Agilent E2960 PCI Express

User Guide

Agilent Technologies

Important Notice

Manual Part Number

E2960-91020

Revision

November 2004

Printed in Germany

Agilent Technologies
Herrenberger Straße 130
D-71034 Böblingen
Germany

Authors: t3 medien GmbH

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING / DANGER

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Content

About this Guide

This Guide provides you with the information you need to get started working with the Agilent test solution for PCI Express. It contains the following chapters:

- *"System Overview" on page 7*

  Briefly explains the main concepts behind the Agilent test solution for PCI Express.

- *"Running Your First Tests" on page 17*

  Provides some examples for using the Agilent test solution for PCI Express.

- *"Exerciser Architecture Overview" on page 61*

  Explains how the Exerciser is structured and gives an overview as to how it works as a requester and completer.

- *"How to Program the Exerciser and Analyzer" on page 69*

  Explains how the Agilent test solution for PCI Express can be used programmatically.

The instructions in this Guide assume a correctly running system (controller PC up and running, software correctly installed on the clients). See the *Getting Started and Installation Guide* for details.

Literature

General and detailed information on PCI Express can be obtained from Intel (*http://www.intel.com/technology*) and the PCI-SIG web site (*http://www.pcisig.com*).

Features and technical data of the Agilent PCI Express hardware and software are published in the data sheet "Agilent Technologies E2960 Series Exerciser and Protocol Analyzer for PCI Express", publication number 5988-8679EN.

Updated product information

For updated product information, please visit also *http://www.agilent.com/find/E2960_series*.

System Overview

PCI Express is designed to replace the PCI bus as the main I/O expansion bus in a wide range of systems. The multi-drop parallel bus approach of PCI is being replaced with a high-speed serial I/O bus that uses point-to-point signaling and a packetized protocol. The PCI Express architecture is scalable and extensible, and includes a rich set of features.

With the E2960 series, Agilent Technologies provides a family of Protocol Analyzers and Exercisers for PCI Express designed to support the increased need for analysis and validation of PCI Express chipsets and systems.

The Agilent test solution for PCI Express is a combined software/hardware solution.

The following sections provide you with an overview of the Agilent solution, its software components, and how they work together:

- *"Intended Use" on page 8*
- *"Software Components" on page 12*
- *"Session Concept" on page 14*

Intended Use

Background

Today’s key trends in the electronics industry aim for smaller and smaller chip geometries and CMOS technology being able to run at higher and higher frequencies. The pin count needed for parallel busses prevents the industry to take full advantage of these technology developments (pad versus gate limitations). Moving from parallel to serial busses will help the industry to focus on further integration in functionality and resulting cost reductions.

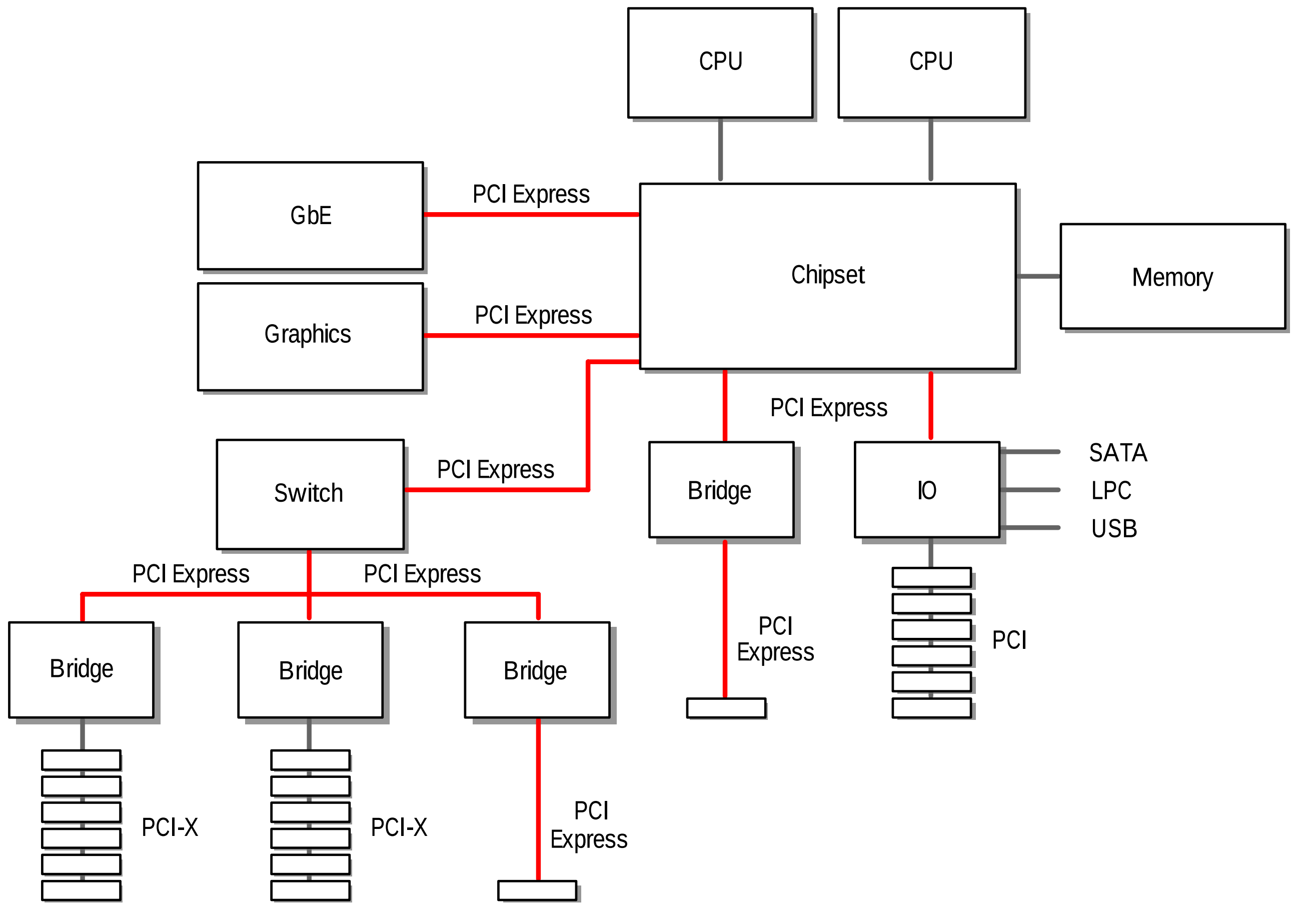


Figure 1 Typical PCI Express Server Architecture

PCI Express, a protocol introduced to the public in 2002, will change the way computer systems will be built in the near future. PCI Express operates at 2.5 GBit/s and uses two low voltage differential signals (LVDS) lines for transmitting and receiving data. Such a four-wire connection, called a PCI Express link x1 (pronounced: by one), is the foundation of PCI Express. Several of these links can be combined to provide higher bandwidth. The specification calls out for PCI Express x1, x2, x4, x8, x16 and x32. A by eight PCI Express link (x8) features a bandwidth of 40 Gbit/s.

The previous figure illustrates a two-way server architecture based on PCI Express. All key elements of the server (Chipset, Ethernet, Graphics core) are connected via PCI Express. Different link widths (x1, x4, x8, x16) are used to accompany the necessary bandwidth needs. Traditional IO standards like PCI or PCI-X are connected with bridges to PCI Express. Switches will be used to route PCI Express traffic in such an architecture.

Test requirements

The move from parallel busses to serial busses will result in different analysis and validation needs:

- Higher frequencies require enhanced test considerations on the physical layer.
- Point-to-point connections require enhanced probing solutions as well as different analyzer and stimulus (exerciser) tools.
- Complexer system components require different test methodology with increased focus on protocol testing.

The Agilent solution

With the E2960 series, Agilent Technologies provides a family of Protocol Analyzers and Exercisers for PCI Express designed to support this increased need for analysis and validation.

The E2960 series provides two types of testers:

- Protocol Exerciser for PCI Express (x1 to x8)
- Protocol Analyzer for PCI Express (x1 to x8)

Both tools are based on the same Serial I/O Module and probes, combined with unique software packages.

The Protocol Exerciser for PCI Express

The Protocol Exerciser for PCI Express is able to generate and respond to all types of PCI Express transactions. In addition, it allows you to create various PCI Express protocol variations and violations.

Another key feature is the ability to insert errors and test the behavior of designs in response to these errors. Errors can be generated and inserted on the physical, data link, and transaction layers.

The Protocol Exerciser for PCI Express is controlled by a graphical user interface, a C/C++ program or a Tcl script.

With these features and more, the Protocol Exerciser for PCI Express is perfectly suited for:

Validating PCI Express systems

You can use the Protocol Exerciser to validate a PCI Express system (*Upstream* testing).

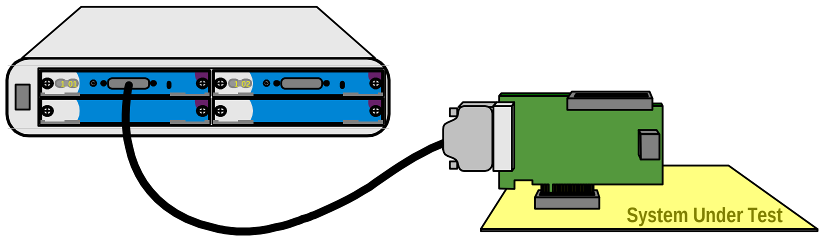


Figure 2 **Protocol Exerciser Setup for Testing a System (To Upstream)**

Bringing up and debugging PCI Express devices

You can use the Protocol Exerciser to bring up and debug a device (*Downstream* testing).

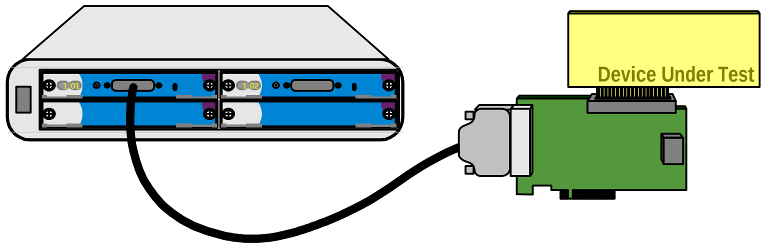


Figure 3 **Protocol Exerciser Setup for Testing a Device (To Downstream)**

The Protocol Exerciser functionality includes testing the physical and data link layer capabilities of a PCI Express device as well as simulating such devices on the transaction layer.

The Protocol Exerciser for PCI Express is also well suited for validating the inter-operability and stability of chipsets and systems.

The Protocol Analyzer for PCI Express

The Protocol Analyzer for PCI Express is used for capturing and analyzing the traffic between two PCI Express endpoints:

Listening to a system and add-in card

You can insert the Protocol Analyzer between a device and a system to capture the traffic.

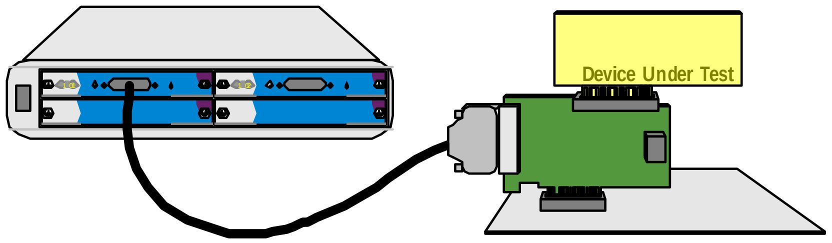


Figure 4 Protocol Analyzer Setup for Testing Communication to an Add-In Card

Listening to two chips on a motherboard

You can connect the Protocol Analyzer to a midbus footprint on the motherboard to listen to the traffic between two chips. See the data sheet for the E2941A Midbus Probe for details.

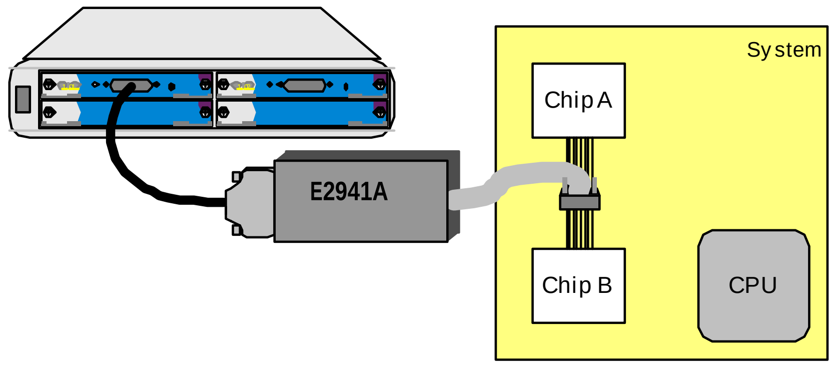


Figure 5 Protocol Analyzer Setup for Testing Communication between two Chips

The Protocol Analyzer captures traffic simultaneously from both directions, including training sequences, packets on the data link layer (DLLPs), and packets on the transaction layer (TLPs). It has 1 GB of trace memory.

The Protocol Analyzer for PCI Express includes a sophisticated trigger sequencer. Its graphical user interface allows you to filter the captured data and to view and investigate the packets in a variety of formats. It can also be controlled by a C/C++ program or a Tcl script.

With these features, the Protocol Analyzer for PCI Express is perfectly suited for testing and troubleshooting PCI Express links.

Software Components

The Agilent test solution for PCI Express contains several advanced software components that work together:

- *Firmware*

  Runs on the Serial I/O module

  Responsible for carrying out the actual tests

  Controlled by the control software running on the controller PC

- *Control software*

  Runs on the controller PC

  Responsible for setting up and maintaining communication with the I/O module, loading tests to the I/O module

  Controlled by the user software

- *User software*

  Runs on the client (local PC)

  Responsible for setting up tests

  Controlled by the user or programmer

  The user software can be either the Agilent Protocol Exerciser and Protocol Analyzer software, or a tcl script or DCOM-based test program.

The following figure illustrates the relationships between these software components:

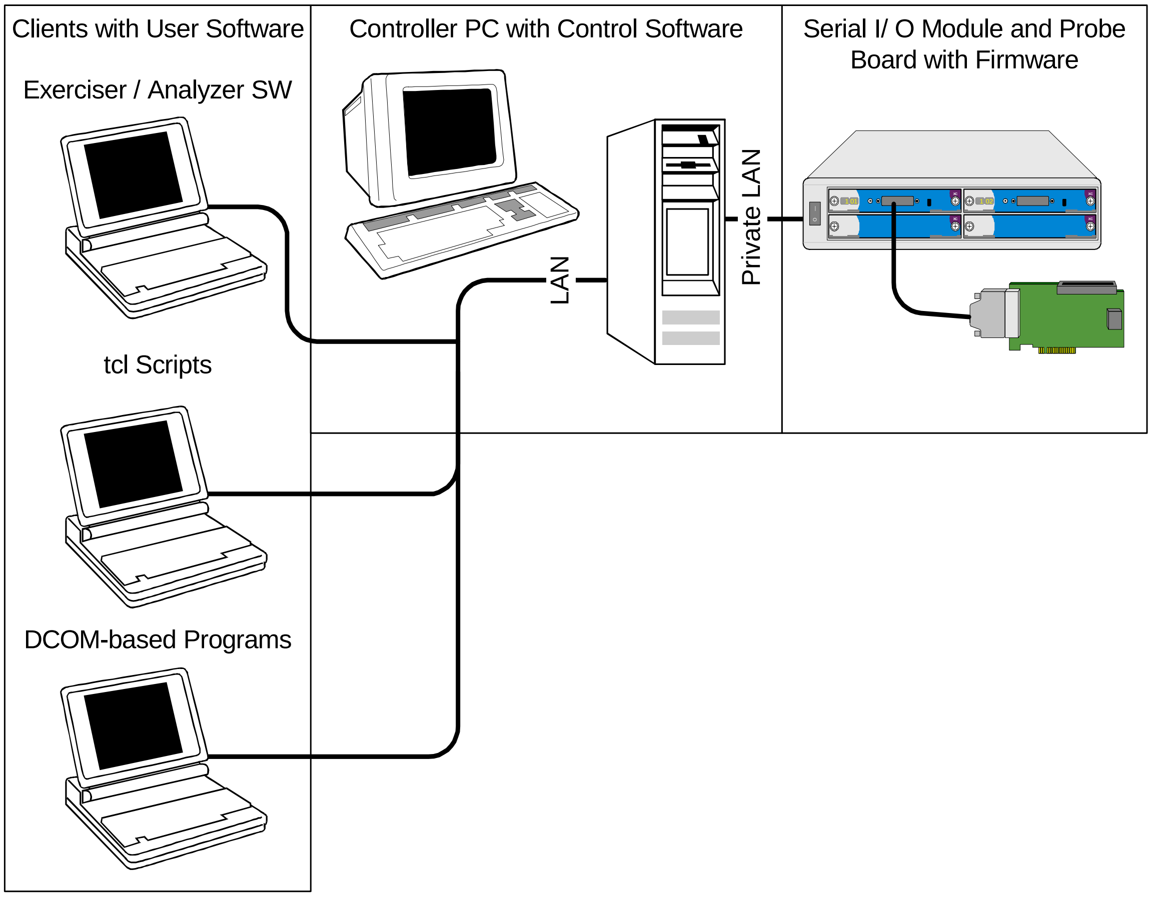

Clients with User Software
Exerciser / Analyzer SW
tcl Scripts
DCOM-based Programs
Controller PC with Control Software
LAN
Private LAN
Serial I/ O Module and Probe Board with Firmware

Figure 6 PCI Express Software Components

NOTE Note that the user software can also run on the controller PC. In this case, the controller PC can be seen as its own client.

Session Concept

The communication between the controller PC and the Serial I/O Modules is based on the concepts of *sessions*. A session is a representation of the instrument components involved in a test:

- The control software running on the controller PC
- The Serial I/O Module and the probe involved in the test

The following figure indicates the components involved in a session. It also indicates how more than one client can log onto one session.

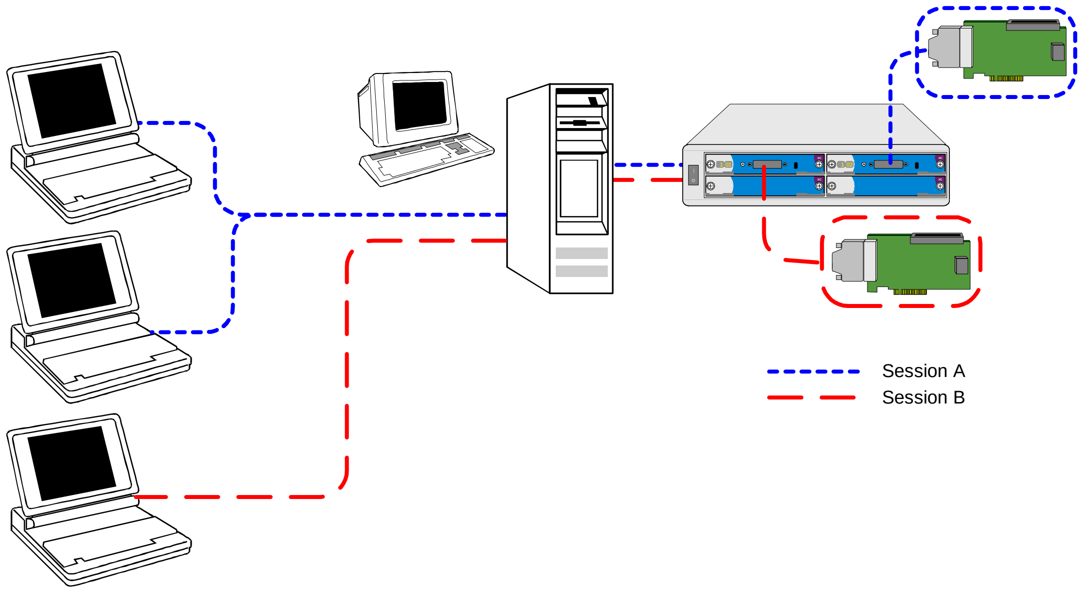


Figure 7 Session components

To use a session, it is necessary to:

- Start and configure the session

  This establishes communication between the client and the control PC and loads the necessary firmware onto the desired Serial I/O Module, which then assumes the personality required for the session (for example, Exerciser in upstream mode).

- End the session

  In the Exerciser and Analyzer software, when you quit using a session and you are the last one using the session, you are prompted as to whether the session should be ended. Ending a session clears the Serial I/O Module, allowing it to be used for a new session.

All accesses to the test system must go over the session. The session is not locked: concurrent access to one session is possible. This is described under *"Using a Session at Several Computers" on page 15*.

Starting and Configuring a Session

When the user software is started, it queries the control software running on the controller PC for a list of currently running sessions of the corresponding type (for example, a query by the Protocol Exerciser software returns a list of Exerciser-based sessions). The user software can either connect to one of these existing sessions, or request a new session.

If a new session is requested, the following occurs:

1. The user requests a session of a particular type (Exerciser or Analyzer).
2. The user adds a Serial I/O Module and its port (to the probe) to the session.
3. The Serial I/O Module downloads the necessary FPGAs and embedded software from the control PC.
4. The embedded software running on the Serial I/O Module configures the probe as necessary.

See *"Opening a New Session" on page 71* to learn how to start and configure a session in your own program.

Using a Session at Several Computers

If a session is up and running, you can use it from multiple instances of the user interface (for example, from different clients) or from tcl scripts or DCOM-based programs simultaneously. This could be necessary if you, for example, want to set up a test system directly at the controller PC and then run a series of tests from your PC.

The controller PC does not protect against meaningless or even conflicting requests. It is therefore recommended that only one user should "own" a particular session at a time.

To connect to a current session in the user interface, you only need to note the session number and then select this session when the software starts.

See *"Accessing a Running Session" on page 72* to learn how to connect to a running session in your own program.

Running Your First Tests

The following sections guide you step-by-step through your first tests with Agilent test solution for PCI Express:

- *"Testing the Communication to an Add-In Card" on page 18*

  This test demonstrates how you set up communication between a probe board to be used as an Exerciser and a PCI Express add-in card that you would like to test. With the information here, you should also be able to set up tests between a probe board and a PCI Express system.

- *"Analyzing the Communication to an Add-In Card" on page 30*

  This test shows you how you capture PCI Express communication between an add-in card and another probe board as Exerciser by using a probe board as Analyzer.

- *"Validating the System" on page 38*

  With this test, you see how you can analyze how a PCI Express system responds to requests with different behaviors and errors.

- *"Bringing Up and Debugging a PCI Express Add-In Card" on page 53*

  With this test, you see how you can verify the correct behavior of an add-in card during link training when the requester changes link settings.

Testing the Communication to an Add-In Card

For this test, we want to set up a PCI Express link between a probe board and an add-in card (for example, graphic card) and then test how the add-in card reacts to an errored packet.

System Setup

The following system setup is required for this test:

- The add-in card to be tested is plugged into the PCI Express slot at the top of the probe board.
- The Serial I/O Module to which the probe board is connected is set up as a Protocol Exerciser.
- An external ATX power supply is connected to the probe board. The power jumper on the probe board should be set to **Ext** (instead **sys**).

  The external power supply is required to power the add-in card. Alternatively, you could plug the probe board into a powered PCI Express slot and set the jumper to **sys**.

The system setup is as follows:

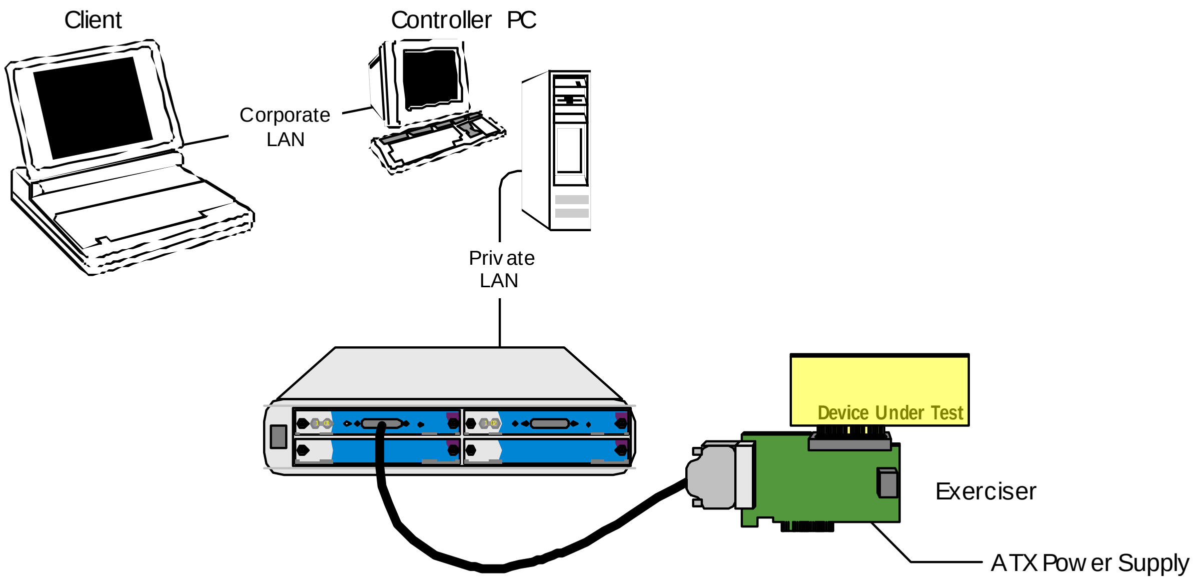


Figure 8 Protocol Exerciser Setup for Testing a PCI Express Add-In Card (To Downstream)

How to Proceed

This test is carried out as follows:

1. We establish an Exerciser session at the client. This establishes the necessary communication and sets up the Serial I/O Module (with probe) to be used as an Exerciser.

