



PCI/PCI-X FlexModel User's Manual

**pcimaster_fx
pcislave_fx
pcimonitor_fx**

July 2002

Copyright © 2002 Synopsys, Inc.
All rights reserved.
Printed in USA.

Information in this document is subject to change without notice.

Synopsys and the Synopsys logo are registered trademarks of Synopsys, Inc. For a list of Synopsys trademarks, refer to this web page:

<http://www.synopsys.com/copyright.html>

All company and product names are trademarks or registered trademarks of their respective owners.

Contents

Preface	13
About This Manual	13
Related Documents	14
Manual Overview	14
Typographical and Symbol Conventions	16
Terminology	17
Getting Help	17
The Synopsys Website	19
Comments?	19
Chapter 1	
Overview of the	
PCI/PCI-X FlexModels	21
Introduction	21
The PCI/PCI-X FlexModel Set	22
Chapter 2	
Programming PCI/PCI-X FlexModels	31
Introduction	31
Setting up Your Simulator	32
Setting the TimingVersion Generic/Defparam	32
Migrating and Updating from Earlier Versions of the Models	33
Using PCI FlexModels in PCI-X Mode	33
Creating Your Design From Examples	34
The PCI and PCI-X System Testbenches	35
Model-specific Testbenches	35
Instantiating a FlexModel into Your Design	36
Working with Timing	36
Troubleshooting Your Design	37
PCI-X Transactions	37
DWORD Transactions	37
Burst Transactions	39
Master Terminations	43
Target Terminations	45
Split Transactions	48
Split Completion Messages (SCM)	50

Wait States and Delays	51
Parity Error Generation for the Master and Slave	52
64-bit Data Transfers	57
Dual Address Cycles	59
Decode Speeds	60
Using the pcimonitor_fx to Track Bus PCI/X Transactions	60
pcimonitor_fx Control Pin	60
Generating the pcimonitor Trace File	61
Using the Trace File with the PCI Test Suite	61
Sample of a pcimonitor Trace File	62
Trace File Report Clarifications	63
Fast Turn Around in PCIX Mode	63
Using Custom Arbiters with the pcimonitor_fx	65
Using Error Check Registers	65
Changing the pcimonitor GNT_ASSERTED_CLKS	68
Proper Usage of the pcislave_request Command	68
Proper Use of the pcislave_read_rslt Command	71

Chapter 3

Using the Conventional Mode

PCI System Testbench	73
Introduction	73
Model Behaviors Demonstrated in the Testbench	74
Verilog, C, and VHDL Testbench Examples	74
VERA Testbench Examples	76
Setting up the Verilog, C, and VHDL Conventional Mode System Testbenches ..	77
Setting Up the VERA Conventional Testbench\	82
VERA Testbench/Script Setup Tasks	82
Command Synchronization in the Verilog, C, and VHDL Testbenches	87
PCI System Testbench Architecture, Files, and Setup Values	88
PCI System Testbench Files	91
VERA PCI System Testbench Files	92
Configuration Details for the PCI System Testbench	93
idsel_maker	94
Primary PCI Master (U1) Characteristics	94
Secondary PCI Master (U2) Characteristics	95
Primary PCI Slave (U3) Characteristics	95
Secondary PCI Slave (U4) Characteristics	97
PCI Monitor (U5) Characteristics	98
PCI System Testbench Operation Sequences	99

Main Test Sequence	99
Primary PCI Master (U1) Sequence	100
Secondary PCI Master (U2) Sequence	101
Primary PCI Slave (U3) Sequence	102
Secondary PCI Slave (U4) Sequence	103
Chapter 4	
Using the PCI-X System Testbench	105
Introduction	105
Using Testbench Command Examples	106
Setting up the Verilog, C, and VHDL Language PCI-X System Testbenches ...	106
Multi-Instance Verilog Testbenches	111
Running the VERA PCI-X System Testbenches	112
VERA Testbench/Script Setup Tasks	112
PCI-X System Testbench Architecture and Setup Values	118
Main Test Sequence Description	121
Basic Configuration Details for the PCI-X System Testbench	121
Verilog, C, and VHDL PCI-X System Testbench Files	123
VERA PCI-X System Testbench Files	124
Primary PCI Master (U1) Configuration	125
Secondary PCI Master (U2) Configuration	125
Primary PCI Slave (U3) Configuration	126
Secondary PCI Slave (U4) Configuration	127
PCI Monitor (U5) Configuration	129
Chapter 5	
Using the pcimaster_fx	131
Introduction	131
pcimaster_fx Supported Functions	131
pcimaster_fx Modeling Exceptions and Unsupported Features	132
Using the pcimaster_fx in PCI-X Mode	133
100 Mhz Support	133
pcimaster_fx Command Summary	134
pcimaster_fx Command Reference	135
pcimaster_configure	136
pcimaster_idle	150
pcimaster_output_enable	153
pcimaster_pin_req	156
pcimaster_pin_rslt	160
pcimaster_read_continue	164
pcimaster_read_cycle	168

pcimaster_read_intr_ack	176
pcimaster_read_rslt	179
pcimaster_reg_req	183
pcimaster_reg_rslt	186
pcimaster_set_msg_level	189
pcimaster_set_pin	193
pcimaster_set_reg	197
pcimaster_set_timing_control	200
pcimaster_write_continue	203
pcimaster_write_cycle	207
pcimaster_write_special_cyc	214
Chapter 6	
Using the pcislave_fx	217
Introduction	217
pcislave_fx Supported Functions	218
pcislave_fx Modeling Exceptions and Unsupported Features	219
Using the pcislave_fx in PCI-X Mode	220
pcislave_fx Command Summary	220
pcislave_fx Command Reference	222
pcislave_addr_req	223
pcislave_configure	226
pcislave_configure_delay	241
pcislave_dump_file	244
pcislave_load_file	248
pcislave_output_enable	253
pcislave_pin_req	256
pcislave_pin_rslt	260
pcislave_read_rslt	264
pcislave_request	268
pcislave_set_addr	275
pcislave_set_msg_level	278
pcislave_set_pin	281
pcislave_set_timing_control	284
Chapter 7	
Using the pcimonitor_fx	287
Introduction	287
pcimonitor_fx Supported Functions	287
pcimonitor_fx Modeling Exceptions and Unsupported Features	288
Using the pcimonitor_fx in PCI-X Mode	288

PCI and PCI-X Error Checks	289
PCI Error Checks	289
PCI-X Error Checks	293
PCI Warning Messages	298
Common PCI and PCIX Error Messages	298
pcimonitor_fx Command Summary	299
pcislave_fx Command Reference	300
pcimonitor_configure	301
pcimonitor_output_enable	307
pcimonitor_pin_req	310
pcimonitor_pin_rslt	314
pcimonitor_reg_req	318
pcimonitor_reg_rslt	321
pcimonitor_set_msg_level	324
pcimonitor_set_pin	328
pcimonitor_set_reg	331
pcimonitor_set_timing_control	334
Appendix A	
Values for timing_param_index Parameter	337
Introduction	337
pcimaster_fx Parameter Values	337
pcislave_fx Parameter Values	348
pcimonitor_fx Parameter Values	357
Appendix B	
Ensuring Compatibility with PCI/PCI-X FlexModels	367
Introduction	367
Compatibility Within the PCI/PCI-X FlexModel Set	368
Compatibility With Existing Testbenches	368
Method 1: Modifying Your Existing Testbench	369
Method 2: Installing the Compatibility Files	370
Compatibility With the PCI FlexModel Test Suite	371
Appendix C	
VERA Testbench Scripts for Verilog-XL, NC-Verilog, and MTI	373
Example Script to Run the PCI VERA Testbench with the NC-Verilog Simulator ...	376
Example Script to Run the PCIX VERA Testbench with the NC-Verilog Simulator ..	379
Example Script to Run the PCI VERA Testbench with Verilog-XL	382

Example Script to Run the PCIX VERA Testbench with Verilog-XL	385
Example Script to Run the PCI VERA Testbench with MTI	388
Example Script to Run the PCIX VERA Testbench with MTI	391
Appendix D	
Frequently Asked Questions (FAQ)	395
All PCI and PCIX models	395
pcislave_fx	397
Index	399
Addendum:	
History and Version Information	1

Figures

Figure 1:	pcimaster_fx and pcislave_fx Pins	24
Figure 2:	pcimonitor_fx Pins	27
Figure 3:	PCI/PCI-X FlexModels in a System Testbench (high-level view)	30
Figure 4:	Conventional Mode PCI System Testbench—HDL/VERA Version	89
Figure 5:	Conventional Mode PCI System Testbench—C Version	90
Figure 6:	PCI-X System Testbench—HDL/VERA Version	119
Figure 7:	PCI-X System Testbench—C Version	120
Figure 8:	Verilog Memory Format	249
Figure 9:	Intel Hexadecimal Object File Format	251

Tables

Table 1:	pcimaster_fx and pcislave_fx PCI(X) Pins	25
Table 2:	pcimonitor_fx PCI(X) Pins	28
Table 3:	Information Provided in Model-specific Testbenches	35
Table 4:	Parity Error Generation with Master and Slave	52
Table 5:	Selecting Bus Halves for Parity (Used in Conjunction with PCIMASTER_PCI_ERR)	54
Table 6:	Selecting Bus Halves for Parity (Used in Conjunction with PCISLAVE_PCI_ERROR or PCISLAVE_SPLIT_PCI_ERROR)	54
Table 7:	pcimonitor_fx PCI and PCIX Error Registers	65
Table 8:	Behaviors Demonstrated in C, Verilog, and VHDL Conventional Mode Testbenches	75
Table 9:	Behaviors Demonstrated in the VERA Testbench By Class File	76
Table 10:	IDSEL Assertion in the Conventional Mode Testbench	94
Table 11:	Primary Master pcimaster_configure Command Settings	95
Table 12:	Secondary Master pcimaster_configure Command Settings	95
Table 13:	Primary Slave pcislave_configure Command Settings	96
Table 14:	Primary Slave pcislave_load_file Command Settings	97
Table 15:	Secondary Slave pcislave_configure Command Settings	98
Table 16:	Secondary Slave pcislave_load_file Command Settings	98
Table 17:	IDSEL Assertion in the PCI-X System Testbench	122
Table 18:	Primary Master pcimaster_configure Command Settings	125
Table 19:	Secondary Master pcimaster_configure Command Settings	126
Table 20:	Primary Slave pcislave_configure Command Settings	127
Table 21:	Primary Slave pcislave_load_file Command Settings	127
Table 22:	Secondary Slave pcislave_configure Command Settings	128
Table 23:	Secondary Slave pcislave_load_file Command Settings	129
Table 24:	pcimaster_fx Command Summary	134
Table 25:	ctype and cvalue Values in pcimaster_configure Command	137
Table 26:	pin_name Values in pcimaster_output_enable Command	153
Table 27:	pin_name Values in pcimaster_pin_req Command	156
Table 28:	pin_name Values in pcimaster_pin_rslt Command	160
Table 29:	rtype Values in pcimaster_read_cycle Command	168
Table 30:	Register Value Definitions for PCIMASTER_STATUS_REG	184
Table 31:	Register Value Definitions for PCIMASTER_STATUS_REG	186
Table 32:	mode Values in pcimaster_set_msg_level Command	189
Table 33:	pin_name Values in pcimaster_set_pin Command	193

Table 34: Register Value Definitions for PCIMASTER_STATUS_REG	197
Table 35: wtype Values in pcimaster_write_cycle Command	207
Table 36: pcislave_fx Command Summary	220
Table 37: ctype and cvalue Values in pcislave_configure Command	227
Table 38: Values for cvalue when ctype is PCISLAVE_PCI_ERROR	235
Table 39: Simulated Split Completion Error Types	235
Table 40: mem_space Values in pcislave_dump_file Command	244
Table 41: mem_space Values in pcislave_load_file Command	248
Table 42: pin_name Values for pcislave_output_enable Command	253
Table 43: pin_name Values in pcislave_pin_req Command	256
Table 44: pin_name Values in pcislave_pin_rslt Command	260
Table 45: decode Index Values in pcislave_request Command	268
Table 46: mode Values in pcislave_set_msg_level Command	278
Table 47: pin_name Values in pcislave_set_pin Command	281
Table 48: pcimonitor_fx Command Summary	299
Table 49: ctype and cvalue Values Used in the pcimonitor_configure Command ..	302
Table 50: Arbitration Schemes for PCIMONITOR_PRIORITY_N Setting	303
Table 51: pin_name Values in pcimonitor_output_enable Command	307
Table 52: pin_name Values in pcimonitor_pin_req Command	310
Table 53: pin_name Values in pcimonitor_pin_rslt Command	314
Table 54: reg_name Values in pcimonitor_reg_req Command	318
Table 55: reg_name Values in pcimonitor_reg_rslt Command	321
Table 56: mode Values in pcimonitor_set_msg_level Command	324
Table 57: pin_name Values in pcimonitor_set_pin Command	328
Table 58: reg_name Values in pcimonitor_set_reg Command	331
Table 59: Group Constant Values for timing_param_index Parameter in pcimaster_set_timing_control Command	337
Table 60: Individual Constant Values for timing_param_index Parameter in pcimaster_set_timing_control Command	338
Table 61: Group Constant Values for timing_param_index Parameter in pcislave_set_timing_control Command	348
Table 62: Individual Constant Values for timing_param_index Parameter in pcislave_set_timing_control Command	349
Table 63: Group Constant Values for timing_param_index Parameter in pcimonitor_set_timing_control Command	357
Table 64: Individual Constant Values for timing_param_index Parameter in pcimonitor_set_timing_control Command	358
Table 65: New and Changed Pins for the pcislave_fx	369

Preface

About This Manual

This manual describes the PCI/PCI-X FlexModel family of bus interface models. The PCI/PCI-X FlexModels (called PCI FlexModels or PCI models throughout this manual) can run in either PCI-X mode or conventional PCI mode. In conventional PCI mode, the PCI FlexModels support the 33MHz and 66MHz timing versions as described in the PCI Local Bus Specification, Revision 2.2. In PCI-X mode, the PCI FlexModels also support the 66MHz and 133MHz timing versions as described in the PCI-X Addendum to the PCI Specification.

To simulate the full range of PCI/PCI-X local bus behavior, you use the individual `pcimaster_fx`, `pcitarget_fx`, and `pcimonitor_fx` FlexModels. The bulk of this manual describes functionality and usage for these models and for the PCI and PCI-X system-level testbenches.

If you are migrating from an earlier version of the PCI FlexModels and want to use PCI-X mode, see [“Using PCI FlexModels in PCI-X Mode” on page 33](#). If you are migrating from an earlier version of the PCI FlexModels and want to continue using conventional PCI mode, see [Appendix B](#) for important information about backward-compatibility.

Information about FlexModels is provided in the following publications:

- Model-specific behavior is documented in this datasheet. The bulk of the datasheet describes model functionality and usage. Model version and history information is provided in the [History and Version Information](#) addendum at the end of the datasheet. Because the individual models are delivered and versioned separately, this model version and history information is specific to the PCI FlexModel with which this manual is delivered. This means that you have one *PCI/PCI-X FlexModel User's Manual* for each individual PCI FlexModel that you install. If you install all three PCI FlexModels (`pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`), you get three manuals that are identical except for the model version and history information in the Addendum.

- General FlexModel behavior and control is documented in the *FlexModel User's Manual*.
- Information about fixed and outstanding FlexModel and SmartModel problems is provided in the *SmartModel Library Release Notes*.

Related Documents

To see a complete list of SmartModel Library documents prior to installing the model, refer to the *Guide to SmartModel Documentation*, available in the “DesignWare Verification IP” section on the Synopsys website at this URL:

<http://www.synopsys.com/products/designware/docs>

or after installation at:

`$LMC_HOME/doc/smartmodel/manuals/intro.pdf`

The following diagram shows where FlexModel documents are located in these environments:



Manual Overview

This manual contains the following chapters:

Preface

Tells you where to find more information about PCI/PCI-X FlexModels. Also describes conventions and terminology used in this manual.

Chapter 1: Overview of the PCI/PCI-X FlexModels

Contains general information about the PCI/PCI-X FlexModels.

Chapter 2 Programming PCI/PCI-X FlexModels

Describes Synopsys resources you can use to test your PCI/PCI-X design.

Chapter 3 Using the Conventional Mode PCI System Testbench	Describes the system-level testbenches designed for use in conventional PCI mode. Contains detailed information on testbench setup and configuration, operation sequences, model synchronization, and model behavior.
Chapter 4 Using the PCI-X System Testbench	Describes the PCI-X system-level testbenches. Contains detailed information on testbench setup and configuration, operation sequences, model synchronization, and model behavior.
Chapter 5: Using the pcimaster_fx	Describes supported functions, modeling exceptions, usage, and commands for the pcimaster_fx model. Contains a command summary and detailed reference pages for all pcimaster_fx commands.
Chapter 6: Using the pcislave_fx	Describes supported functions, modeling exceptions, usage, and commands for the pcislave_fx model. Contains a command summary and detailed reference pages for all pcislave_fx commands.
Chapter 7: Using the pcimonitor_fx	Describes supported functions, modeling exceptions, usage, and commands for the pcimonitor_fx model. Contains a command summary and detailed reference pages for all pcimonitor_fx commands.
Appendix A: Values for timing_param_index Parameter	Lists values for the <i>timing_param_index</i> parameter in the pcimaster_set_timing_control, pcislave_set_timing_control, and pcimonitor_set_timing_control commands.
Appendix B: Ensuring Compatibility with PCI/PCI-X FlexModels	Describes ways to keep your PCI/PCI-X FlexModels both backward- and forward-compatible.
Appendix C: VERA Testbench Scripts for Verilog-XL, NC- Verilog, and MTI	Contains example scripts on running the VERA testbenches with NC-Verilog, Verilog-XL, and ModelSim.

**Appendix D:
Frequently Asked
Questions (FAQ)**

Frequently Asked Questions.

**Addendum:
History and Version
Information**

Provides the current model version number, bibliographic information about the sources used as references during model development, and relevant model history for the last year.

Typographical and Symbol Conventions

- **Default UNIX prompt**

Represented by a percent sign (%).

- **User input** (text entered by the user)

Shown in **bold** monospaced type, as in the following command line example:

```
% cd $LMC_HOME/bin
```

- **System-generated text** (prompts, messages, files, reports)

Shown in monospaced type, as in the following system message:

```
VALIDATION PASSED: No Mismatches during simulation
```

- **Variables** for which you supply a specific value

Shown in italic type, as in the following command line example:

```
% setenv LMC_HOME prod_dir
```

In this example, you substitute a specific name for *prod_dir* when you enter the command.

- **Command syntax**

Choice among alternatives is shown with a vertical bar (|) as in the following:

```
termination_style, 0 | 1
```

In this example, you must choose one of the two possibilities: 0 or 1.

Optional parameters are enclosed in square brackets ([]) as in the following:

```
pin1 [pin2 ... pinN]
```

In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* ... *pinN*]).

Terminology

PCI/PCI-X FlexModels	PCI bus models that can run in either PCI-X mode or conventional PCI mode. Also referred to as <i>PCI FlexModels</i> or <i>PCI models</i> .
bus-functional model	Also BFM. Refers to a model (FlexModel) that emulates device protocol at the pin and bus-cycle level.
filename extension <i>.ext</i>	<i>.ext</i> is either <i>.vhd</i> for VHDL, or <i>.v</i> and <i>.inc</i> for Verilog.
result command	Refers to a command that retrieves results from the model.
request command	Refers to a command that asks the model to perform an operation that will produce results. Must precede a result command in a test sequence.

Getting Help

If you have a question while using Synopsys products, use the following resources:

- Product documentation installed on your network or located at the root level of your Synopsys CD-ROM.
- Product documentation for the latest version of all products on the Web:
<http://www.synopsys.com/products/designware/docs>
- Datasheets for all verification models and Design Ware Implementation IP available using the IP Directory:
<http://www.synopsys.com/products/designware/ipdir.html>
- The online Support Center available at one of the following URLs:
 - DesignWare Macrocells, DesignWare Foundation Library, or coreBuilder Tools customers:
<http://solvnet.synopsys.com/>
 - SmartModel, FlexModel, MemPro, VMC, VhMC, and CMC customers:
<http://www.synopsys.com/support/lm/support.html>

If you still have questions about the following products, you can call a Synopsys support center:

- DesignWare Macrocells, DesignWare Foundation Library, and coreBuilder Tools
 - United States:
Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.

- Canada:
Call 1-650-584-4200 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.
- All other countries:
Find other local support center telephone numbers at the following URL:
http://www.synopsys.com/support/support_ctr/

- SmartModels, FlexModels, MemPro, VMC, and VhMC
 - North America:
Call 1-800-445-1888 from 7:00 AM to 5 PM Pacific Time, Mon—Fri.
 - All other countries:
Call your local sales office.

The Synopsys Website

General information about Synopsys and its products is available at this URL:

<http://www.synopsys.com>

Comments?

To report errors or make suggestions, please send email to:

doc@synopsys.com

To report an error that occurs on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your email message. This will provide us with information to identify the source of the problem.

1

Overview of the PCI/PCI-X FlexModels

Introduction

The Synopsys PCI/PCI-X FlexModels help you test your designs for proper operation against the PCI and PCI-X specifications. By using these bus interface models to create a virtual PCI or PCI-X system around your design, you can generate tests quickly and efficiently. Extensive design testing ensures that your design is compatible with other PCI components you might use in the end system. Built-in timing and usage checks help you verify your design by generating warning messages when protocol violations occur.



Attention

The `pcimonitor_fx` does not verify all scenarios and conditions listed in the PCI or PCIX Compliance Checklists. Refer to the section [“Using the `pcimonitor_fx`” on page 287](#) for a list of what the `pcimonitor_fx` checks.

The Synopsys PCI/PCI-X FlexModel set consists of three separate PCI/PCI-X FlexModels and a set of system-level testbenches. The three FlexModels—the `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`—are behavioral, bus-functional models. These models define in software the interface and function of the electronic hardware and simulate only the PCI or PCI-X inputs and outputs. The system-level testbenches are ready-to-use testbenches that you can use to verify your installation and familiarize yourself with the models in either conventional PCI mode or PCI-X mode. You can also use a system testbench as a template for building your own testbench.

You can use PCI/PCI-X FlexModels in HDL, C, and Vera command modes. For more information on using these command modes, see [“Command Modes,”](#) in the *FlexModel User's Manual*. In conventional PCI mode, PCI/PCI-X FlexModels support the 33MHz and 66MHz timing versions as described in the PCI Local Bus Specification, Revision 2.2. In PCI-X mode, PCI/PCI-X FlexModels also support the 66MHz and 133MHz timing versions as described in the PCI-X Addendum to the PCI Specification.

**Note**

The PCI/PCI-X FlexModels do not support interrupts as do processor device FlexModels. However, the PCI/PCI-X FlexModels do support interrupt transactions on the bus.

The PCI/PCI-X FlexModel Set

The models that make up the PCI/PCI-X FlexModel set are:

- **pcimaster_fx.** Performs timing violation checks and emulates the protocol of PCI/PCI-X initiators at the pin and bus-cycle levels. Initiates read and write cycles. In PCI-X mode, pcimaster_fx can function as a target for split transactions. For more details, see [“Using the pcimaster_fx” on page 131.](#)
- **pcislave_fx.** Responds to cycles initiated by the pcimaster_fx model or by the user's PCI master device. In PCI-X mode, the pcislave_fx also functions as an initiator for split transactions. For more details, see [“Using the pcislave_fx” on page 217.](#)
- **pcimonitor_fx.** Monitors, logs, and arbitrates activity on the PCI or PCI-X bus. For more details, see [“Using the pcimonitor_fx” on page 287.](#)
- **PCI and PCI-X system testbenches.** Provides ready-to-use example testbenches for both conventional PCI mode and PCI-X mode. Each system testbench uses two pcimaster_fx models, two pcislave_fx models, and a pcimonitor_fx model. For more details, see [“Using the Conventional Mode PCI System Testbench” on page 73](#) and [“Using the PCI-X System Testbench” on page 105.](#)

Together, the pcimaster_fx and pcislave_fx models emulate the functions of the PCI or PCI-X local bus and enable you to specify simple and complex data transfer cycles such as reads, writes, and idles. The pcimonitor_fx arbitrates the bus and helps you verify your PCI or PCI-X design. And the system testbenches let you get a quick start on using the models together to verify your design.

The [Figure 1](#) and [Figure 2](#) shows the pin definitions for the pcislave_fx, pcimaster_fx, and pcimonitor_fx.

Note the following about the `pcimaster_fx` and the `pcislave_fx`:

- The PCI AD address/data bus has been divided into two separate busses on the model representing the upper and lower 32 bits of the bus. Thus AD[63:32] maps to the PD bus and AD[31:0] maps to PADATA.
- The PCI C/BE# bus has been mapped to two separate busses: C/BE[7::4]# maps to PBENN, and C/BE[0::3]# maps to PCXBENN.
- The `pcimaster_fx` and `pcislave_fx` have three unique pins: P66HZ, PSBONN, AND PSDONE.
- Note, the models ignore the PSBOON AND PSDONE signals. Snooping was made obsolete in the version 2.2 of the PCI Specification.
- PCI Optional JTAG and Interrupt pins are not supported.

Note the following about the `pcimonitor_fx`:

- The `pcimonitor_fx` has four unique pins: P66HZ, PSBONN, DISABLEMST, and PSDONE.
- Note, the model ignores the PSBOON AND PSDONE signals. Snooping was made obsolete in the version 2.2 of the PCI Specification.
- The `pcimonitor_fx` supports PCI interrupt pins.

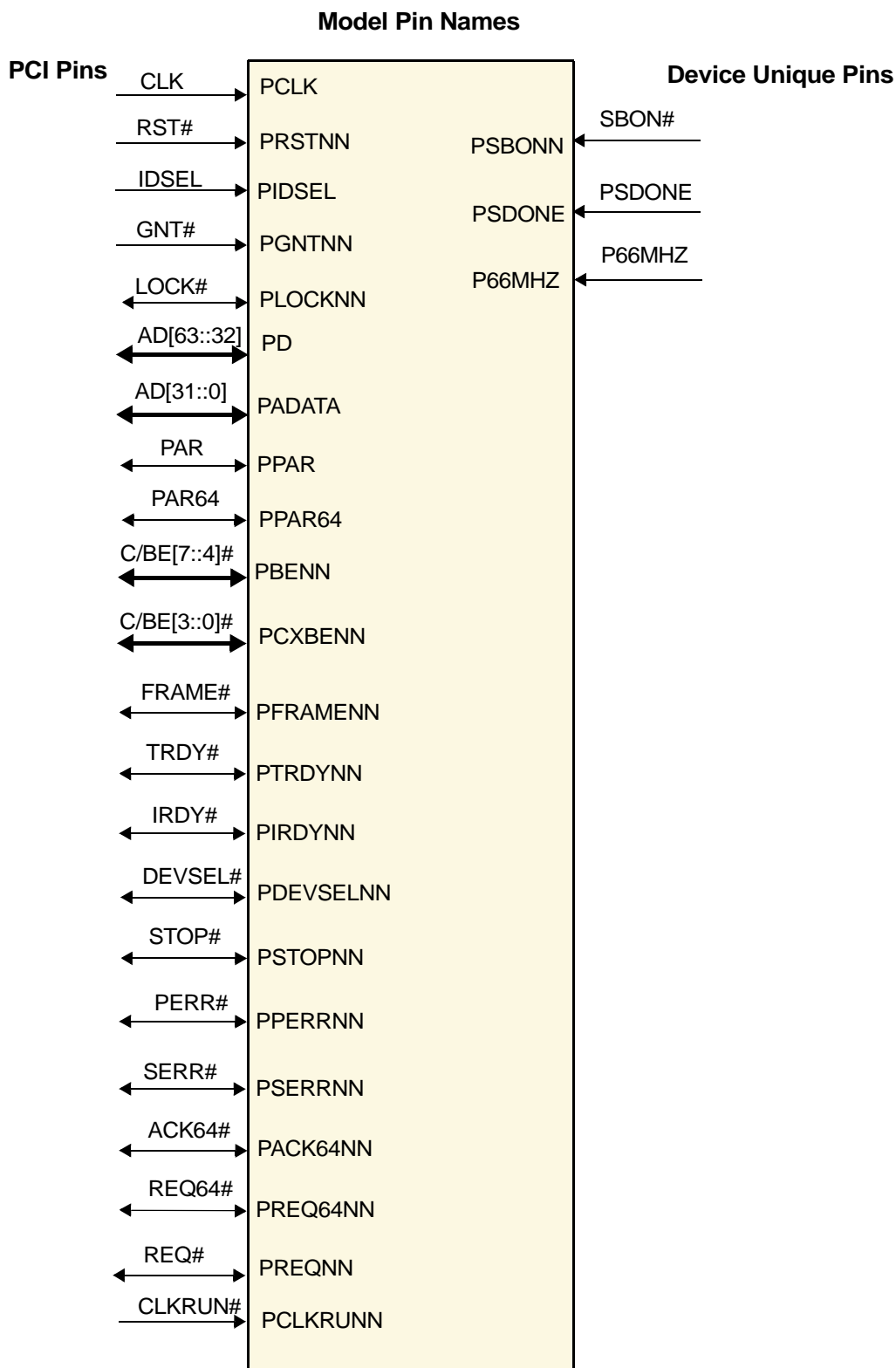


Figure 1: pcimaster_fx and pcislave_fx Pins

Table 1: pcimaster_fx and pcislave_fx PCI(X) Pins

Pin Name	Size	Type ¹	Function
PCLK	1	in	PCI Clock input (CLK).
PRSTN	1	in	PCI Reset input (RST#), active low, asynchronous.
PIDSEL	1	in	PCI Initialization Device Select input (IDSEL).
PGNTNN	1	in	PCI-X Grant input (GNT#), active low.
PLOCKNN	1	s/t/s	PCI LOCK signal. Active low.
PD	32	t/s	The PCI AD address/data bus has been divided into two separate busses on the model representing the upper and lower 32 bits of the bus. Thus AD[63::32] maps to the PD bus and AD[31::0] maps to PADATA.
PADATA	32		The PCI AD address/data bus has been divided into two separate busses on the model representing the upper and lower 32 bits of the bus. Thus AD[63::32] maps to the PD bus and AD[31::0] maps to PADATA
PPAR	1	t/s	PCI Parity signal (PAR) for AD[31:0].
PPAR64	1	t/s	PCI Parity Upper DWORD signal (PAR64) for AD[63::32].
PBENN	4	t/s	The PCI C/BE# bus has been mapped to two separate busses: C/BE[7::4]# maps to PBENN, and C/BE[0::3]# maps to PCXBENN.
PCXBENN	4	t/s	The PCI C/BE# bus has been mapped to two separate busses: C/BE[7:4]# maps to PBENN, and C/BE[0::3]# maps to PCXBENN.
PFRAMENN	1	s/t/s	PCI Cycle Frame signal (FRAME#), active low.
PTRDYNN	1	s/t/s	PCI Target Ready signal (TRDY#), active low.
PIRDYNN	1	s/t/s	PCI Initiator Ready signal (IRDY#), active low.
PDEVSELNN	1	s/t/s	PCI Device Select signal (DEVSEL#), active low.
PSTOPNN	1	s/t/s	PCI Stop signal (STOP#), active low.
1. Refer to <i>PCI Local Bus Specification, Rev. 2.2</i> for descriptions of PCI bus signal types and functions. For PCI-X functions, refer also to the <i>PCI-X Addendum to the PCI Local Bus Specification</i> .			

Table 1: pcimaster_fx and pcislave_fx PCI(X) Pins (cont.)

Pin Name	Size	Type ¹	Function
PPERRNN	1	s/t/s	PCI Parity Error signal (PERR#), active low.
PSERRNN	1	open drain	PCI System Error signal (SERR#), active low.
PACK64NN	1	s/t/s	PCI Acknowledge 64-Bit Transfer signal (ACK64#), active low.
PREQ64NN	1	s/t/s	PCI Request 64-Bit Transfer signal (REQ64#), active low.
PREQNN	1	out	PCI Request signal (REQ#), active low.
P66MHZ	1	in	PCI M66EN (66 MHz Enable) port, indicates that the device is operating on a 66-MHz PCI bus segment.
PCLKRUNN	1	s/t/s	PCI optional signal used as an input for a device to determine the status of CLK. Corresponds to the PCI defined CLKRUN signal.
PSBONN			Obsolete. Note, the model ignores this signal. Snooping was made obsolete in the version 2.2 of the PCI Specification.
PSDONE			Obsolete. Note, the model ignores this signal. Snooping was made obsolete in the version 2.2 of the PCI Specification
1. Refer to <i>PCI Local Bus Specification, Rev. 2.2</i> for descriptions of PCI bus signal types and functions. For PCI-X functions, refer also to the <i>PCI-X Addendum to the PCI Local Bus Specification</i> .			

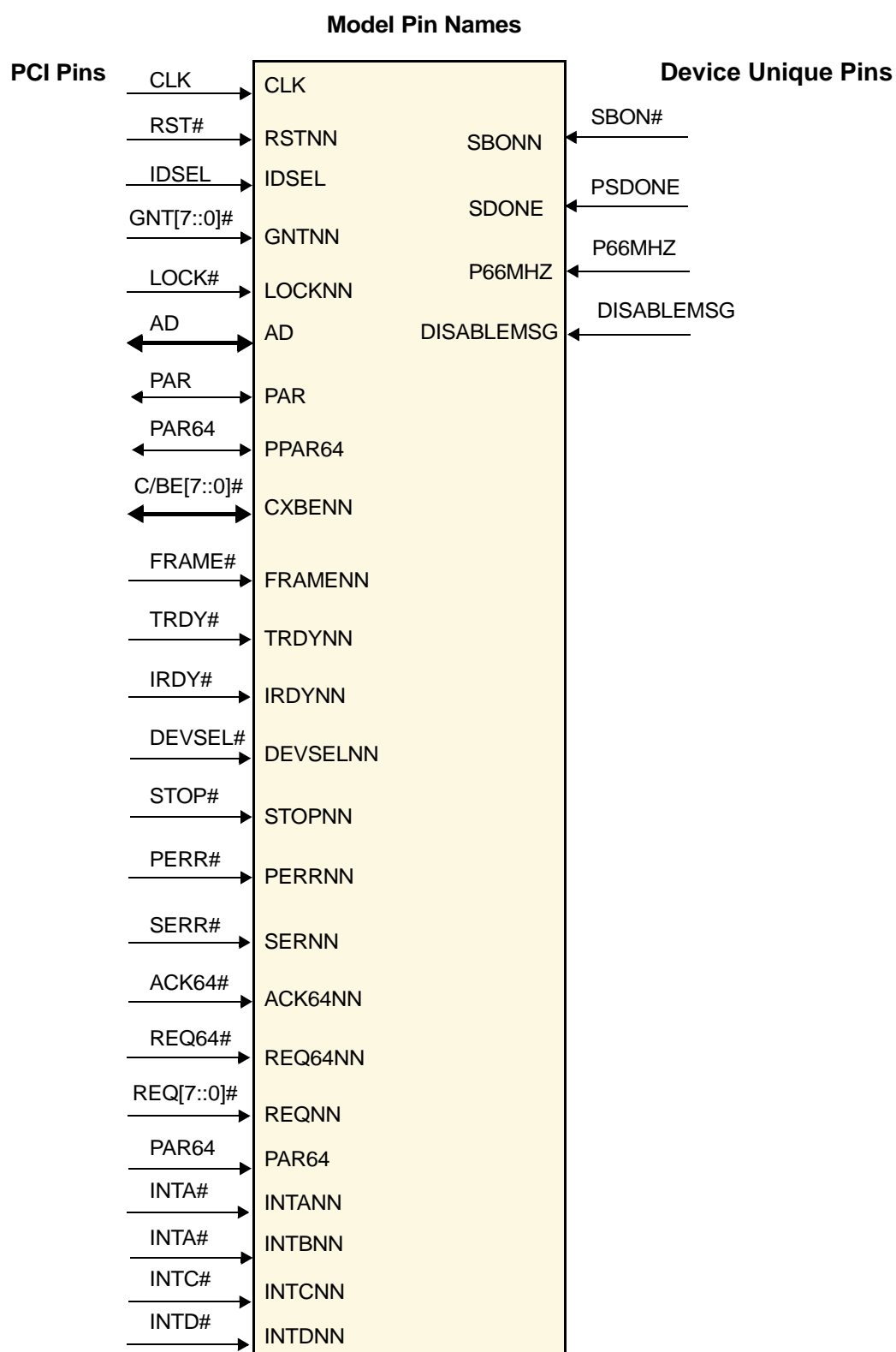
**Figure 2: pcimonitor_fx Pins**

Table 2: pcimonitor_fx PCI(X) Pins

Pin Name	Size	Type ¹	Function
CLK	1	in	PCI Clock input (CLK).
RSTN	1	in	PCI Reset input (RST#), active low, asynchronous.
IDSEL	1	in	PCI Initialization Device Select input (IDSEL).
GNTNN	8	in	PCI-X Grant input (GNT#), active low. On the pcimonitor_fx this is a 8-bit bus. Even if a user is not using the monitor for arbitration, the pins still need to be connected.
LOCKNN	1	s/t/s	PCI LOCK signal. Active low.
AD	32	t/s	PCI time multiplexed address and data bus.
PAR	1	t/s	PCI Parity signal (PAR) for AD[31:0].
PAR64	1	t/s	PCI Parity Upper DWORD signal (PAR64) for AD[63:32].
CXBENN	8	t/s	The PCI C/BE bus.
FRAMENN	1	s/t/s	PCI Cycle Frame signal (FRAME#), active low.
TRDYNN	1	s/t/s	PCI Target Ready signal (TRDY#), active low.
IRDYNN	1	s/t/s	PCI Initiator Ready signal (IRDY#), active low.
DEVSELNN	1	s/t/s	PCI Device Select signal (DEVSEL#), active low.
STOPNN	1	s/t/s	PCI Stop signal (STOP#), active low.
PERRNN	1	s/t/s	PCI Parity Error signal (PERR#), active low.
SERRNN	1	open drain	PCI System Error signal (SERR#), active low.
ACK64NN	1	s/t/s	PCI Acknowledge 64-Bit Transfer signal (ACK64#), active low.
REQ64NN	1	s/t/s	PCI Request 64-Bit Transfer signal (REQ64#), active low.
1. Refer to <i>PCI Local Bus Specification, Rev. 2.2</i> for descriptions of PCI bus signal types and functions. For PCI-X functions, refer also to the <i>PCI-X Addendum to the PCI Local Bus Specification</i> .			

Table 2: pcimonitor_fx PCI(X) Pins (cont.)

Pin Name	Size	Type ¹	Function
REQNN	8	in	PCI Request signal (REQ#), active low.
P66MHZ	1	in	PCI M66EN (66 MHz Enable) port, indicates that the device is operating on a 66-MHz PCI bus segment.
INTANN, INTBNN, INTCNN, INTDNN	1	in	PCI interrupt pins. Corresponds to INTA#, INTB#, INTC# and INTD#.
DISABLEMSG	1	in	Disable all messages. This is not a PCI defined pin.
PSBONN			Obsolete. Note, the model ignores this signal. Snooping was made obsolete in the version 2.2 of the PCI Specification.
PSDONE			Obsolete. Note, the model ignores this signal. Snooping was made obsolete in the version 2.2 of the PCI Specification.
1. Refer to <i>PCI Local Bus Specification, Rev. 2.2</i> for descriptions of PCI bus signal types and functions. For PCI-X functions, refer also to the <i>PCI-X Addendum to the PCI Local Bus Specification</i> .			

Figure 3 provides a high-level view of the PCI/PCI-X FlexModels in a FlexModel system testbench. Note that this illustration does not show the actual configuration of a system testbench that comes with your PCI/PCI-X FlexModels. For illustrations of the conventional mode PCI system testbenches, see Figure 4 on page 89 and Figure 5 on page 90. For illustrations of the PCI-X system testbenches, see Figure 6 on page 119 and Figure 7 on page 120.

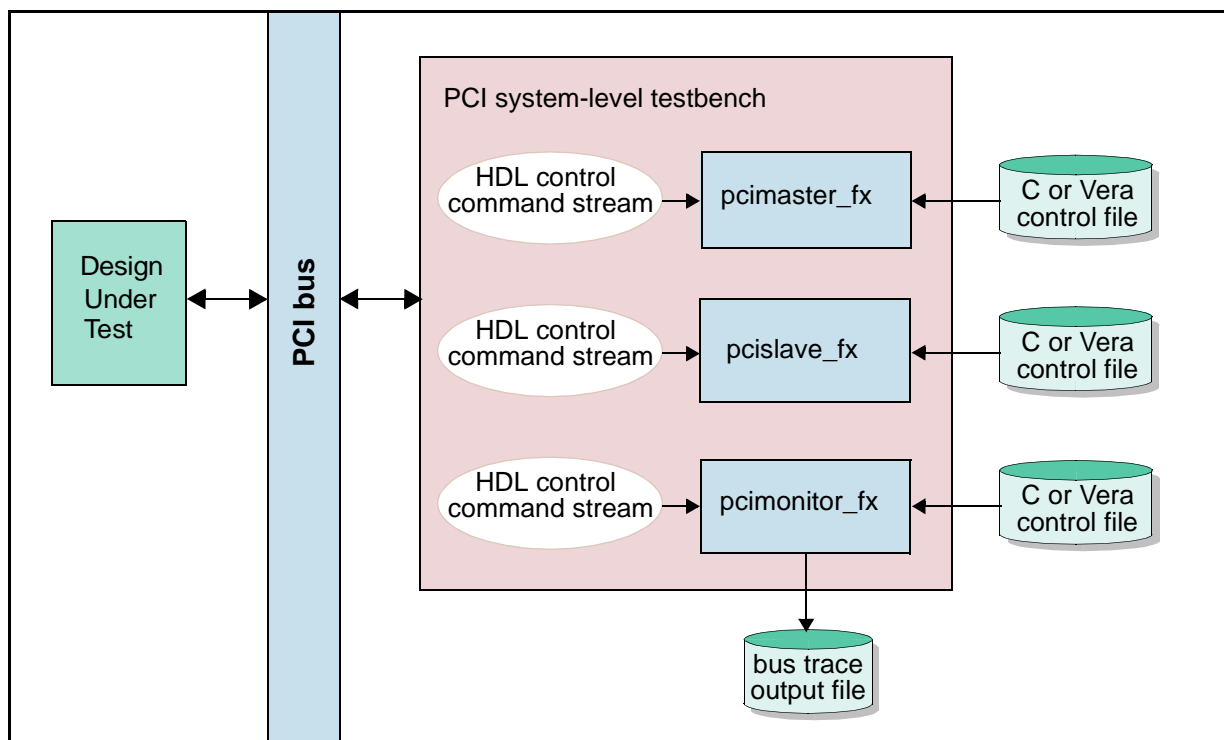


Figure 3: PCI/PCI-X FlexModels in a System Testbench (high-level view)

2

Programming PCI/PCI-X FlexModels

Introduction

This chapter describes how to program various aspects of PCI/PCI-X FlexModels. The topics covered in this chapter are:

- [“Setting up Your Simulator” on page 32](#)
- [“Setting the TimingVersion Generic/Defparam” on page 32](#)
- [“Migrating and Updating from Earlier Versions of the Models” on page 33](#)
- [“Using PCI FlexModels in PCI-X Mode” on page 33](#)
- [“Creating Your Design From Examples” on page 34](#)
- [“The PCI and PCI-X System Testbenches” on page 35](#)
- [“Model-specific Testbenches” on page 35](#)
- [“Instantiating a FlexModel into Your Design” on page 36](#)
- [“Instantiating a FlexModel into Your Design” on page 36](#)
- [“Working with Timing” on page 36](#)
- [“Troubleshooting Your Design” on page 37](#)
- [“PCI-X Transactions” on page 37](#)
 - [“DWORD Transactions” on page 37](#)
 - [“Burst Transactions” on page 39](#)
 - [“Master Terminations” on page 43](#)
 - [“Target Terminations” on page 45](#)
 - [“Split Transactions” on page 48](#)
 - [“Split Completion Messages \(SCM\)” on page 50](#)
 - [“Wait States and Delays” on page 51](#)
 - [“Parity Error Generation for the Master and Slave” on page 52](#)
 - [“64-bit Data Transfers” on page 57](#)
 - [“Dual Address Cycles” on page 59](#)

- “Decode Speeds” on page 60
- “Using the pcimonitor_fx to Track Bus PCI/X Transactions” on page 60
 - “pcimonitor_fx Control Pin” on page 60
 - “Generating the pcimonitor Trace File” on page 61
 - “Using the Trace File with the PCI Test Suite” on page 61
 - “Sample of a pcimonitor Trace File” on page 62
 - “Trace File Report Clarifications” on page 63
 - “Fast Turn Around in PCIX Mode” on page 63
 - “Using Custom Arbiters with the pcimonitor_fx” on page 65
 - “Using Error Check Registers” on page 65
 - “Changing the pcimonitor GNT_ASSERTED_CLKS” on page 68
- “Proper Usage of the pcislave_request Command” on page 68
- “Proper Use of the pcislave_read_rslt Command” on page 71

Setting up Your Simulator

You can find detailed information on setting up various simulators for using FlexModels in the *Simulator Configuration Guide for Synopsys Models*. For information specific to setting up the PCI and PCI-X system-level testbenches, see “[VERA Testbench Examples](#)” on page 76 and “[Setting up the Verilog, C, and VHDL Language PCI-X System Testbenches](#)” on page 106.

Setting the TimingVersion Generic/Defparam

You can use 33MHz or 66MHz timing in conventional PCI mode, and 66MHz or 133MHz timing in PCI-X mode; therefore, there are two different settings for the TimingVersion generic/defparameter:

- To use 33MHz or 66MHz timing in conventional PCI mode, set the TimingVersion generic/defparameter of your PCI FlexModels to “conventional”. When TimingVersion is set to “conventional”, the model uses 33MHz timing when the P66MHz pin is set to 0, and 66MHz timing when the P66MHz pin is set to 1. The default is 0.
- To use 66MHz or 133MHz timing in PCI-X mode, set the TimingVersion generic/defparameter to “pcix”. When TimingVersion is set to “pcix”, the model uses 66MHz timing when the P66MHz pin is set to 0, and 133MHz timing when the P66MHz pin is set to 1. The default is 0.

**Note**

In PCI-X mode, the P66MHz pin can be considered either a slow or a fast pin. When set to true (1), the model uses a faster frequency (133MHz). When set to false (0), the model uses a slower frequency (66MHz). The name P66MHz was retained for backward-compatibility, and does not necessarily indicate that the model is operating at 66MHz.

For information about setting other generics/defparameters, see the *[Simulator Configuration Guide for Synopsys Models](#)*.

Migrating and Updating from Earlier Versions of the Models

You can use the PCI/PCI-X FlexModels in either PCI-X mode or conventional PCI mode. With a few exceptions, the models are backward-compatible when running in conventional PCI mode. [Appendix B on page 367](#) describes these exceptions and shows you how to make older testbenches fully compatible with PCI/PCI-X FlexModels.

As of May, 2002 the PCI/PCIX FlexModels will be released on a regular monthly schedule for bug fixes and enhancements. If there are no significant changes that would warrant a model release, then the interval may increase. The changes incorporated into each release are shown in the model history section of each datasheet (the last section). Bug fixes, enhancements and schedules for upcoming releases can be seen on the web at:

<http://www.synopsys.com/products/designware/vipstatus/pci.html>

You may elect to receive automatic email notification of model releases by following directions when you become a register user of Solvnet at <http://solvnet.synopsys.com>.

Using PCI FlexModels in PCI-X Mode

In PCI-X mode, PCI FlexModels are compatible with the .PCIX specification Release 1.0a

PCI-X mode is turned off by default. To turn PCI-X mode on, use the following command immediately after using the `flex_get_inst_handle` command:

```
pcimaster_configure(PCIMASTER_PCIX, FLEX_TRUE, status);
```

To turn PCI-X mode off, use the following command:

```
pcimaster_configure(PCIMASTER_PCIX, FLEX_FALSE, status);
```

The commands shown above are for the `pcimaster`, but you can substitute “slave” or “monitor” when using the command to turn PCI-X mode on or off for the `pcislave` or `pcimonitor`.

You can use either 66MHz or 133MHz timing in PCI-X mode. To specify the timing version, set the `TimingVersion` generic/default parameter to “pcix”. When `TimingVersion` is set to “pcix”, the model uses 66MHz timing when the `P66MHZ` pin is set to 0, and 133MHz timing when the `P66MHZ` pin is set to 1. The default is 0.

In PCI-X mode, you use the PCI-X system testbench (`pcixsys_tst.ext` or `pcixsys_c_tst.ext`) instead of the conventional mode PCI system testbench. The PCI-X system testbench demonstrates many PCI-X transactions and model behaviors, including DWORD transactions, split transactions, master and target terminations, and parity error generation. For details, see [Chapter 4, “Using the PCI-X System Testbench,”](#) on [page 105](#).

Creating Your Design From Examples

You can find command examples integrated in a working testbench in the PCI and PCI-X system-level testbenches and in model-specific testbenches. To integrate any of these examples into your design, simply copy them into your design testbench and modify as needed, or simply hook up your design to one of the system testbenches. For information on the model-specific testbenches, see [“Model-specific Testbenches”](#) on [page 35](#). For information on the system testbenches, see [“The PCI and PCI-X System Testbenches”](#) on [page 35](#), or [Chapter 3](#) on [page 73](#) and [Chapter 4](#) on [page 105](#).

You can also find command examples documented in the command reference section for each individual PCI FlexModel. Each command description includes an “Examples” section with one or more command examples. You can find command reference pages in three locations in this manual:

- The [pcimaster_fx Command Reference](#) is on [page 135](#).
- The [pcislave_fx Command Reference](#) is on [page 222](#).
- The [pcislave_fx Command Reference](#) is on [page 300](#).

The PCI and PCI-X System Testbenches

The conventional mode PCI system testbench (`pcisys_tst.ext` or `pcisys_c_tst.ext`) is a good starting point for using the PCI FlexModels in conventional PCI mode. The PCI-X system testbench (`pcixsys_tst.ext` or `pcixsys_c_tst.ext`) is a good starting point for using the PCI FlexModels in PCI-X mode. You can use the system testbenches to familiarize yourself with the operation of the models or as a template for building your own testbench. You can also use system testbenches to test the installation of your PCI FlexModels. In many cases, you can just replace one of the Synopsys PCI components with the design you want to test and get a quick start on your verification.

The system testbenches are examples of typical testbenches used for verifying PCI and PCI-X designs. Each testbench uses two `pcimaster_fx` models, two `pcislave_fx` models, and one `pcimonitor_fx` model. The testbenches contain many command examples to help you get started with PCI FlexModels.

For details on using the conventional mode PCI system testbench, see [Chapter 3 on page 73](#). For details on using the PCI-X system testbench, see [Chapter 4 on page 105](#).

Model-specific Testbenches

Each FlexModel comes with a model-specific example testbench. These standalone testbenches are installed with each model. The model specific testbenches are meant to show basic operations of each model. For an example of a working multi-device testbench which more realistically approximates what you will need, start with the PCI or PCIX testbenches. With the system testbenches you can replace the appropriate device with your IUT.

You cannot customize the command code in a model-specific testbench; use the PCI or PCI-X system testbench for this. [Table 3](#) lists the type of information contained in model-specific testbench files.

Table 3: Information Provided in Model-specific Testbenches

Location in File	Type of Information
Banner	Name, date, and purpose
Library “use” or “include”	N/A
Entity/architecture or module	Physical mapping
Constants/defines	ID, address
Instantiation	Port map

Table 3: Information Provided in Model-specific Testbenches (cont.)

Location in File	Type of Information
Processes/task-functions	System clock, ID, status
Command streams	Instance ID assignment, common model commands
Model-specific tests	Input and output tests

For more information on the files that are shipped with every FlexModel, and the data structures of these files within the model directory, see [“FlexModel File Structure”](#) in the *FlexModel User's Manual*.

Instantiating a FlexModel into Your Design

If you are not using a Synopsys example testbench and are instead creating your own testbench, you must instantiate the PCI FlexModels into your testbench before simulation. The PCI FlexModel directories contain instantiation examples in the sample testbench files included with each model. You can use these files as both templates and cut-and-paste resources. For a list of pathnames for individual testbench files, see [“PCI System Testbench Architecture, Files, and Setup Values”](#) on page 88 and [“PCI-X System Testbench Architecture and Setup Values”](#) on page 118.



Note

For information on instantiation changes required by the PCI-X enhancement, see [“Migrating and Updating from Earlier Versions of the Models”](#) on page 33.

Working with Timing

FlexModels are installed with typical timing information (delays and constraints). You can control this timing information using `model_set_timing_control` commands. You can also customize the timing information that the PCI FlexModels use by creating customized, user-defined timing files. For more information, see [“FlexModel Timing”](#) in the *FlexModel User's Manual*.

Troubleshooting Your Design

If you encounter problems using the PCI FlexModels in your design testbench, see [“Troubleshooting and Logging”](#) in the *FlexModel User's Manual* for information on troubleshooting techniques and model logging.

PCI-X Transactions

This section covers typical and critical verification tasks using the PCIX flexmodels. The command examples shown in this section are replicated in the actual testbench code. You can cut and paste these commands into your testbench. All the shown examples are based on a Verilog testbench. See the actual testbench code for similar examples in C, VHDL, and VERA.

The topics covered in this section include:

- [DWORD Transactions](#)
- [Burst Transactions](#)
- [Master Terminations](#)
- [Target Terminations](#)
- [Split Transactions](#)
- [Wait States and Delays](#)
- [Parity Error Generation for the Master and Slave](#)
- [64-bit Data Transfers](#)
- [Dual Address Cycles](#)
- [Decode Speeds](#)

DWORD Transactions

You can use the following PCI-X commands to execute DWORD transactions:

- Interrupt acknowledge
- Special cycle
- I/O read
- I/O write
- Memory read DWORD
- Configuration read
- Configuration write

You must issue DWORD transactions as 32-bit transactions; the length of the transaction cannot exceed one DWORD. Byte enable fields are controlled by the master, and any byte enable value specified in a *byten* parameter is driven during the cycle. For I/O and memory DWORD transactions, there are both legal and illegal combinations of the byte enable value and address, and the *pcimaster_fx* model allows you to specify an illegal combination.

In PCI-X mode DWORD transactions, the model ignores the *tc* parameter.

DWORD transaction examples

```
// DWORD Special Cycle
    pcimaster_write_special_cyc(id_10,4'h2, 32'h11111111, 3, `FLEX_FALSE,
        `FLEX_WAIT_F, status);

// DWORD Configuration Write
    pcimaster_write_cycle(id_10, `PCIMASTER_CONFIG_WRITE,32'h00000001, 1,
        4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD I/O Write
    pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h00000201, 1,
        4'h2, 32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD Read Cycles
// -----

// DWORD I/O Read
    pcimaster_read_cycle(id_10, `PCIMASTER_IO_READ, 32'h00000001, 1, 4'h2,
        32'h23232323, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
// DWORD Configuration Read
    pcimaster_read_cycle(id_10, `PCIMASTER_CONFIG_READ,32'h00000001, 1,
        4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD Memory Read
    pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ, 32'h0F000001, 1,
        4'h2, 32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD Interrupt acknowledge cycle
    pcimaster_read_intr_ack(id_10, 4'h2, 32'h10101010, 3,`FLEX_WAIT_F,
        status);
```

Burst Transactions

You can use the following PCI-X commands to execute burst transactions:

- Split completion
- Memory read block
- Memory write block
- Alias to memory read block
- Alias to memory Write Block

In these burst commands, the *tc* parameter specifies the number of bytes to be transferred. The *tc* parameter is combined with *ad[1:0]* to determine how many read/write_continue commands must follow the read/write_cycle command. You must follow the read_cycle or write_cycle command with an appropriate number of read_continue or write_continue commands. The number of read/write_continue commands must be enough to transfer the specified byte count after offsetting from *ad[1:0]*. If you issue more than the required number of read/write_continue commands, the pcimaster_fx issues an error message and ignores the extra commands. If you issue less than the required number, the model's behavior is unpredictable.

You specify the “wait mode” parameter of the necessary read/write and continue commands by using one of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence.

A typical burst starts with wait_mode set to FLEX_WAIT_F for all the read/write cycle and continue commands. If you wish to block the command stream after the burst, you should set the wait_mode to FLEX_WAIT_T for the last data phase. This can also be used to synchronize or coordinate the burst with other events. For a detailed explanation on command flow and timing, consult the *FlexModel User's Manual*.



Note

During a burst you should not place IDLEs or use multi-cycle commands before the last data phase. This will cause unpredictable results.

Following are a series of general examples illustrating the proper and improper usage of FLEX_WAIT. A way to avoid many problems with bursts is to place all read and write commands “back-to-back.”

Correct usage and an example of a typical burst cycle:

```
pcimaster_write_cycle( , , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );

pcimaster_read_cycle( , , , , , , FLEX_WAIT_F, );
...
...
```

Correct usage and an example of coordinating a burst with an event:

```
pcimaster_write_cycle( , , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
// Make the LAST write_continue FLEX_WAIT_T.
pcimaster_write_continue( , , , , , , FLEX_WAIT_T, );

pcimaster_read_cycle( , , , , , , FLEX_WAIT_F, )
```

The following example shows an incorrect use of FLEX_WAIT and leads to unpredictable results. Setting FLEX_WAIT_T in this pcimaster_write_cycle command causes the model to stop processing commands until the pcimaster_write_cycle completes. But the pcimaster_write_cycle can not complete on the bus without the requisite number of pcimaster_write_continue commands.

```
pcimaster_write_cycle( , , , , , , , FLEX_WAIT_T, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );

pcimaster_read_cycle( , , , , , , FLEX_WAIT_F, )
```

The following example leads to unpredictable results when a write_continue cycle is interrupted.

```
pcimaster_write_cycle( , , , , , , , FLEX_WAIT_T, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
// The following line leads to unpredictable data being read.
pcimaster_write_continue( , , , , , , FLEX_WAIT_T, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );

pcimaster_read_cycle( , , , , , , FLEX_WAIT_F, )
```


The following example leads to unpredictable results as idles are not allowed in a burst in this manner:

```
pcimaster_write_cycle( , , , , , , , FLEX_WAIT_T, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
// No IDLES during a burst.
pcimaster_idle( , , , )
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );
pcimaster_write_continue( , , , , , , FLEX_WAIT_F, );

pcimaster_read_cycle( , , , , , , FLEX_WAIT_F, , )
```

The *byten* parameter is ignored in all burst commands except memory write commands.

Burst transaction examples

```
// Write Cycles

// Write burst of 4 bytes. Note that wait_mode is set to FLEX_WAIT_F.
pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE, 32'h0FFF0000, 4,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// Write burst of 4 bytes
// Continue command needed because AD[1:0] = 01.
// Caution: Setting wait_mode to FLEX_WAIT_T for pcimaster_write_cycle will
// cause unpredictable results.

pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h0FFF0001,
    4, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_write_continue(id_10, 4'hF, 32'h01010101, 1, `FLEX_WAIT_F,
    status);

// Write burst of 80 bytes
for(i=0; i <= 20; i = i + 1) begin
    if (i == 0) begin
        pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK,
            32'h0FFF0000, 80, 4'h0, 32'h44444444, 3, `FLEX_FALSE,
            `FLEX_WAIT_F, status);
    end else begin // if (i == 0)
        pcimaster_write_continue(id_10, 4'h0, 32'h01010101 + i, 1,
            `FLEX_WAIT_F, status);
    end // else: !if(i == 0)
end // for (i=0; i <= 20; i = i + 1)
```

```

// Read cycles

// Read burst of 4 bytes
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0FFF0000, 4,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// Read burst of 4 bytes
// Continue command needed because AD[1:0] = 01.
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0FFF0001, 4,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'hF, 32'h01010101, 0, `FLEX_WAIT_F,
    status);

// Read burst of 80 bytes
for(i=0; i <= 20; i = i + 1) begin
    if (i == 0)

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0FFF0000,
    80, 4'h0, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F,
    status);
    else
        pcimaster_read_continue(id_10, 4'h0, 32'h01010101 + i, 0,
            `FLEX_WAIT_F, status);
end // for

```

When you use `pcislave_configure_delay` and `pcislave_request`, note the following. `pcislave_configure_delay` is intended to set a TRDY# delay value for a burst command data phase. For PCI-X, the only data phase that can have a delay is the first one. However, `pcislave_request` also sets TRDY# delay for the first data phase. When trying to use both commands, the `pcislave_request` command will always overwrite the TRDY# delay value set by the preceding `pcislave_configure_delay` command. The same is true for PCI; however, for PCI, the `pcislave_configure_delay` command can be used on other burst data phases than the first one. For PCI burst commands, the `pcislave_configure_delay` command is used for setting all data phases other than the first, or all data phases if no `pcislave_request` commands are present.

Master Terminations

The `pcimaster_fx` supports the following terminations in PCI-X mode:

- [Master Abort](#)
- [Disconnect at ADB](#)

Master Abort

In a master abort cycle, the pcislave does not respond to the pcimaster's cycle. To emulate this, you can program the `pcimaster_fx` to write or read from a memory region outside the slave's address space. In the PCI-X system testbench, the primary master demonstrates master aborts by writing and reading to locations outside the address range of both the primary and secondary slaves. If a transaction terminates with a master abort, the `pcimaster_fx` does not repeat the command. In a burst transaction, all subsequent continue commands are ignored.

The following example shows a master abort.

```
// DWORD Configuration Read
pcimaster_read_cycle(id_10, `PCIMASTER_CONFIG_READ, 32'h44556677,
    1, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
```

Disconnect at ADB

The initiator of a transaction is allowed to disconnect at ADB during a burst transfer. If the `pcimaster_fx` model is programmed to disconnect at ADB and there is more data left to transfer after the ADB, the `pcimaster_fx` model will continue the burst in another PCI-X cycle. This subsequent PCI-X cycle can also be disconnected at the programmed ADB. Note that if a master is programmed to disconnect at ADB, the configure command remains in effect until it is reset with another configure command.

The first example below shows a master that has been configured for a single disconnect at the first ADB. The second example below shows a master that has been configured for a disconnect at the second ADB.

Example of disconnect at first ADB

```
// Disconnect burst at 1st ADB
//1st ADB is 3 data phases from start of cycle
```

```

pcimaster_configure(id_10, `PCIMASTER_DISCONNECT_ON_ADB, 1, status);
pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h045566F9,
    32, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
    status);
pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
    status);

```

Example of disconnect at second ADB

```

// Disconnect burst at 2nd ADB
pcimaster_configure(id_10, `PCIMASTER_DISCONNECT_ON_ADB, 2, status);

// Burst Write cycle of 160 bytes
pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h045566F8,
    160, 4'h2, 32'hADB00000, 11, `FLEX_FALSE, `FLEX_WAIT_F, status);
for (i = 1; i <= 39; i = i + 1) begin
    pcimaster_write_continue(id_10, 4'h0, 32'hADB00000 + i, 0,
        `FLEX_WAIT_F, status);
end // for (i = 1; i <= 39; i = i + 1)

// Burst Read cycle of 160 bytes
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h045566F8,
    160, 4'h2, 32'hADB00000, 160, `FLEX_FALSE, `FLEX_WAIT_F, status);
for (i = 1; i <= 39; i = i + 1) begin
    pcimaster_read_continue(id_10, 4'h0, 32'hADB00000 + i, 0,
        `FLEX_WAIT_F, status);
end // for (i = 1; i <= 39; i = i + 1)
pcimaster_idle(id_10, 1, `FLEX_WAIT_T, status);
pcimaster_configure(id_10, `PCIMASTER_DISCONNECT_ON_ADB, 0, status);

```

Target Terminations

The pcislave_fx supports the following terminations in PCI-X mode:

- [Split Response](#)
- [Disconnect at Next ADB](#)
- [Target Abort](#)
- [Single Data Phase Disconnect](#)
- [Retry](#)

To emulate one of these terminations, you configure the pcislave_fx model using pcislave_configure commands before the pcimaster issues the cycle.

Split Response

The example below demonstrates a pcislave_fx issuing a split response termination in response to a read_cycle command issued by the pcimaster_fx.

```
// Program slave to respond to split response
pcislave_configure(id_12, `PCISLAVE_SPLIT_RESPONSE, `FLEX_TRUE, status);

// Delay split completion by 10 clocks
pcislave_configure(id_12, `PCISLAVE_SPLIT_COMPL_DELAY, 10, status);

// pcimaster issues cycle which is then terminated with split response.
// Note that if the cycle initiated by the master does not permit
// split response the target will ignore the split response command
// and complete the cycle as a normal termination.

// Write split transaction
pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h0, 4, 4'h2,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// Read split transaction
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0, 16, 4'h2,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
    status);
```

Disconnect at Next ADB

The disconnect at next ADB termination forces the slave to terminate a transfer at the specified ADB. If the command is issued too late for the current ADB and the slave can no longer terminate at that ADB, the transfer is terminated at the following ADB. Once the slave has signaled a disconnect at next ADB, it will continue to signal this type of termination until the pcimaster ends the transaction. Issuing the pcislave_configure command to set up the disconnect at ADB will not apply to transactions already in progress. The easiest way to ensure that the configuration parameters that you set in the slave take effect for the transactions you intend is to use the pcislave_request command. Refer to [“Proper Usage of the pcislave_request Command” on page 68](#) for additional information.

```
//Disconnect at Next ADB
pcislave_configure(id_12, `PCISLAVE_DISCONNECT_ON_ADB, 2, status);

// Burst Write cycle of 160 bytes
pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h005566F8,
    160, 4'h2, 32'hADB00000, 160, `FLEX_FALSE, `FLEX_WAIT_F, status);

    for (i = 1; i <= 39; i = i + 1) begin
        pcimaster_write_continue(id_10, 4'h0, 32'hADB00000 + i, 0,
            `FLEX_WAIT_F, status);
    end // for (i = 1; i <= 39; i = i + 1)

pcimaster_idle(id_10, 1, `FLEX_WAIT_T, status);

pcislave_configure(id_12, `PCISLAVE_DISCONNECT_ON_ADB, 1, status);

// Burst Read cycle of 160 bytes
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h005566F8, 160,
    4'h2, 32'hADB00000, 160, `FLEX_FALSE, `FLEX_WAIT_F, status);

    for (i = 1; i <= 39; i = i + 1) begin
        pcimaster_read_continue(id_10, 4'h0, 32'hADB00000 + i,
            0, `FLEX_WAIT_F,
            status);
    end // for (i = 1; i <= 39; i = i + 1)

pcimaster_idle(id_10, 1, `FLEX_WAIT_T, status);
```

Target Abort

The pcislave can issue target abort responses to the cycles issued to it. The pcimaster_fx will not repeat an aborted cycle. The example below demonstrates a target abort.

```
// Respond with target abort
pcislave_configure(id_12, `PCISLAVE_ABORT_LIMIT, 0, status);

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h04556670,
    16, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
    status);

pcislave_configure(id_12, `PCISLAVE_ABORT_LIMIT, 100, status);
```

Single Data Phase Disconnect

The pcislave_fx signals its intentions to complete a single data phase and then disconnect with a single data phase disconnect. In the following example, the pcislave issues a single data phase disconnect response to the cycles that are issued to it. The burst write cycle is performed as four separate single data phase transactions.

```
// The pcislave will issue a "Single data phase disconnect" response to
// the cycles that are issued to it.

pcislave_configure(id_12, `PCISLAVE_TRANSFER_LIMIT, 1, status);
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0, 16, 4'h2,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
    status);
pcimaster_read_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
    status);
```

Retry

The pcislave indicates that it is temporarily unable to complete a transaction by signaling a retry. In the following example, the pcislave issues a retry response to the cycles that are issued to it. The pcimaster_fx will repeat transactions terminated by a retry until it reaches the retry limit. After reaching the retry limit, the pcimaster discards the transaction.

```
// The pcislave will issue a "retry" response to the cycles that are issued
// to it. The pcimaster_fx will repeat transactions terminated by retry
// until the "retry_limit" parameter in the master is reached after which
// the transaction is discarded.

pcislave_configure(id_12, `PCISLAVE_TRANSFER_LIMIT, 0, status);
    // respond with retries
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_DWORD, 32'h0, 4, 4'h2,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
```

Split Transactions

The pcislave has the option of terminating the following transactions with a split response:

- Memory read block
- Alias to memory read block
- Memory read DWORD
- Interrupt acknowledge
- I/O read
- I/O Write
- Configuration read
- Configuration write

A user has the flexibility to configure the pcimaster_fx to handle split transactions in the following ways:

- **Multiple Outstanding Split Transactions.** A single pcimaster_fx can handle multiple split transactions from multiple pcislave_fx devices. After a slave device signals a split response to a pcimaster, the pcimaster continues to execute other testbench commands, including receiving split responses from different slave devices. The maximum number of outstanding split transactions handled by the master is 32. This is a fixed number and cannot be either decreased or increased.



