	2578.80687992
Data Read			1000001CH	00000025H	000008C2H	X	185674128622	2578.80734197
Data Write			1000001CH	00000024H	000009ACH	X	185681393803	2578.90824726
Data Read			1000001CH	00000024H	000009BCH	X	185681401186	2578.90834981
Data Read			1000001CH	00000024H	000008C2H	X	185681434470	2578.90881208
Data Write			1000001CH	00000026H	000009ACH	X	185688699671	2579.00971765
Data Read			1000001CH	00000026H	000009BCH	X	185688707080	2579.00982056
Data Read			1000001CH	00000026H	000008C2H	X	185688740338	2579.01028247
Data Write			1000001CH	00000025H	000009ACH	X	185696005455	2579.11118687
Data Read			1000001CH	00000025H	000009BCH	X	185696012850	2579.11128958

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 indicated a delay in the time.

9. Open Debug/Debug Settings and click on the Trace tab. Select EXCTRC. You will have to click Run again.
10. Double-click on 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 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.

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

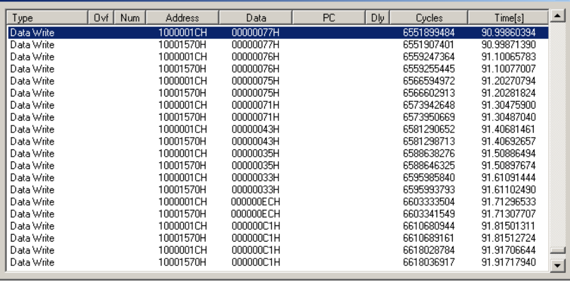
Data Tracing: *displaying variables in a graphical format…*

1. Open the Logic Analyzer Window – this should already have the variable val_Tx entered from before.
2. Optionally enter CAN_RxMsg[0].data[0]. Set Max: to 0xFF and Min to 0x0. Click on Close.
3. With the program running, vary the potentiometer. The screen will display the value of val_Tx in real-time as shown here in the Logic Analyzer.

TIP: If the graph doesn't look right – make sure **Range:** is set to 10 seconds and the scroll bar at the bottom is positioned to the far right. Check the Min and Max settings.

Example: 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 this for demonstrations. 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.



Trace Records

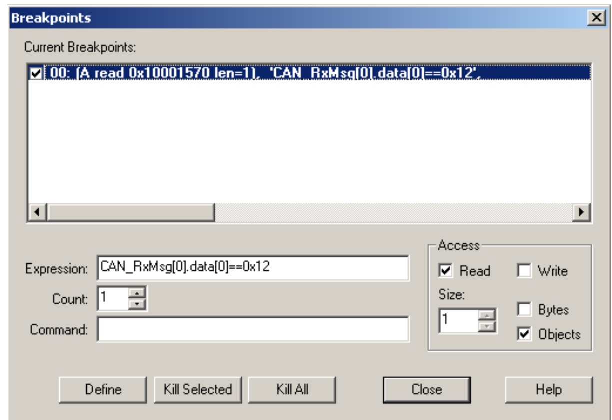
Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
Data Write			1000001CH	00000077H			6551899484	90.99860394
Data Write			10001570H	00000077H			6551907401	90.99871390
Data Write			1000001CH	00000076H			6559247364	91.10065783
Data Write			10001570H	00000076H			6559255445	91.10077007
Data Write			1000001CH	00000075H			6566594972	91.20270794
Data Write			10001570H	00000075H			6566602913	91.20281824
Data Write			1000001CH	00000071H			6573942648	91.30475900
Data Write			10001570H	00000071H			6573950669	91.30487040
Data Write			1000001CH	00000043H			6581290652	91.40681461
Data Write			10001570H	00000043H			6581298713	91.40692657
Data Write			1000001CH	00000035H			6588638276	91.50886494
Data Write			10001570H	00000035H			6588646325	91.50897674
Data Write			1000001CH	00000033H			6595985840	91.61091444
Data Write			10001570H	00000033H			6595993793	91.61102490
Data Write			1000001CH	000000ECH			6603333504	91.71296533
Data Write			10001570H	000000ECH			6603341549	91.71307707
Data Write			1000001CH	000000C1H			6610680944	91.81501311
Data Write			10001570H	000000C1H			6610689161	91.81512724
Data Write			1000001CH	000000C1H			6618028784	91.91706644
Data Write			10001570H	000000C1H			6618036917	91.91717940

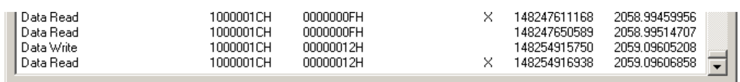
This is one example of the usefulness of Serial Wire Viewer. It could be otherwise very difficult to find this with without Serial Wire Viewer. Another good use is finding interrupts that fire too often using up CPU time and slowing your program down.

