Debugging a Boundary-Scan I²C Script Test with the BusPro - I and I2C Exerciser Software: A Case Study

Overview

When developing and debugging I²C based hardware and software, it is extremely helpful to see exactly what is happening on the I²C bus. This article describes one example where the **BusPro - I** hardware and **I2C Exerciser** software were used to debug an I²C interface. The article presents a specific example of generating a boundary-scan script test in **ScanExpress TPG** to drive the I²C bus signals via JTAG. Although this article describes a specific problem that occurred in a single hardware development environment, the same general techniques described will also apply to other applications as well.

The article begins with a brief description of the hardware environment and a general background that will help to clarify the setup and usage details. The problem will then be presented followed by an explanation of the debug effort which ultimately allows the problem to be quickly identified and fixed.

Hardware Background

One step in the production of Corelis boundary-scan controllers is a structural boundary-scan test to catch any defects in the board level interconnects after assembly. This test detects and isolates any opens or shorts in the solder connections between the components and the PCB. Employing boundary-scan testing helps to ensure that Corelis ships the highest quality hardware products. It also saves time and cuts production costs by being able to program the contents of any in-system programmable parts within the same test setup. In addition to the improved board reliability due to the additional fault coverage, the structural testing also helps to increase efficiency by automating some of the test steps and associated cabling changes that would normally be required during the various production test stages.

Corelis boundary-scan controllers support a variety of TAP voltages. The TAP voltages are controllable through software and are physically generated by an on-board I²C digital to analog converter (DAC). For this application, DAC part number AD5301 from Analog Devices is used. The AD5301 is a single output 8-bit, buffered, voltage-output DAC that is operated from a 3.3 V supply. It uses a 2-wire (I²C-compatible) serial interface that operates at clock rates up to 400 kHz. These parts incorporate a power-on reset circuit, which ensures that the DAC output powers up to 0 V and remains there until a valid write takes place. The DAC's default power up value of 0 V creates a little bit of complication in setting up the hardware for testing.

This particular board has two scan chains. The first boundary-scan chain is used to set up the board for testing by shifting bits through TAP 1 with a scripted test step that controls the I²C bus and configures the DAC. The second chain contains the majority of the boundary-scan components. Both chains need to be serialized during the structural test for maximum interconnectivity test coverage. The variable voltage on the second JTAG chain must be set to 3.3V during the production test for proper operation. Figure 1 shows a simplified block diagram of the general hardware setup. The TAP connectors and JTAG chains are shown to provide a general overview of what is involved in the boundary-scan setup for the production test and why it is important to be able to set the DAC.

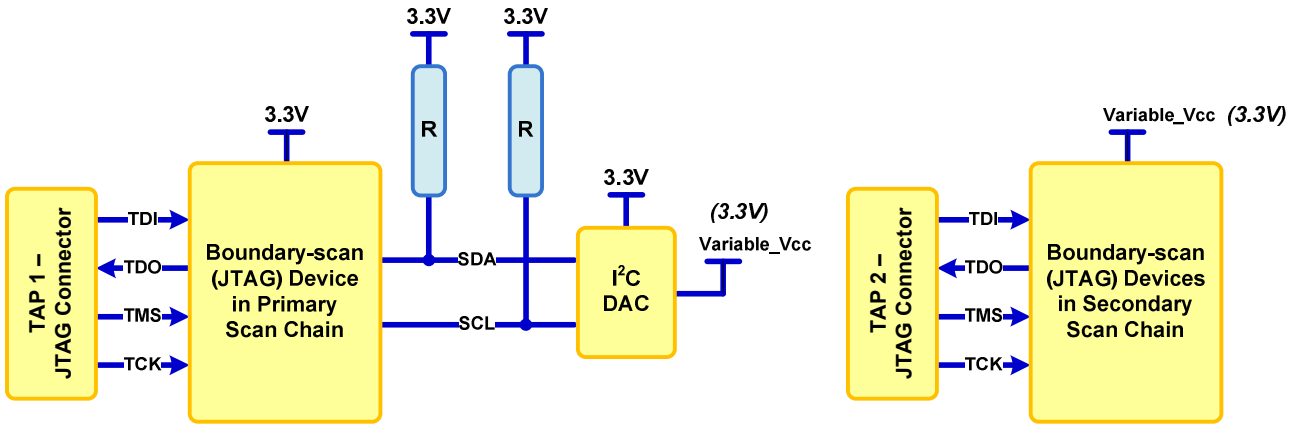

Figure 1. Simplified Hardware Block Diagram

To summarize, **ScanExpress TPG** executes a test script to toggle pins of the boundary-scan device via its JTAG interface. Proper I²C protocol is simulated on the relevant SDA and SCL pins through JTAG scanning operations to set the DAC voltage. This properly sets the voltage to TAP2 for interconnectivity testing.