   See *“Establishing an Exerciser Session” on page 19*.

2. We set up and establish a PCI Express link between the Exerciser and the add-in card.

   See *“Establishing a PCI Express Link” on page 23*.

3. We set up the Exerciser to perform a data read on the add-in card. This is done by sending a data read request as a single packet to the add-in card.

   See *“Sending a Single Data Packet” on page 25*.

Once you have carried out the test, see *“What Happens at the Exerciser when sending a Single Packet?” on page 28* to find out what happens internally in the Exerciser by such tests.

TCL Scripts

The following TCL scripts correspond to the requirements of this example:

- *“Starting an Exerciser Session” on page 77*
- *“Establishing a Link” on page 79*
- *“Sending a Single Packet” on page 80*

Establishing an Exerciser Session

Before you can run any tests with the Exerciser, you have to establish a session at the client. A session is responsible for the following:

- Communication between the client and controller PC
- Configuration of the Serial I/O Module and probe board

To start an Exerciser session from a client:

1 Start the PCI Express Protocol Exerciser software on your PC.

The software requires the network name or IP address of the controller PC so that it can communicate with the controller PC. The *Select type of connection* dialog box opens, which allows you to specify that you want to establish a new session and enter the network name (or IP address) of the controller PC.

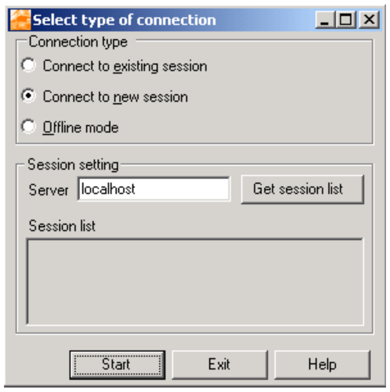


Figure 9 Select type of connection Dialog Box

2 Select *Connect to new session*, enter the network name of the controller PC under *Server* and then click *Start*.

The *Choose a Sessiontype* dialog box opens. This dialog box allows you to define the session type (upstream or downstream).

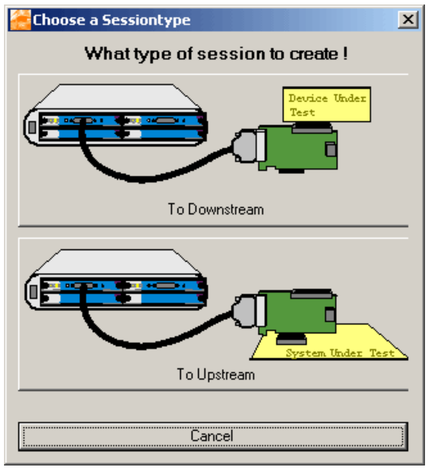
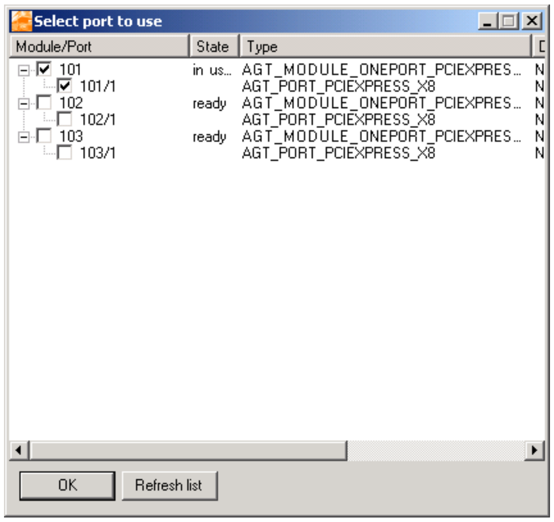


Figure 10 Choose a Sessiontype Dialog Box

3 Because we are testing an add-in card, we need a downstream session. Click *To Downstream*.

The new session is started on the controller PC, the client connects to this session. When the connection has been established, the *Select port to open* dialog box opens. This dialog box indicates which Serial I/O Modules are ready to be used, and which are already in use. When starting a session, you can only select a module that is not in use.

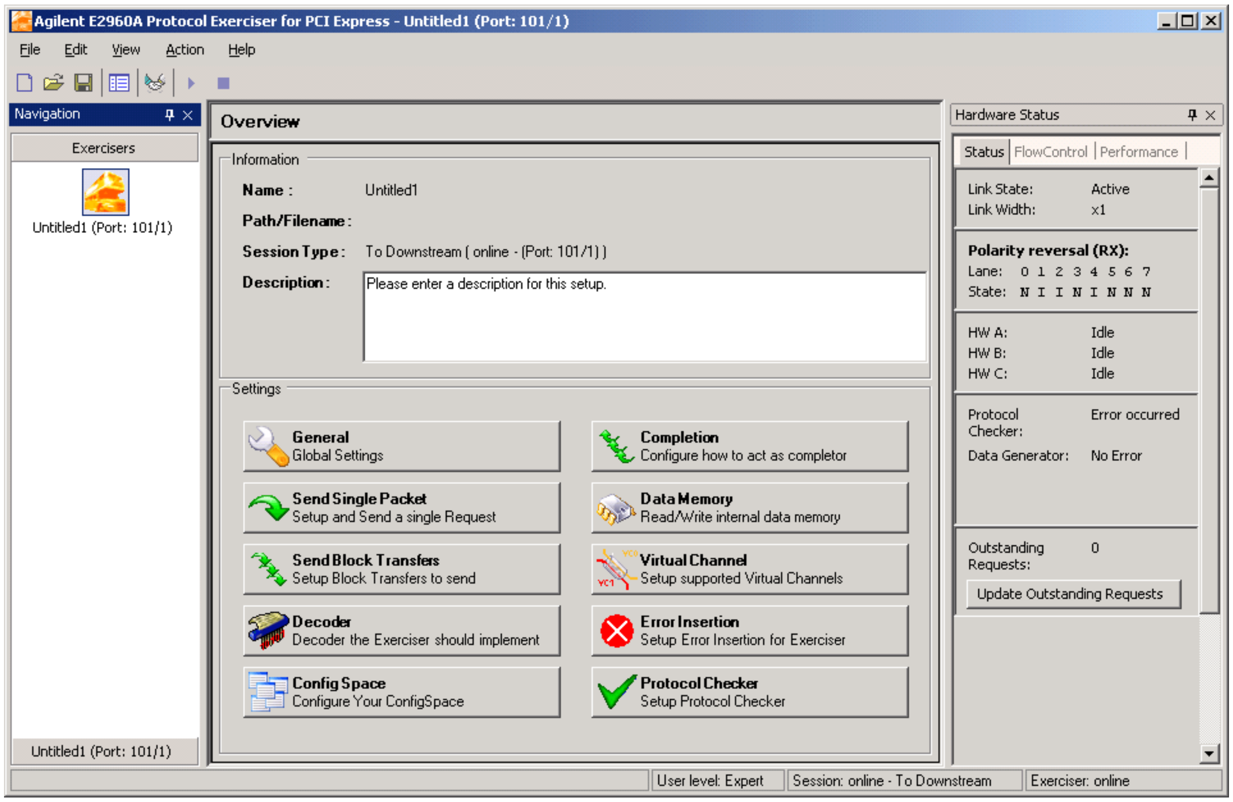


Module/Port	State	Type	D
✓ 101	in us...	AGT_MODULE_ONEPORT_PCIEXPRES...	N
✓ 101/1		AGT_PORT_PCIEXPRESS_X8	N
☐ 102	ready	AGT_MODULE_ONEPORT_PCIEXPRES...	N
☐ 102/1		AGT_PORT_PCIEXPRESS_X8	N
☐ 103	ready	AGT_MODULE_ONEPORT_PCIEXPRES...	N
☐ 103/1		AGT_PORT_PCIEXPRESS_X8	N

Figure 11 Select port to open Dialog Box

4 Select the Serial I/O Module that is connected to the probe board and click *OK*. You can see the module number on the display on the front of the Serial I/O Module.

The embedded software and FPGA contents are now loaded onto the Serial I/O Module and the PCI Express Protocol Exerciser software starts.


Agilent E2960A Protocol Exerciser for PCI Express - Untitled1 (Port: 101/1)
File Edit View Action Help
Navigation
Exercisers
Untitled1 (Port: 101/1)
Overview
Hardware Status
Status FlowControl Performance
Update Outstanding Requests

Information

Name : Untitled1

Path/Filename :

Session Type : To Downstream (online - (Port: 101/1))

Description : Please enter a description for this setup.

Settings

- General — Global Settings
- Completion — Configure how to act as completor
- Send Single Packet — Setup and Send a single Request
- Data Memory — Read/Write internal data memory
- Send Block Transfers — Setup Block Transfers to send
- Virtual Channel — Setup supported Virtual Channels
- Decoder — Decoder the Exerciser should implement
- Error Insertion — Setup Error Insertion for Exerciser
- Config Space — Configure Your ConfigSpace
- Protocol Checker — Setup Protocol Checker

Link State: Active
Link Width: x1

Polarity reversal (RX):
Lane: 0 1 2 3 4 5 6 7
State: N I I N I N N N

HW A: Idle
HW B: Idle
HW C: Idle

Protocol Checker: Error occurred
Data Generator: No Error

Outstanding Requests: 0

User level: Expert | Session: online - To Downstream | Exerciser: online

Figure 12 PCI Express Protocol Exerciser Software

TIP Enter a description and save the setup by clicking the *Save* icon. You can then use this setup for later tests.

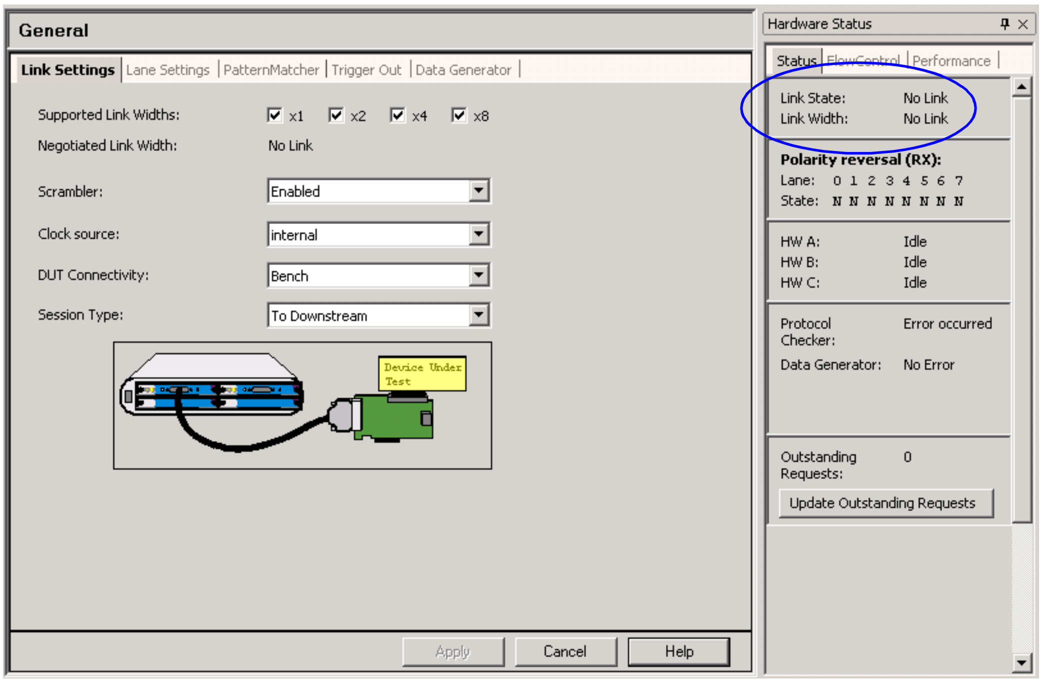
Establishing a PCI Express Link

After you have established a session, you have to establish a PCI Express link to the add-in card. Depending on the add-in card, you may have to configure the link (for example, allowed link width).

To establish a PCI Express link:

1 Select the *General* icon on the Exerciser.

This opens the *General* card with the *Link Settings* tab on top, allowing you to configure the link. The *Hardware Status* informs you about the current link state. We can see here that the link is not up yet.



Supported Link Widths: ☑ x1 ☑ x2 ☑ x4 ☑ x8
Negotiated Link Width: No Link
Scrambler: Enabled
Clock source: internal
DUT Connectivity: Bench
Session Type: To Downstream

Link State: No Link
Link Width: No Link

Polarity reversal (RX):

Lane:	0	1	2	3	4	5	6	7
State:	N	N	N	N	N	N	N	N

HW A: Idle
HW B: Idle
HW C: Idle

Protocol Checker: Error occurred
Data Generator: No Error

Outstanding Requests: 0

Figure 13 Link Settings

2 Select the desired *Supported Link Widths*

This allows you to select which link widths the Exerciser will allow (x1, x2, x4, x8). The support of x1 is mandatory per PCI Express specification and is therefore always selected.

The *Negotiated Link Widths* indicates the link width the two partners have agreed upon after link up. The *Negotiated Link Widths* depend on the capabilities of the Exerciser and add-in card.

3 Select *Bench* for the *DUT Connectivity*. *To Downstream* should already be defined as the *Session Type*.

   DUT Connectivity defines how the probe board is connected to the DUT (Bench or Platform).

NOTE If you were testing a system, *To Upstream* should be selected for the *Session Type*.

4 If not already running, power up the add-in card.

5 Initiate link training by clicking *Link Up* in the *Action* menu.

   The Exerciser now establishes communication with the add-in card. When the link state changes to *Active*, the link has been established.

The Exerciser is now ready for testing.

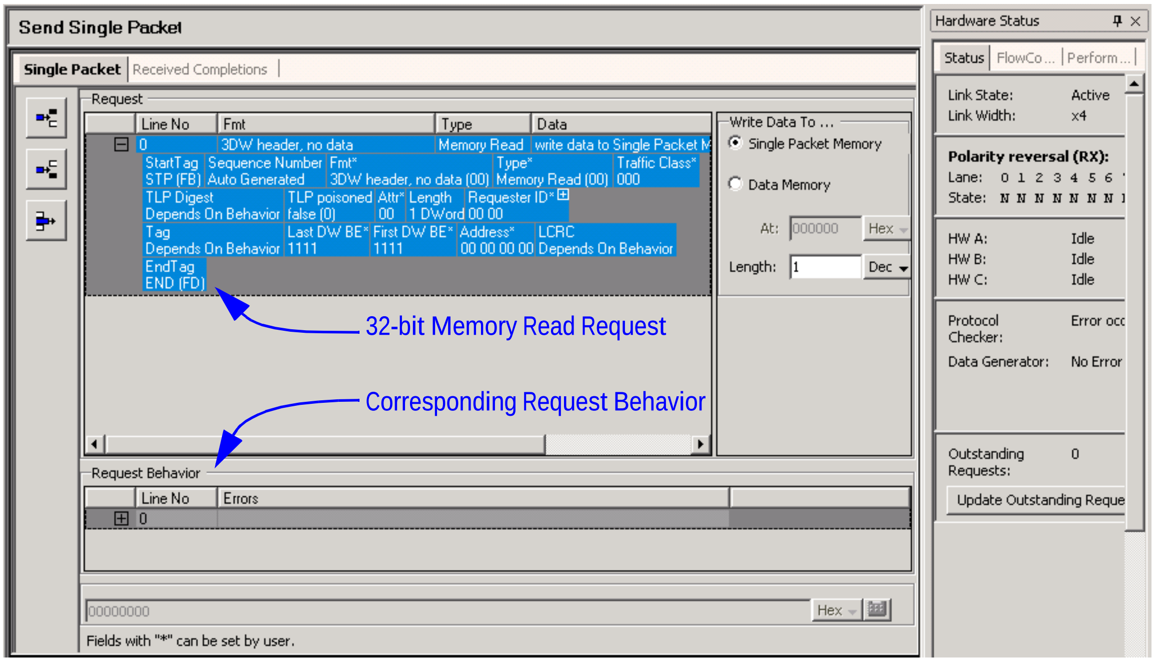
Sending a Single Data Packet

Once we have established a link between the Exerciser and the add-in card, we can quickly perform a memory read on the add-in card by sending a read request as a *single packet*. The Exerciser software provides all standard PCI Express requests. The Exerciser software also allows you to define the request behavior so that specific errors are added to the packets to be transferred.

To send a single data packet:

1 Click the *Send Single Packet* icon.

The *Send Single Packet* card opens with the *Single Packet* tab on top. A 32-bit *Memory Read* request with one corresponding request behavior is included.



Line No	Fmt	Type	Data
0	3DW header, no data	Memory Read	write data to Single Packet M

StartTag: STP (FB); Sequence Number: Auto Generated; Fmt*: 3DW header, no data (00); Type*: Memory Read (00); Traffic Class*: 000
TLP Digest: Depends On Behavior; TLP poisoned: false (0); Attr*: 00; Length: 1 DWord; Requester ID*: 00 00
Tag: Depends On Behavior; Last DW BE*: 1111; First DW BE*: 1111; Address*: 00 00 00 00; LCRC: Depends On Behavior
EndTag: END (FD)

At: 000000 Hex
Length: 1 Dec

Line No	Errors
0	

00000000 Hex

Fields with "*" can be set by user.

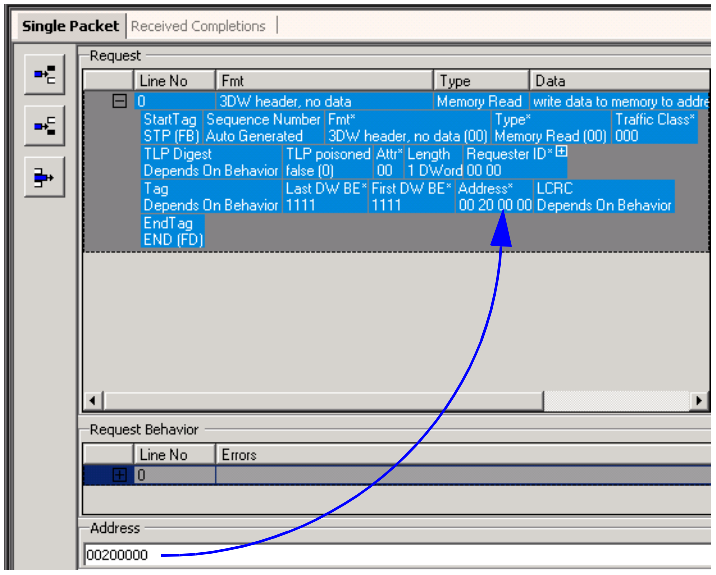
Link State: Active
Link Width: x4
Polarity reversal (RX):
Lane: 0 1 2 3 4 5 6
State: N N N N N N N
HW A: Idle
HW B: Idle
HW C: Idle
Protocol Checker: Error occ
Data Generator: No Error
Outstanding Requests: 0

Figure 14 Send Single Packet Card

Note that you can add a request by clicking the *Add Request* icon.

2 If the PCI Express add-in card has special requirements (for example, the memory area available for access), change any of the request parameters marked with an asterisk (*) as necessary.

In this example, we set the physical address to be read to **0x200000**.



Request

Line No	Fmt	Type	Data
0	3DW header, no data	Memory Read	write data to memory to addre

StartTag	Sequence Number	Fmt*	Type*	Traffic Class*
STP (FB)	Auto Generated	3DW header, no data (00)	Memory Read (00)	000

TLP Digest	TLP poisoned	Attr*	Length	Requester ID*
Depends On Behavior	false (0)	00	1 DWord	00 00

Tag	Last DW BE*	First DW BE*	Address*	LCRC
Depends On Behavior	1111	1111	00 20 00 00	Depends On Behavior

EndTag
END (FD)

Request Behavior

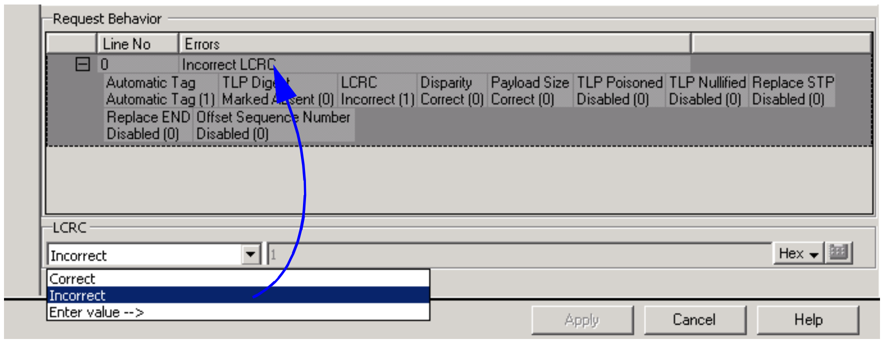
Line No	Errors
0	

Address: 00200000

Figure 15 Request Parameter Setting

3 In *Request Behavior*, define the behavior so that the packet is sent with an incorrect LCRC by clicking *LCRC* in *Request Behavior* and then setting the *LCRC* to *Incorrect* in the field below.

Sending packets with errors allows you to check the packet verification of the add-in card. A packet with an incorrect LCRC must cause the add-in card to reply with a NAK. The Exerciser's replay algorithm then replays the packet with correct LCRC. This packet must then be acknowledged with an ACK by the add-in card.



Request Behavior

Line No	Errors
0	Incorrect LCRC

Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size	TLP Poisoned	TLP Nullified	Replace STP
Automatic Tag (1)	Marked Absent (0)	Incorrect (1)	Correct (0)	Correct (0)	Disabled (0)	Disabled (0)	Disabled (0)

Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)

LCRC: Incorrect 1

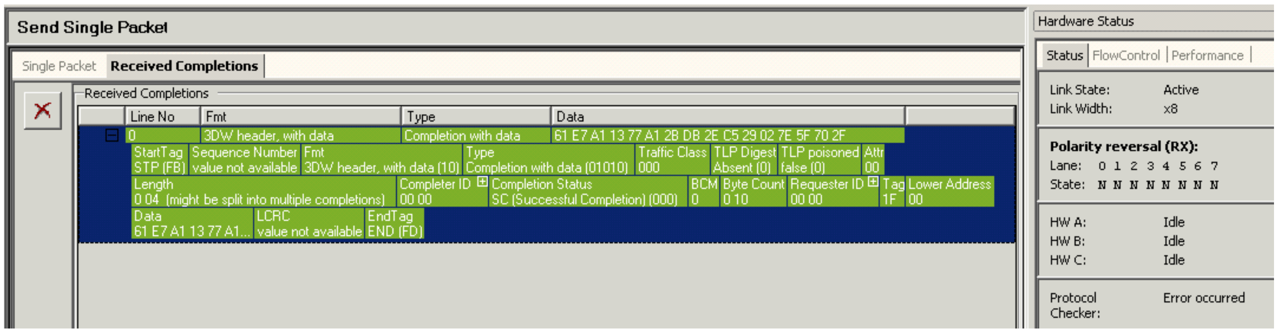
Figure 16 Single Packet with Incorrect LCRC

4 Click *Send Single Packet* in the *Action* menu to send the packet.

The packet is first sent with the error. When the PCI Express system or the PCI Express add-in card returns a NAK (negative-acknowledge), the software sends the packet without the error.

5 Click the *Received Completions* tab to view the completions.

The Exerciser stores completions to single packets and presents these in the *Received Completions* tab. This allows you to see how the add-in card responded to the read request.



	Line No	Fmt	Type	Data
⊟	0	3DW header, with data	Completion with data	61 E7 A1 13 77 A1 2B DB 2E C5 29 02 7E 5F 70 2F

StartTag	Sequence Number	Fmt	Type	Traffic Class	TLP Digest	TLP poisoned	Attr
STP (FB)	value not available	3DW header, with data (10)	Completion with data (01010)	000	Absent (0)	false (0)	00

Length	Completer ID ⊞	Completion Status	BCM	Byte Count	Requester ID ⊞	Tag	Lower Address
0 04 (might be split into multiple completions)	00 00	SC (Successful Completion) (000)	0	0 10	00 00	1F	00

Data	LCRC	EndTag
61 E7 A1 13 77 A1...	value not available	END (FD)

Link State:	Active
Link Width:	x8

Polarity reversal (RX):

Lane:	0	1	2	3	4	5	6	7
State:	N	N	N	N	N	N	N	N

HW A:	Idle
HW B:	Idle
HW C:	Idle
Protocol Checker:	Error occurred

Figure 17 Received Completions

As you can see in this figure, the add-in card sent a completion with data. You do not see the NAK here, because a NAK is not a completion.

What Happens at the Exerciser when sending a Single Packet?

Familiarity with the internal structure of the Exerciser is helpful for understanding how the Exerciser works when sending a single packet. The following figure illustrates the Exerciser components involved in this task.

