Debugging Low-speed Serial Buses in Embedded System Designs

Introduction

Embedded systems are literally everywhere in our society today. A simple definition of an embedded system is a special-purpose computer system that is part of a larger system or machine with the intended purpose of providing monitoring and control services to that system or machine. The typical embedded system starts running some special purpose application as soon as it is turned on and will not stop until it is turned off. Virtually every electronic device designed and produced today is an embedded system. A short list of embedded system examples include:

– Alarm clocks
– Automatic teller machines
– Cellular phones
– Computer printers
– Antilock brake controllers
– Microwave ovens
– Inertial guidance systems for missiles
– DVD players
– Personal digital assistants (PDA's)
– Programmable logic controllers (PLC) for industrial automation and monitoring
– Portable music players
– Maybe even your toaster…

Embedded systems can contain many different types of devices including microprocessors, microcontrollers, DSPs, RAM, EPROMs, FPGAs, A/Ds, D/As, and I/O. These various devices have traditionally communicated with each other and the outside world using wide parallel buses. Today, however, more and more of the building blocks used in embedded system design are replacing these wide parallel buses with serial buses for the following reasons:

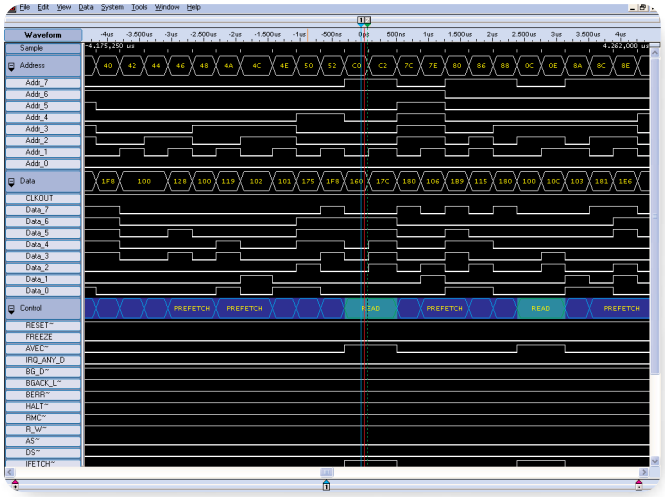
– Less board space required due to fewer signals to route
– Lower cost
– Lower power requirements
– Fewer pins on packages
– Embedded clocks
– Differential signaling for better noise immunity
– Wide availability of components using standard serial interfaces

While serial buses provide a number of advantages, they also pose some significant challenges to an embedded system designer due simply to the fact that information is being transmitted in a serial fashion rather than parallel. This application note discusses common challenges for embedded system designers and how to overcome them using capabilities found in the new DPO4000 Series oscilloscope.

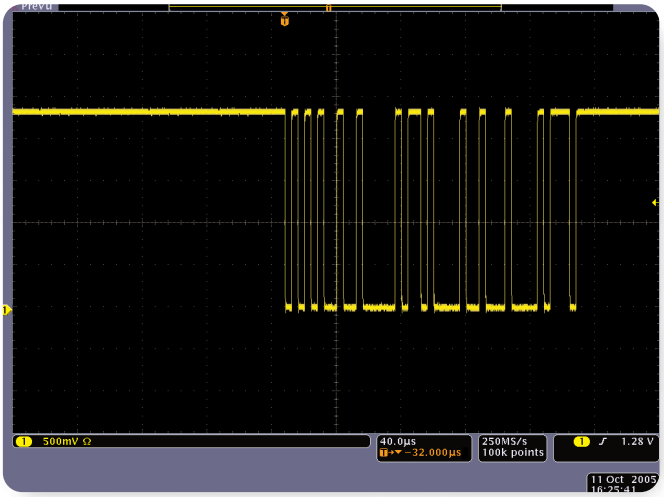
Parallel vs. Serial

With a parallel architecture each component of the bus has its own signal path. There may be 16 address lines, 16 data lines, a clock line and various other control signals. Address or data values sent over the bus are transferred at the same time over all the parallel lines. This makes it relatively easy to trigger on the event of interest using either the State or Pattern triggering found in most oscilloscopes and logic analyzers. It also makes it easy to understand at a glance the data you capture on either the oscilloscope or logic analyzer display. For example, in Figure 1 we've used a logic analyzer to acquire the clock, address, data and control lines from a microcontroller. By using a state trigger, we've isolated the bus transfer we're looking for. To "decode" what's happening on the bus, all we have to do is look at the logical state of each of the address, data, and control lines.

