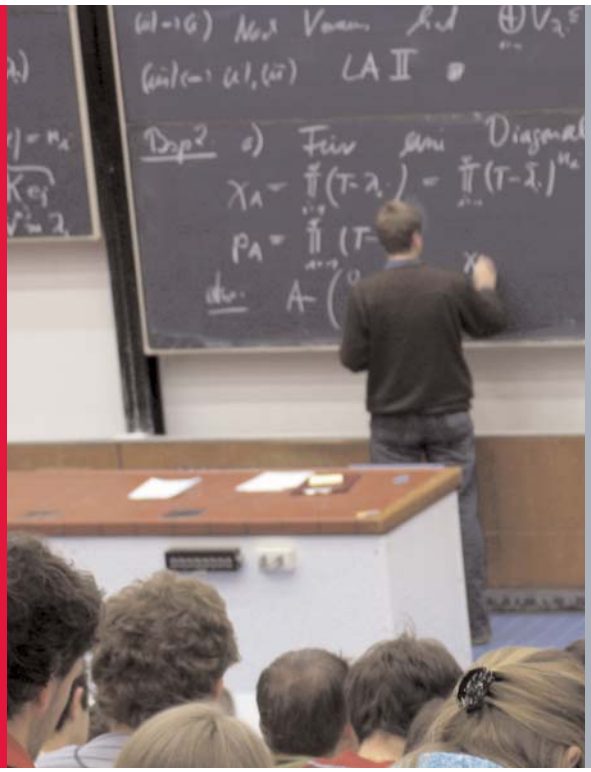


Low Speed Serial Data Fundamentals

Tektronix®

Table of Contents

Introduction 3

Parallel vs. Serial 4

I²C . . . . . 6 - 9

Background 6

How It Works 6

Working with I²C 7

SPI . . . . . 9 - 12

Background 9

How It Works 9

Working with SPI 10

RS-232 . . . . . 12 - 14

Background 12

How It Works 12

Working with RS-232 14

CAN . . . . . 15 - 17

Background 15

How It Works 15

Working with CAN 16

LIN . . . . . 18 - 20

Background 18

How It Works 18

Working with LIN 19

FlexRay . . . . . 20 - 24

Background 20

How It Works 21

Working with FlexRay 23

Audio Buses . . . . . 25 - 28

Background 25

How It Works 25

Working with Audio Buses 28

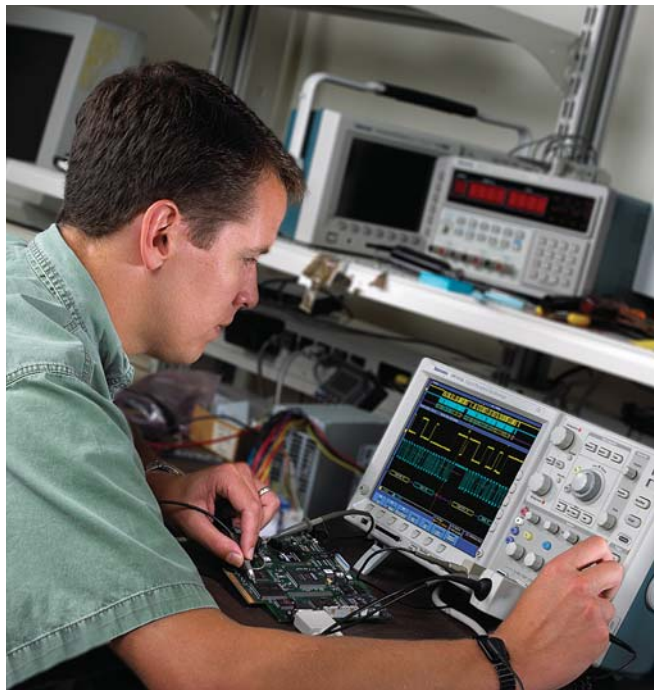
Triggering vs. Search 29

Conclusion 29

Product Comparison Chart 29

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 (PDAs)
- 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

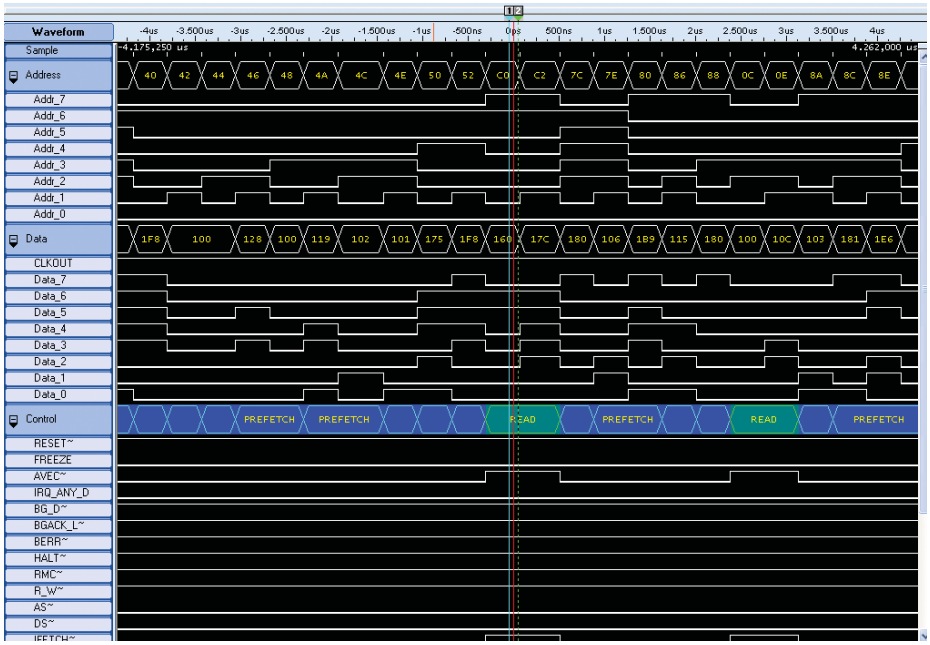


Figure 1. Logic Analyzer acquisition of a microcontroller's clock, address bus, data bus and control lines.

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 MSO/DPO Series – MSO/DPO4000, DPO3000 and MSO/DPO2000 Series – oscilloscopes.

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.

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 I^2C, it is still significantly harder to observe what is being transmitted over the bus than it is with a parallel protocol.

I^2C 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.

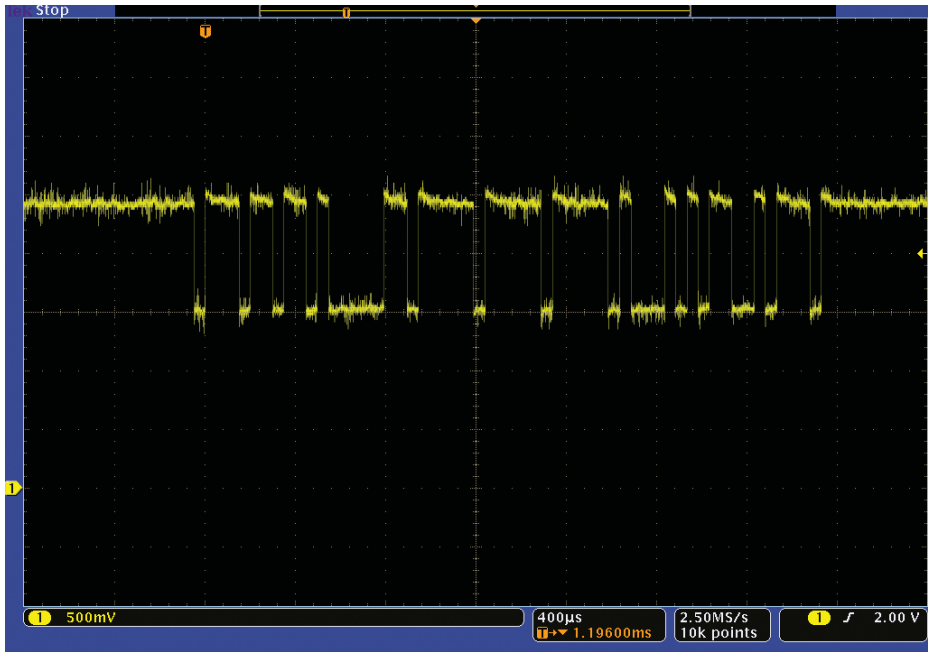


Figure 2. One message acquired from a CAN bus.

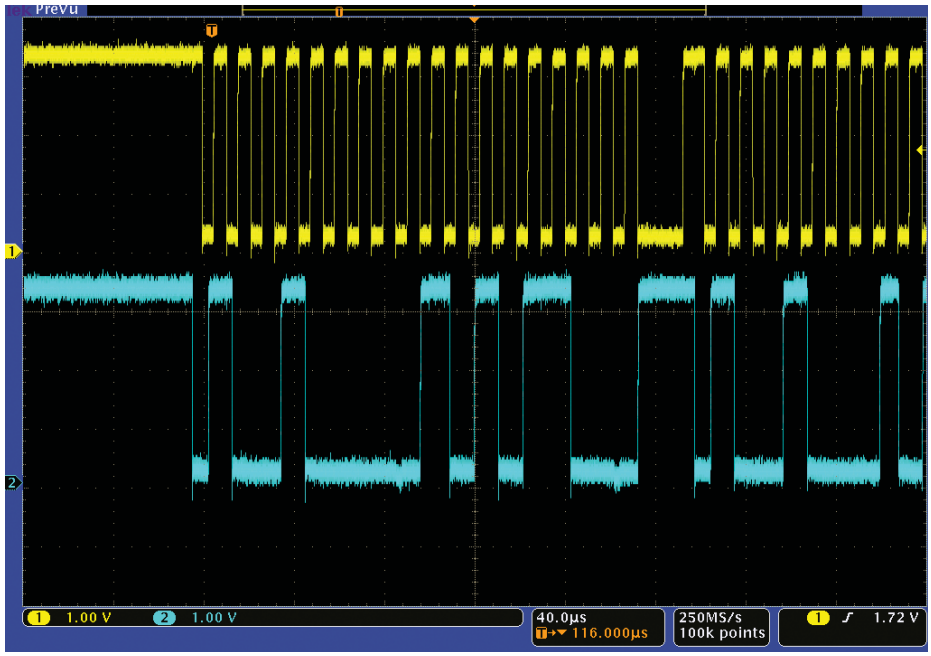


Figure 3. One message acquired from an I^2C bus.