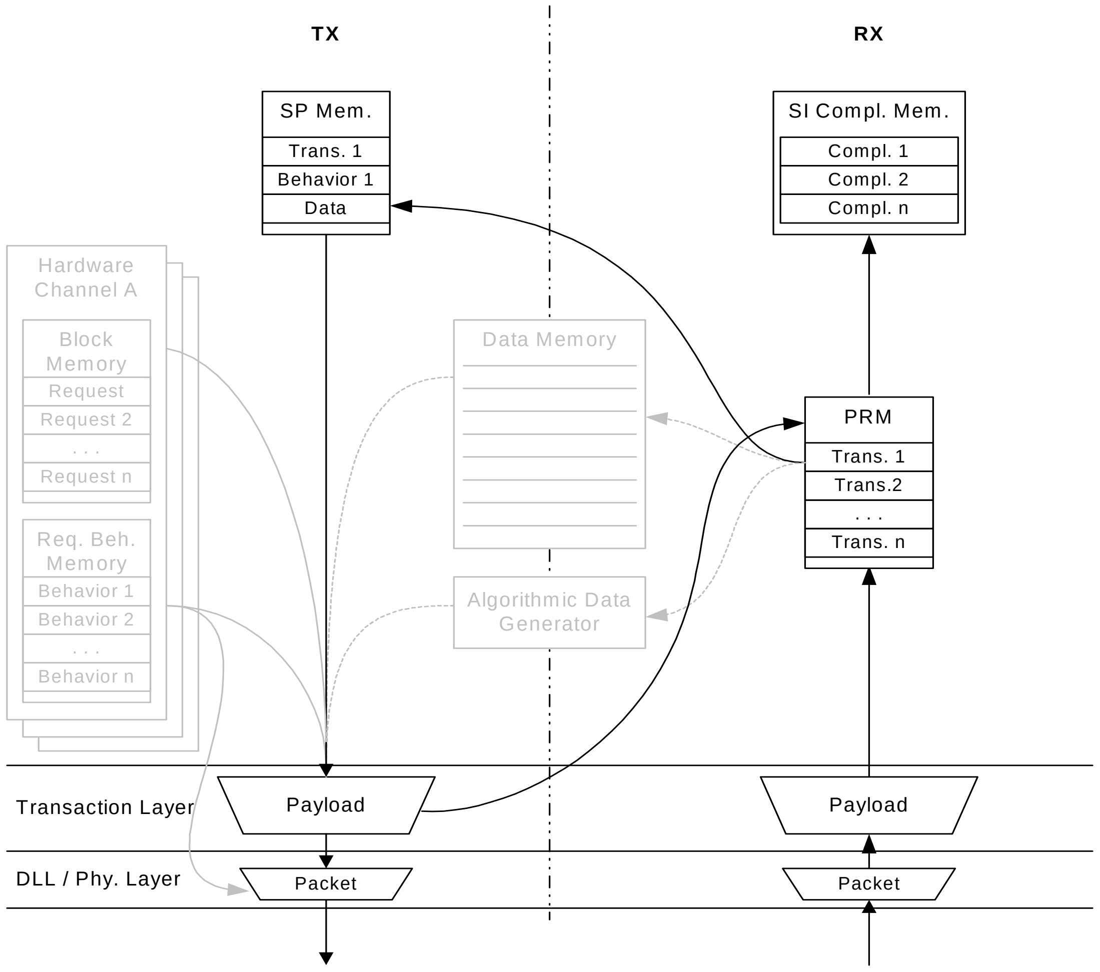


Figure 18 Exerciser as Requester (sending a single packet)

The Exerciser has a memory area for the *Send Single Packet* function (the *SP Mem.* in this figure). When you set up a single packet, the request, request behavior, and any data are written to this area. When you then send the single packet, the packet is assembled at the transaction layer according to the behavior and passed to the data link layer (DLL).

For any requests that expect a response (for example, a memory read), an entry is added in the Pending Request Map (PRM). The PRM keeps track of which responses are still open, which have been returned, and to which memory location returned data from responses should be written.

When a response is returned, the Exerciser looks up the request in the PRM and writes the transaction to the Send Immediate Completion Memory (SI Compl. Mem.). In the user software, this is shown in the *Returned Completions* window. It additionally writes any returned data to the memory location defined with the request (which is typically the Single Packet Memory).

If an error that forces a NAK has to be added to a packet (for example, wrong LCRC), the packet is sent once incorrectly. When the device or system under test returns a NAK, the packet is sent correctly.

See *“Exerciser as Requester” on page 62* for a detailed description of how the Exerciser acts as a requester.

Analyzing the Communication to an Add-In Card

If you have a second Serial I/O Module with probe board and the PCI Express Protocol Analyzer software, you can additionally monitor the packets sent on the link. This allows you to analyze the contents of the individual packets, for example to see how the DUT responds to errored packets.

System Setup

The following system setup is required for this test:

- The add-in card to be tested is plugged into the PCI Express slot at the top of the probe board set up as Analyzer.
- The Analyzer probe board is plugged into the PCI Express slot at the top of the probe board set up as Exerciser.
- An external ATX power supply is connected to the probe board. The power jumper on the probe board should be set to **Ext** (instead **sys**).

  The external power supply is required to power the add-in card. Alternatively, you could plug the probe board into a powered PCI Express slot and set the jumper to **sys**.

The system setup is as follows:

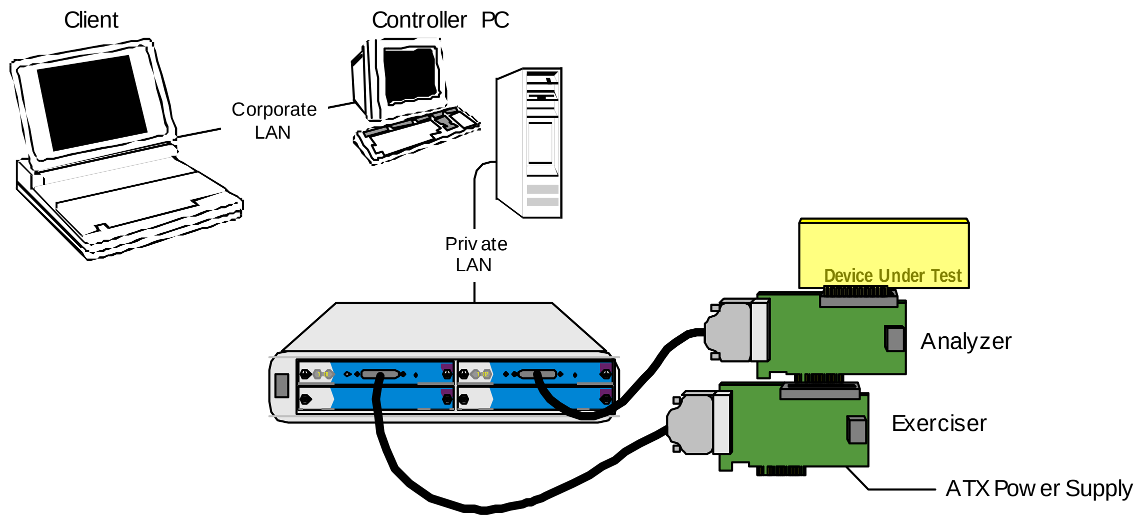


Figure 19 **Protocol Analyzer Setup for Testing Communication with an Add-In Card**

How To Proceed This test is carried out as follows:

1. We have to establish an Exerciser session (described in *"Establishing an Exerciser Session" on page 19*) and an Analyzer session at the client.

   See *"Establishing an Analyzer Session" on page 31*.

2. We set up a trigger on the Analyzer so that the Analyzer captures traffic around a specific packet.

   See *"Setting Up a Trigger" on page 33*.

3. Once the trigger has been set up, we can start the Analyzer and view the captured traffic.

   See *"Starting the Analyzer" on page 36*.

TCL Scripts The following TCL scripts correspond to the requirements of this example:

- *"Starting an Analyzer Session" on page 82*
- *"Downloading a Trigger Setup File to the Analyzer" on page 83*

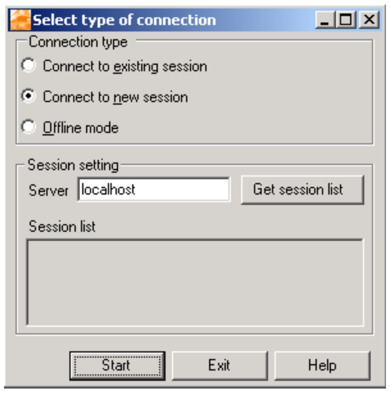
Establishing an Analyzer Session

As with the Exerciser, you must first establish a session at the client.

To start an Anaylzer session:

1 Start the PCI Express Protocol Analyzer software on your PC.

The software requires the network name or IP address of the controller PC so that it can communicate with the controller PC. The *Select type of connection* dialog box opens, which allows you to specify that you want to establish a new session and enter the network name (or IP address) of the controller PC.



Server: localhost

Figure 20 Select type of connection Dialog Box

2 Select *Connect to new session*, enter the IP address or network name of the controller PC under *Server* and then click *Start*.

The client requests a new connection on the controller PC. When the connection has been established, the *Select port to open* dialog box opens with the module and port numbers of the Serial I/O Modules. Because we want to start a new session, we have to select one of the ports that is ready.

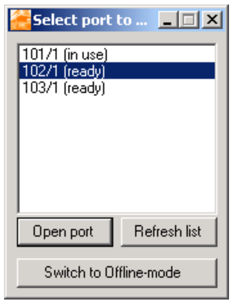


Figure 21 Select port to open Dialog Box

3 Select the port for the Serial I/O Module to be used as Analyzer and click *Open port*. The module numbers are on the display on the front of the module.

The embedded software and FPGA contents are loaded to the Serial I/O Module and the PCI Express Protocol Analyzer software starts.

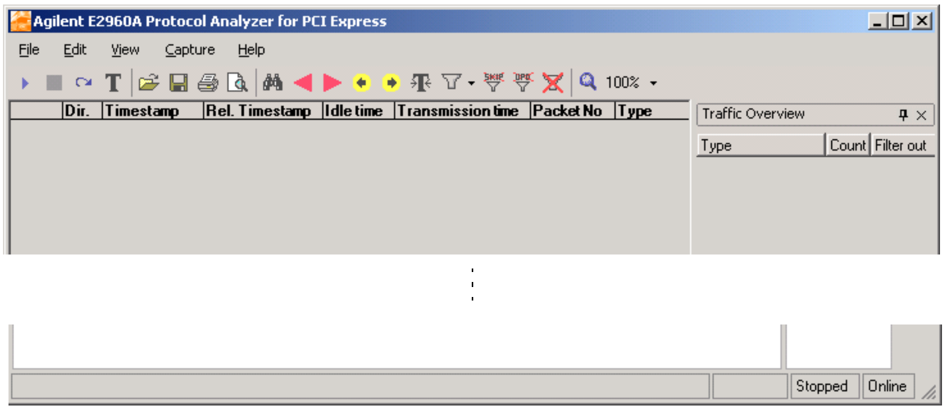


Figure 22 PCI Express Protocol Analyzer Software

The session has now been established. You can see this in the status bar at the bottom of the screen.

Setting Up a Trigger

At this point, both the Exerciser and Analyzer software should be running on the client and sessions should be established to both Serial I/O Modules involved in the test.

It is now necessary to program the Analyzer so that it triggers (starts the capture of packets) upon the occurrence of a particular packet. To keep the test simple, we will set up the Analyzer so that it triggers upon the occurrence of a transaction layer packet (TLP). This us allows us to use the Exerciser as it was previously set up.

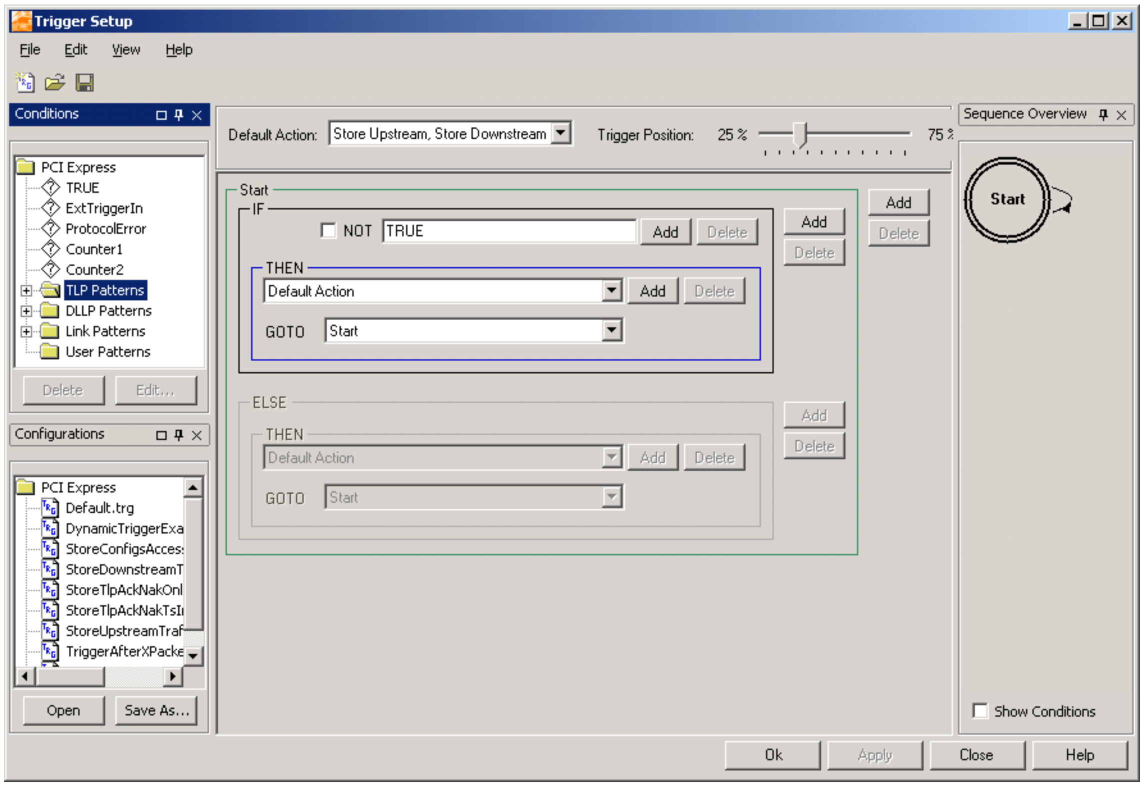
To set up the Analyzer to trigger upon occurrence of a TLP packet:

1 Click *Trigger Setup* in the *Capture* menu.

The *Trigger Setup* dialog box opens, which is used for setting up triggers.

2 On the left side in the *Trigger Setup* dialog box, you can see two columns with *Conditions* and *Configurations*. If these are not visible, check the corresponding items in the *View* menu.

The *Trigger Setup* dialog box should now appear as follows:



Default Action: Store Upstream, Store Downstream
Trigger Position: 25 % 75 %
IF: NOT TRUE
THEN: Default Action
GOTO: Start
ELSE THEN: Default Action
GOTO: Start

Figure 23 Trigger Setup Conditions

3 The *Trigger Setup* dialog box is set up as a series of IF conditions that cause actions to be performed upon their occurrence.

To cause the Analyer to trigger upon occurrence of any TLP pattern, in the Conditions box, open the *TLP Patterns* folder and drag *Any TLP* into the condition field in the *Start* frame.

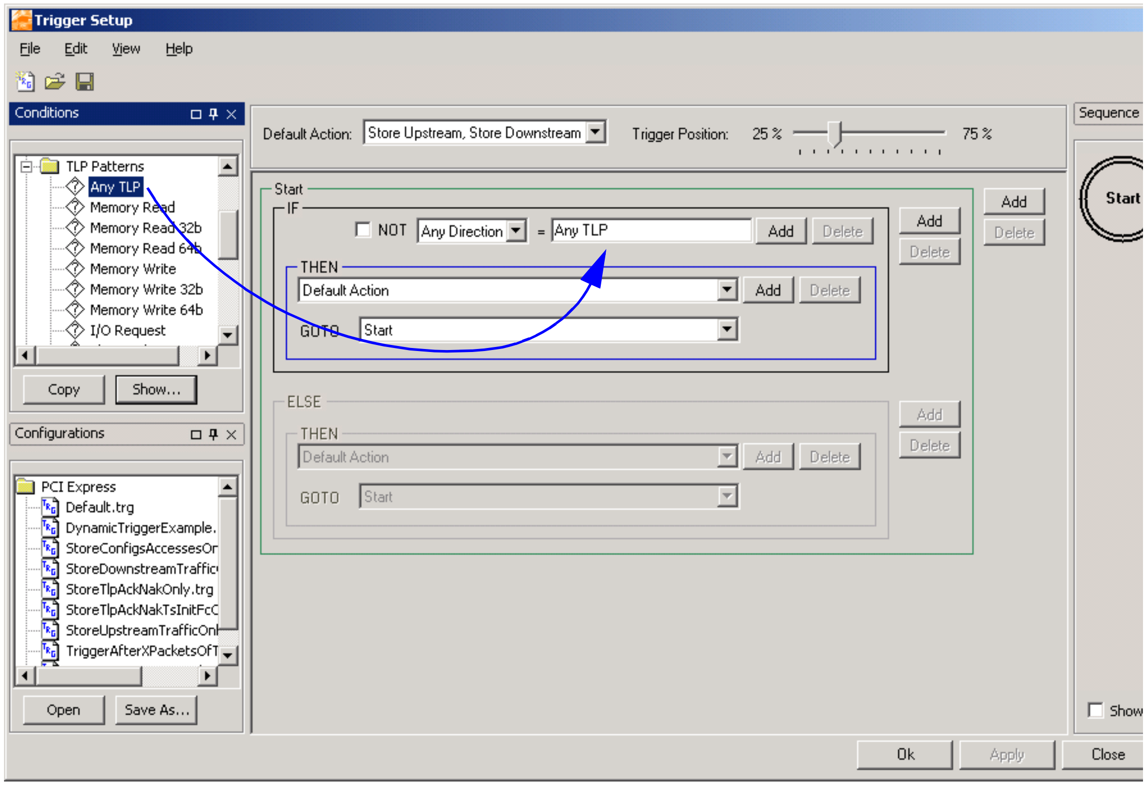


Figure 24 Setting the Condition

4 The *Default Action* defines what to do if the set conditions are not met. Because we want to see all traffic that occurs, we set the *Default Action* to *Store Upstream, Store Downstream*.

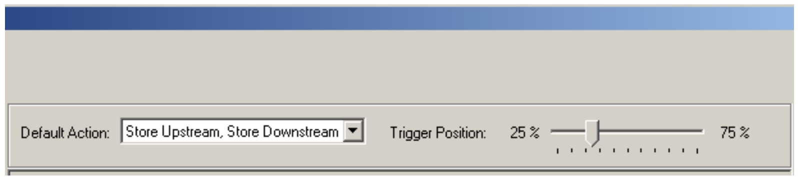
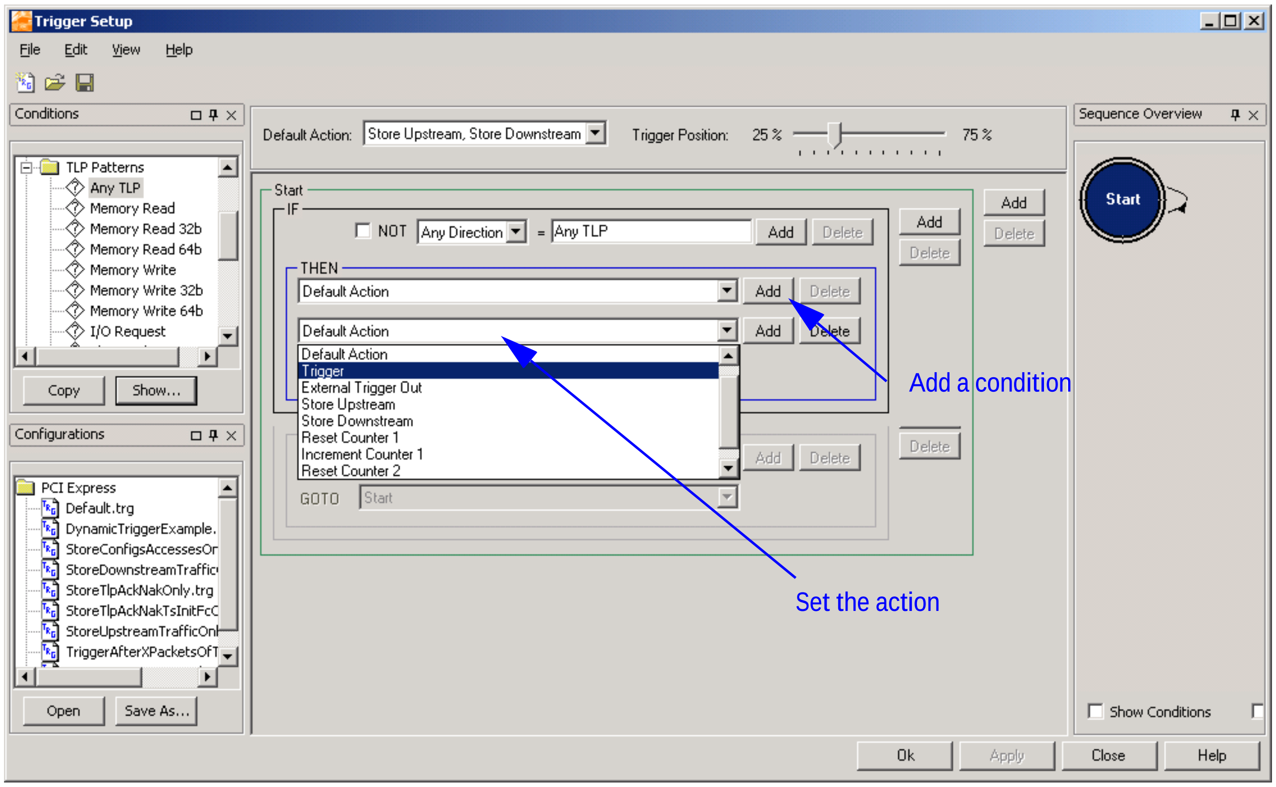

Default Action: Store Upstream, Store Downstream
Trigger Position: 25 % 75 %

Figure 25 Setting the Default Action

5 For our test, we want the Analyzer to trigger upon occurrence of a TLP. We also want the Analyzer to capture the TLP packet that triggers the capture as well. We must therefore:

- Add a condition to the THEN statement.
- Set the first action to *Default Action* (to capture the triggering packet).
- Set the second action to *Trigger* (to start the capture).

This is shown below:

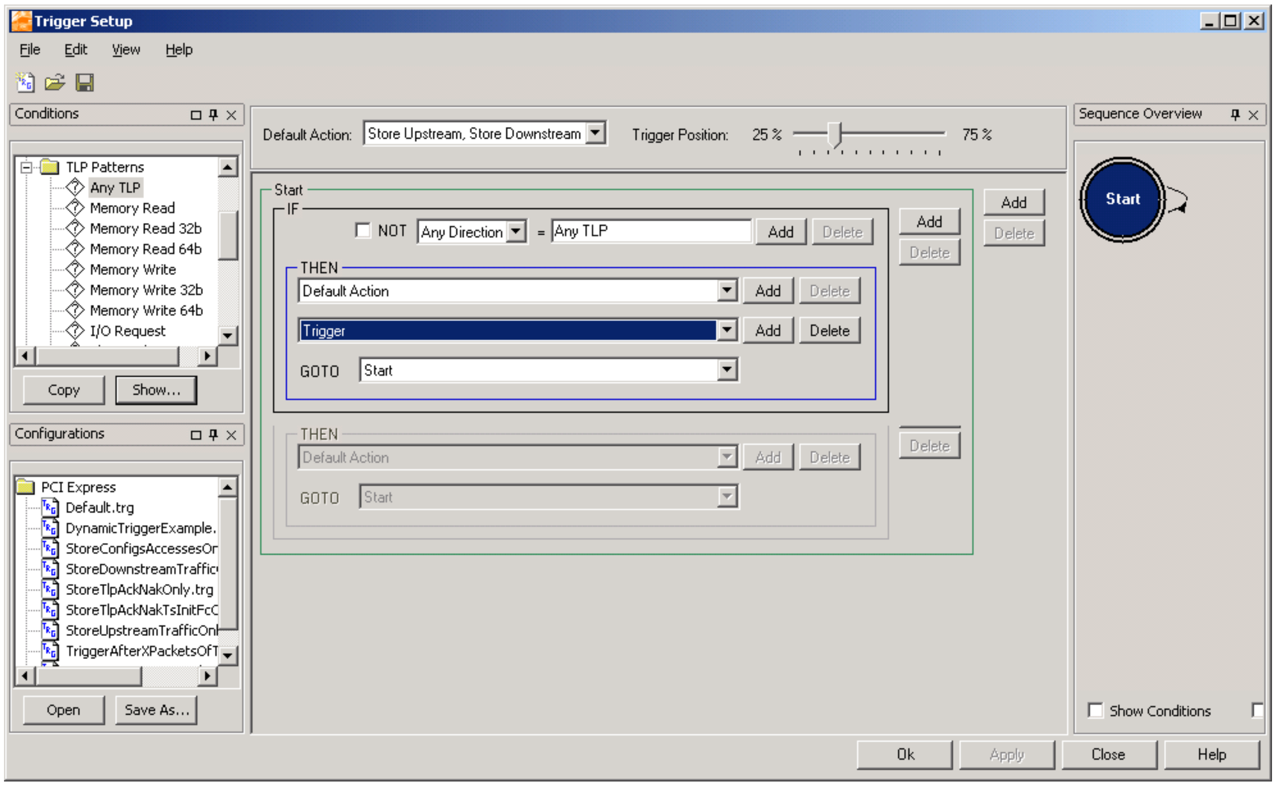


Default Action: Store Upstream, Store Downstream
Trigger Position: 25 % 75 %
IF: NOT Any Direction = Any TLP
THEN: Default Action
Default Action
GOTO Start

Figure 26 Setting the Actions

NOTE Because we only have one IF condition, upon occurrence of the TLP, we go to Start (the only condition block).

The Analyzer appears as shown below:



Default Action: Store Upstream, Store Downstream

Trigger Position: 25 % 75 %

Start

IF

NOT Any Direction = Any TLP

THEN

Default Action

Trigger

GOTO Start

THEN

Default Action

GOTO Start

Figure 27 Analyzer with Setup TLP Trigger

6 Click the *Save* icon, and give the trigger configuration file an appropriate name (for example, “test_trigger_on_tlp.trg”).