During development of the test script in **ScanExpress TPG** to set the DAC voltage, a bug was introduced during script writing. Although the test script appeared to be executing fine, the voltages on the DAC output did not appear as expected.

I2C Exerciser Debugger

The **I2C Exerciser** software features a **Debugger** module that allows the user to perform reads and writes on the I²C bus. This module was used to perform a quick sanity check to make sure that the hardware itself was working correctly and that there was nothing wrong with the physical components or board assembly. The **Debugger** allowed a sequence to write test values to the DAC and read the data back from the DAC. Placing a multi-meter on the DAC output during this sequence allowed verification that the proper voltage appeared on the DAC output.

DAC (AD5301) Write Sequence

Writes to the AD5301 begin with the address byte, after which the DAC acknowledges that it is prepared to receive data by pulling SDA low (ACK). This address byte is followed by the 16-bit command word in the form of two control bytes. The write operation for the DAC is shown in Figure 2.

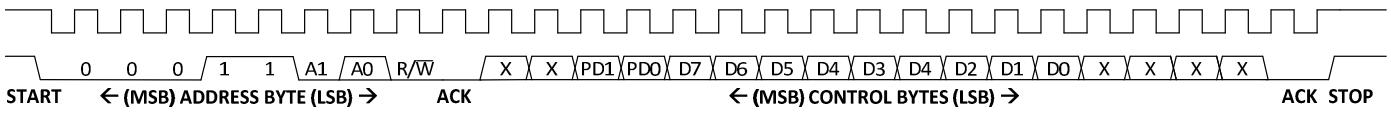

Figure 2. DAC Write Sequence (AD5301)

Having to debug a waveform bit by bit can be tedious work. Fortunately, the **I2C Exerciser** software automatically handles all the I²C address and data sequencing to conform to the protocol and provides a convenient interface to peek and poke values to device registers on the I²C bus. It even sets the Read/Write bit automatically. The **I2C Exerciser** software transparently takes care of all the work to interface with the I²C bus behind the scenes.

AD5301 Address

The AD5301 has a 7-bit slave address. The five MSBs are 00011 and the two LSBs are determined by the state of the A0 and A1 pins that allow up to four of these DACs to be used on a single I²C bus. The default address was used on the Corelis controllers so pins A1 and A0 were grounded.

When A1 and A0 are set to 0, the resulting bit sequence is "**00001100**". This translates to an I²C address of **0x18** when left justified due to the read/write bit. The **I2C Exerciser** software also supports right justified I²C addresses for users that are more comfortable viewing in this format. The address format setting can be changed in the Preferences menu to use the equivalent right justified address value **0x0C**. This setting globally adjusts the address format wherever it is entered or displayed.

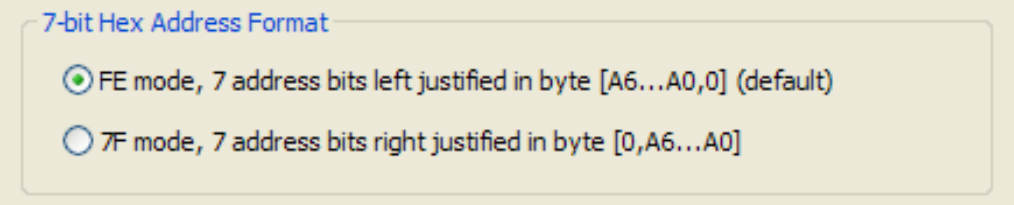

Figure 3. I2C Exerciser Address Format Selection

AD5301 Command Word

The AS5301 command word (16-bits) sets the DAC's operating mode and controls the output voltage on the V_{OUT} pin.