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 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!

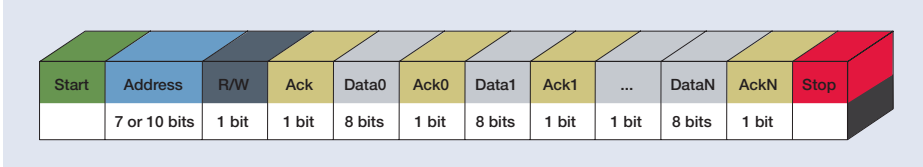


Figure 4. I²C message structure.

With the MSO/DPO Series – MSO/DPO4000, DPO3000 and MSO/DPO2000 Series – there is a better way. The following sections highlight how the MSO/DPO 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 1980s to provide a low-cost way of connecting controllers to peripheral chips in TV sets, but has since evolved into a worldwide standard for communication between devices in embedded systems. The simple two-wire design has found its way into a wide variety of chips like I/O, A/Ds, D/As, 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 a 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. A device can operate as either a transmitter or a receiver, depending on its function. Initially, I²C only used 7-bit addresses, but evolved to allow 10-bit addressing as well. Three bit rates are supported: 100 kb/s (standard mode), 400 kb/s (fast mode), and 3.4 Mb/s (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 from or written to the device.
- Stop - indicates the message is complete and the master has released the bus.

There are two ways to group I²C addresses for decoding: in 7-bits plus a read or write (R/W) bit scheme, and in 8-bits (a byte) where the R/W bit is included as part of the address. The 7-bit address scheme is the specified I²C Standard followed by firmware and software design engineers. But many other engineers use the 8-bit address scheme. The MSO/DPO Series oscilloscopes can decode data in either scheme.

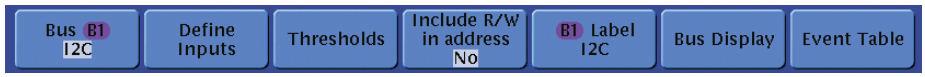


Figure 5. I²C bus set-up menu.

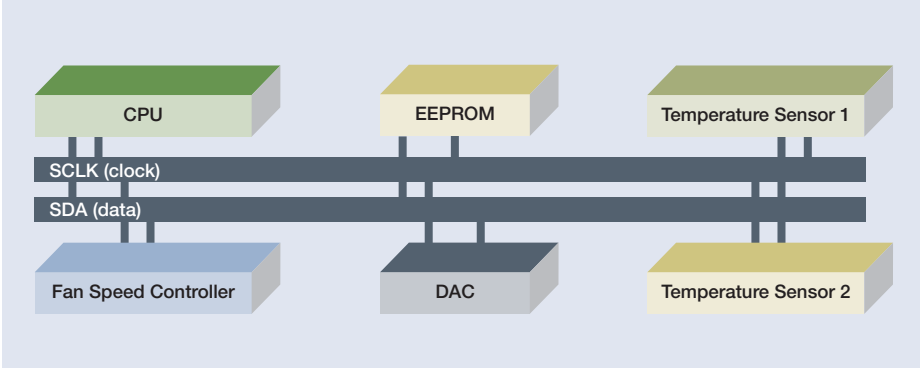


Figure 6. I²C bus example.

Working with I²C

With the DPOxEMBD serial triggering and analysis application module, the MSO/DPO Series becomes a powerful tool for embedded system designers working with I²C buses. The front panel has Bus buttons 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 ones and zeroes, 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 while 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 MSO/DPO Series. 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 MSO/DPO Series use the colors and marks in Table 1 to indicate important parts of the message.

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 MSO/DPO Series.

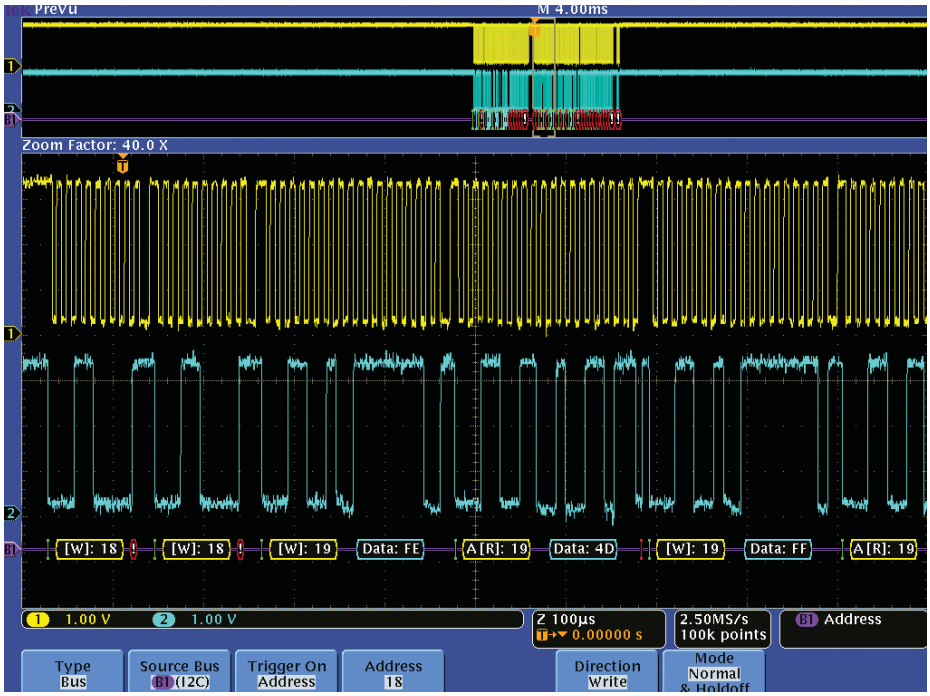


Figure 7. I²C address and data bus waveform decoding.

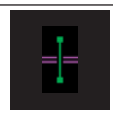
Bus Condition	Indicated by:
Starts are indicated by vertical green bars. Repeated starts occur when another start is shown without a previous Stop.	
Addresses are shown in yellow boxes along with a [W] for write or [R] for read. Address values can be displayed in either hex or binary.	
Data is shown 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 MSO/DPO Series' powerful I^2C 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.

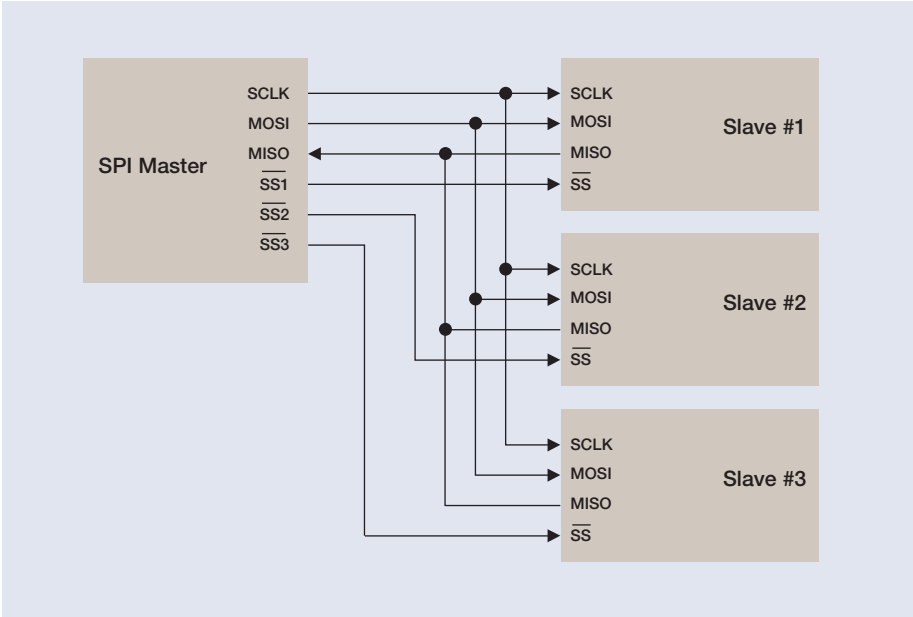


Figure 8. Common SPI configuration.

SPI

Background

The Serial Peripheral Interface bus (SPI) was originally developed by Motorola in the late 1980s 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, PDAs, and other mobile devices to communicate data between the CPU, keyboard, display, and memory chips.

How It Works

The SPI 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.

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. In other cases there is only one master and one slave device and the SS signal is tied to ground. This is commonly referred to as 2-wire SPI.

When an SPI data transfer occurs, an 8-bit data word is shifted out on 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.