Watchpoints:

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

1. Enter Debug mode.
2. Open Debug/Breakpoints and an empty window below opens up.
3. Enter in the field Expression: CAN_RxMsg[0].data[0]==0x12. 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 and run the program.
6. Adjust the pot and when Rx equals 0x12 the program will stop. This is a Watchpoint and this is how it works.
7. The last entry in the Trace Records will be this read or write of 0x12 as shown here:





Data Read	1000001CH	0000000FH	X	148247611168	2058.99459956
Data Read	1000001CH	0000000FH		148247650589	2058.99514707
Data Write	1000001CH	00000012H		148254915750	2059.09605208
Data Read	1000001CH	00000012H	X	148254916938	2059.09606858

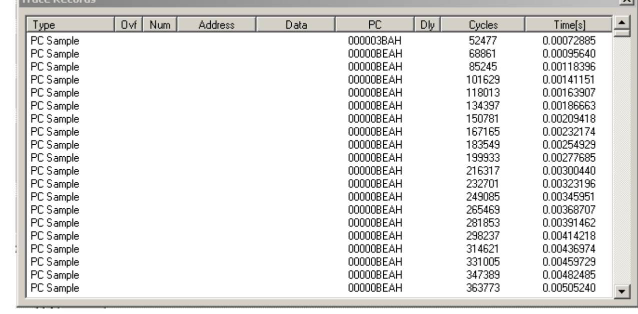
8. Remove the Watchpoint using the Kill All button.

TIP: When the Breakpoints window is open you can modify a Watchpoint by double-clicking on it and it will move below. 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.

PC Tracing: *Program Counter samples…*

1. OpenDebug/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.



Trace Records

Type	Ovf	Num	Address	Data	PC	Dly	Cycles	Time[s]
PC Sample					000003BAH		52477	0.00072885
PC Sample					00000BEAH		68861	0.00095640
PC Sample					00000BEAH		85245	0.00118396
PC Sample					00000BEAH		101629	0.00141151
PC Sample					00000BEAH		118013	0.00163907
PC Sample					00000BEAH		134397	0.00186663
PC Sample					00000BEAH		150781	0.00209418
PC Sample					00000BEAH		167165	0.00232174
PC Sample					00000BEAH		183549	0.00254929
PC Sample					00000BEAH		199933	0.00277685
PC Sample					00000BEAH		216317	0.00300440
PC Sample					00000BEAH		232701	0.00323196
PC Sample					00000BEAH		249085	0.00345951
PC Sample					00000BEAH		265469	0.00368707
PC Sample					00000BEAH		281853	0.00391462
PC Sample					00000BEAH		298237	0.00414218
PC Sample					00000BEAH		314621	0.00436974
PC Sample					00000BEAH		331005	0.00459729
PC Sample					00000BEAH		347389	0.00482485
PC Sample					00000BEAH		363773	0.00505240

TIP: In this example every 16,384th PC is displayed. This can be changed and the ULINK*Pro* can display every 64th PC. PC Samples are very useful as they can give good indication where your CPU is spending its time. Use ETM trace if you need to see every PC for advanced debugging or Code Coverage.

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 µVision Watch and memory windows without stealing CPU cycles to do so.

TIP: In addition, you are able, in real-time, to insert values into these windows. 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, highlight ***type F2 to edit***, press F2 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 and press Enter. You will see the appropriate memory locations change value as the pot is turned. **TIP:** With no "&" prefix, CAN_TxMsg it will point to different addresses.

TIP: Note you can enter these variables when the program is still running. You can also highlight, drag and drop.

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

This finishes a partial demonstration of the Serial Wire Viewer trace feature of the Cortex-M3 processor. Visit www.keil.com for more information concerning the Serial Wire Viewer interface in Cortex-M3 processors.