7 Click *OK* to apply the trigger and close the *Trigger Setup*.

Starting the Analyzer

Once the trigger has been set up, we can start the Analyzer. When the Analyzer detects a TLP packet, it will trigger, allowing us to view any traffic that is captured.

To start the Analyzer:

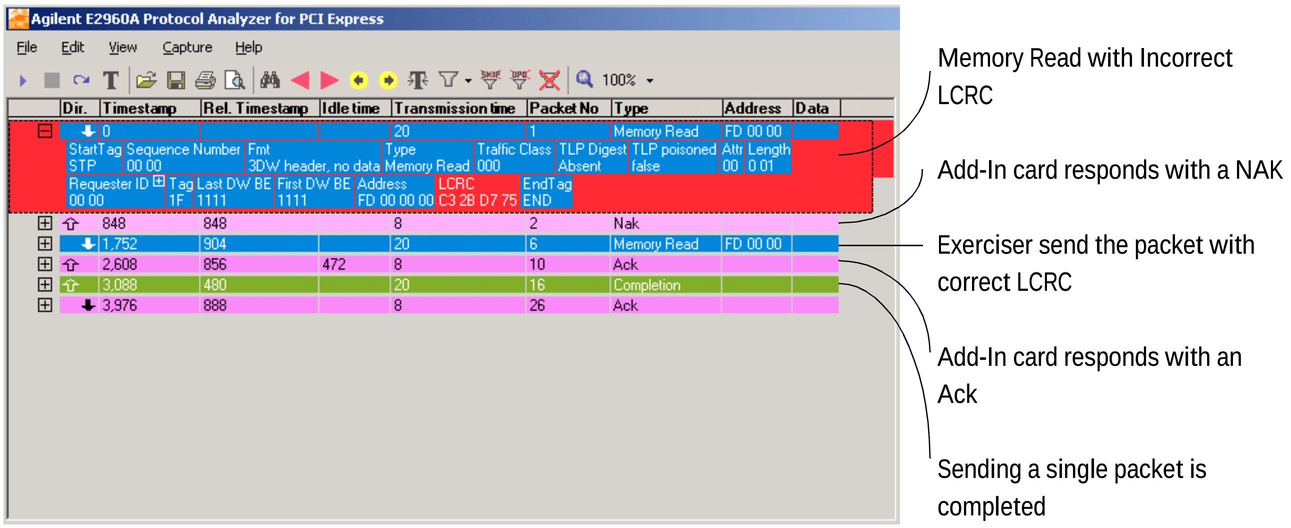
1 In the Protocol Analyzer software, click *Start* in the *Capture* menu.

The Protocol Analyzer now tracks the packets sent through the bus.

2 In the Protocol Exerciser user interface, send a single data packet as described in *“Sending a Single Data Packet” on page 25.*

If any TLPs are sent, the Protocol Analyzer will trigger and fill the memory with traffic 25% before the trigger and 75% after the trigger.

You can now view the captured traffic in the Protocol Analyzer and verify if the behavior of the PCI Express system or add-in card in response to an incorrect LCRC is correct.


Agilent E2960A Protocol Analyzer for PCI Express
File
Edit
View
Capture
Help
100%
Memory Read with Incorrect LCRC
Add-In card responds with a NAK
Exerciser send the packet with correct LCRC
Add-In card responds with an Ack
Sending a single packet is completed

Dir.	Timestamp	Rel. Timestamp	Idle time	Transmission time	Packet No	Type	Address	Data
↓	0			20	1	Memory Read	FD 00 00	
⇧	848	848		8	2	Nak		
↓	1,752	904		20	6	Memory Read	FD 00 00	
⇧	2,608	856	472	8	10	Ack		
⇧	3,088	480		20	16	Completion		
↓	3,976	888		8	26	Ack		

StartTag	Sequence Number	Fmt	Type	Traffic Class	TLP Digest	TLP poisoned	Attr	Length
STP	00 00	3DW header, no data	Memory Read	000	Absent	false	00	0 01

Requester ID	Tag	Last DW BE	First DW BE	Address	LCRC	EndTag
00 00	1F	1111	1111	FD 00 00 00	C3 2B D7 75	END

Figure 28 **PCI Express Protocol Analyzer with Captured Traffic**

In this case, the add-in card responds correctly to the error.

Validating the System

Validating and optimizing your system means exhaustive stress tests, under maximum load, inserted errors and a range of different conditions.

The Protocol Exerciser gives you full and fast control over your test, from simple sequences to worst-case patterns, and, combined with the Protocol Analyzer, deeper investigation and increased test coverage.

In this task, we will write data to the Exerciser’s data memory, write this data to the PCI Express system’s memory, and then read the data from the system memory and compare it with the data that was sent. Comparing the outgoing and incoming data allows you to verify the correct data transfer.

We will also use this task to check if the PCI Express system’s DLL/Phy layer functions correctly by adding header errors to the transferred packets.

We will also set up the Protocol Analyzer to capture the traffic generated between the Protocol Exerciser and the system.

System Setup

The following system setup is applied for basic platform testing; especially for platforms that are not stable enough to support software running on them.

The following system setup is required for this test:

- The Protocol Analyzer probe board is plugged into the PCI Express system.
- The Protocol Exerciser probe board is plugged into the PCI Express slot at the top of the probe board set up as Analyzer.

The system setup is as follows:

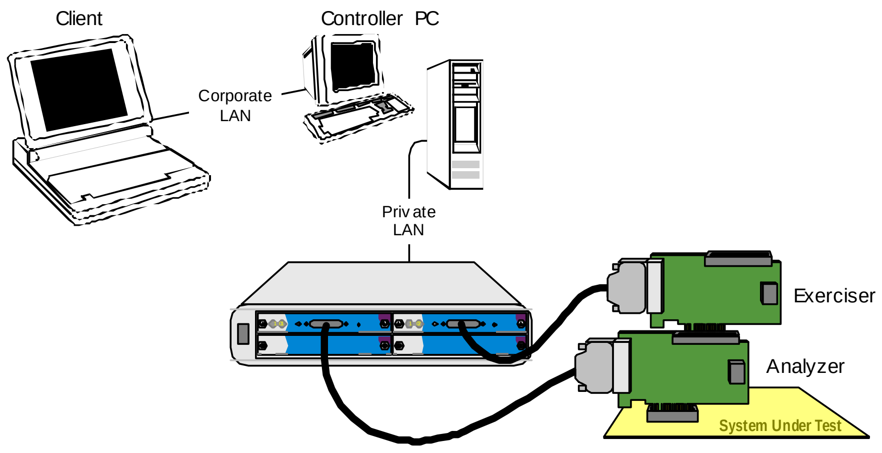


Figure 29 Protocol Exerciser and Analyzer Setup for Testing a System

How To Proceed

This test is carried out as follows:

1. We have to establish an Exerciser session (described in *"Establishing an Exerciser Session" on page 19*) and an Analyzer session at the client.

   See *"Establishing an Analyzer Session" on page 31*.

2. We set up and establish a PCI Express link between the Exerciser and the add-in card.

   See *"Establishing a PCI Express Link" on page 23*.

3. We set up a trigger on the Analyzer so that the Analyzer captures traffic around a memory read request.

   See *"Setting Up the Trigger for the System Validation Test" on page 40*.

4. We write data to the Exerciser's data memory for a data compare.

   See *"Writing Data to the Protocol Exerciser's Data Memory" on page 43*.

5. We set up a block transfer that contains a memory write and a memory read request with different behaviors and an included error.

   See *"Setting Up and Running Block Transfers" on page 44*

6. We view and analyze the results on the Analyzer's user interface.

   See *"Viewing the Results Using the User Interface" on page 49*

TCL Scripts The following TCL scripts correspond to the requirements of this example:

- *“Writing Data to the Protocol Exerciser’s Data Memory” on page 84*
- *“Setting Up and Running Block Transfers” on page 85*

Setting Up the Trigger for the System Validation Test

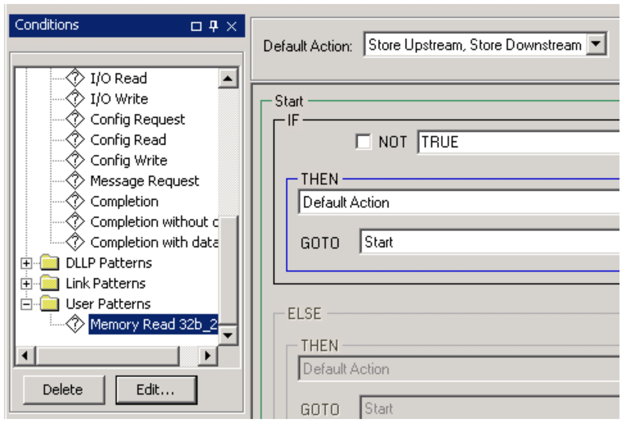
In this example, we will set up a memory write followed by a read to the PCI Express system at **00FD0000** in the memory. We must first set up the Analyzer so that it triggers upon a read to this memory area.

To set up the trigger:

1 In the Analyzer user interface, open the *Trigger Setup* dialog box.

2 Under *Conditions,* open the *TLP Patterns* folder, select *Memory Read 32b*, and click *Copy*.

The condition is saved as a copy under *User Patterns*.



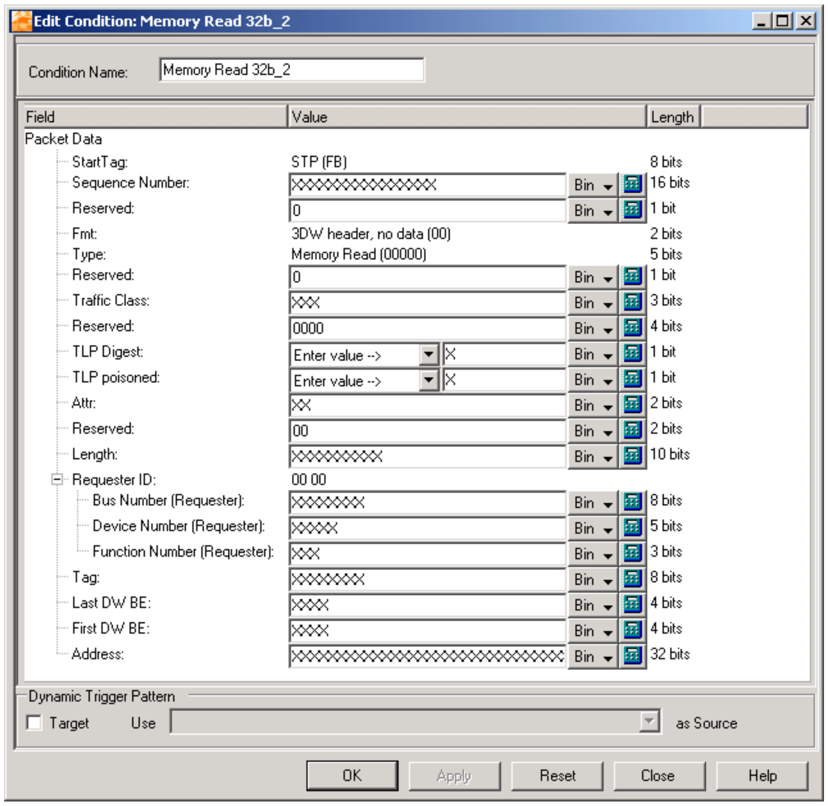
Default Action: Store Upstream, Store Downstream

IF NOT TRUE
THEN Default Action
GOTO Start
ELSE
THEN Default Action
GOTO Start

Figure 30 **Trigger Setup Dialog Box**

3 Double-click the copied condition to edit it (or click *Edit*).

The *Edit Condition* dialog box opens.



Edit Condition: Memory Read 32b_2

Condition Name: Memory Read 32b_2

Field	Value		Length
Packet Data			
StartTag:	STP (FB)		8 bits
Sequence Number:	XXXXXXXXXXXXXXXX	Bin	16 bits
Reserved:	0	Bin	1 bit
Fmt:	3DW header, no data (00)		2 bits
Type:	Memory Read (00000)		5 bits
Reserved:	0	Bin	1 bit
Traffic Class:	XXX	Bin	3 bits
Reserved:	0000	Bin	4 bits
TLP Digest:	Enter value --> X	Bin	1 bit
TLP poisoned:	Enter value --> X	Bin	1 bit
Attr:	XX	Bin	2 bits
Reserved:	00	Bin	2 bits
Length:	XXXXXXXXXX	Bin	10 bits
Requester ID:	00 00		
Bus Number (Requester):	XXXXXXXX	Bin	8 bits
Device Number (Requester):	XXXXX	Bin	5 bits
Function Number (Requester):	XXX	Bin	3 bits
Tag:	XXXXXXXX	Bin	8 bits
Last DW BE:	XXXX	Bin	4 bits
First DW BE:	XXXX	Bin	4 bits
Address:	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	Bin	32 bits

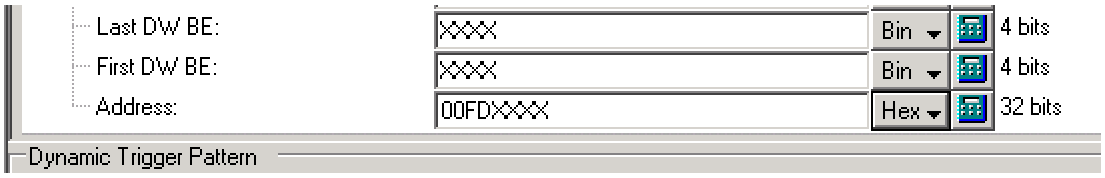
Dynamic Trigger Pattern

☐ Target Use as Source

Figure 31 Edit Condition dialog box

4 Set the format to *Hex* (set with the button next to the field) and set *Address* to **00FDXXXX**.

When this condition is used, the Analyzer triggers at any 32-bit read access to the range from **00FD0000** to **00FDFFFF**.



Last DW BE:	XXXX	Bin	4 bits
First DW BE:	XXXX	Bin	4 bits
Address:	00FDXXXX	Hex	32 bits

Figure 32 Edit a Condition

5 Close the *Edit Condition* dialog box and drag the modified condition into the *Condition* field.

6 Set the *Default Action* to *Store Upstream*, *Store Downstream*.

7 Add an action by clicking *Add* next to the action field and set the first action field to *Default Action* and the second to *Trigger*.

When any 32-bit memory read request to the defined address range is detected, the Analyzer will trigger.

The *Trigger Setup* should now appear as follows:

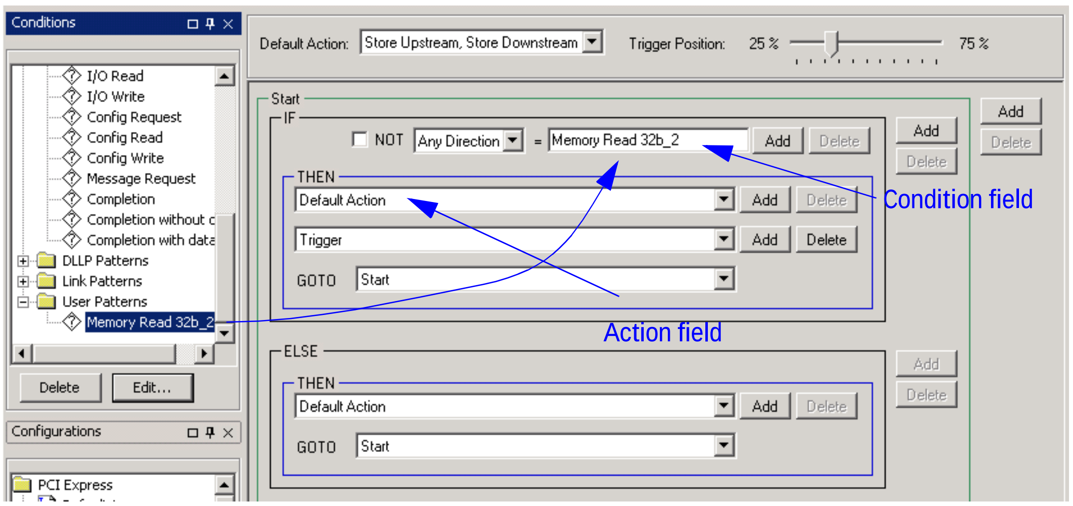

Default Action: Store Upstream, Store Downstream
Trigger Position: 25 % 75 %
IF: NOT Any Direction = Memory Read 32b_2
THEN: Default Action
Trigger
GOTO Start
ELSE
THEN: Default Action
GOTO Start

Figure 33 Trigger Setup for the System Validation Test

8 Save the trigger configuration, for example, to “test_trigger_on_readmem32.trg”.

9 Apply the trigger and close the *Trigger Setup*.

10 In the User Interface, select *Run* in the *Capture* menu to start the Analyzer.

The Protocol Analyzer now tracks the packets sent through the bus. If a 32-bit memory read request at address **00FDXXXX** is sent, the Protocol Analyzer triggers and records the traffic in its trace memory.

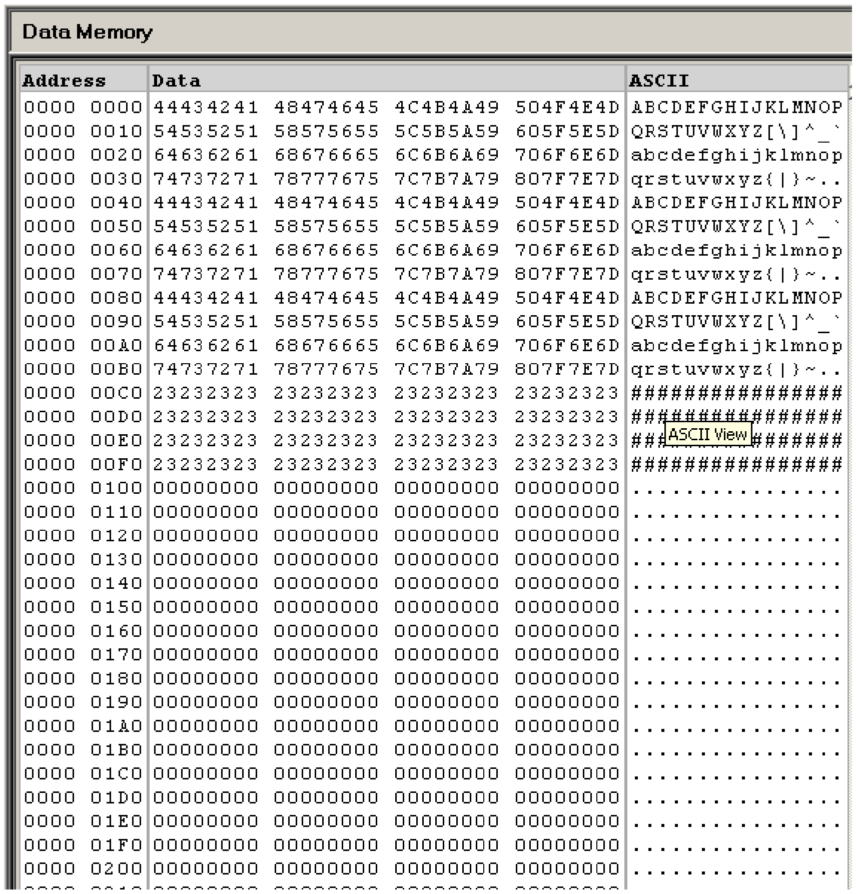
Writing Data to the Protocol Exerciser’s Data Memory

To verify that the data read and write between the Exerciser and the PCI Express add-in card works correctly, the data memory must be filled with defined data.

To write data to the exerciser’s data memory:

1 Click the *Data Memory* icon.

2 For example, fill the *Data Memory* from **0000 0000** to **0000 00FF** with human-readable data.



```
Address    Data                                    ASCII
0000 0000  44434241 48474645 4C4B4A49 504F4E4D     ABCDEFGHIJKLMNOP
0000 0010  54535251 58575655 5C5B5A59 605F5E5D     QRSTUVWXYZ[\]^_`
0000 0020  64636261 68676665 6C6B6A69 706F6E6D     abcdefghijklmnop
0000 0030  74737271 78777675 7C7B7A79 807F7E7D     qrstuvwxyz{|}~..
0000 0040  44434241 48474645 4C4B4A49 504F4E4D     ABCDEFGHIJKLMNOP
0000 0050  54535251 58575655 5C5B5A59 605F5E5D     QRSTUVWXYZ[\]^_`
0000 0060  64636261 68676665 6C6B6A69 706F6E6D     abcdefghijklmnop
0000 0070  74737271 78777675 7C7B7A79 807F7E7D     qrstuvwxyz{|}~..
0000 0080  44434241 48474645 4C4B4A49 504F4E4D     ABCDEFGHIJKLMNOP
0000 0090  54535251 58575655 5C5B5A59 605F5E5D     QRSTUVWXYZ[\]^_`
0000 00A0  64636261 68676665 6C6B6A69 706F6E6D     abcdefghijklmnop
0000 00B0  74737271 78777675 7C7B7A79 807F7E7D     qrstuvwxyz{|}~..
0000 00C0  23232323 23232323 23232323 23232323     ################
0000 00D0  23232323 23232323 23232323 23232323     ################
0000 00E0  23232323 23232323 23232323 23232323     ################
0000 00F0  23232323 23232323 23232323 23232323     ################
0000 0100  00000000 00000000 00000000 00000000     ................
0000 0110  00000000 00000000 00000000 00000000     ................
0000 0120  00000000 00000000 00000000 00000000     ................
0000 0130  00000000 00000000 00000000 00000000     ................
0000 0140  00000000 00000000 00000000 00000000     ................
0000 0150  00000000 00000000 00000000 00000000     ................
0000 0160  00000000 00000000 00000000 00000000     ................
0000 0170  00000000 00000000 00000000 00000000     ................
0000 0180  00000000 00000000 00000000 00000000     ................
0000 0190  00000000 00000000 00000000 00000000     ................
0000 01A0  00000000 00000000 00000000 00000000     ................
0000 01B0  00000000 00000000 00000000 00000000     ................
0000 01C0  00000000 00000000 00000000 00000000     ................
0000 01D0  00000000 00000000 00000000 00000000     ................
0000 01E0  00000000 00000000 00000000 00000000     ................
0000 01F0  00000000 00000000 00000000 00000000     ................
0000 0200  00000000 00000000 00000000 00000000     ................
```

Figure 34 Data Memory Entry

TIP You can also export data from or import data into the *Data Memory* by selecting *Export* or *Import* in the *File* menu.

Setting Up and Running Block Transfers

This section describes how you set up block transfers in the user interface.

To set up block transfers:

1 In the PCI Express Protocol Exerciser software, click the *Send Block Transfers* icon.

The *Send Block Transfers* card opens with the *Hardware Channel A* tab on top.

2 Click the *Insert New Request* icon.

The *Select a Template* dialog box opens.

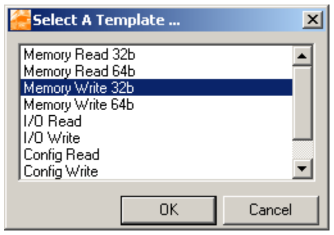


Figure 35 Select a Template Dialog Box

3 Select the *Memory Write 32* template and click *OK*.

A line describing the memory write is added to the *Request* and one *request behavior*.

4 Set the address in the PCI Express system memory to be used for the write.

In this test, address **00FD 0000** is used. If your PCI Express system does not support this address, change the address as necessary.

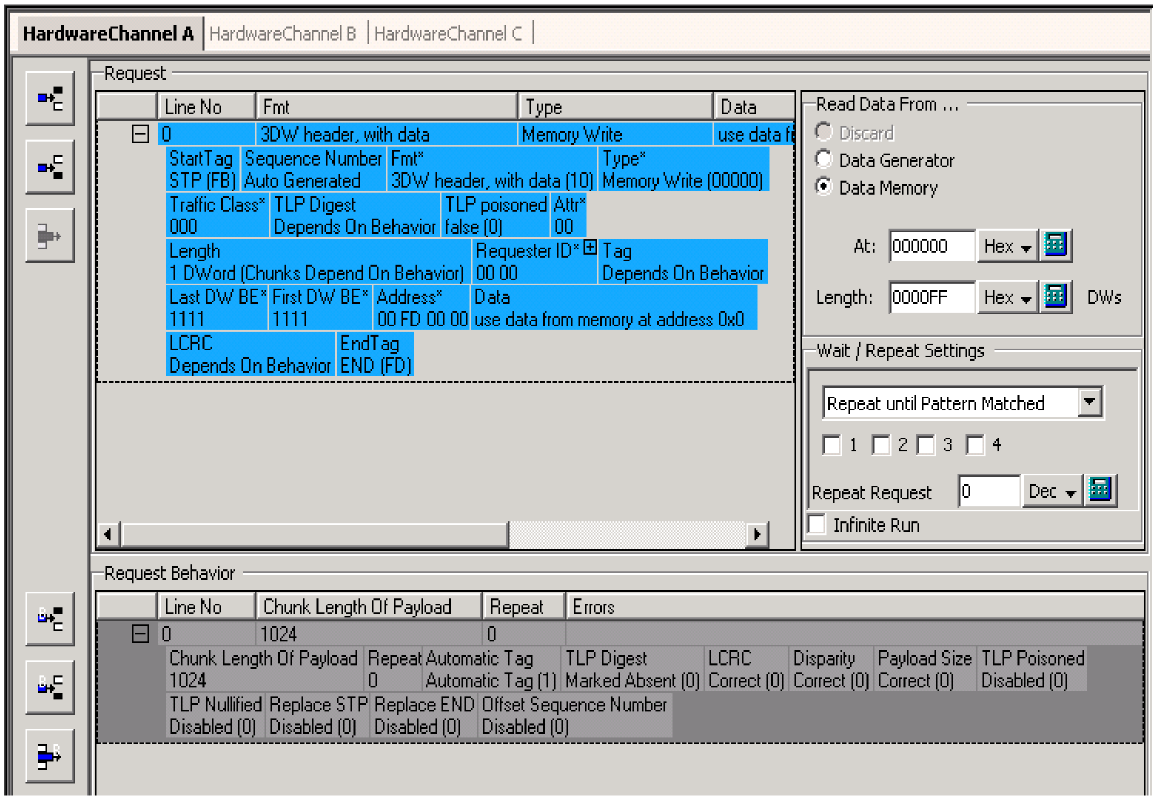
Note that you also have to change the trigger set up in *"Setting Up the Trigger for the System Validation Test" on page 40*.