Caution

You cannot use read_rslt commands if you configure a master to handle multiple split transactions.

- **Single Split Transaction.** A pcimaster handles only one split transaction. After receiving a split response from a slave device, the pcimaster_fx will not issue any transactions on the bus while awaiting a Split Completion. During the wait, the master device issues no additional commands.

For a single split transaction, once the pcislave signals a split response, the pcimaster behaves like a target and waits for the current split transaction to complete before issuing another transaction. The pcislave must complete the split transaction with a split completion cycle. The pcimaster can respond to this split completion with one of several termination types. You can specify this termination type using the `pcimaster_configure` command. (See the [pcimaster_configure](#) command description on [page 136](#).)

The pcislave can handle outstanding split completions from multiple masters, and is capable of returning split completions in a different order than the order in which the transactions were issued.

Note that in the case of an error which occurs during the attribute phase, the split completion transaction is required to abort. However, for this to happen, `PERR#` and `SERR#` must be enabled in the slave. If you do not enable `PERR#` and `SERR#`, then the model will assume that there is no error condition during the attribute phase. The model will then proceed with the split completion transaction.

You use the following command to configure the pcislave to signal a split response:

```
pcislave_configure(inst_handle, PCISLAVE_SPLIT_RESPONSE, FLEX_TRUE,
    status);
```

You can optionally also specify the number of clocks by which the pcislave is to delay before starting the split completion cycle. You do this using the following command:

```
pcislave_configure(inst_handle, PCISLAVE_SPLIT_COMPL_DELAY, cvalue,
    status);
```

where *cvalue* is the number of cycles by which to delay the split completion cycle.

Example of Multiple Outstanding Split Transaction

```
/* Configure the slave to perform a split reponse using the configuration
type PCISLAVE_SPLIT_RESPONSE. Next, configure the pcimaster_fx to perform
and accept multiple split transactions by using the
PCIMASTER_MULTIPLE_SPLITS configuration parameter. */
```

```
u3.pcislave_configure(id_12, `PCISLAVE_SPLIT_RESPONSE, `FLEX_TRUE, status);
u1.pcimaster_configure(id_10, `PCIMASTER_MULTIPLE_SPLITS, `FLEX_TRUE, status);
```

Example of single split transaction by primary master

```
pcislave_configure(id_12, `PCISLAVE_SPLIT_RESPONSE, `FLEX_TRUE, status);
// delay split completion by 10 clocks

pcislave_configure(id_12, `PCISLAVE_SPLIT_COMPL_DELAY, 10, status);

// pcimaster issues cycle which is then terminated with split response.
// Note that if the cycle initiated by the master does not permit split
// response the target will ignore the split response command and complete
// the cycle as a normal termination

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h04556670, 16,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F, status);
```

Example of two outstanding split transactions issued by the primary and secondary masters

```
pcislave_configure(id_12, `PCISLAVE_SPLIT_RESPONSE, `FLEX_TRUE, status);
// delay split completion by 10 clocks

pcislave_configure(id_12, `PCISLAVE_SPLIT_COMPL_DELAY, 10, status);

// Primary master issues split request
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'hA0, 16, 4'h2,
    32'h04556670, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F, status);
pcimaster_read_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F, status);

// Secondary master issues Split request
pcimaster_read_cycle(id_11, `PCIMASTER_MEM_READ_BLOCK, 32'h04556670, 4,
    4'h2, 32'h55555555, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```

Split Completion Messages (SCM)

The slave will generate Split Completion Messages (SCM) appropriate for a normal target device: that is, class 0 (zero), or class 2 error messages. These SCM's are issued when bus conditions trigger them. For example, a read cycle with an address and byte count such that the slave's memory address range will be exceeded would trigger the slave to terminate the split completion at the memory boundary and issue an SCME to indicate the error. The slave does not require any special configuration for this to happen.

The class and index value which a user must set to indicate the error condition are listed in [Table 39 on page 235](#). The following example shows how to tell the master device when a write data parity error occurred during a split completion:

```
// Set device for Message Class = 1
// Message Index = 2 (Write Data Parity Error)
pcislave_configure(id_12, `PCISLAVE_SCEM_CLASS, 'd1, status);
pcislave_configure(id_12, `PCISLAVE_SCEM_INDEX, 'd2, status);
```

From the code fragment, a user can see that the value of the class is 1 and the value of the index is 2. Looking at [Table 39 on page 235](#) this would indicate a write data parity error.

Users who are testing a master device that interacts with PCIX Bridge devices may want to test how their device reacts to SCM's that a bridge might generate. By using the parameters PCISLAVE_SC_MSG and PCISLAVE_SC_ERR, users can create SCM's and SCEM's typical of a bridge device.

When you set the PCISLAVE_SC_MSG parameter to a non-zero value, the model will return only a Split Completion message for any transaction terminated with a Split Response. The contents of the message will be the 32 bits specified by the cvalue as specified in the pcislave_configure command.

The PCISLAVE_SC_ERR will allow a user to turn on/off the Split Completion Error bit in the attributes of the Split Completion.

Because the default value of PCISLAVE_SC_MSG is zero, users cannot use this method to create Split Completion Messages (SCM) with the contents of the message being zero. See [“pcislave_configure” on page 226](#) for additional information on using the pcislave_configure command.

Wait States and Delays

Only PCI-X targets are allowed to issue wait states for PCI-X data transfers. In addition, wait states are permitted on only the first data phase of PCI-X transactions. The pcislave_fx can issue wait states using either the pcislave_request command or the pcislave_configure_delay command. Note, in the case of a conflict, the precedence is given to the values set up by pcislave_request over those of pcislave_configure_delay.

Using the `pcislave_request` command to issue wait states

You can use the `pcislave_request` command to issue wait states. The command's fourth parameter (*delay*) specifies the number of wait states to be issued. You can use the `pcislave_request` command to set up the DEVSEL speed and TRDY# delay at the same time.

```
// Delay trdy# by 3 clocks on 1st data phase
pcislave_request(id_12, 1, 0, 3, status);

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ, 32'h0FFF0000, 1, 4'h0,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```

Using the `pcislave_configure_delay` command to issue wait states

You can use the `pcislave_configure_delay` command to issue wait states. In PCI-X mode, wait states are allowed in the first data phase only, so the command's second parameter (*burst_num*) must have a value of 1 for PCI-X cycles. The command's third parameter (*cvalue*) specifies how many TRDY wait states to insert.

```
// configure 1st data phase to have a trdy# delay of 5
pcislave_configure_delay(id_12, 1, 5, status);

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ, 32'h0FFF0000, 1, 4'h0,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```

Parity Error Generation for the Master and Slave

You generate, detect, and report parity errors using configure commands. In PCI-X mode, you can control error generation on any data phase of the PCI-X cycle, including the new PCI-X attribute phase. Both the master and slave have the option to configure a ctype to force the model to produce a parity error. [Table 4](#) shows the differences in parity error generation between the master and slave.

Table 4: Parity Error Generation with Master and Slave

Master	Slave
Drives PAR on writes. (Note, only PERR is subject to enable/disable.)	Drives PAR on reads.
Reacts to PAR on reads by driving PERR#/SERR# when enabled (see following example).	Reacts to PAR on writes by driving PERR#/SERR# when enabled (see following example).

Table 4: Parity Error Generation with Master and Slave (cont.)

Master	Slave
As an error injection feature, drives PERR#/SERR# independent of PAR for READS/Split Completion by using the PCIMASTER_PCI_ERROR/PCIMASTER_SPLIT_PCI_ERROR, respectively. When PCIMASTER_PCI_ERROR is set to 4 for PCIX mode (3 if in PCI mode) for emulating parity errors (driving PERR) on data phases during READ transactions, you can select individual data phases by setting PCIMASTER_PAR_ERR_DATA_PHASE to the single data phase you want to see the error on.	As an error injection feature, drives PERR#/SERR# independent of PAR for WRITES by using the PCISLAVE_PCI_ERROR configuration variable. When PCISLAVE_PCI_ERROR is set to 4 for PCIX mode (3 if in PCI mode) for emulating parity errors (driving PERR) on data phases during WRITE transactions, you can select individual data phases by setting PCISLAVE_PAR_ERR_DATA_PHASE to the single data phase you want to see the error on.
As an error injection feature, drives PAR and PAR64 to the incorrect value on data phases during WRITE transactions. Set PCIMASTER_PCI_ERROR to 1 to take advantage of this feature. In addition, you can select an individual data phase for the bad PAR and PAR64 values by setting PCIMASTER_PAR_ERR_DATA_PHASE to the single data phase you want to see the error on. Refer to page 141 for additional information on PCIMASTER_PCI_ERROR and on page 141 for PCIMASTER_PAR_ERR_DATA_PHASE.	As an error injection feature, drives PAR and PAR64 to the incorrect value on data phases during WRITE transactions. Set PCISLAVE_PCI_ERROR to 1 to take advantage of this feature. In addition, you can select an individual data phase for the bad PAR and PAR64 values by setting PCISLAVE_PAR_ERR_DATA_PHASE to the single data phase you want to see the error on. Refer to page 230 for additional information on PCISLAVE_PCI_ERROR and on page 230 for PCISLAVE_PAR_ERR_DATA_PHASE.
In addition, you can select only PAR or PAR64 to be driven to the incorrect value. Use the PCIMASTER_PAR_PAR64_SELECT parameter. See Table 5 on selecting bus halves.	In addition, you can select only PAR or PAR64 to be driven to the incorrect value. Use the PCISLAVE_PAR_PAR64_SELECT parameter. See Table 6 on selecting bus halves.
Creates parity errors (PAR) on the address and attribute phases of a transaction (except a Split Completion), or on the data phases of write transactions by using PCIMASTER_PCI_ERROR. Refer to Table 25 on page 137 or information on PCIMASTER_PCI_ERROR.	Creates parity errors (PAR) on the address and attribute phases of a Split Completion by using PCISLAVE_SPLIT_PCI_ERROR, or on the data phases of read transactions by using PCISLAVE_PCI_ERROR. Refer to Table 36 on page 220 or information on PCISLAVE_PCI_ERROR.

[Table 5](#) shows parity generation when using 32-bit and 64-bit devices and the pcimaster_fx. The ctype PCIMASTER_PCI_ERR must be first set before using PCIMASTER_PAR_PAR64_SELECT.

Table 5: Selecting Bus Halves for Parity (Used in Conjunction with PCIMASTER_PCI_ERR)

PCIMASTER_PAR_PAR64_SELECT	Result
1	PAR driven bad.
2	PAR64 driven bad.
0 or 3	PAR and PAR64 driven bad.

Table 6 shows parity generation when using 32-bit and 64-bit devices and the pcislave_fx. The ctype PCISLAVE_PCI_ERROR must be first set before using PCISLAVE_PAR_PAR64_SELECT.

Table 6: Selecting Bus Halves for Parity (Used in Conjunction with PCISLAVE_PCI_ERROR or PCISLAVE_SPLIT_PCI_ERROR)

PCISLAVE_PAR_PAR64_SELECT	Result
1	PAR driven bad.
2	PAR64 driven bad.
0 or 3	PAR and PAR64 driven bad.

There are a number of PCIX error checks (96, 97, 98, 99, 100) which check for errors involving PERR# and SERR#. By default those error checks are turned off. As an example, to activate the parity error checking for Error 96, use the following pcimonitor_fx command:

```
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE96_REG, 'b1,status);
```

You enable the master or slave to drive parity errors by configuring the PCI command register (configure address 4). Following is an example of using pcislave_set_addr to configure parity generation by the slave:

```
pcislave_set_addr(id_12, `PCISLAVE_CFG, 'h4, 'h143, status);
// 0001_0100_0011 - Enable SERR# and PERR#
```

This register can also be set by configure write cycles on the PCI(x) bus. The same register exists in the master. It can be set by a configure write cycle from another master. The pcimaster's equivalent to the pcislave_set_addr(...143) command is the pcimaster_configure(PCIMASTER_DEV_CNTL_CFG) command.

You enable the master or slave to drive parity errors by configuring the PCI command register (configure address 4). Following is an example of using pcislave_set_addr to configure parity generation by the slave:

```
pcislave_set_addr(id_12, `PCISLAVE_CFG, 'h4, 'h143, status);
// 0001_0100_0011 - Enable SERR# and PERR#
```

This register can also be set by configure write cycles on the PCI(x) bus. The same register exists in the master. It can be set by a configure write cycle from another master.

You can use the pcimonitor_fx to help check for parity errors. There are a number of PCIX error checks (96, 97, 98, 99, 100) which check for errors involving PERR# and SERR#. By default those error checks are turned off. To activate the previous error messages, use the following pcimonitor_fx commands:

```
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE96_REG, 'b1,status);
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE97_REG, 'b1,status);
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE98_REG, 'b1,status);
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE99_REG, 'b1,status);
pcimonitor_set_reg(id_1, `PCIMONITOR_ERROR_ENABLE100_REG, 'b1,status);
```

The monitor will issue the following error messages for the appropriate conditions:

- Error 96: If a device detects error on an address phase, the device must assert SERR#.
- Error 97: If a device detects error on an attribute phase, the device must assert SERR#.
- Error 98: PERR# should be asserted on the second clock after PAR64 and PAR are driven.
- Error 99: PERR#, a Sustained Tri-State signal must be driven high for 1 clock before being Tri-States (5.3)
- Error 100: PERR#, a Sustained Tri-State signal must be Tri-States after being driven high. (5.3)

The following examples show how you can control parity errors in various phases of a PCI-X transaction.

The generation of parity on a data phase is a two step process:

1. Set [PCIMASTER_PCI_ERROR](#) or [PCISLAVE_PCI_ERROR](#) to 1.

2. Use `PCISLAVE_PAR_ERR_DATA_PHASE` or `PCIMASTER_PAR_ERR_DATA_PHASE` to specify the data phase on which the error is to occur.

Example of pcimaster_fx generating parity errors on data, address, and attribute phases

```
// Note that the pcislave's parity error response bit has to be set
// for the pcislave to assert PERR# if it detects a parity error.
u3.pcislave_set_addr(id_12, `PCISLAVE_CFG, 'h4, 'h143, status);

// Generate parity error on address phase
ul.pcimaster_configure(id_10, `PCIMASTER_PCI_ERROR, 2, status);
// DWORD I/O Write
ul.pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h00000201, 1, 4'h9,
                        32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);

// Generate parity error on attribute phase
ul.pcimaster_configure(id_10, `PCIMASTER_PCI_ERROR, 3, status);
// DWORD I/O Write
ul.pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h00000201, 1, 4'h9,
                        32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);

// Generate parity error on data phase
ul.pcimaster_configure(id_10, `PCIMASTER_PCI_ERROR, 1, status);
ul.pcimaster_configure(id_10, `PCIMASTER_PAR_ERR_DATA_PHASE, 1, status);
// DWORD I/O Write
ul.pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h00000201, 1, 4'h9,
                        32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```

Example of no parity errors with pcimaster_fx

```
// No parity error. The value of PCIMASTER_PCI_ERROR is 0 indicating
// no parity.
ul.pcimaster_configure(id_10, `PCIMASTER_PCI_ERROR, 0, status);
// DWORD I/O Write
ul.pcimaster_write_cycle(id_10, `PCIMASTER_IO_WRITE, 32'h00000201, 1, 4'h1,
                        32'h00000000, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```


Example of pcislave_fx error parity error generation for read, attribute, 1st dataphase, and 2nd dataphase transactions

```
// Note that the pcimaster's parity error response bit has to be set for
// the pcimaster to assert PERR# if it detects a parity error.
u1.pcimaster_configure(id_10, `PCIMASTER_DEV_CNTRL_CFG, 'h140, status);

// Generate data parity error on Read cycles. Parity error generated
// on address phase
u3.pcislave_configure(id_12, `PCISLAVE_PCI_ERROR, 2, status);
u3.pcislave_request(id_12, 1, 0, 0, status);
u1.pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h04536678, 4,
    4'h2, 32'h45454545, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);

// Parity error generated on attribute phase
u3.pcislave_configure(id_12, `PCISLAVE_PCI_ERROR, 3, status);
u3.pcislave_request(id_12, 1, 0, 0, status);
u1.pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h0453667C, 4,
    4'h2, 32'h67676767, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);

// Parity error generated on 1st dataphase
u3.pcislave_configure(id_12, `PCISLAVE_PCI_ERROR, 1, status);
u3.pcislave_configure(id_12, `PCISLAVE_PAR_ERR_DATA_PHASE, 1, status);
u3.pcislave_request(id_12, 1, 0, 0, status);
u1.pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h04536670, 8,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
u1.pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_T,
    status);

// Parity error generated on 2nd dataphase
u3.pcislave_configure(id_12, `PCISLAVE_PCI_ERROR, 1, status);
u3.pcislave_configure(id_12, `PCISLAVE_PAR_ERR_DATA_PHASE, 2, status);
u3.pcislave_request(id_12, 1, 0, 0, status);
u1.pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, 32'h04536670, 8,
    4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
u1.pcimaster_read_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_T,
    status);
```

64-bit Data Transfers

In PCI and in PCI-X mode, 64-bit data transactions are permitted for only burst transfers. That is, 64-bit transfers in either specification are allowed only when you perform a burst transfer. ADBs are not affected by 64-bit transfers and AD[2] can be either 0 or 1. This is different from conventional PCI mode, in which AD[2] must be 0 for 64-bit access. The pcimaster_fx and pcislave_fx support 64-bit data transfers according to the new rules regarding 64-bit data in the PCI-X Addendum to the PCI Specification. For more information on these new rules, see the PCI-X specification.

The following examples demonstrate how the models support 64-bit transactions.

```
// Example 64 bit data transactions

// Write Cycles

pcislave_configure(id_12, `PCISLAVE_DATA_64, `FLEX_TRUE, status);
    // set up slave as 64bit slave

pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE64, base_ad, 8, 4'h2,
    64'h3333333344444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
    // Write burst of 8 bytes

pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK64, base_ad, 8,
    4'h2, 64'h3333333344444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
    // Write burst of 8 bytes

// Continue command needed because AD[1:0] = 01.
pcimaster_write_continue(id_10, 4'hF, 32'h01010101, 0, `FLEX_WAIT_F,
    status);

// Write burst of 80 bytes
    for(i=0; i <= 9; i = i + 1) begin
// need 10 transfers to transfer 80 bytes
        if (i === 0)
            pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK64,
                base_ad, 80, 4'h0, 64'h3333333344444444, 3, `FLEX_FALSE,
                `FLEX_WAIT_F, status);
        else
            pcimaster_write_continue(id_10, 4'h0, 64'h0101010123232323 + i,
                0, `FLEX_WAIT_F, status);
    end // for

// Read cycles

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK64, base_ad, 8,
    4'h2, 64'h3333333344444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
    // Read burst of 8 bytes

// Continue command needed because AD[1:0] = 01.
pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK64, base_ad, 8, 4'h2,
    64'h3333333344444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_continue(id_10, 4'hF, 64'h0101010123232323, 0, `FLEX_WAIT_F,
    status);
```

```
// Read burst of 80 bytes
for(i=0; i <= 9; i = i + 1) begin
    if (i === 0)
        pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK64,
            base_ad,80, 4'h0, 64'h3333333344444444, 3, `FLEX_FALSE,
            `FLEX_WAIT_F, status);
    else
        pcimaster_read_continue(id_10, 4'h0, 32'h0101010123232323 + i,
            0, `FLEX_WAIT_F, status);
end // for
```

Dual Address Cycles

All devices that support PCI-X memory commands must support 64-bit address spaces. The `pcimaster_fx` model initiates 64-bit address cycles by using the `pcimaster_configure(PCIMASTER_DUAL_AD,...);` command. The `pcislave` supports 64-bit addresses by using the `pcislave_configure(ADDR64, ...);` command. The following examples show how to set up dual address transactions.

```
// Dual Address Cycle

// set upper address of dual address to "10000000"
pcimaster_configure(id_10, `PCIMASTER_DUAL_AD, 32'h10000000, status);

pcislave_configure(id_12, `PCISLAVE_ADDR_64, `FLEX_TRUE, status);
    // set up slave as 64bit addressable

pcimaster_read_cycle(id_10, `PCIMASTER_MEM_READ_BLOCK, base_ad, 4, 4'h2,
    32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
    // read to 64 bit address "10000000FFFF0000"
```

Decode Speeds

The `pcislave_fx` claims transactions by asserting DEVSEL. The `pcislave_fx` supports DEVSEL timings of Decode A, Decode B and Decode C. This is equivalent to 2, 3, and 4 clocks after the address phase. The following example shows how to set up the `pcislave`'s decode speed.

```
// Decode speed example

pcislave_request(id_12, 1, 0, 0, status) ; // Use decode A

pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h0FFF0000, 1,
    4'h0, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);

pcislave_request(id_12, 1, 3, 0, status) ; // Use Subtractive Decode

pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE, 32'h0FFF0000, 1, 4'h5,
    32'h44554455, 3, `FLEX_FALSE, `FLEX_WAIT_T, status);
```

Using the `pcimonitor_fx` to Track Bus PCI/X Transactions

The following sections provide you information and show you code on how to use the `pcimonitor_fx` to track transactions on the PCI/X bus. The `pcimonitor` contains an extensive list of error and warning messages. Refer to [“PCI Error Checks” on page 289](#) and [“PCI-X Error Checks” on page 293](#) a list of error messages produced by the `pcimonitor` when bus and protocol errors occur in your testbench.

`pcimonitor_fx` Control Pin

The `pcimonitor_fx` has only one control pin (signal):

DISABLEMSG	Specifies a positive integer that is used to control when trace messages will be written to the output file specified by the <code>pcislave_fx</code> FlexModelId generic/defparam. Setting DISABLEMSG to 1 stops the bus trace output; all other values enable the bus trace. The <code>pcimonitor_configure</code> command's PCIMONITOR_VALUETRACEFILE parameter must be set to FLEX_TRUE for the bus trace to be enabled.
-------------------	---

Generating the pcimonitor Trace File

You enable bus tracing by executing the following command:

```
pcimonitor_configure(PCIMONITOR_VALUETRACEFILE, FLEX_TRUE, status);
```

This causes the pcislave_fx to log bus activity to the output file pcimonitor_*FlexModelId*.lst, where *FlexModelId* is the value of the FlexModelId generic/defparameter.

Using the Trace File with the PCI Test Suite

When using the pcislave_fx with the PCI Test Suite, you must do the following:

- Make sure the pcimonitor_fx is in conventional PCI mode. Conventional PCI mode is the default mode, but if you have turned on PCI-X mode, you must turn it back off. The monitor should always be in the same mode as the testbench.
- Enable bus tracing as shown in “[Generating the pcimonitor Trace File](#).”
- Set the pcislave_fx FlexModelId generic/defparam to 5. This gives the trace file a filename of pcimonitor_tst.lst, the same filename used in the PCI Test Suite.
- Set the pcimonitor's message level to PCIMONITOR_TESTSUITE_LST using the following command:

```
pcimonitor_set_msg_level(Id_1, PCIMONITOR_TESTSUITE_LST, status);
```

Sample of a pcimonitor Trace File

The following is an example of a pcimonitor bus trace file:

```

TIME (NS)      : 678
Command        : E  Memory Read Block
Address (SAC)   : 00000020
Master size     : 64
=====
TIME (NS)      : 681
Requester Attr  : 0000000a0
Total Byte Count: 0160
Requester ID    : 0000
=====
TIME (NS)      : 684
Speed          : Decode A
Target size     : 64
=====
TIME (NS)      : 687
Transfer Count  : 1
Trdy waits     : 0
C/BE[7:0]      : ff
Data Transferred : ffffffffffffffff
=====
TIME (NS)      : 690
=====
TIME (NS)      : 693
REQ[7:0]       : f7
Termination    : Split Response
=====
TIME (NS)      : 696
GNT[7:0]       : ff
=====
TIME (NS)      : 699
GNT[7:0]       : f7
=====
TIME (NS)      : 702
=====
TIME (NS)      : 705
Command        : C  Split Completion
Partial Lower Address : 20
Master size     : 64
=====

```

Trace File Report Clarifications

This section contains information clarifying certain reports within the tracefile which relate to monitor activity.

- When the target signals Disconnect at ADB, the monitor is reporting termination as Disconnect At ADB, even if the target disconnects before ADB. This situation may lead to confusion when the target actually disconnects before ADB. In the .lst file you may find termination reported as Disconnect at ADB even though it did not happen.

Fast Turn Around in PCIX Mode

When a master initiates a new transaction after only a single clock in PCIX mode, the monitor will detect this and print a message in the list file. The monitor marks the end of the previous transaction by the deassertion of IRDY. If there is only one clock between the deassertion of IRDY and the assertion of FRAME for the next transaction, then the monitor will print "Fast Turn Around" in the list file as follows:

```
*****
TIME (NS)      : 45
REQ[7:0]       : fe
Command        : F  Memory Write Block
Address (SAC)   : 00000050
Master size     : 64
=====
TIME (NS)      : 48
Requester Attr  : 041000840
Total Byte Cnt  : 64
Requester ID    : 0008
=====
TIME (NS)      : 51
Speed          : Decode A
Target size     : 64
=====
TIME (NS)      : 54
Transfer Count  : 1
Trdy waits     : 0
C/BE[7:0]      : ff
Data Transferred : 0000000033333333
=====
TIME (NS)      : 57
=====
TIME (NS)      : 60
=====
TIME (NS)      : 63
```

```

Termination      : Single Data Phase Disconnect
=====
Command          : F  Memory Write Block
Address (SAC)    : 00000058
Master size      : 64
Fast Turn Around
=====
TIME (NS)        : 66
Requester Attr   : 041000838
Total Byte Cnt   : 56
Requester ID     : 0008
=====
TIME (NS)        : 69
Speed            : Decode A
Target size      : 64
=====
TIME (NS)        : 72
Transfer Count   : 1
Trdy waits      : 0
C/BE[7:0]       : ff
Data Transferred : 0000000000ffeedd
=====
TIME (NS)        : 75
=====
TIME (NS)        : 78
=====
TIME (NS)        : 81
Termination      : Single Data Phase Disconnect
*****

```

There are two situations where the monitor will detect fast turn around. If the first transaction is a burst of four or more data phases, then it is common for master devices to assert FRAME# for the next transaction one clock after IRDY deasserts for the previous one. For bursts of four or more data phases, the master in the first transaction deasserts FRAME two cycles before IRDY. The pcimaster_fx will then issue transactions in this manner by default.

The second case is less common and occurs for transactions of fewer than four data phases. In this case, FRAME and IRDY deassert together, or FRAME deasserts one clock ahead of IRDY (see PCIX spec 2.11.1.1). The pcimaster_fx will issue transactions such as these only when the PCIMASTER_FAST parameter is set. For information on how to set PCIMASTER_FAST, refer to the section [“pcimonitor_configure” on page 301](#).

Using Custom Arbiters with the pcimonitor_fx

If you use your own arbiter, you may still use pcimonitor_fx. However, you must in this case disable arbitration in the pcimonitor_fx by using the pcimonitor_configure command. You must still connect the GNT pins to the monitor. This allows the monitor to check for any irregularities with bus arbitration. If you do not connect the GNT pins to the monitor, it will report “Error 39.” Refer to [“PCI-X Error Checks” on page 293](#) for additional information on error checking.

Using Error Check Registers

The pcimonitor_fx has eight error check registers as shown in [Table 7](#). There is a separate set for both PCI and PCIX. One set enables and records error and warning messages for PCI, and another set records and enables error and warning messages for PCIX.



Note

Starting with the January 2002 release, Synopsys is providing new registers which give users greater flexibility and control over PCI(X) error and warning messages ([Table 7, “pcimonitor_fx PCI and PCIX Error Registers” on page 65](#)). Although there are new registers, Synopsys still supports the old error registers PCIMONITOR_ERROR_REG AND PCIMONITOR_ERROR_ENABLE_REG, thus allowing you to keep the previous method without changing your testbenches.

Table 7: pcimonitor_fx PCI and PCIX Error Registers

Register Name	Data Type	Description
PCI Error and Warning Message Registers		
PCIMONITOR_PCI_ERROR_REG PCIMONITOR_PCI_ERROR n _REG	1-bit $n=127:0$ 128-bit vector	Records which PCI errors are present. Bit numbers correspond to the individual error checks as numbered in the list of “PCI Error Checks” on page 289 . Bit 0 is reserved. This register also controls the recording of errors which are common to PCI and PCIX mode. The common errors are displayed as “PCI/X.”
PCIMONITOR_PCI_WARNING_REG PCIMONITOR_PCI_WARNING n _REG	1-bit $n=127:0$ 128-bit vector	Records which PCI warnings are present. Bit numbers correspond to the individual warning messages as numbered in the list of “PCI Error Checks” on page 289 . Bit 0 is reserved.

Table 7: pcimonitor_fx PCI and PCIX Error Registers (cont.)

Register Name	Data Type	Description
PCIMONITOR_PCI_ERROR_ENABLE_REG PCIMONITOR_PCI_ERROR_ENABLE n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual PCI error checks. Bit numbers correspond to the individual error checks as numbered in the list of “ PCI Error Checks ” on page 289. Bit 0 is reserved. All error checks are enabled (set to 1) by default.
PCIMONITOR_PCI_WARNING_ENABLE_REG PCIMONITOR_PCI_WARNING n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual PCI warning messages. Bit numbers correspond to the individual warning messages as numbered in the list of “ PCI Warning Messages ” on page 298. Bit 0 is reserved. All warning messages are enabled (set to 1) by default.
PCIX Error and Warning Message Registers		
PCIMONITOR_PCIX_ERROR_REG PCIMONITOR_PCIX_ERROR n _REG	1-bit $n=127:0$ 128-bit vector	Records which PCIX errors are present. Bit numbers correspond to the individual error checks as numbered in the list “ PCI-X Error Checks ” on page 293 . Bit 0 is reserved.
PCIMONITOR_PCIX_WARNING_REG PCIMONITOR_PCIX_WARNING n _REG	1-bit $n=127:0$ 128-bit vector	Records which PCIX warnings are present. This register is currently reserved for future use when Synopsys adds PCIX Warnings.
PCIMONITOR_PCIX_ERROR_ENABLE_REG PCIMONITOR_PCIX_ERROR_ENABLE n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual PCIX error checks. Bit numbers correspond to the individual error checks as numbered in the list of “ PCI-X Error Checks ” on page 293. Bit 0 is reserved. All error checks are enabled (set to 1) by default.
PCIMONITOR_PCIX_WARNING_ENABLE_REG PCIMONITOR_PCI_WARNING n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual PCIX warning messages. This register is currently reserved for future use when Synopsys adds PCIX Warnings.
Previously Supported Error Registers		

Table 7: pcimonitor_fx PCI and PCIX Error Registers (cont.)

Register Name	Data Type	Description
PCIMONITOR_ERROR_REG PCIMONITOR_ERRORn_REG	1-bit $n=127:0$ 128-bit vector	Records which errors are present. Bit numbers correspond to the individual error checks as numbered in the list of “PCI Warning Messages” on page 298 or “PCI-X Error Checks” on page 293 . Bit 0 is reserved. While this register is still supported, Synopsys recommends using the previously listed error and warning registers as they provided greater flexibility and control.
PCIMONITOR_ERROR_ENABLE_REG PCIMONITOR_ERROR_ENABLEn_REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual error checks. Bit numbers correspond to the individual error checks as numbered in the list of “PCI Warning Messages” on page 298 or “PCI-X Error Checks” on page 293 . Bit 0 is reserved. All error checks are enabled (set to 1) by default. While this register is still supported, Synopsys recommends using the previously listed error and warning registers as they provided greater flexibility and control.

Each bit, starting from bit 1, corresponds to one of the numbered error checks in either the list of [“PCI Warning Messages” on page 298](#) or [“PCI-X Error Checks” on page 293](#). You can specify an entire register, or you can specify an individual bit of a register by appending the bit number to the register name. Bit 0 is reserved.

The [pcimonitor_set_reg](#) command sets the value of a specified register or register bit. The [pcimonitor_reg_req](#) command requests the model to read the value of a specified register or register bit. The [pcimonitor_reg_rslt](#) command returns the result of the [pcimonitor_reg_req](#) command.

Changing the pcimonitor GNT_ASSERTED_CLKS



Note

This section applies only to Test Scenario 1.14, and only if your FRAMENN pin changes two clocks after gntnn.

For Test Scenario 1.14, you might additionally have to change the constant GNT_ASSERTED_CLKS. This constant defaults to 1. If your framenn is asserted one clock after gntnn, you do not need to change the constant's value. However, if your framenn changes two clocks after gntnn, you must change the constant's value to 2.

Change this line:

```
pcimonitor_configure(id_1, PCIMONITOR_GNT_ASSERTED_CLKS, 1, status);
```

to

```
pcimonitor_configure(id_1, PCIMONITOR_GNT_ASSERTED_CLKS, 2, status);
```

Proper Usage of the pcislave_request Command

The primary purpose of the pcislave_request command is to insulate the model from pcislave_configure commands that might come in the middle of bus transactions and confuse the model. The pcislave_request command serves the same purpose in the slave as commands like pcimaster_write_cycle or pcimaster_read_cycle in the master. When the master receives one of these commands, the model turns its attention away from the testbench command stream and executes the prescribed transactions. These transactions are executed with the configuration that was specified *before* the <model>_write_cycle or <model>_read_cycle commands. Similarly in the slave, when the model receives a pcislave_request command, it turns its attention away from the testbench command stream and executes transactions on the bus using whatever configuration was set up with pcislave_configure commands that were received before the pcislave_request command was received.

The number of transactions that the slave will execute is specified by the request_limit parameter. For this purpose, a “transaction” means an assertion of DEVSEL by the slave. Split Response transactions do count against request_limit since the slave is not asserting DEVSEL in these cases. Split Completion transactions do not count against DEVSEL.

The additional parameters in `pcislave_request` specify the decode speed and Initial Target Delay (TRDY delay). After the model executes a `pcislave_request` command, the delays specified by the *decode* and *delay* parameters remain in effect until you explicitly change them by executing another `pcislave_request` command with new parameters.

When you execute a `pcislave_request` command for a burst transfer, the *delay* parameter value applies to only the first transfer of the burst. To specify TRDY# delay for subsequent transfers of a burst cycle in conventional PCI mode, use the `pcislave_configure_delay` command. In PCI-X mode, wait states are not allowed during subsequent transfers of a burst cycle.

The `request_limit` parameter has several important uses. First, the `request_limit` parameter blocks the command stream into the model for the requested number of transactions (DEVSEL driven by this slave). Second, once the `request_limit` expires, and if there are no commands being executed, the model automatically generates a new `pcislave_request` command with the same parameters except for `request_limit`, which will be set to a value of one (1). This is a convenient method to keep the slave active or alive during a simulation.

Following is a situation where the model generates internal `pcislave_request` commands:

```
pcislave_configure
pcislave_configure
pcislave_request // Explicit user command: request_limit = 1.
#delay_value    // Many transactions occurring during this interval.
                // Model generates internal pcislave_request(1)
                // commands.
```

At the time of *#delay*, the model will continue to process bus cycles indefinitely by virtue of its internally generated `pcislave_request` commands. Thus, it is not necessary to issue a `pcislave_request` command with a large `request_limit` value. Also, it is generally not necessary to calculate the exact number of cycles to be executed by the slave. In both cases, the internally generated `pcislave_request` commands will suffice.

**Note**

The internally generated `pcislave_request` commands can sometimes make it appear as though the slave has processed an explicit `pcislave_request` command, when in fact it has not. This can happen from `#delay` statements being intentionally or mistakenly (from parallel tasks) included in the testbench. For further reference, consult the FAQ section “[pcislave_fx](#)” on [page 397](#). While it is possible to make a simulation behave correctly in these circumstances, the testbench may be dependent on having “just the right timing” between `pcislave_configure` commands, the `#delay` insertion, and transactions on the bus. In general, robust testbenches never use `#delay`'s with the `pcislave_fx`. Whether a user inserts `#delays` and then a `pcislave_request` command, or some combination of the two, a testbench writer must then be aware of when the slave is processing transactions on the bus, and when it is receiving new `pcislave_configure` commands.

You should always use `pcislave_request` after a set of `pcislave_configure` commands. Certain internal configuration parameters are not set and read until you block the command stream with `pcislave_request`. In general, if you wish to reconfigure the slave during a simulation, the configuration command sequences should occur in the following pattern:

```
pcislave_configure
pcislave_configure
pcislave_request
    // slave executes request_limit number of transactions.
    // Note, you can set request_limit to a precise value, or
    // use a #delay and let the slave generate internal
    // pcislave_request commands.
```

**Caution**

Do not rely on internally generated `pcislave_request` commands to set configuration values. Always explicitly invoke a `pcislave_request` command after any `pcislave_configure` command. The model's configuration values are not guaranteed to be set and correct without an explicit `pcislave_request` command.

Because the `pcislave_request` command does not contain a `FLEX_WAIT` parameter, you must be careful about its usage in some cases: in particular, when `pcislave_request` is followed by a `pcislave_read_rslt` command. In this case, you may obtain erroneous results because the `pcislave_read_rslt` command may be processed before all the bus cycles generated by `pcislave_request` have completed. The solution is to add a command after `pcislave_request` which uses a `FLEX_WAIT_TRUE` setting to force completion of the bus cycles. A good command to use in this instance would be `pcislave_addr_req`. The command sequence would be as follows:

```
pcislave_request
pcislave_addr_req(FLEX_WAIT_TRUE)
pcislave_read_rslt
```

Because the `pcislave_request` command blocks the command stream in the model, the model will not get another command until the correct number of `DEVSEL` assertions. The flex model command core will continue to get commands from the command stream however. The command core will continue to grab commands until its internal queue is full. The command core in the PCI models has an internal command queue of 20k. A command with `FLEX_WAIT` set to `TRUE` will block the flex model command core from getting further commands until the model returns with the results from the command. Thus, `FLEX_WAIT` can block the execution of the simulation command stream.

Proper Use of the `pcislave_read_rslt` Command

The `pcislave_read_rslt` command retrieves data from a previous `pcislave_addr_req` command. This command is the second half of the command result pair. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or until previous commands complete execution.

If all previous commands have completed and the specified result is not available, the command completes with an error (a negative or 0 returned *status* value). If the `read_cycle` command did not complete normally (for example, after a target or master abort), the `read_result` command will return “XXXXXXX”.

Because the `pcislave_request` command does not contain a `FLEX_WAIT` parameter, you must be careful about its usage in some cases: in particular, when `pcislave_request` is followed by a `pcislave_read_rslt` command. In this case, you may obtain erroneous results because the `pcislave_read_rslt` command may be processed before all the bus cycles generated by `pcislave_request` have completed. The solution is to add a command after `pcislave_request` which uses a `FLEX_WAIT_TRUE` setting to force completion of the bus cycles. A good command to use in this instance would be `pcislave_addr_req`. The command sequence would be as follows:

```
pcislave_request
pcislave_addr_req(FLEX_WAIT_TRUE)
pcislave_read_rslt
```

The `pcimaster_fx` and the `pcislave_fx` models have a queue size of 20k for `addr_req`, `read_cycle`, and `read_continue` commands. The models queue up these commands when they do not have corresponding `read_rslt` commands. The master/slave will discard the results of `addr_req`, `read_cycle`, and `read_continue` commands without corresponding `read_rslt` commands after the 20k limit is reached. The `read_rslt` queue supports random access by the user (see `addr` mode and `tag` mode parameter descriptions). The `read_rslt` queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

3

Using the Conventional Mode PCI System Testbench

Introduction

The conventional mode PCI system testbench (*pcisys_tst.ext* or *pcisys_c_tst.ext*) is a good starting point for using the PCI FlexModels in conventional PCI mode. You can use this system testbench to familiarize yourself with the operation of the models or as a template for building your own testbench. You can also use the PCI system testbench to test the installation of your PCI FlexModels. In many cases, you can just replace one of the Synopsys PCI components with the design you want to test and get a quick start on your verification.

The PCI system testbench is an example of a typical testbench used for verifying PCI designs. The testbench uses two *pcimaster_fx* models, two *pcislave_fx* models, and one *pcimonitor_fx* model. You might not need to use all of the PCI FlexModels at the same time to test your design. For example, you might want to start with only your device and one PCI FlexModel in a testbench. The minimum configuration recommended for testing bus contention is two *pcimaster_fx* models, two *pcislave_fx* models, and one *pcimonitor_fx* model, one of which can be your device.



Note

This chapter documents the PCI system testbench, which operates in conventional PCI mode only. For information on using the PCI-X system testbench, see [Chapter 4 on page 105](#).

Model Behaviors Demonstrated in the Testbench

The PCI system testbench contains numerous command examples to help you get started with the PCI FlexModels. These command examples are labeled and documented in the testbench comment code.

The system testbench examples provide you with:

- Working examples of important model commands in a testbench environment.
- Model instance labels that identify corresponding `pcimaster_fx` and `pcislave_fx` commands, so you can see how the models and their commands interact.
- A way to test the models to verify that they are working properly.
- Instantiation examples.

Verilog, C, and VHDL Testbench Examples

[Table 8](#) lists some of the model behaviors demonstrated in the Verilog, C, and VHDL conventional mode PCI system testbench and identifies which model instantiations perform each behavior. The table also lists input or output files. The VERA conventional model PCI testbench is structured differently and is documented in the following section.

You can find the examples listed in the table within the models' command streams (for both the HDL and C testbenches) and in their C command files (for the C testbench only). To see which command stream or command file controls which model instantiation, see [Figure 4 on page 89](#) and [Figure 5 on page 90](#).

Table 8: Behaviors Demonstrated in C, Verilog, and VHDL Conventional Mode Testbenches

To see an example of...	Look at...
HDL testbench command execution	All instantiations
pcislave memory initialization using Verilog memory format (this example works for C, VHDL, and Verilog testbenches)	Primary slave (pcislave1_tst.dat)
pcimaster memory initialization using Verilog memory format; relationship between addresses in memory initialization file and addresses in pcimaster read commands	For 32-bit reads: primary master and primary slave, Example 0a For 64-bit reads: secondary master and primary slave, Example 27a Primary slave initialization file (pcislave1_tst.dat)
32-bit memory write/read: single data phase cycles with 0 to 3 IRDY# wait states	Primary master and primary slave, Examples 0 through 3
32-bit memory write/read: multiple data phase transactions with 0 to 3 IRDY# wait states	Primary master and primary slave, Example 4
32-bit memory write/read: target (slave) with subtractive, slow, medium, and fast decode of DEVSEL#	Primary master and primary slave, Examples 0 through 4
Slave termination between primary master and primary slave, including disconnects, aborts, and retries	Primary master and primary slave, Examples 5 through 9
Master abort	Primary master, Example 10
Bus cycle arbitration between two masters	Primary and secondary masters, primary and secondary slaves, Examples 11a through 15
Using returned data (ret_val) from read operation	Secondary master and secondary slave, Examples 16 through 17b
Using HDL constructs (looping, variables) with an HDL testbench	Secondary master, Examples 16 through 17b
Configuration write and read from secondary master to primary master	Primary master and secondary master, Example 18
Configuration write and read from primary master to primary slave	Primary master and primary slave, Example 19

Table 8: Behaviors Demonstrated in C, Verilog, and VHDL Conventional Mode Testbenches (cont.)

To see an example of...	Look at...
Forcing PCI parity errors using pcimaster_configure(pci_error...) commands	Primary master and primary slave, Examples 19 through 21
Forcing PCI parity errors using pcislave_configure(pci_error...) commands	Primary master and primary slave, Examples 22 through 26
64-bit memory read and write using dual-address cycle mode	Secondary master and primary slave, Example 27 and Example 29
32-bit memory write with dual-address cycle	Secondary master and primary slave, Example 28
Turning off dual-address cycle mode	Secondary master and primary slave, Example 30
Forced mismatch of memory read, producing message showing expected and actual value returned	Primary master and primary slave, Example 3
pcislave_dump_file command used in external file to dump contents of memory space	Secondary slave (pcislave2_mem.dmp)

VERA Testbench Examples

The VERA testbench is organized differently than the C, Verilog, and VHDL system testbenches. The VERA testbench breaks out critical functionality and examples into various class files. [Table 9](#) breaks out the class file names and the example functionality encapsulated in each of them.



Note

The PCI FlexModels in every conventional testbench are configured in the same way.

Table 9: Behaviors Demonstrated in the VERA Testbench By Class File

Class File Name	Functionality Shown
BURSTClass.vr	BURSTCycle Class. Tests read and write bursts. No split completions.
DECODEandWAITClass.vr	DECODEandWAIT Class. Tests various decode scenarios.

Table 9: Behaviors Demonstrated in the VERA Testbench By Class File

Class File Name	Functionality Shown
DO64BITClass.vr	DO64BIT Class. Tests 64-bit interactions.
DWORDClass.vr	DWORD Class. Test read and writer transactions using Dword length data.
INITClass.vr	INITIALIZE Class. Initializes all devices.
MASTERClass.vr	MASTERterm Class. Tests various types of master device terminations.
PARITYClass.vr	PARITYerrors Class. Setups various parity error scenarios.
READresultClass.vr	READresults Class. Shows and test the proper way to do read_rslt using addresses or command tags.
PCI_RandomizeClass.vr	Randomizecycle Class: shows the use of this VERA feature by randomizing read and write cycles.
GenericClass.vr	Genericcycle Class: allows the user to setup generalized read and write cycles.

Setting up the Verilog, C, and VHDL Conventional Mode System Testbenches

To run the `pcisys_tst.ext` or `pcisys_c_tst.ext` system-level testbench, you must compile all package files from `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`. To make it easier, you can use the `flexm_setup` script to set up a work directory that contains all the package files. The following steps include VHDL and Verilog invocations of the `flexm_setup` script. For details on using the `flexm_setup` script, see the [FlexModel User's Manual](#).

1. Run `flexm_setup`:

To copy all the files required to run the PCI system testbench, run the following commands:

```
% mkdir workdir
% $LMC_HOME/bin/flexm_setup -dir workdir pcislave_fx
% $LMC_HOME/bin/flexm_setup -dir workdir pcimaster_fx
% $LMC_HOME/bin/flexm_setup -dir workdir pcimonitor_fx
```

**Note**

In this step, you have installed all the PCI models into a common directory called “workdir”, which is any name you choose. This was chosen as a convenience. You can also install the models into separate directories. In this case you must provide the proper pathname in the following steps and procedures.

2. Compile the C command files (C mode users only):

If you are using the C mode version of the system testbench, compile the following C command files:

```
workdir/examples/C/pcimaster1_c_commands.c
workdir/examples/C/pcislave1_c_commands.c
workdir/examples/C/pcislave2_c_commands.c
```

Name the object files as follows:

```
pcimaster1_tst.ext
pcislave1_tst.ext
pcislave2_tst.ext
```

where *ext* is **a** for Unix platforms and **exe** for Intel NT.

**Note**

If you are using Microsoft Windows NT, you need to edit the `flex_run_program` command within the `pcisys_c_tst.ext` file to specify the **.exe** extension instead of the **.a** extension for the compiled programs above.

You also need to link in the appropriate library for the platform you are using. The following example shows how to link in the `-LBSD` library for the HP700 platform for the three C command files. For more information, see [“Compiling an External C File”](#) in the *FlexModel User's Manual*.

```
% /bin/c89 -o pcimaster1_tst.ext \  
-I $LMC_HOME/sim/C/src \  
-I workdir/src/C workdir/examples/C/pcimaster1_c_commands.c \  
workdir/src/C/hp700/pcimaster_pkg.o \  
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \  
-lBSD  
  
% /bin/c89 -o pcislave1_tst.ext \  
-I $LMC_HOME/sim/C/src \  
-I workdir/src/C workdir/examples/C/pcislave1_c_commands.c \  
workdir/src/C/hp700/pcislave_pkg.o \  
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \  
-lBSD  
  
% /bin/c89 -o pcislave2_tst.ext \  
-I $LMC_HOME/sim/C/src \  
-I workdir/src/C workdir/examples/C/pcislave2_c_commands.c \  
workdir/src/C/hp700/pcislave_pkg.o \  
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \  
-lBSD
```

3. Copy the files from the work directory:

Copy the files listed below from the *workdir* directory to the current working directory. The following example shows the copy commands for VHDL. For Verilog use the *verilog* directory, which is *workdir/examples/verilog*.

```
% cp workdir/examples/vhdl/pcislave1_tst.cfg .  
% cp workdir/examples/vhdl/pcislave1_tst.dat .  
% cp workdir/examples/vhdl/pcislave1_tst.io .  
% cp workdir/examples/vhdl/pcislave2_tst.cfg .  
% cp workdir/examples/vhdl/pcislave2_tst.dat .  
% cp workdir/examples/vhdl/pcislave2_tst.io .
```

4. Compile VHDL packages for simulation (VHDL only):

This step applies to VHDL users only. If you are using Verilog, see [Step 5](#).

You must make sure that all packages are compiled for simulation. The following is an example for running the MTI simulator on the Solaris platform. If you are running on an HP platform, the commands are slightly different.



Note

The following example is for setting up the HDL mode version of the system testbench. If you are setting up the C mode version of the testbench, you need to use the filename **pcisys_c_tst.vhd** instead of **pcisys_tst.vhd** in the last line of this example.

```
% cp $LMC_HOME/lib/sun4Solaris.lib/slm_mti.so .

% vlib mti_work
% vmap slm_lib ./mti_work
% vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
% vcom -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd

% vcom -work slm_lib workdir/src/vhdl/pcislave_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcislave_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_pkg.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcislave_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimonitor_fx_mti.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcisys_fx_comp.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcimonitor.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcislave.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcisys_tst.vhd
```

Note, the command:

```
vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
```

will work only on Unix. For PCNT use `slm_hdlc_nt.vhd`.

5. Compile Verilog packages for simulation (Verilog only):

This step applies to Verilog users only. If you are using VHDL, see [Step 4](#).

You must make sure that all packages are compiled for simulation. The following is an example for running the VCS simulator on the Solaris platform. If you are running on an HP platform, the commands are slightly different.



Note

The following example is for setting up the HDL mode version of the system testbench. If you are setting up the C mode version of the testbench, you need to use the file name **pcisys_c_tst.v** instead of **pcisys_tst.v** in the second line of this example.

```
% $VCS_HOME/bin/vcs \  
workdir/examples/verilog/pcisys_tst.v \  
workdir/examples/verilog/pcimaster.v \  
workdir/examples/verilog/pcimaster_fx_vcs.v \  
workdir/examples/verilog/pcislave.v \  
workdir/examples/verilog/pcislave_fx_vcs.v \  
workdir/examples/verilog/pcimonitor.v \  
workdir/examples/verilog/pcimonitor_fx_vcs.v \  
${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o -P \  
${LMC_HOME}/sim/pli/src/slm_pli.tab \  
-lmc-swift \  
+incdir+${LMC_HOME}/sim/pli/src +incdir+workdir/src/verilog \  

```

6. Simulate Your Design

If you are using VHDL, simulate your design as follows:

```
% vsim -lib slm_lib cfgtest
```

If you are using Verilog, simulate your design as follows:

```
% simv
```

Setting Up the VERA Conventional Testbench\

This section provides an example script which you can use to run the VERA system testbench for the VCS simulator. For detailed information on using VERA testbenches with FlexModels, consult the [Simulator Configuration Guide for Synopsys Models](#).



Note

A new and improved version of the VERA testbench was released in the July 2002 release. The scripts provided in the datasheet use the July 2002 release, and not earlier versions of the VERA testbench. The scripts in this section and other sections are meant as an example only, and must be edited to ensure proper execution.

VERA Testbench/Script Setup Tasks

The example script assumes the following:

- You will be using Solaris as the testbench platform. You may need to change the path to model object files as necessary. Following are the location for the object files for Solaris, HP-UX, and Linux which are critical for the testbench:
 - `${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/hp700.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/x86_linux.lib/slm_pli.o \`
- You have complete rights to the directory where you place the VERA testbench.
- You have a good LMC_HOME tree with the PCI model set: `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`.
- Your DISPLAY environmental variable is correctly set.
- LD_LIBRARY_PATH is properly set. If not do a “`setenv LD_LIBRARY_PATH 1`” on your command line.

Before you execute the example scripts, do the following:

- Create a directory to hold VERA testbench files. You can give this top level directory any name.

- Create a directory to hold all the compiled *.vro VERA files in your testbench directory. *This directory must be called file_vro.* For example:

```
mkdir vera_tb_folder
cd vera_tb_folder
mkdir file_vro
```

- Copy and paste the example script into a file and execute it from the directory you created to hold the testbench files.
- Change your location to your local testbench directory, and invoke the script from that location.

**Note**

Edit your script to remove any continuation backslashes before the flexm_setup utility. In some script environments, you cannot put a continuation character before the command. If flexm_setup has any problems with the backslash, you will get an error message stating the model name was not found.

The script has five parts. Each part is set off by comments in the script example. Edit each section as is appropriate for your environment if needed. The five sections are:

1. Setup environmental variables for the simulator, LMC_HOME tree, and VERA environment.
2. Setup testbench parameters. These are the *.io and *.cfg files used by the other testbenches.
3. Create the vera_user.o and the vera_local.dl files.
4. Create the *.vro files needed by the PCI FlexModels.
5. Invoke the VCS simulator and run the testbench.

**Note**

The text for the script has been made smaller than normal to allow you to copy and paste it from the PDF file. As a result, reading it on a terminal will be difficult unless you magnify the text with Adobe's Acrobat Reader.

Running the VERA Testbench with the VCS Simulator

```
#!/bin/csh -fx

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
setenv SSI_LIB_FILES ./vera_local.dl

# Setup VCS
setenv VCS_HOME /d/vcs61/vcs6.1
set path = (${VCS_HOME}/bin $path)
setenv VCS_SWIFT_NOTES 1

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /d/ae/work2/test/release
set path = (${LMC_HOME}/bin $path)
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " This part of the script just initialized the VERA_HOME, VCS_HOME, LMC_HOME."
echo " "
echo " Set: LD_LIBRARY_PATH, SNPSLMD_LICENSE_FILE, LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable."
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcislave* .

echo " "
echo " The run_setup_pci script copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCI_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory"

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
    -I$VERA_HOME/lib \
    ${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
    ${LMC_LIB_DIR}/vera_slm_pli.o

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."

##### CREATE *.VRO FILES NEEDED BY THE PCIX FLEXMODELS #####

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_InitClass.vr \
./file_vro/PCI_InitClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_BURSTClass.vr \
./file_vro/PCI_BURSTClass.vro || exit 1
```

```

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DECODEandWAITClass.vr \
./file_vro/PCI_DECODEandWAITClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DO64BITSClass.vr \
./file_vro/PCI_DO64BITSClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DWORDClass.vr \
./file_vro/PCI_DWORDClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_GenericClass.vr \
./file_vro/PCI_GenericClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_MASTERClass.vr \
./file_vro/PCI_MASTERClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_PARITYClass.vr \
./file_vro/PCI_PARITYClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_READresultClass.vr \
./file_vro/PCI_READresultClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_RandomizeClass.vr \
./file_vro/PCI_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_TARGETClass.vr \
./file_vro/PCI_TARGETClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pci_sys_tst.vr \
./file_vro/pci_sys_tst.vro || exit 1

```

```

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/1stmodel.vr \
./file_vro/1stmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftmodel.vr \
./file_vro/swiftmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/src/vera/pcimaster_pkg.vr ./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vera/pcislave_pkg.vr ./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vera/pcimonitor_pkg.vr ./file_vro/pcimonitor_pkg.vro || exit 1

echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pci models"
echo " "

##### INVOKE VCS AND RUN THE PCI TESTBENCH #####

${VCS_HOME}/bin/vcs -lmc-swift -RI \
` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcisys_vera_tst_top.v \
./file_vro/pci_sys_tst.vshell \
` $LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
` $LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vcs.v \
` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vcs.v \
` $LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
` $LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vcs.v \
$ {LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \
-P $ {LMC_HOME}/sim/pli/src/slm_pli.tab \
+incdir+$ {LMC_HOME}/sim/pli/src \
+incdir+` $LMC_HOME/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+` $LMC_HOME/bin/flexm_setup pcislave_fx`/src/verilog \
-l logfile\
-vera_dbind \
+vera_udf=./vera_local.dl +vera_mload=` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vera/pci_sys_test_top.vr1

```

Command Synchronization in the Verilog, C, and VHDL Testbenches

The conventional mode PCI system testbench includes four examples of how the `flex_synchronize` command synchronizes commands between the testbench and the primary and secondary masters. For information on how to use the `flex_synchronize` command, see the *FlexModel User's Manual*.

The `flex_synchronize` command examples used in the command streams for the primary and secondary masters are labeled with Tx and Rx tags to help you identify `flex_synchronize` commands with matching `sync_tag` parameters. The tag Rx1, for example, identifies a `flex_synchronize` command with a `sync_tag` value of 15, while the tag Tx1 identifies a corresponding `flex_synchronize` command with a `sync_tag` value of 15.

Here is a description of each of the four examples:

- Rx1/Tx1:

Rx1: The secondary master uses a “`flex_synchronize(id_11, 2, “sync_15”, 900, status);`” command in `COMMAND_STREAM_2`. This command is executed just after the secondary master configures itself.

Tx1: The primary master uses a “`flex_synchronize(id_10, 2, “sync_15”, 900, status);`” command in `COMMAND_STREAM_1`.

- Rx2/Tx2:

Rx2: The primary master uses a “`flex_synchronize(id_10, 2, “sync_5”, 900, status);`” command in `COMMAND_STREAM_1`. This happens just after primary master Tx1 (above).

Tx2: The secondary master uses a “`flex_synchronize(id_11, 2, “sync_5”, 900, status);`” command in `COMMAND_STREAM_2`.

- Rx3/Tx3:

Rx3: The second master uses a “`flex_synchronize(id_11, 2, “sync_18”, 900, status);`” command in `COMMAND_STREAM_2`. This command is executed just after the secondary master Tx2 (above).

Tx3: The primary master uses a “`flex_synchronize(id_10, 2, “sync_18”, 900, status);`” command in `COMMAND_STREAM_1`.

- Rx4/Tx4:

Rx4: The primary master uses a “flex_synchronize(id_10, 2, “sync_99”, 900, status);” command in COMMAND_STREAM_1. This happens just after primary master Tx3 (above).

Tx4: The second master uses a “flex_synchronize(id_11, 2, “sync_99”, 900, status);” command in COMMAND_STREAM_2.

PCI System Testbench Architecture, Files, and Setup Values

Figure 4 illustrates the HDL testbench configuration of the conventional mode PCI system testbench. Figure 5 illustrates the C testbench configuration of the conventional mode PCI system testbench.



Note

In the following diagrams, note that the VERA testbenches have only one command stream.

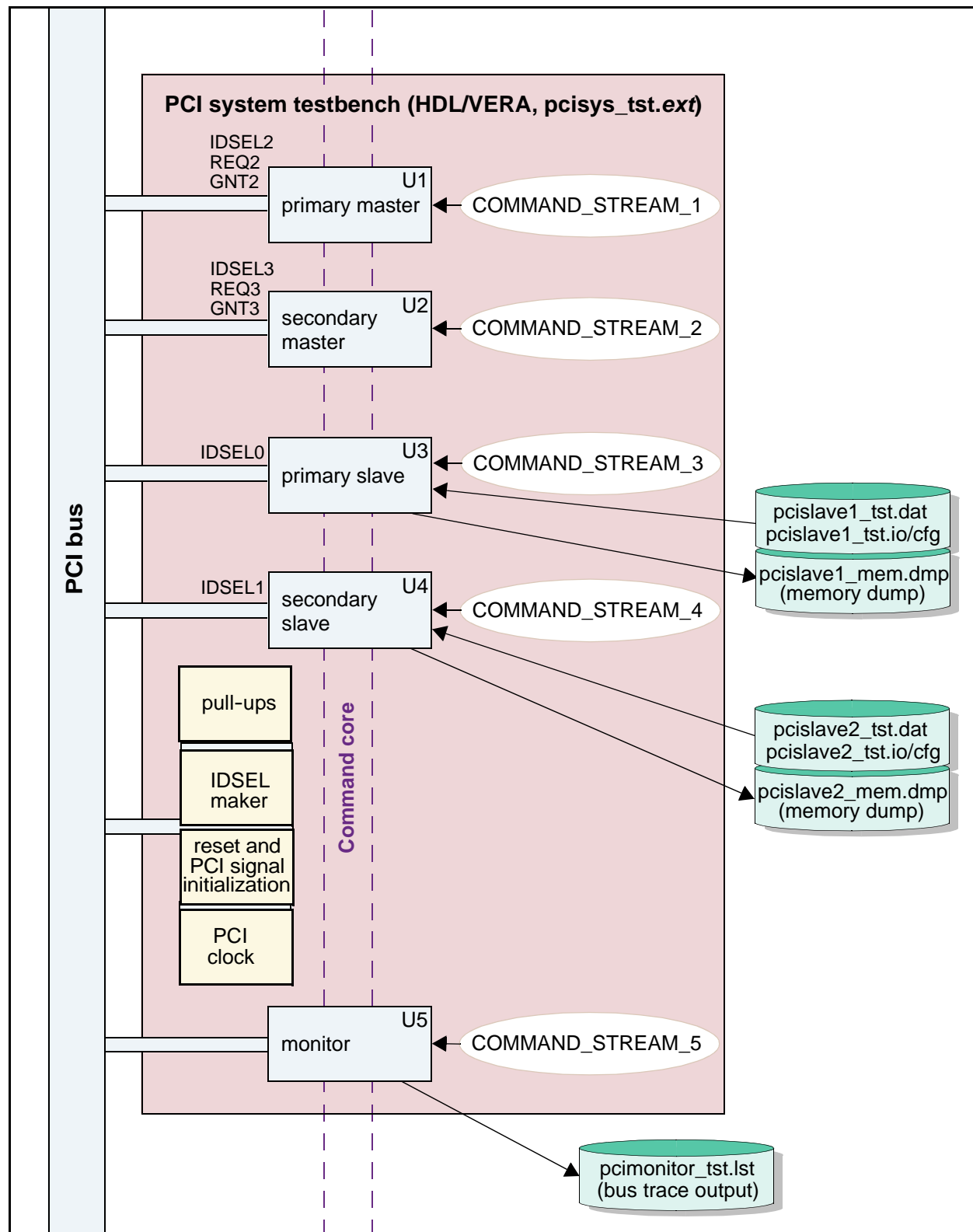


Figure 4: Conventional Mode PCI System Testbench—HDL/VERA Version

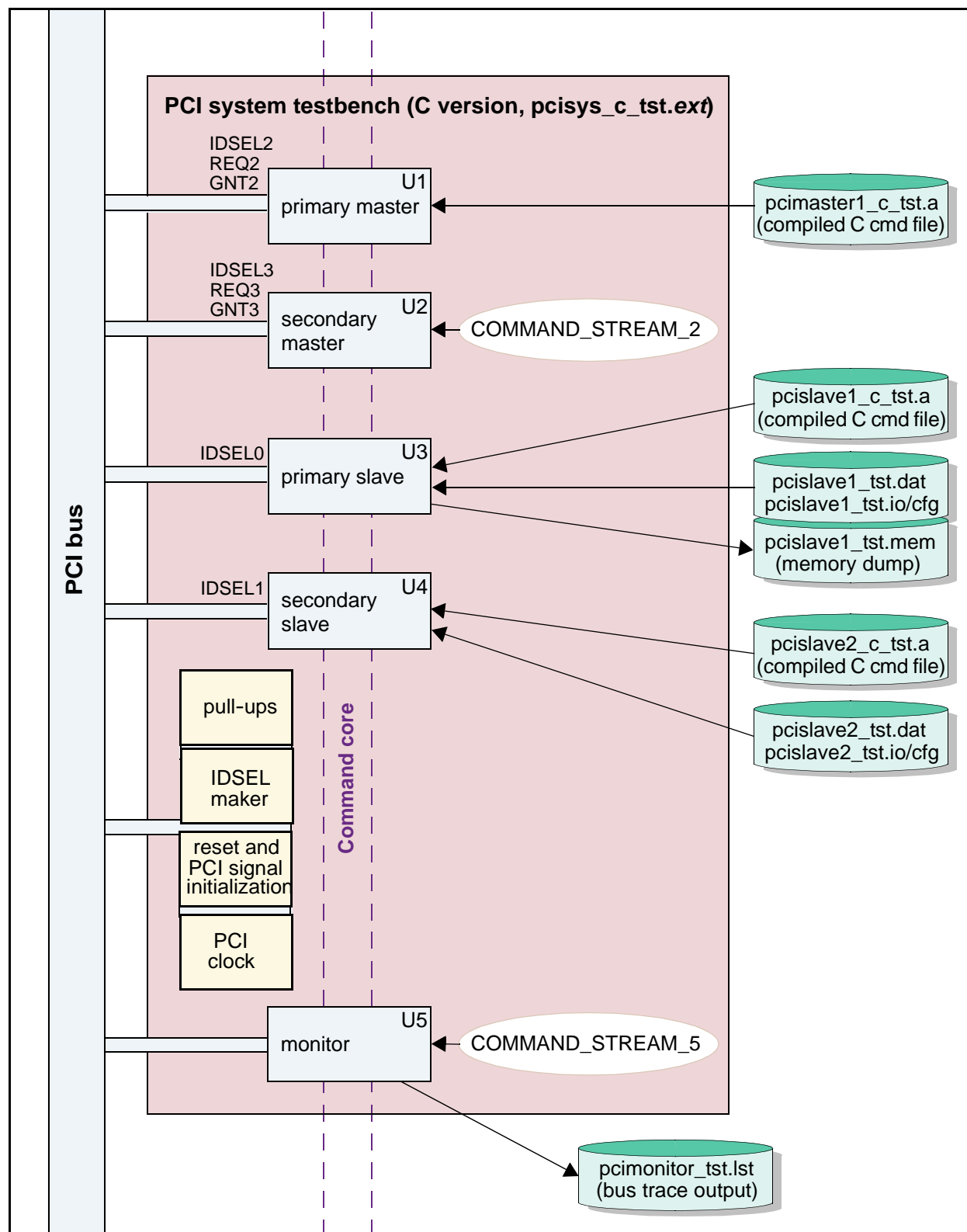


Figure 5: Conventional Mode PCI System Testbench—C Version

PCI System Testbench Files

The PCI system testbench files are listed below according to their location in \$LMC_HOME after installation.

- In \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/C:
 - **pcisys_c_tst.v** — c control verilog testbench
 - **pcisys_c_tst.vhd** —c control vhd testbench
 - **pcimaster1_c_commands.c** —master 1 c command file
 - **pcislave1_c_commands.c** — slave 1 c command file
 - **pcislave2_c_commands.c** — slave 2 c command file
 - **pcislave1_tst.cfg** — slave 1 configure space memory file
 - **pcislave1_tst.dat** — slave 1 memory space memory file
 - **pcislave1_tst.io** — slave 1 io space memory file
 - **pcislave2_tst.cfg** — slave 2 configure space memory file
 - **pcislave2_tst.dat** — slave 2 memory space memory file
 - **pcislave2_tst.io** — slave 2 io space memory file
- In \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/verilog:
 - **pcisys_tst.v**
 - **pcislave1_tst.cfg** — slave 1 configure space memory file
 - **pcislave1_tst.dat** — slave 1 memory space memory file
 - **pcislave1_tst.io** — slave 1 io space memory file
 - **pcislave2_tst.cfg** — slave 2 configure space memory file
 - **pcislave2_tst.dat** — slave 2 memory space memory file
 - **pcislave2_tst.io** — slave 2 io space memory file
- In \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fx<version>/examples/vhdl:
 - **pcisys_tst.vhd**
 - **pcislave1_tst.cfg** — slave 1 configure space memory file
 - **pcislave1_tst.dat** — slave 1 memory space memory file
 - **pcislave1_tst.io** — slave 1 io space memory file
 - **pcislave2_tst.cfg** — slave 2 configure space memory file
 - **pcislave2_tst.dat** — slave 2 memory space memory file
 - **pcislave2_tst.io** — slave 2 io space memory file

VERA PCI System Testbench Files

The PCI system testbench files for VERA are listed below according to their location in \$LMC_HOME after installation.

- Important source files for the VERA testbench in:
\$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/vera:
 - pcisys_tst.vr — top level of the Vera testbench.
 - pcisys_vera_tst_top.v — calling and instantiation of models.
 - pcisys_ver_tst_top.vhd — calling and instantiation of models.
 - class_dir — directory containing source files for the Vera classes called by the top level testbench:
 - PCI_BURSTClass.vrh — handles read/write burst transactions.
 - PCI_DECODEandWAITClass.vrh — tests various decode speed scenarios.
 - PCI_DO64BITSClass.vrh — tests 64-bit interactions.
 - PCI_DWORDClass.vrh — tests DWORD length transactions.
 - PCI_GenericClass.vrh — generic example test class.
 - PCI_InitClass.vrh — initializes all the instantiated models.
 - PCI_MASTERClass.vrh — handles and test various master device termination events.
 - PCI_PARITYClass.vrh — shows how to use the parity error features of both the master and slave models.
 - PCI_READresultClass.vrh — shows the proper way to do read_rslt transactions using command and memory tags.
 - PCI_RandomizeClass.vrh — class showing how to create random tests with VERA.
 - PCI_TARGETClass.vrh — shows a target device reacting to Split Transactions and various termination types.
- Data and configuration files in
\$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/vera
 - pcislave1_tst.cfg — slave 1 configure space memory file
 - pcislave1_tst.dat — slave 1 memory space memory file
 - pcislave1_tst.io — slave 1 io space memory file
 - pcislave2_tst.cfg — slave 2 configure space memory file
 - pcislave2_tst.dat — slave 2 memory space memory file
 - pcislave2_tst.io — slave 2 io space memory file

Configuration Details for the PCI System Testbench

The following are configuration details for the PCI System Testbench (*pcisys_tst.ext* or *pcisys_c_tst.ext*):

- The testbench includes instantiations of primary and secondary PCI masters, primary and secondary PCI slaves (or targets), and a PCI bus monitor.
- A 33-MHz clock drives all the models on the *pcisys_tst.ext* or *pcisys_c_tst.ext* testbench.
- The following model signals are connected to pull-ups:
 - PFRAMENN
 - PTRDYNN
 - PIRDYNN
 - PSTOPNN
 - PREQ64NN
 - PACK64NN
 - PLOCKNN
 - PDEVSELNN
 - PPERRNN
 - PSERRNN
 - PPAR64
 - PCXBENN(7 DOWNT0 4)
 - Padata(63 DOWNT0 32)
- The PREQNN and PGNTNN pins are used only in the PCIX mode. The pins should float in PCI mode.
- Reset is asserted low at 11 ns and deasserted at 41 ns. Reset remains deasserted for the rest of simulation. The signals PSDONE and PSBONN are driven high on the next rising PCLK edge after coming out of reset.

idsel_maker

The `idsel_maker` generates the id-selects for all PCI configuration cycles. The value of `PADATA`(31 down to 8) determines which IDSEL is asserted. The relationship between the model, `PADATA`, and IDSEL asserted is shown in [Table 10](#). Note that IDSEL generation is system-dependent. The `idsel_maker` system is only an example.

Table 10: IDSEL Assertion in the Conventional Mode Testbench

PCI FlexModels	PADATA (31 down to 8)	IDSEL
pcimaster (U1)	400000h	2
pcimaster (U2)	C00000h	3
pcislave (U3)	000000h	0
pcislave (U4)	800000h	1



Note

The PCI conventional testbench supports IDSEL Stepping as defined in the 2.3 version of the PCI Specification. The version of `idsel_maker` which supports IDSEL Stepping is commented out, and immediately follows the default version of `idsel_maker`. If you want to use IDSEL Stepping, comment out or delete the default version of `idsel_maker`, and then uncomment the 2.3 version of `idsel_maker`.

Primary PCI Master (U1) Characteristics

The following are characteristics of the primary PCI master:

- The `FlexModelId` generic/defparameter is set to “1”.
- In the HDL testbench, the primary master executes commands from `pcisys_tst.ext` `COMMAND_STREAM_1`. In the C testbench, the primary master executes commands from `pcimaster1_c_tst.a`.
- The primary master configures itself using `pcimaster_configure` commands with the *c*type and *c*value settings shown in [Table 11](#).

Table 11: Primary Master pcimaster_configure Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_TL	32	Corresponds to the PCI configuration space latency timer, which defines the minimum number of PCI clocks for which a master can own the bus.
PCIMASTER_RL	5	Specifies the number of retries the model will attempt.

Secondary PCI Master (U2) Characteristics

The following are characteristics of the secondary PCI master:

- The FlexModelId generic/defparameter is set to “2”.
- In the HDL testbench, the secondary master executes commands from pcisys_tst.ext, COMMAND_STREAM_2. In the C testbench, the secondary master executes commands from pcisys_c_tst.ext, COMMAND_STREAM_2.
- The secondary master configures itself using pcimaster_configure commands with the *ctype* and *cvalue* settings shown in [Table 12](#).

Table 12: Secondary Master pcimaster_configure Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_TL	64	Corresponds to the PCI configuration space latency timer, which defines the minimum number of PCI clocks for which a master can own the bus.
PCIMASTER_RL	5	Specifies the number of retries the model will attempt.

Primary PCI Slave (U3) Characteristics

The following are characteristics of the primary PCI slave:

- The FlexModelId generic/defparameter is set to “3”.
- In the HDL testbench, the primary slave executes commands from pcisys_tst.ext, COMMAND_STREAM_3. In the C testbench, the primary slave executes commands from pcislave1_c_tst.a.

- Memory and I/O spaces are defined by setting the lower and upper address values of a memory or I/O address range. The slave does not respond to address values outside of these ranges. The ranges can be set by using parameters or by using `pcislave_configure` commands in external or HDL testbench mode. All slaves have three memory spaces and three I/O spaces. The three memory spaces are referred to as “space 0” through “space 2” in the following diagrams.

Primary slave memory space mapping is set to:

```
space 0 0000000000000000 hex <= addr <= 00000000000000FF hex
space 1 1000000000000000 hex <= addr <= 10000000000000FF hex
space 2 2000000000000000 hex <= addr <= 20000000000000FF hex
```

Primary slave I/O space mapping is set to:

```
space 0 0000000000000200 hex <= addr <= 00000000000002FF hex
space 1 4000000000000000 hex <= addr <= 40000000000000FF hex
space 2 5000000000000000 hex <= addr <= 50000000000000FF hex
```

- The primary slave configures itself using `pcislave_configure` and `pcislave_load_file` commands. [Table 13](#) shows the *ctype* and *cvalue* settings used for the `pcislave_configure` command, and [Table 14](#) shows the *load_file* and *mem_space* settings used for the `pcislave_load_file` command.

Table 13: Primary Slave `pcislave_configure` Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_ADDR_64	FLEX_TRUE or 1	Causes the model to respond to 64-bit address cycles.
PCISLAVE_DATA_64	FLEX_TRUE or 1	Causes the model to respond to 64-bit data cycles.

Table 14: Primary Slave pcislave_load_file Command Settings

<i>load_file</i> Parameter Values	<i>mem_space</i> Parameter Values	Description
pcislave1_tst.cfg	PCISLAVE_CFG	Specifies the external file that initializes configuration space at time zero. The format is Verilog memory file format.
pcislave1_tst.dat	PCISLAVE_MEM	Specifies the external file that initializes memory space at time zero. The format is Verilog memory file format.
pcislave1_tst.io	PCISLAVE_IO	Specifies the external file that initializes I/O space at time zero. The format is Verilog memory file format.

Secondary PCI Slave (U4) Characteristics

The following are characteristics of the secondary PCI slave:

- The FlexModelId generic/defparameter is set to “4”.
- In the HDL testbench, the secondary slave executes commands from pcisys_tst.ext COMMAND_STREAM_4. In the C testbench, the secondary slave executes commands from pcislave2_c_tst.a.
- Memory and I/O spaces are defined by setting the lower and upper address values of a memory or I/O address range. The slave does not respond to address values outside of these ranges. The ranges can be set by using parameters or by using pcislave_configure commands in external or HDL testbench mode. All slaves have three memory spaces and three I/O spaces. The three memory spaces are referred to as “space 0” through “space 2” in the following diagrams.
- Secondary slave memory space mapping is set to:
 - space 0 00000000A0000000 hex <= addr <= 00000000A00000FF hex
 - space 1 00000000B0000000 hex <= addr <= 00000000B00000FF hex
 - space 2 00000000C0000000 hex <= addr <= 00000000C00000FF hex
- Secondary slave I/O space mapping is set to:
 - space 0 00000000D0000000 hex <= addr <= 00000000D00000FF hex
 - space 1 00000000E0000000 hex <= addr <= 00000000E00000FF hex
 - space 2 00000000F0000000 hex <= addr <= 00000000F00000FF hex
- The secondary slave configures itself using pcislave_configure and pcislave_load_file commands. Table 15 shows the *ctype* and *cvalue* settings used for the pcislave_configure command, and Table 16 shows the *load_file* and *mem_space* settings used for the pcislave_load_file command.

Table 15: Secondary Slave pcislave_configure Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_ADDR_64	FLEX_FALSE or 0	Causes the model to respond to 64-bit address cycles.
PCISLAVE_DATA_64	FLEX_FALSE or 0	Causes the model to respond to 64-bit data cycles.

Table 16: Secondary Slave pcislave_load_file Command Settings

<i>load_file</i> Parameter Values	<i>mem_space</i> Parameter Values	Description
pcislave2_tst.cfg	PCISLAVE_CFG	Specifies the external file that initializes configuration space at time zero. The format is Verilog memory file format.
pcislave2_tst.dat	PCISLAVE_MEM	Specifies the external file that initializes memory space at time zero. The format is Verilog memory file format.
pcislave2_tst.io	PCISLAVE_IO	Specifies the external file that initializes I/O space at time zero. The format is Verilog memory file format.

PCI Monitor (U5) Characteristics

The following are characteristics of the PCI monitor:

- The FlexModelId generic/defparameter is set to “5”.
- In the HDL testbench, the monitor executes commands from pcisys_tst.ext COMMAND_STREAM_5. In the C testbench, the monitor executes commands from pcisys_c_tst.ext COMMAND_STREAM_5.
- The monitor performs arbitration for the PCI bus (PCIMONITOR_ARBITRATE is set to true using the pcimonitor_configure command).
- The monitor uses arbitration scheme 0 (priority is set to 0 using the pcimonitor_configure command). Scheme 0 is a fixed scheme, with req(0) the highest priority.

- There is no bus parking when the PCI bus is idle (PCIMONITOR_PARKZERO is set to false using the pcimonitor_configure command). The monitor drives the lower 32 bits of Padata to 5a5a5a5a hex, the lower 4 bits of the command byte enable bus to 5 hex, and PAR to low. You can see an example of this when the idle(10); command is used in the primary master command stream (COMMAND_STREAM_1).
- Error checking is enabled (PCIMONITOR_ERRORCHECK is set to true using the pcimonitor_configure command).
- Bus tracing is enabled (PCIMONITOR_VALUETRACEFILE is set to true using the pcimonitor_configure command). This produces a PCI bus trace output file at the end of simulation. This output file is named pcimonitor_tst.lst. For information on naming monitor output files so that they are compatible with the PCI system testbench and PCI test suite, see [“Generating the pcimonitor Trace File” on page 61.](#))

PCI System Testbench Operation Sequences

This section describes the operations the pcisys testbench performs during simulation, including the operation sequence for the main test sequence (pcisys_tst.vhd or pcisys_tst.v) and operation sequences for the PCI components.

Main Test Sequence

The main test sequence is inside the pcisys testbench (pcisys_tst.vhd or pcisys_tst.v). Here is a description of what happens during the main test sequence:

- The top-level testbench performs a system reset.
- There are five command streams, one for each model instance:
 - COMMAND_STREAM_1 is for the primary master
 - COMMAND_STREAM_2 is for the secondary master
 - COMMAND_STREAM_3 is for the primary slave
 - COMMAND_STREAM_4 is for the secondary slave
 - COMMAND_STREAM_5 is for the monitor
- Each command stream starts with the one clock delay required for the FlexModel command core to initialize.
- Each command stream calls the *model_configure* command for its model instance.

Primary PCI Master (U1) Sequence

The following is a summary of events that occur during the primary master sequence.

1. Once the primary slave is configured, the primary master configures some of its own parameters.
2. Next, the primary master performs several write and read cycles with the primary slave. Examples 0 through 4 show how to change IRDY# wait states using pcimaster commands and how to configure the slave for different DEVSEL# decode speeds. Example 0a contains examples of reading back memory contents that have been initialized with a memory initialization file in Verilog format. Note that the addresses in the commands are four times larger than the addresses specified in the initialization file.
3. The primary master then sends write burst cycles to the primary slave. Example 5 demonstrates a target disconnect A cycle by sending a write burst command with two IRDY# wait states. Example 6 demonstrates a disconnect B cycle by sending a memory write burst to the primary slave.
4. The primary master demonstrates target retries by sending memory write cycles to the primary slave (Examples 8 and 9). The master then tries to write to an address location that is not mapped to any pcislave on the PCI bus, resulting in a master abort (Example 10). Once these examples are completed, the primary master issues a flex_synchronize(10, 2 "sync_15", 900, ST_pcimaster1_tst_cmd, status); command.
5. Now the primary and secondary masters are both using the PCI bus. The pcimonitor arbitrates the PCI bus cycles by controlling the masters' grant lines. (Note that there are several possible arbitration algorithms.) The primary master executes a couple of write bursts and some write and read I/O cycles. Once it completes the I/O cycles (Example 12), the primary master suspends all commands by issuing a sync command with a sync_label of "sync_5".
6. At this point, the secondary master has exclusive access to the PCI bus. After the secondary master finishes Examples 13 through 18, it sends a flex_synchronize command with a sync_label value of "sync_5" to the primary master and suspends its own command execution by issuing a flex_synchronize command with a sync_label of "sync_18". The primary master can now continue to execute commands.
7. The primary master performs several cycles to demonstrate parity errors PAR, SERR#, and PERR# (Examples 19 through 21).
8. Next, the primary master accesses the primary slave to demonstrate pcislave_configure(pci_error...) commands (Examples 22 through 26).

9. The primary master sends a flex_synchronize command with a sync_label of "sync_18" to the control bus and then issues a flex_synchronize(10, 2, "sync_99", 900, status); command. The secondary master sees the sync_label and begins to execute more bus cycles. Once the secondary master finishes, it sends a flex_synchronize(id_11, 2, "sync_99", 900, status); command so the primary master can resume command execution.
10. The primary master begins to execute its final commands, demonstrating several set and get commands (Example 31).

Secondary PCI Master (U2) Sequence

The secondary master executes its commands completely in HDL mode. Look in the pcimaster_2.vhd or pcimaster_2.v top-level model files to see HDL mode commands. Here is a description of what happens during the secondary master sequence:

1. The secondary pcimaster begins by configuring itself with a series of configure commands.
2. Once the secondary master has configured itself, it issues a flex_synchronize(id_1, 2, "sync_15", 900, status); command (see the Rx1 tag in the command stream code). This suspends the master's operations until another flex_synchronize command with the same sync_label is executed. In this case, the flex_synchronize command is sent by the primary master (see inside the primary master command stream for the corresponding Tx1 reference tag). This synchronizes the primary and secondary master commands. Although it is possible to run the primary and secondary master commands at the same time using arbitration, the flex_synchronize command gives you more control of when the commands will execute.
3. After the secondary master synchronizes itself with the "sync_15" sync_label, it begins to execute several memory writes and reads to the primary and secondary slaves (Examples 13 through 15).
4. The secondary master now executes an idle command, so that subsequent examples can demonstrate how to use ret_val with an HDL testbench (Examples 16, 17a, and 17b).
5. Once the secondary master completes the last write burst using looping constructs (Example 17b), it writes and reads to the configuration command register of the primary master (Example 18). The primary master's command register must be written to in order for it to respond with parity errors. (See the PCI specification for more information about the values of the configuration command register.)
6. After writing to the primary master's command register, the secondary master executes a flex_synchronize command with a sync_label of "sync_5". Immediately following this command, the secondary master goes into wait mode by issuing a

`flex_synchronize(id_11, 2, "sync_18", 900, status);` command (Rx3). The primary master takes over the bus and executes its commands. The primary master finishes up its commands by issuing a `flex_synchronize` command with a `sync_label` of "sync_18".

7. The primary master immediately issues a `flex_synchronize(10, 2, "sync_18", 900, ST_pcimaster_tst_cmd, status);` command and suspends command execution (Rx4), until a `flex_synchronize` command with the same `sync_label` is sent over the control bus.
8. After a `flex_synchronize(10, 2, "sync_99", 900, status);` is sent over the control bus, the secondary master comes to life and performs several command examples using the `configure(dual_ad);` command (Examples 27 through 30). This command sets the upper 32 bits of the PADATA bus for an additional address cycle. This command is required for executing 64-bit cycles, but optional for executing 32-bit cycles. Example 27a demonstrates reading a 64-bit address location that was initialized with the memory initialization file `pcislave1_tst.dat`. Note that the address specified in the command is four times larger than the address in the memory initialization file.
9. After the secondary master completes the dual address cycles, it issues a `flex_synchronize(id_11, 2, "sync99", 900, status);` command to the primary master (Tx4). The secondary master does nothing after this, because it has no more commands to execute.

Primary PCI Slave (U3) Sequence

The primary slave in the testbench is set up to run in HDL mode. The sequence contains many command and configuration examples. Here is a summary of the events that occur during the primary slave sequence:

1. In Examples 0 through 4, the primary slave uses the `pcislave request` command to change the `DEVSEL#` decode speed.
2. The primary slave then sets itself up to perform a disconnect A (Example 5) and a disconnect B (Example 6) by setting termination style to terminate with data and by setting a transfer limit of 1. Since the primary master's write burst cycles are set to perform two transfers, but the `pcislave` transfer limit is set to 1, the slave issues a disconnect.
3. The primary slave also sets itself up for a target abort (Example 7) by setting the abort limit to smaller than the transfer count of the associated burst operation.

4. Target retries are demonstrated in Examples 8 and 9. In Example 8, a retry is executed and allowed to complete successfully. In Example 9, the retries are not allowed to complete because the transfer limit is set to 0. The primary master now reaches the retry limit specified by `retry_limit`, which is set to 5.
5. Examples 11a and 11b are normal write and read bursts with four transfers from the primary master. The primary master sends an I/O write and read cycle to the primary slave in Example 12. The secondary master sends normal 32-bit write and read cycles to the primary slave in Example 15.
6. In Examples 17a and 17b, the secondary master sends 32-bit write bursts to the primary slave. The primary master writes to the primary slave's configuration command register (Example 19) before attempting to force parity errors (PAR, SERR#, and PERR#).
7. In examples 20 through 21, the primary master uses the command `pcimaster_configure(PCIMASTER_PCI_ERROR...)`; to create parity errors as it is performing writes and reads to the primary slave.
8. The `pcislave` command `configure(pci_error...)`; is demonstrated in Examples 22 to 26. The command can be used to force the parity errors PAR, SERR#, and PERR#.
9. In examples 27 through 29, the primary slave receives several reads and writes from the secondary master that have dual-address cycles.
10. Example 30 consists of normal 32-bit memory write and read cycles from the secondary master.

Secondary PCI Slave (U4) Sequence

The secondary slave starts with a short sequence of commands that demonstrate the `dump` and `print_msg` commands. The `dump` commands cause the secondary slave to write (`dump`) its configuration space contents to the file specified in the `dump` command. Note that the original call to `main` is commented out. You can call `main` several times with the HDL testbench section.

The commands in the command stream demonstrate `configure`, `print_msg`, and `dump` commands. The `dump` command writes its memory contents to the file `pcislave2_mem.dmp`.

The secondary slave participates in Examples 13, 14, and 16.

4

Using the PCI-X System Testbench

Introduction

The PCI-X system testbench (*pcixsys_tst.ext* or *pcixsys_c_tst.ext*) is a good starting point for using the PCI FlexModels in PCI-X mode. You can use the PCI-X system testbench to familiarize yourself with the operation of the models or as a template for building your own testbench. You can also use the system testbench to test the installation of your PCI FlexModels. In many cases, you can just replace one of the Synopsys PCI components with the design you want to test and get a quick start on your verification.

This chapter documents the PCI-X system testbench, which operates in PCI-X mode only. For information on using the conventional mode PCI system testbench, which operates in conventional PCI mode, see [Chapter 3 on page 73](#).

The PCI-X system testbench is an example of a typical testbench used for verifying PCI-X designs. The testbench uses two *pcimaster_fx* models, two *pcislave_fx* models, and one *pcimonitor_fx* model. You might not need to use all of the PCI FlexModels at the same time to test your design. For example, you might want to start with only your device and one PCI FlexModel in a testbench. The minimum configuration recommended for testing bus contention is two pcimasters, two pcislaves, and one pcimonitor, one of which can be your device.



Note

Data mismatches are expected within the testbench. The data mismatch errors coming from the testbench are intentional. The intent is to show usage scenarios through the use of such mismatches.

Using Testbench Command Examples

The PCI-X system testbench contains numerous command examples to help you get started using PCI FlexModels in PCI-X mode. These examples demonstrate how to generate common PCI-X transactions and serve as a practical example of how to use PCI FlexModel commands in PCI-X mode. The testbench examples also show you how to instantiate and configure the models, and provide you with a way to verify that models are working properly.

Setting up the Verilog, C, and VHDL Language PCI-X System Testbenches

To run the `pcixsys_tst.ext` or `pcixsys_c_tst.ext` system-level testbench, you must compile all package files from `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`. To make it easier, you can use the `flexm_setup` script to set up a work directory that contains all the package files. The following steps include VHDL and Verilog invocations of the `flexm_setup` script. For details on using the `flexm_setup` script, see the [FlexModel User's Manual](#). To use the Vera provided testbench, refer to “[Running the PCIX VERA Testbench with the VCS Simulator](#)” on page 115.

1. Run `flexm_setup`:

To copy all the files required to run the PCIX system testbench, run the following commands:

```
% mkdir workdir
% $LMC_HOME/bin/flexm_setup -dir workdir pcislave_fx
% $LMC_HOME/bin/flexm_setup -dir workdir pcimaster_fx
% $LMC_HOME/bin/flexm_setup -dir workdir pcimonitor_fx
```

You typically create `workdir` in the simulation directory.



Note

In this step, you have installed all the PCI models into a common directory called “workdir”, which is any name you choose. This was chosen as a convenience. You can also install the models into separate directories. In this case you must provided the proper pathname in the following steps and procedures.

2. Compile the C command files (C mode users only):

If you are using the C mode version of the system testbench, compile the following C command files:

```
workdir/examples/C/pcixmaster1_c_commands.c
workdir/examples/C/pcixslavel1_c_commands.c
workdir/examples/C/pcixslave2_c_commands.c
```

Name the object files as follows:

```
pcixmaster1_tst.ext
pcixslavel1_tst.ext
pcixslave2_tst.ext
```

where *ext* is **a** for Unix platforms and **exe** for Intel NT.



Note

If you are using Microsoft Window NT, you need to edit the `flex_run_program` command within the `pcixsys_c_tst.ext` file to specify the **.exe** extension instead of the **.a** extension for the compiled programs above.

You also need to link in the appropriate library for the platform you are using. The following example shows how to link in the -LBSD library for the HP700 platform for the three C command files. For more information, see [“Compiling an External C File”](#) in the *FlexModel User's Manual*.

```
% /bin/c89 -o pcixmaster1_tst.ext \
-I $LMC_HOME/sim/C/src \
-I workdir/src/C workdir/examples/C/pcixmaster1_c_commands.c \
workdir/src/C/hp700/pcimaster_pkg.o \
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \
-LBSD
```

```
% /bin/c89 -o pcixslave1_tst.ext \
-I $LMC_HOME/sim/C/src \
-I workdir/src/C workdir/examples/C/pcixslave1_c_commands.c \
workdir/src/C/hp700/pcislave_pkg.o \
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \
-lBSD

% /bin/c89 -o pcixslave2_tst.ext \
-I $LMC_HOME/sim/C/src \
-I workdir/src/C workdir/examples/C/pcixslave2_c_commands.c \
workdir/src/C/hp700/pcislave_pkg.o \
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \
-lBSD
```

3. Copy the data and configuration files from the work directory:

Copy the files listed below from the *workdir* directory to the current working directory. The following example shows the copy commands for VHDL. For Verilog use the *verilog* directory, which is *workdir/examples/verilog*. The testbench relies on these files for a simulation run.

```
% cp workdir/examples/vhdl/pcixslave1_tst.cfg .
% cp workdir/examples/vhdl/pcixslave1_tst.dat .
% cp workdir/examples/vhdl/pcixslave1_tst.io .
% cp workdir/examples/vhdl/pcixslave2_tst.cfg .
% cp workdir/examples/vhdl/pcixslave2_tst.dat .
% cp workdir/examples/vhdl/pcixslave2_tst.io .
```

4. Compile VHDL packages for simulation (VHDL only):

This step applies to VHDL users only. If you are using Verilog, see [Step 5](#).

You must make sure that all packages are compiled for simulation. The following is an example for running the MTI simulator on the Solaris platform. If you are running on an HP platform, the commands are slightly different.

**Note**

The following example is for setting up the HDL mode version of the system testbench. If you are setting up the C mode version of the testbench, you need to use the filename **pcixsys_c_tst.vhd** instead of **pcixsys_tst.vhd** in the last line of this example.

```
% cp $LMC_HOME/lib/sun4Solaris.lib/slm_mti.so .

% vlib mti_work
% vmap slm_lib ./mti_work
% vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
% vcom -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd

% vcom -work slm_lib workdir/src/vhdl/pcislave_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcislave_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_pkg.vhd

# See note below for using NT specific files at this step.
% vcom -work slm_lib workdir/examples/vhdl/pcislave_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimonitor_fx_mti.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcisys_fx_comp.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcimonitor.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcislave.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcixsys_tst.vhd
```

Note, the command:

```
vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
```

will work only on Unix. For PCNT use `slm_hdlc_nt.vhd`.

If you are using MTI on a Microsoft Windows NT platform, you must use the following specific files when using the `vcom` utility:

- `pcimaster_fx_mti_nt.vhd`
- `pcislave_fx_mti_nt.vhd`
- `pcimonitor_fx_mti_nt.vhd`

5. Compile Verilog packages for simulation (Verilog only):

This step applies to Verilog users only. If you are using VHDL, see [Step 4](#).

You must make sure that all packages are compiled for simulation. The following is an example for running the VCS simulator on the Solaris platform. If you are running on an HP platform, the commands are slightly different.



Note

The following example is for setting up the HDL mode version of the system testbench. If you are setting up the C mode version of the testbench, you need to use the filename **pcixsys_c_tst.v** instead of **pcixsys_tst.v** in the second line of this example.

The following example is for setting up the HDL mode version with “multiple instances” only. Refer to [“Multi-Instance Verilog Testbenches” on page 111](#) for additional information on this coding style.

```
% $VCS_HOME/bin/vcs \
workdir/examples/verilog/pcixsys_tst.v \
workdir/examples/verilog/pcimaster.v \
workdir/examples/verilog/pcimaster_fx_vcs.v \
workdir/examples/verilog/pcislave.v \
workdir/examples/verilog/pcislave_fx_vcs.v \
workdir/examples/verilog/pcimonitor.v \
workdir/examples/verilog/pcimonitor_fx_vcs.v \
${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \
-P ${LMC_HOME}/sim/pli/src/slm_pli.tab \
-lmc-swift \
+incdir+${LMC_HOME}/sim/pli/src +incdir+workdir/src/verilog \
+define+flex_multi_inst
```

If you want to set up without multiple instances, then you must do the following:

- Edit the file **pcixsys_tst.v** and remove multi-instance prefixes.

Following is an example using the multi-instance style.

```
u4.pcislave_configure(id_13, `PCISLAVE_PCIX, `TRUE, status);
u4.pcislave_configure(id_13, `PCISLAVE_DEV_ID, 64'h0000000000000000)
```

Following are the lines after removing the multi-instance prefixes:

```
pcislave_configure(id_13, `PCISLAVE_PCIX, `TRUE, status);
pcislave_configure(id_13, `PCISLAVE_DEV_ID, 64'h0000000000000000)
```

- Set up the testbench with the following commands. Note, only the last line from the previous example is deleted.

```
% $VCS_HOME/bin/vcs \
workdir/examples/verilog/pcixsys_tst.v \
workdir/examples/verilog/pcimaster.v \
workdir/examples/verilog/pcimaster_fx_vcs.v \
workdir/examples/verilog/pcislave.v \
workdir/examples/verilog/pcislave_fx_vcs.v \
workdir/examples/verilog/pcimonitor.v \
workdir/examples/verilog/pcimonitor_fx_vcs.v \
${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \
-P ${LMC_HOME}/sim/pli/src/slm_pli.tab \
-lmc-swift \
+incdir+${LMC_HOME}/sim/pli/src +incdir+workdir/src/verilog \
```

6. Simulate your design:

If you are using VHDL, simulate your design as follows:

```
% vsim -lib slm_lib cfgtest
```

If you are using Verilog, simulate your design as follows:

```
% simv
```

Multi-Instance Verilog Testbenches

Multi-instance style coding and compiling is only necessary when potentially might call the `pcimaster_read_rslt` or `pcislave_read_rslt` commands at the same simulation time from multiple `pcimaster_fx` or `pcislave_fx` instances in a Verilog testbench.

Some logic designers at times need to use the “`read_rslt`” command within their FlexModel while running a Verilog testbench. However, Verilog has a bug which causes incorrect results if multiple instances call the “`read_rslt`” commands at the same time. Thus, it is necessary to take steps to avoid this situation. For this reason the `pcixsys_tst.v` testbench is a good starting point since it was coded for multi-instance use. Note, even if you do not plan to use the `read_rslt` commands in the manner just described, you can still use the multi-instance style of coding and command scripting.

The “multi_inst” refers to a style of coding which allows you to prefix each call to a FlexModel command with the module instance. For example, if you have two pcislave_fx models that you have declared with the names “u3” and “u4”, then the following commands refer to those instances:

```
u3.pcislave_configure // refers the command to instance u3
u4.pcislave_configure //refers the command to instance u4
```

Using the multi-instance style of Verilog programming requires one additional flag in the VCS compilation line. That line is as follows:

```
+define+flex_multi_inst
```

Refer to the previous section [“Compile Verilog packages for simulation \(Verilog only\):” on page 110](#) to see an example of how to use this switch.

Remember, Verilog does not allow for the mixing of programming styles.

Running the VERA PCI-X System Testbenches

This section provides an example script which you can use to run the VERA system testbench for the VCS simulator. For detailed information on using VERA testbenches with FlexModels, consult the [Simulator Configuration Guide for Synopsys Models](#).



Note

The script in this section is meant as an example only, and must be edited to ensure proper execution.

VERA Testbench/Script Setup Tasks

The example script assumes the following:

- You will be using Solaris as the testbench platform. You may need to change the path to model object files as necessary. Following are the location for the object files for Solaris, HP-UX, and Linux which are critical for the testbench:
 - `${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/hp700.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/x86_linux.lib/slm_pli.o \`
- You have complete rights to the directory where you place the VERA testbench.

- You have a good LMC_HOME tree with the PCIX model set: pcimaster_fx, pcislave_fx, and pcimonitor_fx.
- Your DISPLAY environmental variable is correctly set.
- LD_LIBRARY_PATH is properly set. If not do a “ setenv LD_LIBRARY_PATH 1” on your command line.

Before you execute the example scripts, do the following:

- Create a directory to hold VERA testbench files. You can give this top level directory any name.
- Create a directory to hold all the compiled *.vro VERA files in your testbench directory. *This directory must be called file_vro.* For example:

```
mkdir vera_tb_folder
cd vera_tb_folder
mkdir file_vro
```

- Copy and paste the example script into a file and execute it from the directory you created to hold the testbench files.
- Change your location to your local testbench directory, and invoke the script from that location.



Note

Edit your script to remove any continuation backslashes before the flexm_setup utility. In some script environments, you cannot put a continuation character before the command. If flexm_setup has any problems with the backslash, you will get an error message stating the model name was not found.

The script has five parts. Each part is set off by comments in the script example. Edit each section as is appropriate for your environment if needed. The five sections are:

1. Setup environmental variables for the simulator, LMC_HOME tree, and VERA environment.
2. Setup testbench parameters. These are the *.io and *.cfg files used by the other testbenches.
3. Create the vera_user.o and the vera_local.dl files.
4. Create the *.vro files needed by the PCIX FlexModels.
5. Invoke the VCS simulator and run the testbench.

**Note**

The text for the script has been made smaller than normal to allow you to copy and paste it from the PDF file. As a result, reading it on a terminal will be difficult unless you magnify the text with Adobe's Acrobat Reader.

Running the PCIX VERA Testbench with the VCS Simulator

```
#!/bin/csh -fx

# This example script setups and runs the PCIX
# VERA testbench with the VCS simulator.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = ({VERA_HOME}/bin $path)
setenv SSI_LIB_FILES ./vera_local.dl

# Setup VCS
setenv VCS_HOME /d/vcs61/vcs6.1
set path = ({VCS_HOME}/bin $path)
setenv VCS_SWIFT_NOTES 1

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /synopsys/release
set path = ({LMC_HOME}/bin $path)
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " This part of the script just initialized the VERA_HOME, VCS_HOME, LMC_HOME."
echo " "
echo " Set: LD_LIBRARY_PATH, SNPSLMD_LICENSE_FILE, LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable."
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixslave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCIX_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
-I$VERA_HOME/lib \
  ${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
  ${LMC_LIB_DIR}/vera_slm_pli.o

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."

##### CREATE *.VRO FILES NEEDED BY THE PCIX FLEXMODELS #####

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_InitClass.vr \
./file_vro/PCIX_InitClass.vro || exit 1
```

```

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_BURSTClass.vr \
./file_vro/PCIX_BURSTClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DECODEandWAITClass.vr \
./file_vro/PCIX_DECODEandWAITClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_D064BITSClass.vr \
./file_vro/PCIX_D064BITSClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DWORDClass.vr \
./file_vro/PCIX_DWORDClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_GenericClass.vr \
./file_vro/PCIX_GenericClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_MASTERClass.vr \
./file_vro/PCIX_MASTERClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_PARITYClass.vr \
./file_vro/PCIX_PARITYClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_READresultClass.vr \
./file_vro/PCIX_READresultClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_RandomizeClass.vr \
./file_vro/PCIX_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_TARGETClass.vr \
./file_vro/PCIX_TARGETClass.vro || exit 1

```

```

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_SPLITClass.vr \
./file_vro/PCIX_SPLITClass.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcix_sys_tst.vr \
./file_vro/pcix_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/lstmodel.vr \
./file_vro/lstmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftdmodel.vr \
./file_vro/swiftdmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup \
pcimaster_fx`/src/vera/pcimaster_pkg.vr ./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vera/pcislave_pkg.vr ./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vera/pcimonitor_pkg.vr ./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE VCS AND RUN THE PCIX TESTBENCH #####

${VCS_HOME}/bin/vcs -lmc-swift -RI \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixsys_vera_tst_top.v \
./file_vro/pcix_sys_tst.vshell \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vcs.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vcs.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vcs.v \
${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \
-P ${LMC_HOME}/sim/pli/src/slm_pli.tab \
+incdir+${LMC_HOME}/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/verilog \
-l logfile\
-vera_dbind \
+vera_udf=./vera_local.dl +vera_mload=`$LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vera/pcix_sys_test_top.vrl

```

PCI-X System Testbench Architecture and Setup Values

[Figure 6](#) illustrates the HDL testbench configuration of the PCI-X system testbench. The basic configuration for the VERA, Verilog, and VHDL testbenches is the same. [Figure 7](#) illustrates the C testbench configuration of the PCI-X system testbench.

The Verilog, VHDL, and C testbench code is all in one main file. However, the Vera testbench source code is spread between the top module and files representing various VERA classes.



Note

The testbench deliberately generates parity errors to show certain related functionality. This is not an error in the testbench.

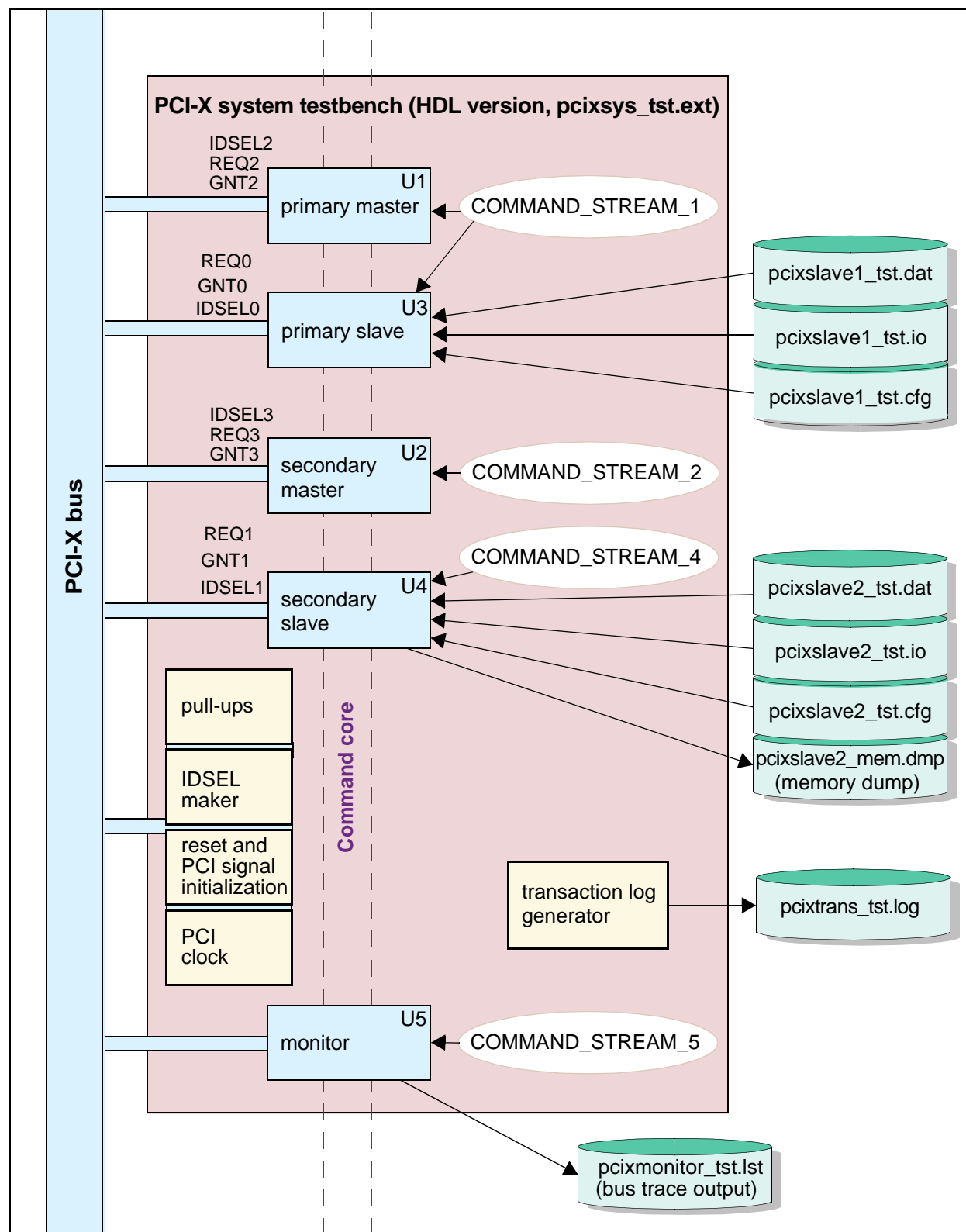


Figure 6: PCI-X System Testbench—HDL/VERA Version

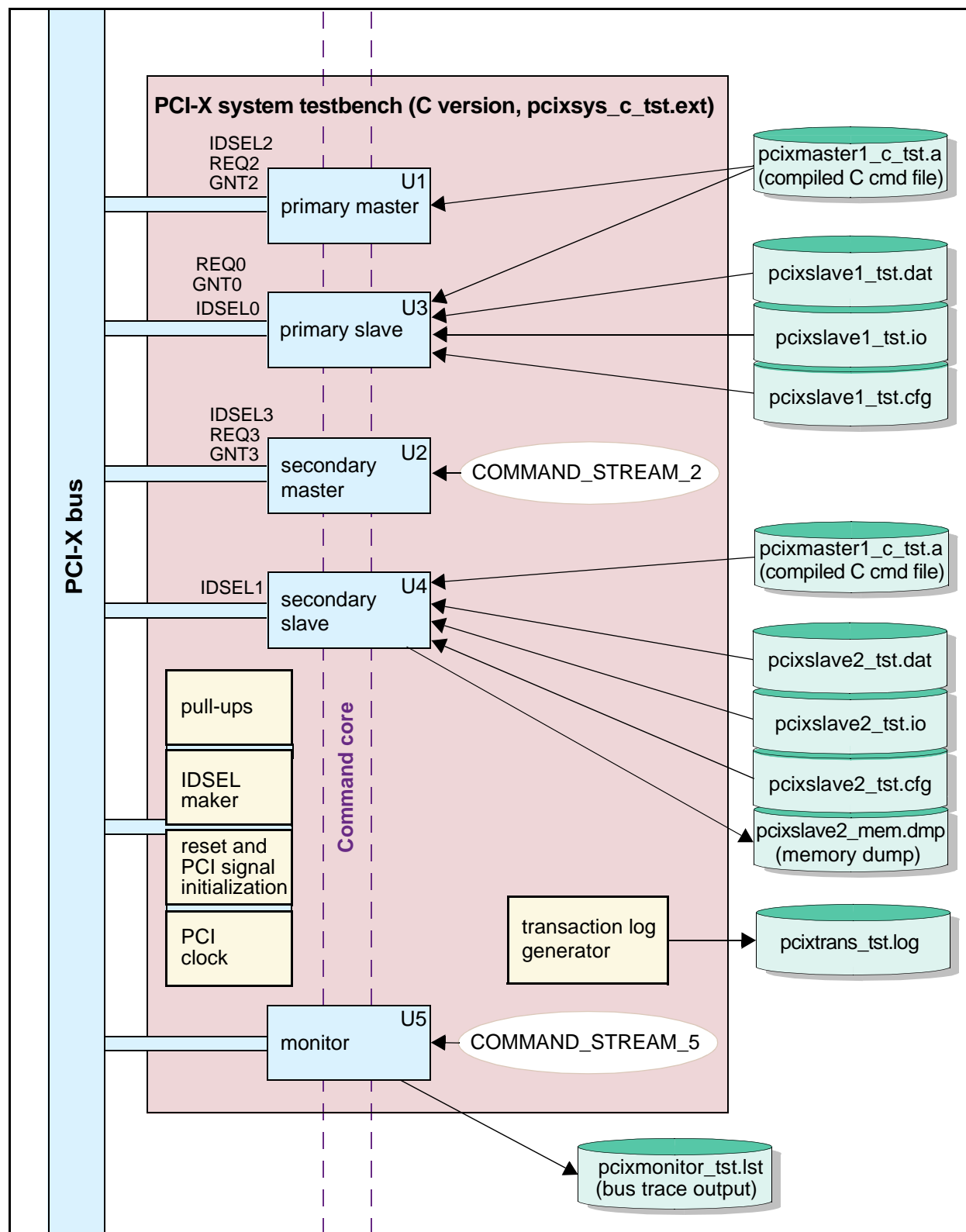


Figure 7: PCI-X System Testbench—C Version

Main Test Sequence Description

The main test sequence is inside the PCI-X system testbench (`pcixsys_tst.vr`, `pcixsys_tst.vhd`, or `pcixsys_tst.v`). Note, the VERA testbench has only one command sequence. Here is a description of what happens during the main test sequence:

- The top-level testbench performs a system reset.
- There are four command streams:
 - `COMMAND_STREAM_1` is for the primary master and primary slave
 - `COMMAND_STREAM_2` is for the secondary master
 - `COMMAND_STREAM_3` is for the secondary slave
 - `COMMAND_STREAM_4` is for the monitor
- Each command stream starts with the one clock delay required for the FlexModel command core to initialize.
- Each command stream calls the *model_configure* command for its model instance.

Basic Configuration Details for the PCI-X System Testbench

The following are configuration details for the PCI-X system testbench (`pcixsys_tst.ext` or `pcixsys_c_tst.ext`):

- The testbench includes instantiations of primary and secondary PCI masters, primary and secondary PCI slaves (or targets), and a PCI bus monitor.
- A 66-MHz clock drives all the models on the `pcixsys_tst.ext` or `pcixsys_c_tst.ext` testbench.

- The following model signals are connected to pull-ups:
 - PFRAMENN
 - PTRDYN
 - PIRDYNN
 - PSTOPNN
 - PREQ64NN
 - PACK64NN
 - PLOCKNN
 - PDEVSELNN
 - PPERRNN
 - PSERRNN
 - PPAR64
 - PCXBENN(7 DOWNT0 4)
 - Padata(63 DOWNT0 32)
- Reset is asserted low at 11 ns and deasserted at 41 ns. Reset remains deasserted for the rest of simulation. The signals PSDONE and PSBONN are driven high on the next rising PCLK edge after coming out of reset.
- The idsel_maker generates the PCI IDSEL signals for all PCI-X configuration cycles. The value of PADATA(31 down to 8) determines which IDSEL is asserted. The relationship between the model, PADATA, and IDSEL asserted is shown in [Table 17](#).

Table 17: IDSEL Assertion in the PCI-X System Testbench

PCI FlexModels	PADATA (31 down to 8)	IDSEL
pcixmaster (U1)	400000h	2
pcixmaster (U2)	C00000h	3
pcixslave (U3)	000000h	0
pcixslave (U4)	800000h	1

**Note**

IDSEL generation is system-dependent. The idsel_maker system is only an example.

Verilog, C, and VHDL PCI-X System Testbench Files

The PCI-X system testbench files for Verilog, VHDL, and C are listed below according to their location in \$LMC_HOME after installation.

- In \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/C:
 - pcixsys_c_tst.v — c control verilog testbench
 - pcixsys_c_tst.vhd — c control vhdl testbench
 - pcixmaster1_c_commands.c — master 1 c command file
 - pcixslave1_c_commands.c — slave 1 c command file
 - pcixslave2_c_commands.c — slave 2 c command file
 - pcixslave1_tst.cfg — slave 1 configure space memory file
 - pcixslave1_tst.dat — slave 1 memory space memory file
 - pcixslave1_tst.io — slave 1 io space memory file
 - pcixslave2_tst.cfg — slave 2 configure space memory file
 - pcixslave2_tst.dat — slave 2 memory space memory file
 - pcixslave2_tst.io — slave 2 io space memory file
- \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/verilog:
 - **pcixsys_tst.v**
 - **pcixslave1_tst.cfg** — c slave 1 configure space memory file
 - **pcixslave1_tst.dat** — c slave 1 memory space memory file
 - **pcixslave1_tst.io** — c slave 1 io space memory file
 - **pcixslave2_tst.cfg** — c slave 2 configure space memory file
 - **pcixslave2_tst.dat** — c slave 2 memory space memory file
 - **pcixslave2_tst.io** — c slave 2 io space memory file
- \$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/vhdl:
 - **pcixsys_tst.vhd**
 - **pcixslave1_tst.cfg** — c slave 1 configure space memory file
 - **pcixslave1_tst.dat** — c slave 1 memory space memory file
 - **pcixslave1_tst.io** — c slave 1 io space memory file
 - **pcixslave2_tst.cfg** — c slave 2 configure space memory file
 - **pcixslave2_tst.dat** — c slave 2 memory space memory file
 - **pcixslave2_tst.io** — c slave 2 io space memory file

VERA PCI-X System Testbench Files

The PCI-X system testbench files for Vera are listed below according to their location in \$LMC_HOME after installation.

- Important source files for the VERA testbench in:
\$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/vera:
 - pcixsys_tst.vr — top level of the Vera testbench.
 - pcixsys_vera_tst_top.v — calling and instantiation of models.
 - pcixsys_ver_tst_top.vhd — calling and instantiation of models.
 - class_dir — directory containing source files for the Vera classes called by the top level testbench:
 - PCIX_BURSTClass.vrh — handles read.write burst transactions.
 - PCIX_DECODEandWAITClass.vrh — tests various decode speed scenarios.
 - PCIX_DO64BITSClass.vrh — tests 64-bit interactions.
 - PCIX_DWORDClass.vrh — tests DWORD length transactions.
 - PCIX_GenericClass.vrh — generic example test class.
 - PCIX_InitClass.vrh — initializes all the instantiated models.
 - PCIX_MASTERClass.vrh — handles and test various master device termination events.
 - PCIX_PARITYClass.vrh — shows how to use the parity error features of both the master and slave models.
 - PCIX_READresultClass.vrh — shows the proper way to do read_rslt transactions using command and memory tags.
 - PCIX_RandomizeClass.vrh — class showing how to create random tests with Vera.
 - PCIX_SPLITClass.vrh — shows how to perform Split Transactions.
 - PCIX_TARGETClass.vrh — shows a target device reacting to Split Transactions and various termination types.
- Data and configuration files in
\$LMC_HOME/models/pcimonitor_fx/pcimonitor_fxversion/examples/vera
 - **pcixslave1_tst.cfg** — slave 1 configure space memory file
 - **pcixslave1_tst.dat** — slave 1 memory space memory file
 - **pcixslave1_tst.io** — slave 1 io space memory file
 - **pcixslave2_tst.cfg** — slave 2 configure space memory file
 - **pcixslave2_tst.dat** — slave 2 memory space memory file
 - **pcixslave2_tst.io** — slave 2 io space memory file

Primary PCI Master (U1) Configuration

The following are characteristics of the primary PCI master:

- The FlexModelId generic/defparameter is set to “PM”.
- In the HDL testbench, the primary master executes commands from `pcixsys_tst.ext` `COMMAND_STREAM_1`. In the C testbench, the primary master executes commands from `pcixmaster1_c_tst.a`.
- The primary master configures itself using `pcimaster_configure` commands with the *ctype* and *cvalue* settings shown in [Table 18](#).

Table 18: Primary Master `pcimaster_configure` Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_PCIX	FLEX_TRUE	Enables PCI-X mode.
PCIMASTER_START_TAG	5'b10000	Specifies the tag used by this pcimaster.
PCIMASTER_TL	32	Corresponds to the PCI configuration space latency timer, which defines the minimum number of PCI clocks for which a master can own the bus.
PCIMASTER_RL	5	Specifies the number of retries the model will attempt.

Secondary PCI Master (U2) Configuration

The following are characteristics of the secondary PCI master:

- The FlexModelId generic/defparameter is set to “SM”.
- In the HDL testbench, the secondary master executes commands from `pcixsys_tst.ext`, `COMMAND_STREAM_2`. In the C testbench, the secondary master executes commands from `pcixsys_c_tst.ext`, `COMMAND_STREAM_2`.
- The secondary master configures itself using `pcimaster_configure` commands with the *ctype* and *cvalue* settings shown in [Table 19](#).

Table 19: Secondary Master pcimaster_configure Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_PCIX	FLEX_TRUE	Enables PCI-X mode.
PCIMASTER_START_TAG	5'b20000	Specifies the tag used by this pcimaster.
PCIMASTER_TL	64	Corresponds to the PCI configuration space latency timer, which defines the minimum number of PCI clocks for which a master can own the bus.
PCIMASTER_RL	5	Specifies the number of retries the model will attempt.

Primary PCI Slave (U3) Configuration

The following are characteristics of the primary PCI slave:

- The FlexModelId generic/defparameter is set to “PT”.
- In the HDL testbench, the primary slave executes commands from pcixsys_tst.ext COMMAND_STREAM_3. In the C testbench, the primary slave executes commands from pcixmaster1_c_tst.a.
- Memory and I/O spaces are defined by setting the lower and upper address values of a memory or I/O address range. The slave does not respond to address values outside of these ranges. The ranges can be set by using parameters or by using pcislave_configure commands in external or HDL testbench mode. All slaves have three memory spaces and three I/O spaces. The three memory spaces are referred to as “space 0” through “space 2” in the following diagrams.

Primary slave memory space mapping is set to:

```
space 0 0000000000000000 hex <= addr <= 000000000FFFFFFF hex
space 1 1000000000000000 hex <= addr <= 10000000000000FF hex
space 2 2000000000000000 hex <= addr <= 20000000000000FF hex
```

Primary slave I/O space mapping is set to:

```
space 0 0000000000000200 hex <= addr <= 00000000000002FF hex
space 1 4000000000000000 hex <= addr <= 40000000000000FF hex
space 2 5000000000000000 hex <= addr <= 50000000000000FF hex
```

- The primary slave configures itself using `pcislave_configure` and `pcislave_load_file` commands. [Table 20](#) shows the *ctype* and *cvalue* settings used for the `pcislave_configure` command, and [Table 21](#) shows the *load_file* and *mem_space* settings used for the `pcislave_load_file` command.

Table 20: Primary Slave `pcislave_configure` Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_PCIX	FLEX_TRUE	Enables PCI-X mode.
PCISLAVE_ADDR_64	FLEX_TRUE or 1	Causes the model to respond to 64-bit address cycles.
PCISLAVE_DATA_64	FLEX_TRUE or 1	Causes the model to respond to 64-bit data cycles.

Table 21: Primary Slave `pcislave_load_file` Command Settings

<i>load_file</i> Parameter Values	<i>mem_space</i> Parameter Values	Description
pcixslave1_tst.cfg	PCISLAVE_CFG	Specifies the external file that initializes configuration space at time zero. The format is Verilog memory file format.
pcixslave1_tst.dat	PCISLAVE_MEM	Specifies the external file that initializes memory space at time zero. The format is Verilog memory file format.
pcixslave1_tst.io	PCISLAVE_IO	Specifies the external file that initializes I/O space at time zero. The format is Verilog memory file format.

Secondary PCI Slave (U4) Configuration

The following are characteristics of the secondary PCI slave:

- The FlexModelId generic/defparameter is set to “ST”.
- In the HDL testbench, the secondary slave executes commands from `pcixsys_tst.ext COMMAND_STREAM_4`. In the C testbench, the secondary slave executes commands from `pcixslave2_c_tst.a`.

- Memory and I/O spaces are defined by setting the lower and upper address values of a memory or I/O address range. The slave does not respond to address values outside of these ranges. The ranges can be set by using parameters or by using `pcislave_configure` commands in external or HDL testbench mode. All slaves have three memory spaces and three I/O spaces. The three memory spaces are referred to as “space 0” through “space 2” in the following diagrams.
- Secondary slave memory space mapping is set to:
 - space 0 00000000A0000000 hex <= addr <= 00000000A00000FF hex
 - space 1 00000000B0000000 hex <= addr <= 00000000B00000FF hex
 - space 2 00000000C0000000 hex <= addr <= 00000000C00000FF hex
- Secondary slave I/O space mapping is set to:
 - space 0 00000000D0000000 hex <= addr <= 00000000D00000FF hex
 - space 1 00000000E0000000 hex <= addr <= 00000000E00000FF hex
 - space 2 00000000F0000000 hex <= addr <= 00000000F00000FF hex
- The secondary slave configures itself using `pcislave_configure` and `pcislave_load_file` commands. [Table 22](#) shows the *ctype* and *cvalue* settings used for the `pcislave_configure` command, and [Table 23](#) shows the *load_file* and *mem_space* settings used for the `pcislave_load_file` command.

Table 22: Secondary Slave `pcislave_configure` Command Settings

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_PCIX	FLEX_TRUE	Enables PCI-X mode.
PCISLAVE_ADDR_64	FLEX_FALSE or 0	Causes the model to respond to 64-bit address cycles.
PCISLAVE_DATA_64	FLEX_FALSE or 0	Causes the model to respond to 64-bit data cycles.

Table 23: Secondary Slave pcislave_load_file Command Settings

<i>load_file</i> Parameter Values	<i>mem_space</i> Parameter Values	Description
pcixslave2_tst.cfg	PCISLAVE_CFG	Specifies the external file that initializes configuration space at time zero. The format is Verilog memory file format.
pcixslave2_tst.dat	PCISLAVE_MEM	Specifies the external file that initializes memory space at time zero. The format is Verilog memory file format.
pcixslave2_tst.io	PCISLAVE_IO	Specifies the external file that initializes I/O space at time zero. The format is Verilog memory file format.

PCI Monitor (U5) Configuration

The following are characteristics of the PCI monitor:

- The FlexModelId generic/defparameter is set to “5”.
- In the HDL testbench, the monitor executes commands from pcixsys_tst.ext COMMAND_STREAM_5. In the C testbench, the monitor executes commands from pcixsys_c_tst.ext COMMAND_STREAM_5.
- The monitor performs arbitration for the PCI-X bus (PCIMONITOR_ARBITRATE is set to true using the pcimonitor_configure command).
- The monitor uses arbitration scheme 0 (priority is set to 0 using the pcimonitor_configure command). Scheme 0 is a fixed scheme, with req(0) the highest priority.
- There is no bus parking when the PCI-X bus is idle (PCIMONITOR_PARKZERO is set to false using the pcimonitor_configure command). The monitor drives the lower 32 bits of Padata to 5a5a5a5a hex, the lower 4 bits of the command byte enable bus to 5 hex, and PAR to low. You can see an example of this when the idle(10); command is used in the primary master command stream (COMMAND_STREAM_1).
- Error checking is enabled (PCIMONITOR_ERRORCHECK is set to true using the pcimonitor_configure command).
- Bus tracing is enabled (PCIMONITOR_VALUETRACEFILE is set to true using the pcimonitor_configure command). This produces a PCI-X bus trace output file at the end of simulation. This output file is named pcimonitor_tst.lst. For information on

naming monitor output files so that they are compatible with the PCI-X system testbench and PCI test suite, see [“Using the Trace File with the PCI Test Suite” on page 61.](#))

- PCI-X mode is enabled using the following command:

```
pcimonitor_configure(PCIMONITOR_PCIX, FLEX_TRUE, status);
```

5

Using the pcimaster_fx

Introduction

The pcimaster_fx FlexModel emulates the protocol of a PCI or PCI-X bus master device at the pin and bus-cycle levels and performs timing violation checks. The pcimaster_fx issues read and write cycles to the pcislave_fx model and verifies the returning data and controls, enabling you to verify the functions of your PCI or PCI-X interface. The pcimaster_fx initiates PCI and PCI-X cycles using bus commands such as reads, writes, and idles. Each pcimaster_fx command performs the required cycles on the bus and handles the entire transaction, from requesting the bus to terminating the transaction.



Note

In conventional PCI mode, the pcimaster_fx is backward-compatible with previous versions of the model, with a few exceptions. For more information, see [Appendix B on page 367](#).

pcimaster_fx Supported Functions

The pcimaster_fx supports the following functions:

- PCI-X mode (see [“Using the pcimaster_fx in PCI-X Mode” on page 133](#))
- Split response and completion
- HDL, C, and Vera command modes (see [“Command Modes,”](#) in the *FlexModel User's Manual*)
- 64-bit extension
- 64-bit addressing
- User specified limit on number of retries the pcimaster_fx performs

- User specified limit on timeout clocks
- Address/data stepping
- Model status register (shows transaction termination status)
- Configuration registers
 - device id, vendor id
 - class code, revision ID
 - status—data parity detected (8), received target abort (12), received master abort (13), detected parity error (15)
 - command—parity error response (6)
- Error reporting (PERR#)
- Error generation (bad data, address parity)
- Exclusive accesses
- Master-initiated termination (completion, abort)
- Target-initiated termination (including disconnect, retry, abort, split response)
- Parity
- Read/write transactions (memory, I/O, configuration)

pcimaster_fx Modeling Exceptions and Unsupported Features

- The pcimaster_fx model differs from the PCI Local Bus Specification in that the following pins are not supported:
 - TDI
 - TDO
 - TCK
 - TMS
 - TRST#
 - INTA#
 - INTB#
 - INTL#

- INTD#
- The pcimaster_fx does not provide cache support, nor does it support JTAG. Cache support is handled by the pcislave_fx model. In addition, the pcimaster_fx model does not support interrupts.
- None of the PCI models support save and restore (restart) capabilities. No FlexModels support these features.
- The pcimaster_fx model may not perform correctly in all circumstances when a target device changes data width between the Split Response and the subsequent Split Completion.

Using the pcimaster_fx in PCI-X Mode

In PCI-X mode, the model is compatible with the PCI-X Addendum to the PCI Specification.

PCI-X mode is turned off by default. To turn PCI-X mode on, use the following command immediately after using the flex_get_inst_handle command:

```
pcimaster_configure(PCIMASTER_PCIX, FLEX_TRUE, status);
```

To turn PCI-X mode off, use the following command:

```
pcimaster_configure(PCIMASTER_PCIX, FLEX_FALSE, status);
```

In PCI-X mode, you need to set the TimingVersion generic/defparameter to “pcix”. This lets you use either 66MHz or 133MHz timing. With TimingVersion set to “pcix”, the model uses 66MHZ timing when the P66MHZ pin is set to 0, and 133MHZ timing when the P66MHZ pin is set to 1. The default is 0.



Note

In PCI-X mode, the P66MHZ pin can be considered as either a slow or fast pin. When set to true (1), the model uses a faster frequency (133MHz). When set to false (0), the model uses a slower frequency (66MHz). The name P66MHZ was retained for backward-compatibility, and does not mean that the model is operating at 66MHz.

100 Mhz Support

To run the pcimaster_fx at 100Mhz, set the timing version to "pcix" and the P66MHZ pin to high, which specifies a speed of 133Mhz. The model uses the same timing file and values at either speed (100Mhz and 133Mhz). The model is functionally the same at both speeds.

pcimaster_fx Command Summary

Table 24 lists the commands that can be issued by the pcimaster_fx model. To see a detailed description of the command, including syntax, usage description, parameters, and examples, click on the command name.

Table 24: pcimaster_fx Command Summary

Command Name	Description
pcimaster_configure	Modifies the model's operating conditions.
pcimaster_idle	Idles the model for the specified number of clock cycles.
pcimaster_output_enable	Enables or disables output for a specified bidirectional or output pin or bus.
pcimaster_pin_req	Requests the model to read the value of a specified pin or bus.
pcimaster_pin_rslt	Returns the results of a previous pcimaster_pin_req command.
pcimaster_read_continue	Provides burst read capabilities.
pcimaster_read_cycle	Executes a model read cycle on the bus.
pcimaster_read_intr_ack	Generates an interrupt acknowledge cycle.
pcimaster_read_rslt	Returns the results of a previous pcimaster_read_cycle, pcimaster_read_continue, or pcimaster_read_intr_ack command.
pcimaster_reg_req	Requests the model to read the value of a specified register.
pcimaster_reg_rslt	Returns the value of a specified register.
pcimaster_set_msg_level	Sets level of messaging used by model during simulation.
pcimaster_set_pin	Sets the specified pin or bus to the supplied value.
pcimaster_set_reg	Sets the specified register to the supplied value.
pcimaster_set_timing_control	Sets runtime values for individual or group timing checks.
pcimaster_write_continue	Provides burst write capabilities.
pcimaster_write_cycle	Executes a model write cycle on the bus.
pcimaster_write_special_cyc	Generates a PCI special cycle.

Table 24: pcimaster_fx Command Summary (cont.)

Command Name	Description
Global FLEX Commands (click on command name to jump to the command's description in the <i>FlexModel User's Manual</i>)	
flex_get_cmd_status	Checks for a valid command tag in the command queue.
flex_clear_queue	Clears command core queues for specified inst_handle.
flex_get_inst_handle	Provides a unique instance handle for a new instance.
flex_print_msg	Outputs the specified message to the screen.
flex_run_program	Switches the command source to a compiled C program.
flex_synchronize	Synchronizes the model instance with the number of instances specified.

pcimaster_fx Command Reference

This section contains command reference pages for all commands used by the pcimaster_fx FlexModel. The reference pages provide descriptions for each command along with syntax, parameters, prototypes, and examples. The commands are listed in alphabetical order.

For information about the command naming structure used with FlexModels, see the *FlexModel User's Manual*.

pcimaster_configure

Modifies the model's operating conditions.

Syntax

```
pcimaster_configure(inst_handle, ctype, cvalue, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>ctype</i>	A constant that determines which of the model's operating conditions are affected by executing this pcimaster_configure command. Table 25 shows the legal values.
<i>cvalue</i>	A 64-bit vector that specifies the new value of the operating condition specified by the <i>ctype</i> parameter. Table 25 shows the legal values. Note, X is not a valid value for any bit position.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .



Note

When using this command in a C control command or VHDL testbench, you must use a full 64-bit vector to specify the *cvalue*. Note, in your C Testbench, you may use the defined types FLEX_TRUE_C64 and FLEX_FALSE_C64 instead of FLEX_TRUE and FLEX_FALSE. Otherwise, you will probably receive the following warning message: “warning:improper pointer/integer combination.”