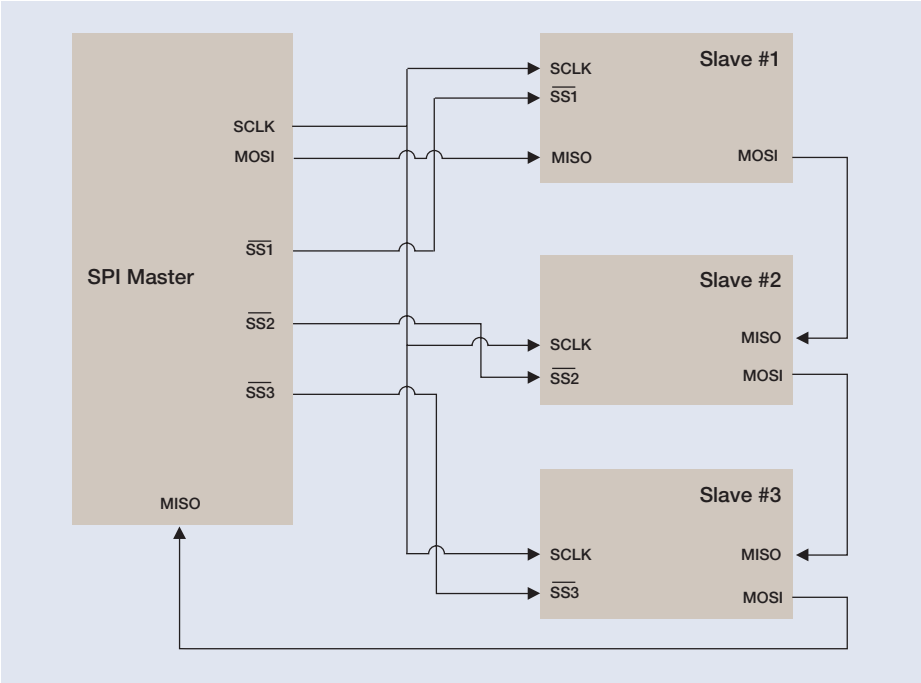


Figure 9. Daisy-chained SPI configuration.

Figure 10. SPI bus setup menu.

Working with SPI

The DPOxEMBD serial triggering and analysis application module enables decoding and triggering for the SPI bus. Again, using the front panel Bus 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 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.

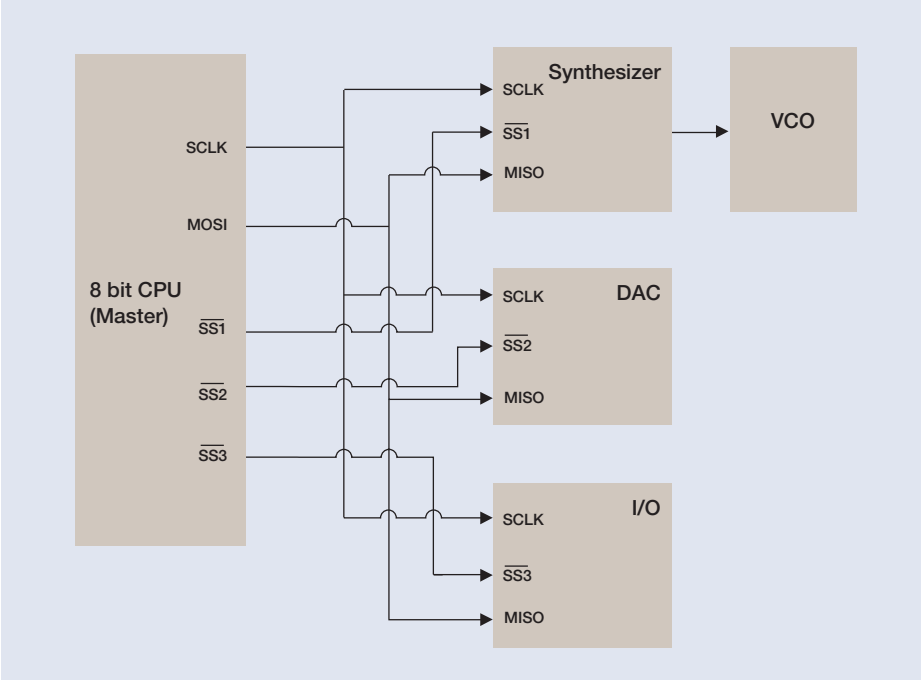


Figure 11. Synthesizer controlled via SPI.

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

Upon seeing all 0s 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 MSO/DPO 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.

Figure 13. Correct synthesizer configuration messages.

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.

RS-232

Background

RS-232 is a widely-used standard for serial communication between two devices over a short distance. It is best known for its use in PC serial ports, but it is also used in embedded systems as a debug port or for linking two devices.

The RS-232-C standard was introduced in 1969. The standard has been revised twice since then, but the changes are minor and the signals are interoperable with RS-232-C. There are also related standards, such as RS-422 and RS-485, which are similar but use differential signaling to communicate over longer distances.

How it Works

The two devices are referred to as the DTE (data terminal equipment) and DCE (data circuit-terminating equipment). In some applications, the DTE device controls the DCE device; in other applications, the two devices are peers and the distinction between DTE and DCE is arbitrary.

The RS-232 standard specifies numerous signals, many of which are not commonly used. The two most important signals are Transmitted Data (Tx) and Received Data (Rx). Tx carries data from the DTE to the DCE. The DTE device's Tx line is the DCE device's Rx line. Similarly, Rx carries data from the DCE to the DTE.

The RS-232 standard does not specify which connectors to use. Twenty-five-pin and nine-pin connectors are most common. Other connectors have ten, eight, or six pins. It's also possible to connect two RS-232 devices on the same board, without using standard connectors.

When connecting two RS-232 devices, a null modem is commonly required. This device swaps several lines, including the Tx and Rx lines. That way, each device can send data on its Tx line and receive data on its Rx line.

Table 2 shows the pinout used for a 9-pin connector, commonly used with RS-232 signals. Remember that if your signal has passed through a null modem, many of the signals will be swapped. Most importantly, Tx and Rx will be swapped.

When probing RS-232 signals, it is often helpful to use a breakout box. This device allows you to easily probe the signals inside an RS-232 cable. Breakout boxes are inexpensive and readily available from electronics dealers.

The RS-232 standard does not specify the content transmitted across the bus. ASCII text is most common, but binary data is also used. The data is often broken up into packets. With ASCII text, packets are commonly terminated by a new line or carriage return character. With binary data other values, such as 00 or FF hex are commonly used.

Devices often implement RS-232 using a universal asynchronous receiver/transmitter (UART). UARTs are widely available in off-the-shelf parts. The UART uses a shift register to convert a byte of data into a serial stream, and vice versa. In embedded designs, UARTs can also communicate directly without the use of RS-232 transceivers.

Signal		Pin
Carrier Detect	DCD	1
Received Data	Rx	2
Transmitted Data	Tx	3
Data Terminal Ready	DTR	4
Common Ground	G	5
Data Set Ready	DSR	6
Request to Send	RTS	7
Clear to Send	CTS	8
Ring Indicator	RI	9

Table 2. Common RS-232 connector pinout.

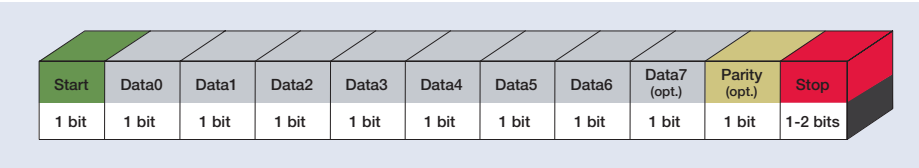


Figure 14. RS-232 byte structure.

Figure 14 shows one byte of RS-232 data. The byte is composed of these bits.

- Start - The byte begins with a start bit.
- Data - Several bits of data follow.
  Eight bits of data is the most common; some applications use seven bits of data. Even when only seven bits are transmitted, the data is often informally referred to as a byte. In UART to UART communication, 9 bit data words are sometimes used.
- Parity - An optional parity bit.
- Stop - 1, 1.5, or 2 stop bits.

An RS-232 bus does not have a clock line. Each device uses its own clock to determine when to sample the data lines. In many designs, a UART uses the rising edges of the Tx and Rx signals to synchronize its clock with the other device's clock.

Working With RS-232

The DPOxCOMP application module enables serial triggering and analysis for the RS-232 bus. You can view your RS-232, RS-422, RS-485, or UART data conveniently on your oscilloscope, without needing to attach to a PC or a specialized decoder.

Using the front-panel bus buttons we can define an RS-232 bus by entering basic parameters, such as the channels being used, bit rate, and parity (see Figure 15).

In this example, we have chosen ASCII decoding; the MSO/DPO Series can also display RS-232 data as binary or hex.

Imagine you have a device that polls a sensor for data over an RS-232 bus. The sensor isn't responding to requests for data. You want to find out if the sensor isn't receiving the requests, or if it is receiving the requests but ignoring them.

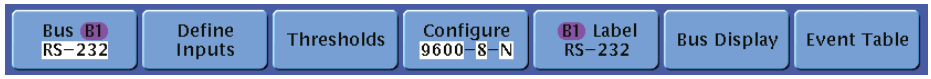


Figure 15. RS-232 bus set-up menu.

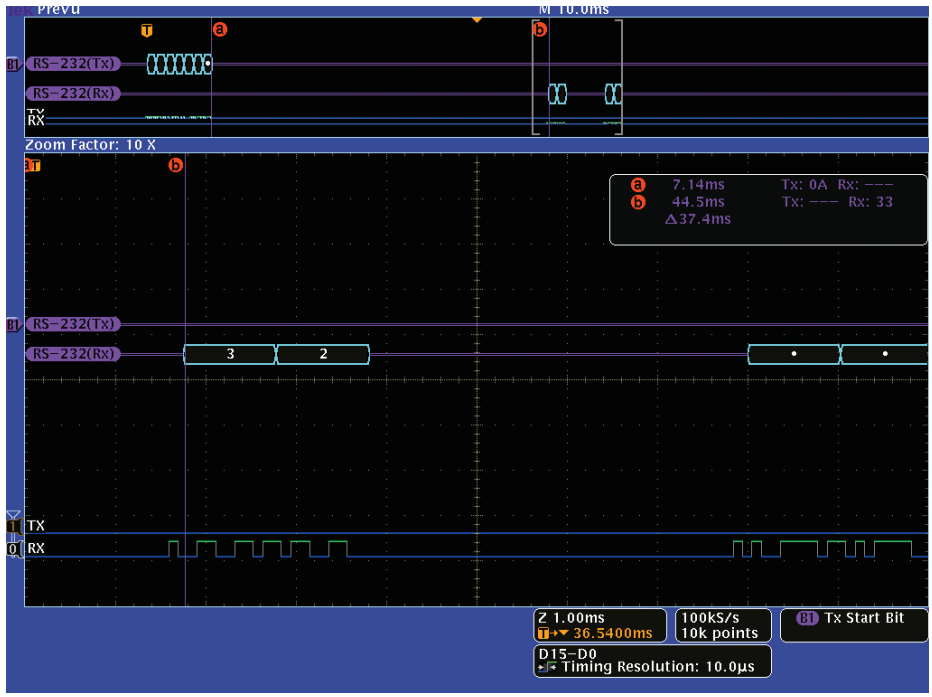


Figure 16. Measuring time delay between messages on two RS-232 buses.