5 Set the address and length of the data to be used from the Data Memory.

The following values are used in this example:

– *At*: **000000**

– *Length*: **0000FF**



HardwareChannel A | HardwareChannel B | HardwareChannel C

Request

Line No	Fmt	Type	Data
0	3DW header, with data	Memory Write	use data f

StartTag	Sequence Number	Fmt*	Type*
STP (FB)	Auto Generated	3DW header, with data (10)	Memory Write (00000)

Traffic Class*	TLP Digest	TLP poisoned	Attr*
000	Depends On Behavior	false (0)	00

Length	Requester ID*	Tag
1 DWord (Chunks Depend On Behavior)	00 00	Depends On Behavior

Last DW BE*	First DW BE*	Address*	Data
1111	1111	00 FD 00 00	use data from memory at address 0x0

LCRC	EndTag
Depends On Behavior	END (FD)

Read Data From ...
- Discard
- Data Generator
- [✓ Data Memory]

At: 000000 Hex
Length: 0000FF Hex DWs

Wait / Repeat Settings
Repeat until Pattern Matched
☐ 1 ☐ 2 ☐ 3 ☐ 4
Repeat Request 0 Dec
☐ Infinite Run

Request Behavior

Line No	Chunk Length Of Payload	Repeat	Errors
0	1024	0	

Chunk Length Of Payload	Repeat	Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size	TLP Poisoned
1024	0	Automatic Tag (1)	Marked Absent (0)	Correct (0)	Correct (0)	Correct (0)	Disabled (0)

TLP Nullified	Replace STP	Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)

Figure 36 Memory Write Settings

6 Select the *Memory Read 32* template and click *OK*.

A line describing the memory read is added to the *Request*.

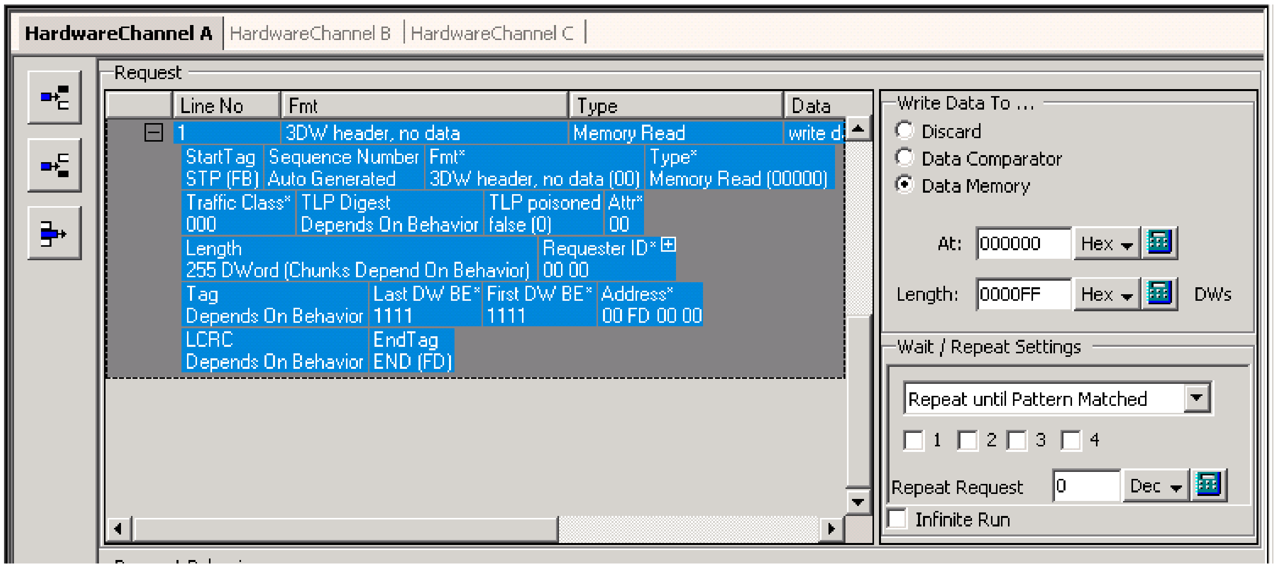
7 Set the memory address of the PCI Express system from which the data should be read to **00FD 0000**.

8 To make the data compare possible, address and length of the location in the *Data Memory* to which the data from the PCI Express system should be written must be the same as for the read request.

In this case:

– *At:* **000000**

– *Length:* **0000FF**



Line No	Fmt	Type	Data
1	3DW header, no data	Memory Read	write d

StartTag	Sequence Number	Fmt*	Type*
STP (FB)	Auto Generated	3DW header, no data (00)	Memory Read (00000)
Traffic Class*	TLP Digest	TLP poisoned	Attr*
000	Depends On Behavior	false (0)	00
Length		Requester ID*	
255 DWord (Chunks Depend On Behavior)		00 00	
Tag	Last DW BE*	First DW BE*	Address*
Depends On Behavior	1111	1111	00 FD 00 00
LCRC	EndTag		
Depends On Behavior	END (FD)		

Write Data To ...
Discard [✓ Data Memory] Data Comparator

At: 000000 Hex
Length: 0000FF Hex DWs

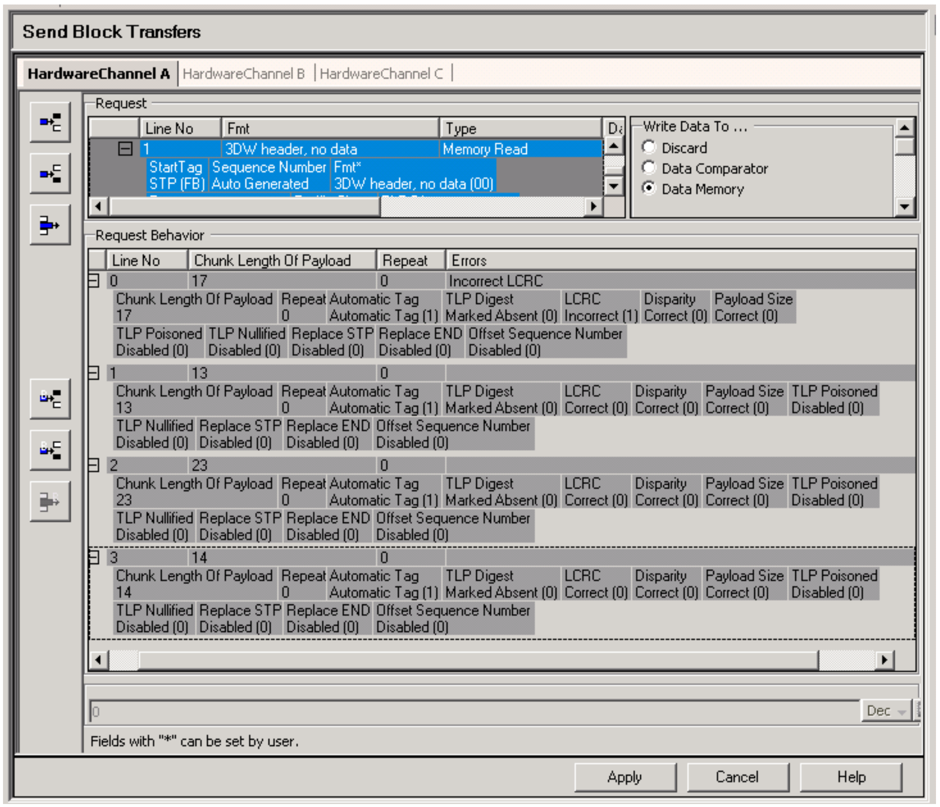
Wait / Repeat Settings
Repeat until Pattern Matched
1 2 3 4
Repeat Request 0 Dec
Infinite Run

Figure 37 Memory Read Settings

9 Click the *Insert New Behavior* icon to insert four *Request Behaviors*.

10 Change the *Chunk Lengths* of the payload to 17, 13, 23, and 14, for example.

11 Change some of the other packet parameters to test how the PCI Express system reacts to faulty packets (for example, if *Payload Size*, *Disparity*, or *LCRC* are incorrect).



Request

Line No	Fmt	Type
1	3DW header, no data	Memory Read

StartTag	Sequence Number	Fmt*
STP (FB)	Auto Generated	3DW header, no data (00)

Write Data To ...
( ) Discard
( ) Data Comparator
(●) Data Memory

Request Behavior

Line No	Chunk Length Of Payload	Repeat	Errors
0	17	0	Incorrect LCRC

Chunk Length Of Payload	Repeat	Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size
17	0	Automatic Tag (1)	Marked Absent (0)	Incorrect (1)	Correct (0)	Correct (0)

TLP Poisoned	TLP Nullified	Replace STP	Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)

Line No	Chunk Length Of Payload	Repeat	Errors
1	13	0	

Chunk Length Of Payload	Repeat	Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size	TLP Poisoned
13	0	Automatic Tag (1)	Marked Absent (0)	Correct (0)	Correct (0)	Correct (0)	Disabled (0)

TLP Nullified	Replace STP	Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)

Line No	Chunk Length Of Payload	Repeat	Errors
2	23	0	

Chunk Length Of Payload	Repeat	Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size	TLP Poisoned
23	0	Automatic Tag (1)	Marked Absent (0)	Correct (0)	Correct (0)	Correct (0)	Disabled (0)

TLP Nullified	Replace STP	Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)

Line No	Chunk Length Of Payload	Repeat	Errors
3	14	0	

Chunk Length Of Payload	Repeat	Automatic Tag	TLP Digest	LCRC	Disparity	Payload Size	TLP Poisoned
14	0	Automatic Tag (1)	Marked Absent (0)	Correct (0)	Correct (0)	Correct (0)	Disabled (0)

TLP Nullified	Replace STP	Replace END	Offset Sequence Number
Disabled (0)	Disabled (0)	Disabled (0)	Disabled (0)

0 Dec

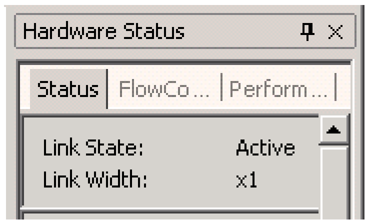
Fields with "*" can be set by user.

Figure 38 Block Transfers with different Lengths of Payload

12 Establish the link:

- In the *General* tab, check the desired link width.
- Select *Platform* for the *DUT Connectivity* and *To Upstream* for the *Session Type*.
- Initiate link training by clicking *Link Up* in the *Action* menu.

The link is established if *Link State* is *Active*.



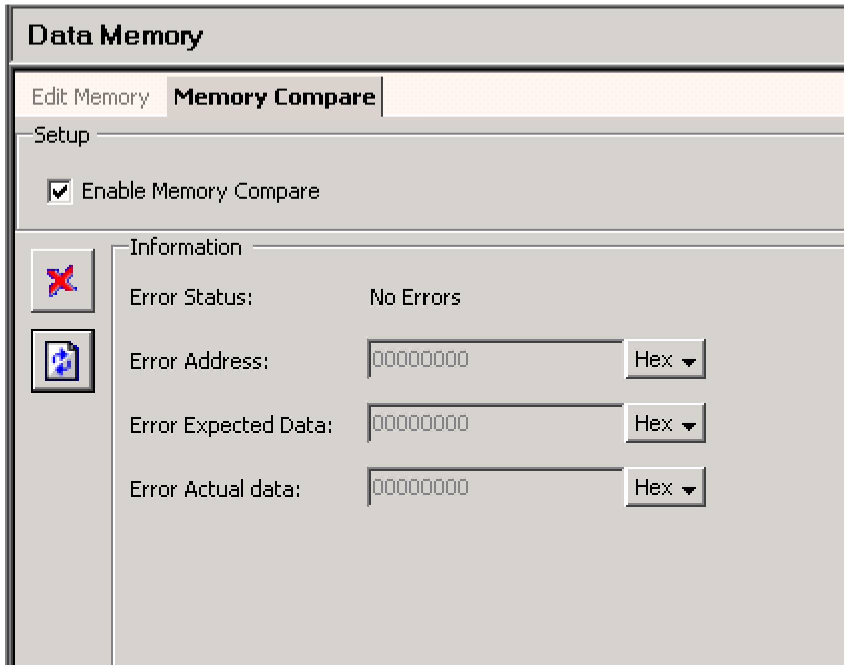
Link State: Active
Link Width: x1

Figure 39 **Active Link State**

13 Enable the data compare:

– Open the *Data Memory* and click the *Memory Compare* tab.

– Check *Enable Memory Compare*.



☑ Enable Memory Compare

Error Status: No Errors

Error Address: 00000000

Error Expected Data: 00000000

Error Actual data: 00000000

When running the block transfer, the Exerciser will compare the original data with the data read from the PCI Express system and give you information about error address, the expected data and the actual data.

14 Click *Send Block Transfer* in the *Action* menu to send the block.

Viewing the Results Using the User Interface

Once the trigger has been set up, we can start the Analyzer. When the Analyzer detects a 32-bit memory read request, it will trigger, allowing us to view any traffic that is captured.

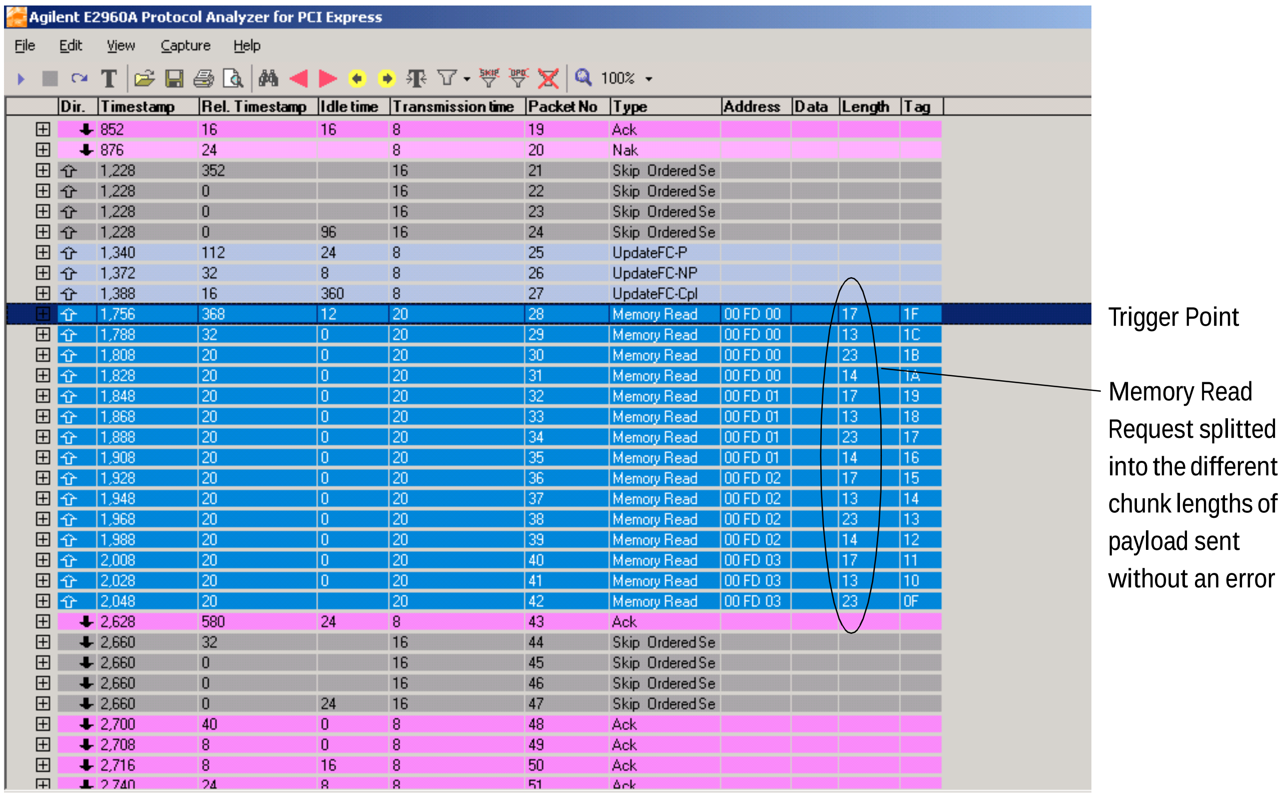
To start the Analyzer:

1 In the Protocol Analyzer software, click *Start* in the *Capture* menu.

  The Protocol Analyzer now tracks the packets sent through the bus.

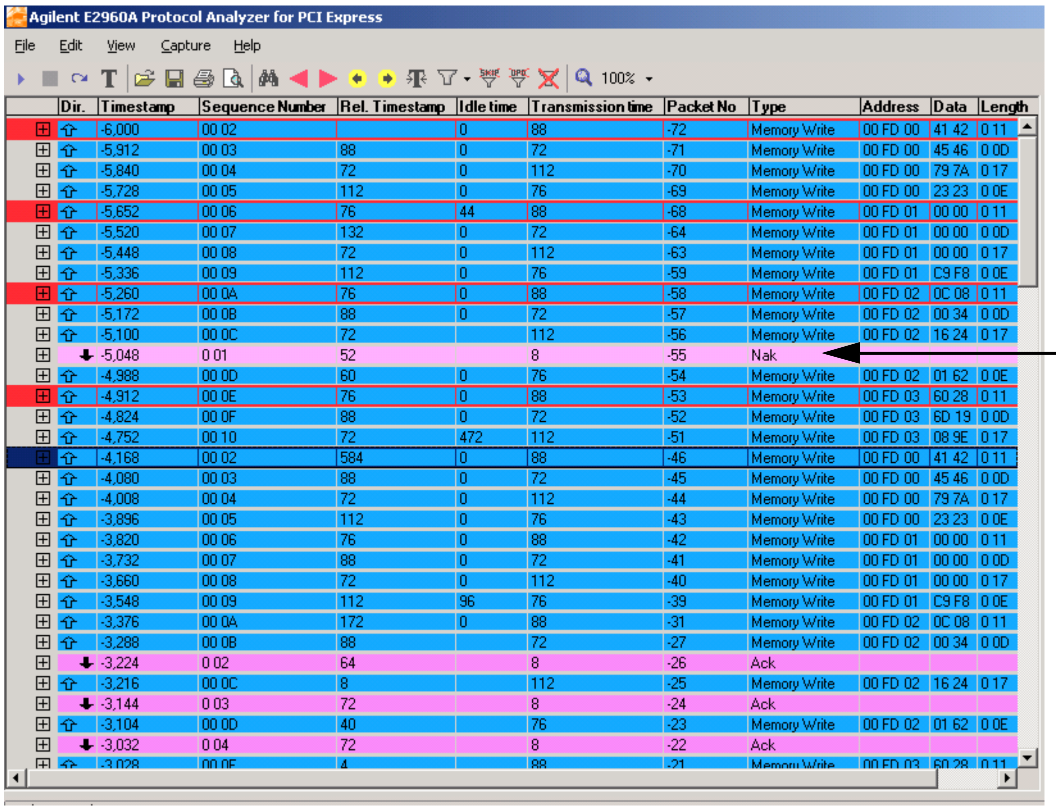
2 In the Protocol Exerciser user interface, send the block transfer as described in *"Setting Up and Running Block Transfers" on page 44*.

You can now view the captured traffic in the Protocol Analyzer and verify if the behavior of the PCI Express system in response to the memory write and memory read requests and the included error (incorrect LCRC) is correct.


Agilent E2960A Protocol Analyzer for PCI Express
File
Edit
View
Capture
Help
100%
Trigger Point
Memory Read Request splitted into the different chunk lengths of payload sent without an error

Dir.	Timestamp	Rel. Timestamp	Idle time	Transmission time	Packet No	Type	Address	Data	Length	Tag
↓	852	16	16	8	19	Ack				
↓	876	24		8	20	Nak				
⇧	1,228	352		16	21	Skip Ordered Se				
⇧	1,228	0		16	22	Skip Ordered Se				
⇧	1,228	0		16	23	Skip Ordered Se				
⇧	1,228	0	96	16	24	Skip Ordered Se				
⇧	1,340	112	24	8	25	UpdateFC-P				
⇧	1,372	32	8	8	26	UpdateFC-NP				
⇧	1,388	16	360	8	27	UpdateFC-Cpl				
⇧	1,756	368	12	20	28	Memory Read	00 FD 00		17	1F
⇧	1,788	32	0	20	29	Memory Read	00 FD 00		13	1C
⇧	1,808	20	0	20	30	Memory Read	00 FD 00		23	1B
⇧	1,828	20	0	20	31	Memory Read	00 FD 00		14	1A
⇧	1,848	20	0	20	32	Memory Read	00 FD 01		17	19
⇧	1,868	20	0	20	33	Memory Read	00 FD 01		13	18
⇧	1,888	20	0	20	34	Memory Read	00 FD 01		23	17
⇧	1,908	20	0	20	35	Memory Read	00 FD 01		14	16
⇧	1,928	20	0	20	36	Memory Read	00 FD 02		17	15
⇧	1,948	20	0	20	37	Memory Read	00 FD 02		13	14
⇧	1,968	20	0	20	38	Memory Read	00 FD 02		23	13
⇧	1,988	20	0	20	39	Memory Read	00 FD 02		14	12
⇧	2,008	20	0	20	40	Memory Read	00 FD 03		17	11
⇧	2,028	20	0	20	41	Memory Read	00 FD 03		13	10
⇧	2,048	20		20	42	Memory Read	00 FD 03		23	0F
↓	2,628	580	24	8	43	Ack				
↓	2,660	32		16	44	Skip Ordered Se				
↓	2,660	0		16	45	Skip Ordered Se				
↓	2,660	0		16	46	Skip Ordered Se				
↓	2,660	0	24	16	47	Skip Ordered Se				
↓	2,700	40	0	8	48	Ack				
↓	2,708	8	0	8	49	Ack				
↓	2,716	8	16	8	50	Ack				
↓	2,740	24	8	8	51	Ack				

Figure 40 **Analyzer with Captured Memory Read Requests**



	Dir.	Timestamp	Sequence Number	Rel. Timestamp	Idle time	Transmission time	Packet No	Type	Address	Data	Length
⊞	↑	-6,000	00 02		0	88	-72	Memory Write	00 FD 00	41 42	0 11
⊞	↑	-5,912	00 03	88	0	72	-71	Memory Write	00 FD 00	45 46	0 0D
⊞	↑	-5,840	00 04	72	0	112	-70	Memory Write	00 FD 00	79 7A	0 17
⊞	↑	-5,728	00 05	112	0	76	-69	Memory Write	00 FD 00	23 23	0 0E
⊞	↑	-5,652	00 06	76	44	88	-68	Memory Write	00 FD 01	00 00	0 11
⊞	↑	-5,520	00 07	132	0	72	-64	Memory Write	00 FD 01	00 00	0 0D
⊞	↑	-5,448	00 08	72	0	112	-63	Memory Write	00 FD 01	00 00	0 17
⊞	↑	-5,336	00 09	112	0	76	-59	Memory Write	00 FD 01	C9 F8	0 0E
⊞	↑	-5,260	00 0A	76	0	88	-58	Memory Write	00 FD 02	0C 08	0 11
⊞	↑	-5,172	00 0B	88	0	72	-57	Memory Write	00 FD 02	00 34	0 0D
⊞	↑	-5,100	00 0C	72		112	-56	Memory Write	00 FD 02	16 24	0 17
⊞	↓	-5,048	0 01	52		8	-55	Nak			
⊞	↑	-4,988	00 0D	60	0	76	-54	Memory Write	00 FD 02	01 62	0 0E
⊞	↑	-4,912	00 0E	76	0	88	-53	Memory Write	00 FD 03	60 28	0 11
⊞	↑	-4,824	00 0F	88	0	72	-52	Memory Write	00 FD 03	6D 19	0 0D
⊞	↑	-4,752	00 10	72	472	112	-51	Memory Write	00 FD 03	08 9E	0 17
⊞	↑	-4,168	00 02	584	0	88	-46	Memory Write	00 FD 00	41 42	0 11
⊞	↑	-4,080	00 03	88	0	72	-45	Memory Write	00 FD 00	45 46	0 0D
⊞	↑	-4,008	00 04	72	0	112	-44	Memory Write	00 FD 00	79 7A	0 17
⊞	↑	-3,896	00 05	112	0	76	-43	Memory Write	00 FD 00	23 23	0 0E
⊞	↑	-3,820	00 06	76	0	88	-42	Memory Write	00 FD 01	00 00	0 11
⊞	↑	-3,732	00 07	88	0	72	-41	Memory Write	00 FD 01	00 00	0 0D
⊞	↑	-3,660	00 08	72	0	112	-40	Memory Write	00 FD 01	00 00	0 17
⊞	↑	-3,548	00 09	112	96	76	-39	Memory Write	00 FD 01	C9 F8	0 0E
⊞	↑	-3,376	00 0A	172	0	88	-31	Memory Write	00 FD 02	0C 08	0 11
⊞	↑	-3,288	00 0B	88		72	-27	Memory Write	00 FD 02	00 34	0 0D
⊞	↓	-3,224	0 02	64		8	-26	Ack			
⊞	↑	-3,216	00 0C	8		112	-25	Memory Write	00 FD 02	16 24	0 17
⊞	↓	-3,144	0 03	72		8	-24	Ack			
⊞	↑	-3,104	00 0D	40		76	-23	Memory Write	00 FD 02	01 62	0 0E
⊞	↓	-3,032	0 04	72		8	-22	Ack			
⊞	↑	-3,028	00 0F	4		88	-21	Memory Write	00 FD 03	60 28	0 11

Exerciser sends the Memory Write Request with incorrect LCRC

PCI Express system responds with a Nak on the last correct packet

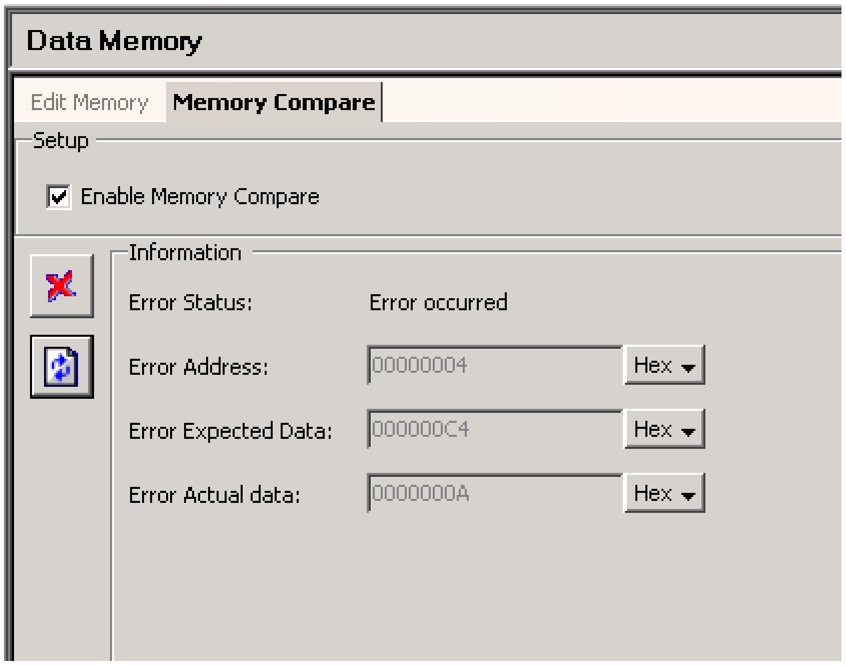
Exerciser sends the packets with correct LCRC

Figure 41 Protocol Analyzer with Captured Memory Write Requests

As you can see, the PCI Express system responds correctly.