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_ADB_DISCONNECT_INCR (PCI-X mode only)	Integer. Default = 0	A positive value indicates the “increments” of ADBs to be disconnected after the first disconnected ADB using the PCIMASTER_DISCONNECT_ON_ADB. Example, if PCIMASTER_DISCONNECT_ON_ADB is 2 and if PCIMASTER_ADB_DISCONNECT_INCR is 3, then the subsequent ADB disconnect occurs at 5, 8, 11, 14, etc. This is for read/write transactions.
PCIMASTER_BACK	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE	Specifies whether to execute the next cycle as a back-to-back transaction. You must set this value to false before the last command of the transaction. For PCI-mode, this should be used with PCIMASTER_SAME for back to back transactions.
PCIMASTER_BUS_NUM (PCI-X mode only)	8-bit vector Default = 'b00000000	Specifies the bus number of the model.
PCIMASTER_C_LINE_SIZE	8-bit vector Default = 'b00000000	Specifies the cacheline size.
PCIMASTER_CAP_POINTER (PCI-X mode only)	8-bit vector from 8'h40 to 8'hff Default = 'b00000000	Sets the Capabilities Pointer in the PCI-X configuration space. This pointer points to the Capabilities List Item.
PCIMASTER_CDELAY	Positive integer Default = 0	Specifies how many cycles the model is to delay before requesting the command on the PCI bus.
PCIMASTER_CL	Positive integer Default = 0	Specifies the Max Clock Stop time prior to requesting clock from host.
PCIMASTER_CLS_CODE	24-bit vector Default = 'h000000	Specifies the class code.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_DES_MAX_CUM_READ_SIZE (PCI-X mode only)	3-bit vector Default = 'b000	Sets the Designated Maximum Cumulative Read Size portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles. In actual devices this value may be hardwired, but here it is programmable so users can read typical values in the system.
PCIMASTER_DES_MAX_READ_BC (PCI-X mode only)	2-bit vector Default = 'b11	Sets the Designated Maximum Memory Read Byte Count portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles. In actual devices this value may be hardwired, but here it is programmable so users can read typical values in the system.
PCIMASTER_DES_MAX_SPLIT_TRAN (PCI-X mode only)	3-bit vector Default = 'b000	Sets the Designated Maximum Outstanding Split Transaction portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles. In actual devices this value may be hardwired, but here it is programmable so users can read typical values in the system.
PCIMASTER_DEV_CNTRL_CFG	16-bit vector. Default = 0 Only bit 6 and 8 may be set.	Sets bits 6 and 8 in Device Control data structure: bit[6] is Parity Error Response, and bit[8] is SERR# Enable.
PCIMASTER_DEV_ID	16-bit vector Default = 'h0000	Specifies the device ID.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_DEV_NUM	5-bit vector Default = DEV_ID[4:0]	Specifies the Device Number.
PCIMASTER_DISCONNECT_ON_ADB (PCI-X mode only)	Positive integer or 0 (0 means no disconnect) Default = 0	Specifies the ADB at which the pcimaster_fx is to initiate a disconnect. If 1, then the master disconnects on the first ADB; if 2, then the master disconnects on the second ADB; if 3, then the third ADB, and so on.
PCIMASTER_DUAL_AD	32-bit address (specifies upper 32 bits for 64-bit addressing) Default = 'h00000000	Defines upper address value for 64-bit transfers. When set to a nonzero value, all subsequent memory cycles are executed as 64-bit cycles.
PCIMASTER_FUNC_NUM (PCI-X mode only)	3-bit vector Default = 'b000	Specifies the model's function number.
PCIMASTER_IO_BURST_ADDR	FLEX_TRUE or FLEX_FALSE Default = FLEX_TRUE	When set to FLEX_TRUE, increments address for each data phase. When set to FLEX_FALSE, does not increment address at all.
PCIMASTER_MAX_READ_BC (PCI-X mode only)	2-bit vector Default = 'b11	Sets the Maximum Memory Read Byte Count portion of the PCI-X command register. This setting does not change the way the model works, but is useful for performing configure cycles. In actual devices this value may be hardwired, but here it is programmable so users can read typical values in the system.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_MAX_SPLIT_TRAN (PCI-X mode only)	3-bit vector Default = 'b000	Sets the Maximum Outstanding Split Transaction portion of the PCI-X command register. This setting does not change the way the model works, but is useful for performing configure cycles. In actual devices this value may be hardwired, but here it is programmable so users can read typical values in the system.
PCIMASTER_MAX_CLK_P	0 to 1000 Default = 0	Specifies the maximum time in nanoseconds that the model is to wait before requesting the clock to start again (per the PCI mobile specification).
PCIMASTER_MODE (PCI mode only)	PCIMASTER_LINEAR(00) PCIMASTER_RESERVED(10) PCIMASTER_WRAP(01) PCIMASTER_RESERVED01(11) Default = 'b00 or PCIMASTER_LINEAR Mappings 00 drives 00 on A[1:0] 01 drives 10 on A[1:0] 10 drives 11 on A[1:0] 11 drives 01 on A[1:0]	Specifies the A1–A0 encodings for memory cycles. Example: PCIMASTER_WRAP drives the A[1:0] bus to A[11].
PCIMASTER_MULTIPLE_SPLITS	FLEX_TRUE = 1 FLEX_FALSE = 0 (default)	Allows the model to handle multiple outstanding split transactions. The maximum allowed number of outstanding split transactions is 32. You cannot increase or decrease this number.
PCIMASTER_NO_SNOOP (PCI-X mode only)	FLEX_TRUE = 1 (default) FLEX_FALSE = 0	Sets bit 30 of the AD bus during the attribute phase.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_PAR_ERR_DATA_PHASE	0 or positive integer	Sets the specific data phase on which you want to see a parity error generated. Note, PCIMASTER_PCI_ERROR must be set to 1 to activate this ctype. For example, if you want a bad PAR on data phase 5, set this value to five. Use this parameter to specify a particular data phase for the master to either assert PERR# during read/Split Completion, or generate bad PAR on a write to a slave. Must be used in conjunction with PCIMASTER_PCI_ERROR or PCIMASTER_SPLIT_ERROR.
PCIMASTER_PAR_PAR64_SELECT	2-bit vector. Default = 0 Refer to Table 5 on page 54 for values and their results.	Determines which bus half has bad parity. Must be used in conjunction with PCIMASTER_PCI_ERROR or PCIMASTER_SPLIT_ERROR.
PCIMASTER_PCI_ERROR	0 = no error 1 = parity error (data) 2 = parity error (address) 3 = parity error (attribute) 4 or greater = drives a bad PERR on reads on that data phase. Default = FLEX_FALSE or 0	Forces the model to execute an illegal PCI or PCI-X cycle. If the ctype is set to a value 4 or greater, then the model generates bad PERR on that particular data phase of the transaction.
PCIMASTER_PCIX	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE	Enables or disables PCI-X mode. FLEX_TRUE turns PCI-X mode on; FLEX_FALSE turns PCI-X mode off.
PCIMASTER_REV_ID	8-bit vector Default = 'h0000	Specifies the revision ID.
PCIMASTER_RL	Positive integer Default = 3	Specifies the retry limit.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_SAME (PCI mode only)	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE or 0	Specifies whether the request targets the same agent targeted in previous request. (Lets you take a shortcut in the bus interface state machine.) This should be used with PCIMASTER_BACK for back to back transactions.
PCIMASTER_SECONDARY_BUS_NUM (PCI-X mode only)	8-bit vector Default = 'b00000000	Specifies the secondary bus number of the model.
PCIMASTER_SPLIT_DECODE (PCIX mode only)	0 = Decode A (default) 1 = Decode B 2 = Decode C 3,4 = SUB decode >4 = Master Abort	Determines decode speed for the response of a split completion.
PCIMASTER_SPLIT_PCI_ERROR	0 = no error 1 = parity error (data) 2 = Address parity error (SERR#) 3 = Attribute parity error (SERR#) 4 or greater = drives a bad PERR on Split Completion. Default = FLEX_FALSE or 0	Forces the model to assert SERR# or PERR# during a Split Completion.
PCIMASTER_SPLIT_STOP_ASSERT_B4_ADB (PCI-X mode only)	Positive integer Default = 4	Determines how many cycles before the next ADB you can assert the STOP# pin for ADB disconnect.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_SPLIT_TRANSFER_LIMIT (PCI-X mode only)	0 = Retry termination 1 > = Normal transaction (default)	Specifies whether retry occurs during a split completion. If you want to specify more than one retry per split completion, you must use the PCIMASTER_SPLIT_RETRY_LIMIT parameter. Specify zero for PCIMASTER_SPLIT_TRANSFER_LIMIT, and then the number of retries with PCIMASTER_SPLIT_RETRY_LIMIT. A value of 1 (or greater) for PCIMASTER_SPLIT_TRANSFER_LIMIT disables PCIMASTER_SPLIT_RETRY_LIMIT.
PCIMASTER_SPLIT_TRDYNN_DELAY (PCI-X mode only)	Positive integer or 0 Default = 0	Specifies the initial TRDY# delay in clock cycles during a split completion.
PCIMASTER_SPLIT_ABORT_LIMIT (PCI-X mode only)	Positive integer Default = 1026 Ranges: 16 bit device: 0-16 32 bit device: 0-32	Specifies the maximum number of burst transfers (number of data phases) during a Split Completion before the master terminates the cycle with a target abort. Can have only two ranges depending on device size. Any values outside range are ignored by the model.
PCIMASTER_SPLIT_DISCONNECT_ON_ADB (PCI-X mode only)	Positive integer or 0 Default = 0	Specifies the ADB at which a transaction will be terminated. The default, 0, indicates no termination.

Table 25: *ctype* and *cvalue* Values in pcimaster_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCIMASTER_SPLIT_RETRY_LIMIT (PCI-X mode only)	Positive integer or 0. Default = 0.	Specifies the number of retries per split completion. (A zero value specifies that a retry will occur, but only on the first split completion, regardless if there are subsequent split completions.) To enable retries on a split completion, you <i>must</i> specify that PCIMASTER_SPLIT_TRANSFER_LIMIT equals zero (0).
PCIMASTER_START_TAG (PCI-X mode only)	5-bit vector from 5'b00000 to 5'b11111 Default = 'b00000	Specifies the starting tag number of the model.
PCIMASTER_STEP	Positive integer Default = 0	Specifies the number of address stepping cycles the model should use.
PCIMASTER_TL	Positive integer Default = 0	Specifies the time-out limit (latency timer).
PCIMASTER_TYPE1_ACCESS (PCI mode only)	FLEX_TRUE or FLEX_FALSE Default = FLEX_TRUE or 1	Specifies whether the pcimaster responds to type 1 configuration cycles.
PCIMASTER_VEN_ID	16-bit vector Default = 'h0000	Specifies the vendor ID.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_configure` command lets you modify the model's operating conditions. You can use the command to define the upper address value for 64-bit transfers, generate errors, specify command stepping and delay, and specify new device, vendor, and revision IDs. For a complete list of the operating conditions you can change, see [Table 25 on page 137](#).

Prototypes

C

```
void pcimaster_configure (
    const int      inst_handle,
    const int      ctype,
    const FLEX_VEC cvalue,
    int            *status);
```

VHDL

```
procedure pcimaster_configure (
    inst_handle      : in integer;
    ctype           : in integer;
    cvalue          : in bit_vector(63 downto 0);
    status          : out integer);
```

Verilog

```
task pcimaster_configure;
    input  [31:0] inst_handle;
    input  [31:0] ctype;
    input  [63:0] cvalue;
    output [31:0] status;
```

Vera

```
ModelObject.configure(
    integer      ctype,
    bit [63:0]   cvalue,
    var integer  status);
```

Examples

The following examples illustrate various forms of syntax for the pcimaster_configure command.

C

```
/* C syntax - set device id to 1234 */
pcimaster_configure(Id_11, PCIMASTER_DEV_ID, "h0000000000001234",
    &status);

/* set vendor id to 5678 */
pcimaster_configure(Id_11, PCIMASTER_VEN_ID, "h0000000000005678",
    &status);

/* set revision id to 2 */
pcimaster_configure(Id_11, PCIMASTER_REV_ID, "h0000000000000002",
    &status);

/* set class code to 1 */
pcimaster_configure(Id_11, PCIMASTER_CLS_CODE, "h0000000000000001",
    &status);

/* maximum time that the model is to wait before requesting */
/* the clock to start again is set to 120 ns */
pcimaster_configure(Id_11, PCIMASTER_MAX_CLK_P, "h0000000000000078",
    &status);

/* set cdelay to 0 */
pcimaster_configure(Id_11, PCIMASTER_CDELAY, "h0000000000000000",
    &status);

/* set address stepping cycles to 0 */
pcimaster_configure(Id_11, PCIMASTER_STEP, "h0000000000000000",
    &status);

/* set same target to false */
pcimaster_configure(Id_11, PCIMASTER_SAME, "h0000000000000000",
    &status);

/* set back-to-back to false */
pcimaster_configure(Id_11, PCIMASTER_BACK, "h0000000000000000",
    &status);

/* set address mode to linear */
pcimaster_configure(Id_11, PCIMASTER_MODE, "h0000000000000000",
    &status);
```

```

/* set address mode to wrap */
pcimaster_configure(Id_11, PCIMASTER_MODE, "h0000000000000001",
    &status);

/* set retry limit to 5, Master will skip cycle after 5th retry */
pcimaster_configure(Id_11, PCIMASTER_RL, "h0000000000000005", &status);

/* Latency Timer = 64 */
pcimaster_configure(Id_11, PCIMASTER_TL, "h0000000000000040", &status);

/* clock limit - 2000 time_unit */
pcimaster_configure(Id_11, PCIMASTER_CL, "h00000000000007d0", &status);

/* no PCI error generation */
pcimaster_configure(Id_11, PCIMASTER_PCI_ERROR, "h0000000000000000",
    &status);

/* set device number to 1f */
pcimaster_configure(Id_11, PCIMASTER_DEV_NUM, "h000000000000001f",
    &status);

```

VHDL

```

-- VHDL syntax - set device id to 1234
pcimaster_configure(Id_11, PCIMASTER_DEV_ID, X"0000000000001234",
    status);

-- set vendor id to 5678
pcimaster_configure(Id_11, PCIMASTER_VEN_ID, X"0000000000005678",
    status);

-- set revision id to 2
pcimaster_configure(Id_11, PCIMASTER_REV_ID, X"0000000000000002",
    status);

-- set class code to 1
pcimaster_configure(Id_11, PCIMASTER_CLS_CODE, X"0000000000000001",
    status);

-- maximum time that the model is to wait before requesting
-- the clock to start again is set to 120 ns
pcimaster_configure(Id_11, PCIMASTER_MAX_CLK_P, X"0000000000000078",
    status);

-- set cdelay to 0
pcimaster_configure(Id_11, PCIMASTER_CDELAY, X"0000000000000000",
    status);

```

```

-- set address stepping cycles to 0
pcimaster_configure(Id_11, PCIMASTER_STEP, X"0000000000000000", status);

-- set same target to false
pcimaster_configure(Id_11, PCIMASTER_SAME, X"0000000000000000", status);

-- set back-to-back to false
pcimaster_configure(Id_11, PCIMASTER_BACK, X"0000000000000000", status);

-- set address mode to linear
pcimaster_configure(Id_11, PCIMASTER_MODE, X"0000000000000000", status);

-- set address mode to wrap
pcimaster_configure(Id_11, PCIMASTER_MODE, X"0000000000000001", status);

-- set retry limit to 5, Master will skip cycle after 5th retry
pcimaster_configure(Id_11, PCIMASTER_RL, X"0000000000000005", status);

-- Latency Timer = 64
pcimaster_configure(Id_11, PCIMASTER_TL, X"0000000000000040", status);

-- clock limit - 2000 time_unit
pcimaster_configure(Id_11, PCIMASTER_CL, X"000000000000007d0", status);

-- no PCI error generation
pcimaster_configure(Id_11, PCIMASTER_PCI_ERROR, X"0000000000000000",
    status);

-- set device number to 1f
pcimaster_configure(Id_11, PCIMASTER_DEV_NUM, X"000000000000001f",
    status);

```

Verilog

```

// Verilog syntax - set device id to 1234
pcimaster_configure(Id_11, `PCIMASTER_DEV_ID, 16'h1234, status);

// set vendor id to 5678
pcimaster_configure(Id_11, `PCIMASTER_VEN_ID, 16'h5678, status);

// set revision id to 2
pcimaster_configure(Id_11, `PCIMASTER_REV_ID, 8'h02, status);

// set class code to 1
pcimaster_configure(Id_11, `PCIMASTER_CLS_CODE, 24'h000001, status);

// maximum time that the model is to wait before requesting
// the clock to start again is set to 120 ns

```

```
pcimaster_configure(Id_11, `PCIMASTER_MAX_CLK_P, 'd120, status);

// set cdelay to 0
pcimaster_configure(Id_11, `PCIMASTER_CDELAY, 'd0, status);

// set address stepping cycles to 0
pcimaster_configure(Id_11, `PCIMASTER_STEP, 'd0, status);

// set same target to false
pcimaster_configure(Id_11, `PCIMASTER_SAME, 'd0, status);

// set back-to-back to false
pcimaster_configure(Id_11, `PCIMASTER_BACK, 'd0, status);

// set address mode to linear
pcimaster_configure(Id_11, `PCIMASTER_MODE, 'b00, status);

// set address mode to wrap
pcimaster_configure(Id_11, `PCIMASTER_MODE, 'b01, status);

// set retry limit to 5, Master will skip cycle after 5th retry
pcimaster_configure(Id_11, `PCIMASTER_RL, 'd5, status);

// Latency Timer = 64
pcimaster_configure(Id_11, `PCIMASTER_TL, 'd64, status);

// clock limit - 2000 time_unit
pcimaster_configure(Id_11, `PCIMASTER_CL, 'd2000, status);

// no PCI error generation
pcimaster_configure(Id_11, `PCIMASTER_PCI_ERROR, 'd0, status);

// set device number to 1f
pcimaster_configure(Id_11, `PCIMASTER_DEV_NUM, 64'h0000000000000001f,
status);
```

VERA

```
//1st ADB is 3 data phases from start of cycle
pcimaster1.configure( `PCIMASTER_DISCONNECT_ON_ADB, 1, status);

// Disconnect burst at 2nd ADB
pcimaster1.configure( `PCIMASTER_DISCONNECT_ON_ADB, 2, status);
```

pcimaster_idle

Idles the model for a specified number of clock cycles.

Syntax

```
pcimaster_idle(inst_handle, cycles, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>cycles</i>	An integer (in the range of 0 to 2147483648) that specifies the number of clock cycles for which the model should idle.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates that the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag that you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The pcimaster_idle command enables you to insert clock cycles between pcimaster commands, effectively pausing (idling) the model for the number of clock cycles specified by the *cycles* parameter.

When the `pcimaster_idle` command is placed before a `read_cycle` or `write_cycle` command, the *cycles* parameter specifies the number of clock cycles for the model to idle after finishing the current command and driving REQ# low. Because `read_cycle` and `write_cycle` commands incur two idle clock cycles before executing, a `pcimaster_idle(id, 1, status)` or `pcimaster_idle(id, 2, status)` command placed before a `write_cycle` or `read_cycle` is redundant and is ignored by the model. A `pcimaster_idle(id, 3, status)` command placed before a `read_cycle` or `write_cycle` command inserts one extra clock delay.

Prototypes

C

```
void pcimaster_idle (
    const int inst_handle,
    const int cycles,
    int      wait_mode,
    int      *status);
```

VHDL

```
procedure pcimaster_idle (
    inst_handle: in integer;
    cycles      : in integer;
    wait_mode: in integer;
    status      : out integer);
```

Verilog

```
task pcimaster_idle;
    input  [31:0] inst_handle;
    input  [31:0] cycles;
    input          wait_mode;
    output [31:0] status;
```

Vera

```
ModelObject.idle(
    integer      cycles,
    integer      wait_mode,
    var integer  status);
```

Examples

The following examples idle the model for five clock cycles.

C

```
/* C syntax */  
pcimaster_idle(id, 5, FLEX_WAIT_F, &status);
```

VHDL

```
-- VHDL syntax  
pcimaster_idle(id, 5, FLEX_WAIT_F, status);
```

Verilog

```
// Verilog syntax  
pcimaster_idle(id, 5, `FLEX_WAIT_F, status);
```

Vera

```
//Idle for three clocks.  
pcimaster1.idle(3,`FLEX_WAIT_F,status);
```


pcimaster_output_enable

Enables or disables output for a specified bidirectional or output pin or bus.

Syntax

pcimaster_output_enable(*inst_handle*, *pin_name*, *enable_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	The name of the pin or bus for which output is to be enabled or disabled. Table 26 lists legal pin and bus names for this parameter.

Table 26: *pin_name* Values in pcimaster_output_enable Command

Device Pin Name	<i>pin_name</i> Parameter Value
ACK64#	PCIMASTER_PACK64NN_PIN
AD[31:0] (lower 32 bits)	PCIMASTER_PADATA_BUS
C/BE[3:0] (lower 4 bits)	PCIMASTER_PCXBENN_BUS
PAR	PCIMASTER_PPAR_PIN
FRAME#	PCIMASTER_PFRAMENN_PIN
TRDY#	PCIMASTER_PTRDYNN_PIN
IRDY#	PCIMASTER_PIRDYNN_PIN
SERR#	PCIMASTER_PSERRNN_PIN
STOP#	PCIMASTER_PSTOPNN_PIN
DEVSEL#	PCIMASTER_PDEVSELNN_PIN
REQ#	PCIMASTER_PREQNN_PIN
CLK	PCIMASTER_PCLKRUNNN_PIN
AD[63:32] (upper 32 bits)	PCIMASTER_PD_BUS
C/BE[7:4] (upper 4 bits)	PCIMASTER_PBENN_BUS
PAR64	PCIMASTER_PPAR64_PIN

Table 26: *pin_name* Values in pcimaster_output_enable Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
REQ64#	PCIMASTER_PREQ64NN_PIN
LOCK#	PCIMASTER_PLOCKNN_PIN
PERR#	PCIMASTER_PPERRNN_PIN

enable_value A value of 1 or FLEX_ENABLE enables the specified output pin and a value of 0 or FLEX_DISABLE disables the specified output pin. FLEX_ENABLE and FLEX_DISABLE are predefined constants.

status A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimaster_output_enable command lets you enable or disable the output for a specified pin. This command is used to specify pins that are not directly controlled by the pcimaster_fx model. Outputs are considered to be bidirectional. In normal operating mode, an output is enabled and could be either 1 or 0. If you use this command to disable a pin, it indicates a Z state and you can drive it from another source.

Prototypes

C

```
void pcimaster_output_enable (
    const int    inst_handle,
    const int    pin_name,
    const int    enable_value,
    int          *status);
```

VHDL

```

procedure pcimaster_output_enable
    inst_handle    : in integer;
    pin_name       : in integer;
    enable_value   : in integer;
    status         : out integer);

```

Verilog

```

task pcimaster_output_enable;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      [31:0] enable_value;
    output     [31:0] status;

```

Vera

```

ModelObject.output_enable(
    integer    pin_name,
    integer    enable_value,
    var integer status);

```

Examples

The following examples disable (set to high-Z state) the PREQ output pin.

C

```

/* C syntax */
pcimaster_output_enable(Inst_1, PCIMASTER_PREQ_PIN, FLEX_DISABLE,
    &status);

```

VHDL

```

-- VHDL syntax
pcimaster_output_enable(Inst_1, PCIMASTER_PREQ_PIN, FLEX_DISABLE,
    status);

```

Verilog

```

// Verilog syntax
pcimaster_output_enable(Inst_1, `PCIMASTER_PREQ_PIN, `FLEX_DISABLE,
    status);

```

Vera

```

// Vera syntax
pcimaster1.output_enable(`PCIMASTER_PREQ_PIN, `FLEX_DISABLE, status);

```

pcimaster_pin_req

Requests the model to retrieve the value of a specified pin or bus.

Syntax

```
pcimaster_pin_req(inst_handle, pin_name, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be retrieved. Table 27 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 27: *pin_name* Values in pcimaster_pin_req Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD(<i>n</i>) (lower 32 bits)	PCIMASTER_PADATAN_PIN
AD[31:0] (lower 32 bits)	PCIMASTER_PADATA_BUS
C/BE(<i>n</i>) (lower 4 bits)	PCIMASTER_PCXBENN _{<i>n</i>} _PIN
C/BE[3:0] (lower 4 bits)	PCIMASTER_PCXBENN_BUS
PAR	PCIMASTER_PPAR_PIN
FRAME#	PCIMASTER_PFRAMENN_PIN
TRDY#	PCIMASTER_PTRDYNN_PIN
IRDY#	PCIMASTER_PIRDYNN_PIN
STOP#	PCIMASTER_PSTOPNN_PIN
DEVSEL#	PCIMASTER_PDEVSELNN_PIN
IDSEL	PCIMASTER_PIDSEL_PIN
REQ#	PCIMASTER_PREQNN_PIN
GNT#	PCIMASTER_PGNTNN_PIN
CLK	PCIMASTER_PCLKRUNNN_PIN

Table 27: *pin_name* Values in pcimaster_pin_req Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
RST#	PCIMASTER_PRSTNN_PIN
AD(<i>n</i>) (upper 32 bits)	PCIMASTER_PD <i>n</i> _PIN
AD[63:32] (upper 32 bits)	PCIMASTER_PD_BUS
C/BE(<i>n</i>) (upper 4 bits)	PCIMASTER_PBENN <i>n</i> _PIN
C/BE[7:4] (upper 4 bits)	PCIMASTER_PBENN_BUS
PAR64	PCIMASTER_PPAR64_PIN
REQ64#	PCIMASTER_PREQ64NN_PI
ACK64#	PCIMASTER_PACK64NN_PIN
LOCK#	PCIMASTER_PLOCKNN_PIN
PERR#	PCIMASTER_PPERRNN_PIN
SERR#	PCIMASTER_PSERRNN_PIN
SBO#	PCIMASTER_PSBONN_PI
SDONE	PCIMASTER_PSDONE_PIN
Timing mode pin (1 or 0)	PCIMASTER_P66MHZ_PIN

wait_mode

One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the [FlexModel User's Manual](#) for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_pin_req` command issues a request to the model to retrieve the value of a specified pin or bus. This command is the first half of a result command pair; the result is returned when the model executes the `pcimaster_pin_rslt` command.

Prototypes

C

```
void pcimaster_pin_req (
    const int      inst_handle,
    const int      pin_name,
    const int      wait_mode,
    int            *status);
```

VHDL

```
procedure pcimaster_pin_req (
    inst_handle      : in integer;
    pin_name         : in integer;
    wait_mode        : in boolean;
    status           : out integer);
```

Verilog

```
task pcimaster_pin_req;
    input  [31:0] inst_handle;
    input  [31:0] pin_name;
    input          wait_mode;
    output [31:0] status;
```

Vera

```
ModelObject.pin_req(
    integer      pin_name,
    integer      wait_mode,
    var integer  status);
```

Examples

C

```
/* C syntax - requests 4-bits of TrcData bus.*/  
pcimaster_pin_req(Id_1, PCIMASTER_PADATA_BUS, FLEX_WAIT_F, &status);
```

VHDL

```
-- VHDL syntax - requests value of TrcEnd pin.  
pcimaster_pin_req(Id_1, PCIMASTER_PTRDYN_PIN, FLEX_WAIT_F, status);
```

Verilog

```
// Verilog syntax - requests 4-bits of TrcData bus.  
pcimaster_pin_req(Id_1, `PCIMASTER_PADATA_BUS, `FLEX_WAIT_F, status);
```

Vera

```
// Vera- requests 4-bits of TrcData bus.  
pcimaster1.pin_req(`PCIMASTER_PADATA_BUS, `FLEX_WAIT_F, status);
```

pcimaster_pin_rslt

Returns the value of a specified pin or bus.

Syntax

```
pcimaster_pin_rslt(inst_handle, pin_name, pin_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be returned. Table 28 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 28: *pin_name* Values in pcimaster_pin_rslt Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD(<i>n</i>) (lower 32 bits)	PCIMASTER_PADATAN_PIN
C/BE(<i>n</i>) (lower 4 bits)	PCIMASTER_PCXBENN_PIN
PAR	PCIMASTER_PPAR_PIN
FRAME#	PCIMASTER_PFRAMENN_PIN
TRDY#	PCIMASTER_PTRDYNN_PIN
IRDY#	PCIMASTER_PIRDYNN_PIN
STOP#	PCIMASTER_PSTOPNN_PIN
DEVSEL#	PCIMASTER_PDEVSELNN_PIN
IDSEL	PCIMASTER_PIDSEL_PIN
REQ#	PCIMASTER_PREQNN_PIN
GNT#	PCIMASTER_PGNTNN_PIN
CLK	PCIMASTER_PCLKRUNNN_PIN
RST#	PCIMASTER_PRSTNN_PIN
AD(<i>n</i>) (upper 32 bits)	PCIMASTER_PDn_PIN

Table 28: *pin_name* Values in pcimaster_pin_rslt Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
C/BE(<i>n</i>) (upper 4 bits)	PCIMASTER_PBENN _{<i>n</i>} _PIN
PAR64	PCIMASTER_PPAR64_PIN
REQ64#	PCIMASTER_PREQ64NN_PI
ACK64#	PCIMASTER_PACK64NN_PIN
LOCK#	PCIMASTER_PLOCKNN_PIN
PERR#	PCIMASTER_PPERRNN_PIN
SERR#	PCIMASTER_PSERRNN_PIN
SBO#	PCIMASTER_PSBONN_PI
SDONE	PCIMASTER_PSDONE_PIN
AD[31:0] (lower 32 bits)	PCIMASTER_PADATA_BUS
C/BE[3:0] (lower 4 bits)	PCIMASTER_PCXBENN_BUS
AD[31:0] (upper 32 bits)	PCIMASTER_PD_BUS
C/BE[3:0] (upper 4 bits)	PCIMASTER_PBENN_BUS
Timing mode pin (1 or 0)	PCIMASTER_P66MHZ_PIN

pin_value

The returned value of the pcimaster_pin_req command.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_pin_rslt` command returns the value of the pin or bus specified in the corresponding `pcimaster_pin_req` command. This command is the second half of the result command pair that begins with the `pcimaster_pin_req` command. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or previous commands complete execution. If all previous commands have completed and the specified result is not available, the command completes with an error.

Prototypes

C

```
void pcimaster_pin_rslt (
    const int      inst_handle,
    const int      pin_name,
    FLEX_VEC       pin_value,
    int            *status);
```

VHDL

```
procedure pcimaster_pin_rslt (
    inst_handle: in integer;
    pin_name   : in integer;
    pin_value  : out std_logic_vector(VR5432_MAX_BUS_WIDTH-1 downto 0);
    status     : out integer);
```

Verilog

```
task pcimaster_pin_rslt;
    input    [31:0] inst_handle;
    input    [31:0] pin_name;
    output   [`PCIMASTER_MAX_BUS_WIDTH:0] pin_value;
    output   [31:0] status;
```

Vera

```
ModelObject.pin_rslt(
    integer      pin_name,
    var bit [`PCIMASTER_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);
```

Examples

The following examples return the result of the previously issued `pcimaster_pin_req` command. The `pin_value(0)` is the returned value of the LSB of the pin result.

C

```
/* C syntax - returns 4 bits of TrcData. */  
pcimaster_pin_rslt(Id_1, PCIMASTER_PTRDYN_PIN, &pin_value, &status);
```

VHDL

```
-- VHDL syntax - returns the TrcEnd 1-bit value.  
pcimaster_pin_rslt(Id_1, PCIMASTER_PTRDYN_PIN, pin_value, status);
```

Verilog

```
// Verilog syntax - returns 4 bits of PADATA.  
pcimaster_pin_rslt(Id_1, `PCIMASTER_PADATA_BUS, pin_value, status);
```

Vera

```
// Vera syntax - returns 4 bits of PADATA.  
pcimaster1.pin_rslt(`PCIMASTER_PADATA_BUS, pin_value, status);
```

pcimaster_read_continue

Provides burst read capabilities.

Syntax

```
pcimaster_read_continue(inst_handle, byten, expected_data, delay, wait_mode,  
                        status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If you use PCIMASTER_MEM_READ_BLOCK or PCIMASTER_MEM_READ_BLOCK64 for the <i>rtype</i> value in the pcimaster_read_cycle command, the model ignores this parameter.
<i>expected_data</i>	The data the model expects to receive from this read operation. The model verifies this value against the actual results of the read operation; if the two do not match, the model displays a warning message. Legal values are any 64-bit std_logic_vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If <i>expected_data</i> contains an x, no data comparison occurs.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted. In PCI-X mode, the model ignores this parameter.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this

command and continues executing the command sequence. Refer to the *FlexModel User's Manual* for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the *FlexModel User's Manual*.

Description

The pcimaster_read_continue command provides burst read capabilities. When you perform data read functions that require multiple data phases, you must use one pcimaster_read_continue command for each data phase executed beyond the first phase. A pcimaster_read_cycle command must begin each read burst.

Always keep pcimaster_read_cycle and pcimaster_read_continues together. Do not intersperse them with other commands. The pcimaster_read_rslt command can only get valid data after the complete read cycle is finished. For example:

```
u2.pcimaster_read_cycle(id_11, `PCIMASTER_MEM_READ_BLOCK64,
32'h000000E0,32, 8'h00, 64'h0000000011111111, 2,`FLEX_FALSE, FLEX_FALSE,
    cmd_tag[0]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h2222222233333333,0,`FLEX_WAIT_F,
    cmd_tag[1]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h4444444455555555,0,`FLEX_WAIT_F,
    cmd_tag[2]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h6666666677777777,0,`FLEX_WAIT_T,
    cmd_tag[3]);
```

The pcimaster_fx and the pcislave_fx models have a queue size of 20k for addr_req, read_cycle, and read_continue commands. The models queue up these commands when they do not have corresponding read_rslt commands. The master/slave will discard the results of addr_req, read_cycle, and read_continue commands without corresponding read_rslt commands after the 20k limit is reached. The read_rslt queue supports random access by the user (see addr mode and tag mode parameter descriptions). The read_rslt queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

Prototypes

C

```
void pcimaster_read_continue (
    const int          inst_handle,
    const FLEX_VEC     byten,
    const FLEX_VEC     expected_data,
    const int          delay,
    int                wait_mode,
    int                *status);
```

VHDL

```
procedure pcimaster_read_continue (
    inst_handle      : in integer;
    byten            : in bit_vector(7 downto 0);
    expected_data    : in std_logic_vector(63 downto 0);
    delay            : in integer;
    wait_mode        : in integer;
    status           : out integer);
```

Verilog

```
task pcimaster_read_continue;
    input          [31:0] inst_handle;
    input          [7:0] byten;
    input          [63:0] expected_data;
    input          [31:0] delay;
    input          wait_mode;
    output         [31:0] status;
```

Vera

```
ModelObject.read_continue(
    bit  [7:0]      byten,
    bit  [63:0]    expected_data,
    integer        delay,
    integer        wait_mode,
    var integer    status);
```

Examples

C

```
/* C syntax */
pcimaster_read_continue(id, "h34", "h06060605050505", FLEX_FALSE,
    &status);
```

VHDL

```
-- VHDL syntax
pcimaster_read_continue(id, X "34", hex("06060605050505"), FLEX_FALSE,
    status);
```

Verilog

```
// Verilog syntax
pcimaster_read_continue(id, 8'h34, 64h06060605050505, `FLEX_FALSE,
    status);
```

Vera

```
// Vera syntax
pcimaster1.read_continue(8'h34, 64h06060605050505, `FLEX_FALSE, status);
```

pcimaster_read_cycle

Executes a read cycle on the bus.

Syntax

```
pcimaster_read_cycle(inst_handle, rtype, addr, tc, byten, expected_data, delay, lock,
    wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>rtype</i>	A constant that specifies the type of read operation to perform. Table 29 shows the legal values.

Table 29: *rtype* Values in pcimaster_read_cycle Command

Constant	Description
PCIMASTER_CONFIG_READ	Configuration read cycle. (In PCI-X mode, this is a DWORD transaction.)
PCIMASTER_IO_READ	I/O read cycle (In PCI-X mode, this is a DWORD transaction.)
PCIMASTER_MEM_READ (PCI Mode only)	Memory read cycle.
PCIMASTER_MEM_READ64 (PCI Mode only)	64-bit memory read cycle.
PCIMASTER_MEM_READ_LINE (PCI Mode only)	Memory read line cycle.
PCIMASTER_MEM_READ_LINE64 (PCI Mode only)	64-bit memory read line cycle.
PCIMASTER_MEM_READ_MUL (PCI Mode only)	Memory read multiple cycle.
PCIMASTER_MEM_READ_MUL64 (PCI Mode only)	64-bit memory read multiple cycle.
PCIMASTER_MEM_READ_DWORD (PCI-X mode only)	PCI-X memory read DWORD cycle. When you use this <i>rtype</i> , the model ignores the <i>tc</i> and <i>delay</i> parameters.

Table 29: *rtype* Values in pcimaster_read_cycle Command (*cont.*)

Constant	Description
PCIMASTER_MEM_READ_BLOCK (PCI-X mode only)	PCI-X memory read block cycle in 32-bit data mode. When you use this <i>rtype</i> , the model ignores the <i>byten</i> and <i>delay</i> parameters in both this command and the pcimaster_read_continue command.
PCIMASTER_MEM_RD_BLK_ALIAS (PCI-X mode only)	Alias to PCIMASTER_MEM_READ_BLOCK.
PCIMASTER_MEM_READ_BLOCK64 (PCI-X mode only)	PCI-X memory read block cycle in 64-bit data mode. When you use this <i>rtype</i> , the model ignores the <i>byten</i> and <i>delay</i> parameters in both this command and the pcimaster_read_continue command.
PCIMASTER_MEM_RD_BLK_ALIAS64 (PCI-X mode only)	Alias to PCIMASTER_MEM_READ_BLOCK64.
PCIMASTER_RSVD4	Reserved command code 4.
PCIMASTER_RSVD8 (PCI Mode only)	Reserved command code 8.

addr The address from which the command is to begin reading data. Legal values are any 32-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.

tc An integer (in the range 1 to 2147483648) that specifies the number of data transfers to occur on this command. When *tc* is set to 1, a single read occurs. When *tc* is greater than 1, this read_cycle command must be immediately followed by the number of read_continue commands needed to complete the burst transfers ($tc - 1$).

In PCI-X mode, the *tc* parameter represents the byte count, part of the attribute for the transfer sequence. The actual transfer count (the number of data phases) is calculated based on this byte count and on the starting address specified using the *addr* parameter. You must calculate the number of data phases required to transfer all the bytes and use the correct number of pcimaster_read_continue commands. If you do not

	use enough pcimaster_read_continue commands, model behavior is unpredictable. Extra pcimaster_read_continue commands are ignored.
<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If you use PCIMASTER_MEM_READ_BLOCK or PCIMASTER_MEM_READ_BLOCK64 as the <i>rtype</i> value, the model ignores this parameter.
<i>expected_data</i>	The data the model expects to receive from this read operation. The model verifies this value against the actual results of the read operation; if the two do not match, the model displays a warning message. Legal values are any 64-bit std_logic_vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If <i>expected_data</i> contains an x, no data comparison occurs.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted. In PCI-X mode, the model ignores this parameter.
<i>lock</i>	A boolean that specifies whether the operation should be locked. A value of FLEX_TRUE locks the request. In a locked sequence of operations, intermediate commands must continue to specify the lock value. The last locked command in the transaction must reset the <i>lock</i> value to FLEX_FALSE. The default is FLEX_FALSE.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_read_cycle` command executes a `pcimaster` read cycle. If the `pcimaster_fx` model does not own the bus when the `pcimaster_read_cycle` command executes, the command begins execution by arbitrating for the bus.

Always keep the `pcimaster_read_cycle` and `pcimaster_read_continue` commands together. Do not intersperse them with other commands. The `pcimaster_read_rslt` command can only read data after the complete read cycle is finished. For example:

```
u2.pcimaster_read_cycle(id_11, `PCIMASTER_MEM_READ_BLOCK64,
32'h000000E0,32, 8'h00, 64'h0000000011111111, 2,`FLEX_FALSE, FLEX_FALSE,
    cmd_tag[0]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h2222222233333333,0,`FLEX_WAIT_F,
    cmd_tag[1]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h4444444455555555,0,`FLEX_WAIT_F,
    cmd_tag[2]);
u2.pcimaster_read_continue(id_11, 8'h0,64'h6666666677777777,0,`FLEX_WAIT_T,
    cmd_tag[3]);
```

The `pcimaster_fx` and the `pcislave_fx` models have a queue size of 20k for `addr_req`, `read_cycle`, and `read_continue` commands. The models queue up these commands when they do not have corresponding `read_rslt` commands. The master/slave will discard the results of `addr_req`, `read_cycle`, and `read_continue` commands without corresponding `read_rslt` commands after the 20k limit is reached. The `read_rslt` queue supports random access by the user (see `addr` mode and `tag` mode parameter descriptions). The `read_rslt` queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

The PCI specification states that a device should only seek to establish a new locked condition (assert the LOCK pin) on a read transaction. Subsequent transactions can be write or read. The `pcimaster_fx` therefore can assert LOCK on a write or a read. The user must ensure that a read transaction is used to establish lock.

Prototypes

C

```
void pcimaster_read_cycle (
    const int          inst_handle,
    const int          rtype,
    const FLEX_VEC     addr,
    const int          tc,
    const FLEX_VEC     byten,
    const FLEX_VEC     expected_data,
    const int          delay,
    const int          lock,
    int                wait_mode,
    int                *status);
```

VHDL

```
procedure pcimaster_read_cycle (
    inst_handle      : in integer;
    rtype            : in integer;
    addr             : in bit_vector(31 downto 0);
    tc               : in integer;
    byten            : in bit_vector(7 downto 0);
    expected_data    : in std_logic_vector(63 downto 0);
    delay            : in integer;
    lock             : in boolean;
    wait_mode        : in integer;
    status           : out integer);
```

Verilog

```
task pcimaster_read_cycle;
    input      [31:0] inst_handle;
    input      [31:0] rtype;
    input      [31:0] addr;
    input      [31:0] tc;
    input      [7:0] byten;
    input      [63:0] expected_data;
    input      [31:0] delay;
    input      lock;
    input      wait_mode;
    output     [31:0] status;
```

Vera

```
ModelObject.read_cycle(
    integer      rtype,
    bit  [31:0]  addr,
    integer      tc,
    bit  [7:0]   byten,
    bit  [63:0]  expected_data,
    integer      delay,
    integer      lock,
    nteger       wait_mode,
    var integer  status);
```

Examples

C

```
/* C syntax */
pcimaster_read_cycle(Id_11, PCIMASTER_MEM_READ64, "h44556677", 1, "h12",
    "h0000000000000000", 3, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_MEM_READ, "h44556677", 1, "h02",
    "h0000000000000000", 0, FLEX_TRUE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_MEM_IO_READ, "h00006677", 1,
    "h02", "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_CONFIG_READ, "h00000004", 1,
    "h00", "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_READ_MUL, "h00000010", 1, "h00",
    "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_READ_LINE, "h00000000", 1, "h00",
    "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_read_cycle(Id_11, PCIMASTER_RSVD4, "h00000000", 1,
    "b00000000", "hxxxxxxxxxxxxxxxx", 0, FLEX_FALSE, FLEX_WAIT_F,
    &status);

pcimaster_read_cycle(Id_11, PCIMASTER_RSVD8, "h00000000", 1,
    "b00000000", "hxxxxxxxxxxxxxxxx", 0, FLEX_FALSE, FLEX_WAIT_F,
    &status);
```

VHDL

```
-- VHDL syntax
pcimaster_read_cycle(Id_11, PCIMASTER_MEM_READ64, X"44556677", 1, X"12",
    hex("0000000000000000"), 3, FLEX_FALSE, FLEX_WAIT_F, status);
```

```

pcimaster_read_cycle(Id_11, PCIMASTER_MEM_READ, X"44556677", 1, X"02",
    hex("0000000000000000"), 0, FLEX_TRUE, FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, PCIMASTER_MEM_IO_READ, X"00006677", 1,
    X"02", hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, PCIMASTER_CONFIG_READ, X"00000004", 1,
    X"00", hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, PCIMASTER_READ_MUL, X"00000010", 1, X"00",
    hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, PCIMASTER_READ_LINE, X"00000000", 1, X"00",
    hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, PCIMASTER_RSVD4, X"00000000", 1,
    B"00000000", hex("xxxxxxxxxxxxxxxx"), 0, FLEX_FALSE, FLEX_WAIT_F,
    status);

pcimaster_read_cycle(Id_11, PCIMASTER_RSVD8, X"00000000", 1,
    B"00000000", hex("xxxxxxxxxxxxxxxx"), 0, FLEX_FALSE, FLEX_WAIT_F,
    status);

```

Verilog

```

// Verilog syntax
pcimaster_read_cycle(Id_11, `PCIMASTER_MEM_READ64, 32'h44556677, 1,
    8'h12, 64'h0000000000000000, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_MEM_READ, 32'h44556677, 1,
    4'h2, 32'h00000000, 0, `FLEX_TRUE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_MEM_IO_READ, 32'h00006677, 1,
    4'h2, 32'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_CONFIG_READ, 32'h00000004, 1,
    4'h0, 32'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_READ_MUL, 32'h00000010, 1, 4'h0,
    'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_READ_LINE, 32'h00000000, 1, 4'h0,
    'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_RSVD4, 'h0000, 1, 'b0000,
    'hxxxxxxxx, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_read_cycle(Id_11, `PCIMASTER_RSVD8, 'h0000, 1, 'b0000,
    'hxxxxxxxx, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

```

Vera

```
// DWORD I/O Read
pcimaster1.read_cycle( `PCIMASTER_IO_READ, 32'h00000201, 1, 4'h1,
32'h232323xx, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD Configuration Read
pcimaster1.read_cycle( `PCIMASTER_CONFIG_READ, 32'h00000001, 1, 4'h2,
32'hx, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);

// DWORD Memory Read
pcimaster1.read_cycle( `PCIMASTER_MEM_READ, 32'h0F000001, 1, 4'h9,
32'hxxxxxxxx, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
```

pcimaster_read_intr_ack

Generates an interrupt acknowledge cycle.

Syntax

```
pcimaster_read_intr_ack(inst_handle, byten, expected_data, delay, wait_mode,
    status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>expected_data</i>	The data the model expects to receive from this read operation. The model verifies this value against the actual results of the read operation; if the two do not match, the model displays a warning message. Legal values are any 64-bit std_logic_vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If <i>expected_data</i> contains an x, no data comparison occurs.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted. Note: The model ignores this parameter in PCI-X mode.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_read_intr_ack` command generates an interrupt acknowledge cycle. If the `pcimaster_fx` model does not own the bus when the `pcimaster_read_intr_ack` command executes, the command begins execution by arbitrating for the bus.

Prototypes

C

```
void pcimaster_read_intr_ack (
    const int          inst_handle,
    const FLEX_VEC     byten,
    const FLEX_VEC     expected_data,
    const int          delay,
    int                wait_mode,
    int                *status);
```

VHDL

```
procedure pcimaster_read_intr_ack (
    inst_handle          : in integer;
    byten                : in bit_vector(7 downto 0);
    expected_data        : in std_logic_vector(63 downto 0);
    delay                : in integer;
    wait_mode            : in integer;
    status               : out integer);
```

Verilog

```
task pcimaster_read_intr_ack;
    input                [31:0] inst_handle;
    input                [7:0] byten;
    input                [63:0] expected_data;
    input                [31:0] delay;
    input                wait_mode;
    output               [31:0] status;
```

Vera

```

ModelObject.read_intr_ack(
    bit  [7:0]      byten,
    bit  [63:0]     expected_data,
    integer         delay,
    integer         wait_mode,
    var integer     status);

```

Examples**C**

```

/* C syntax */
pcimaster_read_intr_ack(id, "h03", "h000000000000", 0, FLEX_WAIT_T,
    &status);

```

VHDL

```

-- VHDL syntax
pcimaster_read_intr_ack(id, X "03", hex("000000000000"), FLEX_WAIT_T, 1,
    status);

```

Verilog

```

// Verilog syntax
pcimaster_read_intr_ack(id, `h3, `h0000xxxx, 1, `FLEX_WAIT_T, status);

```

Vera

```

// DWORD Interrupt acknowledge cycle.
pcimaster1.read_intr_ack( 4'h2, 32'h10101010, 3, `FLEX_WAIT_F, status);

```

pcimaster_read_rslt

Passes read data back to the testbench from a previous `pcimaster_read_cycle`, `pcimaster_read_continue`, or `pcimaster_read_intr_ack` command.

Syntax

```
pcimaster_read_rslt(inst_handle, addr, cmd_tag, return_data, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the <code>flex_get_inst_handle</code> command.
<i>addr</i>	The address from which the command is to begin reading data. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. For address mode to work, <i>cmd_tag</i> must be set to 0. For more information, see “Using Command Results” in the <i>FlexModel User's Manual</i> .
<i>cmd_tag</i>	The tag number of any previously called <code>pcimaster_read_cycle</code> , <code>pcimaster_read_continue</code> , or <code>pcimaster_read_intr_ack</code> command. For address mode to work, this parameter must be set to 0. Address mode is when this command retrieves data by using only the <i>addr</i> parameter. For more information, see the <i>status</i> parameter description and “Using Command Results” in the <i>FlexModel User's Manual</i> .
<i>return_data</i>	A variable name that will hold the returned value of the <code>pcimaster_read_cycle</code> , <code>pcimaster_read_continue</code> , or <code>pcimaster_read_intr_ack</code> command that has a corresponding <i>cmd_tag</i> or <i>addr</i> . (For address mode to work, the <i>cmd_tag</i> parameter must be set to 0.) Legal values are any 64-bit <code>std_logic_vector</code> (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue

and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_read_rslt` command retrieves data from the read cycles generated by the `pcimaster_read_cycle`, `pcimaster_read_continue`, and `pcimaster_read_intr_ack` commands. This command is the second half of a command result pair. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or until previous commands complete execution.

Always keep `pcimaster_read_cycle` and `pcimaster_read_continues` together. Do not intersperse them with other commands. The `pcimaster_read_rslt` command can only get valid data after the complete read cycle is finished. The following example from the system testbench, `pcisys_tst.v`, illustrates this point using `pcimaster_read_rslt` with the command tag method:

```
u2.pcimaster_read_cycle(id_11, `PCIMASTER_MEM_READ_BLOCK64, 32'h000000E0,
    32, 8'h00, 64'h0000000011111111, 2, `FLEX_FALSE, FLEX_FALSE,
    cmd_tag[0]);
u2.pcimaster_read_continue(id_11, 8'h0, 64'h2222222233333333, 0, `FLEX_WAIT_F,
    cmd_tag[1]);
u2.pcimaster_read_continue(id_11, 8'h0, 64'h4444444455555555, 0, `FLEX_WAIT_F,
    cmd_tag[2]);
u2.pcimaster_read_continue(id_11, 8'h0, 64'h6666666677777777, 0, `FLEX_WAIT_T,
    cmd_tag[3]);
/* FLEX_WAIT_T in final read_continue ensures that the read transaction
   completes before testbench calls read_rslt. */

//=====Method 1(cmd_tag mode)=====

for (i=0; i<4; i=i+1)
    begin
        u2.pcimaster_read_rslt(id_11, 64'h0000000000000000, cmd_tag[i],
            data, status);
        $display("cmd_tag value for read_rslt = %0h, data = %h",
            cmd_tag[i], data);
    end
```

If all previous commands have completed and the specified result is not available, the command completes with an error (a negative or 0 returned *status* value). If the `read_cycle` command did not complete normally (for example, after a target or master abort), the `read_result` command will return “XXXXXXX”.

During split transaction reads, the pcimaster_fx always waits for the current split transaction to complete before proceeding to the next transaction. Because of this, the read_rslt command returns the data of a previously issued read. To identify the previously issued read, you must either pass the address of the read as a parameter or reference the read using a *status* value returned by the appropriate read command.

The pcimaster_fx and the pcislave_fx models have a queue size of 20k for addr_req, read_cycle, and read_continue commands. The models queue up these commands when they do not have corresponding read_rslt commands. The master/slave will discard the results of addr_req, read_cycle, and read_continue commands without corresponding read_rslt commands after the 20k limit is reached. The read_rslt queue supports random access by the user (see addr mode and tag mode parameter descriptions). The read_rslt queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

Prototypes

C

```
void pcimaster_read_rslt(
    const int          inst_handle,
    const FLEX_VEC     addr,
    const int          cmd_tag,
    FLEX_VEC           return_data,
    int                *status);
```

VHDL

```
procedure pcimaster_read_rslt(
    inst_handle      : in integer;
    addr             : in bit_vector (63 downto 0);
    cmd_tag          : in integer;
    return_data      : in std_logic_vector (63 downto 0);
    status           : out integer);
```

Verilog

```
task pcimaster_read_rslt;
    input          [31:0] inst_handle;
    input          [63:0] addr;
    input          [31:0] cmd_tag;
    inout          [63:0] return_data;
    output         [31:0] status;
```

Vera

```

ModelObject.read_rslt(
    bit  [63:0]    address,
    integer        cmd_tag,
    var bit  [63:0] return_data,
    var integer    status);

```

Examples**C**

```

/* C syntax , address mode */
pcimaster_read_rslt(id_10, "h2748ab3130341008", 0, data, &status);

/* C syntax , tag mode */
pcimaster_read_rslt(id_10, "h0000000000000000", 78, data, &status);

```

VHDL

```

-- VHDL syntax, address mode
pcimaster_read_rslt(id_10, X"2748ab3130341008", 0, data, status);

-- VHDL syntax, tag mode
pcimaster_read_rslt(id_10, X"0000000000000000", 78, data, status);

```

Verilog

```

// Verilog syntax, address mode
pcimaster_read_rslt(id_10, 64'h2748ab3130341008, 0, data, status);

// Verilog syntax, tag mode
pcimaster_read_rslt(id_10, 'h0000000000000000, 78, data, status);

```

Vera

```

// Verilog syntax, address mode
pcimaster1.read_rslt(64'h2748ab3130341008, 0, data, status);

// Verilog syntax, tag mode
pcimaster1.read_rslt('h0000000000000000, 78, data, status);

```

pcimaster_reg_req

Requests the model to retrieve data from the its internal status register called PCIMASTER_STATUS_REG.

Syntax

pcimaster_reg_req (*inst_handle*, *reg_name*, *wait_mode*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register whose data is to be retrieved. The pcimaster_fx has only one register; therefore, the only possible value for this parameter is PCIMASTER_STATUS_REG.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see "The status Parameter" in the <i>FlexModel User's Manual</i> .

Description

The pcimaster_reg_req command issues a request to the model to retrieve data from its internal register, the PCIMASTER_STATUS_REG register. The PCIMASTER_STATUS_REG register records the termination status of the last transaction on the PCI or PCI-X bus. This command is the first half of a result command pair; the register value is returned when the model executes a pcimaster_reg_rslt command. Possible register values are defined in [Table 30 on page 184](#).

Table 30: Register Value Definitions for PCIMASTER_STATUS_REG

Register Value	Definition
0	Completion
1	Master abort
2	Target abort
3	disconnect_a
4	disconnect_b
5	Retry and disconnect_c
6	Retry limit reached and command discarded
7	Disconnect on ADB from master
8	Disconnect on ADB from slave
9	Single data phase disconnect
10	Split response
11	Split completion discarded
12	Split completion message signal out of range

Prototypes

C

```
void pcimaster_reg_req (
    const int      inst_handle,
    const int      reg_name,
    const int      wait_mode,
    int            *status);
```

VHDL

```
procedure pcimaster_reg_req (
    inst_handle      : in integer,
    reg_name         : in integer,
    wait_mode        : in boolean,
    status           : out integer);
```


Verilog

```
task pcimaster_reg_req;
    input    [31:0] inst_handle;
    input    [31:0] reg_name;
    input          wait_mode;
    output    [31:0] status;
```

Vera

```
ModelObject.reg_req (
    integer      reg_name,
    integer      wait_mode,
    var integer   status);
```

Examples

The following command requests the value of the PCIMASTER_STATUS_REG register:

```
// Verilog Example
pcimaster_reg_req(inst, `PCIMASTER_STATUS_REG, `FLEX_WAIT_F, status);
```

The result is returned using the pcimaster_reg_rslt command:

```
--VHDL Example: The first command below requests the value of the
--PCIMASTER_STATUS_REG register; the second command returns the value.
```

```
pcimaster_reg_req (Id_1, PCIMASTER_STATUS_REG, FLEX_WAIT_F, status);
pcimaster_reg_rslt(Id_1, PCIMASTER_STATUS_REG, Return_value, status);
assert (false) report "STATUS register " severity NOTE;
```

```
/* C Example */
```

```
/* Displays the value of the PCIMASTER_STATUS_REG register*/
pcimaster_reg_req (Id_1, PCIMASTER_STATUS_REG, FLEX_WAIT_F, &status);
pcimaster_reg_rslt(Id_1, PCIMASTER_STATUS_REG, Return_value, &status);
flex_fprintf(stderr, " STATUS register = %H\n", Return_value);
```

```
//Vera example.
```

```
pcimaster1.reg_req (`PCIMASTER_STATUS_REG, `FLEX_WAIT_F, status);
```

pcimaster_reg_rslt

Returns the data of the PCIMASTER_STATUS_REG register.

Syntax

```
pcimaster_reg_rslt (inst_handle, reg_name, reg_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register whose data is to be returned. The pcimaster_fx has only one register; therefore the only possible value for this parameter is PCIMASTER_STATUS_REG.
<i>reg_value</i>	A variable name that will hold the returned value of the PCIMASTER_STATUS_REG register, as requested in the corresponding pcimaster_reg_req command. This returned value is a 4-bit vector that represents the termination status of the last transaction on the PCI or PCI-X bus to be issued before the pcimaster_reg_req command. Table 31 lists the possible register values and their definitions.

Table 31: Register Value Definitions for PCIMASTER_STATUS_REG

Register Value	Definition
0	Completion
1	Master abort
2	Target abort
3	disconnect_a
4	disconnect_b
5	Retry and disconnect_c
6	Retry limit reached and command discarded
7	Disconnect on ADB from master
8	Disconnect on ADB from slave

Table 31: Register Value Definitions for PCIMASTER_STATUS_REG (cont.)

Register Value	Definition
9	Single data phase disconnect
10	Split response
11	Split completion discarded
12	Split completion message signal out of range

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_reg_rslt` command returns the value of the `PCIMASTER_STATUS_REG` register as requested in the corresponding `pcimaster_reg_req` command. The returned value is a 4-bit vector that represents the termination status of the last transaction on the PCI or PCI-X bus to be issued before the `pcimaster_reg_req` command. [Table 31 on page 186](#) lists the possible register values and their definitions.

This command is the second half of the result command pair that begins with the `pcimaster_reg_req` command. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or previous commands complete execution. If all previous commands have completed and the specified result is not available, the command completes with an error.

Prototypes

C

```
void pcimaster_reg_rslt (
    const int      inst_handle,
    const int      reg_name,
    FLEX_VEC      reg_value,
    int            *status);
```

VHDL

```

procedure pcimaster_reg_rslt (
    inst_handle: in integer,
    reg_name   : in integer,
    reg_value  : out bit_vector(PCIMASTER_MAX_REG_WIDTH-1 downto 0),
    status     : out integer);

```

Verilog

```

task pcimaster_reg_rslt;
    input    [31:0] inst_handle;
    input    [31:0] reg_name;
    output   [`PCIMASTER_MAX_REG_WIDTH:0] reg_value;
    output   [31:0] status;

```

Vera

```

ModelObject.reg_rslt (
    integer      reg_name,
    var bit      [`PCIMASTER_MAX_REG_WIDTH:0] reg_value,
    var integer  status);

```

Examples

The `pcimaster_reg_req` commands in the examples below request the value of the `PCIMASTER_STATUS_REG` register. The `pcimaster_reg_rslt` commands return the result, assigning the value of the `PCIMASTER_STATUS_REG` register to the *reg_value* variable:

```

// Verilog Example
pcimaster_reg_req(inst, `PCIMASTER_STATUS_REG, `FLEX_WAIT_F, status);
pcimaster_reg_rslt(inst, `PCIMASTER_STATUS_REG, reg_value, status);

```

```

--VHDL Example
pcimaster_reg_req (Id_1, PCIMASTER_STATUS_REG, FLEX_WAIT_F, status);
pcimaster_reg_rslt(Id_1, PCIMASTER_STATUS_REG, Return_value, status);
assert (false) report "PCIMASTER_STATUS_REG register " severity NOTE;

```

```

/* C Example */
/* Display the value of the PCIMASTER_STATUS_REG register*/
pcimaster_reg_req (Id_1, PCIMASTER_STATUS_REG, FLEX_WAIT_F, &status);
pcimaster_reg_rslt(Id_1, PCIMASTER_STATUS_REG, Return_value, &status);
flex_fprintf(stderr, " PCIMASTER_STATUS_REG register = %H\n",
Return_value);

```

```

//Vera syntax.
//pcimaster1.reg_rslt(`PCIMASTER_STATUS_REG, Return_value, status);

```

pcimaster_set_msg_level

Sets the level of messaging used by model during simulation.

Syntax

```
pcimaster_set_msg_level(inst_handle, mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>mode</i>	A 32-bit integer that controls the level of messaging or a predefined constant that controls a single message type. Each bit in the <i>mode</i> field controls one message category; if a specific bit is set to 1, the messages for that category are enabled. Table 32 lists the individual bit assignments and the corresponding predefined constants.



Note

To get command-by-command trace information and other model messages, which can be useful for debugging, set all the upper bits in the *mode* parameter to 1, as follows:

```
pcimaster_set_msg_level (inst_handle, 32'h0ffffffff, status);
```

Table 32: *mode* Values in pcimaster_set_msg_level Command

Bit Assignment	Predefined Constant	Description
None	FATAL	Stops simulation immediately after a fatal message is reported. Fatal messages are always enabled.
bit-30	PCIMASTER_DEBUG	Displays debug messages, including command-by-command trace information.
0FFFFFFF	FLEX_ALL_MSGS	Turns on all message types except debug messages. For command-by-command trace information, use PCIMASTER_DEBUG.
00000000	FLEX_NO_MSGS	Turns off all message types other than fatal.

Table 32: *mode* Values in pcimaster_set_msg_level Command (cont.)

Bit Assignment	Predefined Constant	Description
bit-0	FLEX_ERROR	Displays error messages for situations in which the model can recover and resume simulation, such as when the model receives a command that would put it into an invalid state.
bit-1	FLEX_WARNING	Displays warning messages for situations that are not necessarily errors, such as when significant bits of an address are ignored.
bit-2	FLEX_TIMING	Displays timing messages.
bit-3	FLEX_XHANDLING	Displays X-handling messages.
bit-4	FLEX_INFO	Informs you of the status or behavior of the model.
bits 8–29	Undefined	Undefined

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimaster_set_msg_level command controls the severity level of the messages displayed by the model during simulation. Model messages are grouped into categories. Except for fatal messages, which are always enabled, you can enable or disable any category for each model instance. This command allows you to change from the default message level for this model.

Prototypes

C

```
void pcimaster_set_msg_level (
    const int      inst_handle,
    const int      mode,
    int            *status);
```

VHDL

```
procedure pcimaster_set_msg_level (  
    inst_handle      : in integer;  
    level            : in integer;  
    status           : out integer);
```

Verilog

```
task pcimaster_set_msg_level;  
    input    [31:0] inst_handle;  
    input    [31:0] level;  
    output   [31:0] status;
```

Vera

```
ModelObject.set_msg_level(  
    integer      mode,  
    var integer  status);
```

Examples

C

```
/* C syntax */  
pcimaster_set_msg_level(Id_1, FLEXINFO, &status);
```

VHDL

```
-- VHDL syntax - turns off all messages  
pcimaster_set_msg_level(Id_1, FLEX_NO_MSGS, status);  
  
-- Enables INFO & WARNING messages  
pcimaster_set_msg_level(Id_1, FLEX_INFO + FLEX_WARNING, status);
```

Verilog

```
// Verilog syntax - turns off all messages  
pcimaster_set_msg_level(Id_1, `FLEX_NO_MSGS, status);  
  
// Enables INFO & WARNING messages  
pcimaster_set_msg_level(Id_1, `FLEX_INFO + `FLEX_WARNING, status);  
  
// Enables all messages  
pcimaster_set_msg_level(Id_1, 32'h0fffffff, status);
```

Vera

```
//Enables all messages.  
pcimaster1.set_msg_level(`FLEX_ALL_MSGS, status);
```


pcimaster_set_pin

Sets a specified pin or bus to the supplied value.

Syntax

```
pcimaster_set_pin(inst_handle, pin_name, pin_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be set. Table 33 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 33: *pin_name* Values in pcimaster_set_pin Command

Device Pin Name	<i>pin_name</i> Parameter Value
ACK64#	PCIMASTER_PACK64NN_PIN
AD(<i>n</i>) (lower 32 bits)	PCIMASTER_PADATAN_PIN
AD(<i>n</i>) (upper 32 bits)	PCIMASTER_PDn_PIN
C/BE(<i>n</i>) (lower 4 bits)	PCIMASTER_PCXBENNn_PIN
C/BE(<i>n</i>) (upper 4 bits)	PCIMASTER_PBENNn_PIN
CLK	PCIMASTER_PCLKRUNNN_PIN
DEVSEL#	PCIMASTER_PDEVSELNN_PIN
FRAME#	PCIMASTER_PFRAMENN_PIN
IRDY#	PCIMASTER_PIRDYNN_PIN
LOCK#	PCIMASTER_PLOCKNN_PIN
PAR	PCIMASTER_PPAR_PIN
PAR64	PCIMASTER_PPAR64_PIN
PERR#	PCIMASTER_PPERRNN_PIN
REQ#	PCIMASTER_PREQNN_PIN

Table 33: *pin_name* Values in pcimaster_set_pin Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
REQ64#	PCIMASTER_PREQ64NN_PIN
SEERR#	PCIMASTER_PSERRNN_PIN
STOP#	PCIMASTER_PSTOPNN_PIN
TRDY#	PCIMASTER_PTRDYN_PIN
AD[31:0] (lower 32 bits)	PCIMASTER_PADATA_BUS
AD[31:0] (upper 32 bits)	PCIMASTER_PD_BUS
C/BE[3:0] (lower 4 bits)	PCIMASTER_PCXBENN_BUS
C/BE[3:0] (upper 4 bits)	PCIMASTER_PBENN_BUS

pin_value The level to which to set the pin. A value of 1 drives the pin high, a value of 0 drives the pin low. Values other than 0 and 1 are not supported.

status A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimaster_set_pin command sets a specified pin or bus to the supplied value. A *pin_value* of 1 drives the pin high, a *pin_value* of 0 drives the pin low, and anything else drives it unknown. Not all pins are driven to *pin_value* at the same time. The model drives a pin to the supplied value at the time listed in the PCI specification.

Prototypes

C

```
void pcimaster_set_pin (
    const int      inst_handle,
    const int      pin_name,
    const FLEX_VEC pin_value,
    int            *status);
```

VHDL

```
procedure pcimaster_set_pin (
    inst_handle      : in integer;
    pin_name         : in integer;
    pin_value        : in bit_vector;
    status           : out integer);
```

Verilog

```
task pcimaster_set_pin;
    input    [31:0] inst_handle;
    input    [31:0] pin_name;
    input    [`PCIMASTER_MAX_BUS_WIDTH:0] pin_value;
    output    [31:0] status;
```

Vera

```
ModelObject.set_pin(
    integer      pin_name,
    bit [`PCIMASTER_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);
```

Examples

The following examples set the PCIMASTER_PTRDYNN_PIN pin to a value of 1.

C

```
/* C syntax */
pcimaster_set_pin(Id_1, PCIMASTER_PTRDYNN_PIN, "b1", &status);
```

VHDL

```
-- VHDL syntax
pcimaster_set_pin(Id_1, PCIMASTER_PTRDYNN_PIN, "1", status);
```

Verilog

```
// Verilog syntax
pcimaster_set_pin(Id_1, `PCIMASTER_PTRDYNN_PIN, 1b'1, status);
```

Vera

```
pcimaster1.set_pin(`PCIMASTER_PTRDYNN_PIN, 1b'1, status);
```

pcimaster_set_reg

Sets the value of the PCIMASTER_STATUS_REG register.