First, probe the Tx and Rx lines and set up a bus on the oscilloscope. Then set the oscilloscope to trigger when the request for data is sent across the Tx line. The triggered acquisition is shown in Figure 16.

Here, we can see the Tx line on digital channel 1, and the Rx line on digital channel 0. But we're more interested in the decoded data, shown above the raw waveforms. We've zoomed in to look at the response from the sensor. The overview shows the request on the Tx line and the response on the Rx line. The cursors show us that the reply comes around 37ms after the end of the request. Increasing the controller's timeout fixes the problem by giving enough time for the sensor to reply.

The MSO/DPO Series oscilloscope's RS-232 trigger includes these capabilities:

- Tx Start Bit - triggers on the bit indicating the start of a byte.
- Tx End of Packet - triggers on the last byte in a packet. A packet can be ended by a specific byte: Null (00 hex), linefeed (0A hex), carriage return (0D hex), space (20 hex), or FF hex.
- Tx Data - triggers on up to 10 bytes of user-specified data values.
- Rx Start Bit, Rx End of Packet, and Rx Data - these are like the Tx triggers, but on the Rx line.

With the MSO/DPO Series oscilloscope, you can easily view RS-232 signals, analyze them, and correlate them to other activity in your device.

CAN

Background

The Controller Area Network (CAN) was originally developed in the 1980s by the Robert Bosch GmbH as a low cost communications bus between devices in electrically noisy environments.

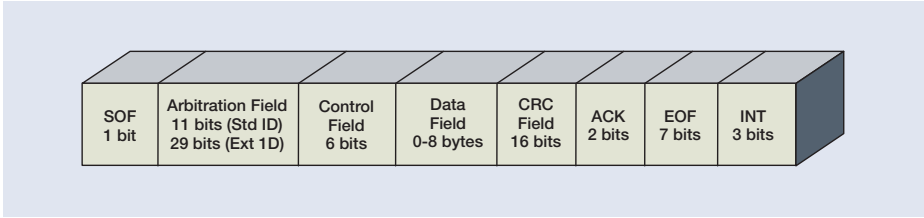


Figure 17. CAN Data/Remote Frame.

Mercedes-Benz became the first automobile manufacturer in 1992 to employ CAN in their automotive systems. Today, every automotive manufacturer uses CAN controllers and networks to control a variety of devices in the automobile. A newer and even lower cost bus called LIN (discussed next) was developed to address applications where the cost, versatility, and speed of CAN were overkill. LIN has displaced CAN in a number of applications, but CAN is still the primary bus used for engine timing controls, anti-lock braking systems and power train controls to name a few. And 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 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 17.

- 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 1Mb/s is typical. Termination resistors are used at each end of the cable.

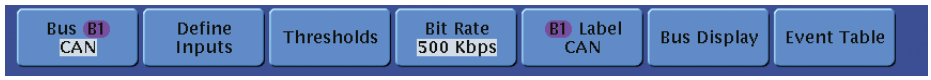


Figure 18. CAN bus setup menu.

Working with CAN

The DPOxAUTO and DPO4AUTOMAX serial triggering and analysis application modules of the MSO/DPO Series enable similar triggering and analysis features for the CAN bus. Again, using the front panel Bus 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 percent of bit time), see Figure 18.

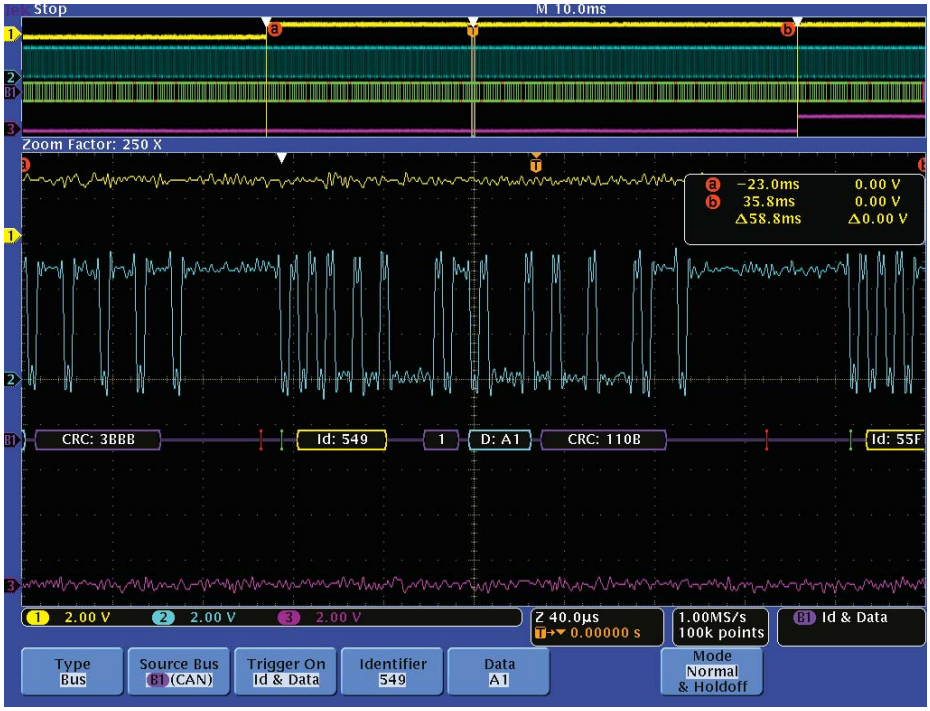


Figure 19. Triggering on specific identifier and Data on a CAN bus and decoding all messages in the acquisition.

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 19.

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 MSO/DPO Series' powerful CAN triggering capability includes the following types:

- Start of Frame – trigger on the SOF field.
- Frame Type – choices are 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 before and after the trigger event. A simple way to view the contents of multiple messages in an acquisition is with the MSO/DPO Series' Event Table, as shown in Figure 20.

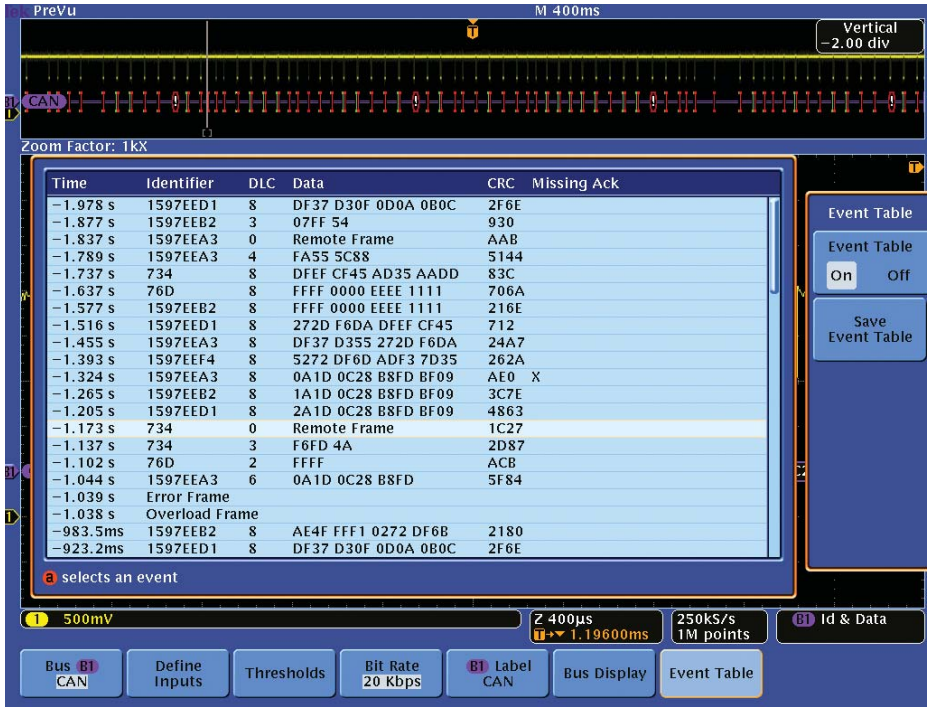


Figure 20. CAN event table.

The event table shows decoded message content for every message in an acquisition in a tabular format with time-stamps. 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 all types of buses the MSO/DPO Series oscilloscope supports.