Additionally, in the Protocol Exerciser software, open the *Data Memory* and click the *Memory Compare* tab.

The Exerciser compares the original data with the data read from the PCI Express system and shows you the error address, the expected data and the actual data. See the figure below for an example of an incorrect data transfer.



☑ Enable Memory Compare

Error Status: Error occurred

Error Address: 00000004 Hex

Error Expected Data: 000000C4 Hex

Error Actual data: 0000000A Hex

Figure 42 Memory Compare with Detected Error

What Happens at the Exerciser when transferring a Block?

The Exerciser also operates as a requester for this task. The following figure illustrates the Exerciser components involved.

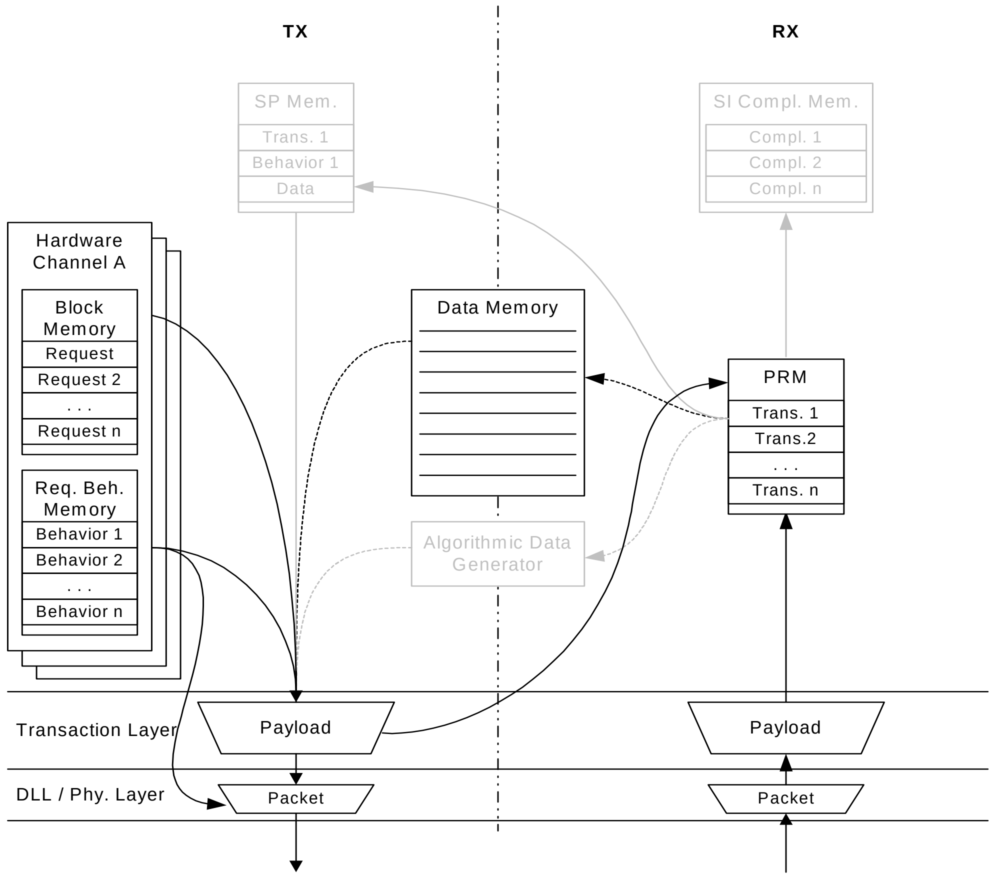


Figure 43 **Exerciser as Requester (transferring a block)**

The desired requests and request behaviors are written to the Block Memory (BM) and Request Behavior Memory (RBM) for a particular Hardware Channel. The required data is written to the Data Memory.

When you transfer a block, the requests are processed one at a time, taking the necessary data from the Data Memory, and according to the current request behavior.

Block transfers also use the Pending Request Map (PRM) to keep track of open responses and which memory areas they should access. For block transfers, the received completions are *not* stored.

See *“Exerciser as Requester” on page 62* for a detailed description of how the Exerciser acts as a requester.

Bringing Up and Debugging a PCI Express Add-In Card

In the early stages of the bring up and debug of your prototype, you need to assess its behavior and root out the causes of errors and performance problems.

The Protocol Exerciser and the Protocol Analyzer for PCI Express helps you to verify the physical and data link layer capabilities of a PCI Express add-in card as well to simulate such add-in cards on the transaction layer.

Task

This test provides an example to check if the PCI Express add-in card's Physical layer functions correctly by modifying the Protocol Exerciser's link settings. After the link setting scrambling has been disabled, link training will be performed to see if the add-in card changes its scrambling mode as well.

The Protocol Analyzer is set up to capture the traffic sent between the Protocol Exerciser and the PCI Express add-in card.

System Setup

The following system setup is required for this test:

- The add-in card to be tested is plugged into the PCI Express slot at the top of the probe board set up as Analyzer.
- The Analyzer probe board is plugged into the PCI Express slot at the top of the probe board set up as Exerciser.
- An external ATX power supply is connected to one probe board. The power jumper on the probe board should be set to **Ext** (instead **sys**).

  The external power supply is required to power the add-in card. Alternatively, you could plug the probe board into a powered PCI Express slot and set the jumper to **sys**.

The system setup is as follows:

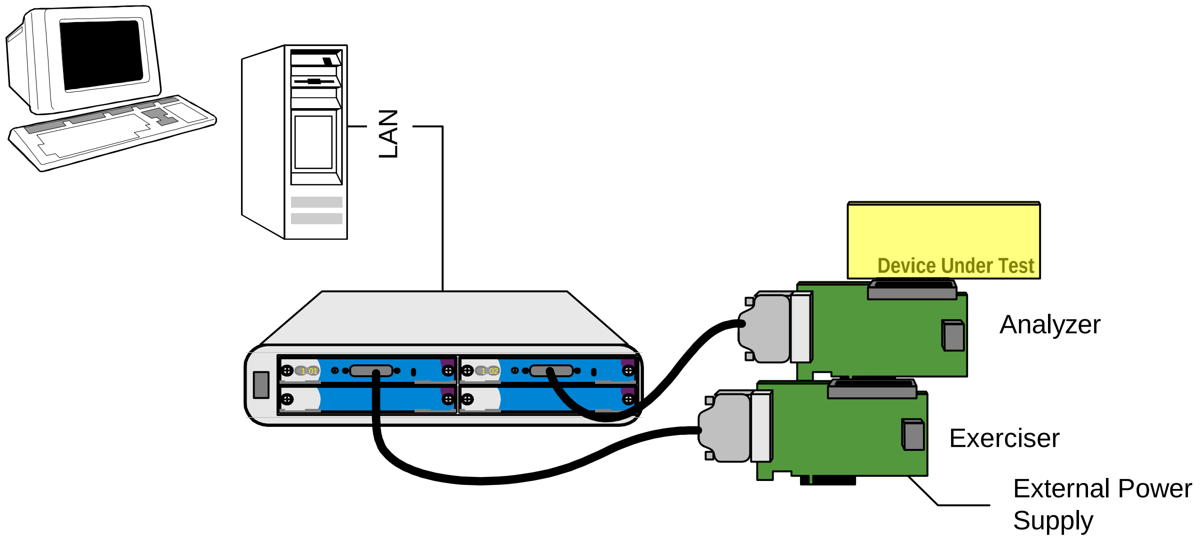


Figure 44　Protocol Exerciser and Analyzer Setup for Testing a PCI Express Add-In Card

How to Proceed This test is carried out as follows:

1. We have to establish an Exerciser session (described in *"Establishing an Exerciser Session" on page 19*) and an Analyzer session at the client.

   See *"Establishing an Analyzer Session" on page 31*.

2. We set up and establish a PCI Express link between the Exerciser and the add-in card.

   See *"Establishing a PCI Express Link" on page 23*.

3. We set up a trigger on the Analyzer so that the Analyzer captures traffic around a specific link state.

   See *"Setting Up a Trigger" on page 33*.

4. We set up the Exerciser with disabled scrambling.

   See *"Modifying Specific Link Settings" on page 59*.

5. We start the Analyzer and view the captured traffic.

   See *"Running the Test and Viewing the Results for the Bring Up and Debug Test" on page 60*.

TCL Script The following TCL script corresponds to the requirements of this example:

- *"Modifying Link Settings on the Exerciser" on page 89*

Setting Up the Trigger for the Bring Up and Debug Test

In this example, we initiate link training with disabled scrambling and check if the PCI Express add-in card disables scrambling as well.

In the user interface, we must first set up the Analyzer so that it triggers upon a link state with disabled scrambling.

To set up the trigger with the user interface:

1 In the Analyzer user interface, open the *Trigger Setup* dialog box.

2 Under *Conditions*, open the *Link Patterns* folder, select *TS1*, and click *Copy*.

The condition is saved as a copy under *User Patterns*.

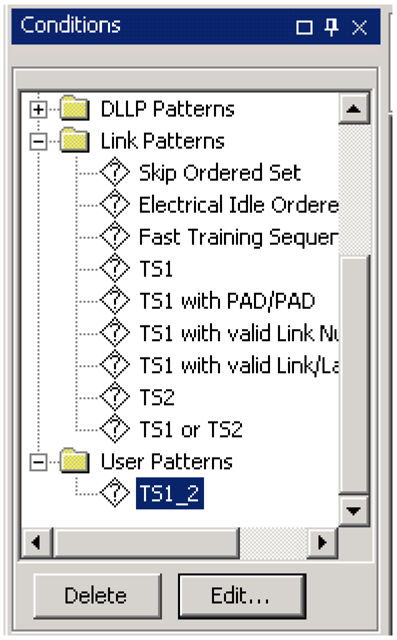
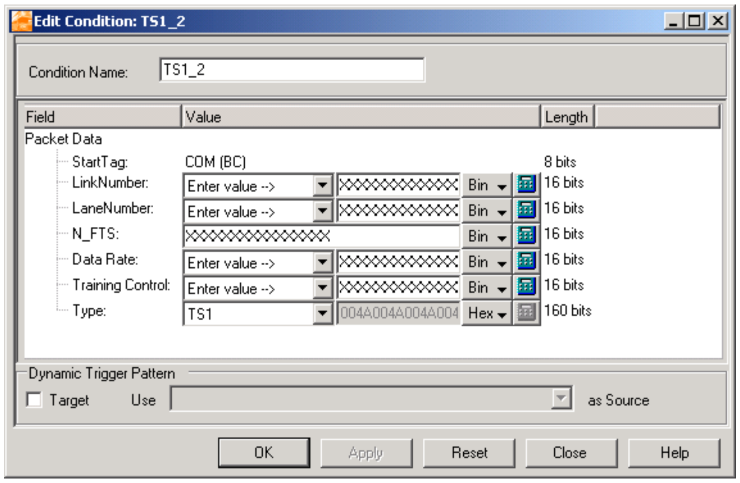


Figure 45 Trigger Setup Dialog Box

We have to define a condition with disabled scrambling and use this as the trigger. We can do this by creating and editing a copy of a condition and loading this new condition.

3 Double-click the copied condition to edit it.

The *Edit Condition* dialog box opens.

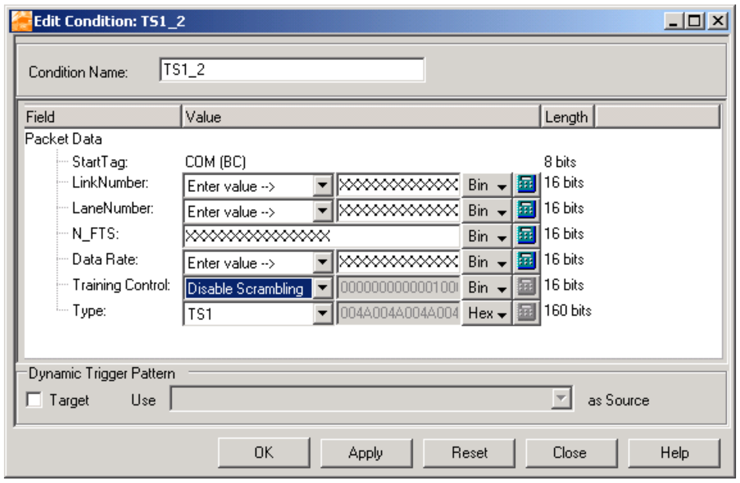

Edit Condition: TS1_2

Condition Name: TS1_2

Field	Value				Length
Packet Data					
StartTag:	COM (BC)				8 bits
LinkNumber:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
LaneNumber:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
N_FTS:	XXXXXXXXXXXXXXXX		Bin		16 bits
Data Rate:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
Training Control:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
Type:	TS1	004A004A004A004	Hex		160 bits

Figure 46 Edit Condition dialog box

4 Set *Training Control* to *Disable Scrambling*.


Edit Condition: TS1_2

Condition Name: TS1_2

Field	Value				Length
Packet Data					
StartTag:	COM (BC)				8 bits
LinkNumber:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
LaneNumber:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
N_FTS:	XXXXXXXXXXXXXXXX		Bin		16 bits
Data Rate:	Enter value -->	XXXXXXXXXXXXXXXX	Bin		16 bits
Training Control:	Disable Scrambling	000000000000100	Bin		16 bits
Type:	TS1	004A004A004A004	Hex		160 bits

Figure 47 Edit a Condition

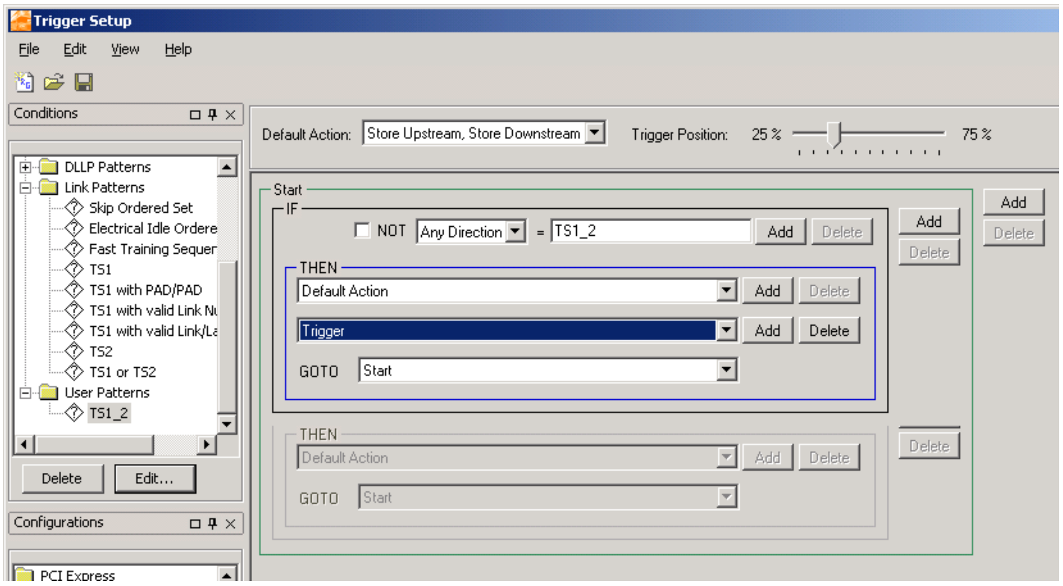
5 Close the *Edit Condition* dialog box and drag the modified condition into the *Condition* field.

6 Set the *Default Action* to *Store Upstream*, *Store Downstream*.

7 Add an action by clicking *Add* next to the condition field and set the first action field to *Default Action* and the second to *Trigger*.

The Analyzer is now set up to view the link training in both directions (upstream and downstream), and to store 25% of the traffic that occurs before the trigger point and 75% after the trigger point.

The *Trigger Setup* should now appear as follows:



Default Action: Store Upstream, Store Downstream
Trigger Position: 25 % 75 %

IF: NOT [ ] Any Direction = TS1_2
THEN: Default Action
Trigger
GOTO: Start

THEN: Default Action
GOTO: Start

Figure 48 **Trigger Setup for the Bring Up and Debug Test**

8 Save the trigger configuration, for example, to "*test_trigger_on_disscrambler.trg*".

9 Apply the trigger and close the *Trigger Setup*.

10 In the User Interface, select *Run* in the *Capture* menu to start the Analyzer.

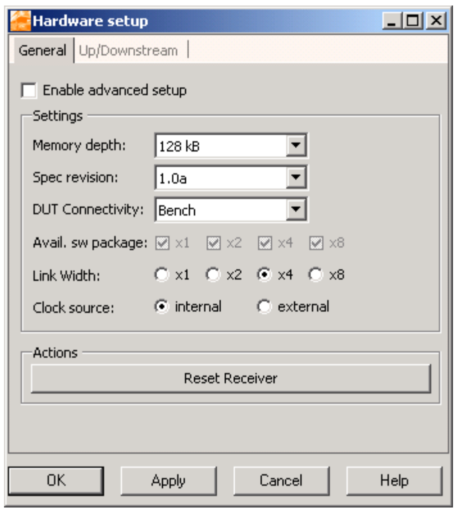
The Protocol Analyzer now tracks the traffic sent in both directions through the bus. If any *TS1* with disabled scrambling is detected, the Analyzer triggers and stores 25% of the link states before the trigger point and 75% after the trigger point.

Disabling Scrambling an the Analyzer

To avoid that the Analyzer shows all-red packets after starting the link training, you can set the analyzer to disable the scrambling as well:

1 Click *Hardware Setup* in the *Capture* menu.

This opens the *Hardware Setup* dialog box with the *General* tab on top.



Hardware setup

General | Up/Downstream

☐ Enable advanced setup

Settings

Memory depth:	128 kB
Spec revision:	1.0a
DUT Connectivity:	Bench
Avail. sw package:	☑ x1 ☑ x2 ☑ x4 ☑ x8
Link Width:	x1 x2 [✓ x4] x8
Clock source:	[✓ internal] external

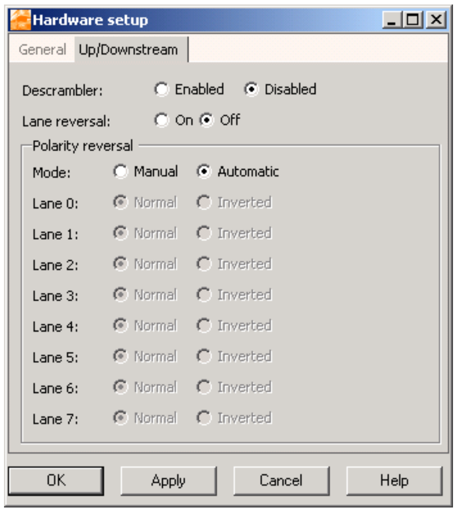
Actions

Reset Receiver

OK Apply Cancel Help

Figure 49 Hardware Setup Dialog Box

2 Select the Up/Downstream tab and set *Descrambler* to *Disabled*.



Hardware setup

General | Up/Downstream

Descrambler:	Enabled [✓ Disabled]
Lane reversal:	On [✓ Off]

Polarity reversal

Mode:	Manual [✓ Automatic]
Lane 0:	[✓ Normal] Inverted
Lane 1:	[✓ Normal] Inverted
Lane 2:	[✓ Normal] Inverted
Lane 3:	[✓ Normal] Inverted
Lane 4:	[✓ Normal] Inverted
Lane 5:	[✓ Normal] Inverted
Lane 6:	[✓ Normal] Inverted
Lane 7:	[✓ Normal] Inverted

OK Apply Cancel Help

Figure 50 Hardware Setup Dialog Box with Disabled Descrambler

Modifying Specific Link Settings

On the Protocol Exerciser, several link settings can be modified to analyze the physical layer capability of the PCI Express add-in card.

In this example, the scrambler will be disabled.

To disable the scrambler:

1 Select the *General* icon on the Exerciser.

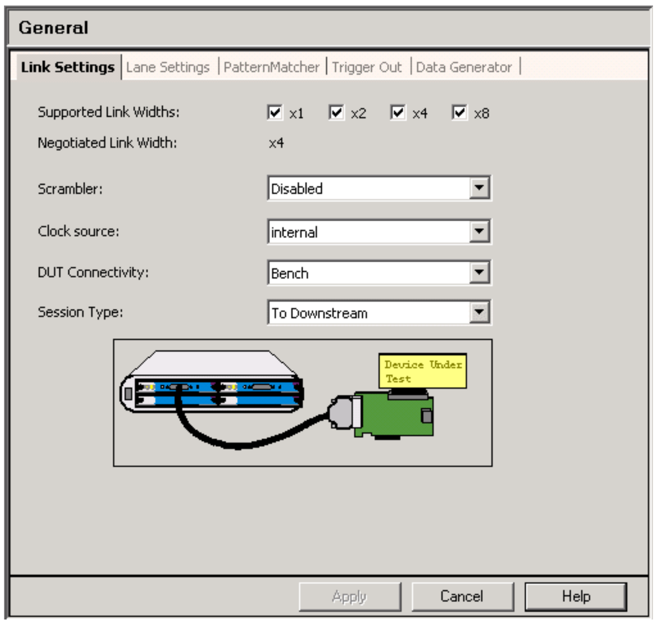
2 Disable the *Scrambler*.

This forces the Protocol Exerciser to disable the scrambler. As soon as another device requests the disabling, the Exerciser reacts correctly.

3 Because an add-in card is tested, select *Bench* for the *DUT Connectivity*.

4 Apply the settings.

The *Link Settings* should now appear as follows:



Supported Link Widths: ✓ x1 ✓ x2 ✓ x4 ✓ x8
Negotiated Link Width: x4
Scrambler: Disabled
Clock source: internal
DUT Connectivity: Bench
Session Type: To Downstream

Figure 51 Modified Link Settings

NOTE Do not initiate the link training yet. We will do this in *“Running the Test and Viewing the Results for the Bring Up and Debug Test” on page 60.*

Running the Test and Viewing the Results for the Bring Up and Debug Test

The Analyzer is set up to trigger on the first link state with disabled scrambling during the link training.