The architecture of the DAC channel consists of a resistor string DAC followed by an output buffer amplifier. The voltage at the VDD pin provides the reference voltage for the DAC. Figure 4 shows a simplified block diagram of the DAC's internal architecture. Since the input coding to the DAC is straight binary, the ideal output voltage is given by the equation:

$$V_{OUT} = \frac{V_{DD} \times D}{2^N}$$

Where:

D = decimal equivalent of the binary code that is loaded into the DAC register (0-255 for AD5301)

N = DAC resolution (8 bits for AD5301)

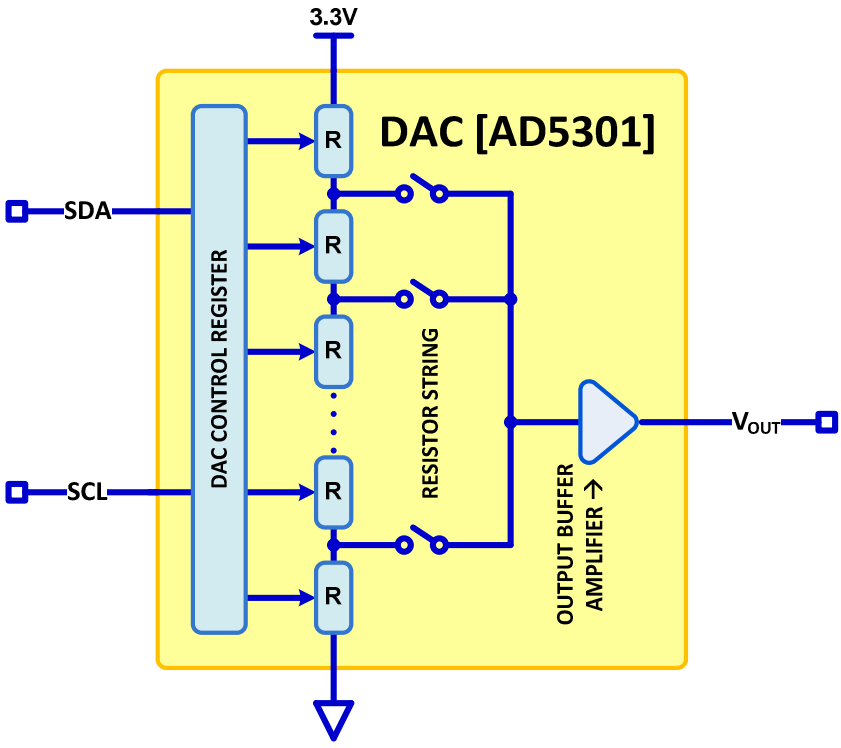

Figure 4. DAC's Internal Architecture (AD5301)

To calculate the binary code required to set the DAC to output 3.3 V, the appropriate values are plugged into the equation to solve for the binary code D. This gives us a result of D = 255 or 0xFF hex.

$$V_{OUT} = 3.3\ V = \frac{3.3\ V \times D}{2^8}$$

Setting the DAC register bits D7:D0 to 0xFF generates the desired output voltage of 3.3 V. For normal operation, the PD1 and PD0 bits should be set to 0. Don't care bits (X) in the control byte should also be set to 0. Thus, the final 16 bit command opcode will be **0x0FF0**.

To send the 16-bit data value **0x0FF0** to I²C address **0x18**, the address and data values are entered in the appropriate fields and the "**Send**" button is used to instruct the **BusPro – I** to communicate the information. This is depicted in Figure 5 below. If successful, a "Passed" status is indicated and the data values are echoed in the text field located at the right side of the dialog. After executing this sequence, the voltage on the DAC output pin reads 3.3 V with a multi-meter as expected.