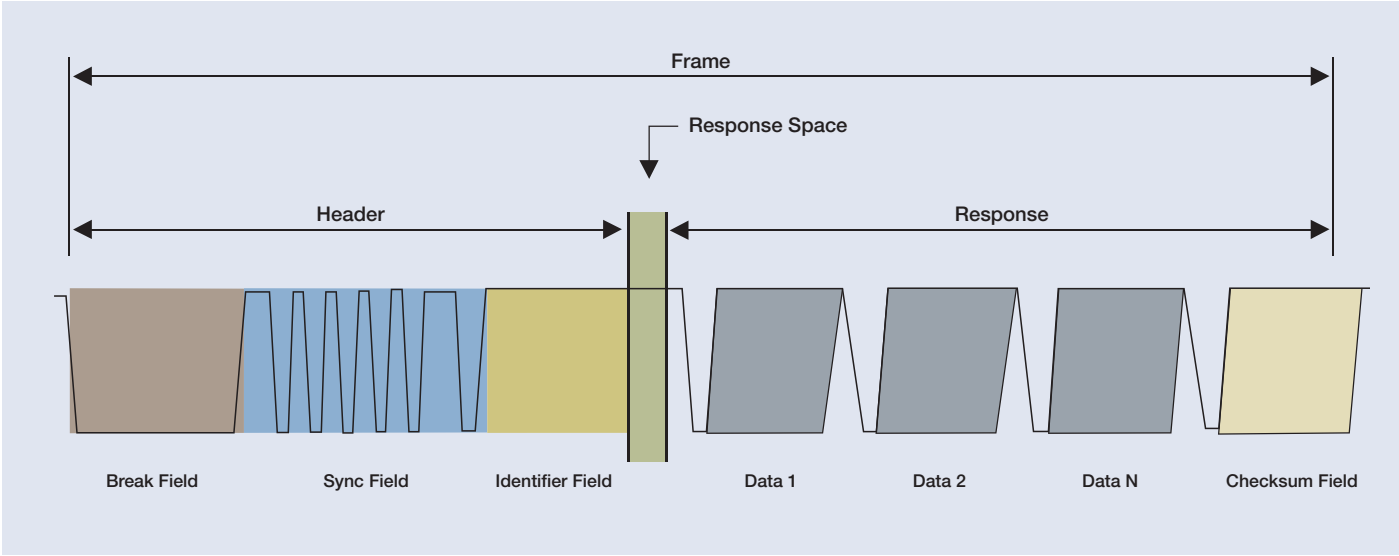


Figure 21. The structure of a LIN frame.

LIN

Background

The Local Interconnect Network (LIN) bus was developed by the LIN consortium in 1999 as a lower cost alternative to the CAN bus for applications where the cost, versatility, and speed of CAN were overkill. These applications typically include communications between intelligent sensors and actuators such as window controls, door locks, rain sensors, windshield wiper controls, and climate control, to name a few.

However, due to its electrical noise tolerance, error detection capabilities, and high speed data transfer, CAN is still used today for engine timing controls, anti-lock braking systems, power train controls and more.

How It Works

The LIN bus is a low-cost, single-wire implementation based on the Enhanced ISO9141 standard. LIN networks have a single master and one or more slaves. All messages are initiated by the master with only one slave responding to each message, so collision detection and arbitration capabilities are not needed as they are in CAN. Communication is based on UART/SCI with data being sent in eight-bit bytes along with a start bit, stop bit and no parity. Data rates range from 1kb/s to 20kb/s. While this may sound slow, it is suitable for the intended applications and minimizes EMI. The LIN bus is always in one of two states: active or sleep. When it's active, all nodes on the bus are awake and listening for relevant bus commands. Nodes on the bus can be put to sleep by either the Master issuing a Sleep Frame or the bus going inactive for longer than a predetermined amount of time. The bus is then awakened by any node requesting a wake up or by the master node issuing a break field.

LIN frames consist of two main parts, the header and the response. The header is sent by the master while the response is sent by the slave. The header and response each have sub-components as shown in Figure 21.

Header Components:

- Break Field – the break field is used to signal the beginning of a new frame. It activates and instructs all slave devices to listen to the remainder of the header.
- Sync Field – the sync field is used by the slave devices to determine the baud rate being used by the master node and synchronize themselves accordingly
- Identifier Field – the identifier specifies which slave device is to take action

Response Components:

- Data – the specified slave device responds with one to eight bytes of data
- Checksum – computed field used to detect errors in data transmission. The LIN standard has evolved through several versions that have used two different forms of checksums. Classic checksums are calculated only over the data bytes and are used in version 1.x LIN systems. Enhanced checksums are calculated over the data bytes and the identifier field and are used in version 2.x LIN systems.

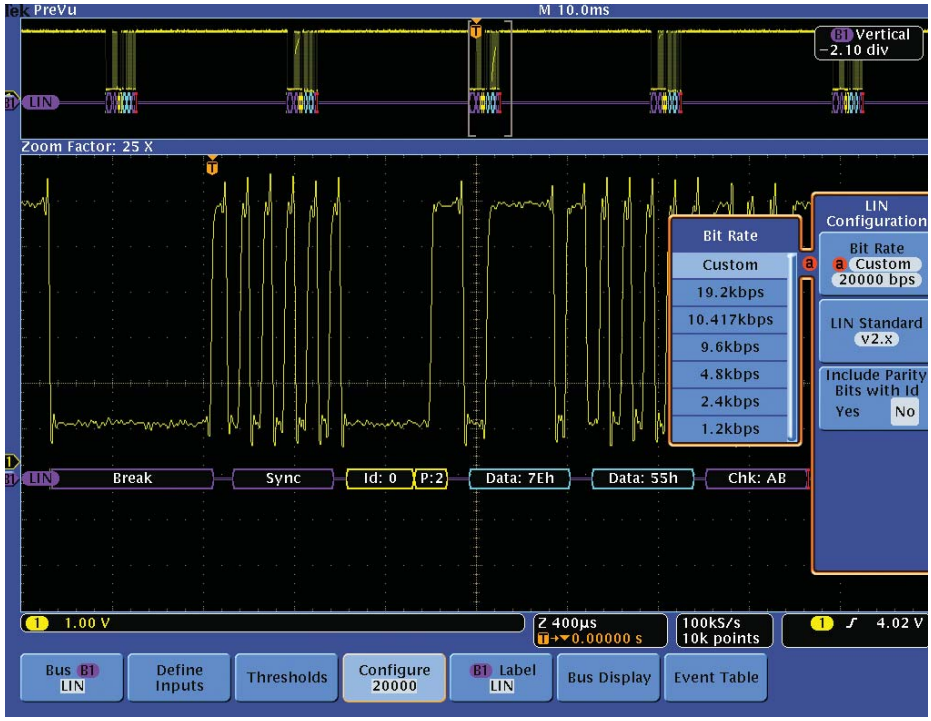


Figure 22. LIN bus setup menu and decoded frame.

Working with LIN

LIN support on the MSO/DPO Series is available via either the DPOxAUTO or DPO4AUTOMAX serial triggering and analysis application module. Again, using the front panel Bus buttons we can define a LIN bus by simply entering the basic parameters of the bus such as the LIN version being used, the bit rate, polarity, threshold, and where to sample the data (as a percent of bit time). The LIN setup menu along with a decoded LIN frame is shown in Figure 22.

A powerful feature of the MSO/DPO Series is the ability to define and decode up to four serial buses simultaneously. Going back to our earlier example with CAN bus; now imagine that the window controls are operated by a LIN bus. When the driver presses the Passenger Window Down control, a message is initiated on a LIN bus in the driver door, passed through a central CAN gateway and then sent on to another LIN network in the passenger door. In this case, we can trigger on the relevant message on one of the buses and capture and decode all three buses simultaneously, making it exceptionally easy to view traffic as it goes from one bus to another through the system. This is shown in Figure 23 where we've triggered on the first LIN message and captured all three buses.

Figure 23. Simultaneous capture and decode of multiple automotive serial buses.

The MSO/DPO Series LIN triggering capability includes the following types:

- Sync – trigger on the sync field
- Identifier – trigger on a specific identifier
- Data – trigger on 1-8 bytes of specific data values or data ranges
- Identifier & Data – trigger on a combination of both identifier and data
- Wakeup Frame – trigger on a wakeup frame
- Sleep Frame – trigger on a sleep frame
- Error – trigger on sync errors, ID parity errors, or checksum errors

These trigger types allow you to isolate anything you're looking for on a LIN bus faster than ever before. And with the other advanced serial features found in the MSO/DPO Series such as event tables and search & mark, debugging LIN based automotive designs has never been easier.

FlexRay

Background

FlexRay is a relatively new automotive bus that is still being developed by a group of leading automotive companies and suppliers known as the FlexRay Consortium. As cars get smarter and electronics find their way into more and more automotive applications, manufacturers are finding that existing automotive serial standards such as CAN and LIN do not have the speed, reliability, or redundancy required to address X-by-wire applications such as brake-by-wire or steer-by-wire. Today, these functions are dominated by mechanical and hydraulic systems. In the future they will be replaced by a network of sensors and highly reliable electronics that will not only lower the cost of the automobile, but also significantly increase passenger safety due to intelligent electronic based features such as anticipatory braking, collision avoidance, adaptive cruise control, etc.