 © 2009 Keil™ All rights reserved

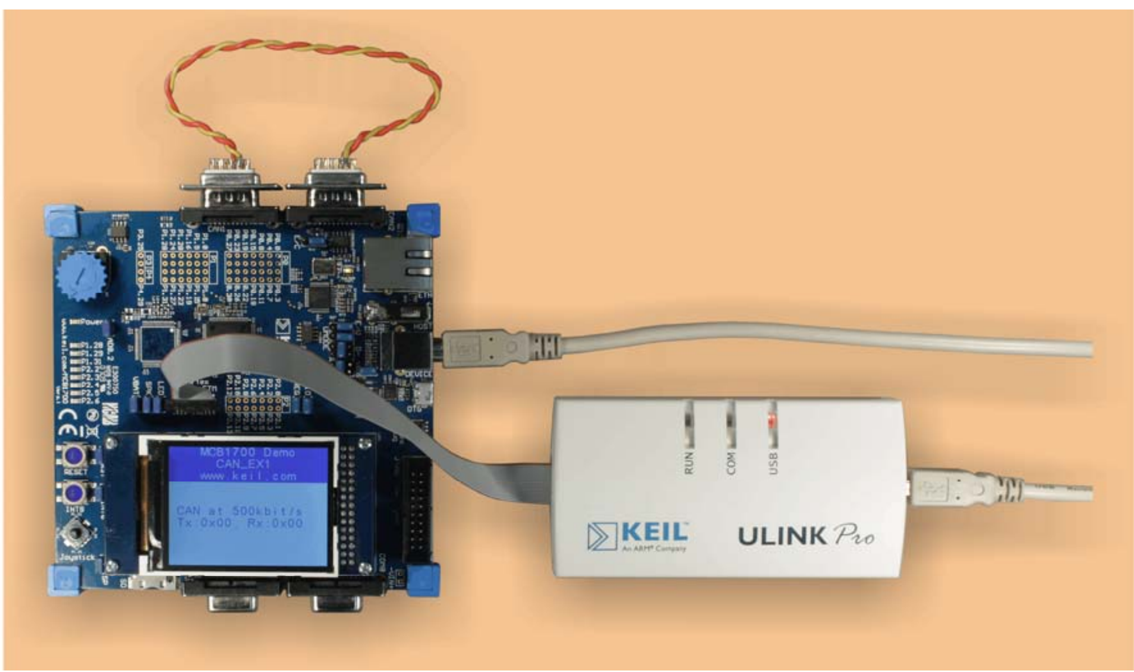
ETM Tracing: Embedded Trace Macrocell

The NXP LPC1700 family has 4 bit ETM trace. Recall the Serial Wire Viewer has PC Samples. This means not all PC values are captured. However, ETM provides all the program counter values, a timestamp and interrupts. 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 uVision using either a Keil ULINK*Pro* (pictured below), a Signum Systems JtagJetTrace or a Segger J-Trace. The ULINK*Pro* is available in September 2009. We will demonstrate ETM using the Signum adapter. A later version of this document will demonstrate the ULINK*Pro* as well. See www.keil.com.

The ETM is connected via a new ARM specified 20 pin connector. It is labeled Cortex Debug + ETM on the MCB1700. The ULINK*Pro* is shown connected to it. A close-up of this connector is shown below. The JtagJetTrace is connected in the same fashion.

Rather than provide detailed instructions of using ETM we will discuss some of the information you can obtain with this powerful debugging tool. For instructions on using ETM with ULINK*Pro*, JtagJetTrace or the Segger J-Link, please contact the author, Keil technical support or Signum Systems www.signum.com or Segger www.segger.com.

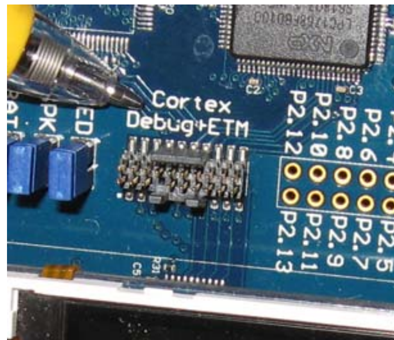


Debugging connections to the MCB1700:

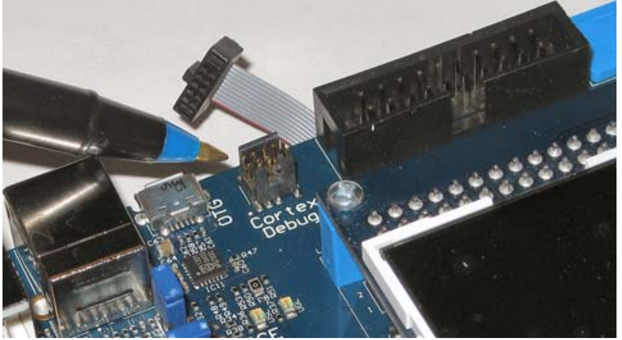
In addition to the standard 20 pin JTAG connector; the MCB1700 has two additional connectors. One is the 20 pin Hi-Density ETM connector as already described. The other is the new 10 pin Hi-Density connector pictured below:

Connector Summary: Search www.arm.com for DDI0314F_coresight_component_trm.pdf for complete details.

1. Legacy JTAG: JTAG, SWD and SWO. No ETM.
2. ARM 20 pin Hi-Density: JTAG, SWD, SWO and 4 bit ETM.
3. ARM 10 pin Hi-Density: JTAG, SWD and SWO. No ETM.



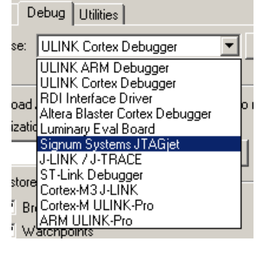
20 Pin Hi-Density connector



10 Pin Hi-Density beside the legacy 20 pin JTAG connector

 © 2009 Keil™ All rights reserved

Configuring the JtagJetTrace:

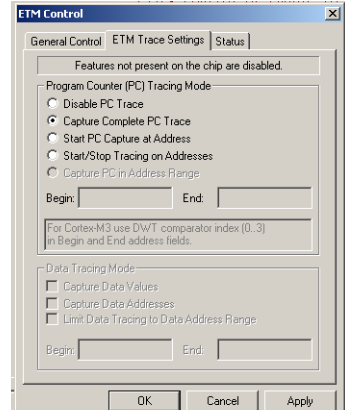


The JtagJetTrace software is completely integrated into µVision.

TIP: Note the types of adapters µVision supports. ULINK*Pro* support is listed as well as the Segger J-Link. Note the Keil ULINK has both ARM and Cortex modes.

General Features:

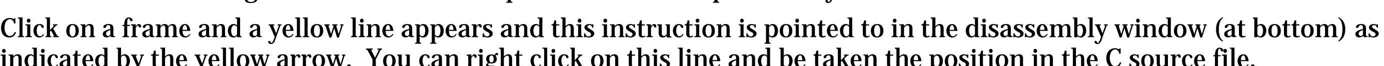
1. Can stop the trace capture on a processor stop or a trigger or when the buffer is full.
2. Can accumulate trace collections between runs or clear the buffer every time.
3. Can start or stop trace collection, configure triggers and filters and view the trace *without stopping the CPU.*
4. Start or stop trace collection with ranges of specified addresses, data and many other events.
5. Can pre- or post-filter information to see only what you want to.
6. Automatically tracks the CPU clock. Can have over 1 million frames saved.



Information Displayed in the Trace window as shown below:

1. Program counter addresses. (all of them !)
2. Exceptions or Interrupts.
3. Disassembled instructions.
4. Source code: will point back to your source files for reference.
5. Timestamp: relative, delta and absolute in cycles or time.
6. Data read/write data values and source/destination address.