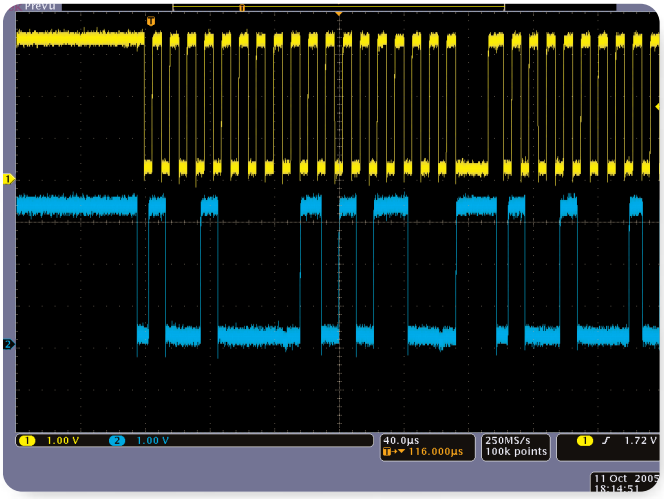
With a serial bus all this information must be sent serially on the same few conductors (sometimes one). This means that a single signal may include address, control, data, and clock information. As an example, look at the Controller Area Network (CAN) serial signal shown in Figure 2.



▶ **Figure 1.** *Logic Analyzer acquisition of a microcontroller's clock, address bus, data bus and control lines.*



▶ **Figure 2.** *One message acquired from a CANbus.*



▶ **Figure 3.** *One message acquired from an I2C bus.*

This message contains a start of frame, an identifier (address), a data length code, data, CRC, and end of frame as well as a few other control bits. To further complicate matters, the clock is embedded in the data and bit stuffing is used to ensure an adequate number of edges for the receiving device to lock to the clock. Even to the very trained eye, it would be extremely difficult to quickly interpret the content of this message. Now imagine this is a faulty message that only occurs once a day and you need to trigger on it. Traditional oscilloscopes and logic analyzers are simply not well equipped to deal with this type of signal.

Even with a simpler serial standard such as I²C it is still significantly harder to observe what is being transmitted over the bus than it is with a parallel protocol.

I²C uses separate clock and data lines, so at least in this case you can use the clock as a reference point. However, you still need to find the start of the message (data going low while the clock is high), manually inspect and write down the data value on every rising edge of the clock, and then organize the bits into the message structure. It can easily take a couple of minutes of work just to decode a single message in a long acquisition and you have no idea if that's the message you are actually looking for. If it's not, then you need to

Start	Address	R/W	Ack	Data0	Ack0	Data1	Ack1	.....	DataN	AckN	Stop
	7 or 10 bits	1-bit	1-bit	8-bits	1-bit	8-bits	1-bit		8-bits	1-bit	

▶ **Figure 4.** *I²C message structure.*

start this tedious and error-prone process over on the next one. It would be nice to just trigger on the message content you are looking for, however the state and pattern triggers you've used for years on scopes and logic analyzers won't do you any good here. They are designed to look at a pattern occurring at the same time across multiple channels. To work on a serial bus, their trigger engines would need to be tens to hundreds of states deep (one state per bit). Even if this trigger capability existed, it would not be a fun task programming it state-by-state for all these bits. There has to be a better way!

With the DPO4000 Series there is a better way. The following sections highlight how the DPO4000 Series can be used with some of the most common low-speed serial standards used in embedded system design.

I²C

Background

I²C, or "I squared C" stands for Inter Integrated Circuit. It was originally developed by Philips in the early 1980's to provide a low-cost way of connecting controllers to peripheral chips in TV sets, but has since evolved into a worldwide standard for communications between devices in embedded systems. The simple two-wire design has found its way into a wide variety of chips like I/O, A/D's, D/A's, temperature sensors, microcontrollers and microprocessors from numerous leading chipmakers including: Analog Devices, Atmel, Infineon, Cyprus, Intel, Maxim, Philips, Silicon Laboratories, ST Microelectronics, Texas Instruments, Xicor, and others.

How It Works

I²C's physical two-wire interface is comprised of bi-directional serial clock (SCL) and data (SDA) lines. I²C supports multiple masters and slaves on the bus, but only one master may be active at any one time. Any I²C device can be attached to the bus allowing any master device to exchange information with a slave device. Each device is recognized by a unique address and can operate as either a transmitter or receiver, depending on the function of the device. Initially, I²C only used 7 bit addresses, but evolved over time to allow 10 bit addressing as well. Three bit rates are supported; 100 kbps (standard mode), 400 kbps (fast mode), and 3.4 Mbps (high speed mode). The maximum number of devices is determined by a maximum capacitance of 400 pf or roughly 20-30 devices. The I²C standard specifies the following format in Figure 4:

– Start - indicates the device is taking control of the bus and that a message will follow

– Address – a 7 or 10 bit number representing the address of the device that will either be read from or written to.