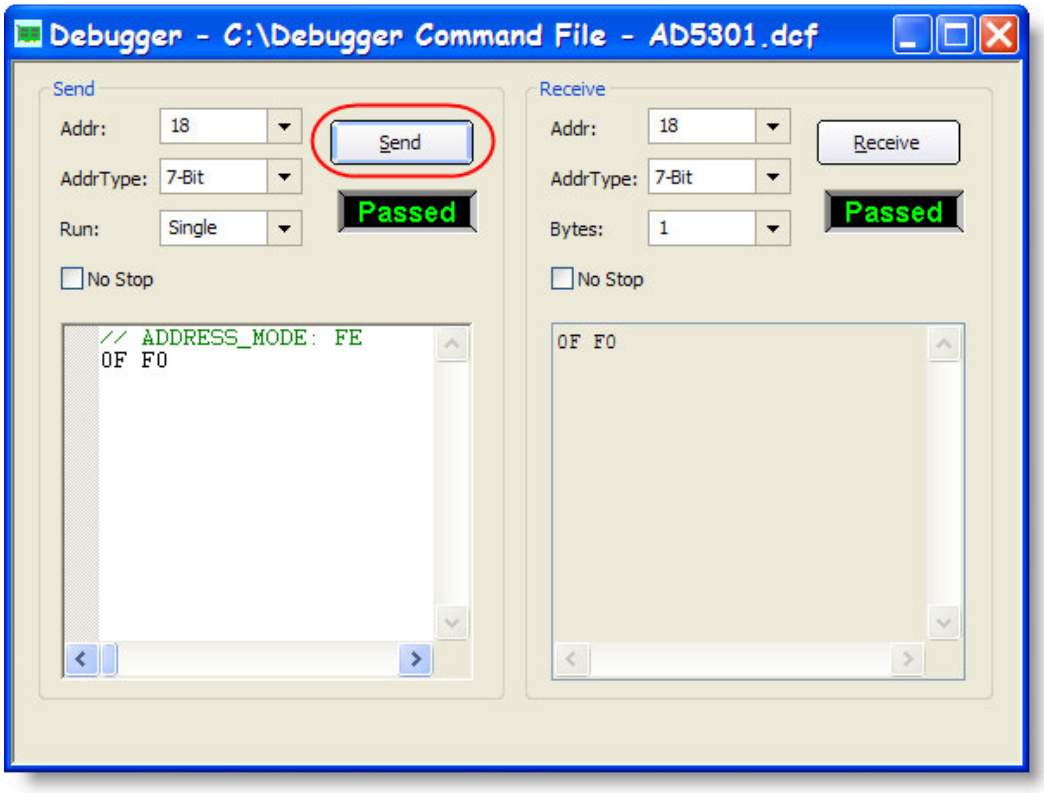

Figure 5. Write to the DAC with the I2C Exerciser Debugger Module

The **Debugger** module supports both 7-bit and 10-bit I²C addressing schemes and the data values are easily sent multiple times by changing the **Run:** dropdown box from **Single** to **Continuous** or entering any arbitrary integer.

Several more values were used to verify that the voltage was being set correctly in each case. The next step was to readback the data register in the DAC to make sure the I²C interface was completely operational.

DAC (AD5301) Readback Sequence

When reading data back from the AD5301 DAC, the transaction must begin with an address byte after which the DAC acknowledges that it is prepared to transmit data by pulling SDA low (ACK). The readback value is a single byte that consists of the eight data bits in the DAC register. The read operation for the DAC is shown in Figure 6.

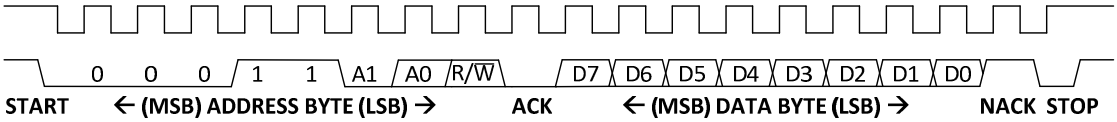

Figure 6. DAC Readback Sequence (AD5301)

Similar to the write sequence, the **Debugger** module in the **I2C Exerciser** software automatically handles all the I²C address and data protocols for the read sequence.

The "**Receive**" button in the **Debugger** interface is used to readback data. A simple test is to write the values **0x05A0** and readback the expected data **0x5A**. Figure 7 shows the **Debugger** module in the **I2C Exerciser** software being used to readback the data value of **0x5A**.