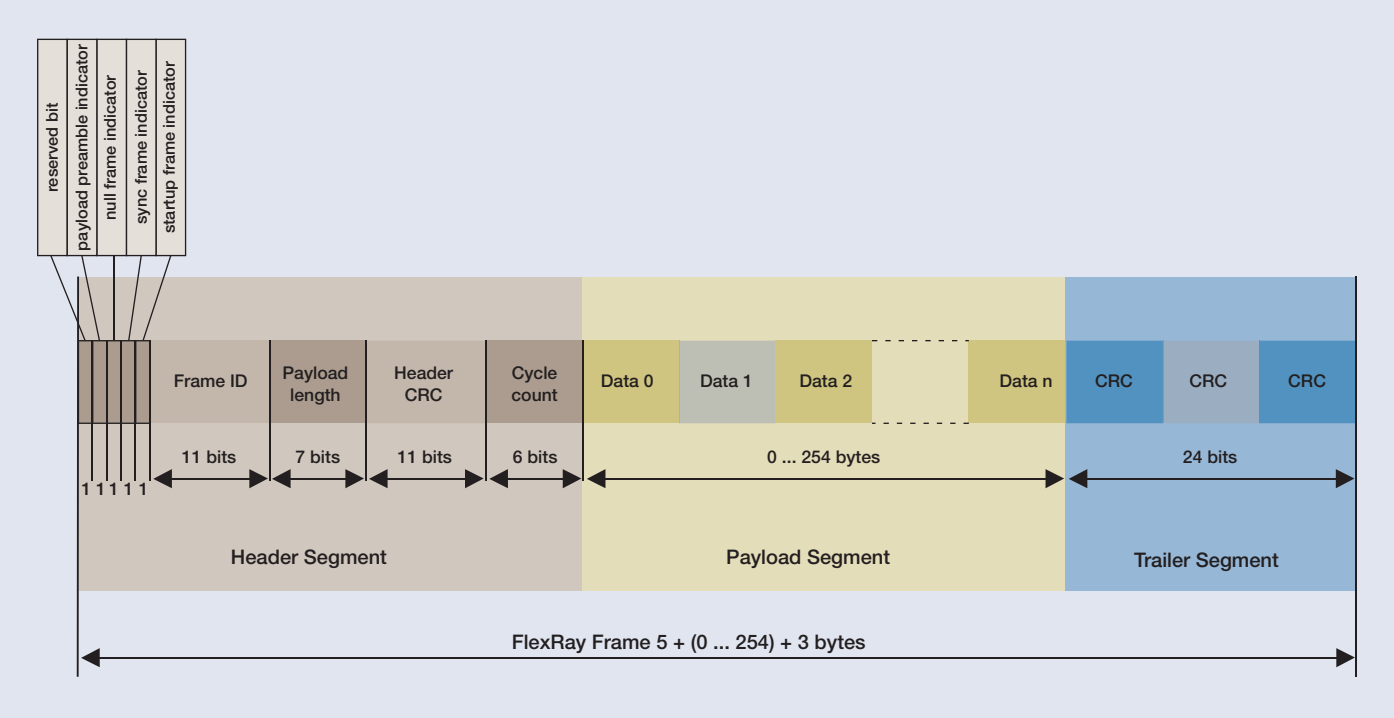


Figure 24. FlexRay frame structure.

How It Works

FlexRay is a differential bus running over either a Shielded Twisted Pair (STP) or an Un-shielded Twisted Pair (UTP) at speeds up to 10 Mb/s, significantly faster than LIN's 20 kb/s or CAN's 1 Mb/s rates. FlexRay uses a dual channel architecture which has two major benefits. First, the two channels can be configured to provide redundant communication in safety critical applications such as x-by-wire to ensure the message gets through. Second, the two channels can be configured to send unique information on each at 10 Mb/s, giving an overall bus transfer rate of 20 Mb/s in less safety-critical applications.

FlexRay uses a time triggered protocol that incorporates the advantages of prior synchronous and asynchronous protocols via communication cycles that include both static and dynamic frames. Static frames are time slots of predetermined length allocated for each device on the bus to communicate during each cycle. Each device on the bus is also given a chance to communicate during each cycle via a Dynamic frame which can vary in length (and time). The FlexRay frame is made up of three major segments; the header segment, the payload segment, and the trailer segment. These segment each have their own components as shown in Figure 24.

Header Segment Components:

- Indicator Bits – the first five bits are called the indicator bits and indicate the type of frame being transmitted. Choices include Normal, Payload, Null, Sync, and Startup.
- Frame ID – the frame ID defines the slot in which the frame should be transmitted. Frame IDs range from 1 to 2047 with any individual frame ID being used no more than once on each channel in a communication cycle.
- Payload Length – the payload length field is used to indicate how many words of data are in the payload segment.
- Header CRC – a cyclic redundancy check (CRC) code calculated over the sync frame indicator, the startup frame indicator, the frame ID and the payload length.
- Cycle Count – the value of the current communication cycle, ranging from 0-63.

Figure 25. FlexRay bus setup menu.

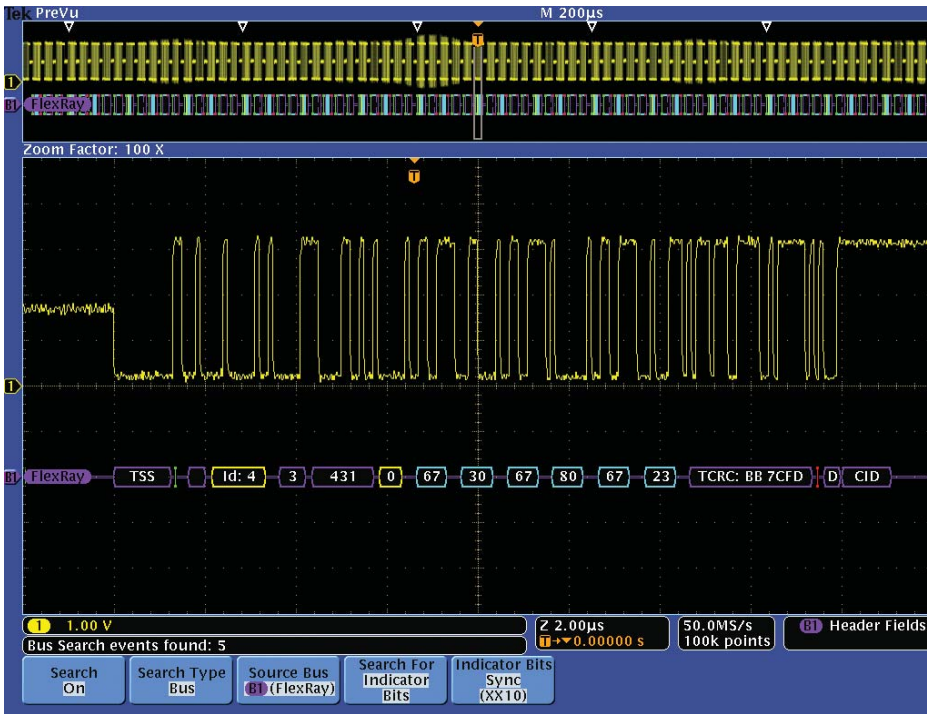


Figure 26. Triggering on Frame ID and Cycle Count, Searching through acquired data for Startup Frames.

Payload Segment Components:

- Data – the data field contains up to 254 bytes of data. For frames transmitted in the static segment the first 0 to 12 bytes of the payload segment may optionally be used as a network management vector. The payload preamble indicator in the frame header indicates whether the payload segment contains the network management vector. For frames transmitted in the dynamic segment the first two bytes of the payload segment may optionally be used as a message ID field, allowing receiving nodes to filter or steer data based on the contents of this field. The payload preamble indicator in the frame header indicates whether the payload segment contains the message ID.

Trailer Segment Components:

- CRC – a cyclic redundancy check (CRC) code calculated over all of the components of the header segment and the payload segment of the frame.

Dynamic frames have one additional component that follows the Trailer CRC called the Dynamic Trailing Sequence (DTS) that prevents premature channel idle detection by the bus receivers.

Working with FlexRay

FlexRay support on the MSO/DPO4000 Series is available via the DPO4AUTOMAX module which provides serial triggering and analysis capabilities on all three automotive standards - CAN, LIN, and FlexRay, as well as eye diagram analysis and critical timing measurements on FlexRay. To define a FlexRay bus, we go to the bus menu and select FlexRay from the list of supported standards. The FlexRay setup menu is shown in Figure 25.

Next, we use the Define Inputs menu to tell the scope whether we're looking at FlexRay channel A or B, what type of signal we're probing (differential, half the differential pair, or the logic signal between the controller and the bus driver), and then set the thresholds and the bit rate. Unlike other serial standards supported on the MSO/DPO4000 Series, FlexRay requires two thresholds to be set when looking at non-Tx/Rx signals as it is a three-level bus. This enables the oscilloscope to recognize Data High and Data Low as well as the idle state where both signals are at the same voltage.

The MSO/DPO4000 Series' powerful FlexRay feature set is illustrated in Figure 26 where we've triggered on a combination of Frame ID = 4 and Cycle Count = 0, captured approximately 80 FlexRay frames, decoded the whole acquisition and then had the oscilloscope search through the acquisition to find and mark all occurrences of sync frames. And all of this was done with only 100,000 point record lengths. The MSO/DPO4000 Series can go as deep as 10 million points on all channels to capture long time windows of serial activity.

The MSO/DPO4000 Series FlexRay triggering capability includes the following types:

- Start of Frame – triggers on the trailing edge of the Frame Start Sequence (FSS).
- Indicator Bits – trigger on Normal, Payload, Null, Sync, or Startup frames.
- Identifier – trigger on specific Frame IDs or a range of Frame IDs.
- Cycle Count – trigger on specific Cycle Count values or a range of Cycle Count values.
- Header Fields – trigger on a combination of user specified values in any or all of the header fields including the Indicator Bits, Frame ID, Payload Length, Header CRC, and Cycle Count.

Figure 27. Eye Diagram analysis of a FlexRay signal.

- Data – trigger on up to 16 bytes of data. Data window can be offset by a user specified number of bytes in a frame with a very long data payload. Desired data can be specified as a specific value or a range of values.
- Identifier & Data – trigger on a combination of Frame ID and data.
- End of Frame – trigger on static frames, dynamic frames, or all frames.
- Error – trigger on a number of different error types including Header CRC errors, Trailer CRC errors, Null frame errors, Sync frame errors, and Startup frame errors.

In addition to the triggering and decode features described above, DPO4AUTOMAX also provides eye diagram analysis of FlexRay signals to assist in diagnosing physical layer issues. Simply load the software package on a PC, connect it to the scope via LAN or USB, and click the Acquire Data button to get the information rich display shown in Figure 27. Analysis features include:

- Eye Diagram – built from all messages in the acquisition with the currently selected frame highlighted in blue. Easily compare against TP1 or TP4 masks with violations highlighted in red.
- Decode – currently selected frame is decoded over the analog waveform while the whole acquisition is decoded in the table below.

- Time Interval Error (TIE) Plot – provides for easy visual investigation of jitter within frames.
- Error Checking – errors are highlighted in red. Header and trailer CRCs are calculated and compared with transmitted frame.
- Timing Measurements – rise time, fall time, TSS duration, frame time, average bit time, previous sync, next sync, previous cycle frame, next cycle frame.
- Find – isolate the particular frame of interest based on packet content.
- Save – save decoded acquisition to a .csv file for further offline analysis.