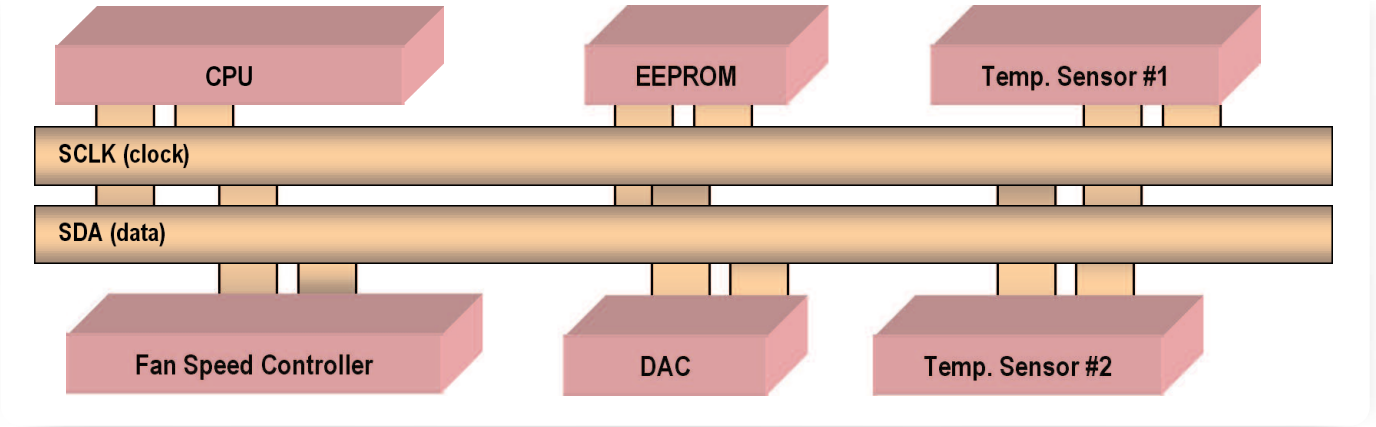
– R/W Bit – one bit indicating if the data will be read from or written to the device

– Ack – one bit from the slave device acknowledging the master's actions. Usually each address and data byte has an acknowledge, but not always.

– Data - an integer number of bytes read or written to the device.

– Stop - indicates the message is complete and the master has released the bus.

▶ **Figure 5.** *I²C bus setup menu.*



▶ **Figure 6.** *I²C bus example.*

Working with I²C

With the DPO4EMBD serial triggering and analysis application module, the DPO4000 Series becomes a powerful tool for embedded system designers working with I²C buses. The front panel has two Bus buttons (B1 and B2) that allow the user to define inputs to the scope as a bus. The I²C bus setup menu is shown in Figure 5.

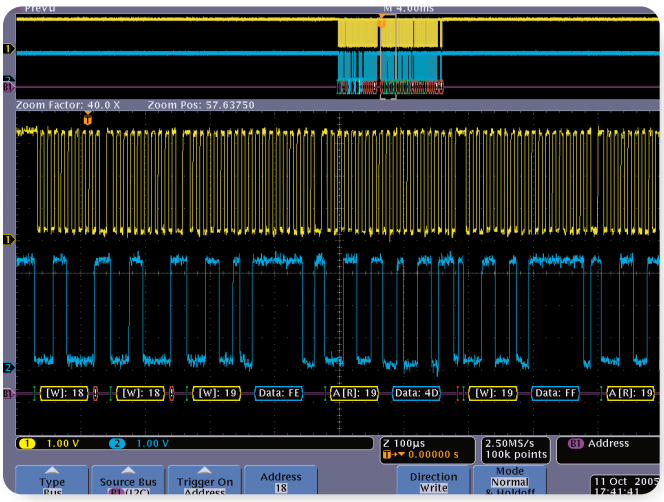
By simply defining which channels clock and data are on, along with the thresholds used to determine logic 1's and 0's, you've enabled the oscilloscope to understand the protocol being transmitted across the bus.

With this knowledge, the oscilloscope can trigger on any specified message level information and then decode the resulting acquisition into meaningful, easily interpreted results. Gone are the days of edge triggering, hoping you acquired the event of interest and then manually decoding message after message looking for the problem.

As an example, consider the embedded system in Figure 6. An I²C bus is connected to multiple devices including a CPU, an EEPROM, a fan speed controller, a digital to analog converter, and a couple of temperature sensors.