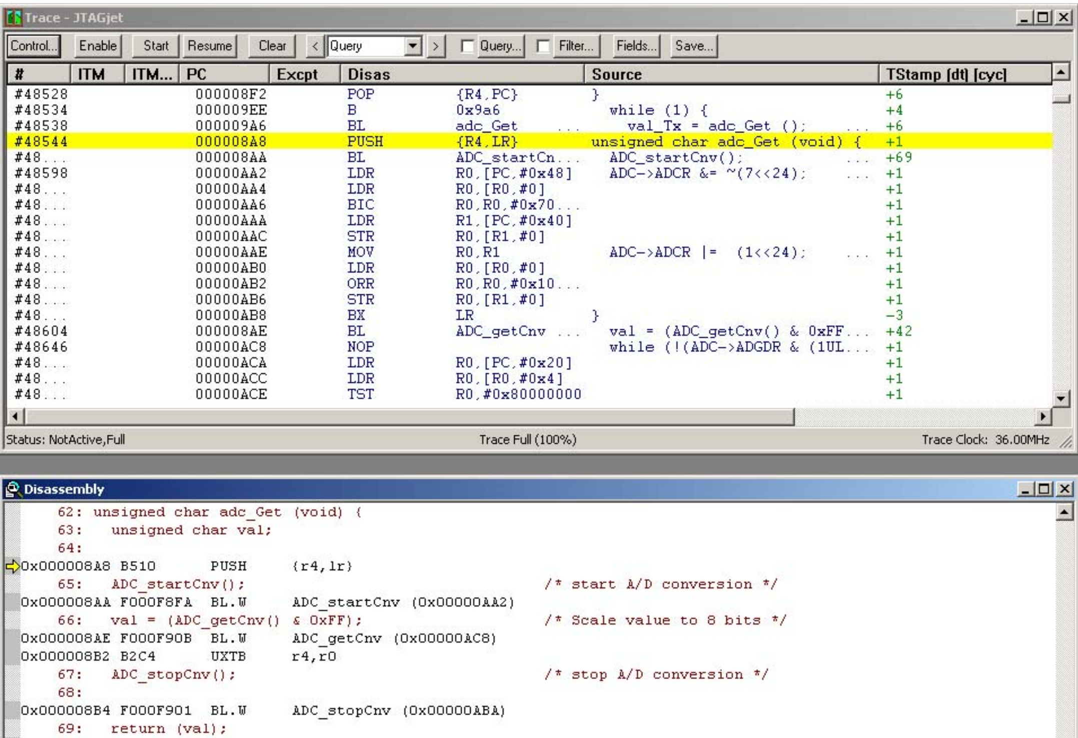
A Trace Window Example: Below is a trace window capture of our CAN demonstration program. There are about 1 million assembly instructions stored in the trace buffer. The JtagJetTrace has buffer options available up to 8 Mbytes.



Click on a frame and a yellow line appears and this instruction is pointed to in the disassembly window (at bottom) as indicated by the yellow arrow. You can right click on this line and be taken the position in the C source file.

The timestamp is in delta and cycles. Most instructions take 1 CPU cycle. At the bottom right is the trace clock frequency of 36 MHz which is one half of the CPU clock of 72 MHz. The ETM is using half-clock mode which means the trace information is collected on both the rising and falling edge of the trace clock.

The timestamp is very useful. It can tell you how long you spent in a function, the time between function calls, time between data reads and writes and how often interrupts are triggered.



 © 2009 Keil™ All rights reserved

Triggers and Filters:

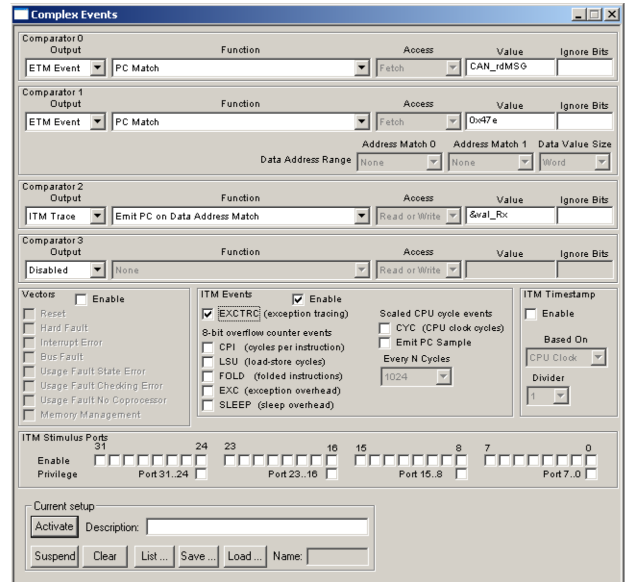
This is the Complex Events window which provides the triggers and filters for both ETM and SWV in the JtagJetTrace as implemented in µVision. You should recognize parts of it from the Serial Wire Viewer configuration when used with a ULINK. This handles both ETM, SWV and Watchpoints.