This comprehensive set of FlexRay solutions, along with the previously discussed CAN and LIN capabilities, make theMSO/DPO4000 Series the ultimate debugging tool for automotive designs.

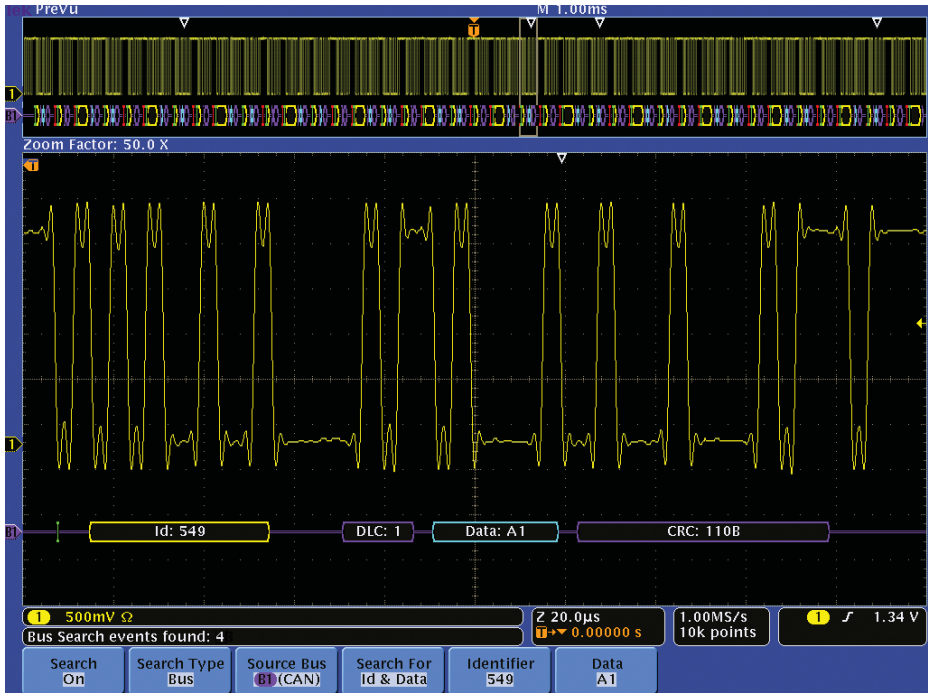


Figure 28. Searching on specified identifier and Data in a CAN bus aquisition.

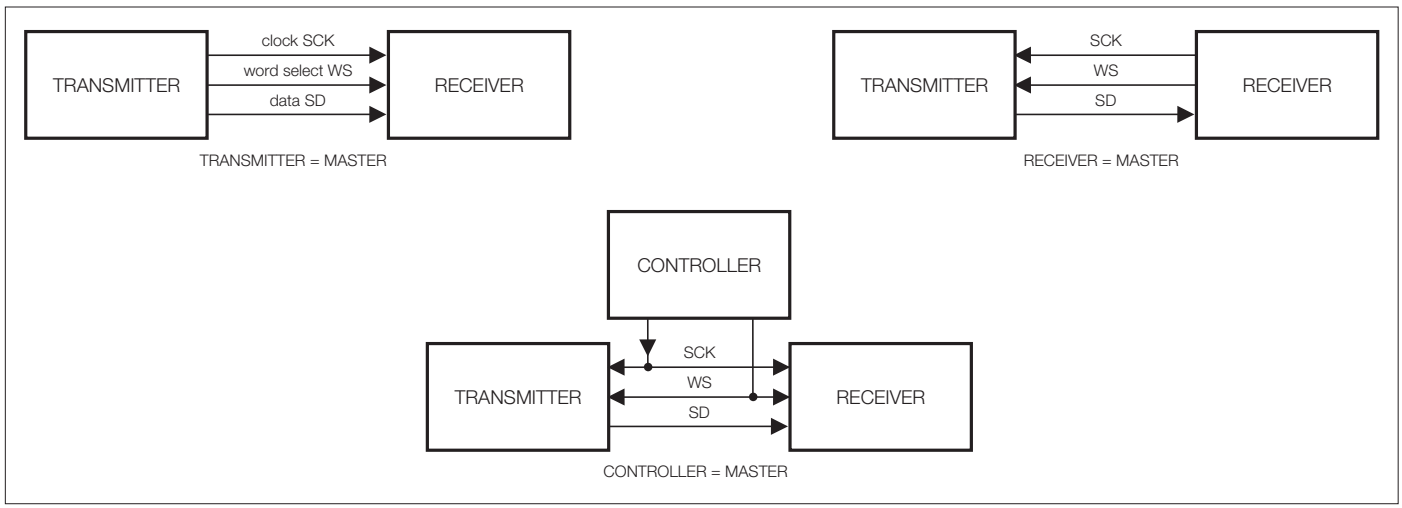


Figure 29. Different I²S bus configurations.

Audio Buses

Background

I²S, or "I squared S", stands for Inter-IC Sound. It was originally developed by Philips in the mid-1980s to provide a standardized communication path for digital audio signals in consumer electronic devices such as CD players and digital televisions. The consumer electronics market has continued to evolve over the last 20 years and so have the applications for the I²S bus. Today it's commonly found in cell phones, MP3 players, set top boxes, professional audio equipment and gaming systems to name a few.

How It Works

The I²S bus is a master/slave 3-wire serial communications bus. The three signals are clock (SCK), word select (WS), and data (SD). Typically, the transmitter is the Master and the receiver is the Slave. However, in some cases, the receiver can act as the Master by generating the clock and the word select signals. Or the transmitter and the receiver can be controlled by another device if desired. These configuration scenarios are illustrated in Figure 29.

Serial data is transmitted in two's complement with the most significant bit (MSB) first. The MSB is transmitted first because the transmitter and receiver may have different word lengths. It isn't necessary for the transmitter to know how many bits the receiver can handle, nor does the receiver need to know how many bits are being transmitted. When the system word length is greater than the transmitter word length, the word is truncated (least significant data bits are set to '0') for data transmission. If the receiver is sent more bits than its word length, the bits after the least significant bit (LSB) are ignored. On the other hand, if the receiver is sent fewer bits than its word length, the missing bits are set to zero internally. And so, the MSB has a fixed position, whereas the position of the LSB depends on the word length. The transmitter always sends the MSB of the next word one clock period after the WS changes.

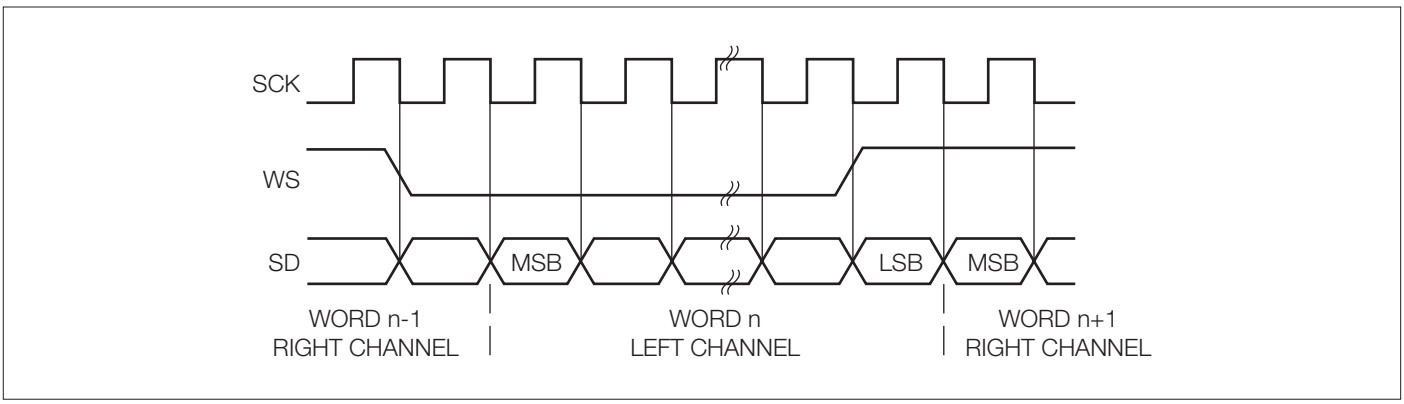


Figure 30 part I. I²S Format.

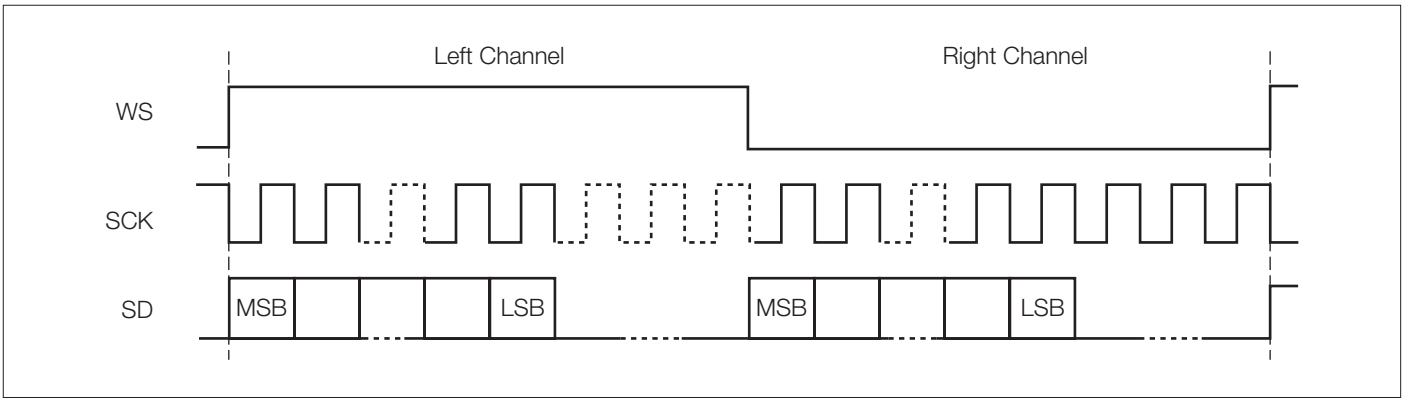


Figure 30 part II. LJ Format.