This instrument was returned to engineering for failure analysis as the product was consistently getting too hot and shutting itself off. The first thing to check is the fan controller and the fans themselves, but they both appear to be working correctly. The next thing to check for is a faulty temperature sensor. The fan speed controller polls the two temperature sensors (located in different areas of the instrument) periodically and adjusts the fan speed to regulate internal temperature. You're suspicious that one or both of these temperature sensors is not reading correctly. To see the interaction between the sensors and the fan speed controller, we simply need to connect to the I2C clock and data lines and set up a bus on the DPO4000. We know that the two sensors are addresses 18 and 19 on the I2C bus, so we decide to set up a trigger event to look for a write to address 18 (the fan speed controller polling the sensor for the current temperature). The triggered acquisition is shown in the screenshot Figure 7.

In this case, channel 1 (yellow) is connected to SCLK and channel 2 (cyan) to SDA. The purple waveform is the I2C bus we've defined by inputting just a few simple parameters to the oscilloscope. The upper portion of the display shows the entire acquisition. In this case we've captured a lot of bus idle time with a burst of activity in the middle which we've zoomed in on. The lower, larger portion of the display is the zoom window. As you can see, the oscilloscope has decoded the content of each message going across the bus. Buses on the DPO4000 Series use the colors/marks in Table 1 to indicate important parts of the message.



► **Figure 7.** *I2C address and data bus waveform decoding.*

Taking a look at the acquired waveforms, we can see that the oscilloscope did indeed trigger on a Write to address 18 (shown in the lower left of the display). In fact, the fan speed controller attempted to write to address 18 twice, but in both cases it did not receive an acknowledge after attempting to write to the temperature sensor. It then checked the temperature sensor at Address 19 and received back the desired information. So, why isn't the first temperature sensor responding to the fan controller? Taking a look at the actual part on the board we find that one of the address lines isn't soldered correctly. The temperature sensor was not able to communicate on the bus and the unit was overheating as a result. We've managed to isolate this potentially elusive problem in a matter of a couple minutes due to the I2C trigger and bus decoding capability of the DPO4000 Series.

Bus Condition	Indicated by:
Starts are indicated by vertical green bars. Repeated starts occur when another start is shown without a previous Stop.	
Addresses are shown in yellow boxes along with a [W] for write or [R] for read. Address values can be displayed in either hex or binary.	
Data is shown in cyan boxes. Data values can be displayed in either hex or binary.	
Missing Acks are indicated by an exclamation point inside a red box	
Stops are indicated by red vertical bars.	

▶ **Table 1.** *Bus conditions.*

In the example in Figure 7 we triggered on a write, but the DPO4000's powerful I²C triggering includes many other capabilities.

– Start – triggers when SDA goes low while SCL is high

– Repeated Start – triggers when a start condition occurs without a previous stop condition. This is usually when a master sends multiple messages without releasing the bus.

– Stop – triggers when SDA goes high while SCL is high

– Missing Ack – slaves are often configured to transmit an acknowledge after each byte of address and data. The oscilloscope can trigger on cases where the slave does not generate the acknowledge bit.

– Address – triggers on a user specified address or any of the pre-programmed special addresses including General Call, Start Byte, HS-mode, EEPROM, or CBUS. Addressing can be either 7 or 10 bits and is entered in binary or hex.

– Data – triggers on up to 12 bytes of user specified data values entered in either binary or hex

– Address and Data – this allows you to enter both address and data values as well as read vs. write to capture the exact event of interest.

These triggers allow you to isolate the particular bus traffic you're interested in, while the decoding capability enables you to instantly see the content of every message transmitted over the bus in your acquisition.

SPI

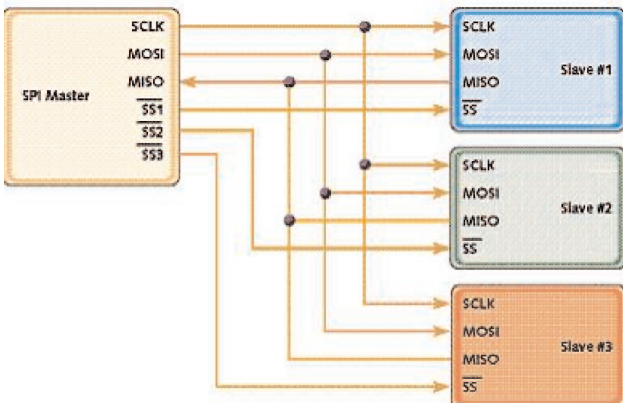
Background

The Serial Peripheral Interface bus (SPI) was originally developed by Motorola in the late 1980's for their 68000 series micro-controllers. Due to the simplicity and popularity of the bus, many other manufacturers have adopted the standard over the years. It is now found in a broad array of components commonly used in embedded system design. SPI is primarily used between micro-controllers and their immediate peripheral devices. It's commonly found in cell phones, PDA's, and other mobile devices to communicate data between the CPU, keyboard, display, and memory chips.