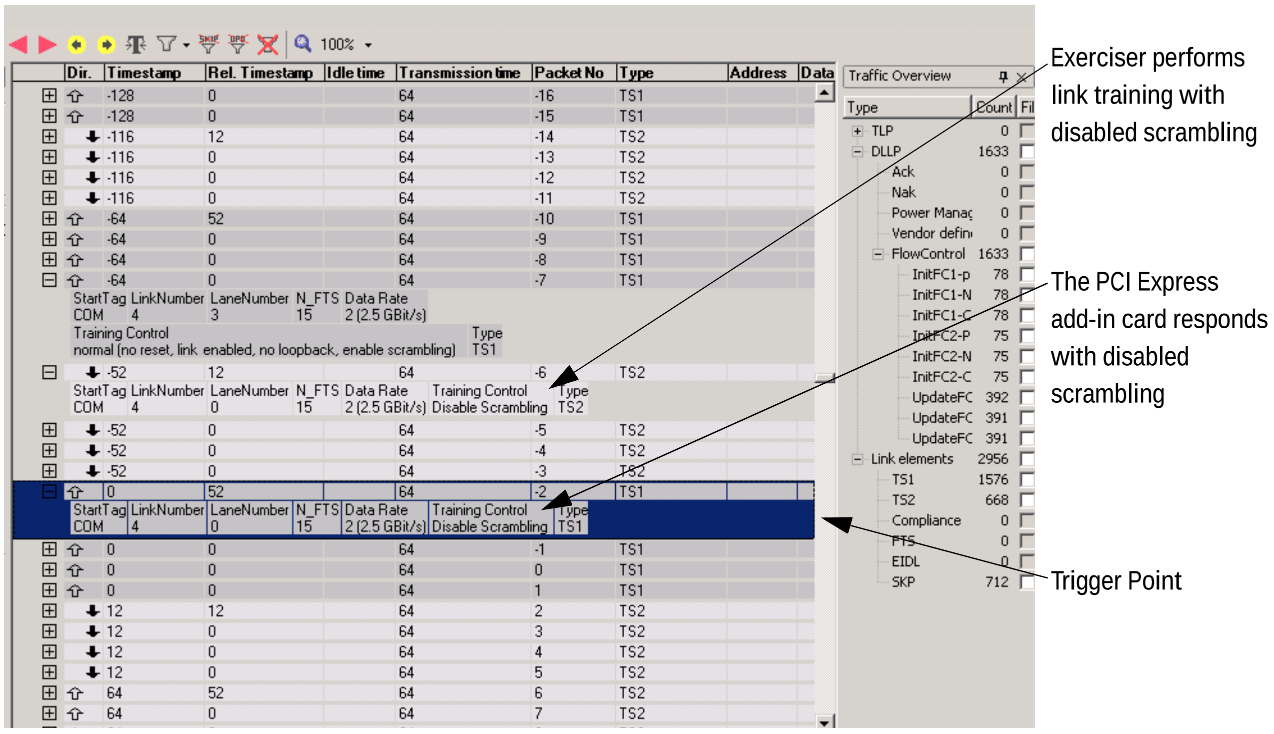
To run the test and view the results:

1. In the Protocol Analyzer software, click *Start* in the *Capture* menu.

   The Protocol Analyzer now tracks the link states sent through the bus.

2. In the Protocol Exerciser user interface, initiate the link training by clicking *Link Up* in the *Action* menu.

Once the link has been initiated, you can open the Protocol Analyzer to view the captured traffic captured around the first link state with disabled scrambling.


100%
Exerciser performs link training with disabled scrambling
The PCI Express add-in card responds with disabled scrambling
Trigger Point

```
Dir. Timestamp Rel. Timestamp Idle time Transmission time Packet No Type Address Data
 ⇧   -128      0                        64                -16       TS1
 ⇧   -128      0                        64                -15       TS1
 ⬇   -116      12                       64                -14       TS2
 ⬇   -116      0                        64                -13       TS2
 ⬇   -116      0                        64                -12       TS2
 ⬇   -116      0                        64                -11       TS2
 ⇧   -64       52                       64                -10       TS1
 ⇧   -64       0                        64                -9        TS1
 ⇧   -64       0                        64                -8        TS1
 ⇧   -64       0                        64                -7        TS1
     StartTag LinkNumber LaneNumber N_FTS Data Rate
     COM      4          3          15    2 (2.5 GBit/s)
     Training Control                                               Type
     normal (no reset, link enabled, no loopback, enable scrambling) TS1
 ⬇   -52       12                       64                -6        TS2
     StartTag LinkNumber LaneNumber N_FTS Data Rate        Training Control    Type
     COM      4          0          15    2 (2.5 GBit/s)   Disable Scrambling  TS2
 ⬇   -52       0                        64                -5        TS2
 ⬇   -52       0                        64                -4        TS2
 ⬇   -52       0                        64                -3        TS2
 ⇧   0         52                       64                -2        TS1
     StartTag LinkNumber LaneNumber N_FTS Data Rate        Training Control    Type
     COM      4          0          15    2 (2.5 GBit/s)   Disable Scrambling  TS1
 ⇧   0         0                        64                -1        TS1
 ⇧   0         0                        64                0         TS1
 ⇧   0         0                        64                1         TS1
 ⬇   12        12                       64                2         TS2
 ⬇   12        0                        64                3         TS2
 ⬇   12        0                        64                4         TS2
 ⬇   12        0                        64                5         TS2
 ⇧   64        52                       64                6         TS2
 ⇧   64        0                        64                7         TS2
```

```
Traffic Overview
Type                 Count
TLP                  0
DLLP                 1633
  Ack                0
  Nak                0
  Power Manag        0
  Vendor defin       0
  FlowControl        1633
    InitFC1-p        78
    InitFC1-N        78
    InitFC1-C        78
    InitFC2-P        75
    InitFC2-N        75
    InitFC2-C        75
    UpdateFC         392
    UpdateFC         391
    UpdateFC         391
Link elements        2956
  TS1                1576
  TS2                668
  Compliance         0
  FTS                0
  EIDL               0
  SKP                712
```

Figure 52 Protocol Analyzer with Captured Link States

As you can see, the PCI Express add-in card responds correctly.

Exerciser Architecture Overview

This chapter provides in-depth information into the Exerciser architecture. The information here is essential for understanding the internal structures of the Serial I/O Modules when they are set up as Exercisers.

NOTE The Analyzer architecture is simple and is therefore not explained in detail in this document.

This chapter explains how the Protocol Exerciser works as:

- A requester

  As requester, the Exerciser requests data to be tranferred to or from the add-in card or PCI Express system under test and then waits for the completion.

- A completer

  As completer, the Exerciser responds to requests from the add-in card or PCI Express system under test.

Additionally, a short functional description of all Exerciser components is provided.

NOTE Typically, the Exerciser will function both as a requester and completer in any test. This depends on the device or system being tested.

Exerciser as Requester

As requester, the Exerciser requests the add-in card or PCI Express system under test to perform certain actions (for example, to read to or write from its memory). The following diagram illustrates what typically happens at the Exerciser in this case.

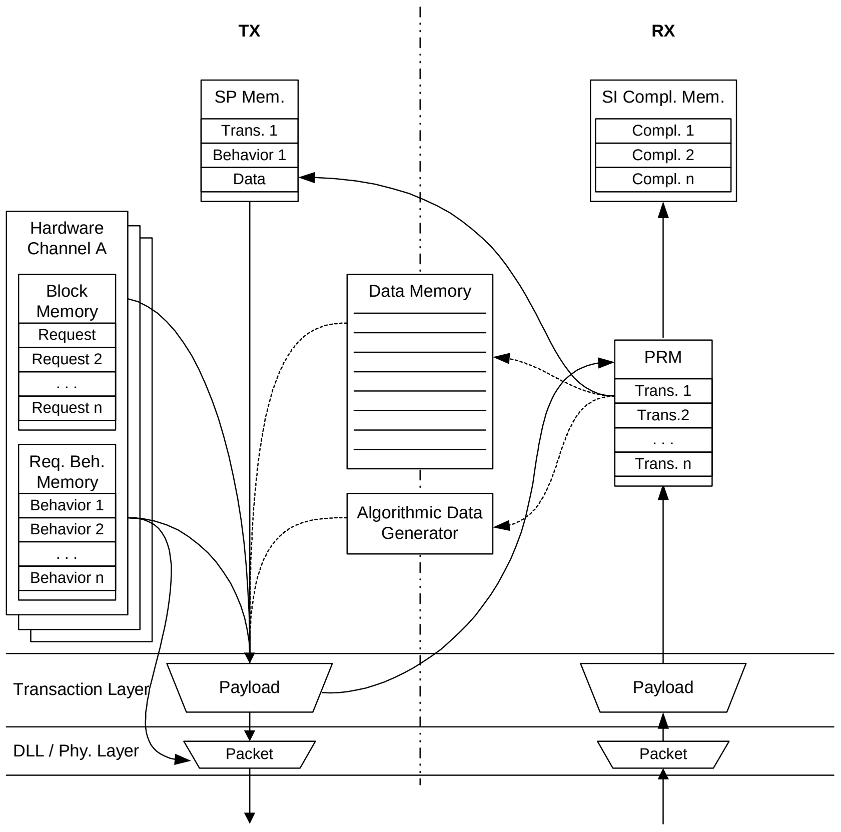


Figure 53 Exerciser as Requester

As indicated in this figure, there are three hardware channels (A, B, and C). Each of these hardware channels has its own Block Memory and Requester Behavior Memory (Req. Beh. Memory).

NOTE The Request Behavior Memory acts as a loop. It contains any number of request behaviors that it uses progressively. There is no direct correlation between the number of entries in a hardware channel's Request Behavior Memory and the number of entries in its Block Memory.

The Exerciser's workflow as requester for a Block Transfer is as follows.

1. The requests to be carried out are written to one of the three Hardware Channel's Block Memories.

   User Interface: Send Block Transfers page, *Request* section

   Programming: Exerciser's **Block Setup** functions

2. The data memory is filled with data as necessary.

   User Interface: Data Memory page

   Programming: Exerciser's **Data Memory Access** functions

3. The required behaviors are written to the hardware channel's Requester Behavior Memory.

   User Interface: Send Block Transfers page, *Request Behavior* section

   Programming: Exerciser's **Behavior Setup** functions

4. The block transfer is started.

   User Interface: *Action* menu, *Run* item

   Programming: Exerciser's **Run Control** functions

5. When the block transfer is started, the packets are assembled on the Transaction Layer according to the current request, the current entry in the behavior memory, and any data required from the Data Memory or the Algorithmic Data Generator (as defined in the request).

   For each request that expects a response (for example, for a data read), an entry is written to the Pending Request Map (PRM). This entry contains the memory location for any data to be returned. Note that no new requests are sent while the Pending Request Map is full (because too many requests are pending).

   The PRM has either 32 or 256 entries, depending on the extended tag mode (for more information, see the PCI Express Specification, Extended Tag Field Enable bit, section 7.8.4).

   This is done internally and requires no additional effort.

6. The packets are then sent to the data link layer (DLL) for transmission. The data link layer adds the sequence number, LCRC and framing to the packets and then sends them.

If the current behavior specifies that the packet is to be sent with an error that forces a NAK (for example, wrong LCRC), the packet is sent once with the error and when the add-in card or PCI Express system under test returns a NAK (negative acknowledge), it is sent without the error. Other errors, like nullified TLP or wrong length, are not checked in the DLL and therefore do not result in an automatic (correct) replay.

This is done internally and requires no additional effort.

7. If the request requires a response, when the response is returned, the entry is looked up in the Pending Requests Map (PRM). Any returned data is written to the memory location or the Comparison Algorithm (specified when the request was created).

   This is done internally and requires no external effort.

NOTE If the add-in card or PCI Express system under test does not respond to a request, the entry in the PRM is not removed. If too many requests are ignored (because the add-in card or PCI Express system under test cannot handle certain errors added to the packets), the PRM will fill up, and no further requests will be allowed. In such a case, you have to retrain the link to clear the PRM.

The workflow for *Send Single Packet* differs from the above as follows:

- The requests and behaviors for the transactions are written to the Single Packet Memory (SP Memory). The data can be written to either the Single Packet Memory or the Data Memory (as configured in the request).
- When a response containing data is returned, it can be written to either the Single Packet Memory or the Data Memory (as configured in the request).
- Received completions are written to the Send Immediate Completion memory (SI Compl. Mem.) instead of being discarded. This gives you the possibility of inspecting the packets’ headers.

User Interface: *Send Single Packet* page

Programming: Exerciser’s **SI Control** (Send Immediate) functions

Exerciser as Completer

When the Exerciser acts as completer, it responds to requests sent by the add-in card or PCI Express system under test (for example, a memory read). The following figure illustrates the various components of an Exerciser in this case.

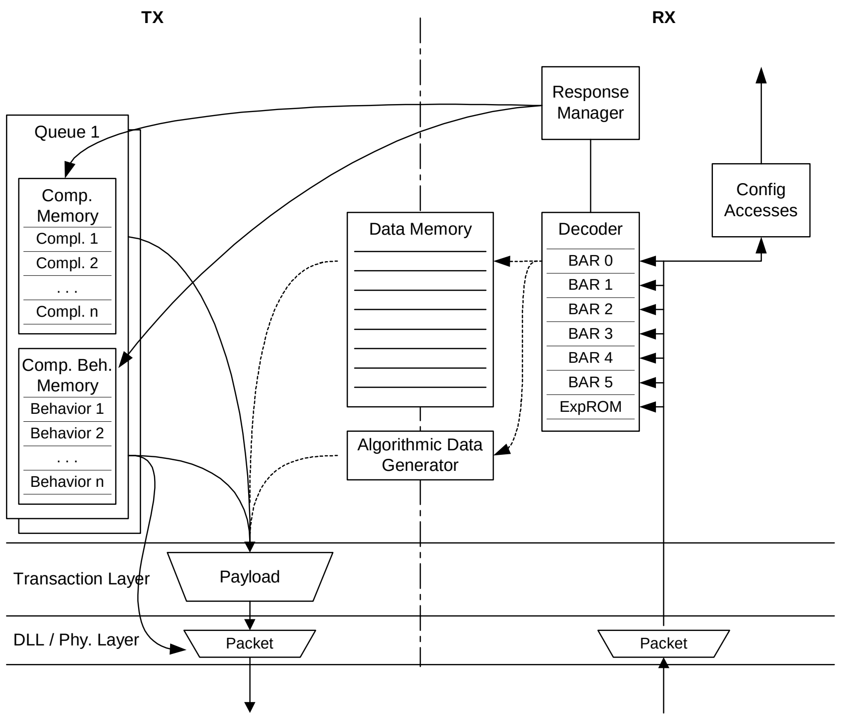


Figure 54 Exerciser as Completer

As indicated in this figure, there are two queues. Both of these queues have its own Completion Memory (Comp. Memory) and Completion Behavior Memory (Comp. Beh. Memory). The Queue used depends on the settings in the Decoder.

NOTE A Queue's Completion Behavior Memory acts as a loop. It contains any number of completion behaviors that it uses progressively. There is no direct correlation between the number of entries in a Queue's Completion Behavior Memory and the number of entries in its Completion Memory.

The Exerciser’s workflow as completer is as follows:

1. The Decoder is set up as necessary to configure access to the I/O area, data memory, or data generator.

   If no BIOS is available, the Decoder has to be configured by the user.

   User Interface: Decoder page

   Programming: Exerciser’s **Configuration Space Control** functions

2. The desired completion behaviors for each Queue are written to the corresponding Completion Behavior Memory.

   User Interface: Completion page

   Programming: Exerciser’s **Behavior Setup** functions

3. The Exerciser receives a request from the add-in card or PCI Express system under test.

4. If the request requires access to the configuration space, it is passed to the embedded software running on the Serial I/O Module.

   If the request requires access to the memory, the memory area to be accessed is defined according to the BARs and the expansion ROM.

   This is done internally and requires no additional effort.

5. The Response Manager writes the completion necessary for responding to the request in the Completion Memory of the corresponding Queue.

   This is done internally and requires no additional effort.

6. The packets are assembled on the Transaction Layer according to the completion and the current behavior and then passed to the data link layer for sending.

   This is done internally and requires no additional effort.

7. If the current behavior specifies that the packet is to be sent with an error that forces a NAK (for example, wrong LCRC), tha packet is sent once *with* the error and a NAK (negative acknowledge) is returned, it is sent *without* the error.

   This is done internally and requires no additional effort.

Exerciser Components

As described in *“Exerciser as Requester” on page 62* and *“Exerciser as Completer” on page 65*, the Exerciser consists of various components that work together. This section provides an functional overview of these elements (in alphabetical order).

Algorithmic Data Generator

The algorithmic data generator is used to generate semi-random data. This data is written to the add-in card or PCI Express system under test. When it is read, the returned data is compared to the expected data. Thus, large memory access tests can be carried out, despite the memory restrictions of the Serial I/O Modules.

Block Memory

The block memory is used to hold the information necessary to build up a request transaction: it holds for example the transaction type, a pointer to the data memory for the data to be included, and the information for the bus to be used.

Completion Behavior Memory

The completion behavior memory contains a series of entries that describe how the completions are to be assembled (for example, maximum payload length, number of times the transaction is to be repeated without incrementing the transaction number, or any errors that should be included when the packet is assembled.

Completion Memory

Similar to the block memory, the response memory is a FIFO list of the completions to be processed for a particular queue. These responses are used to generate the payloads and packets to be sent to the requester.

Data Memory

The data memory is a 2-MByte volatile memory space on the Serial I/O Modules that is used to temporarily store data. This data can either be used for writes to or reads from a device.

The data in the data memory can be imported from files, or exported to files. There are no special formatting requirements on this data.

Accesses to memory areas up to 8 EBytes (2^63 bytes) can be set up in the decoder. The available data memory is wrapped around to supply this amount of memory space.

Decoder

The decoder is used to configure the access to the Exerciser’s configuration space (BAR0 – BAR5, expansion ROM). It provides the necessary data locations for accesses to the configuration space.

Hardware Channels

The three hardware channels use their own blocks in the Serial I/O Module’s memory to manage the channel’s requests (the Request Block Memory) and request behaviors (Behavior Memory). The hardware channels are further divided into Virtual Channels.

Pending Request Map

When the Exerciser sends a request to which it expects a response (for example, a memory read), an entry is written to the pending request map to keep track of the responses to be expected (so that the Exerciser can handle them as necessary).

Queue

Two queues are implemented for handling responses to requests. These queues both have a block memory (for holding the requests) and a request behavior memory (for describing how the requests are to be processed).

It is recommended to use one queue for handling the “normal” traffic and to keep one queue free to handle high-priority traffic.

Request Behavior Memory

The request behavior memory contains a series of entries that describe how the requests are to be assembled (for example, maximum payload length, number of times the transaction is to be repeated without incrementing the transaction number, or any errors that should be included when the packet is assembled.

Request Memory

The request memory is a FIFO list of the requests to be processed for a particular hardware channel. These requests are used to generate the payloads and packets to be sent.

Single Packet Memory

The single packet memory is almost identical to the memory for a hardware channel: besides a Request Memory and Request Behavior Memory, it also contains its own Data Memory.

Virtual Channels

The Exerciser has three physical hardware channels but has been designed to emulate up to eight virtual channels. Hardware channel A and B are mapped directly to virtual channels 0 and 1, respectively. Virtual channels 2 – 7 are emulated on hardware channel C.

How to Program the Exerciser and Analyzer

The following sections provide you with some basic information that helps you familiarize yourself with developing programs for the Agilent Protocol Exerciser and Protocol Analyzer:

- *"Using the DCOM API" on page 70*

  Provides a class overview of the DCOM API and describes how the classes and interfaces interact. See the *E2960 PCI Express Series API Reference* in the online Help for details.

- *"Controlling the Exerciser from a SUT" on page 73*

  Provides the information you need when developing a program for controlling the Exerciser over the PCI Express bus from the system under test.

Using the DCOM API

The following figure provides a short overview of how the PCI Express classes and interfaces are related.

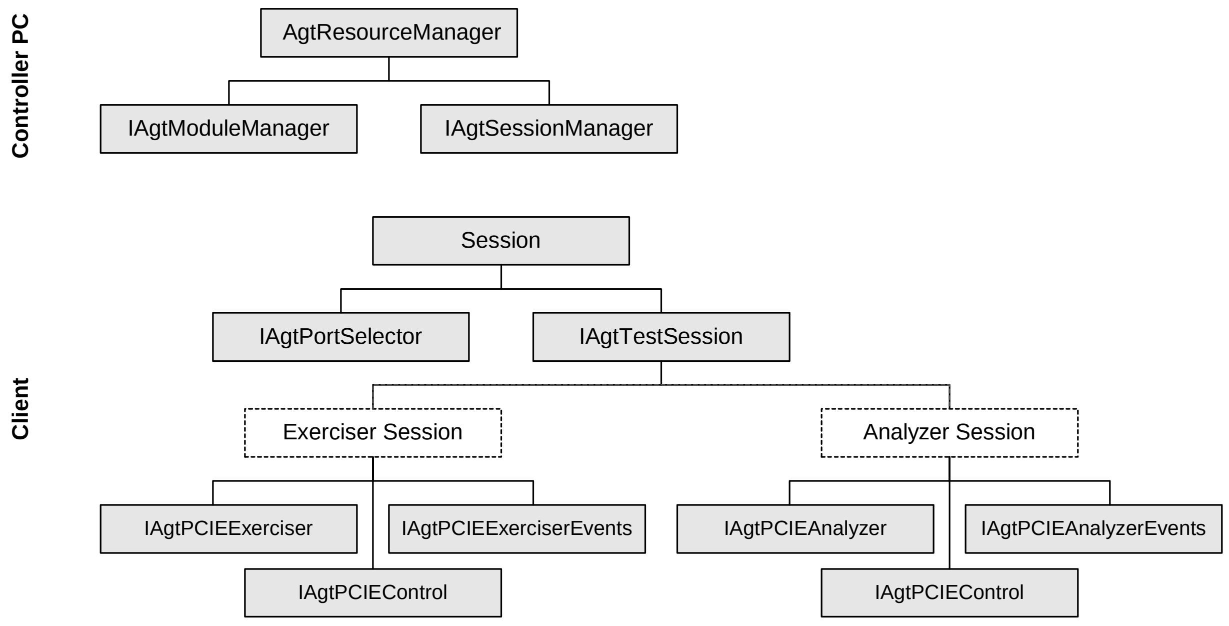


Figure 55 Overview of PCI Express Classes

The Resource Manager (AgtResourceManager) runs on the controller PC. The Session Manager (IAgtSessionManager) and Module Manager (IAgtModuleManager) are interfaces of the Resource Manager.

The Session Manager manages all sessions (number of sessions and their types), the Module Manager manages the hardware (module numbers, module types).

Opening a New Session

You would use the following commands to open a new session from a DCOM-based program. Use this description in conjunction with the *E2960 PCI Express Series API Reference* in the online Help to get started in creating DCOM-based programs.

- Create an instance of the Test Session to the controller PC:

```
#define HOSTNAME L"localhost"
COSERVERINFO serverInfo = {0, HOSTNAME, NULL, 0};
MULTI_QI mqi;
mqi.hr = 0;
mqi.pItf = NULL;
mqi.pIID = &(__uuidof(IAgtTestSession));
HRESULT hRes =
HRESULT hRes = CoCreateInstanceEx(__uuidof(AgtTestSession), NULL,
CLSCTX_REMOTE_SERVER, &serverInfo, 1, &mqi);
  if (FAILED(hRes))
  {_com_error e(hRes);
  throw e;
  }
}
IAgtTestSessionPtr pTestSession = mqi.pItf;
```

- Open a session with the necessary configuration:

```
long sessionHandle =
    pTestSession->OpenSession("PCIEExerciserToDownstream",
        AGT_SESSION_ONLINE);
```

- Get the Port Selector from the Test Session.

```
IAgtPortSelectorPtr pPortSelector =
    pTestSession->GetInterfaceByName("AgtPortSelector");
```

- Add a port to the session:

```
long portHandle;
pPortSelector->AddPort (101, 1, portHandle);
```

- Get the session’s Exerciser interface:

```
    IAgtPCIEExerciserPtr pExerciser =
        pTestSession->GetInterfaceByName("AgtPCIEExerciser");
```

- And when you are finished, close the session:

```
pTestSession->CloseSession()
```

Accessing a Running Session

You would use the following commands to access running sessions from a DCOM-based program. Use this description in conjunction with the *E2960 PCI Express Series API Reference* in the online Help to get started in creating DCOM-based programs.

- Create an instance of the Session Manager to the controller PC:

```
#define HOSTNAME L"localhost"
AGTRESOURCEMANAGERLib::IAgtSessionManagerPtr pSessionManager;
COSERVERINFO serverInfo = {0, HOSTNAME, NULL, 0};
MULTI_QI mqi;
mqi.hr = 0;
mqi.pItf = NULL;
mqi.pIID = &(__uuidof(AgtSessionManager));
HRESULT hRes =
    CoCreateInstanceEx(__uuidof(AgtSessionManager),
        NULL, CLSCTX_REMOTE_SERVER, &serverInfo, 1, &mqi);
if (FAILED(hRes))
{    _com_error e(hRes);
    throw e;
}
pSessionManager = mqi.pItf;
```

- Use the instance to get a list of the handles of the open sessions:

```
long count;
SAFEARRAY *psa;
psa = SafeArrayCreateVector (VT_UI4,0,0);
pSessionManager->ListOpenSessions(&count,&psa);
```

- Check the session type using the session handles:

```
BSTR test = pSessionManager->GetSessionType(1);
```

- If the session type corresponds to the required session, use the session handle to get the Test Session:

```
IAgtTestSessionPtr pTestSession =
pSessionManager->GetSessionInterface(1);
```

- Get the Exerciser interface:

```
IAgtPCIEExerciserPtr pExerciser =
    ppSessionInterface->GetInterfaceByName("AgtPCIEExerciser");
```

- And when you are finished, close the session:

```
ppSessionInterface->CloseSession()
```

Controlling the Exerciser from a SUT

The Protocol Exerciser features an in-system PCI Express port that can be used for controlling its behavior over the PCI Express bus. This allows programs running on the system under test (SUT) to request actions from the Exerciser over PCI Express, thus eliminating the need for a separate LAN connection on the SUT.

The requests are sent as PCI Express packets that contain Exerciser commands or requests as data. The Exerciser can then carry out the necessary actions and send replies to the requester using PCI Express.

This type of setup is required if you have a test program that runs on a system under test (SUT) that requests actions from the Exerciser.