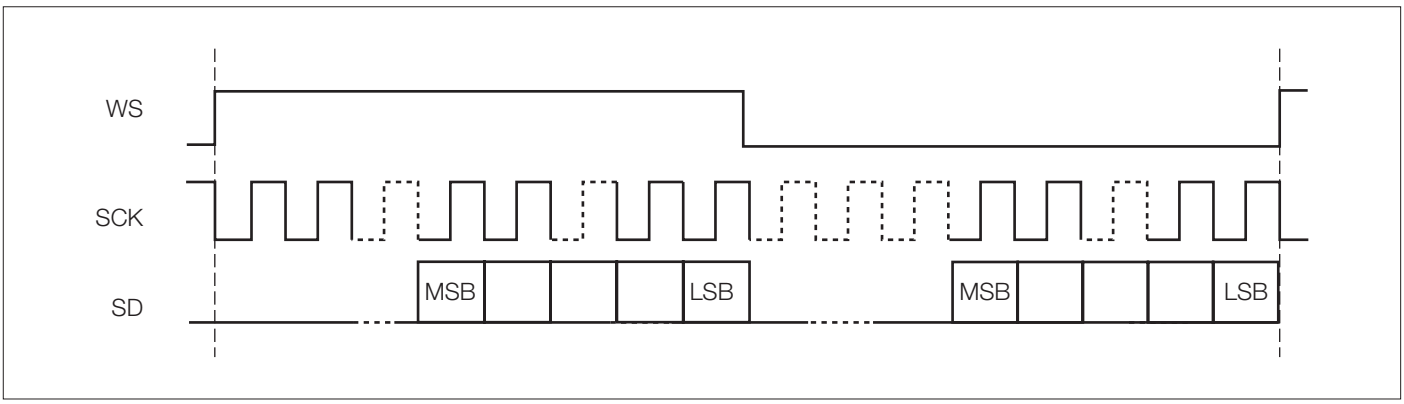


Figure 30 part III. RJ Format.

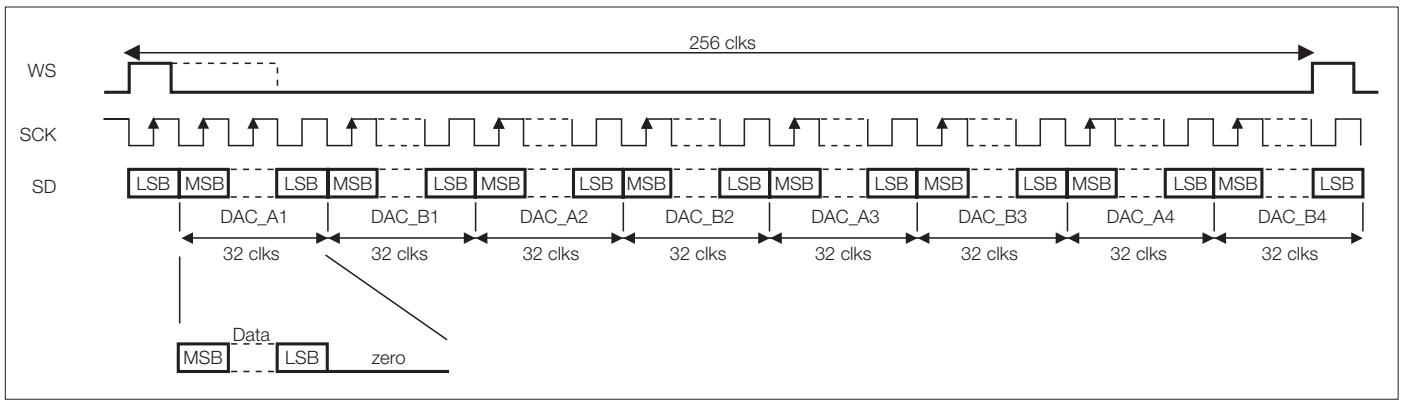


Figure 31. TDM Format.

There are several variants of the I²S bus that are also commonly used called Left Justified (LJ), Right Justified (RJ), and Time Division Multiplexing (TDM). The major difference between I²S, LJ, and RJ is where the data is placed in time relative to the Word Select signal. With I²S, the MSB is delayed one clock after WS. With LJ, the data bits are aligned with WS and with RJ, the data bits are right aligned with WS. These are all illustrated in Figure 30 (NXP Online)*.

TDM is similar to I²S, LJ, and RJ, but allows for more than two audio channels. The example shown in Figure 31 has eight audio channels, each with 32 data bits.

All of these digital audio buses have a very simple data structure. Many of the other buses we've looked at in this application note have address fields, CRC fields, parity bits, start/stop bits, and various other indicator bits, but the digital audio buses simply have data values for each channel.

* *NXP Online*. I²S Bus Specification. June 1996. December 2008. http://www.nxp.com/acrobat_download/various/I2SBUS.pdf

Figure 32. Triggering outside a range of values on an I²S bus.

Working with Audio Buses

Support for digital audio buses is available via the DPOxAUDIO serial triggering and analysis application modules. Using a front panel Bus button, we can define an audio bus by simply entering basic bus parameters such as word size, signal polarities, bit order, and thresholds. TDM definition also requires the number of data bits per channel, clock bits per channel, bit delay, and the number of channels per frame.

Once the bus is setup, you can quickly trigger on specific data content on the bus, decode entire acquisitions and search through acquisitions to find the specific data you're looking for. In the following example, we're looking at an I²S bus being driven by an analog to digital converter (ADC). Channel 1 (yellow) is the clock signal, channel 2 (blue) is the word select signal, and channel 3 (pink) is the data signal. We've set the trigger to look for data values outside a specified range to see if the signal we're sampling is hitting the limits of the ADC. As Figure 32 shows, we did capture an extreme value (-128) with this Outside Range trigger.

The MSO/DPO Series' powerful audio triggering capability includes the
following types:

- Word Select – trigger on the edge of Word Select that starts the frame in I²S, LJ, and RJ buses
- Frame Sync – trigger on the Frame Sync signal that starts a frame in TDM
- Data – trigger on user specified data in the Left Word, Right Word, or Either Word in I²S, LJ, and RJ. With TDM, you specify the channel number to look for the data value in. Data qualifiers include =, ≠, ≤, <, >, ≥, inside range, and outside range.

As with all the other serial bus types supported by the MSO/DPO Series, these trigger criteria are also available as search criteria for investigating long acquisitions and the decoded audio data can be presented in event table format.

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 MSO/DPO 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 28 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 MSO/DPO Series changes everything. With its powerful trigger, decode, and search capabilities today's design engineers can solve embedded system design issues with exceptional efficiency.

The MSO/DPO Series offers a range of models to meet your needs and your budget:

	MSO/DPO4000 Series	DPO3000 Series	MSO/DPO2000 Series
Bandwidth	1 GHz, 500 MHz, 350 MHz	500 MHz, 300 MHz, 100 MHz	200 MHz, 100 MHz
Channels	2 or 4 analog, 16 digital (MSO Series)	2 or 4 analog	2 or 4 analog, 16 digital (MSO Series)
Record Length (All Channels)	10 M	5 M	1 M
Sample Rate (Analog)	5 GS/s*, 2.5 GS/s	2.5 GS/s	1 GS/s
Color Display	10.4 in. XGA	9 in. WVGA	7 in. WQVGA
Serial Bus Triggering and Analysis Application Modules	DPO4EMBD: I^2C, SPI DPO4COMP: RS-232/422/485/UART DPO4AUTO: CAN, LIN DPO4AUTOMAX: CAN, LIN, FlexRay DPO4AUDIO: I^2S/LJ/RJ/TDM	DPO3EMBD: I^2C, SPI DPO3COMP: RS-232/422/485/UART DPO3AUTO: CAN, LIN DPO3AUDIO: I^2S/LJ/RJ/TDM	DPO2EMBD: I^2C, SPI DPO2COMP: RS-232/422/485/UART DPO2AUTO: CAN, LIN
Number of Simultaneously Displayed Serial Buses	4	2	2

* 1 GHz bandwidth models.

Contact Tektronix:

ASEAN / Australasia (65) 6356 3900
Austria +41 52 675 3777
Balkans, Israel, South Africa and other ISE Countries +41 52 675 3777
Belgium 07 81 60166
Brazil +55 (11) 40669400
Canada 1 (800) 661-5625
Central East Europe, Ukraine and the Baltics +41 52 675 3777
Central Europe & Greece +41 52 675 3777
Denmark +45 80 88 1401
Finland +41 52 675 3777
France +33 (0) 1 69 86 81 81
Germany +49 (221) 94 77 400
Hong Kong (852) 2585-6688
India (91) 80-42922600
Italy +39 (02) 25086 1
Japan 81 (3) 6714-3010
Luxembourg +44 (0) 1344 392400
Mexico, Central/South America & Caribbean 52 (55) 54247900
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) 6917-5000
Russia & CIS +7 (495) 7484900
South Africa +27 11 206 8360
Spain (+34) 901 988 054
Sweden 020 08 80371
Switzerland +41 52 675 3777
Taiwan 886 (2) 2722-9622
United Kingdom & Ireland +44 (0) 1344 392400
USA 1 (800) 426-2200
For other areas contact Tektronix, Inc. at: 1 (503) 627-7111

Updated 30 October 2008

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**

Copyright © 2009, Tektronix. All rights reserved. Tektronix products are covered by U.S. and foreign patents, issued and pending. Information in this publication supersedes that in all previously published material. Specification and price change privileges reserved. TEKTRONIX and TEK are registered trademarks of Tektronix, Inc. All other trade names referenced are the service marks, trademarks or registered trademarks of their respective companies.

3/09 JS/WWW 48W-19040-5_edu

Tektronix®