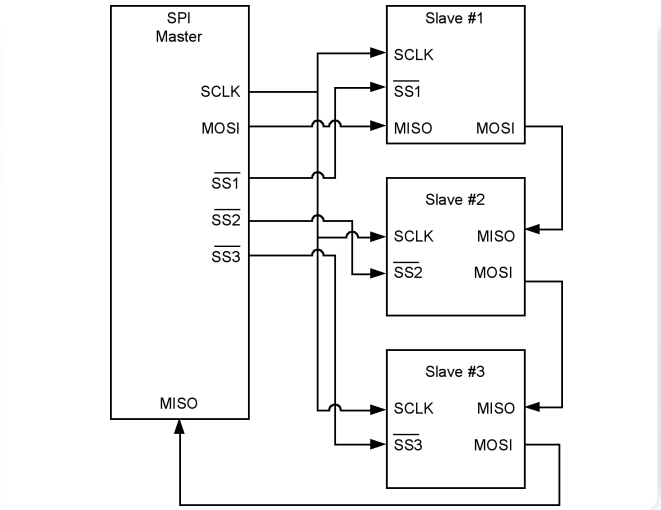
How It Works

The SPI (Serial Peripheral Interface)-bus is a master/slave, 4-wire serial communications bus. The four signals are clock (SCLK), master output/slave input (MOSI), master input/slave output (MISO), and slave select (SS). Whenever two devices communicate, one is referred to as the "master" and the other as the "slave". The master drives the serial clock. Data is simultaneously transmitted and received, making it a full-duplex protocol. Rather than having unique addresses for each device on the bus, SPI uses the SS line to specify which device data is being transferred to or from. As such, each unique device on the bus needs its own SS signal from the master. If there are 3 slave devices, there are 3 SS leads from the master, one to each slave as shown in Figure 8.

In Figure 8, each slave only talks to the master. However, SPI can be wired with the slave devices daisy-chained, each performing an operation in turn, and then sending the results back to the master as shown in Figure 9.



► **Figure 8.** *Common SPI configuration.*



► **Figure 9.** *Daisy-chained SPI configuration.*

So, as you can see, there is no "standard" for SPI implementation. In some cases, where communication from the slave back to the master is not required, the MISO signal may be left out all together.

▶ **Figure 10.** *SPI bus setup menu.*

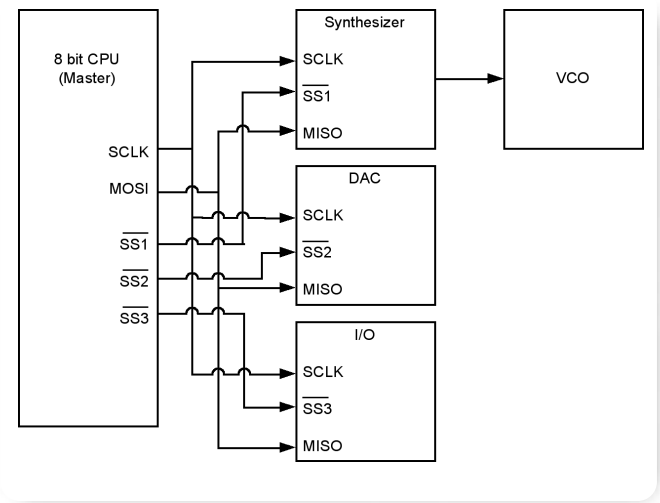
When an SPI data transfer occurs, an 8-bit data word is shifted out MOSI while a different 8-bit data word is being shifted in on MISO. This can be viewed as a 16-bit circular shift register. When a transfer occurs, this 16-bit shift register is shifted 8 positions, thus exchanging the 8-bit data between the master and slave devices. A pair of registers, clock polarity (CPOL) and clock phase (CPHA) determine the edges of the clock on which the data is driven. Each register has two possible states which allows for four possible combinations, all of which are incompatible with one another. So a master/slave pair must use the same parameter values to communicate. If multiple slaves are used that are fixed in different configurations, the master will have to reconfigure itself each time it needs to communicate with a different slave.

Working with SPI

The DPO4EMBD serial triggering and analysis application module also enables similar features for the SPI bus. Again, using the front panel B1 or B2 buttons we can define an SPI bus by simply entering the basic parameters of the bus including which channels SCLK, SS, MOSI, and MISO are on, thresholds, and polarities (see Figure 10).