Syntax

```
pcimaster_set_reg (inst_handle, reg_name, reg_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register whose data is to be returned. The pcimaster_fx has only one register; therefore the only possible value for this parameter is PCIMASTER_STATUS_REG.
<i>reg_value</i>	The value to which the PCIMASTER_STATUS_REG register will be set. This value is a 4-bit vector that represents the termination status of a transaction on the PCI or PCI-X bus. Table 34 lists the possible register values and their definitions.

Table 34: Register Value Definitions for PCIMASTER_STATUS_REG

Register Value	Definition
0	Completion
1	Master abort
2	Target abort
3	disconnect_a
4	disconnect_b
5	Retry and disconnect_c
6	Retry limit reached and command discarded
7	Disconnect on ADB from master
8	Disconnect on ADB from slave
9	Single data phase disconnect

Table 34: Register Value Definitions for PCIMASTER_STATUS_REG (cont.)

Register Value	Definition
10	Split response
11	Split completion discarded
12	Split completion message signal out of range

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_set_reg` command sets the value of the `PCIMASTER_STATUS_REG` register. [Table 34 on page 197](#) defines the values to which you can set the register.

Prototypes

C

```
void pcimaster_set_reg (
    const int      inst_handle,
    const int      reg_name,
    const FLEX_VEC reg_value,
    int            *status);
```

VHDL

```
procedure pcimaster_set_reg (
    inst_handle      : in integer,
    reg_name         : in integer,
    reg_value        : in bit_vector,
    status           : out integer);
```

Verilog

```
task pcimaster_set_reg;
    input    [31:0] inst_handle;
    input    [31:0] reg_name;
    input    [`PCIMASTER_MAX_REG_WIDTH:0] reg_value;
    output   [31:0] status;
```

Vera

```
ModelObject.set_reg (
    integer    reg_name,
    bit        [ `PCIMASTER_MAX_REG_WIDTH:0] reg_value,
    var integer    status);
```

Examples

The following command sets the PCIMASTER_STATUS_REG register to a value of 1.

```
//Verilog Syntax
pcimaster_set_reg(Id_1, `PCIMASTER_STATUS_REG, 4'h0,status);

--VHDL Syntax
pcimaster_set_reg(Id_1, PCIMASTER_STATUS_REG, hex("0"), status);
```

The following C example configures PCIMASTER_STATUS_REG registers for interrupts.

```
/* C Example */
/* Testbench: Configure the PCIMASTER_STATUS_REG register to recognize
all interrupts.
pcimaster_set_reg(Id_1, PCIMASTER_STATUS_REG, "h0", &status);
```

pcimaster_set_timing_control

Enables and disables timing checks and access delays. You can change group or individual timing checks.

Syntax

```
pcimaster_set_timing_control(inst_handle, timing_param_index, enable_value,  
                             status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>timing_param_index</i>	A constant that indicates the name of the timing check to be enabled or disabled. Table 59 on page 337 lists the group constant values. Table 60 on page 338 lists the individual constant values.
<i>enable_value</i>	One of two constants that specifies whether to enable (FLEX_ENABLE) or disable (FLEX_DISABLE) the specified timing check.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The pcimaster_set_timing_control command lets you enable or disable a specified timing check or a group of timing checks at runtime. You can also use this command to control individual pin-to-pin timing checks for existing paths in the timing model. The model defaults to all timing checks enabled.

Prototypes

C

```
void pcimaster_set_timing_control (
    const int      inst_handle,
    const int      timing_param_index,
    const int      enable_value,
    int            *status);
```

VHDL

```
procedure pcimaster_set_timing_control (
    inst_handle          : in integer;
    timing_param_index   : in integer;
    enable_value         : in integer;
    status               : out integer);
```

Verilog

```
task pcimaster_set_timing_control;
    input    [31:0] inst_handle;
    input    [31:0] timing_param_index;
    input    [31:0] enable_value;
    output   [31:0] status;
```

Vera

```
ModelObject.set_timing_control (
    integer      timing_param_index,
    integer      enable_value,
    var integer  status);
```

Examples

C

```
/* C syntax - turn off all setup timing checks */
pcimaster_set_timing_control(Id_1, PCIMASTER_SETUP,
    FLEX_DISABLE, &status);

/* C syntax - turn off the hold check from CLKOUT(1h) to PREQNN */
pcimaster_set_timing_control(Id_1,
    PCIMASTER_TH_PLLK_PREQNN_AV_P66MHZ_FAST, `FLEX_DISABLE, &status);
```

VHDL

```
-- VHDL syntax - turn off all setup timing checks
pcimaster_set_timing_control(Id_1, PCIMASTER_SETUP,
    FLEX_DISABLE, status);

-- VHDL syntax - turn off the hold check from CLKOUT(1h) to PREQNN
pcimaster_set_timing_control(Id_1,
    PCIMASTER_TH_PCLK_LH_PREQNN_AV_P66MHZ_FAST, FLEX_DISABLE, status);
```

Verilog

```
// Verilog syntax - turn off all setup timing checks
pcimaster_set_timing_control(Id_1, `PCIMASTER_SETUP,
    `FLEX_DISABLE, status);

// Verilog syntax - turn off the hold check from CLKOUT(1h) to PREQNN
pcimaster_set_timing_control(Id_1,
    `PCIMASTER_TH_PLK_LH_PREQNN_AV_P66MHZ_FAST, `FLEX_DISABLE, status);
```

Vera

```
//Set all timing checks.
pcimaster1.set_timing_control(`PCIMASTER_TCHECKS, `FLEX_DISABLE,
    status);
```

pcimaster_write_continue

Provides burst write capabilities.

Syntax

```
pcimaster_write_continue(inst_handle, byten, data, delay, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If you use PCIMASTER_MEM_WRITE_BLOCK or PCIMASTER_MEM_WRITE_BLOCK64 for the <i>wtype</i> value in the pcimaster_write_cycle command, the model ignores this parameter.
<i>data</i>	The vector value that the operation must write. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted. In PCI-X mode, the model ignores this parameter.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero

indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_write_continue` command provides burst write capabilities. When you perform data write functions that require multiple data phases, you must use one `pcimaster_write_continue` command for each data phase executed beyond the first.

Always keep the `pcimaster_write_cycle` and `pcimaster_write_continue` commands together. Do not intersperse them with other commands. For example:

```
ul.pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h045166F9,
32, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
status);
```

Prototypes

C

```
void pcimaster_write_continue (
    const int      inst_handle,
    const FLEX_VEC byten,
    const FLEX_VEC data,
    const int      delay,
    int            wait_mode,
    int            *status);
```

VHDL

```

procedure pcimaster_write_continue (
    inst_handle : in integer;
    byten       : in bit_vector(7 downto 0);
    data        : in std_logic_vector(63 downto 0);
    delay       : in integer;
    wait_mode   : in integer;
    status      : out integer);

```

Verilog

```

task pcimaster_write_continue;

    input    [31:0] inst_handle;
    input    [7:0] byten;
    input    [63:0] data;
    input    [31:0] delay;
    input    wait_mode;
    output   [31:0] status;

```

Vera

```

ModelObject.write_continue(
    bit  [7:0]    byten,
    bit  [63:0]   data,
    integer      delay,
    integer      wait_mode,
    var integer   status);

```

Examples**C**

```

/* C syntax */
pcimaster_write_cycle(id, PCIMASTER_IO_WRITE, "h00000000 00000000", 2,
    "b0000 0011", "h00000000 22220000", FLEX_FALSE, FLEX_WAIT_F,
    &status);

pcimaster_write_continue(id, "b0000 0011", "h00000000 33330000", 1,
    FLEX_WAIT_T, &status);

```

VHDL

```

-- VHDL syntax
pcimaster_write_cycle(id, PCIMASTER_IO_WRITE, hex("00000000 00000000"),
    2, B "0000 0011", hex("00000000 22220000"), FLEX_FALSE, FLEX_WAIT_F,
    status)

pcimaster_write_continue(id, B "0000 0011", hex("00000000 33330000"), 1,
    FLEX_WAIT_T, status);

```

Verilog

```
// Verilog syntax
pcimaster_write_cycle(id, `PCIMASTER_IO_WRITE, 'h0000, 2, 'b0011,
    'h22220000, 0, `FLEX_FALSE, 'FLEX_WAIT_F, status);

pcimaster_write_continue(id, 'b0011, 'h33330000, 1, 'FLEX_WAIT_T,
    status);
```

Vera

```
// Write burst of 4 bytes
// Continue command needed because AD[1:0] = 01.
pcimaster1.write_cycle(`PCIMASTER_MEM_WRITE_BLOCK, 32'h0FFF0001, 4,
    4'h0, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster1.write_continue( 4'hF, 32'h01010101, 1, `FLEX_WAIT_F, status);
```

pcimaster_write_cycle

Executes a model write cycle on the bus.

Syntax

```
pcimaster_write_cycle(inst_handle, wtype, addr, tc, byten, data, delay, lock,  
                     wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>wtype</i>	A constant that specifies the type of write operation to perform. Table 35 shows the legal values.

Table 35: *wtype* Values in pcimaster_write_cycle Command

Constant Values for <i>wtype</i> Parameter	Description
PCIMASTER_CONFIG_WRITE	Configuration write cycle. (In PCI-X mode, this is a DWORD transaction.)
PCIMASTER_IO_WRITE	I/O write cycle. (In PCI-X mode, this is a DWORD transaction.)
PCIMASTER_MEM_WRITE	Memory write cycle.
PCIMASTER_MEM_WRITE64	64-bit memory write cycle.
PCIMASTER_MEM_WRITE_INV (PCI Mode only)	Memory write and invalidate cycle.
PCIMASTER_MEM_WRITE_INV64 (PCI Mode only)	64-bit memory write and invalidate cycle.
PCIMASTER_MEM_WRITE_BLOCK (PCI-X mode only)	PCI-X memory write block cycle in 32-bit data mode. When you use this <i>wtype</i> , the model ignores the <i>byten</i> and <i>delay</i> parameters.
PCIMASTER_MEM_WR_BLK_ALIAS (PCI-X mode only)	Alias to PCIMASTER_MEM_WRITE_BLOCK.
PCIMASTER_MEM_WRITE_BLOCK64 (PCI-X mode only)	PCI-X memory write block cycle in 64-bit data mode. When you use this <i>wtype</i> , the model ignores the <i>byten</i> and <i>delay</i> parameters.

Table 35: wtype Values in pcimaster_write_cycle Command (cont.)

Constant Values for <i>wtype</i> Parameter	Description
PCIMASTER_MEM_WR_BLK_ALIAS64 (PCI-X mode only)	Alias to PCIMASTER_MEM_WRITE_BLOCK64.
PCIMASTER_RSVD5	Reserved cycle.
PCIMASTER_RSVD9 (PCI Mode only)	Reserved cycle.

addr

The address at which the command is to begin writing data. For I/O write bursts, if the supplied *addr*[1:0] and *byten* values in the first pcimaster_write_cycle command are an illegal combination according to the PCI specification, the pcimaster_fx drives the user-supplied *addr* value. However, if *addr*[1:0] and *byten* are a legal combination, then the value for *addr*[1:0] in subsequent pcimaster_write_continue commands is derived from the supplied *byten* value. Legal values are any 32-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.

tc

An integer (in the range 1 to 2147483648) that specifies the number of data transfers to occur on this command. When *tc* is set to 1, a single write occurs. When *tc* is greater than 1, the write_cycle command must be immediately followed by the number of write_continue commands needed to complete the burst transfers (*tc* – 1).

In PCI-X mode, the *tc* parameter represents the byte count, part of the attribute for the sequence. The actual transfer count (the number of data phases) is calculated based on this byte count and on the starting address specified using the *addr* parameter. You must calculate the number of data phases required to transfer all the bytes and use the correct number of pcimaster_write_continue commands. The model ignores extra pcimaster_write_continue commands. If you do not use enough pcimaster_write_continue commands, model behavior is unpredictable.

<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. If you use the <i>wtype</i> value of PCIMASTER_MEM_WRITE_BLOCK or PCIMASTER_MEM_WRITE_BLOCK64, the model ignores this parameter.
<i>data</i>	The vector value that the operation must write. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted. In PCI-X mode, the model ignores this parameter.
<i>lock</i>	A boolean that specifies whether the operation should be locked. A value of FLEX_TRUE locks the request. In a locked sequence of operations, intermediate commands must continue to specify the lock value. The last locked command in the transaction must reset the <i>lock</i> value to FLEX_FALSE. The default is FLEX_FALSE.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue

and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimaster_write_cycle` command executes PCI write cycles. If the `pcimaster_fx` does not own the bus when the command starts execution, the command begins execution by arbitrating for the bus.

Always keep the `pcimaster_write_cycle` and `pcimaster_write_continue` commands together. Do not intersperse them with other commands. For example:

```
ul.pcimaster_write_cycle(id_10, `PCIMASTER_MEM_WRITE_BLOCK, 32'h045166F9,
32, 4'h2, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h23232323, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h45454545, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_F,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
status);
ul.pcimaster_write_continue(id_10, 4'h0, 32'h67676767, 0, `FLEX_WAIT_T,
status);
```

The PCI specification states that a device should only seek to establish a new locked condition (assert the LOCK pin) on a read transaction. Subsequent transactions can be write or read. The `pcimaster_fx` therefore can assert LOCK on a write or a read. The user must ensure that a read transaction is used to establish a lock.

Prototypes

C

```
void pcimaster_write_cycle (
    const int          inst_handle,
    const int          wtype,
    const FLEX_VEC     addr,
    const int          tc,
    const FLEX_VEC     byten,
    const FLEX_VEC     data,
    const int          delay,
    const int          lock,
    int                wait_mode,
    int                *status);
```

VHDL

```
procedure pcimaster_write_cycle (
    inst_handle      : in integer;
    wtype            : in integer;
    addr             : in bit_vector(31 downto 0);
    tc               : in integer;
    byten            : in bit_vector(7 downto 0);
    data             : in std_logic_vector(63 downto 0);
    delay            : in integer;
    lock             : in boolean;
    wait_mode        : in integer;
    status           : out integer);
```

Verilog

```
task pcimaster_write_cycle;
    input      [31:0] inst_handle;
    input      [31:0] wtype;
    input      [31:0] addr;
    input      [31:0] tc;
    input      [7:0] byten;
    input      [63:0] data;
    input      [31:0] delay;
    input      lock;
    input      wait_mode;
    output     [31:0] status;
```

Vera

```

ModelObject.write_cycle(
    integer          wtype,
    bit  [31:0]      addr,
    integer          tc,
    bit  [7:0]       byten,
    bit  [63:0]      data,
    integer          delay,
    integer          lock,
    integer          wait_mode,
    var integer      status);

```

Examples**C**

```

/* C syntax */
pcimaster_write_cycle(Id_11, PCIMASTER_CONFIG_WRITE, "h00000004", 1,
    "h00", "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_IO_WRITE, "h00000000", 1, "h00",
    "h0000000011110000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE, "hff000004", 1, "h00",
    "h0000000022220000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE64, "h00000000", 1,
    "h00", "haaaabbbb00000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE_INV, "h00000000", 1,
    "h00", "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_RSVD5, "h00000000", 1, "h00",
    "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

pcimaster_write_cycle(Id_11, PCIMASTER_RSVD9, "h00000000", 1, "h00",
    "h0000000000000000", 0, FLEX_FALSE, FLEX_WAIT_F, &status);

```

VHDL

```

-- VHDL syntax
pcimaster_write_cycle(Id_11, PCIMASTER_CONFIG_WRITE, X"00000004", 1,
    X"00", hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, PCIMASTER_IO_WRITE, X"00000000", 1, X"00",
    hex("0000000011110000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE, X"ff000004", 1, X"00",
    hex("0000000022220000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

```

```

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE64, X"00000000", 1,
    X"00", hex("aaaabbbb00000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, PCIMASTER_MEM_WRITE_INV, X"00000000", 1,
    X"00", hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, PCIMASTER_RSVD5, X"00000000", 1, X"00",
    hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, PCIMASTER_RSVD9, X"00000000", 1, X"00",
    hex("0000000000000000"), 0, FLEX_FALSE, FLEX_WAIT_F, status);

```

Verilog

```

// Verilog syntax
pcimaster_write_cycle(Id_11, `PCIMASTER_CONFIG_WRITE, 'h00000004, 1,
    'h0, 'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_IO_WRITE, 'h00000000, 1, 'h0,
    'h11110000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_MEM_WRITE, 'hff000004, 1, 'h0,
    'h22220000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_MEM_WRITE64, 'h00000000, 1,
    'h00, 'haaaabbbb00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_MEM_WRITE_INV, 'h00000000, 1,
    'h0, 'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_RSVD5, 'h00000000, 1, 'h0,
    'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

pcimaster_write_cycle(Id_11, `PCIMASTER_RSVD9, 'h00000000, 1, 'h0,
    'h00000000, 0, `FLEX_FALSE, `FLEX_WAIT_F, status);

```

Vera

```

// Write burst of 4 bytes
// Continue command needed because AD[1:0] = 01.
pcimaster1.write_cycle(`PCIMASTER_MEM_WRITE_BLOCK, 32'h0FFF0001, 4,
    4'h0, 32'h44444444, 3, `FLEX_FALSE, `FLEX_WAIT_F, status);
pcimaster1.write_continue( 4'hF, 32'h01010101, 1, `FLEX_WAIT_F, status);

```

pcimaster_write_special_cyc

Generates a PCI special cycle.

Syntax

```
pcimaster_write_special_cyc(inst_handle, byten, data, delay, lock, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>byten</i>	The byte enables for this data phase. Legal values are any 8-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>data</i>	The vector value that the operation must write. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>delay</i>	An integer (in the range of 0 to 2147483648) that specifies the number of wait cycles to insert before each assertion of IRDY#. The <i>delay</i> value represents the number of clock states the model requires before it can send or receive data; however, the actual delay before data is transferred may be longer, depending on the delay states specified by the pcislave_fx. A value of 0 means that no wait states are inserted.
<i>lock</i>	A boolean that specifies whether the operation should be locked. A value of FLEX_TRUE locks the request. In a locked sequence of operations, intermediate commands must continue to specify the lock value. The last locked command in the transaction must reset the <i>lock</i> value to FLEX_FALSE. The default is FLEX_FALSE.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimaster_write_special_cyc command generates a PCI special cycle. The Synopsys' pcislave_fx, while reading and understanding the special cycle, does not respond to it. Typically, responses to the special cycle are handled based on the particular features of a customer design.

Prototypes

C

```
void pcimaster_write_special_cyc (
    const int      inst_handle,
    const FLEX_VEC byten,
    const FLEX_VEC data,
    const int      delay,
    const int      lock,
    int            wait_mode,
    int            *status);
```

VHDL

```
procedure pcimaster_write_special_cyc (
    inst_handle : in integer;
    byten       : in bit_vector(7 downto 0);
    data        : in std_logic_vector(63 downto 0);
    delay       : in integer;
    lock        : in boolean;
    wait_mode   : in integer;
    status      : out integer);
```

Verilog

```

task pcimaster_write_special_cyc;
    input      [31:0] inst_handle;
    input      [7:0] byten;
    input      [63:0] data;
    input      [31:0] delay;
    input      lock;
    input      wait_mode;
    output     [31:0] status;

```

Vera

```

ModelObject.write_special_cyc(
    bit  [7:0]      byten,
    bit  [63:0]    data,
    integer        delay,
    integer        lock,
    integer        wait_mode,
    var integer    status);

```

Examples**C**

```

/* C syntax */
pcimaster_write_special_cyc(id, "h00", "h0000000012345678", FLEX_FALSE,
    FLEX_WAIT_T, &status);

```

VHDL

```

-- VHDL syntax
pcimaster_write_special_cyc(id, X"00", hex("0000000012345678"),
    FLEX_FALSE, FLEX_WAIT_T, status);

```

Verilog

```

// Verilog syntax
pcimaster_write_special_cyc(id, `h0, `h12345678, 0, `FLEX_FALSE,
    `FLEX_WAIT_T, status);

```

Vera

```

// Verilog syntax
pcimaster1.write_special_cyc(4'h2, 32'h11111111, 3, `FLEX_FALSE,
    `FLEX_WAIT_T, status);

```

6

Using the pcislave_fx

Introduction

The pcislave_fx FlexModel emulates the protocol of a PCI or PCI-X bus slave device at the pin and bus-cycle levels and performs timing violation checks. This enables you to verify the functions of a user-designed PCI or PCI-X interface.

Each pcislave_fx command performs the required cycles on the bus and handles the entire transaction, from requesting the bus to terminating the transaction. In conventional PCI mode, the pcislave_fx model is a dependent device that does not initiate PCI bus cycles or arbitrate for bus ownership. Instead, it responds to cycles initiated by the pcimaster_fx and verifies returning data and controls. In PCI-X mode, the pcislave_fx initiates split completion cycles.



Note

In conventional PCI mode, the pcislave_fx is backward-compatible with previous versions of the model, with a few exceptions. For more information, see [Appendix B on page 367](#).

pcislave_fx Supported Functions

The pcislave_fx FlexModel supports dynamic memory allocation for the storage of the data. You can later retrieve the data by executing a read operation to the same address. The model also supports the following functions:

- PCI-X mode (see [“Using the pcislave_fx in PCI-X Mode” on page 220](#))
- Split Completions and Split Completion Messages (class 2). Refer to [“pcislave_fx Command Summary” on page 220](#) for more information on how to generate Class 1 Split Completion Error messages.
- Split completion
- HDL, C, and Vera command modes (see [“Command Modes,” in the *FlexModel User's Manual*](#))
- 64-bit extension
- 64-bit addressing
- Address and data stepping
- Cache support
- Configuration registers
 - cacheline size
 - class code, revision ID
 - command (I/O space access enable, memory space access enable, special cycle enable, parity error response, SERR# enable)
 - device id, vendor id
 - header type
 - status (target abort, system error, and parity error)
- Error reporting (PERR# and SERR#)
- Exclusive accesses
- Master-initiated termination (completion and abort; plus disconnect on next ADB in PCI-X mode)
- Target-initiated termination (disconnect, retry, and abort; plus split response in PCI-X mode)
- Parity

- Read and write transactions (memory, I/O, and configuration), with split completion in PCI-X mode
- Special cycle parity monitoring
- Three memory spaces and three I/O spaces
- Error generation (bad data and address parity)
- Bus command (pcislave_load_file) to load a memory file into the testbench.
- The pcislave_fx contains some features that may be helpful in simulating how a master device interacts with a bridge. First, the pcislave_fx may be configured for up to three memory and I/O address ranges. The slave may also be configured to issue split responses. The slave can be configured to respond to both type 0 and 1 configuration write and read commands. In addition, the slave can be configured to issue SCM's typical of a bridge device. Refer to [Table 37 on page 227](#) for a listing of configuration types.

pcislave_fx Modeling Exceptions and Unsupported Features

- The pcislave_fx model differs from the PCI Local Bus Specification in that the following pins are not supported:
 - TDI
 - TDO
 - TCK
 - TMS
 - TRST#
- The pcislave_fx model does not support the JTAG standard.
- None of the PCI models support save and restore (restart) capabilities. No FlexModels support these features.
- The pcislave_fx model may not perform correctly in all circumstances when a master device (requester) changes data width between the Split Response and the subsequent Split Completion.

Using the pcislave_fx in PCI-X Mode

In PCI-X mode, the model is compatible with the PCI-X Addendum to the PCI Specification.

PCI-X mode is turned off by default. To turn PCI-X mode on, use the following command immediately after using the flex_get_inst_handle command:

```
pcislave_configure(PCISLAVE_PCIX, FLEX_TRUE, status);
```

To turn PCI-X mode off, use the following command:

```
pcislave_configure(PCISLAVE_PCIX, FLEX_FALSE, status);
```

You can use either 66MHz or 133MHz timing in PCI-X mode. In PCI-X mode, you need to set the TimingVersion generic/defparameter to “pcix”. When TimingVersion is set to “pcix”, the model uses 66MHZ timing when the P66MHZ pin is set to 0, and 133MHZ timing when the P66MHZ pin is set to 1. The default is 0.



Note

In PCI-X mode, the P66MHZ pin can be considered as either a slow or fast pin. When set to true (1), the model uses a faster frequency (133MHz). When set to false (0), the model uses a slower frequency (66MHz). The name P66MHZ was retained for backward-compatibility, and does not mean that the model is operating at 66MHz.

pcislave_fx Command Summary

Table 36 lists the commands that can be issued by the pcislave_fx model. To see a detailed description of the command, including syntax, usage description, parameters, and examples, click on the command name.

Table 36: pcislave_fx Command Summary

Command Name	Description
pcislave_addr_req	Requests the model to read the value of a specified memory location.
pcislave_configure	Modifies the model's operating conditions.
pcislave_configure_delay	Specifies the TRDY# delay for burst cycles.
pcislave_dump_file	Dumps memory space contents into an external file.
pcislave_load_file	Loads memory space contents from an external file.

Table 36: pcislave_fx Command Summary (cont.)

Command Name	Description
pcislave_output_enable	Enables or disables output for a specified bidirectional or output pin or bus.
pcislave_pin_req	Requests the model to read the value of a specified pin or bus.
pcislave_pin_rslt	Returns the results of a previous pcislave_pin_req command.
pcislave_read_rslt	Returns the results of a previous pcislave_addr_req command.
pcislave_request	Allows the response of the model to be delayed by a specific number of cycles.
pcislave_set_msg_level	Controls the level of messaging that the model displays during simulation.
pcislave_set_addr	Sets the value of a specified memory location.
pcislave_set_pin	Sets the specified pin or bus to the supplied value.
pcislave_set_timing_control	Sets runtime values for individual or group timing checks.
Global FLEX Commands (click on command name to jump to the command's description in the <i>FlexModel User's Manual</i>)	
flex_get_cmd_status	Checks for a valid command tag in the command queue.
flex_clear_queue	Clears command core queues for specified inst_handle.
flex_get_inst_handle	Provides a unique instance handle for a new instance.
flex_print_msg	Outputs the specified message to the screen.
flex_run_program	Switches the command source to a compiled C program.
flex_synchronize	Synchronizes the model instance with the number of instances specified.

pcislave_fx Command Reference

This section contains command reference pages for all commands used by the pcislave_fx FlexModel. The reference pages provide descriptions for each command along with syntax, parameters, prototypes, and examples. The commands are listed in alphabetical order.

For information about the command naming structure used with FlexModels, see the *[FlexModel User's Manual](#)*.

pcislave_addr_req

Requests the model to retrieve the value of a specified memory location.

Syntax

```
pcislave_addr_req(inst_handle, mem_space, addr, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>mem_space</i>	A constant that indicates the memory space whose value is to be retrieved. Legal values are PCISLAVE_MEM, PCISLAVE_IO, and PCISLAVE_CFG.
<i>addr</i>	The address from which the command is to begin reading data. Legal values are any 64-bit std_logic (VHDL) or register (Verilog) or FLEXVEC (C) values represented in hex or binary. Because the data returned is in 32-bit, the last two bits of this byte-address value are ignored.
<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see "The status Parameter" in the <i>FlexModel User's Manual</i> .

Description

The pcislave_pin_req command issues a request to the model to retrieve the value of a specified memory location. This command is the first half of a result command pair; the result is returned when the model executes the pcislave_read_rslt command.

The pcimaster_fx and the pcislave_fx models have a queue size of 20k for addr_req, read_cycle, and read_continue commands. The models queue up these commands when they do not have corresponding read_rslt commands. The master/slave will discard the results of addr_req, read_cycle, and read_continue commands without corresponding read_rslt commands after the 20k limit is reached. The read_rslt queue supports random access by the user (see addr mode and tag mode parameter descriptions). The read_rslt queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

Prototypes

C

```
void pcislave_addr_req (
    const int      inst_handle,
    const int      mem_space,
    const FLEX_VEC addr,
    const int      wait_mode,
    int            *status),
```

VHDL

```
procedure pcislave_addr_req (
    inst_handle      : in integer;
    mem_space        : in integer;
    addr             : in bit_vector (63 downto 0);
    wait_mode        : in boolean;
    status           : out integer);
```

Verilog

```
task pcislave_addr_req;
    input      [31:0] inst_handle;
    input      [31:0] mem_space;
    input      [63:0] addr;
    input      wait_mode;
    output     [31:0] status;
```

Vera

```
ModelObject.addr_req(
    integer      mem_space,
    bit [63:0]   addr
    integer      wait_mode,
    var integer  status);
```


Examples

C

```
/* C syntax */
pcislave_addr_req(Id_1, PCISLAVE_MEM, "h0000000011110000", FLEX_WAIT_F,
&status);
```

VHDL

```
-- VHDL syntax
pcislave_addr_req(Id_1, PCISLAVE_MEM, X"0000000011110000", FLEX_WAIT_F,
status);
```

Verilog

```
// Verilog syntax
pcislave_addr_req(Id_1, `PCISLAVE_MEM, 64'h0000000011110000,
`FLEX_WAIT_F, status);
```

Vera

```
// Vera syntax
pcislave1.addr_req(`PCISLAVE_MEM, 64'h0000000011110000, `FLEX_WAIT_F,
status);
```

pcislave_configure

Modifies the model's operating conditions.

Syntax

```
pcislave_configure(inst_handle, ctype, cvalue, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>ctype</i>	A constant that determines which of the model's operating conditions are affected by executing this pcislave_configure command. Table 37 shows the legal values.
<i>cvalue</i>	A 64-bit vector that specifies the new value of the operating condition specified by the <i>ctype</i> parameter. Table 37 shows the legal values. Note, X is not a valid value for any bit position.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .



Note

When using this command in a C control command or VHDL testbench, you must use a full 64-bit vector to specify the *cvalue*. Note, in your C Testbench, you may use the defined types FLEX_TRUE_C64 and FLEX_FALSE_C64 instead of FLEX_TRUE and FLEX_FALSE. Otherwise, you will probably receive the following warning message: “warning:improper pointer/integer combination.”

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_ABORT_LIMIT	Positive integer Default = 1026	Specifies the maximum number of burst transfers (number of data phases) before the slave terminates the cycle with a target abort.
PCISLAVE_ADDR_64	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE or 0	Enables/disables 64-bit address support.
PCISLAVE_BUS_NUM (PCI-X mode only)	8-bit vector Default = 'b00000000	Specifies the bus number of the model.
PCISLAVE_C_LINE_SIZE	8-bit vector Default = 'b00000000	Specifies the cacheline size.
PCISLAVE_CAP_POINTER (PCI-X mode only)	8-bit vector from 8'h40 to 8'hff Default = 'b00000000	Sets the Capabilities Pointer in the PCI-X configuration space. This pointer points to the Capabilities List Item.
PCISLAVE_CLS_CODE	24-bit vector Default = 'h000000	Specifies the class code.
PCISLAVE_DATA_64	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE or 0	Enables/disables 64-bit data support.
PCISLAVE_DES_MAX_CUM_READ_SIZE (PCI-X mode only)	3-bit vector Default = 'b000	Sets the Designated Maximum Cumulative Read Size portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles.
PCISLAVE_DES_MAX_READ_BC (PCI-X mode only)	2-bit vector Default = 'b11	Sets the Designated Maximum Memory Read Byte Count portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles.

Table 37: ctype and cvalue Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_DES_MAX_SPLIT_TRAN (PCI-X mode only)	3-bit vector Default = 'b000	Sets the Designated Maximum Outstanding Split Transaction portion of the PCI-X status register. This setting does not change the way the model works, but is useful for performing configure cycles.
PCISLAVE_DEV_ID	16-bit vector Default = 'h0000	Specifies the device ID
PCISLAVE_DEV_NUM	5 bit vector Default = 0	Specifies the Device Number.
PCISLAVE_DISCONNECT_ON_ADB (PCI-X mode only)	Positive integer or 0 Default = 0	Specifies the ADB at which a transaction will be terminated. The default, 0, indicates no termination. If the value is 1, the slave disconnects on every ADB. If 2, then disconnects on every other ADB; if 3, disconnects on every third ADB, and so on.
PCISLAVE_FUNC_NUM (PCI-X mode only)	3-bit vector Default = 'b000	Specifies the model's function number.
PCISLAVE_H_TYPE	8-bit vector Default = 'h00	Specifies the header type to be used.
PCISLAVE_INT_ACK	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE or 0	Specifies whether pcislave responds to an interrupt acknowledge cycle.
PCISLAVE_INT_ACK_VECTOR	32-bit vector Default = 'hFFFFFFFF	Specifies the value that the pcislave uses to respond to an interrupt acknowledge cycle.
PCISLAVE_IO_L_0	64-bit vector Default = 'h000000000000000100	Sets the lower limit on I/O space 0.
PCISLAVE_IO_L_1	64-bit vector Default = 'h000000000000000000	Sets the lower limit on I/O space 1.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_IO_L_2	64-bit vector Default = 'h00000000000000000000	Sets the lower limit on I/O space 2.
PCISLAVE_IO_U_0	64-bit vector Default = 'h00000000000000001FF	Sets the upper limit on I/O space 0.
PCISLAVE_IO_U_1	64-bit vector Default = 'h00000000000000000000	Sets the upper limit on I/O space 1.
PCISLAVE_IO_U_2	64-bit vector Default = 'h00000000000000000000	Sets the upper limit on I/O space 2.
PCISLAVE_MAX_READ_BC (PCI-X mode only)	2-bit vector Default = 'b11	Sets the Maximum Memory Read Byte Count portion of the PCI-X command register. This setting does not change the way the model works, but is useful for performing configure cycles.
PCISLAVE_MEM_L_0	64-bit vector Default = 'h00000000000000000000	Sets the lower limit on memory space 0.
PCISLAVE_MEM_L_1	64-bit vector Default = 'h00000000000000000000	Sets the lower limit on memory space 1.
PCISLAVE_MEM_L_2	64-bit vector Default = 'h00000000000000000000	Sets the lower limit on memory space 2.
PCISLAVE_MEM_U_0	64-bit vector Default = 'h00000000000000000000	Sets the upper limit on memory space 0.
PCISLAVE_MEM_U_1	64-bit vector Default = 'h00000000000000000000	Sets the upper limit on memory space 1.
PCISLAVE_MEM_U_2	64-bit vector Default = 'h00000000000000000000	Sets the upper limit on memory space 2.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_PAR_ERR_DATA_PHASE	0 or positive integer.	Sets the specific data phase on which you want to see a parity error generated. Note, PCISLAVE_PCI_ERROR must be set to 1 to activate this ctype. For example, if you want a bad PAR on data phase 5, set this value to five. Use this parameter to specify a particular data phase for the slave to either assert PERR on a write to the slave, or generate bad PAR on a read from the slave.
PCISLAVE_PAR_PAR64_SELECT	2-bit vector. Default = 0 Refer to Table 6 on page 54 for values and their results.	Determines which bus half has bad parity. Must be used in conjunction with PCISLAVE_PCI_ERROR or PCISLAVE_SPLIT_PCI_ERROR.
PCISLAVE_PCI_ERROR	0 or positive integer Note: See Table 38 on page 235 for a description of these values. Default = 0	Forces model to execute an illegal PCI or PCI-X cycle. See Table 38 on page 235 for a description of possible <i>cvalue</i> values.
PCISLAVE_PCIX	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE	Enables or disables PCI-X mode. FLEX_TRUE turns PCI-X mode on, and FLEX_FALSE turns it off.
PCISLAVE_REV_ID	8-bit vector Default = h'0000	Specifies the revision ID.
PCISLAVE_SC_ERR	1-bit vector Default = 0	Turns on or off the Split Completion Error bit. It is used only in conjunction with PCISLAVE_SC_MSG. By default this feature is turned off. Turn it on by setting it to a non-zero value. See “pcislave_fx Command Summary” on page 220 for additional usage information.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_SC_MSG	32-bit vector Default = 0	When set, the model will return only a Split Completion message regardless which transaction was terminated with Split Response. By default this feature is turned off. See “pcislave_fx Command Summary” on page 220 for additional usage information.
PCISLAVE_SPLIT_ADB_DISC_INCR	Integer. Default = 0	A positive value indicates the “increments” of ADBs to be disconnected after the first disconnected ADB using the PCISLAVE_SPLIT_DISC_ON_ADB. Example, if PCISLAVE_SPLIT_DISC_ON_ADB is 2 and if PCISLAVE_SPLIT_ADB_INCR is 3, then the subsequent ADB disconnect occurs at 5, 8, 11, 14, etc. This is for Split Completion transactions.
PCISLAVE_SCEM_ADDR	Any positive seven bit value. Default = 0	PCISLAVE_SCEM_ADDR value is use to replace the "Remaining Lower Address[18:12]" field of the Split Completion message. Set this field to for any outstanding split completions. This creates unexpected split completions for the master.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_SCEM_CLASS	Positive integer Default = 0, no action	<p>Slave will use this value in the message/data field of any split completion message generated while this parameter is set to a non-zero value. Consult Table 39 on page 235 for which class of errors match a given integer value.</p> <p>The slave will not automatically generate a split completion message from the setting of this parameter. Rather, on the next split completion message that is generated by the slave, the split completion message will use this value. The only split completion message that the slave will generate independent of these parameters is when Class=2, and Index=0 or 1 (byte count out of range or split write data parity error).</p>
PCISLAVE_SCEM_INDEX	Positive integer Default = 0, no action.	<p>The slave will use this value in the message/data field on any split completion message when PCISLAVE_SCEM_CLASS is set to a non-zero value. Consult Table 39 on page 235 for which class of errors match a given integer value.</p>
PCISLAVE_SPLIT_RESPONSE (PCI-X mode only)	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE or 0	Determines whether a transaction terminates with a split response. The default, 0, indicates no split response.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_SPLIT_TRAN (PCI-X mode only) (Same as PCISLAVE_MAX_SPLIT_TRAN in previous versions. The model recognizes both as the same parameter and does not break any testbenches.)	3-bit vector Default = 'b000	Sets the Maximum Outstanding Split Transaction portion of the PCI-X command register. This setting does not change the way the model works, but is useful for performing configure cycles.
PCISLAVE_SPLIT_COMPL_DELAY (PCI-X mode only)	value actual delay in clk cycles 0 = 6 delay clk cycles 1 = 5 delay clk cycles 2 = 5 delay clk cycles 3 = 6 delay clk cycles 4 = 7 delay clk cycles 5 = 8 delay clk cycles After cvalue of 5, delay clock cycle is [cvalue +3]. For example, 7 = 10 delay clk cycles.	Determines when a transaction terminating with a split response can send a request for a split completion transaction.
PCISLAVE_SPLIT_DISCONNECT_ADB (PCI-X mode only)	Positive integer or 0 Default = 0	Specifies the ADB at which the pcislave_fx is to initiate a disconnect in a split completion.
PCISLAVE_SPLIT_PCI_ERROR	0 = no error 1 = parity error (data) 2 = parity error (address) 3 = parity error (attribute) Default = FLEX_FALSE or 0	Forces the model to execute an illegal PCI-X split completion cycle. When PCISLAVE_SPLIT_PCI_ERROR is set to 1 (data), then the user cannot isolate any individual data transfers for a unique behavior. As a result, in a split completion with five data transfers, all five transfers are required to be identical: either they all have parity errors, or they have no parity errors.
PCISLAVE_START_TAG (PCI-X mode only)	5-bit vector from 5'b00000 to 5'b11111 Default = 'b00000	Specifies the starting tag number of the model.

Table 37: *ctype* and *cvalue* Values in pcislave_configure Command (cont.)

<i>ctype</i> Parameter Values	<i>cvalue</i> Parameter Values	Description
PCISLAVE_STOP_ASSERT_B4_ADB (PCI-X mode only)	Positive integer Default = 4 Ranges: 16 bit device: 0-16 32 bit device: 0-32	Determines how many cycles before the next ADB you can assert the STOP# pin for ADB disconnect. Can have only two ranges depending on device size. Any values outside range are ignored by the model.
PCISLAVE_STOP_ON_1ST_DATA_PHASE	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE	Forces the model to assert STOP# on the first data phase of a transaction for disconnecting on ADB. When this is on, it over-rides PCISLAVE_STOP_ASSERT_B4_ADB.
PCISLAVE_TERMINATION_STYLE	0 = terminate without data 1 = terminate with data Default = 0	Determines whether cycle terminates with or after data transfer.
PCISLAVE_TRANSFER_LIMIT	In conventional PCI mode: Positive integer, default = 16 Retry = 0. In PCI-X mode: 0 = Retry termination 1 = Single data phase disconnect termination	In conventional PCI mode, specifies the maximum number of transfers before disconnect. If zero, then specifies a retry. In PCI-X mode, specifies either a retry or a single data phase disconnect; only a <i>cvalue</i> of 0 or 1 is valid. Values greater than 1 force the model to ignore this setting. Set this value to 2 or greater to see uninterrupted functionality from the slave.
PCISLAVE_TYPE1_ACCESS	FLEX_TRUE or FLEX_FALSE Default = FLEX_TRUE or 1	Specifies whether the slave responds to type 1 configuration cycles.
PCISLAVE_VEN_ID	16-bit vector Default = 'h0000	Specifies the vendor ID.

Table 38: Values for *cvalue* when *ctype* is PCISLAVE_PCI_ERROR

<i>cvalue</i> Value	Conventional PCI Mode	PCI-X Mode
0	No error	No error
1	Data parity error on read (PAR)	Data parity error on read (PAR)
2	Address parity error (SERR#)	Address parity error (SERR#)
3	Data parity error reported on first data phase (PERR#). This applies to write transactions.	Attribute parity error (SERR#)
4	Data parity error reported on second data phase (PERR#). This applies to write transactions	Data parity error reported on first data phase (PERR#). This applies to write transactions

Table 39: Simulated Split Completion Error Types

PCISLAVE_SCEM_ CLASS	PCISLAVE_SCEM_ INDEX	Error Type
1	0	Master-Abort
1	1	Target-Abort
1	2	Write Data Parity Error
2	0	Byte Count Out of Range
2	1	Split Write Data Parity Error
2	2	Device-Specific

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue

and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcislave_configure command lets you modify the model's operating conditions. You can use the command to define the upper address value for 64-bit transfers, generate errors, specify command stepping and delay, and specify new device, vendor, and revision IDs. For a complete list of the operating conditions you can change, see [Table 37 on page 227](#).

Note, the pcislave_fx model does not have special registers for holding or changing the base memory and I/O addresses. You can only set the base addresses for memory and I/O with the pcislave_configure command. Refer to the section [“Examples” on page 238](#) for an example on how to set memory and I/O locations using pcislave_configure.

In terms of configuring memory addresses, you can configure the slave for up to three distinct memory segments. You can set the ranges to have identical ranges if you do not require three separate and distinct memory segments. If you wish to only have two ranges, configure all three, but give two of the ranges the same upper and lower limits. The same applies if you want only one range: set all three ranges with the same upper and lower limits. If you desire less than three memory segments, it is recommended that you not leave any of the memory bounds at their default value, as doing so could lead to unintended responses from the slave.



Note

You must use the pcislave_request command to properly set configuration variables. Refer to the command [“pcislave_request” on page 268](#) for additional information.

Regarding parity generation, note that the pcimaster's equivalent to the pcislave_set_addr(...143) command is the pcimaster_configure(PCIMASTER_DEV_CNTL_CFG) command.

Prototypes

C

```
void pcislave_configure (
    const int      inst_handle,
    const int      ctype,
    const FLEX_VEC cvalue,
    int            *status);
```

VHDL

```
procedure pcislave_configure (  
    inst_handle      : in integer;  
    ctype            : in integer;  
    cvalue           : in std_logic_vector(63 downto 0);  
    status           : out integer);
```

Verilog

```
task pcislave_configure;  
    input      [31:0] inst_handle;  
    input      [31:0] ctype;  
    input      [63:0] cvalue;  
    output     [31:0] status;
```

Vera

```
ModelObject.configure(  
    integer      ctype,  
    bit  [63:0]  cvalue,  
    var integer  status);
```

Examples

The following examples illustrate the various forms of syntax for the pcislave_configure command.

C

```
/* C syntax */
pcislave_configure(u1_id, PCISLAVE_DEV_ID, "h0000000000001234", &status);
pcislave_configure(u1_id, PCISLAVE_VEN_ID, "h0000000000005678", &status);
pcislave_configure(u1_id, PCISLAVE_REV_ID, "h0000000000000000", &status);
pcislave_configure(u1_id, PCISLAVE_H_TYPE, "h0000000000000000", &status);
pcislave_configure(u1_id, PCISLAVE_CLS_CODE, "h0000000000000000", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_0, "h0000000000000000", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_U_0, "h00000000000000ff", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_1, "h1000000000000000", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_U_1, "h100000000000ffff", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_2, "h00000000a0000000", &status);
pcislave_configure(u1_id, PCISLAVE_MEM_U_2, "h00000000a000ffff", &status);
pcislave_configure(u1_id, PCISLAVE_IO_L_0, "h0000000000000100", &status);
pcislave_configure(u1_id, PCISLAVE_IO_U_0, "h00000000000001ff", &status);
pcislave_configure(u1_id, PCISLAVE_IO_L_1, "h00000000ffffff00", &status);
pcislave_configure(u1_id, PCISLAVE_IO_U_1, "h00000000ffffffff", &status);
pcislave_configure(u1_id, PCISLAVE_IO_L_2, "h00000000b0000000", &status);
pcislave_configure(u1_id, PCISLAVE_IO_U_2, "h00000000b00000ff", &status);
pcislave_configure(u1_id, PCISLAVE_C_LINE_SIZE, "h0000000000000020",
    &status);
pcislave_configure(u1_id, PCISLAVE_ADDR_64, "h0000000000000001", &status);
pcislave_configure(u1_id, PCISLAVE_DATA_64, "h0000000000000001", &status);
pcislave_configure(u1_id, PCISLAVE_INT_ACK, "h0000000000000001", &status);
pcislave_configure(u1_id, PCISLAVE_INT_ACK_VECTOR, "h0000000000000000",
    &status);
pcislave_configure(u1_id, PCISLAVE_TERMINATION_STYLE, "h0000000000000000",
    &status);
pcislave_configure(u1_id, PCISLAVE_DEV_NUM, "h000000000000001e", &status);
```

VHDL

```
-- VHDL syntax
pcislave_configure(u1_id, PCISLAVE_DEV_ID, X"0000000000001234", status);
pcislave_configure(u1_id, PCISLAVE_VEN_ID, X"0000000000005678", status);
pcislave_configure(u1_id, PCISLAVE_REV_ID, X"0000000000000000", status);
pcislave_configure(u1_id, PCISLAVE_H_TYPE, X"0000000000000000", status);
pcislave_configure(u1_id, PCISLAVE_CLS_CODE, X"0000000000000000", status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_0, X"0000000000000000", status);
pcislave_configure(u1_id, PCISLAVE_MEM_U_0, X"00000000000000ff", status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_1, X"1000000000000000", status);
pcislave_configure(u1_id, PCISLAVE_MEM_U_1, X"100000000000ffff", status);
pcislave_configure(u1_id, PCISLAVE_MEM_L_2, X"00000000a0000000", status);
```

```

pcislave_configure(u1_id, PCISLAVE_MEM_U_2, X"00000000a000ffff", status);
pcislave_configure(u1_id, PCISLAVE_IO_L_0, X"0000000000000100", status);
pcislave_configure(u1_id, PCISLAVE_IO_U_0, X"00000000000001ff", status);
pcislave_configure(u1_id, PCISLAVE_IO_L_1, X"00000000ffffff00", status);
pcislave_configure(u1_id, PCISLAVE_IO_U_1, X"00000000ffffffff", status);
pcislave_configure(u1_id, PCISLAVE_IO_L_2, X"00000000b0000000", status);
pcislave_configure(u1_id, PCISLAVE_IO_U_2, X"00000000b00000ff", status);
pcislave_configure(u1_id, PCISLAVE_C_LINE_SIZE, X"0000000000000020",
    status);
pcislave_configure(u1_id, PCISLAVE_ADDR_64, X"0000000000000001", status);
pcislave_configure(u1_id, PCISLAVE_DATA_64, X"0000000000000001", status);
pcislave_configure(u1_id, PCISLAVE_INT_ACK, X"0000000000000001", status);
pcislave_configure(u1_id, PCISLAVE_INT_ACK_VECTOR, X"0000000000000000",
    status);
pcislave_configure(u1_id, PCISLAVE_TERMINATION_STYLE, X"0000000000000000",
    status);
pcislave_configure(u1_id, PCISLAVE_DEV_NUM, X"000000000000001e", status);

```

Verilog

// Verilog syntax

```

pcislave_configure(u1_id, `PCISLAVE_DEV_ID, 16'h1234, status);
pcislave_configure(u1_id, `PCISLAVE_VEN_ID, 16'h5678, status);
pcislave_configure(u1_id, `PCISLAVE_REV_ID, 8'h00, status);
pcislave_configure(u1_id, `PCISLAVE_H_TYPE, 8'h00, status);
pcislave_configure(u1_id, `PCISLAVE_CLS_CODE, 24'h000000, tatus);
pcislave_configure(u1_id, `PCISLAVE_MEM_L_0, 64'h0000000000000000, status);
pcislave_configure(u1_id, `PCISLAVE_MEM_U_0, 64'h00000000000000ff, status);
pcislave_configure(u1_id, `PCISLAVE_MEM_L_1, 64'h1000000000000000, status);
pcislave_configure(u1_id, `PCISLAVE_MEM_U_1, 64'h100000000000ffff, status);
pcislave_configure(u1_id, `PCISLAVE_MEM_L_2, 64'h00000000a0000000, status);
pcislave_configure(u1_id, `PCISLAVE_MEM_U_2, 64'h00000000a000ffff, status);
pcislave_configure(u1_id, `PCISLAVE_IO_L_0, 64'h0000000000000100, status);
pcislave_configure(u1_id, `PCISLAVE_IO_U_0, 64'h00000000000001ff, status);
pcislave_configure(u1_id, `PCISLAVE_IO_L_1, 64'h00000000ffffff00, status);
pcislave_configure(u1_id, `PCISLAVE_IO_U_1, 64'h00000000ffffffff, status);
pcislave_configure(u1_id, `PCISLAVE_IO_L_2, 64'h00000000b0000000, status);
pcislave_configure(u1_id, `PCISLAVE_IO_U_2, 64'h00000000b00000ff, status);
pcislave_configure(u1_id, `PCISLAVE_C_LINE_SIZE, 8'h20, status);
pcislave_configure(u1_id, `PCISLAVE_ADDR_64, `FLEX_TRUE, status);
pcislave_configure(u1_id, `PCISLAVE_DATA_64, `FLEX_TRUE, status);
pcislave_configure(u1_id, `PCISLAVE_INT_ACK, `FLEX_TRUE, status);
pcislave_configure(u1_id, `PCISLAVE_INT_ACK_VECTOR, 32'h00000000, status);
pcislave_configure(u1_id, `PCISLAVE_TERMINATION_STYLE, 0, status);
pcislave_configure(u1_id, `PCISLAVE_DEV_NUM, 64'h000000000000001e, status);

```

Vera

```
// Program slave to respond to split response
pcislave1.configure( `PCISLAVE_SPLIT_RESPONSE, `FLEX_TRUE, status);

// delay split completion by 10 clocks
pcislave1.configure( `PCISLAVE_SPLIT_COMPL_DELAY, 10, status);

//Disconnect at Next ADB
pcislave1.configure( `PCISLAVE_DISCONNECT_ON_ADB, 2, status);

//other examples
pcislave1.configure( `PCISLAVE_TRANSFER_LIMIT, 'd20, status);
pcislave1.configure( `PCISLAVE_ABORT_LIMIT, 'd100, status);
pcislave1.configure( `PCISLAVE_TERMINATION_STYLE, 'd0, status);
pcislave1.configure( `PCISLAVE_PCIX, `FLEX_TRUE, status);
```


pcislave_configure_delay

Specifies the TRDY# delay for burst cycles.

Syntax

```
pcislave_configure_delay(inst_handle, burst_num, cvalue, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>burst_num</i>	An integer or constant that specifies the memory operation within a burst. In conventional PCI mode, legal values are integers in the range of 1 to 2147483648 or the all_delays constant. The all_delays constant is predefined to equal -1 and causes the delay value to be applied across all transfers. In PCI-X mode, 1 is the only legal value, because wait states are allowed in only the first data phase. This value may be set with integer values greater than one; however, the command interprets a greater-than-one value as simply the integer one.
<i>cvalue</i>	An integer in the range of 0 to 2147483648 that specifies the TRDY# delay to occur during the <i>burst_num</i> data phase. Zero is the default value. The value range allows you to violate the PCI standard if you wish. But to meet the PCI standard, you must use a value of seven or less. A value of seven will put in seven wait states with the subsequent TRDY assertion on the eighth clock interval. This meets the PCI requirement for Target Subsequent Latency. A cvalue of eight will violate the Target Subsequent Latency requirement for the PCI standard.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The `pcislave_configure_delay` command lets you specify the TRDY# delay for burst cycles.

Prototypes

C

```
void pcislave_configure_delay (
    const int      inst_handle,
    const int      burst_num,
    const int      cvalue,
    int            *status);
```

VHDL

```
procedure pcislave_configure_delay (
    inst_handle      : in integer;
    burst_num       : in integer;
    cvalue          : in integer;
    status          : out integer);
```

Verilog

```
task pcislave_configure_delay;
    input      [31:0] inst_handle;
    input      [31:0] burst_num;
    input      [31:0] cvalue;
    output     [31:0] status;
```

Vera

```
ModelObject.configure_delay(
    integer      burst_num,
    integer      cvalue,
    var integer  status);
```

Examples

C

```
/*C syntax - configure all TRDY delays to be 2 */
pcislave_configure_delay(id, PCISLAVE_ALL_DELAYS, 2 &status);

/* Configure third transaction within burst to have a TRDY delay of 1 */
pcislave_configure_delay(id, 3, 1, &status);
```

VHDL

```
-- VHDL syntax - configure all TRDY delays to be 2
pcislave_configure_delay(id, PCISLAVE_ALL_DELAYS, 2, status);

-- Configure third transaction within burst to have a TRDY delay of 1
pcislave_configure_delay(id, 3, 1, status);
```

Verilog

```
// Verilog syntax - configure all TRDY delays to be 2
pcislave_configure_delay(id, `PCISLAVE_ALL_DELAYS, 2, status);

// Configure third transaction within burst to have a TRDY delay of 1
pcislave_configure_delay(id, 3, 1, status);
```

Vera

```
// configure 1st data phase to have a trdy# delay of 5
pcislave1.configure_delay( 1, 5, status);
```

pcislave_dump_file

Dumps memory space contents into an external file.

Syntax

```
pcislave_dump_file(inst_handle, dump_file, mem_space, start_addr, end_addr, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>dump_file</i>	A 256-character string that specifies the output file name. Use Verilog memory format, as shown in the following example. Note that these are hexadecimal addresses, and the address specified after the @ sign is a double-word address.

```
11111111
44444444
55555555
@5
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
```

<i>mem_space</i>	A constant that determines which memory space to dump. Table 40 lists the legal values.
------------------	---

Table 40: *mem_space* Values in pcislave_dump_file Command

mem_space Value	Description
PCISLAVE_MEM	Memory read or write space
PCISLAVE_IO	I/O read or write space
PCISLAVE_CFG	Configure read or write space

<i>start_addr</i>	A 64-bit vector, within the range of the specified memory space, that determines the start address of memory space to be dumped.
<i>end_addr</i>	A 64-bit vector, within the range of the specified memory space, that determines the end address of memory space to be dumped.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The `pcislave_dump_file` command dumps the contents of a `pcislave` internal memory space into an external file. The slave supports three memory spaces: memory space (`mem`), IO_space (`io`), configuration space (`cfg`).

The dump command can be used to dump only the memory regions assigned to the `pcislave_fx` model. For example, if the slave supports the memory region 00 to FF, any attempts to dump the values of a region outside the range will be ignored by the slave.

The output file uses hexadecimal Verilog memory file format—which is supported, with the exception of the comment type.



Note

The format for a memory file uses a (32 bit) word address. Hence, an address in a memory file (MIF file) is equivalent to the actual memory, I/O, or configuration address divided by 4. For example, address `h40` in an MIF represents address `h100` in the slave's memory. For configuration space addresses, you must remove the upper address bits [31:16] before dividing by 4. For example, the actual configuration address (on the bus) of `31'hc0000040` equals a `pcislave_dump_file` address of `8'h10`.

The “/*” and “*/” comment syntax in C format must occur on a separate line from the command code. For example, the following syntax is not supported:

```
@00000 /* @00000 is lost */
/* Also lost is */ abcdef01
55555555 /*
This does not work either
*/
/* neither does this
*/ 66666666
```

The correct syntax for the above would be:

```
/* @00000 is not lost */
@00000
/* abcdef01 would not be lost */
abcdef01
/* Comments should be on one line */
55555555
/* 66666666 will not be lost if it's not on the comment line */
66666666
```

Prototypes

C

```
void pcislave_dump_file (
    const int      inst_handle,
    const char*    dump_file,
    const int      mem_space,
    const FLEX_VEC start_addr,
    const FLEX_VEC end_addr,
    int            *status);
```

VHDL

```
procedure pcislave_dump_file (
    inst_handle      : in integer;
    dump_file        : in string;
    mem_space        : in integer;
    start_addr       : in std_logic_vector(63 downto 0);
    end_addr         : in std_logic_vector(63 downto 0);
    status           : out integer);
```

Verilog

```

task pcislave_dump_file;
    input      [31:0] instance;
    input      [8*`PCISLAVE_CHARMAXCNT:1] dump_file;
    input      [31:0] mem_space;
    input      [63:0] start_addr;
    input      [63:0] end_addr;
    output     [31:0] status;

```

Vera

```

ModelObject.dump_file(
    string      dump_file,
    integer     mem_space,
    bit  [63:0] start_addr,
    bit  [63:0] end_addr,
    var integer status);

```

Examples**C**

```

/* C syntax */
pcislave_dump_file(id, "mem.dump", PCISLAVE_MEM, "h0000000000000000",
    "h0000000000000000FF", &status);

```

VHDL

```

-- VHDL syntax
pcislave_dump_file(id, "mem.dump", PCISLAVE_MEM,
    X"0000000000000000",X"00000000000000FF", status);

```

Verilog

```

// Verilog syntax
pcislave_dump_file(id, "mem.dump", PCISLAVE_MEM, 64 'h0000000000000000,
    64'h00000000000000FF, status);

```

Vera

```

//Vera syntax
pcislave2.dump_file("pcislave2_mem.dmp", `PCISLAVE_MEM, 64'h0000000000000000
00, 64'h00000000000000ff, status);

```

pcislave_load_file

Loads memory space contents from an external file.

Syntax

```
pcislave_load_file(inst_handle, load_file, mem_space, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>load_file</i>	A 256-character string that specifies the input filename. The input file can be formatted in Verilog memory format, Intel Hex format, or Memory Image File format. These formats are described in the Description section for this command, on the next page. The pcislave_fx uses 64-bit addresses to store data.
<i>mem_space</i>	A constant that specifies which memory space to load. Table 41 lists the legal values.

Table 41: *mem_space* Values in pcislave_load_file Command

mem_space Value	Description
PCISLAVE_MEM	Memory read or write space
PCISLAVE_IO	I/O read or write space
PCISLAVE_CFG	Configure read or write space

<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .
---------------	---

Description

The pcislave_load_file command loads a memory space's contents from an external file. The pcislave_fx uses 64-bit addresses to store data.

You can format the input file in Verilog memory format, Intel Hex format, or Memory Image File format:

- Verilog memory file format—which is supported, with the exception of the comment type—is shown in [Figure 8](#). Note that these are hexadecimal addresses, and the address specified after the @ sign is a double-word address.

```
0000000011111111
0000000044444444
0000000055555555
@5
0000000000000001
0000000000000002
0000000000000003
0000000000000004
0000000000000005
0000000000000006
0000000000000007
0000000000000008
```

Figure 8: Verilog Memory Format



Note

The format for a memory file uses a (32 bit) word address. Hence, an address in a memory file (MIF file) is equivalent to the actual memory address, I/O address, or configuration space address divided by 4. For example, address h40 in an MIF represents address h100 in the slave's memory. For configuration space addresses, you must remove the upper address bits [31:16] before dividing by 4. For example, the actual configuration address (on the bus) of 31'hc0000040 equals a pcislave_dump_file address of 8'h10.

The “/*” and “*/” comment syntax in C format must occur on a separate line from the command code. For example, the following syntax is not supported:

```
@000000 /* @000000 is lost */
/* Also lost is */ abcdef01
55555555 /*
This does not work either
*/
/* neither does this
*/ 66666666
```

The correct syntax for the above would be:

```
/* @000000 is not lost */
@000000
/* abcdef01 would not be lost */
abcdef01
/* Comments should be on one line */
55555555
/* 66666666 will not be lost if it's not on the comment line */
66666666
```

- Memory Image File format is described in the section, “[Memory Image File \(MIF\) Format](#),” in the *SmartModel Library User's Manual*.
- Intel Hex file format contains data records with the format:

```
:BcLoff00DataCs
```

where the characters are interpreted as follows:

:	Colon—record mark (required)
Bc	Byte count—the number of bytes in data
Loff	Load offset—where in memory to store the first byte
00	Record type (8-bit, 16-bit, or 32-bit)
Data	Values to load in memory, starting at the address specified by Loff and continuing for number of bytes specified by Bc
Cs	Checksum—the value that causes the sum of all bytes in the record, including checksum, to be a multiple of 256

For example, a 10-byte set of data consisting of the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (hexadecimal 00-09) in Intel Hex file format would appear as in [Figure 9](#).

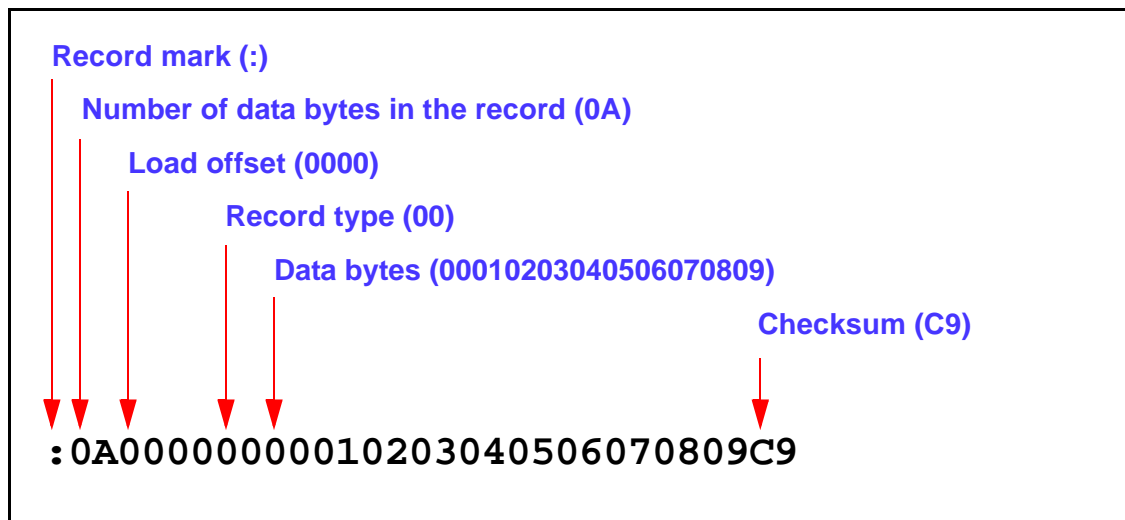


Figure 9: Intel Hexadecimal Object File Format

Prototypes

C

```
void pcislave_load_file (
    const int      inst_handle,
    const char*    load_file,
    const int      mem_space,
    int            *status);
```

VHDL

```
procedure pcislave_load_file (
    inst_handle      : in integer;
    load_file        : in string;
    mem_space        : in integer;
    status           : out integer);
```

Verilog

```
task pcislave_load_file;
    input      [31:0] inst_handle;
    input      [8*`PCISLAVE_CHARMAXCNT:1] load_file;
    input      [31:0] mem_space;
    output     [31:0] status;
```

Vera

```
ModelObject.load_file(  
    string          load_file,  
    integer         mem_space,  
    var integer     status);
```

Examples**C**

```
/* C syntax */  
pcislave_load_file(id, "mem.dat", PCISLAVE_MEM, &status);
```

VHDL

```
-- VHDL syntax  
pcislave_load_file(id, "mem.dat", PCISLAVE_MEM, status);
```

Verilog

```
// Verilog syntax  
pcislave_load_file(id, "mem.dat", `PCISLAVE_MEM, status);
```

Vera

```
//Vera syntax  
pcislave1.load_file("pcixslave1_tst.cfg", `PCISLAVE_CFG, status);  
pcislave1.load_file("pcixslave1_tst.dat", `PCISLAVE_MEM, status);  
pcislave1.load_file("pcixslave1_tst.io", `PCISLAVE_IO, status);
```

pcislave_output_enable

Enables or disables output for a specified bidirectional or output pin or bus.

Syntax

pcislave_output_enable(*inst_handle*, *pin_name*, *enable_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	The name of the pin for which output is to be enabled or disabled. Table 42 lists the legal pin names for this parameter.

Table 42: *pin_name* Values for pcislave_output_enable Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD[31:0] (lower 32 bits)	PCISLAVE_PADATA_BUS
AD[63:32] (upper 32 bits)	PCISLAVE_PD_BUS
C/BE#[3:0]	PCISLAVE_PCXBENN_BUS
C/BE#[7:4]	PCISLAVE_PBENN_BUS
ACK64#	PCISLAVE_PACK64NN_PIN
DEVSEL#	PCISLAVE_PDEVSELNN_PIN
FRAME#	PCISLAVE_PFRAMENN_PIN
IRDY#	PCISLAVE_PIRDYNN_PIN
LOCK#	PCISLAVE_PLOCKNN_PIN
PAR	PCISLAVE_PPAR_PIN
PAR64	PCISLAVE_PPAR64_PIN
PERR#	PCISLAVE_PPERRNN_PIN
REQ#	PCISLAVE_PREQNN_PIN
REQ64#	PCISLAVE_PREQ64NN_PIN
SERR#	PCISLAVE_PSERRNN_PIN

Table 42: *pin_name* Values for pcislave_output_enable Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
STOP#	PCISLAVE_PSTOPNN_PIN
TRDY#	PCISLAVE_PTRDYNN_PIN

enable_value A value of 1 or FLEX_ENABLE enables the specified output pin and a value of 0 or FLEX_DISABLE disables the specified output pin. FLEX_ENABLE and FLEX_DISABLE are predefined constants.

status A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcislave_output_enable command lets you enable or disable the output for a specified pin. This command is used to specify pins that are not directly controlled by the pcislave_fx model. Outputs are considered to be bidirectional. In normal operating mode, an output is enabled and can be either 1 or 0. If you use this command to disable a pin, it indicates a Z state and you can drive it from another source.

Prototypes

C

```
void pcislave_output_enable (
    const int      inst_handle,
    const int      pin_name,
    const int      enable_value,
    int            *status);
```

VHDL

```

procedure pcislave_output_enable (
    inst_handle      : in integer;
    pin_name         : in integer;
    enable_value     : in integer;
    status           : out integer);

```

Verilog

```

task pcislave_output_enable;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      [31:0] enable_value;
    output     [31:0] status;

```

Vera

```

ModelObject.output_enable(
    integer      pin_name,
    integer      enable_value,
    var integer  status);

```

Examples

The following examples disable (set to high-Z state) the PREQ output pin.