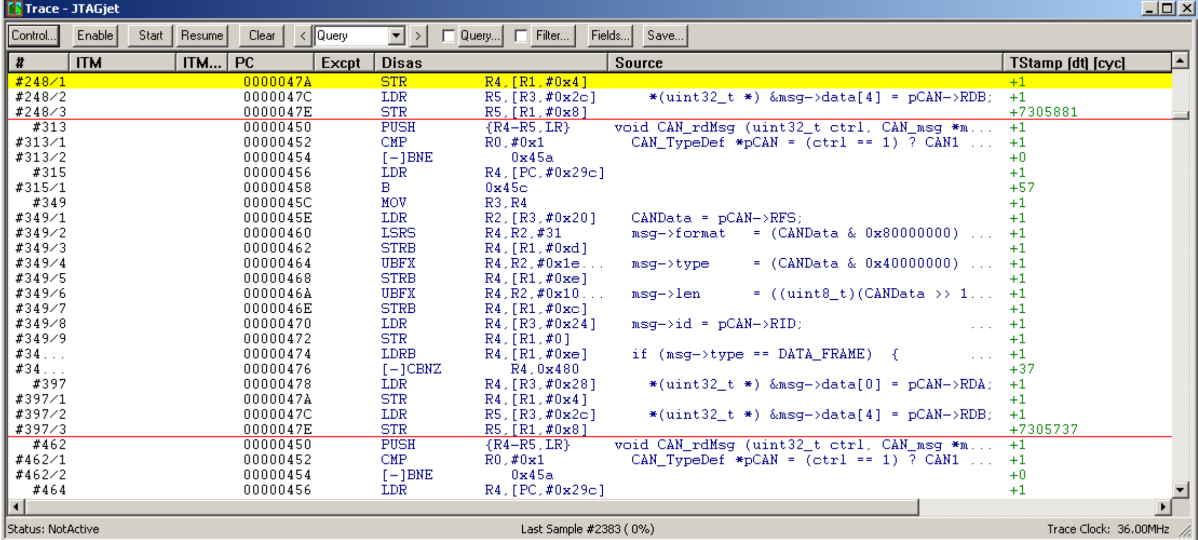
In this example is a trigger that starts and stops the trace collection when the program is in the function CAN_rdMsg. Comparator 0 starts the trace collection and Comparator 1 stops it at 0x47E. You can use either labels or numbers.

These settings can be changed while the program is running.

Below is the trace window capture CAN_rdMsg program. There are about 1 million assembly instructions stored in the trace buffer.

The red line is where the trace was stopped and started. The time stamps show 7,305,735 CPU cycles between each run. When I can change it to time I get 101.48 msec. Inside the function you can see how long each instruction took to execute.



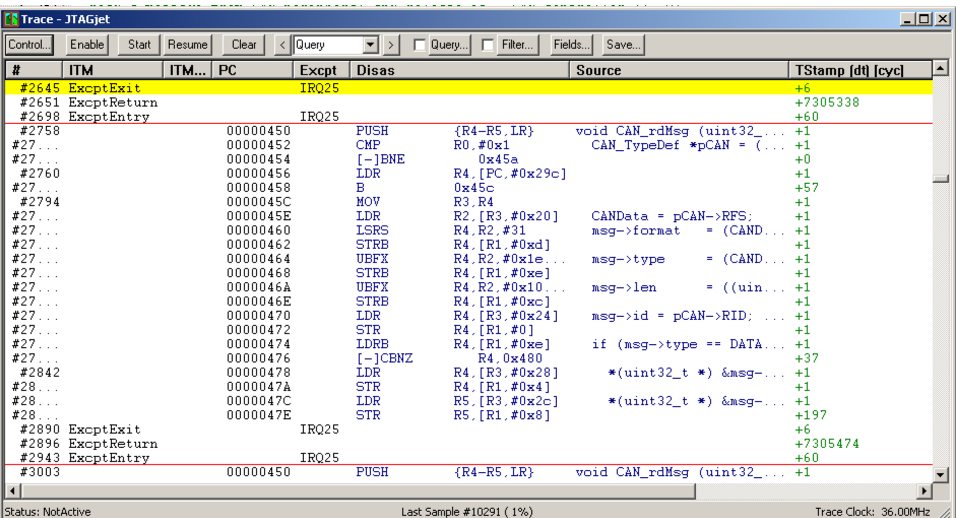


Displaying Exceptions:

In the Complex Events above I then selected EXCTRC under ITM Events. This is the same EXCTRC setting as used in µVision and ULINK. The window below shows the addition of IRQ 25 and its associated time stamps. This is very valuable for finding excessive exception calls.