The following figure shows the software structure with the PCIe port API:

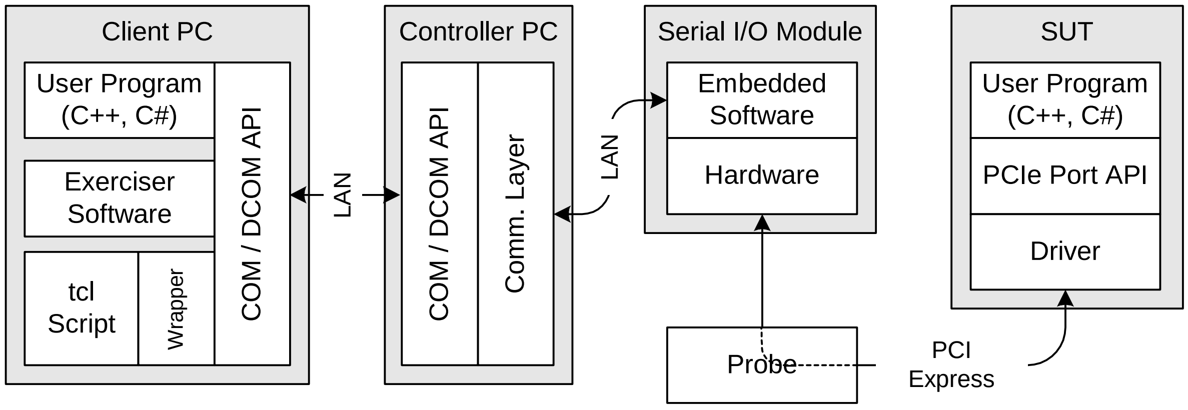


Figure 56 Software Structure for Testing with a System Under Test

Differences between the PCIe Port API and DCOM API

There are several differences in how the PCIe port API is used compared to the DCOM API:

- The PCIe port API does not use the COM interface.

  Therefore, the COM initialization calls (CoComInitialize, etc.) that are required with the DCOM API are not required.

- No configuration of the session is possible over the PCIe port API.

  The System I/O Module must already be set up as Exerciser, the session must be started, the required port(s) must be added, and the link must be brought up before the PCIe port API can be used.

- The system performance is generally slower than if the Exerciser is controlled over LAN.

  This is due to the fact that the PCI Express bus has additional traffic for the communication with the requester.

- Testing on the DLL/Phy layer cannot be performed.

  Any manipulation of the link causes the PCIe port API to lose the connection to the requester.

- Event notifications are not supported by the PCIe port API.

- The port handle must be obtained over the port's physical address.

  Most DCOM API functions use the port handle to reference to a particular port within a session. The port handle is typically derived from a call to `PortSelector->AddPort`. The PCIe port API provides a new function to get the port handle:

  `ConnectPort(physicalAddress, &porthandle)`

Working with the PCIe Port API

The PCIe port API provides two classes required for sending commands to an Exerciser over the in-system port:

- CAgtPortSelector

  Contains functions for opening and closing the connection to PCI driver.

- CAgtPCIEExerciser

  Provides all exerciser API functions needed for communicating with the Serial I/O Modules.

These modules are described in more detail below.

CAgtPortSelector This class is required to establish a connection to the in-system port on the Serial I/O Module. It is used to get the port handle of the in-system port, which is required when calling any CAgtPCIEExerciser functions.

CAgtPCIEExerciser This class contains the necessary exerciser API functions for requesting actions from the Serial I/O Modules.

Example of Using the In-System Port

The following example shows you how to look for an Exerciser from the PCI Express bus. It requires that the corresponding Serial I/O Module has an Exerciser personality and that the link is up.

```
// Source code from the intro sample
#include "AgtPCIEExerciser.h"
#include "AgtPortSelector.h"

#include <iostream>
using namespace std;

void main(int argc, char* argv[])
{
  CAgtPortSelector *myPortSelector=NULL; // Connection handling
  CAgtPCIEExerciser *myExerciser=NULL;   // Exerciser programming

  cout << "Starting main" << std::endl;

  try
   // All error handling is done via exceptions of CAgtException
  {
   // Create (the one and only) PortSelector instance.
   // This class provides functionality for
   // handling the connection to the PCIE driver.
   myPortSelector = new CAgtPortSelector;

   // Create (the one and only) Exerciser instance.
   // This class contains contains all
   // functions for programming the exerciser.
   myExerciser = new CAgtPCIEExerciser;

   UInt16 deviceId=AGT_INVALID_DEVID;
   // bus- (bits 15:8) slot- (7:4) and function (3:0) number

   // Get deviceId for first (parameter index=0) card found.
   // This looks for E2960 probeboards found on the PCIE bus.
   // To enumerate all cards in the system,
   you can call this function repeatedly with index=0,1,2,3,...
   // until the returned deviceId = AGT_INVALID_DEVID.
   myPortSelector->DeviceIdGet(0,&deviceId);
   if (deviceId==AGT_INVALID_DEVID)
   {
     cerr<<"No board found, exiting."<<endl;
     delete myExerciser;
     delete myPortSelector;
     return;
   }

   hex(cout);
   cout << "Found board at index 0, devid 0x"  << deviceId << endl;
```

```
    // Handle to device
    AgtPortHandleT myHandle=AGT_INVALID_PORTHANDLE;

    // Open connection to found board
    myPortSelector->ConnectPort(deviceId,&myHandle);
    cout << "Connection established." << endl;

    // Do something with myHandle here
    UInt32 val=0;
    myExerciser->ConfRegRead(myHandle,0,&val);
    cout << "Read " <<val<< " from configspace"<< endl;

    // Close connection to card
    myPortSelector->DisconnectPort(myHandle);
    cout << "Connection closed." << endl;

    delete myExerciser;
    delete myPortSelector;
   }
   catch (CAgtException &e)
   {
    // API error
    cerr << e << endl;
   }
   catch (...)
   {
    // catch all other errors here
   }
 }
```

Appendix

This chapter contains the tcl scripts that correspond with the actions carried out in the user interface. The following scripts are available:

- *“Starting an Exerciser Session” on page 77*
- *“Establishing a Link” on page 79*
- *“Sending a Single Packet” on page 80*
- *“Starting an Analyzer Session” on page 82*
- *“Downloading a Trigger Setup File to the Analyzer” on page 83*
- *“Writing Data to the Protocol Exerciser’s Data Memory” on page 84*
- *“Setting Up and Running Block Transfers” on page 85*
- *“Modifying Link Settings on the Exerciser” on page 89*

Starting an Exerciser Session

The following TCL script shows you how to start an Exerciser session for upstream testing (for example, to a PCI Express system).

It requires as input parameter the module and port number of the Serial I/O Module to be used for the test and returns the port handle of the Serial I/O Module (which can then be used by other scripts).

```
# The hostname is the IP address or network name
# of the controller PC
set ::hostName [info PCIX_Controller]

# Load the AgtClient library
# This makes all Agt??? procedures available to TCL
package require AgtClient

# Define the server hostname.
```

```
AgtSetServerHostname $::hostName

proc openExerciser { {port "101/1" } } {
# Establish a connection to an exerciser, acting as end node.
# The probe board will be set up so that the signals are
# routed from the exerciser connector
# to the SYS (bottom) connector.
# Parameter:
#    port: module/port number of the module you want to connect to
#
# Return value: the port handle
#
# Example: openExerciserToUpstream "103/1"
#

# Create an online exerciser to upstream session
# (exerciser is the upstream device)
  set session [AgtOpenSession PCIEExerciserToUpstream \
               AGT_SESSION_ONLINE]

# Or if a downstream session is required
#  set session [AgtOpenSession PCIEExerciserToDownstream \
#              AGT_SESSION_ONLINE]

# set the label of the test session to Analyzer
  AgtInvoke AgtSessionManager SetSessionLabel \
            $session "UpStream"

# For an downstream session
# AgtInvoke AgtSessionManager SetSessionLabel \
#            $session "DownStream"

# Attach the module/port to the active session
# (which is the one we created before)
  set portHandle [AgtInvoke AgtPortSelector AddPorts $port]

  return $portHandle
}

puts ""
puts "usage:"
puts "  openExerciserToUpstream \[<port>\]  "
puts "          - Exerciser upstream session on <port>."
puts "            (default: 101/1)"
puts ""
```

Establishing a Link

The following TCL script shows you how to use the Serial I/O Module to establish a PCI Express link to the system or add-in card under test.

It requires as input the Serial I/O Module’s port handle (which you can get in the example *“Starting an Exerciser Session” on page 77*) and the desired link widths as bit map. It returns the link state.

```
# Change the value from 0 to 1 if you want to use
# the exerciser in loopback mode.
set ::loopbackMode 0

# Load the AgtClient library.
# This makes all Agt??? procedures available to TCL.
package require AgtClient

proc linkUp { {portHandle 1} {linkWidth 0xf} } {
# Parameters:
#       portHandle:       The port handle you get when you opened
#                         the exerciser session.
#                         Typically, this is '1'
#       linkWidth:        The desired link width. This parameter
#                         is a bit map. A '1' per bit enables the
#                         appropriate width. Bit0: X1,
#                         Bit1: X2, Bit2: X4, Bit3: X8.
#
# Return value:           The actual link state
#
# Example: linkUp 1 0xf

# Program FC Update resend period. This determines the time
# between FC update packets.
# The unit is 16 symbol times, which results
# to (0x80 * 16 * 4 ns) 8.192 microseconds.
  AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
            PCIE_DLLPHY_FC_UPDATE_PERIOD 0x80

# Program the link width with the <linkWidth> parameter
  AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
            PCIE_DLLPHY_LINKMODE_CAPABLE $linkWidth

# If we are in loopback mode, we need to advertise
# link and lane number during link training.
# Which means, the exerciser needs to act as upstream device.
  if { $::loopbackMode == 1 } {
    AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
              PCIE_DLLPHY_UPSTREAM_PORT_ENABLE 0
  }
# Now, train the link
  AgtInvoke AgtPCIEExerciser LinkUp $portHandle

# wait 100 ms so that the link can establish
  after 100

# And read out the link state 0: inactive, 1: init, 2: active
  AgtInvoke AgtPCIEExerciser DataLinkStateRead $portHandle
}

puts ""
puts "usage:"
puts "  linkUp \[<portHandle>\] \[<linkWidth>\]
\[<disableScrambler>\]"
puts ""
puts "         - train the PCI Express link with the parameters: "
puts ""
puts "         <portHandle>            (default 1)"
puts "         <linkWidth>             (default 0xf)"
puts "         <disableScrambler>      (default 0)"
puts ""
```

Sending a Single Packet

The following TCL script shows how to send a single packet to the system or add-in card.

It requires as input the Serial I/O Module’s port handle and returns whether a completion was returned.

```
# Load the AgtClient library.
# This makes all Agt??? procedures available to TCL.
package require AgtClient

proc sendTlp { {portHandle 1} } {
# Parameters:
#       portHandle:       The port handle you get when
#                         you opened the exerciser session.
#                         Typically, this is '1'
#
# Return value:           The completion status.
#
# Example: sendTlp 1
#
# Set the immediate header defaults.
  AgtInvoke AgtPCIEExerciser SiDefaultSet $portHandle

# Program the immediate header fields that are
# different from the default.

# Memory read operation.
# The FMT and TYPE header fields define the operation.
# FMT and TYPE 0 -> memory read 32.
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_FMT 0
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_TYPE 0

# The packet should have 4 dwords payload.
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_LEN 4

# We want to use traffic class 0  .
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_TC 0x0

# And we want to use tag # 5.
# Need to set AUTOTAG to 0 in addition to enable programmable TAG.
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_TAG 5
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle PCIE_SI_AUTOTAG 0

# We want to do a memory read from physical address 0x200000.
  AgtInvoke AgtPCIEExerciser SiReqSet \
            $portHandle PCIE_SI_MEM32_ADDR 0x200000

# Now, send the immediate packet
  AgtInvoke AgtPCIEExerciser SiSend $portHandle

  set retval "No Completion"
# Check if we received a completion.
  if { [AgtInvoke AgtPCIEExerciser SiStatusGet $portHandle \
                PCIE_SISTATUS_COMP_AVAILABLE] == 1} {
    # Read out the completion status and print it.
    set cs [AgtInvoke AgtPCIEExerciser SiCompGet $portHandle \
                  PCIE_SI_COMP_STATUS]
    set retval "Completion Status: $cs"
  }

  return $retval
}

puts ""
puts "usage:"
puts "  sendTlp \[<portHandle>\]"
puts ""
puts "      - send a TLP with the exerciser at port " & \
            "<portHandle> (default 1) "
puts "        The TLP is a Memory Read32 of " & \
            "4 DWORDS from 0x200000"
puts ""
```

Starting an Analyzer Session

The following TCL script shows you how to start an Analyzer session.

It requires as input parameter the module and port number of the Serial I/O Module to be used for the test and returns the port handle of the Serial I/O Module (which can then be used by other scripts).

```
# The hostname is the IP address or network name
# of the controller PC
set ::hostName [info PCIX_Controller]

# Load the AgtClient library.
# This makes all Agt??? procedures available to TCL.
package require AgtClient

# Define the server hostname.
AgtSetServerHostname $::hostName

# Establish a connection to an protocol analyzer.
# Parameter:
#       port: module/port number of the module to be connected
#
# Return value: the port handle
#
# Example: openAnalyzer "103/1"
#
proc openAnalyzer { {port "101/1" } } {

# Create an online analyzer session
  set session [AgtOpenSession PCIEAnalyzer AGT_SESSION_ONLINE]

# set the label of the test session to Analyzer
  AgtInvoke AgtSessionManager SetSessionLabel $session "Analyzer"

# Attach the module/port to the active session
# (the one just created)
  set portHandle [AgtInvoke AgtPortSelector AddPorts $port]

  return $portHandle
}

puts ""
puts "usage:"
puts "  openAnalyzer \[<port>\]  "
puts "         - Analyzer session on <port>. (default: 101/1)"
puts ""
```

Downloading a Trigger Setup File to the Analyzer

The following TCL script shows how to download a trigger setup to the Analyzer.

It requires as input the Serial I/O Module's port handle (which you can get in the example *"Starting an Exerciser Session" on page 77*) and the name of the trigger setup file.

```
# The hostname is the IP address or network name
# of the controller PC
set ::hostName [info PCIX_Controller]

# Load the AgtClient library
package require AgtClient

# Define the server hostname.
AgtSetServerHostname $::hostName

# Load a previously saved trigger file
proc setupTrigger { {portHandle 1} \
                  {fileName "test_trigger_on_tlp.trg"} } {
# download the trigger condition defined in
# <fileName> to the protocol analyzer
# Parameters:
#      portHandle:      The port handle you get when
#                       you opened the analyzer session.
#      fileName:        The file name that contains
#                       the trigger condition.
#
# Return value: none
#
# Example: setupTrigger 1 default.trg
#

# open the trigger condition file <fileName>
  set fp [open $fileName]

# read the complete file int <triggerCond>
  set triggerCond [read $fp]

# check if the current session is an analyzer session
  set cid [AgtGetActiveConnection]
  set sessionType [AgtGetSessionType $cid]
  if {$sessionType == "PCIEAnalyzer"} {
      # yes, this is an analyzer session -- download the trigger
      AgtInvoke AgtPCIEAnalyzer TrigSeqSet $portHandle $triggerCond
  }

}

puts “”
puts “usage:”
puts “  setupTrigger \[portHandle\] \[fileName\]”
puts “         - download the trigger condition “ & \
                 “defined in <fileName> “
puts “           to the protocol analyzer”
puts “”
```

Writing Data to the Protocol Exerciser's Data Memory

You cannot write data to the Exerciser's data memory using TCL. You can use TCL to load files to the data memory.

The following TCL script shows how to load files to the Exerciser's data memory.

It requires as input the Serial I/O Module's port handle, the starting location in the Exerciser's data memory for the file, and the name of the file.

```
# Change the hostname here if you use a remote server (host)
set ::hostName [info PCIX_Controller]

# Load the AgtClient library
package require AgtClient

# Define the server hostname.
AgtSetServerHostname $::hostName

proc dataMemoryAccess { {portHandle 1} {offset 0} \
                        {fileName "default.txt"} } {
# download the data in <fileName> to the exerciser data memory
# Parameters:
#       portHandle:      The port handle you get when
#                        you opened the exerciser session.
#       offset:          The offset from 0 to place the data.
#       fileName:        The file name that contains the data.
#
# Return value: none
#
# Example: dataMemoryAccess 1 0 default.txt
#

# open the data file <fileName>
  set fp [open $fileName]

# read the complete file into <data>
  binary scan [read $fp] c* data

# check if the current session is an exerciser session
  set cid [AgtGetActiveConnection]
  set sessionType [AgtGetSessionType $cid]
  if {($sessionType == "PCIEExerciserToUpstream") || \
      ($sessionType == "PCIEExerciserToDownstream")} {
      # yes, this is an exerciser session -- download the data
      AgtInvoke AgtPCIEExerciser DataMemWrite $portHandle \
                                 $offset $data
  }

  close $fp
}

puts ""
puts "usage:"
puts "  dataMemoryAccess \[portHandle\] \[offset\] \[fileName\]"
puts "          - download the data defined in <fileName> " & \
                "to the exerciser"
puts "            at offset <offset>"
puts ""
```

Setting Up and Running Block Transfers

The following TCL script provides some examples for running block transfers.

It requires as input the command (configRead0, configRead1, configWrite0, configWrite1), the config bus and device number, function number and register number to access, and the Serial I/O Module’s port handle. The read functions return the data read.

```
# Load the AgtClient library.
# This makes all Agt??? procedures available to TCL.
package require AgtClient
```

```
proc waitForCompletion { {portHandle 1} {timeout 500} } {
# wait until the exerciser reports that there's a completion
available
#   Parameters:        portHandle:     the port handle to use
#                      timeout:        wait <timeout> ms maximum.
#                                      Throw an error if
#                                      this time is exceeded
#   Return value:      none

for {set i 0 } {$i < $timeout} {incr i} {
    # wait one ms
    after 1
    if {[AgtInvoke AgtPCIEExerciser SiStatusGet \
                $portHandle PCIE_SISTATUS_COMP_AVAILABLE] == 1 } {
      break
    }
  }

  if {$i >= $timeout} {error "no completion"}
}

  puts ""
  puts "usage:"
  puts "        configRead0 | configRead1 | configWrite0 | " & \
       "configWrite1 \[<bus>\] \[<dev>\] \[<func>\] \[<reg>\] " & \
       "\[<portHandle>\]"
  puts ""
  puts "     - create a configuration access read or write, " & \
       "type 0 or 1, with the following parameters:"
  puts ""
  puts "      <bus> config bus number (default 0)"
  puts "      <dev> config device number (default 0)"
  puts "      <func> function number to access (default 0)"
  puts "      <reg> register number to access (default 0)"
  puts "      <portHandle> port handle to use (default 1)"

proc configAccess { {rw 0} {type 0} {bus 0} {dev 0} {func 0}  \
                    {reg 0} {dat 0} {retry 3} {portHandle 1} } {
# This is the main procedure. It will create a configuration
# access and send it. After that it will wait for the completion
# and retry the config access if the completion status was 'CRS'.
# If the config access was a read,
# this procedure will return the data.
#   Parameters:
#       rw:     0 creates a read, 1 creates a write
#       type:   0 creates a type0, 1 creates a type1 config access
#       bus:    the config bus number
#       dev:    the config device number
#       func:   the function number to access
#       reg:    the register number to access
#       dat:    this contains the data to be written
#       retry:  number of times this access will be repeated
#               if 'CRS' completion is signaled
#       portHandle:     the port handle to use
#   Return value:       none if it was a write,
#                       data if it was a read
#   Errors:             This will throw a "no completion" error if
#                       no completion was received within 500 ms
#

# set up the Send Immediate TLP
  AgtInvoke AgtPCIEExerciser SiDefaultSet $portHandle
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_TYPE [expr 4 + $type]
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_FMT [expr $rw * 2]
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_AUTOTAG 1
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_CFG_FUNCNUM $func
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_CFG_DEVNUM $dev
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_CFG_BUSNUM $bus
  AgtInvoke AgtPCIEExerciser SiReqSet $portHandle \
            PCIE_SI_CFG_REGNUM $reg

  if {$rw == 1} {
    # this is a write - set si req memory
    AgtInvoke AgtPCIEExerciser SiReqMemDWSet $portHandle 0 $dat
  }

  # now, send the config request
  AgtInvoke AgtPCIEExerciser SiSend $portHandle

  # wait max. 500 ms for a completion
  waitForCompletion $portHandle 500

  # get the completion status
  set cs [AgtInvoke AgtPCIEExerciser SiCompGet 1 \
                    PCIE_SI_COMP_STATUS]
  # successful completion
  if { $cs == 0 } {
    # successful completion -- get the data if it was a read
    if {$rw == 0} {
      set dat [AgtInvoke AgtPCIEExerciser SiCompMemDWGet \
                         $portHandle 0]
    }
  } elseif {$cs == 2} {
    # config retry status - repeat the request if retry is < 1
    if {$retry > 0} {
      set retry [expr $retry - 1]
      configAccess $rw $type $bus $dev $func $reg \
                   $dat $retry $portHandle
    }
  }
}

proc configRead0 { {bus 0} {dev 0} {func 0} {reg 0} \
                   {portHandle 1} } {
# Create and send a config type0 read access.
# Repeat the request up to three times if
# a 'CRS' response is received.
#   Parameters:
#       bus:    the config bus number
#       dev:    the config device number
#       func:   the function number to access
#       reg:    the register number to access
#   Return value:       the read data
#   Errors:             This will throw a "no completion" error
#                       if no completion was received within 500 ms
#
  configAccess 0 0 $bus $dev $func $reg 0 3 $portHandle
}

proc configRead1 { {bus 0} {dev 0} {func 0} {reg 0} \
                   {portHandle 1} } {
# Create and send a config type1 read access. Repeat the
# request up to three times if a 'CRS' response is received.
#   Parameters:
#       bus:    the config bus number
#       dev:    the config device number
#       func:   the function number to access
#       reg:    the register number to access
#   Return value:       the read data
#   Errors:             This will throw a "no completion" error
#                       if no completion was received within 500
ms.
#
  configAccess 0 1 $bus $dev $func $reg 0 3 $portHandle
}

proc configWrite0 { {bus 0} {dev 0} {func 0} {reg 0} \
                    {dat 0} {portHandle 1} } {
# Create and send a config type0 write access.
# Repeat the request up to three times
# if a 'CRS' response is received.
#   Parameters:
#       bus:    the config bus number
#       dev:    the config device number
#       func:   the function number to access
#       reg:    the register number to access
#       dat:    this contains the data to be written
#   Return value:       none
#   Errors:             This will throw a "no completion" error if
#                       no completion was received within 500 ms.
#
  configAccess 1 0 $bus $dev $func $reg $dat 3 $portHandle
}

proc configWrite1 { {bus 0} {dev 0} {func 0} {reg 0} \
                    {dat 0} {portHandle 1} } {
# Create and send a config type1 write access.
# Repeat the request up to three times
# if a 'CRS' response is received.
#   Parameters:
#       bus:    the config bus number
#       dev:    the config device number
#       func:   the function number to access
#       reg:    the register number to access
#       dat:    this contains the data to be written
#   Return value:       none
#   Errors:             This will throw a "no completion" error if
#                       no completion was received within 500 ms.
#
  configAccess 1 1 $bus $dev $func $reg $dat 3 $portHandle
}
```

Modifying Link Settings on the Exerciser

The following TCL script shows you how to configure an Exerciser's link parameters.

It requires as input the Serial I/O Module's port handle, the desired link widths as bit map, and whether or not scrambling should be disabled. It returns the link state.

```
# Change the value from 0 to 1 if you want to use
# the exerciser in loopback mode.
set ::loopbackMode 0

# Load the AgtClient library.
# This makes all Agt??? procedures available to TCL.
package require AgtClient

proc linkUp { {portHandle 1} {linkWidth 0xf} {disableScrambler 0} }
{
# Parameters:
```

```
#       portHandle:       The port handle you get when you opened
#                         the exerciser session.
#                         Typically, this is '1'
#       linkWidth:        The desired link width. This parameter
#                         is a bit map. A '1' per bit enables the
#                         appropriate width. Bit0: X1,
#                         Bit1: X2, Bit2: X4, Bit3: X8.
#       disableScrambler: 0: enable scrambling,
#                         1: disable scrambling
#
# Return value:           The actual link state
#
# Example: linkUp 1 0xf 0

# Program the scrambler with the <disableScrambler> parameter
  AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
            PCIE_DLLPHY_DISABLE_SCRAMBLE $disableScrambler

# Program FC Update resend period. This determines the time
# between FC update packets.
# The unit is 16 symbol times, which results
# to (0x80 * 16 * 4 ns) 8.192 microseconds.
  AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
            PCIE_DLLPHY_FC_UPDATE_PERIOD 0x80

# Program the link width with the <linkWidth> parameter
  AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
            PCIE_DLLPHY_LINKMODE_CAPABLE $linkWidth

# If we are in loopback mode, we need to advertise
# link and lane number during link training.
# Which means, the exerciser needs to act as upstream device.
  if { $::loopbackMode == 1 } {
    AgtInvoke AgtPCIEExerciser DllPhySet $portHandle \
              PCIE_DLLPHY_UPSTREAM_PORT_ENABLE 0
  }

# Now, train the link
  AgtInvoke AgtPCIEExerciser LinkUp $portHandle

# wait 100 ms so that the link can establish
  after 100

# And read out the link state 0: inactive, 1: init, 2: active
  AgtInvoke AgtPCIEExerciser DataLinkStateRead $portHandle
}

puts ""
puts "usage:"
puts "  linkUp \[<portHandle>\] \[<linkWidth>\]
\[<disableScrambler>\]"
puts ""
puts "          - train the PCI Express link with the parameters: "
puts ""
puts "          <portHandle>            (default 1)"
puts "          <linkWidth>             (default 0xf)"
puts "          <disableScrambler>      (default 0)"
puts ""
```

Index

E2960-91020

Agilent Technologies