C

```

/* C syntax */
pcislave_output_enable(Inst_1, PCISLAVE_PREQNN_PIN, FLEX_DISABLE,
    &status);

```

VHDL

```

-- VHDL syntax
pcislave_output_enable(Inst_1, PCISLAVE_PREQNN_PIN, FLEX_DISABLE,
    status);

```

Verilog

```

// Verilog syntax
pcislave_output_enable(Inst_1, `PCISLAVE_PREQNN_PIN, `FLEX_DISABLE,
    status);

```

Vera

```

// Vera syntax
pcislave1.output_enable(`PCISLAVE_PREQNN_PIN, `FLEX_DISABLE, status);

```

pcislave_pin_req

Requests the model to retrieve the value of a specified pin or bus.

Syntax

pcislave_pin_req(*inst_handle*, *pin_name*, *wait_mode*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be retrieved. Table 43 lists the legal values for <i>pin_name</i> and the signals to which each name applies.

Table 43: *pin_name* Values in pcislave_pin_req Command

Device Pin Name	<i>pin_name</i> Parameter Value
ACK64#	PCISLAVE_PACK64NN_PIN
AD[31:0] (lower 32 bits)	PCISLAVE_PADATA_BUS
AD[63:32] (upper 32 bits)	PCISLAVE_PD_BUS
C/BE[3:0]# (lower 4 bits)	PCISLAVE_PCXBENN_BUS
C/BE[7:4]# (upper 4 bits)	PCISLAVE_PBENN_BUS
DEVSEL#	PCISLAVE_PDEVSELNN_PIN
FRAME#	PCISLAVE_PFRAMENN_PIN
GNT#	PCISLAVE_PGNTNN_PIN
IDSEL	PCISLAVE_PIDSEL_PIN
IRDY#	PCISLAVE_PIRDYNN_PIN
LOCK#	PCISLAVE_PLOCKNN_PIN
PAR	PCISLAVE_PPAR_PIN
PAR64	PCISLAVE_PPAR64_PIN
PERR#	PCISLAVE_PPERRNN_PIN
REQ#	PCISLAVE_PREQNN_PIN

Table 43: *pin_name* Values in pcislave_pin_req Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
REQ64#	PCISLAVE_PREQ64NN_PIN
RST#	PCISLAVE_PRSTNN_PIN
SBO#	PCISLAVE_PSBONN_PIN
SDONE	PCISLAVE_PSDONE_PIN
SERR#	PCISLAVE_PSERRNN_PIN
STOP#	PCISLAVE_PSTOPNN_PIN
Timing mode pin (1 or 0)	PCISLAVE_P66MHZ_PIN
TRDY#	PCISLAVE_PTRDYNN_PIN

wait_mode

One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the [FlexModel User's Manual](#) for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see ["The status Parameter"](#) in the *FlexModel User's Manual*.

Description

The pcislave_pin_req command issues a request to the model to retrieve the value of a specified pin or bus. This command is the first half of a result command pair; the result is returned when the model executes the pcislave_pin_rslt command.

Prototypes

C

```
void pcislave_pin_req (
    const int      inst_handle,
    const int      pin_name,
    const int      wait_mode,
    int            *status);
```

VHDL

```
procedure pcislave_pin_req (
    inst_handle      : in integer;
    pin_name         : in integer;
    wait_mode        : in boolean;
    status           : out integer);
```

Verilog

```
task pcislave_pin_req;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      wait_mode;
    output     [31:0] status;
```

Vera

```
ModelObject.pin_req(
    integer      pin_name,
    integer      wait_mode,
    var integer  status);
```

Examples

C

```
/* C syntax */
pcislave_pin_req(Id_1, PCISLAVE_PTRDYNN_PIN, FLEX_WAIT_F, &status);
```

VHDL

```
-- VHDL syntax
pcislave_pin_req(Id_1, PCISLAVE_PTRDYNN_PIN, FLEX_WAIT_F, status);
```

Verilog

```
// Verilog syntax  
pcislave_pin_req(Id_1, `PCISLAVE_PADATA_BUS, `FLEX_WAIT_F, status);
```

Vera

```
// Vera syntax  
pcislave1.pin_req(`PCISLAVE_PADATA_BUS, `FLEX_WAIT_F, status);
```

pcislave_pin_rslt

Returns the value of a specified pin or bus.

Syntax

```
pcislave_pin_rslt(inst_handle, pin_name, pin_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be returned. Table 44 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 44: *pin_name* Values in pcislave_pin_rslt Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD[31:0] (lower 32 bits)	PCISLAVE_PADATA_BUS
AD[63:32] (upper 32 bits)	PCISLAVE_PD_BUS
C/BE[3:0] (lower 4 bits)	PCISLAVE_PCXBENN_BUS
C/BE[7:4] (upper 4 bits)	PCISLAVE_PBENN_BUS
ACK64#	PCISLAVE_PACK64NN_PIN
AD(<i>n</i>) (lower 32 bits)	PCISLAVE_PADATAN_PIN
AD(<i>n</i>) (upper 32 bits)	PCISLAVE_PDn_PIN
C/BE(<i>n</i>) (lower 4 bits)	PCISLAVE_PCXBENNn_PIN
C/BE(<i>n</i>) (upper 4 bits)	PCISLAVE_PBENNn_PIN
DEVSEL#	PCISLAVE_PDEVSELNN_PIN
FRAME#	PCISLAVE_PFRAMENN_PIN
GNT#	PCISLAVE_PGNTNN_PIN
IDSEL	PCISLAVE_PIDSEL_PIN
IRDY#	PCISLAVE_PIRDYNN_PIN

Table 44: *pin_name* Values in pcislave_pin_rslt Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
LOCK#	PCISLAVE_PLOCKNN_PIN
PAR	PCISLAVE_PPAR_PIN
PAR64	PCISLAVE_PPAR64_PIN
PERR#	PCISLAVE_PPERRNN_PIN
REQ#	PCISLAVE_PREQNN_PIN
REQ64#	PCISLAVE_PREQ64NN_PIN
RST#	PCISLAVE_PRSTNN_PIN
SBO#	PCISLAVE_PSBONN_PIN
SDONE	PCISLAVE_PSDONE_PIN
SERR#	PCISLAVE_PSERRNN_PIN
STOP#	PCISLAVE_PSTOPNN_PIN
Timing mode pin (1 or 0)	PCISLAVE_P66MHZ_PIN
TRDY#	PCISLAVE_PTRDYN_PIN

pin_value

The returned value of the pcislave_pin_req command.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcislave_pin_rslt` command returns the value of the pin or bus specified in the corresponding `pcislave_pin_req` command. This command is the second half of the result command pair that begins with the `pcislave_pin_req` command. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or previous commands complete execution. If all previous commands have completed and the specified result is not available, the command completes with an error.

Prototypes

C

```
void pcislave_pin_rslt (
    const int      inst_handle,
    const int      pin_name,
    FLEX_VEC      pin_value,
    int            *status);
```

VHDL

```
procedure pcislave_pin_rslt (
    inst_handle      : in integer;
    pin_name         : in integer;
    pin_value        : out std_logic_vector(PCISLAVE_MAX_BUS_WIDTH-1
                                           downto 0);
    status           : out integer);
```

Verilog

```
task pcislave_pin_rslt;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    output     [`PCISLAVE_MAX_BUS_WIDTH:0] pin_value;
    output     [31:0] status;
```

Vera

```
ModelObject.pin_rslt(
    integer      pin_name,
    var bit     [`PCISLAVE_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);
```

Examples

The following examples return the result of the previously issued `pcislave_pin_req` command. The `pin_value(0)` is the returned value of the LSB of the pin result.

C

```
/* C syntax */
pcislave_pin_rslt(Id_1, PCISLAVE_PTRDYN_PIN, &pin_value, &status);
```

VHDL

```
-- VHDL syntax - returns the TrcEnd 1-bit value.
pcislave_pin_rslt(Id_1, PCISLAVE_PTRDYN_PIN, pin_value, status);
```

Verilog

```
// Verilog syntax - returns 4 bits of TrcData.
pcislave_pin_rslt(Id_1, `PCISLAVE_PADATA_BUS, pin_value, status);
```

Vera

```
// Vera syntax - returns 4 bits of TrcData.
pcislave1.pin_rslt(`PCISLAVE_PADATA_BUS, pin_value, status);
```

pcislave_read_rslt

Passes read data back to the testbench from a previous pcislave_addr_req command.

Syntax

```
pcislave_read_rslt(inst_handle, addr, cmd_tag, return_data, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>addr</i>	The address from which the command is to begin reading data. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. For address mode to work, <i>cmd_tag</i> must be set to 0. For more information, see “Using Command Results” in the <i>FlexModel User's Manual</i> .
<i>cmd_tag</i>	The tag number (the returned <i>status</i> parameter value) of any previously called pcislave_addr_req command. For address mode to work, this parameter must be set to 0. Address mode is when this command retrieves data by using only the <i>addr</i> parameter. For more information, see the <i>status</i> parameter description and “Using Command Results” in the <i>FlexModel User's Manual</i> .
<i>return_data</i>	A variable name that will hold the returned value of the pcislave_addr_req command that has a corresponding <i>cmd_tag</i> or <i>addr</i> . (For address mode to work, the <i>cmd_tag</i> parameter must be set to 0.) Legal values are any 64-bit std_logic_vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The `pcislave_read_rslt` command retrieves data from a previous `pcislave_addr_req` command. This command is the second half of the command result pair. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or until previous commands complete execution.

If all previous commands have completed and the specified result is not available, the command completes with an error (a negative or 0 returned *status* value). If the `read_cycle` command did not complete normally (for example, after a target or master abort), the `read_result` command will return “XXXXXXX”.

Because the `pcislave_request` command does not contain a `FLEX_WAIT` parameter, you must be careful about its usage in some cases: in particular, when `pcislave_request` is followed by a `pcislave_read_rslt` command. In this case, you may obtain erroneous results because the `pcislave_read_rslt` command may be processed before all the bus cycles generated by `pcislave_request` have completed. The solution is to add a command after `pcislave_request` which uses a `FLEX_WAIT_TRUE` setting to force completion of the bus cycles. A good command to use in this instance would be `pcislave_addr_req`. The command sequence would be as follows:

```
pcislave_request
pcislave_addr_req(FLEX_WAIT_TRUE)
pcislave_read_rslt
```

The `pcimaster_fx` and the `pcislave_fx` models have a queue size of 20k for `addr_req`, `read_cycle`, and `read_continue` commands. The models queue up these commands when they do not have corresponding `read_rslt` commands. The master/slave will discard the results of `addr_req`, `read_cycle`, and `read_continue` commands without corresponding `read_rslt` commands after the 20k limit is reached. The `read_rslt` queue supports random access by the user (see `addr` mode and `tag` mode parameter descriptions). The `read_rslt` queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

Prototypes

C

```
void pcislave_read_rslt(
    const int          inst_handle,
    const FLEX_VEC     addr,
    const int          cmd_tag,
    FLEX_VEC           return_data,
    int                *status);
```

VHDL

```

procedure pcislave_read_rslt(
    inst_handle      : in integer;
    addr             : in bit_vector (63 downto 0);
    cmd_tag          : in integer;
    return_data      : inout std_logic_vector (31 downto 0);
    status           : out integer);

```

Verilog

```

task pcislave_read_rslt;
    input      [31:0] inst_handle;
    input      [63:0] addr;
    input      [31:0] cmd_tag;
    inout      [31:0] return_data;
    output     [31:0] status;

```

Vera

```

ModelObject.read_rslt(
    bit  [63:0]    addr,
    integer        cmd_tag,
    var bit  [31:0] return_data,
    var integer    status);

```

Examples**C**

```

/* C syntax , address mode */
pcislave_read_rslt(id_9, "h2748ab3130341008", 0, &data, &status);

/* C syntax , tag mode */
pcislave_read_rslt(id_9, "h0000000000000000", 78, &data, &status);

```

VHDL

```

-- VHDL syntax, address mode
pcislave_read_rslt(id_9, X"2748ab3130341008", 0, data, status);

-- VHDL syntax, tag mode
pcislave_read_rslt(id_9, X"0000000000000000", 78, data, status);

```

Verilog

```

// Verilog syntax, address mode
pcislave_read_rslt(id_9, 64'h2748ab3130341008, 0, data, status);

// Verilog syntax, tag mode
pcislave_read_rslt(id_9, 'h0000000000000000, 78, data, status);

```

Vera

```
// Vera syntax, address mode
pcislave1.read_rslt(64'h2748ab3130341008, 0, data, status);

// Vera syntax, tag mode
pcislave1.read_rslt('h0000000000000000, 78, data, status);
```

pcislave_request

Delays the model's response by a specific number of cycles.

Syntax

```
pcislave_request(inst_handle, request_limit, decode, delay, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>request_limit</i>	An integer (in the range of 1 to 2147483648) that specifies the number cycles to be handled by this pcislave_request command. The request_limit corresponds to the number of times the pcislave_fx model asserts DEVSEL#. A burst transfer counts as one, and each retried cycle decrements request_limit by 1. Split completions do not decrement request_limit.
<i>decode</i>	An integer that specifies the index determining the decode speed. Table 45 shows the legal values.

Table 45: decode Index Values in pcislave_request Command

<i>decode index</i>	Decode Speed (Conventional PCI Mode)	Decode Speed (PCI-X Mode)
0	Fast	Decode A
1	Medium	Decode B
2	Slow	Decode C
3	Subtractive	SUB decode
>4	<p>If the decode label is greater than 4, the model will delay DEVSEL by (decode - 3) cycles beyond the SUBTRACTIVE decode value. Thus the delay formula is:</p> $\text{delay} = [(\text{decode_index} - 3) + \text{SUBTRACTIVE}]$ <p>This can be used to cause the master to abort on the bus. If the master abort occurs, the slave will not assert DEVSEL.</p>	

The left column values are an "index" into the decode speed settings. For example, when a value of 2 is used, the Decode Speed for PCIX mode is set to Decode C. However, you can use label values greater than three (3), in which case the label/index number helps establish the delay.

delay

An integer (in the range of 0 to 2147483648) that specifies the number of PCI clock cycles the pcislave_fx model delays before asserting TRDY#. A pcislave_request command with a decode *index* value of 1 references the same delay that is specified in this command. Note that if you set up the slave as cacheable, more delays than specified may occur, due to the cacheline status. For PCI-X burst write transactions, if you specify an odd number for this parameter, the model increments the number by one to make it even.

When you execute a pcislave_request command for a burst transfer, the *delay* value applies to only the first transfer of the burst. To specify TRDY# delay for subsequent transfers of a burst cycle in conventional PCI mode, use the pcislave_configure_delay command. In PCI-X mode, wait states are not allowed during subsequent transfers of a burst cycle.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The primary purpose of the `pcislave_request` command is to insulate the model from `pcislave_configure` commands that might come in the middle of bus transactions and confuse the model. The `pcislave_request` command serves the same purpose in the slave as commands like `pcimaster_write_cycle` or `pcimaster_read_cycle` in the master. When the master receives one of these commands, the model turns its attention away from the testbench command stream and executes the prescribed transactions. These transactions are executed with the configuration that was specified *before* the `<model>_write_cycle` or `<model>_read_cycle` commands. Similarly in the slave, when the model receives a `pcislave_request` command, it turns its attention away from the testbench command stream and executes transactions on the bus using whatever configuration was set up with `pcislave_configure` commands that were received before the `pcislave_request` command was received.

The number of transactions that the slave will execute is specified by the `request_limit` parameter. For this purpose, a “transaction” means an assertion of DEVSEL by the slave. Split Response transactions do count against `request_limit` as the slave asserts DEVSEL for these transactions. Split Completion transactions do not count against DEVSEL.

The additional parameters in `pcislave_request` specify the decode speed and Initial Target Delay (TRDY delay). After the model executes a `pcislave_request` command, the delays specified by the *decode* and *delay* parameters remain in effect until you explicitly change them by executing another `pcislave_request` command with new parameters.

When you execute a `pcislave_request` command for a burst transfer, the *delay* parameter value applies to only the first transfer of the burst. To specify TRDY# delay for subsequent transfers of a burst cycle in conventional PCI mode, use the `pcislave_configure_delay` command. In PCI-X mode, wait states are not allowed during subsequent transfers of a burst cycle.

The `request_limit` parameter has several important uses. First, the `request_limit` parameter blocks the command stream into the model for the requested number of transactions (DEVSEL driven by this slave). Second, once the `request_limit` expires, and if there are no commands being executed, the model automatically generates a new `pcislave_request` command with the same parameters except for `request_limit`, which will be set to a value of one (1). This is a convenient method to keep the slave active or alive during a simulation.

Following is a situation where the model generates internal pcislave_request commands:

```
pcislave_configure
pcislave_configure
pcislave_request // Explicit user command: request_limit = 1.
#delay_value    // Many transactions occurring during this interval.
                // Model generates internal pcislave_request(1)
                // commands.
```

At the time of *#delay*, the model will continue to process bus cycles indefinitely by virtue of its internally generated pcislave_request commands. Thus, it is not necessary to issue a pcislave_request command with a large request_limit value. Also, it is generally not necessary to calculate the exact number of cycles to be executed by the slave. In both cases, the internally generated pcislave_request commands will suffice.



Note

The internally generated pcislave_request commands can sometimes make it appear as though the slave has processed an explicit pcislave_request command, when in fact it has not. This can happen from #delay statements being intentionally or mistakenly (from parallel tasks) included in the testbench. For further reference, consult the FAQ section “[pcislave_fx](#)” on [page 397](#). While it is possible to make a simulation behave correctly in these circumstances, the testbench may be dependent on having “just the right timing” between pcislave_configure commands, the #delay insertion, and transactions on the bus. In general, robust testbenches never use #delay's with the pcislave_fx. Whether a user inserts #delays and then a pcislave_request command, or some combination of the two, a testbench writer must then be aware of when the slave is processing transactions on the bus, and when it is receiving new pcislave_configure commands.

You should always use pcislave_request after a set of pcislave_configure commands. Certain internal configuration parameters are not set and read until you block the command stream with pcislave_request. In general, if you wish to reconfigure the slave during a simulation, the configuration command sequences should occur in the following pattern:

```
pcislave_configure
pcislave_configure
pcislave_request
    // slave executes request_limit number of transactions.
    // Note, you can set request_limit to a precise value, or
    // use a #delay and let the slave generate internal
    // pcislave_request commands.
```

**Caution**

Do not rely on internally generated pcislave_request commands to set configuration values. Always explicitly invoke a pcislave_request command after any pcislave_configure command. The model's configuration values are not guaranteed to be set and correct without an explicit pcislave_request command.

Because the pcislave_request command does not contain a FLEX_WAIT parameter, you must be careful about its usage in some cases: in particular, when pcislave_request is followed by a pcislave_read_rslt command. In this case, you may obtain erroneous results because the pcislave_read_rslt command may be processed before all the bus cycles generated by pcislave_request have completed. The solution is to add a command after pcislave_request which uses a FLEX_WAIT_TRUE setting to force completion of the bus cycles. A good command to use in this instance would be pcislave_addr_req. The command sequence would be as follows:

```
pcislave_request
pcislave_addr_req(FLEX_WAIT_TRUE)
pcislave_read_rslt
```

Because the pcislave_request command blocks the command stream in the model, the model will not get another command until the correct number of DEVSEL assertions. The flex model command core will continue to get commands from the command stream however. The command core will continue to grab commands until its internal queue is full. The command core in the PCI models has an internal command queue of 20k. A command with FLEX_WAIT set to TRUE will block the flex model command core from getting further commands until the model returns with the results from the command. Thus, FLEX_WAIT can block the execution of the simulation command stream.

Prototypes

C

```
void pcislave_request (
    const int    inst_handle,
    const int    request_limit,
    const int    decode,
    const int    delay,
    int          *status);
```


VHDL

```

procedure pcislave_request (
    inst_handle      : in integer;
    request_limit    : in integer;
    decode           : in integer;
    delay            : in integer;
    status           : out integer);

```

Verilog

```

task pcislave_request;
    input      [31:0] inst_handle;
    input      [31:0] request_limit;
    input      [31:0] decode;
    input      [31:0] delay;
    output     [31:0] status;

```

Vera

```

ModelObject.request(
    integer      request_limit,
    integer      decode,
    integer      delay,
    var integer  status);

```

Examples

In the following command examples, the first cycle matching the slave's address has an address decode speed of FAST and no TRDY# delays on the data transfer. The second and third cycles have SLOW decode speeds and one TRDY# delay on the first data transfer. The fourth cycle has an address decode speed of MEDIUM and two TRDY# delays on the first data transfer.

C

```

/* C syntax */
pcislave_request(id, 1, 0, 0, &status);
pcislave_request(id, 2, 2, 1, &status);
pcislave_request(id, 1, 1, 2, &status);

```

VHDL

```

-- VHDL syntax
pcislave_request(id, 1, 0, 0, status);
pcislave_request(id, 2, 2, 1, status);
pcislave_request(id, 1, 1, 2, status);

```

Verilog

```
// Verilog syntax
pcislave_request(id, 1, 0, 0, status);
pcislave_request(id, 2, 2, 1, status);
pcislave_request(id, 1, 1, 2, status);
```

Vera

```
//Vera syntax.
pcislave1.request( 1, 0, 0, status)
pcislave1.request( 1, 3, 0, status)
pcislave1.request( 1, 2, 0, status)
```

pcislave_set_addr

Sets the value of a specified memory location.

Syntax

```
pcislave_set_addr(inst_handle, mem_space, addr, data, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>mem_space</i>	A constant that indicates the memory space whose value is to be set. Legal values are PCISLAVE_MEM, PCISLAVE_IO, and PCISLAVE_CFG.
<i>addr</i>	The absolute address at which the command is to set the data. Legal values are any 64-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right.
<i>data</i>	The data with which to set the address. Legal values are any 32-bit bit-vector (VHDL), register (Verilog), bit array (Vera), or FLEXVEC (C) represented in hex or binary, with LSB on right. Note, X is not a valid value. In other words, only 0's and 1's are legal values for any bit position.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The pcislave_set_addr command is a convenient way to write to the pcislave's internal memory without actually performing a PCI or PCI-X cycle. The pcislave_set_addr command lets you set the value of a specified memory location. You can write to memory space, IO space, or configuration space. All addresses are interpreted as DWORD aligned. The lowest two bits are ignored. To write to a nonaligned address, use a sequence of READ-MODIFY-WRITE commands.

Prototypes

C

```
void pcislave_set_addr (
    const int      inst_handle,
    const int      mem_space,
    const FLEX_VEC addr,
    const FLEX_VEC data,
    int            *status);
```

VHDL

```
procedure pcislave_set_addr (
    inst_handle      : in integer;
    mem_space        : in integer;
    addr             : in bit_vector (63 downto 0);
    data             : in bit_vector (31 downto 0);
    status           : out integer);
```

Verilog

```
task pcislave_set_addr;
    input      [31:0] inst_handle;
    input      [31:0] mem_space;
    input      [63:0] addr;
    input      [31:0] data;
    output     [31:0] status;
```

Vera

```
ModelObject.set_addr(
    integer      mem_space,
    bit [63:0]   addr,
    bit [31:0]   data,
    var integer  status);
```

Examples

C

```
/* C syntax */
pcislave_set_addr(Id_1, PCISLAVE_MEM, "h0000000011110000", "h12345678",
&status);
```

VHDL

```
-- VHDL syntax
pcislave_set_addr(Id_1, PCISLAVE_MEM, X"0000000011110000", X"12345678",
status);
```

Verilog

```
// Verilog syntax
pcislave_set_addr(Id_1, `PCISLAVE_MEM, 64'h0000000011110000,
32'h12345678, status);
```

Vera

```
// Vera syntax
pcislave1.set_addr(`PCISLAVE_MEM, 64'h0000000011110000, 32'h12345678,
status);
```

pcislave_set_msg_level

Controls the level of messaging that the model displays during simulation.

Syntax

```
pcislave_set_msg_level(inst_handle, mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>mode</i>	A 32-bit integer that controls the level of messaging or a predefined constant that controls a single message type. Each bit in the <i>mode</i> field controls one message category; if a specific bit is set to 1, the messages for that category are enabled. Table 46 lists the individual bit assignments and the corresponding predefined constants.



Note

To get command-by-command trace information and other model messages, which can be useful for debugging, set all the upper bits in the *mode* parameter to 1, as follows:

```
pcislave_set_msg_level (inst_handle, 6FFFFFFF, status);
```

Table 46: *mode* Values in pcislave_set_msg_level Command

Bit Assignment	Predefined Constant	Description
None	FATAL	Stops simulation immediately after a fatal message is reported. Fatal messages are always enabled.
bit-30	PCISLAVE_DEBUG	Displays debug messages, including command-by-command trace information.
0FFFFFFF	FLEX_ALL_MSGS	Turns on all message types except debug messages. For command-by-command trace information, use PCIMONITOR_DEBUG.
00000000	FLEX_NO_MSGS	Turns off all message types other than fatal.

Table 46: *mode* Values in pcislave_set_msg_level Command (cont.)

Bit Assignment	Predefined Constant	Description
bit-0	FLEX_ERROR	Displays error messages for situations in which the model can recover and resume simulation, such as when the model receives a command that would put it into an invalid state.
bit-1	FLEX_WARNING	Displays warning messages for situations that are not necessarily errors, such as when significant bits of an address are ignored.
bit-2	FLEX_TIMING	Displays timing messages.
bit-3	FLEX_XHANDLING	Displays X-handling messages.
bit-4	FLEX_INFO	Informs you of the status or behavior of the model.
bits 8–29	Undefined	Undefined

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcislave_set_msg_level command controls the severity level of the messages displayed by the model during simulation. Model messages are grouped into categories. Except for fatal messages, which are always enabled, you can enable or disable any category for each model instance. This command allows you to change from the default message level for this model.

Prototypes

C

```
void pcislave_set_msg_level (
    const int      inst_handle,
    const int      mode,
    int            *status);
```

VHDL

```

procedure pcislave_set_msg_level (
    inst_handle    : in integer;
    mode           : in integer;
    status         : out integer);

```

Verilog

```

task pcislave_set_msg_level;
    input    [31:0] inst_handle;
    input    [31:0] mode;
    output   [31:0] status;

```

Vera

```

ModelObject.set_msg_level(
    integer    mode,
    var integer status);

```

Examples**C**

```

/* C syntax */
pcislave_set_msg_level(Id_1, FLEXINFO, &status);

```

VHDL

```

-- VHDL syntax - turns off all messages
pcislave_set_msg_level(Id_1, FLEX_NO_MSGS, status);

-- enables INFO & WARNING messages
pcislave_set_msg_level(Id_1, FLEX_INFO + FLEX_WARNING, status);

```

Verilog

```

// Verilog syntax - turns off all messages
pcislave_set_msg_level(Id_1, `FLEX_NO_MSGS, status);

// enables INFO & WARNING messages
pcislave_set_msg_level(Id_1, `FLEX_INFO + `FLEX_WARNING, status);

// enables all messages
pcislave_set_msg_level(Id_1, 32'h0fffffff, status);

```

Vera

```

//Vera syntax
pcislave1.set_msg_level(`FLEX_NO_MSGS, status);

```


pcislave_set_pin

Sets the specified pin or bus to the supplied value.

Syntax

```
pcislave_set_pin(inst_handle, pin_name, pin_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be set. Table 47 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 47: *pin_name* Values in pcislave_set_pin Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD[31:0] (lower 32 bits)	PCISLAVE_PADATA_BUS
AD[63:32] (upper 32 bits)	PCISLAVE_PD_BUS
C/BE#[3:0]	PCISLAVE_PCXBENN_BUS
ACK64#	PCISLAVE_PACK64NN_PIN
AD(<i>n</i>) (lower 32 bits)	PCISLAVE_PADATAN_PIN
AD(<i>n</i>) (upper 32 bits)	PCISLAVE_PDn_PIN
AD(<i>n</i>) (upper 32 bits)	PCISLAVE_PDn_PIN
C/BE#(<i>n</i>)	PCISLAVE_PCXBENNn_PIN
DEVSEL#	PCISLAVE_PDEVSELNN_PIN
FRAME#	PCISLAVE_PFRAMENN_PIN
IRDY#	PCISLAVE_PIRDYNN_PIN
LOCK#	PCISLAVE_PLOCKNN_PIN
PAR	PCISLAVE_PPAR_PIN
PAR64	PCISLAVE_PPAR64_PIN

Table 47: *pin_name* Values in pcislave_set_pin Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
PERR#	PCISLAVE_PPERRNN_PIN
REQ#	PCISLAVE_PREQNN_PIN
REQ64#	PCISLAVE_PREQ64NN_PIN
SERR#	PCISLAVE_PSERRNN_PIN
STOP#	PCISLAVE_PSTOPNN_PIN
TRDY#	PCISLAVE_PTRDYN_PIN

pin_value The level to which to set the pin. A value of 1 drives the pin high, a value of 0 drives the pin low. Values other than 0 and 1 are not supported.

status A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcislave_set_pin command sets a specified pin or bus to a supplied value. A *pin_value* of 1 drives the pin high, a *pin_value* of 0 drives the pin low, and anything else drives it unknown. Not all pins are driven to *pin_value* at the same time. The model drives a pin to the supplied value at the time listed in the PCI specification.

Prototypes

C

```
void pcislave_set_pin (
    const int      inst_handle,
    const int      pin_name,
    const FLEX_VEC pin_value,
    int            *status);
```

VHDL

```

procedure pcislave_set_pin (
    inst_handle      : in integer;
    pin_name         : in integer;
    pin_value        : in bit_vector;
    status           : out integer);

```

Verilog

```

task pcislave_set_pin;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      [`PCISLAVE_MAX_BUS_WIDTH:0] pin_value;
    output     [31:0] status;

```

Vera

```

ModelObject.set_pin(
    integer      pin_name,
    bit [`PCISLAVE_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);

```

Examples

These examples set the PAR64 pin to a value of 1.

C

```

/* C syntax */
pcislave_set_pin(Id_1, PCISLAVE_PPAR64_PIN, "b1", &status);

```

VHDL

```

-- VHDL syntax
pcislave_set_pin(Id_1, PCISLAVE_PPAR64_PIN, "1", status);

```

Verilog

```

// Verilog syntax
pcislave_set_pin(Id_1, `PCISLAVE_PPAR64_PIN, 1b'1, status);

```

Vera

```

// Verilog syntax
pcislave1.set_pin(`PCISLAVE_PPAR64_PIN, 1b'1, status);

```

pcislave_set_timing_control

Enables and disables timing checks and access delays. You can change group or individual timing checks.

Syntax

pcislave_set_timing_control(*inst_handle*, *timing_param_index*, *enable_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>timing_param_index</i>	A constant that indicates the name of the timing check to be enabled or disabled. Table 61 on page 348 lists the group constant values. Table 62 on page 349 lists the individual constant values.
<i>enable_value</i>	One of two constants that specify whether to enable (FLEX_ENABLE) or disable (FLEX_DISABLE) the specified timing check.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The pcislave_set_timing_control command lets you enable or disable a specified timing check or a group of timing checks at runtime. You can also use this command to control individual pin-to-pin timing checks for existing paths in the timing model. The model defaults to all timing checks enabled.

Prototypes

C

```
void pcislave_set_timing_control (
    const int      inst_handle,
    const int      timing_param_index,
    const int      state,
    int            *status);
```

VHDL

```
procedure pcislave_set_timing_control (
    inst_handle          : in integer;
    timing_param_index   : in integer;
    state                : in integer;
    status                : out integer);
```

Verilog

```
task pcislave_set_timing_control;
    input      [31:0] inst_handle;
    input      [31:0] timing_param_index;
    input      [31:0] state;
    output     [31:0] status;
```

Vera

```
ModelObject.set_timing_control (
    integer      timing_param_index,
    integer      state,
    var integer  status);
```

Examples

C

```
/* C syntax */
pcislave_set_timing_control(Id_1, PCISLAVE_SETUP, FLEX_DISABLE,
    &status);

pcislave_set_timing_control(Id_1,
    PCISLAVE_TH_PCLK_LH_PIRDYNN_VV_P66MHZ_FAST, FLEX_DISABLE, &status);
```

VHDL

```
-- VHDL syntax - turn off all setup timing checks
pcislave_set_timing_control(Id_1, PCISLAVE_SETUP, FLEX_DISABLE,
    status);

-- Turn off the hold check from CLKOUT(1h) to INT4(ha)
pcislave_set_timing_control(Id_1,
    PCISLAVE_TH_PCLK_LH_PIRDYNN_VV_P66MHZ_FAST, FLEX_DISABLE, status);
```

Verilog

```
// Verilog syntax - turn off all setup timing checks
pcislave_set_timing_control(Id_1, `PCISLAVE_SETUP,
    `FLEX_DISABLE, status);

// Turn off the hold check from CLKOUT(1h) to INT4(ha)
pcislave_set_timing_control(Id_1,
    `PCISLAVE_TH_PCLK_LH_PIRDYNN_VV_P66MHZ_FAST, `FLEX_DISABLE, status);
```

Vera

```
//Vera example.
pcislave1.set_timing_control(`PCISLAVE_TCHECKS, `FLEX_DISABLE, status);
```

7

Using the pcimonitor_fx

Introduction

The pcimonitor_fx is a software bus analyzer you use in conjunction with the pcimaster_fx and pcislave_fx to verify your PCI or PCI-X design. The pcimonitor_fx model performs the following:

- Checks for illegal signal changes to verify that PCI and PCI-X cycles comply with the PCI and PCI-X specifications.
- Generates a log of bus activity in the pcimonitor_tst.lst file.
- Arbitrates the PCI or PCI-X bus.



Note

In conventional PCI mode, the pcislave_fx is backward-compatible with previous versions of the model, with a few exceptions. For more information, see [Appendix B on page 367](#).

pcimonitor_fx Supported Functions

The pcimonitor_fx FlexModel supports the following functions:

- **PCI-X mode.** See [“Using the pcimonitor_fx in PCI-X Mode” on page 288](#).
- **HDL, C, and Vera command modes.** See [“Command Modes,”](#) in the *FlexModel User's Manual*.
- **Protocol tracking.** Checks for illegal signal changes. See [“PCI and PCI-X Error Checks” on page 289](#).

- **Bus tracking.** Provides a text listing of bus activity. See [“Generating the pcimonitor Trace File” on page 61](#).
- **Arbitration.** Arbitrates the PCI and PCI-X buses.

pcimonitor_fx Modeling Exceptions and Unsupported Features

There are no modeling specific exceptions for the pcimonitor_fx.

None of the PCI models support save and restore (restart) capabilities. No flexmodels support these features.

Using the pcimonitor_fx in PCI-X Mode

In PCI-X mode, the model is compatible with the PCI-X Addendum to the PCI Specification.

PCI-X mode is turned off by default. To turn PCI-X mode on, use the following command immediately after using the flex_get_inst_handle command:

```
pcimonitor_configure(PCIMONITOR_PCIX, FLEX_TRUE, status);
```

To turn PCI-X mode off, use the following command:

```
pcimonitor_configure(PCIMONITOR_PCIX, FLEX_FALSE, status);
```

You can use either 66MHz or 133MHz timing in PCI-X mode. In PCI-X mode, you need to set the TimingVersion generic/defparameter to “pcix”. When TimingVersion is set to “pcix”, the model uses 66MHZ timing when the P66MHZ pin is set to 0, and 133MHZ timing when the P66MHZ pin is set to 1. The default is 0.



Note

In PCI-X mode, the P66MHZ pin can be considered as either a slow or fast pin. When set to true (1), the model uses a faster frequency (133MHz). When set to false (0), the model uses a slower frequency (66MHz). The name P66MHZ was retained for backward-compatibility, and does not mean that the model is operating at 66MHz.

PCI and PCI-X Error Checks

The pcimonitor_fx model checks for and reports both PCI and PCI-X errors. You can use the pcimonitor's error registers to determine which errors are present or to enable or disable any of the individual error checks listed. For more information on using the error registers, see [“Using Error Check Registers” on page 65](#).

Note the following about certain error messages:

- If a Split Completion returns a Retry and the device has reached its Retry limit, then the pcimonitor_fx model does not keep track of that Split Completion any more. In this case, the monitor would return an error message because that Split Completion would not have been dequeued completely.
- PCIX Error 98, 99, and 100 should be used only under the following conditions:
 - There is no pull up on the PERRNN pin.
 - No uninitialized memory. For example, memory values are not 32'h12xxxxx, but rather 32'h12345678.
- You can use the pcimonitor_fx to help check for parity errors. There are a number of PCIX error checks (96, 97, 98, 99, 100) which check for errors involving PERR# and SERR#. By default those error checks are turned off.
- PCIX Error 95 is turned off by default. You can turn on the messages by using the ERROR_ENABLE registers. “PCIX Error 95 states the following: “Byte enables are deasserted for bytes before the starting address and after the ending address. (1.10.1)”

PCI Error Checks

The following error checks and reports are based on criteria in the PCI specification. The number of the PCI specification section that addresses the error is shown in parentheses.

1. Only a single GNT# may be asserted on any clock (3.4)
2. AD2 must be 0 during the address phase of 64-bit transaction (3.10)
3. Only memory commands make sense when doing 64-bit transfers (3.10)
4. STOP# must be deasserted the cycle immediately after FRAME# is deasserted (3.3.3.2.1)
5. TRDY# must be deasserted during turnaround cycle on a read (3.3.1)
6. Target must issue DEVSEL# before any other response (3.7.1)
7. FRAME# cannot be deasserted before IRDY# is asserted (3.3.3.1)

8. Illegal combination of AD1, AD0, and C/BE# for I/O cycle (3.2.2)
9. Targets must not respond to reserved encodings (3.2.2)
10. Targets must not respond to the special cycle (3.7.2)
11. Targets must not respond to configuration cycles unless selected (3.2.2)
12. Once FRAME# is deasserted it cannot be reasserted during the same transaction (3.3.3.1)
13. Once a master has asserted IRDY# it cannot change FRAME# until the current data phase completes (3.2.1)
14. Once a master has asserted IRDY# it cannot change IRDY# until the current data phase completes (3.2.1)
15. Once a target has asserted IRDY# it cannot change IRDY# until the current data phase completes (3.2.1)
16. Once a target has asserted TRDY# it cannot change DEVSEL# until the current data phase completes (3.2.1)
17. Once a target has asserted TRDY# it cannot change STOP# until the current data phase completes (3.2.1) *Currently Not Implemented.*
18. DEVSEL# must be asserted for one or more clocks before target-abort can be signaled (3.3.3.2)
19. TRDY# must be deasserted before target-abort can be signaled (3.3.3.2.1)
20. Once a target has asserted STOP# it cannot change TRDY# until the current data phase completes (3.2.1) *Currently Not Implemented.*
21. Once a target has asserted STOP# it cannot change DEVSEL# until the current data phase completes (3.2.1)
22. Once a target has asserted STOP# it cannot change STOP# until the current data phase completes (3.2.1)
23. Once asserted, STOP# must remain asserted until FRAME# is deasserted (3.3.3.2.1)
24. IRDY# must always be asserted on the first clock edge that FRAME# is deasserted (3.3.3.1)
25. If DEVSEL# is asserted, master abort termination is not allowed (3.3.3.1)
26. Once a target has asserted DEVSEL# it must not be deasserted until the last data phase has completed (3.7.1)
27. The C/BE# output buffers must remain enabled until the end of the transaction (3.3.1)

28. The C/BE# must be valid during data phase (3.3.1)
29. The AD output buffers must remain enabled until the end of the transaction (3.3.1)
30. Dual command (1101) is not valid after a dual command: cycle ignored (not in specification)
31. FRAME# must remain asserted for the cycle following dual command: cycle ignored (3.10.1)
32. DEVSEL# asserted during address phase (3.7.1)
33. DEVSEL# negated before cycle completion: ignored (3.7.1)
34. DEVSEL# unknown during cycle: ignored (3.7.1)
35. Parity must be driven within one PCI clock of C/BE# and AD being driven (3.7.1)
36. C/BE# and AD must be set to the high-impedance state within one PCI clock after GNT# negation (3.4.3)
37. C/BE# and AD must be driven within eight PCI clocks of GNT# assertion (3.4.3)
38. Parity error detected (3.7.1)
39. Cycle must not start unless GNT# is asserted (3.4.1)
40. One clock delay required between grants (GNT#) during an IDLE cycle (3.4.1)
41. Master must deassert REQ# for a minimum of two PCI clocks after target abort (3.3.3.2.2)
42. An agent must never use REQ# to park itself on the bus (3.4.1)
43. Problem with current master, GNT# asserted more than 16 clocks with no access (3.4.1)
44. TRDY# must be asserted within eight clocks on all data phases (3.5.2)
45. Linear burst ordering must be used for memory write invalidate cycles (3.2.1)
46. All byte enables should be asserted during each data phase of a memory write invalidate cycle (3.2.1)
47. TRDY# or STOP# should be asserted within 16 clocks of an initial data phase (3.5.1.1)
48. Reserved.
49. TRDY# must be deasserted for the cycle immediately following the completion of the last data phase (3.3.3.2.1), or within 50ns of reset for pcix init.
50. DEVSEL# must be deasserted for the cycle immediately following the completion of the last data phase (3.3.3.2.1), or within 50ns of reset for pcix init.

51. Frame must be deasserted whenever STOP# is asserted (that is, as soon as IRDY# can be asserted) (3.3.3.2.1)
52. TRDY#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
53. IRDY#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
54. FRAME#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
55. DEVSEL#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
56. STOP#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
57. LOCK#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
58. PERR#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
59. REQ64#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
60. ACK64#, a sustained tri-state signal must be driven high for one clock before being tri-stated (2.1)
61. Master never uses reserved burst ordering ($AD(1:0) = 01$) (3.2.2)
62. Master never uses reserved burst ordering ($AD(1:0) = 11$) (3.2.2)
63. Master never signals abort earlier than five clock cycles after FRAME# is asserted
64. STOP# must be deasserted during the cycle immediately following the completion of the last data phase (3.3.3.2.1), or within 50ns of reset for pcix init.
65. Master must begin a lock operation with a read transaction (3.6)
66. Master must release LOCK# when access is terminated by a target-abort or master-abort (3.6)
67. Master never generates a DUAL cycle when the upper 32-bits of address are zero (3.10.1)
68. Target always disconnects after the first data phase when reserved burst mode is detected (3.2.2)
69. AD must be driven to stable values during every address and data phase through the end of the transaction (3.2.4)

70. PERR# is never asserted until a cycle has been claimed and a data phase completed (2.2.5)
71. Target never deasserts STOP# and continues the transaction (3.3.3.2.1)
72. Master always deasserts LOCK# for minimum of one idle clock between consecutive lock operations (3.6)
73. Target always locks a minimum of 16 bytes (3.6)
74. TRDY# or STOP# should be asserted within eight clocks of assertion of TRDY# from the previous data phase (3.5.1.2)
75. IRDY#, TRDY#, DEVSEL#, STOP#, and ACK64# should not be asserted when FRAME# is asserted (3.2.4)
76. C/BE# should be valid from the first clock of the data phase and must not change until the data phase completes (3.3.1)
77. ACK64# must not be asserted unless REQ64# was sampled asserted during the same transaction (3.10)
78. The master must deassert IRDY# for the clock cycle following completion of the last data phase (3.3.3.2.1)
79. REQ64# must exactly mirror FRAME# (3.10)
80. ACK64# must exactly mirror DEVSEL# (3.10)
81. Reserved
82. Illegal combination of AD1 and AD0 during configuration cycles.
83. Addresses in I/O and Configuration Spaces are always 32-bit.
84. A Reserved bus is not specified and is ignored by the device receiving the bus. (2.4)

PCI-X Error Checks

The following error checks and reports are based on criteria in the PCI-X specification.

1. In the attribute phase AD[63:32] and C/BE#[7:4] must be reserved and driven high by the 64-bit initiators. (2.5)
2. The C/BE# bus must be reserved (driven high) the clock after the attribute phase. (2.6)
3. In DWORD transactions the C/BE# bus during the data phase must be reserved and driven high by the initiator. (2.7)
4. The target response phase must end when the target asserts DEVSEL#. (1.10.1)

5. The C/BE# bus is must be reserved and driven high throughout all burst transactions except Memory Write. (2.6.1)
6. For Memory Write transactions byte enables must be deasserted for bytes before the starting address and after the ending address. (1.10.1)
7. The initiator must assert FRAME# within two clocks after GNT# is asserted. (1.10.2)
8. For Configuration cycles the initiator must assert FRAME# within six clocks after GNT# is asserted. (1.10.2)
9. The initiator must deassert FRAME# one clock before the last data phase if the transaction has four or more data phases.(2.11.1)
10. The initiator must deassert FRAME# two clocks after the target asserts TRDY# if the transaction has less than four data phases. (2.11.1)
11. The initiator must deassert IRDY# one clock after the last data phase if the transaction has four or more data phases. (2.11.1)
12. The initiator must deassert IRDY# two clocks after the target asserts TRDY# if the transaction has less than four data phases.(2.11.1)
13. Initiator wait states are not permitted in PCIX transactions. (1.6)
14. The initiator must assert IRDY# two clocks after the attribute phase. (1.10.2)
15. Master never signals abort earlier than 7 clocks after FRAME# is asserted. (2.11.1.2)
16. For write and split completion transactions, the initiator must drive data on the AD bus two clocks after the attribute phase. (1.10.2)
17. The target is not permitted to signal wait state after the first data phase. (1.6)
18. If the target signals split response, target-abort or retry, the target must do so within 8 clocks of the assertion of FRAME#. (2.9.1)
19. If the target signals single data phase disconnect, data transfer or disconnect at next ADB, target-abort or retry, the target must do so within 16 clocks of the assertion of FRAME#. (2.9.1)
20. Once the target has signaled Disconnect at Next ADB, it must continue to do so until the end of the transaction. (2.11.2.2)
21. When the arbiter asserts GNT#, it must keep GNT# asserted for a minimum of five clocks while the bus is idle, or until the initiator asserts FRAME# or deasserts REQ#. (1.10.4)

22. If the arbiter deasserts GNT# to one device, it must wait until the next clock to assert GNT# to another device. (1.10.4)
23. All fast back-to-back transactions are not permitted in PCIX mode. (1.10.4)
24. If only one request is asserted, the arbiter keeps GNT# asserted to that device. (1.10.4)
25. Initiators must drive the address for four clocks before asserting FRAME# for configuration transactions.(1.10.5)
26. The target is permitted to signal single data phase disconnect only on the first data phase (with or without preceding wait states). (2.11.2.1)
27. DWORD transactions must be initiated as 32-bit transfers. (2.7)
28. Split completions contain either read data or a split completion message but not both.(1.10.8)
29. A Split Completion message is always a single data phase. (2.10.6)
30. The requester must never terminate a Split Completion transaction with Split Response, Single Data Phase disconnect. (1.10.8)
31. The target must assert devsel before signaling split response. (2.10.5)
32. Illegal combination of AD1, AD0 and C/BE# for I/O and DWORD Memory transactions. (2.3)
33. If the target signals data transfer, the target is limited to disconnecting the transaction only on an ADB. (1.10.3)
34. An initiator is permitted to start a new transaction on any clock N in which the initiator's GNT# was asserted on clock N-2. (4.1.1)
35. The initiator must deassert FRAME# and IRDY# two clocks after the transaction is terminated.(1.10.2)
36. Reserved.
37. The target should deassert TRDY# within two clocks after FRAME# is deasserted. (2.6.1)
38. The initiator cannot deassert IRDY# before TRDY# is deasserted. (2.6.1)
39. AD turn-around violation; AD bus must be floated after the attribute phase for read transactions. (2.6.1)
40. The master must deassert IRDY# at the same clock cycle when FRAME# is deasserted for less than three data phases. (2.11.1.1)

41. In a 64-bit data transfer, the upper 32 bits must be copied to the lower 32-bits. (2.12.3)
42. AD turn-around violation; AD bus must be floated at the end of the transactions.
43. The requester must never terminate a Split Completion transaction with Retry or Disconnect at Next ADB. (1.10.8)
44. If the target inserts wait states, the initiator must toggle between its first and second data values until the target asserts TRDY#. (2.9.2)
45. For burst write and Split Completion transactions, the target is permitted to insert wait states only in pairs of clocks. (2.9)
46. BCM bit should be 0 in Split Completion Message. (2.10.4)
47. SCE/SCM bit should never be 10, this combination is reserved. (2.10.4)
48. STOP# and DEVSEL# cannot assert at the same time.
49. Disconnect at ADB was signalled at four or more dataphases from ADB, but the transaction did not stop at that ADB. (2.11.2.2)
50. In the second address phase of Dual Address Cycle, AD[63:32] and AD[31:0] should contain duplicate copies of the upper half of the address. (2.12.1)
51. In the second address phase of Dual Address Cycle, C/BE#[7:4] and C/BE#[3:0] should contain duplicate copies of the transaction command. (2.12.1)
52. In the first address phase of Dual Address Cycle, C/BE#[7:4] should contain the transaction command. (2.12.1)
53. The starting address of Configuration Read and Configuration Write transactions should be aligned to a DWORD boundary. (2.7.2)
54. A Reserved bus is not specified and is ignored by the device receiving the bus. (2.12)
55. If the target signals Disconnect At ADB less than four dataphases from an ADB, then transaction continues the first ADB and disconnect on the next ADB or until the byte count expires. (2.11.2.2)
56. The target signals Split Response and Retry only on the first data phase. (1.10.2)
57. DEVSEL# is never asserted earlier than the clock after the attribute phase. (1.10.1)
58. Special Cycle transactions have no initiator wait states and have only one data phase. (2.7.3)
59. The device does not set the No Snoop and Relaxed Order attribute bits on any transaction that is not a memory transaction. (2.5)

60. Burst transactions do not cross the end of the 64-bit address space.(1.10.2)
61. If the target signals Disconnect at Next ADB four data phases before an ADB or on the first data phase, the initiator deasserts FRAME# two clocks later and disconnects the transaction on the ADB (2.11.2.2)
62. TRDY# and DEVSEL# cannot assert at the same time.
63. The target of a write or Split Completion transaction should not check parity while it is inserting initial wait states. (5.1)
64. Split Completions should always have a single address phase both for 64-bit and 32-bit initiators. (2.10.2)
65. Only burst transactions (Split Completions or memory commands other than Memory Read DWORD) should use 64-bit data (1.10.1)
66. If the completer signals Split Response, it must initiate one or more Split Completion transactions with that Requester ID and Tag. (2.10.2)
67. If the completer disconnects the Split Completion on the first ADB by adjusting the byte count, the completer must set the Byte Count Modified attribute bit for the affected transactions. (2.10.4)
68. Each time the Split Completion resumes after a disconnection at an ADB, the initiator should adjust the byte count (and starting address) to indicate the number of bytes remaining in the Sequence. (2.10.2)
69. If the completer returns read data, the Completer should return all the data (the full byte count) unless an error occurs.
70. Unexpected Split Completion: Previously aborted or Split Completion with incorrect address or byte count.
71. The initiator does not initiate a new Sequence using the same Tag until the previous Sequence is complete.
72. The target never signals Split Response on burst write or Split Transactions.(2.10.1)
73. The target must not assert DEVSEL# for any transactions using a reserved command.
74. I/O Read, I/O Write, Configuration Read, Configuration Write, Interrupt Acknowledge, Special Cycle, and Memory Read DWORD commands and Split Completion Messages are limited to a single data phase.
75. For 64-bit devices, AD[63:32] and C/BE[7:4] should be reserved during the address phase of transactions using single address cycles.

76. The Completer never uses a byte count in the Split Completion other than the full remaining byte count for the sequence, except to terminate the Split Completion on the first ADB of the sequence.
77. - 93. Reserved.
94. If GNT# is asserted and the bus is idle for four consecutive clocks, the device actively drives the bus (AD, C/BE#, PAR, and (if a 64-bit device) PAR64) no later than the sixth clock.
95. Byte enables are deasserted for bytes before the starting address and after the ending address. (1.10.1)
96. If a device detects error on an address phase, the device must assert SERR#. (1.10.6)
97. If a device detects error on an attribute phase, the device must assert SERR#. (1.10.6)
98. PERR# should be asserted on the second clock after PAR64 and PAR are driven. (5.3)
99. PERR#, a Sustained Tri-State signal must be driven high for 1 clock before being Tri-States (5.3)
100. PERR#, a Sustained Tri-State signal must be Tri-States after being driven high. (5.3)

PCI Warning Messages

1. Current cycle is not the same as previous cycle terminated by retry (3.3.3.2.2).

Common PCI and PCIX Error Messages

The following PCI errors are also applicable to PCIX: 4, 5, 6, 9, 10, 11, 31, 32, 35, 38, 49, 50, 52, 53, 54, 55, 56, 57, 58, 59, 60, 64, 65, 66, 67, 69, 72, 73, 75, 77, 79, 80, 83. For these PCI errors the protocol checks are also performed in PCIX mode. These errors are reported as “PCI/X Error” in PCIX mode. In PCI mode they are reported as “PCI Error.” In both modes these errors can be controlled by the PCIMONITOR_PCI_ERROR_ENABLE_REG or the PCIMONITOR_ERROR_ENABLE_REG register ([“Using Error Check Registers” on page 65](#)).

Note, the reported “number” of the protocol checks are not the same between modes. For example, PCI/X Error 4 is not the same as PCIX Error 4. The slash indicates that it is a common protocol check with PCI.

pcimonitor_fx Command Summary

Table 48 lists the commands that can be issued by the pcimonitor_fx model. To see a detailed description of the command, including syntax, usage description, parameters, and examples, click on the command name.

Table 48: pcimonitor_fx Command Summary

Command Name	Description
pcimonitor_configure	Modifies the model's operating conditions.
pcimonitor_output_enable	Enables or disables output for a specified bidirectional or output pin or bus.
pcimonitor_pin_req	Requests the model to read the value of a specified pin or bus.
pcimonitor_pin_rslt	Returns the value of a specified pin or bus.
pcimonitor_reg_req	Requests the model to read the value of a specified register.
pcimonitor_reg_rslt	Returns the value of a specified register.
pcimonitor_set_msg_level	Controls the severity level of the messages displayed by the model during simulation.
pcimonitor_set_pin	Sets the specified pin or bus to the supplied value.
pcimonitor_set_reg	Sets the specified register to the supplied value.
pcimonitor_set_timing_control	Enables or disables timing checks and access delays.
Global FLEX Commands (click on command name to jump to the command's description in the <i>FlexModel User's Manual</i>)	
flex_get_cmd_status	Checks for a valid command tag in the command queue.
flex_clear_queue	Clears command core queues for specified inst_handle.
flex_get_inst_handle	Provides a unique instance handle for a new instance.
flex_print_msg	Outputs the specified message to the screen.
flex_run_program	Switches the command source to a compiled C program.
flex_synchronize	Synchronizes the model instance with the number of instances specified.

pcislave_fx Command Reference

This section contains command reference pages for all commands used by the pcislave_fx FlexModel. The reference pages provide descriptions for each command along with syntax, parameters, prototypes, and examples. The commands are listed in alphabetical order.

For information about the command naming structure used with FlexModels, see the *[FlexModel User's Manual](#)*.

pcimonitor_configure

Modifies the model's operating conditions.

Syntax

```
pcimonitor_configure(inst_handle, ctype, cvalue, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>ctype</i>	A constant that determines which of the model's operating conditions are affected by executing this command. Table 49 shows the legal values for <i>ctype</i> :
<i>cvalue</i>	An integer that specifies the new value of the operating condition specified by the <i>ctype</i> parameter. Table 49 shows the legal values for <i>cvalue</i> .
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .



Note

When using this command in a C control command or VHDL testbench, you must use a full 64-bit vector to specify the *cvalue*. Note, in your C Testbench, you may use the defined types FLEX_TRUE_C64 and FLEX_FALSE_C64 instead of FLEX_TRUE and FLEX_FALSE. Otherwise, you will probably receive the following warning message: “warning:improper pointer/integer combination.”

Table 49: *ctype* and *cvalue* Values Used in the pcimonitor_configure Command

<i>ctype</i> Parameter Values	Description	<i>cvalue</i> Parameter Values
PCIMONITOR_PCIX	FLEX_TRUE or FLEX_FALSE Default = FLEX_FALSE	Enables/disables PCI-X mode. FLEX_TRUE turns PCI-X mode on; FLEX_FALSE turns it off.
PCIMONITOR_ARBITRATE	With <i>cvalue</i> set to FLEX_TRUE, causes pcislave_fx to arbitrate for your circuit.	FLEX_TRUE or FLEX_FALSE
PCIMONITOR_ERRORCHECK	With <i>cvalue</i> set to FLEX_TRUE, turns on error and warning checking for PCI and PCIX cycles.	FLEX_TRUE or FLEX_FALSE
PCIMONITOR_GNT_ASSERTED_CLKS	Lets you specify correct delays between GNT# and FRAME# assertion when using pcislave_fx to arbitrate the bus. Default for <i>cvalue</i> is 1.	Positive integer. Note this <i>ctype</i> is only used in the PCI Test Scenario 1.14. If you have the PCI Test Suite Manual, consult the section Changing the pcimonitor GNT_ASSERTED_CLKS .
PCIMONITOR_PARKZERO	When <i>cvalue</i> is true, allows device connected to GNT0 to receive GNT# when no other devices are requesting the bus (when the pcimaster connected to GNT# is parked on the bus). When <i>cvalue</i> is false, pcimonitor does not park device connected to GNT0 and instead drives the AD, PAR, and C/BE# lines, if no other device owns the bus to stabilize the signal.	FLEX_TRUE or FLEX_FALSE
PCIMONITOR_PRIORITY_N	Lets you specify an arbitration scheme using the <i>cvalue</i> parameter. See Table 50 for a list of arbitration schemes.	Integer. See Table 50 for a list of possible values.

Table 49: *ctype* and *cvalue* Values Used in the pcimonitor_configure Command (cont.)

<i>ctype</i> Parameter Values	Description	<i>cvalue</i> Parameter Values
PCIMONITOR_SPLIT_COMP_DELAY_CHECK	Checks for Split Completions. A value of zero causes the monitor to immediately begin checking for Split Completions. A value greater than zero sets the number of clock cycles when the monitor will check the queue to determine if all the Split Completions have completed.	Integer.
PCIMONITOR_VALUETRACEFILE	With <i>cvalue</i> set to FLEX_TRUE, logs bus activity to trace file whose name is specified in the FlexModelId generic/defparameter. See “Generating the pcimonitor Trace File” on page 61 for details.	FLEX_TRUE or FLEX_FALSE

Table 50: Arbitration Schemes for PCIMONITOR_PRIORITY_N Setting

<i>cvalue</i> Parameter Value	Arbitration Scheme	Example
0	FIXED—req(0) always highest	req(0),req(1),req(2),req(3)
1	ROTATING—priority order rotates after each arbitration	req(0),req(1),req(2),req(3) req(1),req(2),req(3),req(0) req(2),req(3),req(0),req(1) req(3),req(0),req(1),req(2)
2	REVERSING—priority order reverses after each arbitration	req(0),req(1),req(2),req(3) req(3),req(2),req(1),req(0) req(0),req(1),req(2),req(3)
3	For PCI Test Suite Test 1.15 gnt(2) is high for 10 clocks, then low for 10 clocks, then repeats	N/A
4	FIXED (for PCI Test Suite 1.14) req(0) always the highest except gnt(2) only asserted for 1 clock.	req(0),req(1),req(2),req(3)

Table 50: Arbitration Schemes for PCIMONITOR_PRIORITY_N Setting

<i>cvalue</i> Parameter Value	Arbitration Scheme	Example
Other	Other numbers default to FIXED	

Description

The `pcimonitor_configure` command lets you modify the model's operating conditions. For a complete list of the operating conditions you can change, see [Table 49 on page 302](#).

Prototypes

C

```
void pcimonitor_configure (
    const int      inst_handle,
    const int      ctype,
    const int      cvalue,
    int            *status);
```

VHDL

```
procedure pcimonitor_configure (
    inst_handle      : in integer;
    ctype           : in integer;
    cvalue          : in integer;
    status          : out integer);
```

Verilog

```
task pcimonitor_configure;
    input      [31:0] inst_handle;
    input      [31:0] ctype;
    input      [31:0] cvalue;
    output     [31:0] status;
```

Vera

```
ModelObject.configure(
    integer      ctype,
    integer      cvalue,
    var integer  status);
```


Examples

C

```
/* C syntax */
pcimonitor_configure(Id_1, PCIMONITOR_ARBITRATE, 1, &status);

pcimonitor_configure(Id_1, PCIMONITOR_PRIORITY_N, 0, &status);

pcimonitor_configure(Id_1, PCIMONITOR_PARKZERO, 0, &status);

pcimonitor_configure(Id_1, PCIMONITOR_VALUETRACEFILE, 0, &status);

pcimonitor_configure(Id_1, PCIMONITOR_ERRORCHECK, 1, &status);
```

VHDL

```
-- VHDL syntax
pcimonitor_configure(Id_1, PCIMONITOR_ARBITRATE, 1, status);

pcimonitor_configure(Id_1, PCIMONITOR_PRIORITY_N, 0, status);

pcimonitor_configure(Id_1, PCIMONITOR_PARKZERO, 0, status);

pcimonitor_configure(Id_1, PCIMONITOR_VALUETRACEFILE, 0, status);

pcimonitor_configure(Id_1, PCIMONITOR_ERRORCHECK, 1, status);
```

Verilog

```
// Verilog syntax
pcimonitor_configure(Id_1, `PCIMONITOR_ARBITRATE, `FLEX_TRUE, status);

pcimonitor_configure(Id_1, `PCIMONITOR_PRIORITY_N, 0, status);

pcimonitor_configure(Id_1, `PCIMONITOR_PARKZERO, `FLEX_FALSE, status);

pcimonitor_configure(Id_1, `PCIMONITOR_VALUETRACEFILE, `FLEX_FALSE,
    status);

pcimonitor_configure(Id_1, `PCIMONITOR_ERRORCHECK, `FLEX_TRUE, status);
```

Vera

```
//Vera syntax.  
pcimonitor1.configure(`PCIMONITOR_ARBITRATE, `TRUE, status);  
  
pcimonitor1.configure(`PCIMONITOR_PRIORITY_N, 0, status);  
  
pcimonitor1.configure(`PCIMONITOR_PARKZERO, `FALSE, status);  
  
pcimonitor1.configure(`PCIMONITOR_ERRORCHECK, `TRUE, status);  
  
pcimonitor1.configure(`PCIMONITOR_VALUETRACEFILE, `TRUE, status);  
  
pcimonitor1.configure(`PCIMONITOR_PCIX, `TRUE, status);  
  
pcimonitor1.set_timing_control(`PCIMONITOR_TCHECKS, `FLEX_DISABLE,  
    status);
```

pcimonitor_output_enable

Enables or disables output for a specified bidirectional or output pin or bus.

Syntax

pcimonitor_output_enable(*inst_handle*, *pin_name*, *enable_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	The name of the pin or bus for which output is to be enabled or disabled. Table 51 lists the legal pin and bus names for this parameter.

Table 51: *pin_name* Values in pcimonitor_output_enable Command

Device Pin Name	<i>pin_name</i> Parameter Value
PAR	PCIMONITOR_PAR_PIN
AD (all 64 bits)	PCIMONITOR_AD_BUS
C/BE# (all 8 bits)	PCIMONITOR_CXBENN_BUS
GNT# (all 8 bits)	PCIMONITOR_GNTNN_BUS

<i>enable_value</i>	A value of 1 or FLEX_ENABLE enables the specified output pin and a value of 0 or FLEX_DISABLE disables the specified output pin. FLEX_ENABLE and FLEX_DISABLE are predefined constants.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The `pcimonitor_output_enable` command lets you enable or disable the output for a specified pin. This command is used to specify pins that are not directly controlled by the `pcislave_fx` model. Outputs are considered to be bidirectional. In normal operating mode, an output is enabled and could be either 1 or 0. If you use this command to disable a pin, it indicates a Z state and you can drive it from another source.

Prototypes

C

```
void pcimonitor_output_enable (
    const int      inst_handle,
    const int      pin_name,
    const int      enable_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_output_enable (
    inst_handle      : in integer;
    pin_name         : in integer;
    enable_value     : in integer;
    status           : out integer);
```

Verilog

```
task pcimonitor_output_enable;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      [31:0] enable_value;
    output     [31:0] status;
```

Vera

```
ModelObject.output_enable(
    integer      pin_name,
    integer      enable_value,
    var integer  status);
```

Examples

The following examples disable (set to high-Z state) the PREQ output pin.

C

```
/* C syntax */
pcimonitor_output_enable(Inst_1, PCIMONITOR_PREQNN_PIN, FLEX_DISABLE,
    &status);
```

VHDL

```
-- VHDL syntax
pcimonitor_output_enable(Inst_1, PCIMONITOR_PREQNN_PIN, FLEX_DISABLE,
    status);
```

Verilog

```
// Verilog syntax
pcimonitor_output_enable(Inst_1, `PCIMONITOR_PREQNN_PIN, `FLEX_DISABLE,
    status);
```

Vera

```
// Vera syntax
pcimonitor1.output_enable(`PCIMONITOR_PREQNN_PIN, `FLEX_DISABLE, status);
```

pcimonitor_pin_req

Requests the model to retrieve the value of a specified pin or bus.

Syntax

```
pcimonitor_pin_req(inst_handle, pin_name, wait_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be retrieved. Table 52 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 52: *pin_name* Values in pcimonitor_pin_req Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD(<i>n</i>) (all 64 bits)	PCIMONITOR _{<i>n</i>} _PIN
C/BE(<i>n</i>) (all 8 bits)	PCIMONITOR_CXBENN _{<i>n</i>} _PIN
PAR	PCIMONITOR_PAR_PIN
GNT#(<i>n</i>) (all 8 bits)	PCIMONITOR_GNTNN _{<i>n</i>} _PIN
RST#	PCIMONITOR_RSTNN_PIN
FRAME#	PCIMONITOR_FRAMENN_PIN
TRDY#	PCIMONITOR_TRDYNN_PIN
IRDY#	PCIMONITOR_IRDYNN_PIN
STOP#	PCIMONITOR_STOPNN_PIN
DEVSEL#	PCIMONITOR_DEVSELNN_PIN
IDSEL(<i>n</i>) (all 8 bits)	PCIMONITOR_IDSEL _{<i>n</i>} _PIN
PERR#	PCIMONITOR_PERRNN_PIN
SERR#	PCIMONITOR_SERRNN_PIN
REQ#(<i>n</i>) (all 8 bits)	PCIMONITOR_REQNN _{<i>n</i>} _PIN

Table 52: *pin_name* Values in pcimonitor_pin_req Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
LOCK#	PCIMONITOR_LOCKNN_PIN
PAR64	PCIMONITOR_PAR64_PIN
REQ64#	PCIMONITOR_REQ64NN_PIN
ACK64#	PCIMONITOR_ACK64NN_PIN
SBO#	PCIMONITOR_SBONN_PIN
SDONE	PCIMONITOR_SDONE_PIN
INTA#	PCIMONITOR_INTANN_PIN
INTB#	PCIMONITOR_INTBNN_PIN
INTC#	PCIMONITOR_INTCNN_PIN
INTD#	PCIMONITOR_INTDNN_PIN
Disable message pin	PCIMONITOR_DISABLEMSG_PIN
Timing mode pin	PCIMONITOR_P66MHZ_PIN

wait_mode

One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the [FlexModel User's Manual](#) for additional information on using FLEX_WAIT.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimonitor_pin_req` command issues a request to the model to retrieve the value of a specified pin or bus. This command is the first half of a result command pair; the result is returned when the model executes the `pcimonitor_pin_rslt` command.