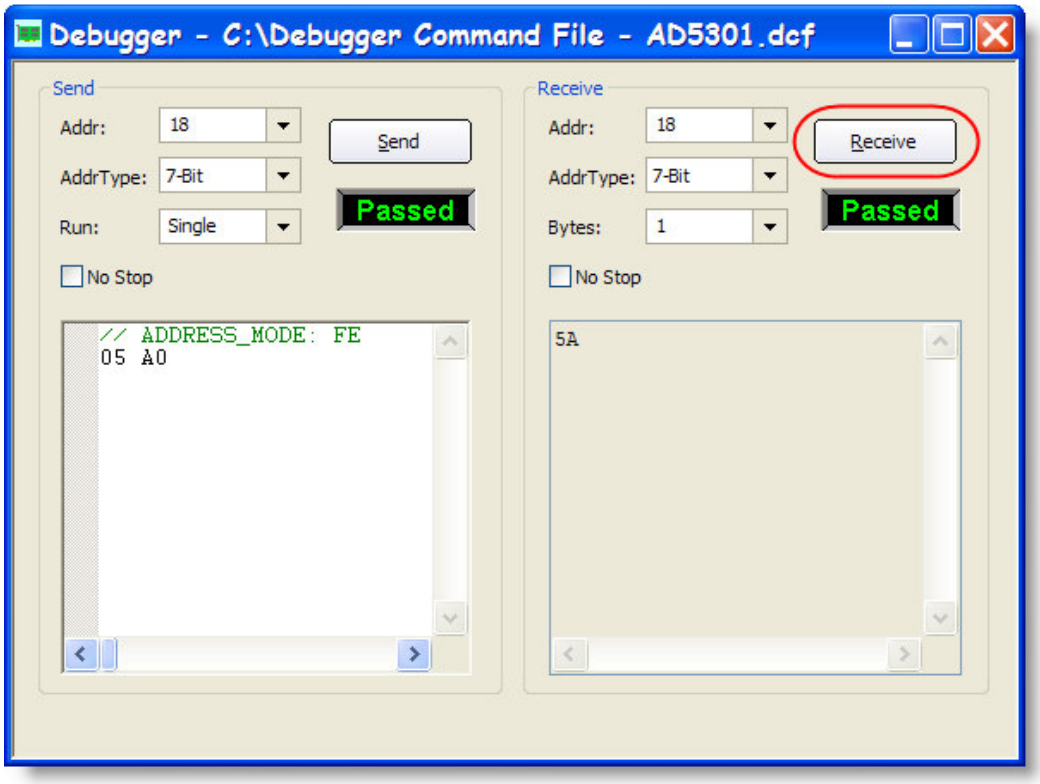

Figure 7. Reading the DAC through the I2C Exerciser Debugger Module

Going through this process confirmed that the serial interface to the DAC was indeed functioning correctly. If a problem had come up while reading or writing values in the DAC data register, the **Debugger** would have clearly shown the incorrect data. Problems at this level can be caused by a variety of reasons such as no power, missing pull-up resistors, incorrect I²C address, a damaged device, etc. The **BusPro – I** diagnostics helps narrow down the possibilities.

I2C Exerciser Monitor

The previous results provided confidence that the prototype hardware was behaving as expected. This helped narrow the problem to the test script. The built-in **I2C Exerciser Monitor** function is the perfect tool to provide easy trace capability for the test script. Using the **Monitor** is as simple as clicking the "run" arrow (▶) and the **I2C Exerciser** software immediately begins to collect the traffic on the I²C bus. Figure 8 depicts a trace of the test script which captures a problem in the I²C protocol implementation.