TIP: You can stop the trace when the PC lands on an error vector such as a Bus Fault and the trace will be filled up with executed instructions before this unfortunate event. Very close to the end of the trace buffer will be the offending instruction that cause the fault. This is excellent for when your program goes off "into the weeds".

TIP: You can filter out the PCs and see only the exceptions. Very handy for timing issues.

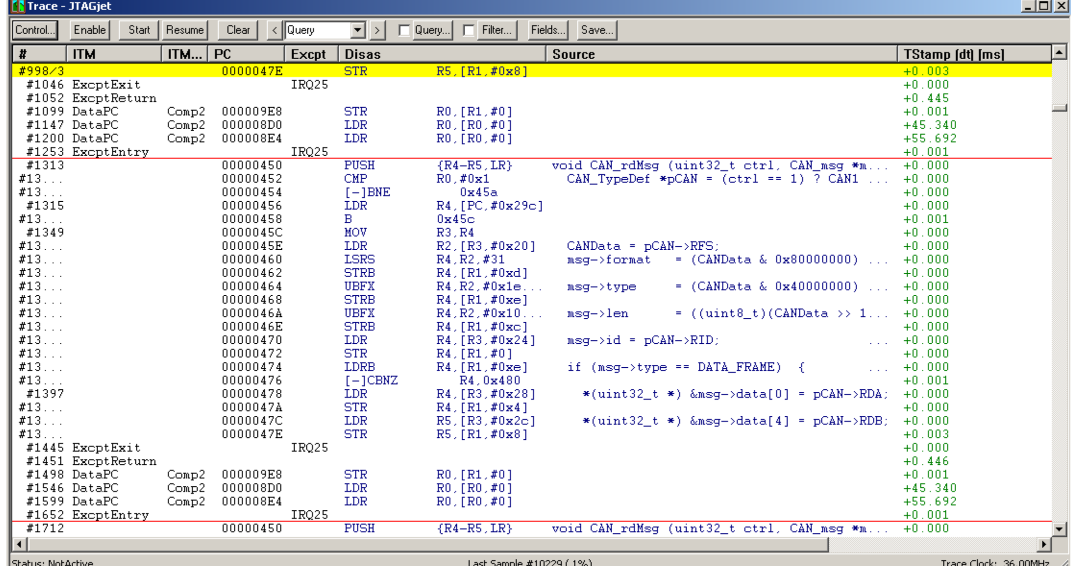


 © 2009 Keil™ All rights reserved

Displaying Data Read and Write Instructions:

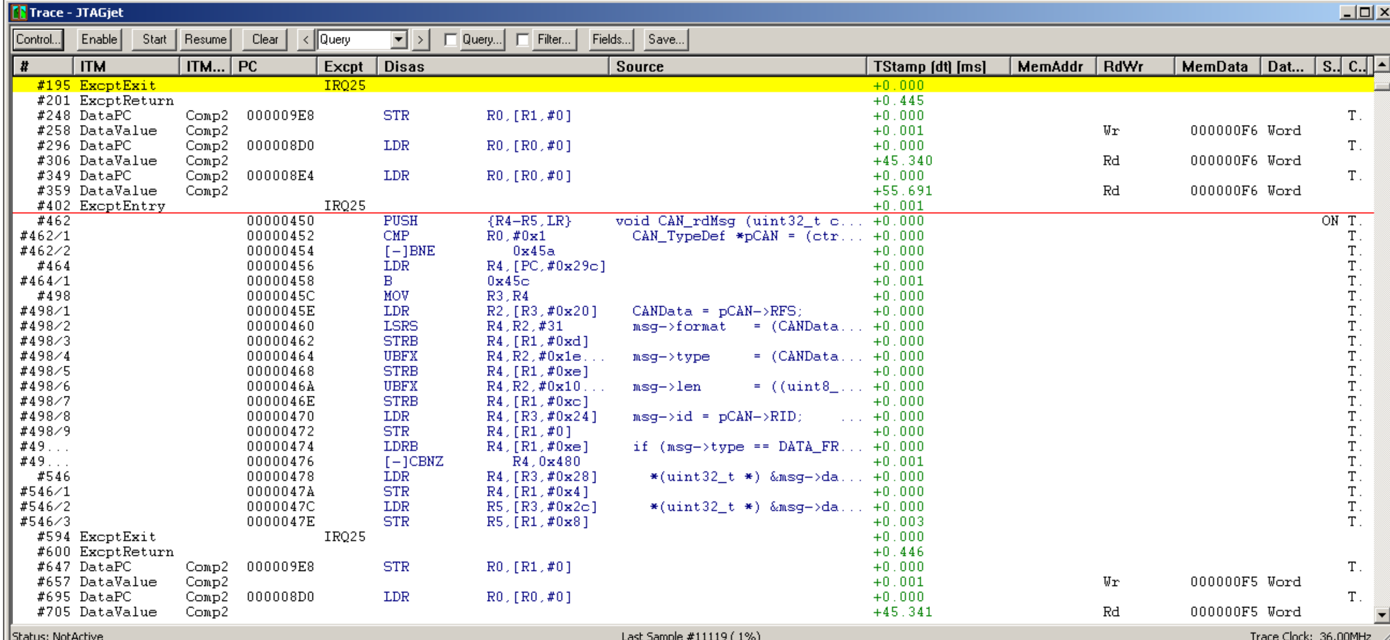
Recall that we showed how to show data reads and writes using the Keil ULINK with Serial Wire Viewer. We can do the same here. In the Complex Events window above I entered in Comparator 2 the request to display all reads or writes to the variable tx_Msg. I selected "Emit PC on Data Address Match". This is displayed in the window below:

You can see two reads and one write executed by the instructions at the addressed requested (9E8, 8D0 and 8E4) with associated timestamps. This is displayed in lines 1099, 1147 and 1200 just above the red line after IRQ25.



TIP: If you double-click in a column – you can select to display data only where there is something in that column. For example, if you selected Excpt, all frames would be filtered out except for those containing exception information. A handy way to quickly see how often exceptions are executed.

Here the "Emit PC on Data Address Match" was changed to "Emit PC and Data on Data Address Match" and the following screen results. Now we have not only the address of the instructions but whether it is R or W, the size of the transfer and the data values transferred. In this case, 000000F6 was displayed on the MCB1700 LCD. The "T"on the right side indicates a Thumb instruction. Cortex-M3 processors have only Thumb2 which are all 16 bit Thumb instructions and a subset of 32 bit ARM instructions.



TIP: I did all this without stopping the CPU for even one cycle ! Now, that's real-time.

TIP: This data can be saved to a file in various formats for later manipulation.

TIP: There is more to trace…but this gives you a really good idea of its abilities.

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 t 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.
- Corrupted stack. *How did I get here ?* Stack overflows. What causes the stack to grow bigger than it should ?
- Out of bounds data. Uninitialized variables and arrays. Did someone else change this data and who was it ?
- Runaway programs: your program has gone off into the weeds and you need to know what instruction caused this. It is very tough to find these problems without a trace. Trace does this easily.
- Communication protocol and timing issues. System timing problems.
- 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 ?

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-M3 Serial Wire Viewer to see the CAN messages and interrupts displayed in real time. You can compile these examples with the evaluation version of the software.

Keil makes NXP evaluation boards for the LPC2300 and LPC1700 families with CAN examples. .

Keil also provides complete support for the ETM trace on NXP ARM7 and ARM9 as well as the Cortex-M3.

Keil completely supports the new NXP Cortex-M0 devices as well as many NXP 8051 devices with the same µVision. 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.

RL-ARM™: Keil offers a complete CAN stack for all ARM7™, ARM9™ and Cortex-M3/M1/M0 processors. This comes as part of RL-ARM™. Please visit www.keil.com/rl-arm/ for more information. This comprehensive package contains the RTX™ RTOS source (the actual RTOS is already included free with the MDK toolset), USB, TCP/IP networking and the CAN interface.

For more information

Please contact Keil Sales or Technical Support. In USA: sales.us@keil.com or support.us@keil.com or 800-348-8051. Outside of the US: sales.intl@keil.com or support.intl@keil.com.

Comments and inquiries are welcome. Please email bob.boys@arm.com