Prototypes

C

```
void pcimonitor_pin_req (
    const int      inst_handle,
    const int      pin_name,
    const int      wait_mode,
    int            *status);
```

VHDL

```
procedure pcimonitor_pin_req (
    inst_handle      : in integer;
    pin_name         : in integer;
    wait_mode        : in boolean;
    status           : out integer);
```

Verilog

```
task pcimonitor_pin_req;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      wait_mode;
    output     [31:0] status;
```

Vera

```
ModelObject.pin_req(
    integer      pin_name,
    integer      wait_mode,
    var integer  status);
```

Examples

C

```
/* C syntax */
pcimonitor_pin_req(Id_1, PCIMONITOR_TRDYN_PIN, FLEX_WAIT_F, &status);
```

VHDL

```
-- VHDL syntax - requests value of TrcEnd pin.
pcimonitor_pin_req(Id_1, PCIMONITOR_TRDYN_PIN, FLEX_WAIT_F, status);
```


Verilog

```
// Verilog syntax - requests 4-bits of TrcData bus.  
pcimonitor_pin_req(Id_1, `PCIMONITOR_AD_BUS, `FLEX_WAIT_F, status);
```

Vera

```
// Vera syntax - requests 4-bits of TrcData bus.  
pcimonitor1.pin_req(`PCIMONITOR_AD_BUS, `FLEX_WAIT_F, status);
```

pcimonitor_pin_rslt

Returns the value of a specified pin or bus.

Syntax

pcimonitor_pin_rslt(*inst_handle*, *pin_name*, *pin_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be returned. Table 53 the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 53: *pin_name* Values in pcimonitor_pin_rslt Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD(<i>n</i>) (all 64 bits)	PCIMONITOR_PADATAN_PIN
C/BE(<i>n</i>) (all 8 bits)	PCIMONITOR_PCXBENN_PIN
PAR	PCIMONITOR_PAR_PIN
GNT#(<i>n</i>) (all 8 bits)	PCIMONITOR_GNTNN_PIN
RST#	PCIMONITOR_RSTNN_PIN
FRAME#	PCIMONITOR_FRAMENN_PIN
TRDY#	PCIMONITOR_TRDYNN_PIN
IRDY#	PCIMONITOR_IRDYNN_PIN
STOP#	PCIMONITOR_STOPNN_PIN
DEVSEL#	PCIMONITOR_DEVSELNN_PIN
IDSEL(<i>n</i>) (all 8 bits)	PCIMONITOR_IDSELn_PIN
PERR#	PCIMONITOR_PERRNN_PIN
SERR#	PCIMONITOR_SERRNN_PIN
REQ#(<i>n</i>) (all 8 bits)	PCIMONITOR_REQNNn_PIN

Table 53: *pin_name* Values in pcimonitor_pin_rslt Command (cont.)

Device Pin Name	<i>pin_name</i> Parameter Value
LOCK#	PCIMONITOR_LOCKNN_PIN
PAR64	PCIMONITOR_PAR64_PIN
REQ64#	PCIMONITOR_REQ64NN_PIN
ACK64#	PCIMONITOR_ACK64NN_PIN
SBO#	PCIMONITOR_SBONN_PIN
SDONE	PCIMONITOR_SDONE_PIN
INTA#	PCIMONITOR_INTANN_PIN
INTB#	PCIMONITOR_INTBNN_PIN
INTC#	PCIMONITOR_INTCNN_PIN
INTD#	PCIMONITOR_INTDNN_PIN
Disable message pin	PCIMONITOR_DISABLEMSG_PIN
Timing mode pin	PCIMONITOR_P66MHZ_PIN
AD[63:0]	PCIMONITOR_AD_BUS
C/BE[7:0]	PCIMONITOR_CXBENN_BUS
IDSEL[7:0]	PCIMONITOR_IDSEL_BUS
REQ#[7:0]	PCIMONITOR_REQNN_BUS
GNT#[7:0]	PCIMONITOR_GNTNN_BUS

pin_value

The returned value of the pcimonitor_pin_req command.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The `pcimonitor_pin_rslt` command returns the value of the pin or bus specified in the corresponding `pcimonitor_pin_req` command. This command is the second half of the result command pair that begins with the `pcimonitor_pin_req` command. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or previous commands complete execution. If all previous commands have completed and the specified result is not available, the command completes with an error.

Prototypes

C

```
void pcimonitor_pin_rslt (
    const int      inst_handle,
    const int      pin_name,
    FLEX_VEC       pin_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_pin_rslt (
    inst_handle: in integer;
    pin_name   : in integer;
    pin_value: out std_logic_vect(PCIMONITOR_MAX_BUS_WIDTH-1 downto 0);
    status     : out integer);
```

Verilog

```
task pcimonitor_pin_rslt;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    output     [`PCIMONITOR_MAX_BUS_WIDTH:0] pin_value;
    output     [31:0] status;
```

Vera

```
ModelObject.pin_rslt(
    integer      pin_name,
    var bit [`PCIMONITOR_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);
```

Examples

The following command examples return the result of the previously issued `pcimonitor_pin_req` command. The `pin_value(0)` is the returned value of the LSB of the pin result.

C

```
/* C syntax */
pcimonitor_pin_rslt(Id_1, PCIMONITOR_TRDYNN_PIN, &pin_value, &status);
```

VHDL

```
-- VHDL syntax - returns the TrcEnd 1-bit value.
pcimonitor_pin_rslt(Id_1, PCIMONITOR_TRDYNN_PIN, pin_value, status);
```

Verilog

```
// Verilog syntax - returns 4 bits of TrcData.
pcimonitor_pin_rslt(Id_1, `PCIMONITOR_AD_BUS, pin_value, status);
```

Vera

```
// Vera syntax - returns 4 bits of TrcData.
pcimonitor1.pin_rslt(`PCIMONITOR_AD_BUS, pin_value, status);
```

pcimonitor_reg_req

Requests the model to retrieve data from the model registers.

Syntax

pcimonitor_reg_req (*inst_handle*, *reg_name*, *wait_mode*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register, or a specific bit in the register, whose data is to be retrieved. You can access all bits of the register by specifying the register name (for example, PCIMONITOR_ERROR_ENABLE_REG) or access an individual bit of a register by appending the bit position to the register name (PCIMONITOR_ERROR_ENABLE2_REG for bit 2 of the register). Table 54 lists the legal values of <i>reg_name</i> . There is no default.

Table 54: *reg_name* Values in pcimonitor_reg_req Command

<i>reg_name</i> Values	Data Type	Description
PCIMONITOR_ERROR_REG PCIMONITOR_ERROR n _REG	1-bit $n=127:0$ 128-bit vector	Records which errors are present. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved.
PCIMONITOR_ERROR_ENABLE_REG PCIMONITOR_ERROR_ENABLE n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual error checks. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved. All error checks are enabled (set to 1) by default.

<i>wait_mode</i>	One of two predefined constants: FLEX_WAIT_T or FLEX_WAIT_F. When true (FLEX_WAIT_T or 1), the model suspends command execution until this command completes. When false (FLEX_WAIT_F or 0), the model queues this command and continues executing the command sequence. Refer to the FlexModel User's Manual for additional information on using FLEX_WAIT.
------------------	--

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimonitor_reg_req command issues a request to the model to retrieve data from a specified register. This command is the first half of a result command pair; the result is returned when the model executes a pcimonitor_reg_rslt command. [Table 54 on page 318](#) lists the programmable registers of the pcislave_fx.

Prototypes

C

```
void pcimonitor_reg_req (
    const int      inst_handle,
    const int      reg_name,
    const int      wait_mode,
    int            *status);
```

VHDL

```
procedure pcimonitor_reg_req (
    inst_handle      : in integer,
    reg_name         : in integer,
    wait_mode        : in boolean,
    status           : out integer);
```

Verilog

```
task pcimonitor_reg_req;
    input    [31:0] inst_handle;
    input    [31:0] reg_name;
    input          wait_mode;
    output    [31:0] status;
```

Vera

```
ModelObject.reg_req (
    integer      reg_name,
    integer      wait_mode,
    var integer  status);
```

Examples

The following command requests the value of bit 1 of the ERROR register:

```
// Verilog Example
pcimonitor_reg_req(inst, `PCIMONITOR_ERROR1_REG, `FLEX_WAIT_F, status);
```

The result is held in the command core until the pcimonitor_reg_rslt command retrieves the result into the testbench.

--VHDL Example: The first command below requests the value of the entire ERROR register; the second command returns the value.

```
pcimonitor_reg_req (Id_1, pcimonitor_ERROR_REG, FLEX_WAIT_F, status);
pcimonitor_reg_rslt(Id_1, pcimonitor_ERROR_REG, Return_value, status);
assert (false) report "ERROR register " severity NOTE;
```

```
/* C Example */
/* Displays the value of the ERROR register*/
pcimonitor_reg_req (Id_1, pcimonitor_ERROR_REG, FLEX_WAIT_F, &status);
pcimonitor_reg_rslt(Id_1, pcimonitor_ERROR_REG, Return_value, &status);
flex_fprintf(stderr, " ERROR register = %H\n", Return_value);
```


pcimonitor_reg_rslt

Returns the data of a specified register.

Syntax

pcimonitor_reg_rslt (*inst_handle*, *reg_name*, *reg_value*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register, or a specific bit in the register, whose data is to be returned. You can access all bits of the register by specifying the register name (for example, PCIMONITOR_ERROR_ENABLE_REG) or access an individual bit of a register by appending the bit position to the register name (PCIMONITOR_ERROR_ENABLE2_REG for bit 2 of the register). Table 55 lists the legal values of <i>reg_name</i> . There is no default.

Table 55: *reg_name* Values in pcimonitor_reg_rslt Command

<i>reg_name</i> Values	Data Type	Description
PCIMONITOR_ERROR_REG PCIMONITOR_ERROR n _REG	1-bit $n=127:0$ 128-bit vector	Records which errors are present. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved.
PCIMONITOR_ERROR_ENABLE_REG PCIMONITOR_ERROR_ENABLE n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual error checks. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved. All error checks are enabled (set to 1) by default.

<i>reg_value</i>	A variable name that will hold the returned value of the register specified in the pcimonitor_reg_req command. The model will return a vector value with a width of the specified register. Note that you can specify the entire register (vector) or a single bit of the register (bit).
------------------	---

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimonitor_reg_rslt command returns the data from the register specified in the corresponding pcimonitor_reg_req command. This command is the second half of the result command pair that begins with the pcimonitor_reg_req command. If the data is available when this command is issued, the command returns in zero time. Otherwise, it waits until the data becomes available or previous commands complete execution. If all previous commands have completed and the specified result is not available, the command completes with an error. [Table 55 on page 321](#) lists the programmable registers of the pcislave_fx.

Prototypes

C

```
void pcimonitor_reg_rslt (
    const int      inst_handle,
    const int      reg_name,
    FLEX_VEC      reg_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_reg_rslt (
    inst_handle: in integer,
    reg_name   : in integer,
    reg_value  : out bit_vector(PCIMONITOR_MAX_REG_WIDTH-1 downto 0),
    status     : out integer);
```

Verilog

```
task pcimonitor_reg_rslt;
    input    [31:0] inst_handle;
    input    [127:0] reg_name;
    output   [`pcimonitor_MAX_REG_WIDTH-1:0] reg_value;
    output   [127:0] status;
```

Vera

```

ModelObject.reg_rslt (
    integer      reg_name,
    var bit      [`PCIMONITOR_MAX_REG_WIDTH-1:0] reg_value,
    var integer   status);

```

Examples

The following command returns the result of the pcimonitor_reg_req command, assigning the value of the ERROR register bit '1' to the 'reg_value' variable.

// Verilog Example

```
pcimonitor_reg_rslt(inst, `PCIMONITOR_ERROR1_REG, reg_value, status);
```

--VHDL Example: Displays the value of the ERROR register

```

pcimonitor_reg_req (Id_1, PCIMONITOR_ERROR_REG, FLEX_WAIT_F, status);
pcimonitor_reg_rslt(Id_1, PCIMONITOR_ERROR_REG, Return_value, status);
assert (false) report "ERROR register " severity NOTE;

```

/* C Example */

/* Display the value of the ERROR register*/

```

pcimonitor_reg_req (Id_1, PCIMONITOR_ERROR_REG, FLEX_WAIT_F, &status);
pcimonitor_reg_rslt(Id_1, PCIMONITOR_ERROR_REG, Return_value, &status);
flex_fprintf(stderr, " ERROR register = %H\n", Return_value);

```

// Vera Example

```
pcimonitor1.reg_rslt(`PCIMONITOR_ERROR1_REG, reg_value, status);
```

pcimonitor_set_msg_level

Controls the severity level of the messages displayed by the model during simulation.

Syntax

```
pcimonitor_set_msg_level(inst_handle, mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>mode</i>	A 32-bit integer that controls the level of messaging or a predefined constant that controls a single message type. Each bit in the <i>mode</i> field controls one message category; if a specific bit is set to 1, the messages for that category are enabled. Table 56 lists the individual bit assignments and the corresponding predefined constants.



Note

To get command-by-command trace information and other model messages, which can be useful for debugging, set all the upper bits in the *mode* parameter to 1, as follows:

```
pcimonitor_set_msg_level (inst_handle, 6FFFFFFF, status);
```

Table 56: *mode* Values in pcimonitor_set_msg_level Command

Bit Assignment	Predefined Constant	Description
None	FATAL	Stops simulation immediately after a fatal message is reported. Fatal messages are always enabled.
bit-30	PCIMONITOR_DEBUG	Displays debug messages, including command-by-command trace information.
0FFFFFFF	FLEX_ALL_MSGS	Turns on all message types except debug messages. For command-by-command trace information, use PCIMONITOR_DEBUG.
00000000	FLEX_NO_MSGS	Turns off all message types other than fatal.

Table 56: *mode* Values in pcimonitor_set_msg_level Command (cont.)

Bit Assignment	Predefined Constant	Description
10000000	PCIMONITOR_TESTSUITE_LST	Formats the trace file so that is compatible with the PCI Test Suite. You must use this message level if you are using the pcislave_fx with the PCI Test Suite.
bit-0	FLEX_ERROR	Displays error messages for situations in which the model can recover and resume simulation, such as when the model receives a command that would put it into an invalid state.
bit-1	FLEX_WARNING	Displays warning messages for situations that are not necessarily errors, such as when significant bits of an address are ignored.
bit-2	FLEX_TIMING	Displays timing messages.
bit-3	FLEX_XHANDLING	Displays X-handling messages.
bit-4	FLEX_INFO	Informs you of the status or behavior of the model.
bits 8–29	Undefined	Undefined.

status

A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimonitor_set_msg_level command controls the severity level of the messages displayed by the model during simulation. Model messages are grouped into categories. Except for fatal messages, which are always enabled, you can enable or disable any category for each model instance. This command allows you to change from the default message level for this model.

Prototypes

C

```
void pcimonitor_set_msg_level (
    const int      inst_handle,
    const int      mode,
    int            *status);
```

VHDL

```
procedure pcimonitor_set_msg_level (
    inst_handle      : in integer;
    mode             : in integer;
    status           : out integer);
```

Verilog

```
task pcimonitor_set_msg_level;
    input      [31:0] inst_handle;
    input      [31:0] mode;
    output     [31:0] status;
```

Vera

```
ModelObject.set_msg_level(
    integer      mode,
    var integer  status);
```

Examples

C

```
/* C syntax - enables INFO messages */
pcimonitor_set_msg_level(Id_1, FLEX_INFO, &status);

/* Enables all messages */
pcimonitor_set_msg_level(Id_1, "h0fffffff", &status);
```

VHDL

```
-- VHDL syntax - turns off all messages
pcimonitor_set_msg_level(Id_1, FLEX_NO_MSGS, status);

-- Enables INFO & WARNING messages
pcimonitor_set_msg_level(Id_1, FLEX_INFO + FLEX_WARNING, status);

-- Specifies message level that is compatible with PCI Test Suite
pcimonitor_set_msg_level(Id_1, PCIMONITOR_TESTSUITE_LST, status);
```

Verilog

```
// Verilog syntax - turns off all messages
pcimonitor_set_msg_level(Id_1, `FLEX_NO_MSGS, status);

// Enables INFO & WARNING messages
pcimonitor_set_msg_level(Id_1, `FLEX_INFO + `FLEX_WARNING, status);

// Enables all messages
pcimonitor_set_msg_level(Id_1, 32'h0fffffff, status);
```

Vera

```
//Vera example.
pcislave1.set_msg_level(`FLEX_NO_MSGS,status);
```

pcimonitor_set_pin

Set a specified pin or bus to the supplied value.

Syntax

```
pcimonitor_set_pin(inst_handle, pin_name, pin_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>pin_name</i>	A constant that indicates the name of the pin or bus whose value is to be set. Table 57 lists the legal values for <i>pin_name</i> and the signals to which each name applies. (Note that the <i>n</i> in the pin name represents the numeric position of the pin in the bus.)

Table 57: *pin_name* Values in pcimonitor_set_pin Command

Device Pin Name	<i>pin_name</i> Parameter Value
AD(<i>n</i>) (all 64 bits)	PCIMONITOR_PADATAN_PIN
C/BE(<i>n</i>) (all 8 bits)	PCIMONITOR_PCXBENN_PIN
PAR	PCIMONITOR_PPAR_PIN
GNT#(<i>n</i>) (all 8 bits)	PCIMONITOR_GNTNN_PIN
AD (all 64 bits)	PCIMONITOR_AD_BUS
C/BE (all 8 bits)	PCIMONITOR_CXBENN_BUS
GNT# (all 8 bits)	PCIMONITOR_GNTNN_BUS

<i>pin_value</i>	The level to which to set the pin. A value of 1 drives the pin high, a value of 0 drives the pin low, and anything else drives it unknown.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue

and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimonitor_set_pin command sets a specified pin or bus to a supplied value. A *pin_value* of 1 drives the pin high, a *pin_value* of 0 drives the pin low, and anything else drives it unknown. Not all pins are driven to *pin_value* at the same time. The model drives a pin to the supplied value at the time listed in the PCI specification.

Prototypes

C

```
void pcimonitor_set_pin (
    const int      inst_handle,
    const int      pin_name,
    const FLEX_VEC pin_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_set_pin (
    inst_handle      : in integer;
    pin_name         : in integer;
    pin_value        : in bit_vector;
    status           : out integer);
```

Verilog

```
task pcimonitor_set_pin;
    input      [31:0] inst_handle;
    input      [31:0] pin_name;
    input      [`PCIMONITOR_MAX_BUS_WIDTH:0] pin_value;
    output     [31:0] status;
```

Vera

```
ModelObject.set_pin(
    integer      pin_name,
    bit [`PCIMONITOR_MAX_BUS_WIDTH:0] pin_value,
    var integer  status);
```

Examples

The following command examples set the TRDY# pin to a value of 1.

C

```
/* C syntax */  
pcimonitor_set_pin(Id_1, PCIMONITOR_TRDYNN_PIN, "b1", &status);
```

VHDL

```
-- VHDL syntax  
pcimonitor_set_pin(Id_1, PCIMONITOR_TRDYNN_PIN, "1", status);
```

Verilog

```
// Verilog syntax  
pcimonitor_set_pin(Id_1, `PCIMONITOR_TRDYNN_PIN, 1b'1, status);
```

Vera

```
// Verilog syntax  
pcimonitor1.set_pin(`PCIMONITOR_TRDYNN_PIN, 1b'1, status);
```

pcimonitor_set_reg

Sets the value of a specified register.

Syntax

```
pcimonitor_set_reg (inst_handle, reg_name, reg_value, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>reg_name</i>	A constant that indicates the name of the register, or a specific bit in the register, whose data is to be set. You can access all bits of the register by specifying the register name (for example, PCIMONITOR_ERROR_ENABLE_REG) or access an individual bit of a register by appending the bit position to the register name (PCIMONITOR_ERROR_ENABLE2_REG for bit 2 of the register). Table 58 lists the legal values of <i>reg_name</i> . There is no default.

Table 58: *reg_name* Values in pcimonitor_set_reg Command

<i>reg_name</i> Values	Data Type	Description
PCIMONITOR_ERROR_REG PCIMONITOR_ERROR n _REG	1-bit $n=127:0$ 128-bit vector	Records which errors are present. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved.
PCIMONITOR_ERROR_ENABLE_REG PCIMONITOR_ERROR_ENABLE n _REG	1-bit $n=127:0$ 128-bit vector	Enables or disables individual error checks. Bit numbers correspond to the individual error checks as numbered in the list of PCI Error Checks (page 289) or PCI-X Error Checks (page 293) . Bit 0 is reserved. All error checks are enabled (set to 1) by default.

<i>reg_value</i>	A bit_vector with a width of the register specified in <i>reg_name</i> . This is the value to which the entire register or single bit of the register will be set. Note that you can set the value for either the entire register or a single bit of the register.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero

indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see [“The status Parameter”](#) in the *FlexModel User's Manual*.

Description

The pcimonitor_set_reg command sets the value of a specified register. [Table 58 on page 331](#) lists the programmable registers of the pcislave_fx. You can specify an entire register, or you can specify an individual bit of a register by appending the bit number to the register name. Each bit, starting from bit 1, corresponds to one of the numbered error checks in either the list of [PCI Error Checks \(page 289\)](#) or [PCI-X Error Checks \(page 293\)](#). Bit 0 is reserved.

Prototypes

C

```
void pcimonitor_set_reg (
    const int      inst_handle,
    const int      reg_name,
    const FLEX_VEC reg_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_set_reg (
    inst_handle      : in integer,
    reg_name         : in integer,
    reg_value        : in bit_vector,
    status           : out integer);
```

Verilog

```
task pcimonitor_set_reg;
    input  [31:0] inst_handle;
    input  [31:0] reg_name;
    input  [`PCIMONITOR_MAX_REG_WIDTH:0] reg_value;
    output [31:0] status;
```

Vera

```
ModelObject.set_reg (
    integer      reg_name,
    bit          [`PCIMONITOR_MAX_REG_WIDTH:0] reg_value,
    var integer  status);
```

Examples

The following command sets the entire ERROR_ENABLE register to a value of “1”.

```
//Verilog Syntax
pcimonitor_set_reg(Id_1, `PCIMONITOR_ERROR_ENABLE_REG,
128'hffffffffffffffffffffffffffffffff, status);
```

--VHDL Syntax

```
pcimonitor_set_reg(Id_1, PCIMONITOR_ERROR_ENABLE_REG, "1", status);
```

The following command sets bit 5 of the ERROR_ENABLE register to a value of “1”.

```
//Verilog Syntax
pcimonitor_set_reg(Id_1, `PCIMONITOR_ERROR_ENABLE5_REG, 1b'1, status);
```

--VHDL Syntax

```
pcimonitor_set_reg(Id_1, PCIMONITOR_ERROR_ENABLE5_REG, "1", status);
```

The following C example configures ERROR_ENABLE registers for interrupts.

```
/* C Example */
/* Testbench: Configure the ERROR_ENABLE register to recognize all
interrupts.
pcimonitor_set_reg(Id_1, PCIMONITOR_ERROR_ENABLE_REG,
"hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", &status);
```

//Vera Syntax

```
pcimonitor1.set_reg(`PCIMONITOR_ERROR_ENABLE5_REG, 1b'1, status);
```

pcimonitor_set_timing_control

Enables and disables timing checks and access delays. You can change group or individual timing checks.

Syntax

```
pcimonitor_set_timing_control(inst_handle, timing_control_reg, enable_value,  
                                status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the <code>flex_get_inst_handle</code> command.
<i>timing_param_index</i>	A constant that indicates the name of the timing check to be enabled or disabled. Table 63 on page 357 lists the group constant values. Table 64 on page 358 lists the individual constant values.
<i>enable_value</i>	One of two constants that specifies whether to enable (FLEX_ENABLE) or disable (FLEX_DISABLE) the specified timing check.
<i>status</i>	A variable name that will hold the returned status value (integer) after this command completes. If the returned integer is positive, the command was received successfully. A zero indicates the command was not received successfully, and a negative integer provides error information. An integer greater than 1 represents the command's order in the command queue and provides a command tag you can use to access command results. For information on command tags and error codes, see “The status Parameter” in the <i>FlexModel User's Manual</i> .

Description

The `pcimonitor_set_timing_control` command lets you enable or disable a specified timing check or a group of timing checks at runtime. You can also use this command to control individual pin-to-pin timing checks for existing paths in the timing model. The model defaults to all timing checks enabled.

Prototypes

C

```
void pcimonitor_set_timing_control (
    const int      inst_handle,
    const int      timing_param_index,
    const int      enable_value,
    int            *status);
```

VHDL

```
procedure pcimonitor_set_timing_control (
    inst_handle      : in integer;
    timing_param_index : in integer;
    enable_value     : in integer;
    status           : out integer);
```

Verilog

```
task pcimonitor_set_timing_control;
    input      [31:0] inst_handle;
    input      [31:0] timing_param_index;
    input      [31:0] enable_value;
    output     [31:0] status;
```

Vera

```
ModelObject.set_timing_control (
    integer      timing_param_index,
    integer      enable_value,
    var integer  status);
```

Examples

C

```
/* C syntax - turn off all setup timing checks */
pcimonitor_set_timing_control(Id_1, PCIMONITOR_SETUP,
    FLEX_DISABLE, &status);

/* Turn off the hold check from CLKOUT(1h) to INT4(ha) */
pcimonitor_set_timing_control(Id_1,
    PCIMONITOR_TH_CLK_LH_STOPNN_VV_P66MHZ_FAST, FLEX_DISABLE, &status);
```

VHDL

```
-- VHDL syntax - turn off all setup timing checks
pcimonitor_set_timing_control(Id_1, PCIMONITOR_SETUP, FLEX_DISABLE,
    status);

-- Turn off the hold check from CLKOUT(1h) to INT4(ha)
pcimonitor_set_timing_control(Id_1,
    PCIMONITOR_TH_CLK_LH_STOPNN_VV_P66MHZ_FAST, FLEX_DISABLE, status);
```

Verilog

```
// Verilog syntax - turn off all setup timing checks
pcimonitor_set_timing_control(Id_1, `PCIMONITOR_SETUP,
    `FLEX_DISABLE, status);

// Turn off the hold check from CLKOUT(1h) to INT4(ha)
pcimonitor_set_timing_control(Id_1,
    `PCIMONITOR_TH_CLK_LH_STOPNN_VV_P66MHZ_FAST, `FLEX_DISABLE, status);
```

Vera

```
//Vera example.
pcimonitor1.set_timing_control(`PCIMONITOR_TCHECKS, `FLEX_DISABLE,
    status);
```

A

Values for *timing_param_index* Parameter

Introduction

The following tables list possible values for the *timing_param_index* parameter in the *model_set_timing_control* commands.

For more information about these commands, see:

- [“pcimaster_set_timing_control” on page 200](#)
- [“pcislave_set_timing_control” on page 284](#)
- [“pcimonitor_set_timing_control” on page 334](#).

pcimaster_fx Parameter Values

[Table 59](#) lists group constant values for the *timing_param_index* parameter in the *pcimaster_set_timing_control* command. [Table 60 on page 338](#) lists individual constant values.

Table 59: Group Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command

Constant Name	Description
PCIMASTER_ALL_TIMING	All timing checks and access delays
PCIMASTER_TCHECKS	All timing checks
PCIMASTER_HOLD	All hold timing checks
PCIMASTER_PERIOD	All period checks

Table 59: Group Constant Values for *timing_param_index* Parameter in pcimaster_set_timing_control Command (cont.)

Constant Name	Description
PCIMASTER_PERMIN	Only minimum period checks
PCIMASTER_PERMAX	Only maximum period checks
PCIMASTER_PWIDTH	All pulse width checks
PCIMASTER_PWLMIN	Pulse width low minimum checks
PCIMASTER_PWHMIN	Pulse width high minimum checks
PCIMASTER_SETUP	All setup checks
PCIMASTER_NO_TIMING	No timing checks or access delays

Table 60: Individual Constant Values for *timing_param_index* Parameter in pcimaster_set_timing_control Command

Constant Name	Description
PERIOD MINIMUM	
PCIMASTER_PERMIN_PCLK_P66MHZ_FAST	Minimum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_PERMIN_PCLK_P66MHZ_SLOW	Minimum period of SYSCLOCK in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PERIOD MAXIMUM	
PCIMASTER_PERMAX_PCLK_P66MHZ_FAST	Maximum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PULSE WIDTH HIGH MINIMUM	
PCIMASTER_PWHMIN_PCLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK high in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_PWHMIN_PCLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK high in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PULSE WIDTH LOW MINIMUM	
PCIMASTER_PWLMIN_PCLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK low in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_PWLMIN_PCLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK low in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_PWLMIN_PRSTNN_P66MHZ_FAST	Minimum pulse duration of RESET low in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_PWLMIN_PRSTNN_P66MHZ_SLOW	Minimum pulse duration of RESET low in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
HOLD CONSTANTS	
PCIMASTER_TH_PCLK_LH_PACK64NN_VV_PACK64NNFAST_MODEON	Hold time from positive edge of PCLK to PACK64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PACK64NN_VV_PACK64NNSLOW_MODEON	Hold time from positive edge of PCLK to PACK64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PADATA_VV_PADATAFAST_MODEON	Hold time from positive edge of PCLK to PADATA valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PADATA_VV_PADATASLOW_MODEON	Hold time from positive edge of PCLK to PADATA valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PBENN_VV_PBENNFAST_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PBENN_VV_PBENNNSLOW_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TH_PCLK_LH_PCLKRUNNN_VV_PCLKRUNNNFAST_MODEON	Hold time from positive edge of PCLK to PCLKRUNNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PCLKRUNNN_VV_PCLKRUNNNSLOW_MODEON	Hold time from positive edge of PCLK to PCLKRUNNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PCXBENN_VV_PCXBENNFAST_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PCXBENN_VV_PCXBENNNSLOW_MODEON	Hold time from positive edge of PCLK to PCXBENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PD_VV_PDFAST_MODEON	Hold time from positive edge of PCLK to PD valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PD_VV_PDSLOW_MODEON	Hold time from positive edge of PCLK to PD valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PDEVSELNN_VV_PDEVSELNNFAST_MODEON	Hold time from positive edge of PCLK to PDEVSELNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PDEVSELNN_VV_PDEVSELNNNSLOW_MODEON	Hold time from positive edge of PCLK to PDEVSELNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PFRAMENN_VV_PFRAMENNFAST_MODEON	Hold time from positive edge of PCLK to PFRAMENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PFRAMENN_VV_PFRAMENNNSLOW_MODEON	Hold time from positive edge of PCLK to PFRAMENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TH_PCLK_LH_PGNTNN_VV_PGNTNNFAST_MODEON	Hold time from positive edge of PCLK to PGNTNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PGNTNN_VV_PGNTNNSLOW_MODEON	Hold time from positive edge of PCLK to PGNTNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PIDSEL_VV_PIDSELFFAST_MODEON	Hold time from positive edge of PCLK to PIDSEL valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PIDSEL_VV_PIDSELSLOW_MODEON	Hold time from positive edge of PCLK to PIDSEL valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PIRDYNN_VV_PIRDYNNFAST_MODEON	Hold time from positive edge of PCLK to PIRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PIRDYNN_VV_PIRDYNNNSLOW_MODEON	Hold time from positive edge of PCLK to PIRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PLOCKNN_VV_PLOCKNNFAST_MODEON	Hold time from positive edge of PCLK to PLOCKNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PLOCKNN_VV_PLOCKNNNSLOW_MODEON	Hold time from positive edge of PCLK to PLOCKNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PPAR64_VV_PPAR64FAST_MODEON	Hold time from positive edge of PCLK to PPAR64 valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PPAR64_VV_PPAR64SLOW_MODEON	Hold time from positive edge of PCLK to PPAR64 valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in pcimaster_set_timing_control Command (cont.)

Constant Name	Description
PCIMASTER_TH_PCLK_LH_PPAR_VV_PPARFAST_MODEON	Hold time from positive edge of PCLK to PPAR valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PPAR_VV_PPARSLOW_MODEON	Hold time from positive edge of PCLK to PPAR valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PPERRNN_VV_PPERRNNFAST_MODEON	Hold time from positive edge of PCLK to PPERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PPERRNN_VV_PPERRNNSLOW_MODEON	Hold time from positive edge of PCLK to PPERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PREQ64NN_VV_PREQ64NNFAST_MODEON	Hold time from positive edge of PCLK to PREQ64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PREQ64NN_VV_PREQ64NNSLOW_MODEON	Hold time from positive edge of PCLK to PREQ64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PREQNN_VV_PREQNNFAST_MODEON	Hold time from positive edge of PCLK to PREQNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PREQNN_VV_PREQNNNSLOW_MODEON	Hold time from positive edge of PCLK to PREQNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSBONN_VV_PSBONNFAST_MODEON	Hold time from positive edge of PCLK to PSBONN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSBONN_VV_PSBONNSLOW_MODEON	Hold time from positive edge of PCLK to PSBONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TH_PCLK_LH_PSDONE_VV_PSDONEFAST_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSDONE_VV_PSDONESLOW_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSERRNN_VV_PSERRNNFAST_MODEON	Hold time from positive edge of PCLK to PSERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSERRNN_VV_PSERRNNSLOW_MODEON	Hold time from positive edge of PCLK to PSERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSTOPNN_VV_PSTOPNNFAST_MODEON	Hold time from positive edge of PCLK to PSTOPNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PSTOPNN_VV_PSTOPNNSLOW_MODEON	Hold time from positive edge of PCLK to PSTOPNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PTRDYNN_VV_PTRDYNNFAST_MODEON	Hold time from positive edge of PCLK to PTRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TH_PCLK_LH_PTRDYNN_VV_PTRDYNNNSLOW_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
SETUP CONSTANTS	
PCIMASTER_TS_PCLK_LH_PACK64NN_AV_PACK64NNFAST_MODEON	Setup time from positive edge of PCLK to PACK64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in pcimaster_set_timing_control Command (cont.)

Constant Name	Description
PCIMASTER_TS_PCLK_LH_PACK64NN_AV_PACK64NNSLOW_MODEON	Setup time from positive edge of PCLK to PACK64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PADATA_AV_PADATAFAST_MODEON	Setup time from positive edge of PCLK to PADATA any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PADATA_AV_PADATASLOW_MODEON	Setup time from positive edge of PCLK to PADATA any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PBENN_AV_PBENNFAST_MODEON	Setup time from positive edge of PCLK to PBENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PBENN_AV_PBENNSLOW_MODEON	Setup time from positive edge of PCLK to PBENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PCLKRUNNN_AV_PCLKRUNNNFAST_MODEON	Setup time from positive edge of PCLK to PCLKRUNNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PCLKRUNNN_AV_PCLKRUNNNSLOW_MODEON	Setup time from positive edge of PCLK to PCLKRUNNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PCXBENN_AV_PCXBENNFAST_MODEON	Setup time from positive edge of PCLK to PCXBENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PCXBENN_AV_PCXBENNSLOW_MODEON	Setup time from positive edge of PCLK to PCXBENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PD_AV_PDFAST_MODEON	Setup time from positive edge of PCLK to PD any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PD_AV_PDSLOW_MODEON	Setup time from positive edge of PCLK to PD any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TS_PCLK_LH_PDEVSELNN_AV_PDEVSELNNFAST_MODEON	Setup time from positive edge of PCLK to PDEVSELNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PDEVSELNN_AV_PDEVSELNNSLOW_MODEON	Setup time from positive edge of PCLK to PDEVSELNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PFRAMENN_AV_PFRAMENNFASMODEON	Setup time from positive edge of PCLK to PFRAMENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PFRAMENN_AV_PFRAMENNSLOW_MODEON	Setup time from positive edge of PCLK to PFRAMENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PGNTNN_AV_PGNTNNFAST_MODEON	Setup time from positive edge of PCLK to PGNTNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PGNTNN_AV_PGNTNNSLOW_MODEON	Setup time from positive edge of PCLK to PGNTNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PIDSEL_AV_PIDSELFASMODEON	Setup time from positive edge of PCLK to PIDSEL any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PIDSEL_AV_PIDSELSLOW_MODEON	Setup time from positive edge of PCLK to PIDSEL any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PIRDYNN_AV_PIRDYNNFAST_MODEON	Setup time from positive edge of PCLK to PIRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PIRDYNN_AV_PIRDYNNNSLOW_MODEON	Setup time from positive edge of PCLK to PIRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TS_PCLK_LH_PLOCKNN_AV_PLOCKNNFAST_MODEON	Setup time from positive edge of PCLK to PLOCKNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PLOCKNN_AV_PLOCKNNSLOW_MODEON	Setup time from positive edge of PCLK to PLOCKNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPAR64_AV_PPAR64FAST_MODEON	Setup time from positive edge of PCLK to PPAR64 any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPAR64_AV_PPAR64SLOW_MODEON	Setup time from positive edge of PCLK to PPAR64 any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPAR_AV_PPARFAST_MODEON	Setup time from positive edge of PCLK to PPAR any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPAR_AV_PPARSLOW_MODEON	Setup time from positive edge of PCLK to PPAR any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPERRNN_AV_PPERRNNFAST_MODEON	Setup time from positive edge of PCLK to PPERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PPERRNN_AV_PPERRNNSLOW_MODEON	Setup time from positive edge of PCLK to PPERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PREQ64NN_AV_PREQ64NNFAST_MODEON	Setup time from positive edge of PCLK to PREQ64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PREQ64NN_AV_PREQ64NNSLOW_MODEON	Setup time from positive edge of PCLK to PREQ64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in *pcimaster_set_timing_control* Command (cont.)

Constant Name	Description
PCIMASTER_TS_PCLK_LH_PREQNN_AV_PREQNNFAST_MODEON	Setup time from positive edge of PCLK to PREQNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PREQNN_AV_PREQNNNSLOW_MODEON	Setup time from positive edge of PCLK to PREQNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSBONN_AV_PSBONNFAST_MODEON	Setup time from positive edge of PCLK to PSBONN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSBONN_AV_PSBONNSLOW_MODEON	Setup time from positive edge of PCLK to PSBONN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSDONE_AV_PSDONEFAST_MODEON	Setup time from positive edge of PCLK to PSDONE any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSDONE_AV_PSDONESLOW_MODEON	Setup time from positive edge of PCLK to PSDONE any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSERRNN_AV_PSERRNNFAST_MODEON	Setup time from positive edge of PCLK to PSERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSERRNN_AV_PSERRNNNSLOW_MODEON	Setup time from positive edge of PCLK to PSERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSTOPNN_AV_PSTOPNNFAST_MODEON	Setup time from positive edge of PCLK to PSTOPNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PSTOPNN_AV_PSTOPNNNSLOW_MODEON	Setup time from positive edge of PCLK to PSTOPNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMASTER_TS_PCLK_LH_PTRDYNN_AV_PTRDYNNFAST_MODEON	Setup time from positive edge of PCLK to PTRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 60: Individual Constant Values for *timing_param_index* Parameter in pcimaster_set_timing_control Command (cont.)

Constant Name	Description
PCIMASTER_TS_PCLK_LH_PTRDYNN_AV_PTRDYNN_SLOW_MODEON	Setup time from positive edge of PCLK to PTRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

pcislave_fx Parameter Values

Table 61 lists group constant values for the *timing_param_index* parameter in the pcislave_set_timing_control command. Table 62 on page 349 lists individual constant values.

Table 61: Group Constant Values for *timing_param_index* Parameter in pcislave_set_timing_control Command

Constant Name	Description
PCISLAVE_ALL_TIMING	All timing checks and access delays
PCISLAVE_TCHECKS	All timing checks
PCISLAVE_HOLD	All hold timing checks
PCISLAVE_PERIOD	All period checks
PCISLAVE_PERMIN	Only minimum period checks
PCISLAVE_PERMAX	Only maximum period checks
PCISLAVE_PWIDTH	All pulse width checks
PCISLAVE_PWLMIN	Pulse width low minimum checks
PCISLAVE_PWHMIN	Pulse width high minimum checks
PCISLAVE_SETUP	All setup checks
PCISLAVE_NO_TIMING	No timing checks or access delays

Table 62: Individual Constant Values for *timing_param_index* Parameter in pcislave_set_timing_control Command

Constant Name	Description
PERIOD MINIMUM	
PCISLAVE_PERMIN_PCLK_P66MHZ_FAST	Minimum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_PERMIN_PCLK_P66MHZ_SLOW	Minimum period of SYSCLOCK in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PERIOD MAXIMUM	
PCISLAVE_PERMAX_PCLK_P66MHZ_FAST	Maximum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PULSE WIDTH HIGH MINIMUM	
PCISLAVE_PWHMIN_PCLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK high in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_PWHMIN_PCLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK high in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PULSE WIDTH LOW MINIMUM	
PCISLAVE_PWLMIN_PCLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK low in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_PWLMIN_PCLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK low in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_PWLMIN_PRSTNN_P66MHZ_FAST	Minimum pulse duration of RESET low in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_PWLMIN_PRSTNN_P66MHZ_SLOW	Minimum pulse duration of RESET low in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
HOLD CONSTANTS	
PCISLAVE_TH_PCLK_LH_PACK64NN_VV_PACK64NNFAST_MODEON	Hold time from positive edge of PCLK to PACK64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in *pcislave_set_timing_control* Command (cont.)

Constant Name	Description
PCISLAVE_TH_PCLK_LH_PACK64NN_VV_PACK64NNSLOW_MODEON	Hold time from positive edge of PCLK to PACK64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PADATA_VV_PADATAFAST_MODEON	Hold time from positive edge of PCLK to PADATA valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PADATA_VV_PADATASLOW_MODEON	Hold time from positive edge of PCLK to PADATA valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PBENN_VV_PBENNFAST_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PBENN_VV_PBENNSLOW_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PCXBENN_VV_PCXBENNFAST_MODEON	Hold time from positive edge of PCLK to PBENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PCXBENN_VV_PCXBENNSLOW_MODEON	Hold time from positive edge of PCLK to PCXBENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PD_VV_PDFAST_MODEON	Hold time from positive edge of PCLK to PD valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PD_VV_PDSLOW_MODEON	Hold time from positive edge of PCLK to PD valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PDEVSELNN_VV_PDEVSELNNFAST_MODEON	Hold time from positive edge of PCLK to PDEVSELNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PDEVSELNN_VV_PDEVSELNNNSLOW_MODEON	Hold time from positive edge of PCLK to PDEVSELNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in *pcislave_set_timing_control* Command (cont.)

Constant Name	Description
PCISLAVE_TH_PCLK_LH_PFRAMENN_VV_PFRAMENNFAST_MODEON	Hold time from positive edge of PCLK to PFRAMENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PFRAMENN_VV_PFRAMENNSLOW_MODEON	Hold time from positive edge of PCLK to PFRAMENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PGNTNN_VV_PGNTNNFAST_MODEON	Hold time from positive edge of PCLK to PGNTNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PGNTNN_VV_PGNTNNSLOW_MODEON	Hold time from positive edge of PCLK to PGNTNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PIDSEL_VV_PIDSELFAST_MODEON	Hold time from positive edge of PCLK to PIDSEL valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PIDSEL_VV_PIDSELSLOW_MODEON	Hold time from positive edge of PCLK to PIDSEL valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PIRDYNN_VV_PIRDYNNFAST_MODEON	Hold time from positive edge of PCLK to PIRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PIRDYNN_VV_PIRDYNNNSLOW_MODEON	Hold time from positive edge of PCLK to PIRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PLOCKNN_VV_PLOCKNNFAST_MODEON	Hold time from positive edge of PCLK to PLOCKNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PLOCKNN_VV_PLOCKNNNSLOW_MODEON	Hold time from positive edge of PCLK to PLOCKNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PPAR64_VV_PPAR64FAST_MODEON	Hold time from positive edge of PCLK to PPAR64 valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PPAR64_VV_PPAR64SLOW_MODEON	Hold time from positive edge of PCLK to PPAR64 valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in *pcislave_set_timing_control* Command (cont.)

Constant Name	Description
PCISLAVE_TH_PCLK_LH_PPAR_VV_PPARFAST_MODEON	Hold time from positive edge of PCLK to PPAR valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PPAR_VV_PPARSLOW_MODEON	Hold time from positive edge of PCLK to PPAR valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PPERRNN_VV_PPERRNNFAST_MODEON	Hold time from positive edge of PCLK to PPERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PPERRNN_VV_PPERRNNSLOW_MODEON	Hold time from positive edge of PCLK to PPERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PREQ64NN_VV_PREQ64NNFAST_MODEON	Hold time from positive edge of PCLK to PREQ64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PREQ64NN_VV_PREQ64NNSLOW_MODEON	Hold time from positive edge of PCLK to PREQ64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PREQNN_VV_PREQNNFAST_MODEON	Hold time from positive edge of PCLK to PREQNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PREQNN_VV_PREQNNNSLOW_MODEON	Hold time from positive edge of PCLK to PREQNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSBONN_VV_PSBONNFAST_MODEON	Hold time from positive edge of PCLK to PSBONN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSBONN_VV_PSBONNSLOW_MODEON	Hold time from positive edge of PCLK to PSBONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSDONE_VV_PSDONEFAST_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSDONE_VV_PSDONESLOW_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in pcislave_set_timing_control Command (cont.)

Constant Name	Description
PCISLAVE_TH_PCLK_LH_PSERRNN_VV_PSERRNNFAST_MODEON	Hold time from positive edge of PCLK to PSERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSERRNN_VV_PSERRNNSLOW_MODEON	Hold time from positive edge of PCLK to PSERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSTOPNN_VV_PSTOPNNFAST_MODEON	Hold time from positive edge of PCLK to PSTOPNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PSTOPNN_VV_PSTOPNNSLOW_MODEON	Hold time from positive edge of PCLK to PSTOPNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PTRDYNN_VV_PTRDYNNFAST_MODEON	Hold time from positive edge of PCLK to PTRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TH_PCLK_LH_PTRDYNN_VV_PTRDYNNNSLOW_MODEON	Hold time from positive edge of PCLK to PSDONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
SETUP CONSTANTS	
PCISLAVE_TS_PCLK_LH_PACK64NN_AV_PACK64NNFAST_MODEON	Setup time from positive edge of PCLK to PACK64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PACK64NN_AV_PACK64NNSLOW_MODEON	Setup time from positive edge of PCLK to PACK64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PADATA_AV_PADATAFAST_MODEON	Setup time from positive edge of PCLK to PADATA any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PADATA_AV_PADATASLOW_MODEON	Setup time from positive edge of PCLK to PADATA any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PBENN_AV_PBENNFAST_MODEON	Setup time from positive edge of PCLK to PBENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in pcislave_set_timing_control Command (cont.)

Constant Name	Description
PCISLAVE_TS_PCLK_LH_PBENN_AV_PBENNSLOW_MODEON	Setup time from positive edge of PCLK to PBENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PCXBENN_AV_PCXBENNFAST_MODEON	Setup time from positive edge of PCLK to PCXBENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PCXBENN_AV_PCXBENNSLOW_MODEON	Setup time from positive edge of PCLK to PCXBENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PD_AV_PDFAST_MODEON	Setup time from positive edge of PCLK to PD any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PD_AV_PDSLOW_MODEON	Setup time from positive edge of PCLK to PD any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PDEVSELNN_AV_PDEVSELNNFAST_MODEON	Setup time from positive edge of PCLK to PDEVSELNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PDEVSELNN_AV_PDEVSELNNSLOW_MODEON	Setup time from positive edge of PCLK to PDEVSELNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PFRAMENN_AV_PFRAMENNFAST_MODEON	Setup time from positive edge of PCLK to PFRAMENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PFRAMENN_AV_PFRAMENNSLOW_MODEON	Setup time from positive edge of PCLK to PFRAMENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PGNTNN_AV_PGNTNNFAST_MODEON	Setup time from positive edge of PCLK to PGNTNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PGNTNN_AV_PGNTNNSLOW_MODEON	Setup time from positive edge of PCLK to PGNTNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in *pcislave_set_timing_control* Command (cont.)

Constant Name	Description
PCISLAVE_TS_PCLK_LH_PIDSEL_AV_PIDSELFFAST_MODEON	Setup time from positive edge of PCLK to PIDSEL any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PIDSEL_AV_PIDSELSLOW_MODEON	Setup time from positive edge of PCLK to PIDSEL any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PIRDYNN_AV_PIRDYNNFAST_MODEON	Setup time from positive edge of PCLK to PIRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PIRDYNN_AV_PIRDYNNNSLOW_MODEON	Setup time from positive edge of PCLK to PIRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PLOCKNN_AV_PLOCKNNFAST_MODEON	Setup time from positive edge of PCLK to PLOCKNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PLOCKNN_AV_PLOCKNNNSLOW_MODEON	Setup time from positive edge of PCLK to PLOCKNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPAR64_AV_PPAR64FAST_MODEON	Setup time from positive edge of PCLK to PPAR64 any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPAR64_AV_PPAR64SLOW_MODEON	Setup time from positive edge of PCLK to PPAR64 any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPAR_AV_PPARFAST_MODEON	Setup time from positive edge of PCLK to PPAR any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPAR_AV_PPARSLOW_MODEON	Setup time from positive edge of PCLK to PPAR any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPERRNN_AV_PPERRNNFAST_MODEON	Setup time from positive edge of PCLK to PPERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PPERRNN_AV_PPERRNNNSLOW_MODEON	Setup time from positive edge of PCLK to PPERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in *pcislave_set_timing_control* Command (cont.)

Constant Name	Description
PCISLAVE_TS_PCLK_LH_PREQ64NN_AV_PREQ64NNFAST_MODEON	Setup time from positive edge of PCLK to PREQ64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PREQ64NN_AV_PREQ64NNSLOW_MODEON	Setup time from positive edge of PCLK to PREQ64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PREQNN_AV_PREQNNFAST_MODEON	Setup time from positive edge of PCLK to PREQNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PREQNN_AV_PREQNNNSLOW_MODEON	Setup time from positive edge of PCLK to PREQNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSBONN_AV_PSBONNFAST_MODEON	Setup time from positive edge of PCLK to PSBONN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSBONN_AV_PSBONNNSLOW_MODEON	Setup time from positive edge of PCLK to PSBONN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSDONE_AV_PSDONEFAST_MODEON	Setup time from positive edge of PCLK to PSDONE any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSDONE_AV_PSDONESLOW_MODEON	Setup time from positive edge of PCLK to PSDONE any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSERRNN_AV_PSERRNNFAST_MODEON	Setup time from positive edge of PCLK to PSERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSERRNN_AV_PSERRNNNSLOW_MODEON	Setup time from positive edge of PCLK to PSERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSTOPNN_AV_PSTOPNNFAST_MODEON	Setup time from positive edge of PCLK to PSTOPNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PSTOPNN_AV_PSTOPNNNSLOW_MODEON	Setup time from positive edge of PCLK to PSTOPNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 62: Individual Constant Values for *timing_param_index* Parameter in pcislave_set_timing_control Command (cont.)

Constant Name	Description
PCISLAVE_TS_PCLK_LH_PTRDYNN_AV_PTRDYNNFAST_MODEON	Setup time from positive edge of PCLK to PTRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCISLAVE_TS_PCLK_LH_PTRDYNN_AV_PTRDYNNNSLOW_MODEON	Setup time from positive edge of PCLK to PTRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

pcimonitor_fx Parameter Values

Table 63 lists group constant values for the *timing_param_index* parameter in the pcimonitor_set_timing_control command. Table 64 on page 358 lists individual constant values.

Table 63: Group Constant Values for *timing_param_index* Parameter in pcimonitor_set_timing_control Command

Constant Name	Description
PCIMONITOR_ALL_TIMING	All timing checks and access delays
PCIMONITOR_TCHECKS	All timing checks
PCIMONITOR_HOLD	All hold timing checks
PCIMONITOR_PERIOD	All period checks
PCIMONITOR_PERMIN	Only minimum period checks
PCIMONITOR_PERMAX	Only maximum period checks
PCIMONITOR_PWIDTH	All pulse width checks
PCIMONITOR_PWLMIN	Pulse width low minimum checks
PCIMONITOR_PWHMIN	Pulse width high minimum checks
PCIMONITOR_SETUP	All setup checks
PCIMONITOR_NO_TIMING	No timing checks or access delays

Table 64: Individual Constant Values for *timing_param_index* Parameter in pcimonitor_set_timing_control Command

Constant Name	Description
PERIOD MINIMUM	
PCIMONITOR_PERMIN_CLK_P66MHZ_SLOW	Minimum period of SYSCLOCK in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_PERMIN_CLK_P66MHZ_FAST	Minimum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PERIOD MAXIMUM	
PCIMONITOR_PERMAX_CLK_P66MHZ_FAST	Maximum period of SYSCLOCK in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PULSE WIDTH LOW MINIMUM	
PCIMONITOR_PWLMIN_CLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK low in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_PWLMIN_CLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK low in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PULSE WIDTH HIGH MINIMUM	
PCIMONITOR_PWHMIN_CLK_P66MHZ_FAST	Minimum pulse duration of SYSCLOCK high in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_PWHMIN_CLK_P66MHZ_SLOW	Minimum pulse duration of SYSCLOCK high in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
HOLD CONSTANTS	
PCIMONITOR_TH_CLK_LH_ACK64NN_VV_ACK64NNSLOW_MODEON	Hold time from positive edge of CLK to ACK64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_ACK64NN_VV_ACK64NNFAST_MODEON	Hold time from positive edge of CLK to ACK64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_AD_VV_ADSLOW_MODEON	Hold time from positive edge of CLK to AD valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in *pcimonitor_set_timing_control* Command (cont.)

Constant Name	Description
PCIMONITOR_TH_CLK_LH_AD_VV_ADFAST_MODEON	Hold time from positive edge of CLK to AD valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_CXBENN_VV_CXBENNSLOW_MODEON	Hold time from positive edge of CLK to CXBENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_CXBENN_VV_CXBENNFASFAST_MODEON	Hold time from positive edge of CLK to CXBENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_DEVSELNN_VV_DEVSELNNSLOW_MODEON	Hold time from positive edge of CLK to DEVSELNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_DEVSELNN_VV_DEVSELNNFAST_MODEON	Hold time from positive edge of CLK to DEVSELNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_FRAMENN_VV_FRAMENNSLOW_MODEON	Hold time from positive edge of CLK to FRAMENN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_FRAMENN_VV_FRAMENNFASFAST_MODEON	Hold time from positive edge of CLK to FRAMENN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_GNTNN_VV_GNTNNSLOW_MODEON	Hold time from positive edge of CLK to GNTNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_GNTNN_VV_GNTNNFAST_MODEON	Hold time from positive edge of CLK to GNTNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_IDSEL_VV_IDSELSLOW_MODEON	Hold time from positive edge of CLK to IDSEL valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_IDSEL_VV_IDSELFASFAST_MODEON	Hold time from positive edge of CLK to IDSEL valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTANN_VV_INTANNSLOW_MODEON	Hold time from positive edge of CLK to INTANN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in pcimonitor_set_timing_control Command (cont.)

Constant Name	Description
PCIMONITOR_TH_CLK_LH_INTANN_VV_ INTANNFAST_MODEON	Hold time from positive edge of CLK to INTANN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTBNN_VV_ INTBNNFAST_MODEON	Hold time from positive edge of CLK to INTBNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTBNN_VV_ INTBNNFAST_MODEON	Hold time from positive edge of CLK to INTBNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTCNN_VV_ INTCNNSLOW_MODEON	Hold time from positive edge of CLK to INTCNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTCNN_VV_ INTCNNSLOW_MODEON	Hold time from positive edge of CLK to INTCNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTDNN_VV_ INTDNNFAST_MODEON	Hold time from positive edge of CLK to INTDNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_INTDNN_VV_ INTDNNFAST_MODEON	Hold time from positive edge of CLK to INTDNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_IRDYNN_VV_ IRDYNNFAST_MODEON	Hold time from positive edge of CLK to IRDYNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_IRDYNN_VV_ IRDYNNFAST_MODEON	Hold time from positive edge of CLK to IRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_LOCKNN_VV_ LOCKNNFAST_MODEON	Hold time from positive edge of CLK to LOCKNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_LOCKNN_VV_ LOCKNNFAST_MODEON	Hold time from positive edge of CLK to LOCKNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_PAR64_VV_ PAR64FAST_MODEON	Hold time from positive edge of CLK to PAR64 valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in *pcimonitor_set_timing_control* Command (cont.)

Constant Name	Description
PCIMONITOR_TH_CLK_LH_PAR64_VV_ PAR64FAST_MODEON	Hold time from positive edge of CLK to PAR64 valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_PAR_VV_ PARSLOW_MODEON	Hold time from positive edge of CLK to PAR valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_PAR_VV_ PARFAST_MODEON	Hold time from positive edge of CLK to PAR valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_PERRNN_VV_ PERRNNSLOW_MODEON	Hold time from positive edge of CLK to PERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_PERRNN_VV_ PERRNNFAST_MODEON	Hold time from positive edge of CLK to PERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_REQ64NN_VV_ REQ64NNSLOW_MODEON	Hold time from positive edge of CLK to REQ64NN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_REQ64NN_VV_ REQ64NNFAST_MODEON	Hold time from positive edge of CLK to REQ64NN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_REQNN_VV_ REQNNSLOW_MODEON	Hold time from positive edge of CLK to REQNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_REQNN_VV_ REQNNFAST_MODEON	Hold time from positive edge of CLK to REQNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_SBONN_VV_ SBONNSLOW_MODEON	Hold time from positive edge of CLK to SBONN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_SBONN_VV_ SBONNFAST_MODEON	Hold time from positive edge of CLK to SBONN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_SDONE_VV_ SDONESLOW_MODEON	Hold time from positive edge of CLK to SDONE valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in pcimonitor_set_timing_control Command (cont.)

Constant Name	Description
PCIMONITOR_TH_CLK_LH_SDONE_VV_ SDONEFAST_MODEON	Hold time from positive edge of CLK to SDONE valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_SERRNN_VV_ SERRNNSLOW_MODEON	Hold time from positive edge of CLK to SERRNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_SERRNN_VV_ SERRNNFAST_MODEON	Hold time from positive edge of CLK to SERRNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_STOPNN_VV_ STOPNNSLOW_MODEON	Hold time from positive edge of CLK to STOPNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_STOPNN_VV_ STOPNNFAST_MODEON	Hold time from positive edge of CLK to STOPNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_TRDYNN_VV_ TRDYNNNSLOW_MODEON	Hold time from positive edge of CLK to TRDYNN valid-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TH_CLK_LH_TRDYNN_VV_ TRDYNNFAST_MODEON	Hold time from positive edge of CLK to TRDYNN valid-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
SETUP CONSTANTS	
PCIMONITOR_TS_CLK_LH_ACK64NN_AV_ ACK64NNSLOW_MODEON	Setup time from positive edge of CLK to ACK64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_ACK64NN_AV_ ACK64NNFAST_MODEON	Setup time from positive edge of CLK to ACK64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_AD_AV_ ADSLOW_MODEON	Setup time from positive edge of CLK to AD any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_AD_AV_ ADFAST_MODEON	Setup time from positive edge of CLK to AD any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in *pcimonitor_set_timing_control* Command (cont.)

Constant Name	Description
PCIMONITOR_TS_CLK_LH_CXBENN_AV_CXBENNSLOW_MODEON	Setup time from positive edge of CLK to CXBENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_CXBENN_AV_CXBENNFAST_MODEON	Setup time from positive edge of CLK to CXBENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_DEVSELNN_AV_DEVSELNNSLOW_MODEON	Setup time from positive edge of CLK to DEVSELNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_DEVSELNN_AV_DEVSELNNFAST_MODEON	Setup time from positive edge of CLK to DEVSELNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_FRAMENN_AV_FRAMENNSLOW_MODEON	Setup time from positive edge of CLK to FRAMENN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_FRAMENN_AV_FRAMENNFAST_MODEON	Setup time from positive edge of CLK to FRAMENN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_GNTNN_AV_GNTNNSLOW_MODEON	Setup time from positive edge of CLK to GNTNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_GNTNN_AV_GNTNNFAST_MODEON	Setup time from positive edge of CLK to GNTNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_IDSEL_AV_IDSELSLOW_MODEON	Setup time from positive edge of CLK to IDSEL any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_IDSEL_AV_IDSELFFAST_MODEON	Setup time from positive edge of CLK to IDSEL any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTANN_AV_INTANNSLOW_MODEON	Setup time from positive edge of CLK to INTANN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTANN_AV_INTANNFAST_MODEON	Setup time from positive edge of CLK to INTANN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in *pcimonitor_set_timing_control* Command (cont.)

Constant Name	Description
PCIMONITOR_TS_CLK_LH_INTBNN_AV_INTBNN_SLOW_MODEON	Setup time from positive edge of CLK to INTBNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTBNN_AV_INTBNN_FAST_MODEON	Setup time from positive edge of CLK to INTBNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTCNN_AV_INTCNN_SLOW_MODEON	Setup time from positive edge of CLK to INTCNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTCNN_AV_INTCNN_FAST_MODEON	Setup time from positive edge of CLK to INTCNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTDNN_AV_INTDNN_SLOW_MODEON	Setup time from positive edge of CLK to INTDNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_INTDNN_AV_INTDNN_FAST_MODEON	Setup time from positive edge of CLK to INTDNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_IRDYNN_AV_IRDYNN_SLOW_MODEON	Setup time from positive edge of CLK to IRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_IRDYNN_AV_IRDYNN_FAST_MODEON	Setup time from positive edge of CLK to IRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_LOCKNN_AV_LOCKNN_SLOW_MODEON	Setup time from positive edge of CLK to LOCKNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_LOCKNN_AV_LOCKNN_FAST_MODEON	Setup time from positive edge of CLK to LOCKNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_PAR64_AV_PAR64_SLOW_MODEON	Setup time from positive edge of CLK to PAR64 any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_PAR64_AV_PAR64_FAST_MODEON	Setup time from positive edge of CLK to PAR64 any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in *pcimonitor_set_timing_control* Command (cont.)

Constant Name	Description
PCIMONITOR_TS_CLK_LH_PAR_AV_PARSLOW_MODEON	Setup time from positive edge of CLK to PAR any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_PAR_AV_PARFAST_MODEON	Setup time from positive edge of CLK to PAR any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_PERRNN_AV_PERRNNSLOW_MODEON	Setup time from positive edge of CLK to PERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_PERRNN_AV_PERRNNFAST_MODEON	Setup time from positive edge of CLK to PERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_REQ64NN_AV_REQ64NNSLOW_MODEON	Setup time from positive edge of CLK to REQ64NN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_REQ64NN_AV_REQ64NNFAST_MODEON	Setup time from positive edge of CLK to REQ64NN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_REQNN_AV_REQNNSLOW_MODEON	Setup time from positive edge of CLK to REQNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_REQNN_AV_REQNNFAST_MODEON	Setup time from positive edge of CLK to REQNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_SBONN_AV_SBONNSLOW_MODEON	Setup time from positive edge of CLK to SBONN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_SBONN_AV_SBONNFAST_MODEON	Setup time from positive edge of CLK to SBONN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_SDONE_AV_SDONESLOW_MODEON	Setup time from positive edge of CLK to SDONE any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_SDONE_AV_SDONEFAST_MODEON	Setup time from positive edge of CLK to SDONE any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

Table 64: Individual Constant Values for *timing_param_index* Parameter in pcimonitor_set_timing_control Command (cont.)

Constant Name	Description
PCIMONITOR_TS_CLK_LH_SERRNN_AV_SERRNNSLOW_MODEON	Setup time from positive edge of CLK to SERRNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_SERRNN_AV_SERRNNFAST_MODEON	Setup time from positive edge of CLK to SERRNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_STOPNN_AV_STOPNNSLOW_MODEON	Setup time from positive edge of CLK to STOPNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_STOPNN_AV_STOPNNFAST_MODEON	Setup time from positive edge of CLK to STOPNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_TRDYNN_AV_TRDYNNNSLOW_MODEON	Setup time from positive edge of CLK to TRDYNN any-to-valid in 33MHz (conventional PCI mode) or 66MHz (PCI-X mode)
PCIMONITOR_TS_CLK_LH_TRDYNN_AV_TRDYNNFAST_MODEON	Setup time from positive edge of CLK to TRDYNN any-to-valid in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)

B

Ensuring Compatibility with PCI/PCI-X FlexModels

Introduction

The PCI FlexModels support the PCI-X Addendum to the PCI Specification, Version 1.0. The PCI/PCI-X FlexModel set includes both PCI and PCI-X system testbenches and the `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx` models with the MDL version numbers shown below:

- **`pcimaster_fx`**: MDL version 01018 (1.18) or higher
- **`pcislave_fx`**: MDL version 01016 (1.16) or higher
- **`pcimonitor_fx`**: MDL version 01017 (1.17) or higher

For PCI FlexModel Test Suite users, the set also includes an updated PCI Test Suite (MDL version 01006 or higher) that supports the PCI/PCI-X FlexModels.

When using the PCI/PCI-X FlexModels, there are some compatibility issues of which you should be aware. These issues fall into four categories, described in the following sections:

- [“Compatibility Within the PCI/PCI-X FlexModel Set” on page 368](#)
- [“Compatibility With Existing Testbenches” on page 368](#)
- [“Compatibility With the PCI FlexModel Test Suite” on page 371](#)

Compatibility Within the PCI/PCI-X FlexModel Set

It is especially important that you reinstall all models in the PCI/PCI-X FlexModel set each time you update one or more PCI FlexModels. This ensures that you are using a tested and supportable combination. If you are a PCI FlexModel Test Suite user, this applies to the FlexModel Test Suite as well.

Compatibility With Existing Testbenches

With the PCI/PCI-X FlexModels, you can use the model in either conventional PCI mode or PCI-X mode. In conventional PCI mode, PCI models are backward-compatible with the exception of some pin changes and a change to the TimingVersion generic/defparameter. These exceptions require you to use one of two methods to make your existing testbench fully compatible with the PCI/PCI-X FlexModels:

- Method 1: You can modify your existing testbench (recommended)
- or
- Method 2: You can install the Synopsys-supplied compatibility files.

Method 1 lets you use PCI-X mode and all other enhancements. This is the recommended method. The changes required by this method need only be done once, making future upgrades easier and version control simpler. This is the method you need to use if you are running the updated PCI FlexModel Test Suite (MDL version 01006 or higher).

Method 2 lets you use the PCI/PCI-X FlexModels with your existing testbench, but does not allow you to use the models in PCI-X mode. If you use Method 2, you must reinstall the compatibility files every time you upgrade your PCI models. This method makes the PCI/PCI-X FlexModels incompatible with the updated PCI FlexModel Test Suite (MDL version 01006 or higher).

Method 1: Modifying Your Existing Testbench

To change your existing testbench to make it fully compatible with the PCI/PCI-X FlexModels, follow these steps:

1. Change the TimingVersion generic/defparameter for each instance.

You can use 33MHz or 66MHz timing in conventional PCI mode, and 66MHz or 133MHz timing in PCI-X mode; therefore, there are two new settings for the TimingVersion generic/defparameter: “conventional” and “pcix”. To use 33MHz or 66MHz timing in conventional PCI mode, you must change the TimingVersion generic/defparameter for each model instance to “conventional”.

2. Modify your pcimaster_fx instantiation, if necessary, to make the pack64nn and pserrnn pins bidirectional.

When PCI-X mode was added to the pcimaster_fx, it was necessary to change two pins (pack64nn and pserrnn) from input to bidirectional. This may require you to modify your instantiation of the pcimaster_fx.