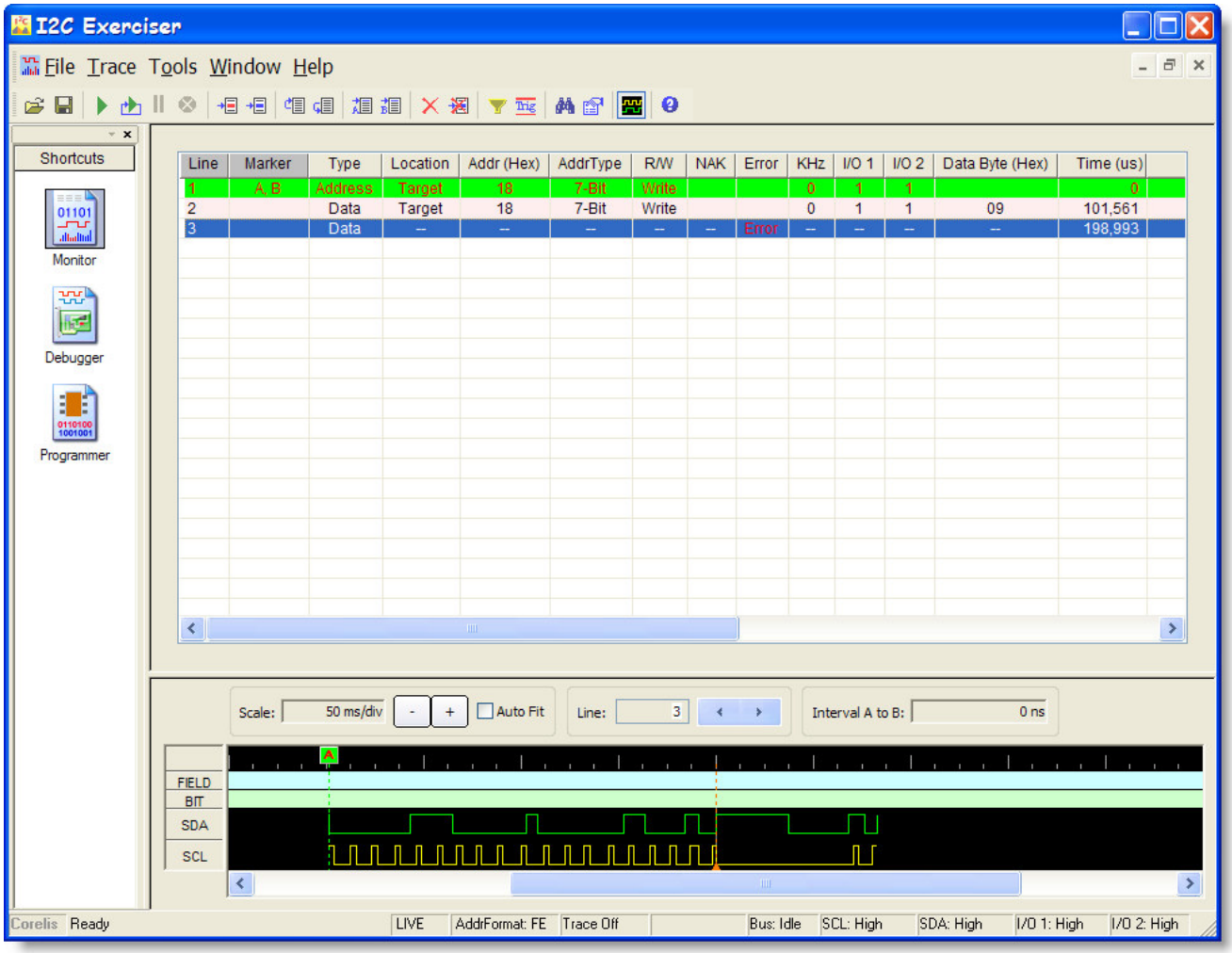

Figure 8. I2C Exerciser Monitor Window

The **I2C Exerciser Monitor** indicates that an I²C protocol error has occurred and shows the context around where the problem occurred. Additionally, the SDA/SCL bit level timing information is displayed as a graphical waveform at the bottom of the screen. This timing diagram clearly identifies the problem (see Figure 9).

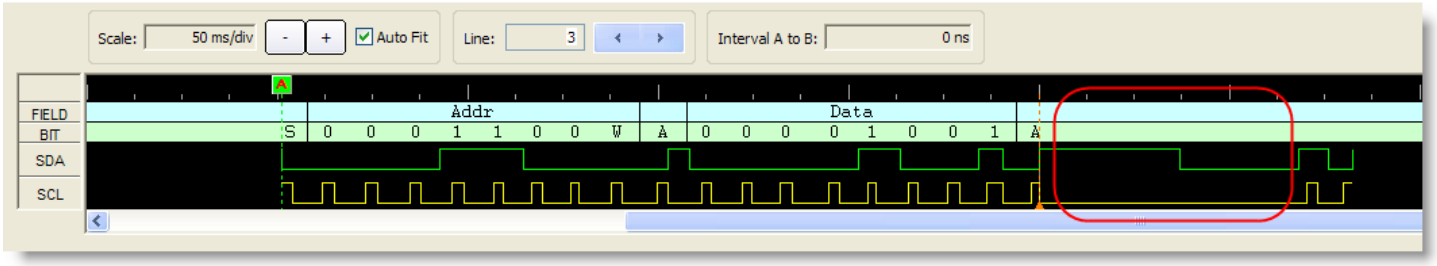

Figure 9. I2C Exerciser Timing Waveform (Bad)

The script wasn't working correctly because the SCL signal was not being allowed to return high to generate the rising edge of the clock. The trace listing and graphical waveform display immediately helped pinpoint exactly where the issue was occurring. Understanding where the failure was occurring allowed quick location within the test script for the offending code. Because this problem was logical and not electrical it would have been difficult to capture it with an oscilloscope. Figure 10 shows the section of the test script that is the root cause of the problem.