As an example, consider the embedded system in Figure 11.

An SPI bus is connected to a synthesizer, a DAC, and some I/O. The synthesizer is connected to a VCO that



▶ **Figure 11.** *Synthesizer controlled via SPI.*

provides a 2.5 GHz clock to the rest of the system. The synthesizer is supposed to be programmed by the CPU at startup. However, something isn't working correctly as the VCO is stuck at its rail generating 3 GHz. The first step in debugging this problem is to inspect the signals between the CPU and the synthesizer to be sure the signals are present and there are no physical connection problems, but we don't find anything wrong. Next we decide to take a look at the actual information being transmitted across the SPI bus to program the synthesizer. To capture the information we set the

oscilloscope to trigger on the synthesizer's Slave Select signal going active and power up the DUT to capture the start up programming commands. The acquisition is shown in Figure 12.

Channel 1 (yellow) is SCLK, channel 2 (cyan) is MOSI and channel 3 (magenta) is SS. To help determine if we're programming the device correctly we take a look at the data sheet for the synthesizer. The first three messages on the bus are supposed to initialize the synthesizer, load the divider ratio, and latch the data. According to the spec, the last nibble (single hex character) in the first three transfers should be 3, 0, and 1, respectively, but we're seeing 0, 0, and 0. Upon seeing all 0's at the end of the messages we realize we've made one of the most common mistakes with SPI by programming the bits in each 24-bit word in reverse order in the software. A quick change in the software results in the following acquisition and a VCO correctly locked at 2.5 GHz as shown in Figure 13.

In the example above we used a simple SS Active trigger. The full SPI triggering capability in the DPO4000 Series includes the following types:

– SS Active – triggers when the slave select line goes true for a slave device.

– MOSI – trigger on up to 16 bytes of user specified data from the master to a slave.

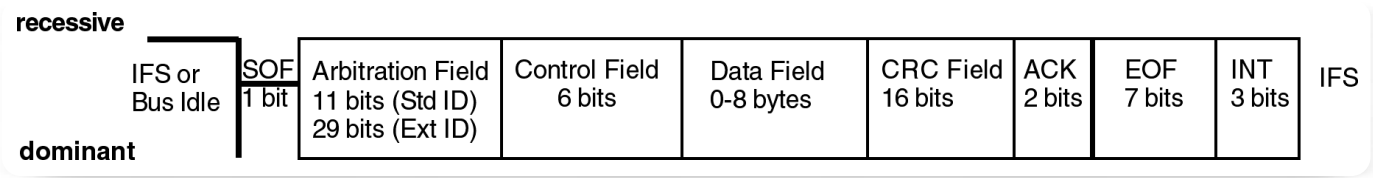
– MISO – trigger on up to 16 bytes of user specified data from a slave to the master.

– MOSI/MISO – trigger on up to 16 bytes of user specified data for both master to slave and slave to master.

Again, these triggers allow you to isolate the particular bus traffic you're interested in, while the decoding capability enables you to instantly see the content of every message transmitted over the bus in your acquisition.

▶ **Figure 12.** *Acquiring synthesizer configuration messages off the SPI bus.*

▶ **Figure 13.** *Correct synthesizer configuration messages.*



▶ **Figure 14.** *CAN Data/Remote Frame.*

CAN

Background

The Controller Area Network (CAN) was originally developed in the 1980's by the Robert Bosch GmbH, as a low cost communications bus between devices in electrically noisy environments. Mercedes-Benz became the first automobile manufacturer in 1992 to employ CAN in their automotive systems. Today almost every automobile manufacturer uses CAN controllers and networks to control devices such as: windshield wiper motor controllers, rain sensors, airbags, door locks, engine timing controls, anti-lock braking systems, power train controls and electric windows, to name a few. Due to its electrical noise tolerance, minimal wiring, excellent error detection capabilities and high speed data transfer, CAN is rapidly expanding into other applications such as industrial control, marine, medical, aerospace and more.