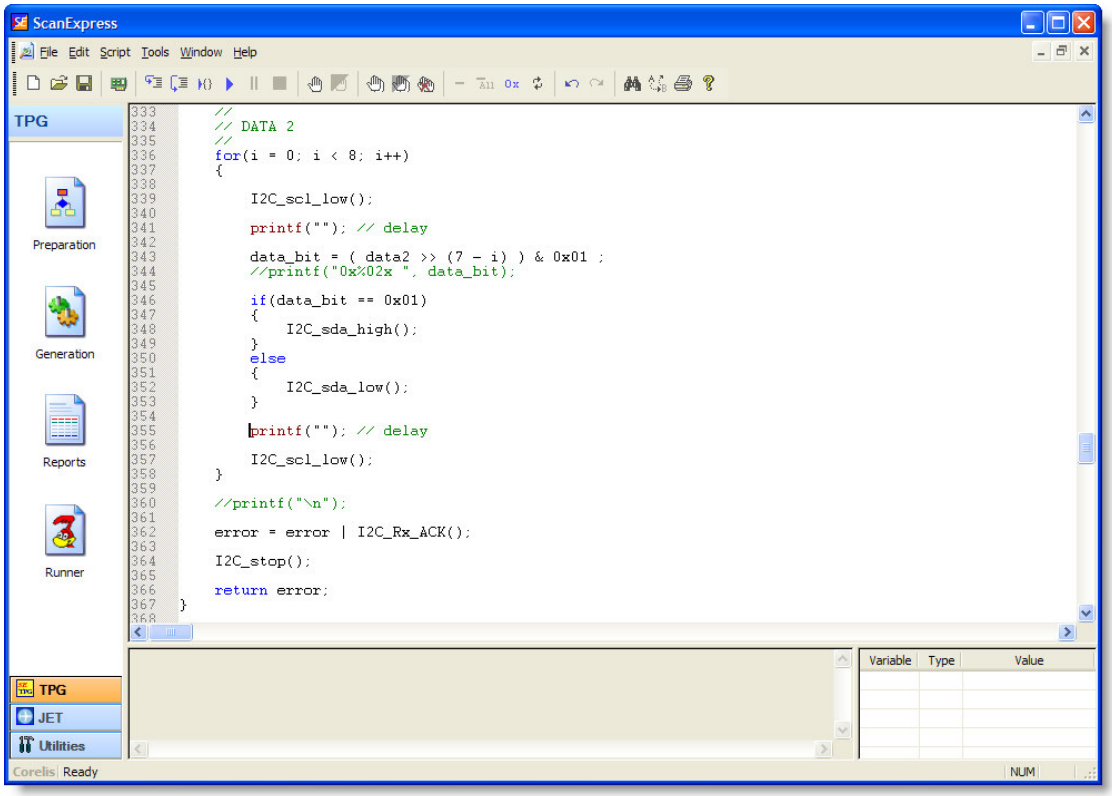

Figure 10. Problematic I²C Script in ScanExpress TPG

Once the cause was identified, the fix was straightforward. Adding a line of code to make the SCL signal go high at the appropriate time and generate the rising edge of the clock is all that was necessary.

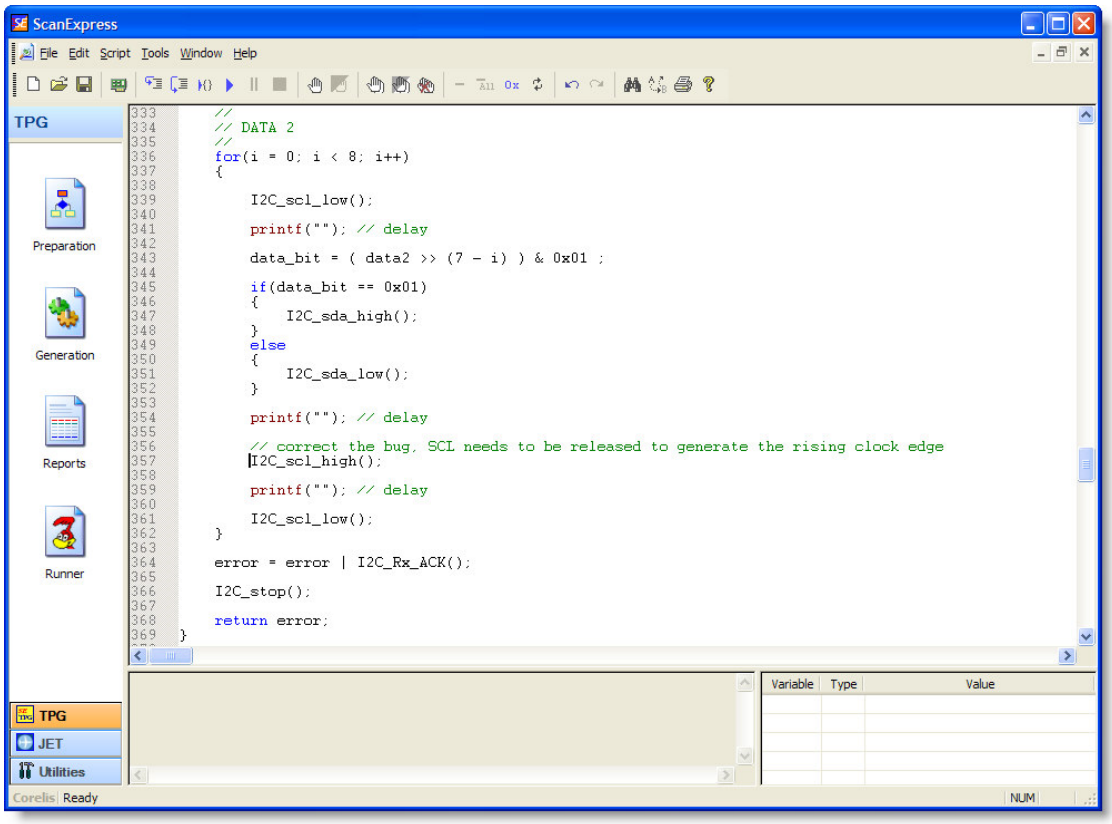

Figure 11. Corrected I²C Script in ScanExpress TPG

Figure 11 shows the corrected script. Running this new script and capturing the resulting trace provided a correct timing waveform as shown in Figure 12.

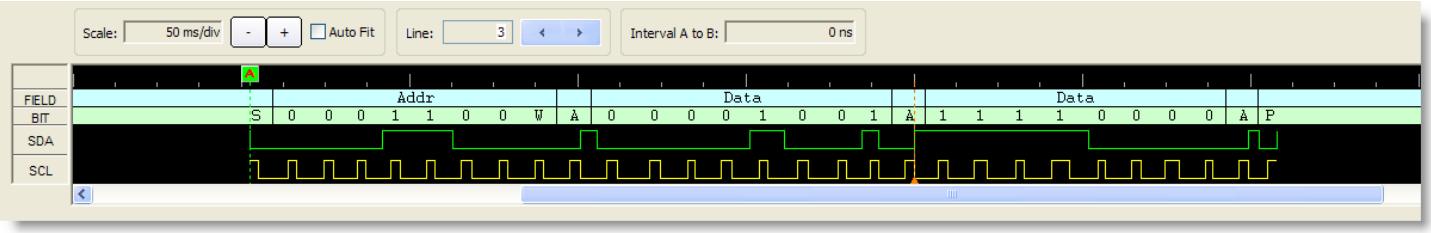

Figure 12. I2C Exerciser Timing Waveform (Good)

Conclusion

The type of problem presented in this article is very cumbersome to identify using traditional test equipment. For example, it may take well over an hour to correctly connect a logic analyzer and search through the trace to find the exact place where the I^2C protocol violation was occurring. An oscilloscope may not have even helped find the problem at all.

With the right equipment, debug efforts can be reduced from hours to minutes. The Corelis **BusPro - I** hardware and **I2C Exerciser** software are able to pinpoint the exact place problems occur in I^2C transactions. They represent the right tools for the job.

This article shows just a fraction of the capabilities of the **BusPro - I** and **I2C Exerciser** software. Corelis now provides a set of I^2C script functions in a standard test library to accelerate your own hardware and software debugging processes.

The article also shows the scripting capability in **ScanExpress TPG** which provides a convenient method to accomplish a variety of testing tasks using boundary-scan. The **ScanExpress TPG** scripting language is a subset of the "C" programming language and makes it easy to control boundary-scan devices to communicate with external peripheral chips that would not normally be controllable. An assortment of example script files are available to perform operations like Flash programming, manipulating LEDs, and reading from a file.

The full **I2C Exerciser** project files and **ScanExpress TPG** I^2C script source files are available for download through the Corelis support download website. Visit Corelis at www.corelis.com.