How It Works

The CAN bus is a balanced (differential) 2-wire interface running over either a Shielded Twisted Pair (STP), Un-shielded Twisted Pair (UTP), or ribbon cable. Each node uses a Male 9-pin D connector. Non Return to Zero (NRZ) bit encoding is used with bit stuffing to ensure compact messages with a minimum number of transitions and high noise immunity. The CAN Bus interface uses an asynchronous transmission scheme where any node may begin transmitting anytime the bus is free. Messages are broadcast to all nodes on the network. In cases where multiple nodes initiate messages at the same time, bitwise arbitration is used to determine which message is higher priority. Messages can be one of four types: Data Frame, Remote Transmission Request (RTR) Frame, Error Frame, or Overload Frame. Any node on the bus that detects an error transmits an error frame which causes all nodes on the bus to view the current message as incomplete and the transmitting node to resend the message. Overload frames are initiated by receiving devices to indicate they are not ready to receive data yet. Data frames are used to transmit data while Remote frames request data. Data and Remote frames are controlled by start and stop bits at the beginning and end of each frame and include the following fields: Arbitration field, Control field, Data field, CRC field and an ACK field as shown Figure 14.

– SOF - The frame begins with a start of frame (SOF) bit

– Arbitration - The Arbitration field includes an Identifier (address) and the Remote Transmission Request (RTR) bit used to distinguish between a data frame and a data request frame, also called a remote frame. The identifier can either be standard format (11 bits - version 2.0A) or extended format (29 bits - version 2.0B).

– Control - The Control Field consists of six bits including the Identifier Extension (IDE) bit which distinguishes between a CAN 2.0A (11 bit identifier) standard frame and a CAN 2.0B (29 bit identifier) extended frame. The Control Field also includes the Data Length Code (DLC). The DLC is a four bit indication of the number of bytes in the data field of a Data frame or the number of bytes being requested by a Remote frame.

– Data – The data field consists of zero to eight bytes of data.

– CRC - A fifteen bit cyclic redundancy check code and a recessive delimiter bit

– ACK - The Acknowledge field is two bits long. The first is the slot bit, transmitted as recessive, but then overwritten by dominant bits transmitted from any node that successfully receives the transmitted message. The second bit is a recessive delimiter bit

– EOF - Seven recessive bits indicate the end of frame (EOF).

The intermission (INT) field of three recessive bits indicates the bus is free. Bus Idle time may be any arbitrary length including zero.

A number of different data rates are defined, with 1Mb/s being the fastest, and 5kb/s the minimum rate. All modules must support at least 20kb/s. Cable length depends on the data rate used. Normally all devices in a system transfer information at uniform and fixed bit rates. The maximum line length can be thousands of meters at low speeds; 40 meters at 1Mbps is typical. Termination resistors are used at each end of the cable.

▶ **Figure 15.** *CAN bus setup menu.*

Working with CAN

The DPO4AUTO serial triggering and analysis application module enables similar triggering and analysis features for the CAN bus. Again, using the front panel B1 or B2 buttons we can define a CAN bus by simply entering the basic parameters of the bus including the type of CAN signal being probed and on which channel, the bit rate, threshold and sample point (as a % of bit time), see Figure 15.

Imagine you need to make timing measurements associated with the latency from when a driver presses the Passenger Window Down switch to when the CAN module in the driver's door issues the command and then the time to when the passenger window actually starts to move. By specifying the ID of the CAN module in the driver's door as well as the data associated with a "roll the window down" command, you can trigger on the exact data frame you're looking for. By simultaneously probing the window down switch on the driver's door and the motor drive in the passenger's door this timing measurement becomes exceptionally easy, as shown in Figure 16.

▶ **Figure 16.** *Triggering on specific Identifier and Data on a CAN bus and decoding all messages in the acquisition.*

The white triangles in the figure are marks that we've placed on the waveform as reference points. These marks are added to or removed from the display by simply pressing the Set/Clear Mark button on the front panel of the oscilloscope. Pressing the Previous and Next buttons on the front panel causes the zoom window to jump from one mark to the next making it simple to navigate between events of interest in the acquisition.

Now imagine performing this task without these capabilities. Without the CAN triggering you would have to trigger on the switch itself, capture what you hope is a long enough time window of activity and then begin manually decoding frame after frame after frame on the CAN bus until you finally find the right one. What could have taken tens of minutes or hours before can now be accomplished in moments.

The DPO4000's powerful CAN triggering capability includes the following types:

– Start of Frame – trigger on the SOF field

– Frame Type – choices include Data Frame, Remote Frame, Error Frame, and Overload Frame

– Identifier – trigger on specific 11 or 29 bit identifier values with Read / Write qualification.

– Data – trigger on 1-8 bytes of user specified data

– Missing Ack – trigger anytime the receiving device does not provide an acknowledge

– End of Frame – trigger on the EOF field

These trigger types enable you to isolate virtually anything you're looking for on a CAN bus effortlessly. Triggering is just the beginning though. Troubleshooting will often require inspecting message content both

▶ **Figure 17.** *CAN event table.*

before and after the trigger event. A simple way to view the contents of multiple messages in an acquisition is with the DPO4000 Series' Event Table, as shown in Figure 17.

The event table shows decoded message content for every message in an acquisition in a tabular format with timestamps. This makes it easy to not only view all the traffic on the bus but also enables easy timing measurements between messages. Event Tables are available for I^2C and SPI buses as well.

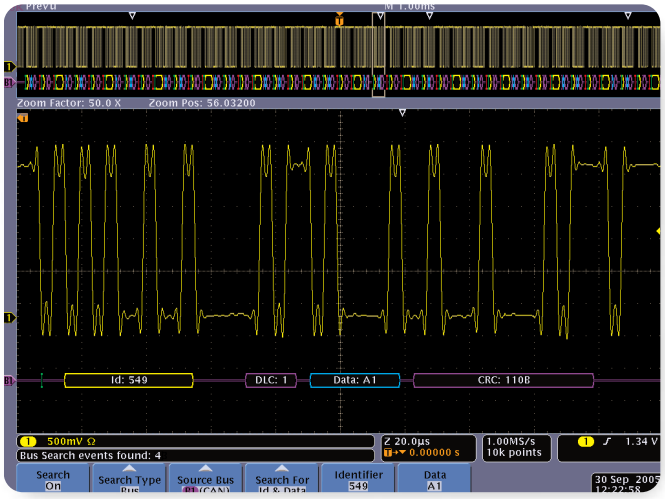
Triggering vs. Search

As we've discussed throughout this application note, a capable triggering system is required to isolate the event of interest on the serial bus. However, once you've acquired the data (the scope is stopped), and you want to analyze it, triggering doesn't apply any more. Wouldn't it be nice if the scope had trigger-like resources for analyzing stopped waveform data? The DPO4000 Series' Wave Inspector provides this capability with its powerful search feature. All of the bus trigger features discussed throughout this document are also available as search criteria on already acquired data. For example, in Figure 18 the oscilloscope has searched through a long acquisition for every CAN message that has specific address and data content and marked each one with a hollow white triangle at the top of the display. Navigating between occurrences is as simple as pressing the front panel Previous and Next buttons.

Of course, searches are also available for the more traditional trigger types as well. Search types include edges, pulse widths, runt, setup & hold times, logic and rise/fall times.

Conclusion

While there are many benefits in transitioning from parallel to serial buses in embedded systems design, there are also a number of challenges the design engineer faces. With traditional test and measurement tools it's much more difficult to trigger on the event you're looking for, it can be nearly impossible to tell what information is present by just looking at the analog signal and it's an extremely time consuming and error prone process to have to manually decode a long period of bus activity to diagnose problems. The DPO4000 Series changes everything. With its powerful trigger, decode, and search capabilities today's design engineers can solve embedded system design issues with exceptional efficiency.



▶ **Figure 18.** *Searching on specified Identifier and Data in a CAN bus acquisition.*

Contact Tektronix:

ASEAN / Australasia / Pakistan (65) 6356 3900
Austria +41 52 675 3777
Balkan, Israel, South Africa and other ISE Countries +41 52 675 3777
Belgium 07 81 60166
Brazil & South America 55 (11) 3741-8360
Canada 1 (800) 661-5625
Central East Europe, Ukraine and Baltics +41 52 675 3777
Central Europe & Greece +41 52 675 3777
Denmark +45 80 88 1401
Finland +41 52 675 3777
France & North Africa +33 (0) 1 69 86 81 81
Germany +49 (221) 94 77 400
Hong Kong (852) 2585-6688
India (91) 80-22275577
Italy +39 (02) 25086 1
Japan 81 (3) 6714-3010
Luxembourg +44 (0) 1344 392400
Mexico, Central America & Caribbean 52 (55) 56666-333
Middle East, Asia and North Africa +41 52 675 3777
The Netherlands 090 02 021797
Norway 800 16098
People's Republic of China 86 (10) 6235 1230
Poland +41 52 675 3777
Portugal 80 08 12370
Republic of Korea 82 (2) 528-5299
Russia & CIS 7 095 775 1064
South Africa +27 11 254 8360
Spain (+34) 901 988 054
Sweden 020 08 80371
Switzerland +41 52 675 3777
Taiwan 886 (2) 2722-9622
United Kingdom & Eire +44 (0) 1344 392400
USA 1 (800) 426-2200

For other areas contact Tektronix, Inc. at: 1 (503) 627-7111

Last Updated June 15 2005

For Further Information
Tektronix maintains a comprehensive, constantly expanding collection of application notes, technical briefs and other resources to help engineers working on the cutting edge of technology. Please visit **www.tektronix.com**

10/05 FLG/WOW 48W-19040-0

Tektronix
Enabling Innovation