3. Modify your pcislave_fx instantiation, if necessary, to include new pins or new pin directions.

When PCI-X mode was added to the pcislave_fx, it was necessary to add some pins and change the direction of others. This may require you to modify your instantiation of the pcislave_fx. [Table 65](#) lists the pins that are new or have changed.

Table 65: New and Changed Pins for the pcislave_fx

Device Pin Name	Direction	Model Pin Name
C/BE#[3:0]	In/out	PCISLAVE_PCXBENN_BUS
C/BE#[7:4]	In/out	PCISLAVE_PBENN_BUS
GNT# (new pin)	In	PCISLAVE_PGNTNN_PIN
IRDY#	In/out	PCISLAVE_PIRDYNN_PIN
LOCK#	In/out	PCISLAVE_PLOCKNN_PIN
REQ# (new pin)	In/out	PCISLAVE_PREQNN_PIN
REQ64#	In/out	PCISLAVE_PREQ64NN_PIN
FRAME#	In/out	PCISLAVE_PFRAMENN_PIN

Table 65: New and Changed Pins for the pcislave_fx (cont.)

Device Pin Name	Direction	Model Pin Name
ACK64#	In/out	PCISLAVE_PACK64NN_PIN
DEVSEL#	In/out	PCISLAVE_PDEVSELNN_PIN
PERR#	In/out	PCISLAVE_PPERRNN_PIN
SERR#	In/out	PCISLAVE_PSERRNN_PIN

Method 2: Installing the Compatibility Files

The easiest way to install the compatibility files is to use the `flexm_setup` script to set up a work directory that contains all the package files. This procedure is shown below. For details on using the `flexm_setup` script, see the [FlexModel User's Manual](#).

1. Run `flexm_setup`:

To copy all the files required to run your existing testbench, including the compatibility files, run the following commands:

```
% mkdir workdir
% $LMC_HOME/lib/bin/flexm_setup -dir workdir pcislave_fx
% $LMC_HOME/lib/bin/flexm_setup -dir workdir pcimaster_fx
% $LMC_HOME/lib/bin/flexm_setup -dir workdir pcimonitor_fx
```

You typically create *workdir* in the simulation directory.

2. Copy the compatibility files:

This procedure works as shown only after you run `flexm_setup` as shown in Step 1.

For VHDL:

```
% cd workdir/examples/vhdl
% cp pcislave_old.vhd pcislave.vhd
% cp pcimaster_old.vhd pcimaster.vhd
% cp pcimonitor_old.vhd pcimonitor.vhd

% cd ../../src/vhdl
% cp pcislave_pkg_old.vhd pcislave_pkg.vhd
% cp pcimaster_pkg_old.vhd pcimaster_pkg.vhd
% cp pcimonitor_pkg_old.vhd pcimonitor_pkg.vhd
```

For Verilog:

```
% cd workdir/examples/verilog
% cp pcislave_old.v pcislave.v
% cp pcimaster_old.v pcimaster.v
% cp pcimonitor_old.v pcimonitor.v
```

3. Compile VHDL packages for simulation (VHDL only):

For VHDL only, you must make sure that all packages are compiled for simulation. The following is an example for running the MTI simulator using the Solaris platform. If you are running on an HP platform, the commands are slightly different.

```
% cp $LMC_HOME/lib/sun4Solaris.lib/slm_mti.so .

% vlib mti_work
% vmap slm_lib ./mti_work
% vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
% vcom -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd

% vcom -work slm_lib workdir/src/vhdl/pcislave_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcislave_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimonitor_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_user_pkg.vhd
% vcom -work slm_lib workdir/src/vhdl/pcimaster_pkg.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcislave_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster_fx_mti.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimonitor_fx_mti.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcisys_fx_comp.vhd

% vcom -work slm_lib workdir/examples/vhdl/pcimonitor.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcislave.vhd
% vcom -work slm_lib workdir/examples/vhdl/pcimaster.vhd
```

Compatibility With the PCI FlexModel Test Suite

To ensure full compatibility between the PCI/PCI-X FlexModels and the PCI FlexModel Test Suite, you must install the updated PCI Test Suite that comes with your PCI/PCI-X FlexModel set. The updated version of the PCI Test Suite (MDL version 01006 or higher) fully supports the PCI/PCI-X FlexModels.

C

VERA Testbench Scripts for Verilog-XL, NC-Verilog, and MTI

This addendum provides you with example scripts to run both the PCIX and PCI VERA testbenches for the following simulators:

- Verilog-XL
- NC-Verilog
- MTI ModelSim



Note

A new and improved version of the VERA testbench was released in the July 2002 release. The scripts provided in the datasheet use the July 2002 release, and not earlier versions of the VERA testbench. The scripts in this section and other sections are meant as an example only, and must be edited to ensure proper execution.

The example scripts assume the following:

- You will be using Solaris as the testbench platform. You may need to change the path to model object files as necessary. Following are the location for the object files for Solaris, HP-UX, and Linux which are critical for the testbench:
 - `${LMC_HOME}/lib/sun4Solaris.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/hp700.lib/slm_pli.o \`
 - `${LMC_HOME}/lib/x86_linux.lib/slm_pli.o \`
- You have complete rights to the directory where you place the VERA testbench.
- You have a good LMC_HOME tree with the PCIX model set: `pcimaster_fx`, `pcislave_fx`, and `pcimonitor_fx`.

- Your DISPLAY environmental variable is correctly set.
- LD_LIBRARY_PATH is properly set .If not do a “ setenv LD_LIBRARY_PATH 1” on your command line.

Before you excuted the example scripts do the following:

- Create a directory to hold VERA testbench files. You can give this top level directory any name.
- Create a directory to hold all the compiled *.vro VERA files in your testbench directory. *This directory must be called file_vro.* For example:

```
mkdir vera_tb_folder
cd vera_tb_folder
mkdir file_vro
```

- Copy and paste the example script into a file and execute it from the directory you created to hold the testbench files.
- Change your location to your local testbench directory, and invoke the script from that location.



Note

Edit your script to remove any continuation backslashes before the flexm_setup utility. In some script environments, you cannot put a continuation character before the command. If flexm_setup has any problems with the backslash, you will get an error message stating the model name was not found.

The scripts all have the same five parts. Each part is set off by comments in the script example. Edit each section as is appropriate for your environment if needed. The five sections are:

1. Setup environmental variables for the simulator, LMC_HOME tree, and VERA environment.
2. Setup testbench parameters. These are the *.io and *.cfg files used by the other testbenches.
3. Create the vera_user.o and the vera_local.dll files.
4. Create the *.vro files needed by the PCI(X) FlexModels.
5. Invoke the simulator and run the testbench.

**Note**

The text for the script has been made smaller than normal to allow you to copy and paste it from the PDF file. As a result, reading it on a terminal will be difficult unless you magnify the text with Adobe's Acrobat Reader.

Example Script to Run the PCI VERA Testbench with the NC-Verilog Simulator

```
#!/bin/csh -fx

# This is an example script to setup and execute
# the PCI VERA testbench with NC-Verilog.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup NC Verilog
setenv CDS_INST_DIR /d/cds_ldv31/solaris251

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /d/ae/work2/test/release
set path = (${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $path)
setenv LD_LIBRARY_PATH .:${LMC_HOME}/lib/sun4Solaris.lib:${CDS_INST_DIR}/tools/lib:${LD_LIBRARY_PATH}
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, CDS_INST_DIR, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH, set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcislave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCI_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
    -I$VERA_HOME/lib \
    ${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
    ${LMC_LIB_DIR}/vera_slm_pli.o

cc -O -KPIC -c -o ./veriuser.o -I${CDS_INST_DIR}/tools/include \
    -I${CDS_INST_DIR}/tools/inca/include \
    -I${VERA_HOME}/lib/vlog ${VERA_HOME}/lib/vlog/veriuser.c
ld -G -o ./libpli.so ./veriuser.o \
    ${VERA_HOME}/lib/vlog/libSysSciTask.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```



```
##### CREATE *.VRO FILES NEEDED BY THE PCI FLEXMODELS #####
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_InitClass.vr \
./file_vro/PCI_InitClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_BURSTClass.vr \
./file_vro/PCI_BURSTClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DECODEandWAITClass.vr \
./file_vro/PCI_DECODEandWAITClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DO64BITSClass.vr \
./file_vro/PCI_DO64BITSClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DWORDClass.vr \
./file_vro/PCI_DWORDClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_GenericClass.vr \
./file_vro/PCI_GenericClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_MASTERClass.vr \
./file_vro/PCI_MASTERClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_PARITYClass.vr \
./file_vro/PCI_PARITYClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_READresultClass.vr \
./file_vro/PCI_READresultClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
```

```

-I. `$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_RandomizeClass.vr \
./file_vro/PCI_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$(LMC_HOME)/sim/vera/src \
-I`$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcislave_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_TARGETClass.vr \
./file_vro/PCI_TARGETClass.vro || exit 1

vera -cmp -g -I$(LMC_HOME)/sim/vera/src \
-I`$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcislave_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/pci_sys_tst.vr \
./file_vro/pci_sys_tst.vro || exit 1

vera -cmp -g -I$(LMC_HOME)/sim/vera/src $(LMC_HOME)/sim/vera/src/1stmodel.vr \
./file_vro/1stmodel.vro || exit 1
vera -cmp -g -I$(LMC_HOME)/sim/vera/src $(LMC_HOME)/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$(LMC_HOME)/sim/vera/src $(LMC_HOME)/sim/vera/src/swiftdmodel.vr \
./file_vro/swiftdmodel.vro || exit 1
vera -cmp -g -I$(LMC_HOME)/sim/vera/src -I. `$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/src/vera/pcimaster_pkg.vr \
./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$(LMC_HOME)/sim/vera/src -I. `$(LMC_HOME)/bin/flexm_setup pcislave_fx`/src/vera/pcislave_pkg.vr \
./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$(LMC_HOME)/sim/vera/src -I. `$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/src/vera/pcimonitor_pkg.vr \
./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE NC Verilog AND RUN THE PCI TESTBENCH #####

unsetenv SSI_LIB_FILES

$CDS_INST_DIR/tools/bin/ncverilog +loadpli=swiftpli:swift_boot \
`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/vera/pcisys_vera_tst_top.v \
./file_vro/pci_sys_tst.vshell \
`$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
`$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vxl.v \
`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vxl.v \
`$(LMC_HOME)/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
`$(LMC_HOME)/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vxl.v \
+incdir+$(LMC_HOME)/sim/pli/src \
+incdir+`$(LMC_HOME)/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+`$(LMC_HOME)/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+`$(LMC_HOME)/bin/flexm_setup pcislave_fx`/src/verilog \
+vera_udf=./vera_local.dl +vera_mload=`$(LMC_HOME)/bin/flexm_setup \
pcimonitor_fx`/examples/vera/pci_sys_test_top.vr1

```

Example Script to Run the PCIX VERA Testbench with the NC-Verilog Simulator

```
#!/bin/csh -fx

# This example script setups and executes
# the PCIX VERA testbench for NC-Verilog.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup NC Verilog
setenv CDS_INST_DIR /d/cds_ldv31/solaris251

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /synopsys/release
set path = (${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $path)
setenv LD_LIBRARY_PATH .:${LMC_HOME}/lib/sun4Solaris.lib:${CDS_INST_DIR}/tools/lib:$LD_LIBRARY_PATH
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, CDS_INST_DIR, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH, set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixslave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCIX_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
-I$VERA_HOME/lib \
${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
${LMC_LIB_DIR}/vera_slm_pli.o

cc -O -KPIC -c -o ./veriuser.o -I${CDS_INST_DIR}/tools/include \
-I${CDS_INST_DIR}/tools/inca/include \
-I${VERA_HOME}/lib/vlog ${VERA_HOME}/lib/vlog/veriuser.c
ld -G -o ./libpli.so ./veriuser.o \
${VERA_HOME}/lib/vlog/libSysSciTask.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```

```
##### CREATE *.VRO FILES NEEDED BY THE PCIX FLEXMODELS #####

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_InitClass.vr ./file_vro/PCIX_InitClass.vro
|| exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_BURSTClass.vr
./file_vro/PCIX_BURSTClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DECODEandWAITClass.vr
./file_vro/PCIX_DECODEandWAITClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_D064BITSClass.vr
./file_vro/PCIX_D064BITSClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DWORDClass.vr
./file_vro/PCIX_DWORDClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_GenericClass.vr
./file_vro/PCIX_GenericClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_MASTERClass.vr
./file_vro/PCIX_MASTERClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_PARITYClass.vr
./file_vro/PCIX_PARITYClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_READresultClass.vr
./file_vro/PCIX_READresultClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_RandomizeClass.vr
```

```

./file_vro/PCIX_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_TARGETClass.vr
./file_vro/PCIX_TARGETClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_SPLITClass.vr
./file_vro/PCIX_SPLITClass.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcix_sys_tst.vr ./file_vro/pcix_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/lstmodel.vr ./file_vro/lstmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr ./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swifftmodel.vr ./file_vro/swifftmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera/pcimaster_pkg.vr
./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera/pcislave_pkg.vr
./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera/pcimonitor_pkg.vr
./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE NC Verilog AND RUN THE PCIX TESTBENCH #####

unsetenv SSI_LIB_FILES

$CDS_INST_DIR/tools/bin/ncverilog +loadpli=swiftpli:swift_boot \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixsys_vera_tst_top.v \
./file_vro/pcix_sys_tst.vshell \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vxl.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vxl.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vxl.v \
+incdir+{$LMC_HOME}/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/verilog \
+vera_udf=./vera_local.dl +vera_mload=`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcix_sys_test_top.vr1

```

Example Script to Run the PCI VERA Testbench with Verilog-XL

```
#!/bin/csh -fx

# This example script setups and executes
# the PCI VERA testbench for Verilog-XL.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup Verilog-XL
setenv CDS_INST_DIR /d/cds_ldv31/solaris251

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /synopsys/release
set path = (${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $path)
setenv LD_LIBRARY_PATH .:${LMC_HOME}/lib/sun4Solaris.lib:${CDS_INST_DIR}/tools/lib:${LD_LIBRARY_PATH}
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:5280@grymoire
setenv LMC_LIB_DIR ${LMC_HOME}/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, CDS_INST_DIR, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH,set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp `${LMC_HOME}/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcislave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCI_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
-I${VERA_HOME}/lib \
${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
${LMC_LIB_DIR}/vera_slm_pli.o

cc -O -KPIC -c -o ./veriuser.o -I${CDS_INST_DIR}/tools/include \
-I${CDS_INST_DIR}/tools/inca/include \
-I${VERA_HOME}/lib/vlog ${VERA_HOME}/lib/vlog/veriuser.c
ld -G -o ./libpli.so ./veriuser.o \
${VERA_HOME}/lib/vlog/libSysSciTask.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```

```
##### CREATE *.VRO FILES NEEDED BY THE PCI FLEXMODELS #####

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_InitClass.vr \
./file_vro/PCI_InitClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_BURSTClass.vr \
./file_vro/PCI_BURSTClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DECODEandWAITClass.vr \
./file_vro/PCI_DECODEandWAITClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DO64BITSClass.vr \
./file_vro/PCI_DO64BITSClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_DWORDClass.vr \
./file_vro/PCI_DWORDClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_GenericClass.vr \
./file_vro/PCI_GenericClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_MASTERClass.vr \
./file_vro/PCI_MASTERClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_PARITYClass.vr \
./file_vro/PCI_PARITYClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_READresultClass.vr \
./file_vro/PCI_READresultClass.vro || exit 1
```

```

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_RandomizeClass.vr \
./file_vro/PCI_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCI_TARGETClass.vr \
./file_vro/PCI_TARGETClass.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pci_sys_tst.vr \
./file_vro/pci_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/lstmodel.vr \
./file_vro/lstmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftdmodel.vr \
./file_vro/swiftdmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera/pcimaster_pkg.vr \
./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera/pcislave_pkg.vr \
./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera/pcimonitor_pkg.vr \
./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE Verilog-XL AND RUN THE PCI TESTBENCH #####

unsetenv SSI_LIB_FILES

$CDS_INST_DIR/tools/bin/verilog +loadpli=swiftpli:swift_boot \
`LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcisys_vera_tst_top.v \
./file_vro/pci_sys_tst.vshell \
`LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
`LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vxl.v \
`LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
`LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vxl.v \
`LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
`LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vxl.v \
+incdir+{$LMC_HOME}/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/verilog \
+vera_udf=./vera_local.dl +vera_mload=`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pci_sys_test_top.vr1

```


Example Script to Run the PCIX VERA Testbench with Verilog-XL

```
#!/bin/csh -fx

# This example script setups and executes
# the PCIX VERA testbench for Verilog-XL.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup NC Verilog
setenv CDS_INST_DIR /d/cds_ldv31/solaris251

# Setup LMC_HOME and Licensing.
# Change values as necessary for your network.
# setenv LMC_HOME /synopsys/release
set path = (${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin $path)
setenv LD_LIBRARY_PATH .:${LMC_HOME}/lib/sun4Solaris.lib:${CDS_INST_DIR}/tools/lib:${LD_LIBRARY_PATH}
setenv SNPSLMD_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:26585@grymoire
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:5280@venice:5280@void:26585@grymoire
setenv LMC_LIB_DIR ${LMC_HOME}/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, CDS_INST_DIR, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH, set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp `${LMC_HOME}/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixslave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCIX_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
    -I${VERA_HOME}/lib \
    ${LMC_HOME}/sim/vera/src/vera_user.c

ld -G -z text -o ./vera_local.dl ./vera_user.o \
    ${LMC_LIB_DIR}/vera_slm_pli.o

cc -O -KPIC -c -o ./veriususer.o -I${CDS_INST_DIR}/tools/include \
    -I${CDS_INST_DIR}/tools/inca/include \
    -I${VERA_HOME}/lib/vlog ${VERA_HOME}/lib/vlog/veriususer.c
ld -G -o ./libpli.so ./veriususer.o \
    ${VERA_HOME}/lib/vlog/libSysSciTask.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```

```
##### CREATE *.VRO FILES NEEDED BY THE PCIX FLEXMODELS #####
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_InitClass.vr \
./file_vro/PCIX_InitClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_BURSTClass.vr \
./file_vro/PCIX_BURSTClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DECODEandWAITClass.vr \
./file_vro/PCIX_DECODEandWAITClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_D064BITSClass.vr \
./file_vro/PCIX_D064BITSClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DWORDClass.vr \
./file_vro/PCIX_DWORDClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_GenericClass.vr \
./file_vro/PCIX_GenericClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_MASTERClass.vr \
./file_vro/PCIX_MASTERClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_PARITYClass.vr \
./file_vro/PCIX_PARITYClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_READresultClass.vr \
./file_vro/PCIX_READresultClass.vro || exit 1
```

```

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_RandomizeClass.vr \
./file_vro/PCIX_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_TARGETClass.vr \
./file_vro/PCIX_TARGETClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_SPLITClass.vr \
./file_vro/PCIX_SPLITClass.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcix_sys_tst.vr \
./file_vro/pcix_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/1stmodel.vr \
./file_vro/1stmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftmodel.vr \
./file_vro/swiftmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/src/vera/pcimaster_pkg.vr \
./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vera/pcislave_pkg.vr \
./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vera/pcimonitor_pkg.vr \
./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE Verilog-XL AND RUN THE PCIX TESTBENCH #####

unsetenv SSI_LIB_FILES

$CDS_INST_DIR/tools/bin/verilog +loadpli=swiftpli:swift_boot \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixsys_vera_tst_top.v \
./file_vro/pcix_sys_tst.vshell \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster.v \
`$LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/verilog/pcimaster_fx_vxl.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor.v \
`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/verilog/pcimonitor_fx_vxl.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave.v \
`$LMC_HOME/bin/flexm_setup pcislave_fx`/examples/verilog/pcislave_fx_vxl.v \
+incdir+{$LMC_HOME}/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/verilog \
+vera_udf=./vera_local.dl +vera_mload=`$LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vera/pcix_sys_test_top.vrl

```

Example Script to Run the PCI VERA Testbench with MTI

```
#!/bin/csh -fx

# This example scripts setups and executes the
# VERA PCI testbench with the MTI simulator.

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup MTI
setenv MTI_HOME /d/mti54e/modeltech
set path = (${MTI_HOME}/bin $path)

# Setup LMC_HOME
# Change this value to match your network configuration.
# setenv LMC_HOME /d/ae/work2/test/release
set path = (${LMC_HOME}/bin $path)
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$VERA_HOME/lib/mti:$LD_LIBRARY_PATH
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:1650@venice:1650@void:1650@grymoire
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, MTI_HOME, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH, set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcmonitor_fx`/examples/vera/pcislave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCI_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
    -I$VERA_HOME/lib \
    ${LMC_HOME}/sim/vera/src/vera_user.c

    /usr/ucb/ld -G -o vera_mti.dl \
    ${LMC_LIB_DIR}/vera_slm_mti.o \
    vera_user.o \
    ${VERA_HOME}/lib/mti/mti_bridge.a \
    ${VERA_HOME}/lib/libVERA.a -lsocket -lnsl -lintl -lm -lc -ldl

gcc -fpic -DMTI -DaccVersionLatest -c -I${VERA_HOME}/lib \
    -I${MTI_HOME}/include ${VERA_HOME}/lib/veriususer.c
ld -G -o veriususer.so veriususer.o \
    ${VERA_HOME}/lib/libSysSciTaskPIC.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```

```

##### CREATE *.VRO FILES NEEDED BY THE PCI FLEXMODELS #####

vera -cmp -g -mti -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pci_sys_tst.vr \
./file_vro/pci_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/1stmodel.vr \
./file_vro/1stmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr \
./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftmodel.vr \
./file_vro/swiftmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/src/vera/pcimaster_pkg.vr ./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vera/pcislave_pkg.vr ./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vera/pcimonitor_pkg.vr ./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE MTI AND RUN THE PCI TESTBENCH #####

vlib slm_lib

vmap slm_lib slm_lib

vcom -93 -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
vcom -93 -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
\pcimaster_fx`/src/vhdl/pcimaster_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/src/vhdl/pcimaster_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/examples/vhdl/pcimaster_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimaster_fx`/examples/vhdl/pcimaster_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vhdl/pcislave_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/src/vhdl/pcislave_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/examples/vhdl/pcislave_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcislave_fx`/examples/vhdl/pcislave_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vhdl/pcimonitor_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/src/vhdl/pcimonitor_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vhdl/pcimonitor_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vhdl/pcimonitor_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup \
pcimonitor_fx`/examples/vhdl/pcisys_fx_comp.vhd

```

```
vcom -93 -work slm_lib `$(LMC_HOME)/bin/flexm_setup \  
pcimonitor_fx`/examples/vhdl/pcimonitor.vhd  
vcom -93 -work slm_lib `$(LMC_HOME)/bin/flexm_setup \  
pcislave_fx`/examples/vhdl/pcislave.vhd  
vcom -93 -work slm_lib `$(LMC_HOME)/bin/flexm_setup \  
pcimaster_fx`/examples/vhdl/pcimaster.vhd  
  
vcom -93 -work slm_lib ./file_vro/pcix_sys_tst_shell.vhd  
vcom -93 -work slm_lib `$(LMC_HOME)/bin/flexm_setup \  
pcimonitor_fx`/examples/vera/pcisys_vera_tst_top.vhd  
  
vsim slm_lib.cfgtest
```

Example Script to Run the PCIX VERA Testbench with MTI

```
#!/bin/csh -fx

# References:
# Simulator Configuration Guide
# FlexModel Users Manual
# VERA Users Manual

##### SETUP BASIC ENVIRONMENT #####
# Setup VERA */
setenv VERA_HOME /d/vera50/vera-5.0-solaris2.5
set path = (${VERA_HOME}/bin $path)
# setenv SSI_LIB_FILES ./vera_local.dl

# Setup MTI
setenv MTI_HOME /d/mti54e/modeltech
set path = (${MTI_HOME}/bin $path)

# Setup LMC_HOME based on your network configuration.
# setenv LMC_HOME /d/ae/work2/test/release
set path = (${LMC_HOME}/bin $path)
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$VERA_HOME/lib/mti:$LD_LIBRARY_PATH
setenv LM_LICENSE_FILE 26585@venice:26585@void:26585@grymoire:5300@cougar:1650@venice:1650@void:1650@grymoire
setenv LMC_LIB_DIR $LMC_HOME/lib/sun4Solaris.lib

# Give Yourself or Other Users Feedback
echo " "
echo " The Setup script just initialized the VERA_HOME, MTI_HOME, LMC_HOME,"
echo " "
echo " set LD_LIBRARY_PATH, set SNPSLMD_LICENSE_FILE and set LMC_LIB_DIR "
echo " "
echo " Please verify that you have the correct paths set for each environment variable"
echo " "

##### SETUP TESTBENCH PARAMETERS #####

cp ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixslave* .

# Give Yourself or Other Users Feedback */
echo " "
echo " This part copies three files to the work directory."
echo " "
echo " These three files are used by the load_file commands in the PCIX_InitClass."
echo " "
echo " The load_file command allows you to preload the target's config memory, "
echo " "
echo " I/O memory and MEM memory. These files used by C, VHDL, and Verilog testbenches also."

##### CREATES THE VERA_USER.O AND THE VERA_LOCAL.DL FILES #####

cc -DVERA_UDF -Kpic -c -o ./vera_user.o \
    -I$VERA_HOME/lib \
    ${LMC_HOME}/sim/vera/src/vera_user.c

    /usr/ucb/ld -G -o vera_mti.dl \
    ${LMC_LIB_DIR}/vera_slm_mti.o \
    vera_user.o \
    ${VERA_HOME}/lib/mti/mti_bridge.a \
    ${VERA_HOME}/lib/libVERA.a -lsocket -lnsl -lintl -lm -lc -ldl

gcc -fpic -DMTI -DaccVersionLatest -c -I$VERA_HOME/lib \
    -I${MTI_HOME}/include ${VERA_HOME}/lib/veriususer.c
ld -G -o veriususer.so veriususer.o \
    ${VERA_HOME}/lib/libSysSciTaskPIC.a

# Give Yourself or Other Users Feedback
echo " "
echo " vera_user.o and vera_local.dl files created."
```

```
##### CREATE *.VRO FILES NEEDED BY THE PCIX FLEXMODELS #####
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_InitClass.vr ./file_vro/PCIX_InitClass.vro
|| exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_BURSTClass.vr
./file_vro/PCIX_BURSTClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DECODEandWAITClass.vr
./file_vro/PCIX_DECODEandWAITClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_D064BITSClass.vr
./file_vro/PCIX_D064BITSClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_DWORDClass.vr
./file_vro/PCIX_DWORDClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_GenericClass.vr
./file_vro/PCIX_GenericClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_MASTERClass.vr
./file_vro/PCIX_MASTERClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_PARITYClass.vr
./file_vro/PCIX_PARITYClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_READresultClass.vr
./file_vro/PCIX_READresultClass.vro || exit 1
```

```
vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. `LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_RandomizeClass.vr
```



```

./file_vro/PCIX_RandomizeClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_TARGETClass.vr
./file_vro/PCIX_TARGETClass.vro || exit 1

vera -cmp -g -h -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/class_dir/PCIX_SPLITClass.vr
./file_vro/PCIX_SPLITClass.vro || exit 1

vera -cmp -g -mti -I$LMC_HOME/sim/vera/src \
-I`$LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera \
-I`$LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/ \
-I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcix_sys_tst.vr ./file_vro/pcix_sys_tst.vro || exit 1

vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/lstmodel.vr ./file_vro/lstmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/flexmodel_pkg.vr ./file_vro/flexmodel_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src $LMC_HOME/sim/vera/src/swiftmodel.vr ./file_vro/swiftmodel.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vera/pcimaster_pkg.vr
./file_vro/pcimaster_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcislave_fx`/src/vera/pcislave_pkg.vr
./file_vro/pcislave_pkg.vro || exit 1
vera -cmp -g -I$LMC_HOME/sim/vera/src -I. ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vera/pcimonitor_pkg.vr
./file_vro/pcimonitor_pkg.vro || exit 1

# Give yourself and others feedback. #
echo " "
echo " "
echo " Now that all the Vera files are compiled, You will only need to compile a class "
echo " "
echo " file or the testbench file when you modify it. You do not need to compile all"
echo " "
echo " the files again unless you update the pcix models"
echo " "

##### INVOKE MTI AND RUN THE PCIX TESTBENCH #####

vlib slm_lib

vmap slm_lib slm_lib

vcom -93 -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
vcom -93 -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vhdl/pcimaster_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/src/vhdl/pcimaster_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/vhdl/pcimaster_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/vhdl/pcimaster_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcislave_fx`/src/vhdl/pcislave_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcislave_fx`/src/vhdl/pcislave_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcislave_fx`/examples/vhdl/pcislave_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcislave_fx`/examples/vhdl/pcislave_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vhdl/pcimonitor_user_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/src/vhdl/pcimonitor_pkg.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vhdl/pcimonitor_fx_comp.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vhdl/pcimonitor_fx_mti.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vhdl/pcisys_fx_comp.vhd

vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vhdl/pcimonitor.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcislave_fx`/examples/vhdl/pcislave.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimaster_fx`/examples/vhdl/pcimaster.vhd

vcom -93 -work slm_lib ./file_vro/pcix_sys_tst_shell.vhd
vcom -93 -work slm_lib ` $LMC_HOME/bin/flexm_setup pcimonitor_fx`/examples/vera/pcixsys_vera_tst_top.vhd

vsim slm_lib.cfgtest

```

D

Frequently Asked Questions (FAQ)

This appendix was added in August of 2001. Its purpose is to answer various questions, particularly those involving how Synopsys has interpreted the PCI and PCIX specifications within the pcislave_fx, pcimaster_fx, and the pcimonitor_fx FlexModels.

All PCI and PCIX models

1. Will the slave decode TYPE 1 configuration accesses?

Only IDSEL is necessary for our slave to respond to a config cycle. If the slave is configured to respond to TYPE 1 configs, it will do so when IDSEL is asserted. The slave does not look at bus_num in decoding configuration transactions. This ability to recognize TYPE 1 accesses (as long as IDSEL is asserted and the slave is properly configured) may help simulate and debug master devices that have to talk to PCI Bridge devices.

2. What happens if I try resetting a model once commands have been issued?

The PCI FlexModels were not designed to be reset after commands have been processed or bus activity has commenced. A reset signal received at such a time will put the models in an undefined state.

The pcix init signals (DEVSEL#, TRDY# and STOP#) may be asserted at the time of reset deassertion within the setup and hold parameters defined in the PCIX spec but they will not influence model behavior. To set the models to pcix mode use the pci(master/slave/monitor)_configure commands.

3. What are the constraints on command queue sizes?

The pcimaster_fx and the pcislave_fx models have a queue size of 20k for addr_req, read_cycle, and read_continue commands. The models queue up these commands when they do not have corresponding read_rslt commands. The master/slave will discard the results of addr_req, read_cycle, and read_continue commands without

corresponding read_rslt commands after the 20k limit is reached. The read_rslt queue supports random access by the user (see addr mode and tag mode parameter descriptions). The read_rslt queue is managed as a FIFO to match data discards after the queue limit has been reached. After the queue's limit has been reached, the oldest data will be discarded first (FIFO).

4. How do I find values for timing parameters?

Like all Synopsys FlexModels/SmartModels, the PCI models contain timing information in the .td files within their respective model/src directories. These .td files basically define the timing shell around the model. To understand the contents and structure of the .td files, consult the [SmartModel Library User's Manual](#).

Once you have learned how to read the .td files, you can then find what timing constraints have been set in the .td file. For example, the pcislave_fx.td file contains a parameter called "pwhmin_PCLK_P66MHZ_fast." To find what this pcislave_fx timing parameter defines, look it up in [Appendix A, "Values for timing_param_index Parameter"](#) on page 337. Looking it up, this parameter refers to the following: "Minimum pulse duration of SYSCLOCK high in 66MHz (conventional PCI mode) or 133MHz (PCI-X mode)."

5. What are some important points about Disconnect on ADB?

When two devices interact each other on the bus, both of them can signal to disconnect on ADB regardless of transaction type. The disconnection should be applied for burst Write, burst Read and Split Completion transactions. The disconnection can be applied for DWORD transaction; however, the transaction will be finished in one data phase anyway. The master adjusts the byte count to disconnect the burst Write/Read on ADB if the starting address is three data phases or less from ADB. If the starting address is four or more data phases from ADB, the master will use FRAME# to signal disconnect on ADB. If the transaction is a Split Completion, the master has to use STOP#, TRDY# and DEVSEL# to do that. These scenarios (Burst WR/RD and SC) will be reversed for the Slave.

6. Do the models support Message Signal Interrupts (MSI) in PCI Mode?

No. In the case of the Command Register, you may even write a valid value into bit 10, but none of the models will respond to it. The same is true of the Status Register: bit 3 may contain a 1 or 0 to indicate an Interrupt Status, even though MSI is not supported.

pcislave_fx

1. Will the slave assert STOP to signal disconnect on ADB if the transaction address and byte count is such that the transaction will not cross an ADB?

Yes, but you must use the configuration parameter
PCISLAVE_STOP_ON_1ST_DATA_PHASE to make this happen.

2. I am seeing very odd behavior from the pcislave_fx in my random testing environment. Transactions that the slave used to handle correctly are now getting incorrect results. What might be going on?

Users with testbenches that generate random scenarios often introduce unintended delays. When time passes in a simulation without command input to the pcislave_fx model, the slave may respond to a transaction on the bus before processing intended configuration commands and pcislave_request commands. Be sure to check for and eliminate sources of unintended time delay in your testbench. Recall that it is pcislave_request commands that allow the slave to temporarily ignore the FlexModel command stream while a bus transaction is in progress. Unintended delays in your testbench might leave the slave in an unintended configuration for some transactions.

3. How/where does the slave store data it receives? Does each pcislave instance have unique memory?

The pcislave stores the data it receives over the PCI/X bus or through pcislave_load_file commands in a dynamically allocated memory. This memory space is common to all the pcislaves in a given simulation. Thus, data stored by one pcislave can be seen via pcislave_dump_file commands from any of the pcislaves. This is not an issue for bus operation because the PCI spec requires that each target device be configured to its own memory range(s).

4. How can I use the pcislave_fx for Bridge testing?

The pcislave_fx contains some features that may be helpful in simulating how a master device interacts with a bridge. First, the pcislave_fx may be configured for up to three memory and I/O address ranges. The slave may also be configured to issue split responses. The slave can be configured to respond to both type 0 and 1 configuration write and read commands. In addition, the slave can be configured to issue SCM's typical of a bridge device. Refer to [Table 37 on page 227](#) for a listing of configuration types.

Index

A

access delays, enabling and disabling [200](#),
[284](#), [334](#)
 ADB [228](#), [233](#)
 and 64-bit data transfers [57](#)
 disconnect at [142](#), [143](#), [234](#)
 Addendum [13](#)
 address stepping cycles [144](#)
 alias to memory read block [48](#)
 arbitration [98](#), [129](#), [302](#)
 schemes [98](#), [129](#), [302](#), [303](#)

B

back-to-back transactions [137](#)
 backward-compatibility. *See* compatibility
 behaviors, model [37](#), [74](#), [75](#), [76](#)
 burst cycles [39](#), [100](#), [102](#), [143](#), [164](#), [203](#),
[204](#), [241](#), [269](#)
 transaction examples [41](#)
 bus names [153](#), [156](#), [160](#), [193](#), [253](#), [256](#),
[260](#), [281](#), [310](#), [314](#), [328](#)
 bus number [227](#)
 bus parking [302](#)
 bus trace [61](#), [99](#), [129](#)
 example [62](#)
 generating [61](#)
 bus-functional models [21](#)

C

C mode [22](#)
 cacheline size [137](#), [227](#)
 cacheline status [269](#)
 Capabilities List Item [137](#), [227](#)
 Capabilities Pointer [137](#), [227](#)
 class code [137](#), [227](#)
 command modes [22](#)
 command streams [74](#), [87](#), [99](#), [121](#)
 commands
 examples in PCI-X mode [37](#)

examples, using [106](#)
 pcimaster_fx command reference [135](#)
 pcimaster_fx command summary [134](#)
 pcimonitor_fx command reference [300](#)
 pcimonitor_fx command summary [299](#)
 pcislave_fx command reference [222](#)
 pcislave_fx command summary [220](#)
 reference sections [135](#), [222](#), [300](#)
 summaries of [134](#), [220](#), [299](#)
 synchronization [87](#)
 syntax notation [16](#)
 compatibility
 ensuring [367](#)
 files [368](#)
 installing compatibility files [368](#), [370](#)
 modifying existing testbenches [368](#), [369](#)
 with earlier FlexModel versions [13](#)
 with existing testbenches [13](#), [368](#)
 with PCI Test Suite [13](#), [367](#), [368](#), [371](#)
 within PCI/PCI-X FlexModel set [368](#)
 compatibility files, installing [370](#)
 compiling C command files [78](#), [107](#)
 compiling Verilog packages for simulation
 [81](#), [110](#)
 compiling VHDL packages for simulation
 [80](#), [108](#)
 configuratin types, pcimaster_fx
 PCIMASTER_SPLIT_RETRY_LIMIT
 [144](#)
 configuratin types, pcislave_fx
 PCISLAVE_CAP_POINTER [227](#)
 configuration read [48](#)
 configuration space latency timer [125](#), [126](#)
 configuration types, pcimaster_fx
 PCIMASTER_ADB_DISCONNECT_I
 NCR [137](#)
 PCIMASTER_BACK [137](#)
 PCIMASTER_BUS_NUM [137](#)
 PCIMASTER_C_LINE_SIZE [137](#)
 PCIMASTER_CAP_POINTER [137](#)
 PCIMASTER_CDELAY [137](#)

- PCIMASTER_CL 137
- PCIMASTER_CLS_CODE 137
- PCIMASTER_DES_MAX_CUM_READ_SIZE 138
- PCIMASTER_DES_MAX_READ 138
- PCIMASTER_DES_MAX_SPLIT_TRAN 138
- PCIMASTER_DEV_CNTRL 138
- PCIMASTER_DEV_ID 138
- PCIMASTER_DEV_NUM 139
- PCIMASTER_DISCONNECT_ON_ADB 139
- PCIMASTER_DUAL_AD 139
- PCIMASTER_FUNC_NUM 139
- PCIMASTER_IO_BURST_ADDR 139
- PCIMASTER_MAX_CLK 140
- PCIMASTER_MAX_READ_BC 139
- PCIMASTER_MAX_SPLIT_TRAN 140
- PCIMASTER_MODE 140
- PCIMASTER_NO_SNOOP 140
- PCIMASTER_PAR_ERR_DATA_PHASE 141
- PCIMASTER_PAR_PAR84_SELECT 141
- PCIMASTER_PCI_ERROR 141
- PCIMASTER_PCIX 141
- PCIMASTER_REV_ID 141
- PCIMASTER_RL 141
- PCIMASTER_SAME 142
- PCIMASTER_SECONDARY_BUS_NUM 142
- PCIMASTER_SPLIT_ABORT_LIMIT 143
- PCIMASTER_SPLIT_DECODE 142
- PCIMASTER_SPLIT_DISCONNECT_ON_ADB 143
- PCIMASTER_SPLIT_PCI_ERROR 142
- PCIMASTER_SPLIT_STOP_ASSERT_B4_ADB 142
- PCIMASTER_SPLIT_TRANSFER_LIMIT 143
- PCIMASTER_SPLIT_TRDYN_DELAY 143
- PCIMASTER_START_TAG 144
- PCIMASTER_STEP 144
- PCIMASTER_TL 144
- PCIMASTER_TYPE1_ACCESS 144
- PCIMASTER_VEN_ID 144
- configuration types, pcimonitor_fx
 - PCIMONITOR_ 303
 - PCIMONITOR_ARBITRATE 302
 - PCIMONITOR_ERRORCHECK 302
 - PCIMONITOR_GNT_ 302
 - PCIMONITOR_PARKZERO 302
 - PCIMONITOR_PCIX 302
 - PCIMONITOR_PRIORITY_N 302
 - PCIMONITOR_SPLIT_ 303
- configuration types, pcislave_fx
 - PCISLAVE_ABORT_LIMIT 227
 - PCISLAVE_ADDR_64 227
 - PCISLAVE_BUS_NUM 227
 - PCISLAVE_C_LINE_SIZE 227
 - PCISLAVE_CLS_CODE 227
 - PCISLAVE_DATA_64 227
 - PCISLAVE_DES_MAX_CUM_READ_SIZE 227
 - PCISLAVE_DES_MAX_READ_BC 227
 - PCISLAVE_DES_MAX_SPLIT_TRAN 228
 - PCISLAVE_DEV_ID 228
 - PCISLAVE_DEV_NUM 228
 - PCISLAVE_DISCONNECT_ON_ADB 228
 - PCISLAVE_FUNC_NUM 228
 - PCISLAVE_H_TYPE 228
 - PCISLAVE_INT_ACK 228
 - PCISLAVE_INT_ACK_VECTOR 228
 - PCISLAVE_IO_L_1 228
 - PCISLAVE_IO_L_2 229
 - PCISLAVE_IO_L_O 228
 - PCISLAVE_IO_U_1 229
 - PCISLAVE_MAX_ 229
 - PCISLAVE_MEM_L_0 229
 - PCISLAVE_MEM_L_1 229
 - PCISLAVE_MEM_L_2 229
 - PCISLAVE_MEM_U_0 229
 - PCISLAVE_MEM_U_1 229
 - PCISLAVE_MEM_U_2 229
 - PCISLAVE_PAR_ERR_DATA 230

- PCISLAVE_PAR_PAR64_ 230
- PCISLAVE_PCI_ERROR 230
- PCISLAVE_PCIX 230
- PCISLAVE_REV_ID 230
- PCISLAVE_SC_ERR 230
- PCISLAVE_SC_MSG 231
- PCISLAVE_SCEM_ADDR 231
- PCISLAVE_SCEM_CLASS 232
- PCISLAVE_SCEM_INDEX 232
- PCISLAVE_SPLIT_ 232, 233
- PCISLAVE_SPLIT_ADB_ 231
- PCISLAVE_SPLIT_COMPL_ 233
- PCISLAVE_SPLIT_DISC_ 233
- PCISLAVE_SPLIT_PCI_ 233
- PCISLAVE_START_TAG 233
- PCISLAVE_STOP_ON_1ST_DATA_P
HASE 234
- PCISLAVE_TRANSFER_ 234
- PCISLAVE_VEN_ID 234
- configuration write 48
- configuring
 - pcimaster_fx 145
 - pcimonitor_fx 301, 304
 - pcimonitor_fx for test suite 61
 - pcislave_fx 236
- control pin, pcimonitor_fx 60
- conventional mode
 - error checks 289
- conventional mode system testbench 35, 73
 - bus tracing 99
 - C configuration 90
 - clock 93
 - command synchronization 87
 - configuration details 93
 - error checking 99
 - files 91
 - HDL configuration 89
 - IDSEL assertion 94
 - main test sequence 99
 - model behaviors demonstrated 74, 75, 76
 - operation sequences 99
 - pcimonitor instantiation 98
 - primary master 94
 - primary master sequence 100
 - primary slave 95, 102

- primary slave sequence 102
- pull-ups 93
- secondary master 95, 101
- secondary master sequence 101
- secondary slave 97, 103
- secondary slave sequence 103
- setting up 77
- conventions, typographical and symbol 16

D

- data transfers, 64-bit 57
 - examples 58
- decode speeds 60, 102, 268
 - example 60
- delays 49, 51, 137, 143, 241, 269, 302
 - examples 52
 - See also* wait states
- Designated Maximum Cumulative Read
Size 138, 227
- Designated Maximum Memory Read Byte
Count 138, 227
- Designated Maximum Outstanding Split
Transaction 138, 228
- device ID 138, 228
- disconnect 43, 102
 - in split completions 233
- disconnect at ADB 43, 142, 143
 - examples 43, 44
- disconnect at next ADB 46
 - examples 46
- dual address cycles 59, 102, 139
 - examples 59
- dump command 103
- dump file 244
- dumping memory space contents 244
- DWORD transactions 37
 - examples 38

E

- enabling bus tracing 61
- error check registers 65, 318, 321, 331
- error checking
 - in conventional mode testbench 99

- in PCI-X system testbench [129](#)
- PCI error checks [289](#)
- PCI-X error checks [293](#)
- turning on [302, 332](#)

F

- files
 - in conventional mode system testbench [91](#)
 - in PCI-X system testbench [92, 123, 124](#)
- FLEX commands, global [135, 221, 299](#)
- flex_synchronize command [87](#)
- flexm_setup script [77, 106, 370](#)
- FlexModelId generic/defparameter [60, 61, 94, 95, 97, 98, 125, 126, 127, 129](#)
 - setting for PCI test suite [61](#)
- forcing illegal PCI cycles [141, 142, 230](#)
- forcing illegal PCI-X split completion cycles [233](#)
- function number [139, 228](#)
- functions
 - supported by pcimaster_fx [131](#)
 - supported by pcimonitor_fx [287](#)
 - supported by pcislave_fx [218](#)

G

- generating bus trace file [61](#)
- global FLEX commands [135, 221, 299](#)
- GNT_ASSERTED_CLKS [68](#)
- group timing checks
 - setting [200, 284, 334](#)
 - values for [337, 348, 357](#)

H

- HDL mode [22](#)
- header type [228](#)
- history, model [13](#)

I

- I/O read [48](#)
- I/O space [228](#)
- I/O space mapping [96, 97, 126, 128](#)

- I/O write [48](#)
- IDSEL [94, 122](#)
- illegal PCI cycles, forcing [141, 142, 230](#)
- illegal PCI-X split completion cycles, forcing [233](#)
- individual timing checks
 - setting [200, 284, 334](#)
 - values for [337, 348, 357](#)
- initialization file [102](#)
- instantiating PCI models [36](#)
- Intel NT [78, 107](#)
- interrupt acknowledge [48, 228](#)
- interrupts [22](#)

L

- list file [61, 99, 129](#)
 - example [62](#)
- load file [248](#)
- loading memory space contents [248](#)
- logging bus activity [61](#)
- logging, model [37](#)
- looping constructs [101](#)

M

- manual, overview of chapters [14](#)
- master abort [43](#)
 - example [43](#)
- master terminations [43](#)
- master. *See* pcimaster_fx
- Max Clock Stop time [137](#)
- Maximum Memory Read Byte Count [139, 229](#)
- Maximum Outstanding Split Transaction [140, 233](#)
- MDL version [367, 371](#)
 - for PCI Test Suite [367](#)
- memory initialization file [100, 102](#)
- memory locations
 - pcislave_fx [223](#)
- memory read block [48](#)
- memory read DWORD [48](#)
- memory space [229](#)

- dumping contents of 244
- loading contents 248
- mapping 96, 97, 126, 128
- memory write burst cycle 100
- message levels
 - pcimaster_fx 189, 190
 - pcimonitor_fx 324
 - pcislave_fx 278
 - setting for test suite 61
- migrating
 - from earlier FlexModel versions 13, 367
- model behaviors shown in testbench 37, 74, 75, 76
- model history 13
- model logging 37
- model version 13, 367
- modeling exceptions
 - pcimaster_fx 132
 - pcimonitor_fx 288
 - pcislave_fx 219
- model-specific testbenches 35
 - information provided in 35
- modes
 - C 22
 - command 22
 - HDL 22
 - PCI-X. *See* PCI-X mode
 - Vera 22
- modifying existing testbenches 368, 369
- monitor. *See* pcimonitor_fx
- multiple outstanding split transactions 48, 140

O

- output file 61
 - example 62
- overview of PCI/PCI-X FlexModels 21

P

- package files 80, 81, 106, 108, 110
- parity error generation
 - master and slave 52
- parity errors 52, 101, 141, 142, 230, 233

- examples 56, 57
- PCI
 - error checks 289
 - specification 13, 22
 - warning messages 298
- PCI Test Suite
 - compatibility with PCI/PCI-X FlexModels 367, 368, 371
 - configuring pcimonitor_fx for 61
 - MDL version 367, 371
 - using trace file with 61
- PCI/PCI-X FlexModel set 22
- PCI/PCI-X FlexModels
 - high-level diagram 30
 - instantiating 36
 - model logging 37
 - overview 21
 - troubleshooting 37
- PCIMASTER_DISCONNECT_ON_ADB 139
- PCIMASTER_SECONDARY_BUS_NUM 142
- PCIMASTER_ADB_DISCONNECT_INCREMENT 137
- PCIMASTER_BACK 137
- PCIMASTER_BUS_NUM 137
- PCIMASTER_C_LINE_SIZE 137
- PCIMASTER_CAP_POINTER 137
- PCIMASTER_CDELAY 137
- PCIMASTER_CL 137
- PCIMASTER_CLS_CODE 137
- PCIMASTER_DES_MAX_CUM_READ_SIZE 138
- PCIMASTER_DES_MAX_READ_BC 138
- PCIMASTER_DES_MAX_SPLIT_TRAN 138
- PCIMASTER_DEV_CNTRL_CFG 138
- PCIMASTER_DEV_ID 138
- PCIMASTER_DEV_NUM 139
- PCIMASTER_DUAL_AD 139
- PCIMASTER_FUNC_NUM 139
- pcimaster_fx 131
 - access delays 200

- bus names [153, 156, 160](#)
- command reference [135](#)
- command summary [134](#)
- configuring [145](#)
- message levels [189, 190](#)
- modeling exceptions [132](#)
- pin names [153, 156, 160](#)
- register values [197](#)
- supported functions [131](#)
- termination status registers [183, 185, 186, 187, 188, 197](#)
- timing checks [200](#)
- unsupported features [132](#)
- See also* primary master and secondary master
- PCIMASTER_IO_BURST_ADDR [139](#)
- PCIMASTER_MAX_CLK_P [140](#)
- PCIMASTER_MAX_READ_BC [139](#)
- PCIMASTER_MAX_SPLIT_TRAN [140](#)
- PCIMASTER_MODE [140](#)
- PCIMASTER_MULTIPLE_SPLITS [140](#)
- PCIMASTER_NO_SNOOP [140](#)
- PCIMASTER_PAR_ERR_DATA_PHASE [141](#)
- PCIMASTER_PAR_PAR64_SELECT [53, 141](#)
- PCIMASTER_PCI_ERROR [141](#)
- PCIMASTER_PCIX [141](#)
- PCIMASTER_REV_ID [141](#)
- PCIMASTER_RL [141](#)
- PCIMASTER_SAME [142](#)
- PCIMASTER_SPLIT_ABORT_LIMIT [143](#)
- PCIMASTER_SPLIT_DECODE [142](#)
- PCIMASTER_SPLIT_DISCONNECT_ON_ADB [143](#)
- PCIMASTER_SPLIT_RETRY_LIMIT [144](#)
- PCIMASTER_SPLIT_STOP_ASSERT_B4_ADB [142](#)
- PCIMASTER_SPLIT_TRANSFER_LIMIT [143](#)
- PCIMASTER_SPLIT_TRDYNN_DELAY [143](#)
- PCIMASTER_SPLIT_PCI_ERROR [142](#)
- PCIMASTER_START_TAG [144](#)
- PCIMASTER_STATUS_REG [183, 186, 187, 197, 198](#)
- PCIMASTER_STEP [144](#)
- PCIMASTER_TL [144](#)
- PCIMASTER_TYPE1_ACCESS [144](#)
- PCIMASTER_VEN_ID [144](#)
- pcimonitor instantiation
 - configuration [98, 129](#)
 - settings in conventional mode testbench [98](#)
 - settings in PCI-X system testbench [129](#)
- PCIMONITOR_VALUETRACEFILE [303](#)
- PCIMONITOR_ARBITRATE [302](#)
- PCIMONITOR_ERROR_ENABLE_REG [67, 318, 321, 331](#)
- PCIMONITOR_ERROR_REG [66, 67, 318, 321, 331](#)
- PCIMONITOR_ERRORCHECK [302](#)
- pcimonitor_fx [287](#)
 - access delays [334](#)
 - arbitration [302](#)
 - arbitration schemes [98, 129, 302, 303](#)
 - bus names [310, 314, 328](#)
 - command reference [300](#)
 - command summary [299](#)
 - configuring [301](#)
 - control pin [60](#)
 - custom arbiters [65](#)
 - error check enabling [302, 332](#)
 - error check registers [65, 318, 321, 331](#)
 - errors reported by [289](#)
 - GNT_ASSERTED_CLKS [68](#)
 - list of PCI error checks [289](#)
 - list of PCI-X error checks [293](#)
 - message levels [324](#)
 - modeling exceptions [288](#)
 - pin names [310, 314, 328](#)
 - supported functions [287](#)
 - timing checks [334](#)
 - trace file example [62](#)
 - unsupported features [288](#)
- See also* pcimonitor instantiation

- PCIMONITOR_GNT_
 - ASSERTED_CLKS 302
- PCIMONITOR_PARKZERO 302
- PCIMONITOR_PCI_ERROR_
 - ENABLE_REG 66
- PCIMONITOR_PCI_WARNING_
 - ENABLE_REG 66
- PCIMONITOR_PCI_ERROR_REG 65
- PCIMONITOR_PCI_WARNING_REG 65, 66
- PCIMONITOR_PCIX 302
- PCIMONITOR_PCIX_ERROR_
 - ENABLE_REG 66
- PCIMONITOR_PRIORITY_N 302, 303
- PCIMONITOR_SPLIT_
 - COMP_DELAY_CHECK 303
- PCISLAVE_ABORT_LIMIT 227
- PCISLAVE_ADDR_64 227
- PCISLAVE_BUS_NUM 227
- PCISLAVE_C_LINE_SIZE 227
- PCISLAVE_CAP_POINTER 227
- PCISLAVE_CLS_CODE 227
- PCISLAVE_DATA_64 227
- PCISLAVE_DES_MAX_
 - CUM_READ_SIZE 227
- PCISLAVE_DES_MAX_READ_BC 227
- PCISLAVE_DES_MAX_SPLIT_TRAN 228
- PCISLAVE_DEV_ID 228
- PCISLAVE_DEV_NUM 228
- PCISLAVE_DISCONNECT_ON_ADB 228
- PCISLAVE_FUNC_NUM 228
- pcislave_fx 217
 - access delays 284
 - bus names 193, 253, 256, 260, 281
 - command reference 222
 - command summary 220
 - configuring 236
 - memory locations 223
 - message levels 278
 - modeling exceptions 219
 - pin names 193, 253, 256, 260, 281
 - supported functions 218
 - timing checks 284
 - unsupported features 219
 - See also* primary slave *and* secondary slave
- PCISLAVE_H_TYPE 228
- PCISLAVE_INT_ACK 228
- PCISLAVE_INT_ACK_VECTOR 228
- PCISLAVE_IO_L_0 228
- PCISLAVE_IO_L_1 228
- PCISLAVE_IO_L_2 229
- PCISLAVE_IO_U_0 229
- PCISLAVE_IO_U_1 229
- PCISLAVE_IO_U_2 229
- PCISLAVE_MAX_READ_BC 229
- PCISLAVE_MEM_L_0 229
- PCISLAVE_MEM_L_1 229
- PCISLAVE_MEM_L_2 229
- PCISLAVE_MEM_U_0 229
- PCISLAVE_MEM_U_1 229
- PCISLAVE_MEM_U_2 229
- PCISLAVE_PAR_ERR_DATA_PHASE 230
- PCISLAVE_PAR_PAR64_SELECT 230
- PCISLAVE_PAR_PAR64_SELECT 54
- PCISLAVE_PCI_ERROR 230, 235
- PCISLAVE_PCIX 230
- PCISLAVE_REV_ID 230
- PCISLAVE_SC_ERR 230
- PCISLAVE_SC_MSG 231
- PCISLAVE_SCEM_CLASS 235
- PCISLAVE_SCEM_ADDR 231
- PCISLAVE_SCEM_CLASS 232
- PCISLAVE_SCEM_INDEX 232
- PCISLAVE_SPLIT_RESPONSE 232
- PCISLAVE_SPLIT_TRAN 233
- PCISLAVE_SPLIT_ADB_DISC_INCR 231
- PCISLAVE_SPLIT_COMPL_DELAY 233
- PCISLAVE_SPLIT_DISC_ON_ADB 233
- PCISLAVE_SPLIT_PCI_ERROR 233
- PCISLAVE_START_TAG 233

PCISLAVE_STOP_ON_1ST_DATA_PHASE [234](#)

PCISLAVE_TERMINATION_STYLE [234](#)

PCISLAVE_TRANSFER_LIMIT [234](#)

PCISLAVE_TYPE1_ACCESS [234](#)

PCISLAVE_VEN_ID [234](#)

PCI-X

- command register [139](#), [140](#), [229](#), [233](#)
- configuration space [137](#), [227](#)
- error checks [293](#)
- specification [13](#), [22](#), [33](#)
- status register [138](#), [227](#), [228](#)
- transactions shown in testbench [37](#)

PCI-X mode

- 64-bit data transfers [57](#), [58](#)
- and pcimaster_fx [133](#)
- and pcimonitor_fx [288](#)
- and pcislave_fx [220](#)
- command examples [37](#)
- enabling [141](#), [230](#), [302](#)
- error checks [293](#)
- fast turn around [63](#)
- system testbench. *See* PCI-X system testbench
- turning on [33](#)
- using [33](#)

PCI-X system testbench [35](#), [73](#)

- arbitration [129](#)
- bus tracing [129](#)
- C configuration [120](#)
- clock [121](#)
- configuration details [121](#)
- error checking [129](#)
- files [92](#), [123](#), [124](#)
- HDL configuration [119](#)
- IDSEL assertion [122](#)
- main test sequence [121](#)
- pcimonitor instantiation [129](#)
- primary master [125](#)
- primary slave [126](#)
- pull-ups [122](#)
- secondary master [125](#)
- secondary slave [127](#)
- transactions [37](#)

pin names [153](#), [156](#), [160](#), [193](#), [253](#), [256](#), [260](#), [281](#), [310](#), [314](#), [328](#)

Preface [13](#)

primary master [94](#), [100](#), [125](#)

- configuration [94](#), [125](#)
- sequence in conventional mode testbench [100](#)
- settings in conventional mode testbench [94](#)
- settings in PCI-X system testbench [125](#)

primary slave [95](#), [102](#), [126](#)

- configuration [95](#), [96](#), [126](#)
- I/O space mapping [96](#), [97](#), [126](#), [128](#)
- memory space mapping [96](#), [126](#)
- sequence in conventional mode testbench [102](#)
- settings in conventional mode testbench [95](#)
- settings in PCI-X system testbench [126](#)

protocol

- PCI [13](#), [22](#)
- PCI-X [13](#), [22](#), [33](#)

R

registers

- error check [65](#), [318](#), [321](#), [331](#)
- pcimaster_fx [183](#), [185](#), [186](#), [187](#), [188](#), [197](#)
- PCIMASTER_STATUS_REG [183](#), [186](#), [187](#), [197](#), [198](#)
- PCIMONITOR_ERROR_ENABLE_REG [67](#), [318](#), [321](#), [331](#)
- PCIMONITOR_ERROR_REG [66](#), [67](#), [318](#), [321](#), [331](#)
- pcimonitor_fx [65](#), [318](#), [321](#), [331](#)
- PCIMONITOR_PCI_ERROR_ENABLE_REG [66](#)
- PCIMONITOR_PCI_ERROR_REG [65](#)
- PCIMONITOR_PCI_WARNING_ENABLE_REG [66](#)
- PCIMONITOR_PCI_WARNING_REG [65](#), [66](#)
- PCIMONITOR_PCIX_ERROR_ENABLE_REG [66](#)

- termination status [183](#), [185](#), [186](#), [187](#),
[188](#), [197](#)
- ret_val [101](#)
- retry [47](#), [234](#)
 - example [47](#)
 - limit [47](#), [141](#)
- revision ID [141](#), [230](#)

S

- secondary bus number [142](#)
- secondary master [95](#), [101](#), [125](#)
 - configuration [95](#), [125](#)
 - sequence in conventional mode testbench
[101](#)
 - settings in conventional mode testbench
[95](#)
 - settings in PCI-X system testbench [125](#)
- secondary slave [97](#), [103](#), [127](#)
 - configuration [97](#), [127](#)
 - memory space mapping [97](#), [128](#)
 - sequence in conventional mode testbench
[103](#)
 - settings in conventional mode testbench
[97](#)
 - settings in PCI-X system testbench [127](#)
- setting the TimingVersion generic/
defparameter [32](#), [34](#)
- simulating your design [81](#), [111](#)
- simulator, setting up [32](#)
- single data phase disconnect [47](#), [234](#)
 - example [47](#)
- slave. *See* pcislave_fx
- specification
 - PCI [13](#), [22](#)
 - PCI-X Addendum [13](#), [22](#), [33](#)
- split completion cycles [48](#), [181](#), [233](#)
 - alias to memory read block [48](#)
 - configuration read [48](#)
 - configuration write [48](#)
 - delays [49](#)
 - examples [50](#)
 - I/O read [48](#)
 - I/O write [48](#)
 - interrupt acknowledge [48](#)

- memory read block [48](#)
- memory read DWORD [48](#)
- order [49](#)
- outstanding [49](#)
- pcimaster_fx [48](#)
- split response [45](#), [48](#), [49](#), [232](#), [233](#)
- Split Completion Error Types [235](#)
- split transactions [48](#)
- standalone testbenches. *See* model-specific
testbenches
- standard
 - PCI [13](#), [22](#)
 - PCI-X [13](#), [22](#), [33](#)
- start tag [125](#), [126](#), [144](#), [233](#)
- symbol conventions [16](#)
- sync_label [100](#), [101](#)
- sync_tag parameters [87](#)
- synchronization, commands [87](#)
- system-generated text notation [16](#)

T

- target abort [47](#), [143](#), [227](#)
 - example [47](#)
- target disconnect [100](#)
- target terminations [45](#)
- terminations [228](#), [232](#), [234](#)
 - master [43](#)
 - split response [45](#)
 - target [45](#)
- terminology [17](#)
- test suite. *See* PCI Test Suite
- testbenches
 - model-specific. *See* model-specific
testbenches
 - system. *See* conventional mode system
testbench *or* PCI-X system
testbench
- time-out limit [144](#)
- timing [32](#), [34](#), [36](#)
- timing checks
 - enabling and disabling [200](#), [284](#), [334](#)
 - group [200](#), [284](#), [334](#), [337](#), [348](#), [357](#)
 - individual [200](#), [284](#), [334](#), [337](#), [348](#), [357](#)

TimingVersion generic/defparameter,
 setting [32](#), [34](#), [133](#), [220](#), [288](#), [369](#)
trace file [61](#), [99](#), [129](#)
 example [62](#)
 generating [61](#)
transfers, 64-bit data [57](#)
transition to PCI/PCI-X FlexModels [367](#)
troubleshooting [37](#)
type 1 configuration cycles [144](#), [234](#)
typographical conventions [16](#)

U

UNIX prompt notation [16](#)
unsupported features
 pcimaster_fx [132](#)
 pcimonitor_fx [288](#)
 pcislave_fx [219](#)
upper address value [139](#)
user input notation [16](#)
user-defined timing files [36](#)

V

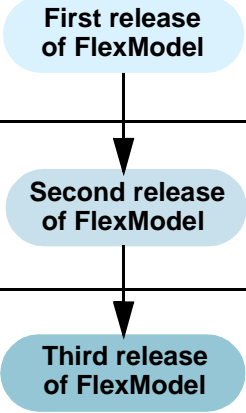
variables, notation for [16](#)
vendor ID [144](#), [234](#)
Vera mode [22](#)
version
 MDL [367](#), [371](#)
 model [13](#), [367](#)
 PCI Test Suite [367](#), [371](#)

W

wait states [49](#), [51](#)
 examples [52](#)
 issuing with configure_delay command
 [52](#)
 issuing with request command [52](#)
 See also delays
write burst cycles [100](#), [101](#), [102](#)

Addendum: History and Version Information

This addendum specifies the current model version and provides history about significant changes that have occurred to the model during the past year. This information is provided in a format common to all SmartModel datasheets. Note that in the current release of the SmartModel Library, model versions are referred to as MDL versions. You can find the MDL version of the model on the next page of this addendum, along with the bibliographic source for the device modeled and any applicable model history.

	MDL Version 01001
	MDL Version 01002
	MDL Version 01003



Attention

This addendum contains information about *fixed* problems, enhancements, and known problems at the time of the release.

1 pcislave_fx

- MDL Version: **01071**
 - Title: **32bit_and_64bit_33Mhz_66Mhz_PCI_and_66Mhz_133Mhz_**
 - Date: **28-Jun-2002**
 - Function: **bus**
 - Subfunction: **system**
 - Base Part Number(s): **pcislave**
-

1.1 SUPPORTED COMPONENTS AND DEVICES

Peripheral Component Interface		
Component Name	Device Name	Source(s)
CONVENTIONAL	pcislave	1
PCIX	pcixslave	2

1.2 SOURCES

Sources cited in the “SUPPORTED COMPONENTS AND DEVICES” section were used as reference(s) for behavioral and timing characteristics in the development of this model. Any other sources listed below were used only as references for device behavior.

1. Peripheral Component Interface “PCI Local Bus Specification Rev 2.2” June 1995
2. Peripheral Component Interface “PCI-X Addendum to the PCI Local Bus Specification” June 1999

1.3 PCIMASTER_FX KNOWN PROBLEMS

There are no known problems at the time of this release.

1.4 PCISLAVE_FX KNOWN PROBLEMS

There are no known problems at the time of this release.

1.5 PCIMONITOR_FX KNOWN PROBLEMS

There are no known problems at the time of this release.

1.6 PCISLAVE_FX MODEL HISTORY

This section documents significant model changes that have occurred in the past year. Not all model changes are significant, so there isn't a model history entry for every model MDL version change. For example, editorial changes to this datasheet will cause the model's MDL version to increment but will have no affect on the model's behavior.

If you see gaps in the model MDL version history or a difference between the current version of this model (as shown on the title banner at the top of this datasheet) and the most recent MDL version listed in the following model history, this means that there were one or more MDL version number changes that did not affect the model's interface or behavior.

For more information about model history, refer to the [*SmartModel Library User's Manual*](#).

- Reference:: 54939
 - MDL Version:: 01071
 - MDL Date:: 28-Jun-2002
 - SRC Version:: v3.59
 - Problem:: The pcislave_set_pin command was not driving the TRDY# pin when the bus was idle.
 - Resolution:: The pcislave_set_pin command correctly drives TRDY# when the bus is idle.
-

- Reference:: 54935
- MDL Version:: 01071
- MDL Date:: 28-Jun-2002
- SRC Version:: v3.59
- Problem:: The pcislave_fx FlexModel was asserting STOP# and DEVSEL# at the same time under the following conditions: 1) A locked state had been established in the model; 2) A different master was trying to interact with the model; and, 3) The decode speed of the slave device was B, C, or Subtractive.

- Resolution:: The problem has been fixed. The model no longer incorrectly asserts STOP# and DEVSEL# at the same time.

-
- Reference:: 54888
 - MDL Version:: 01070
 - MDL Date:: 24-May-2002
 - SRC Version:: v3.58

- Problem:: The pcislave_fx FlexModel did not discard a Split Completion transaction under the following conditions: 1) The model was configured to disconnect the Split Completion on ADB; 2) The transaction was terminated with a Target-Abort on the data phase just before the ADB.

- Resolution:: The model has been corrected.

-
- Reference:: 54873
 - MDL Version:: 01070
 - MDL Date:: 24-May-2002
 - SRC Version:: v3.58

- Problem:: While in PCI mode, a user could not inject a wrong PAR on the second data phase by setting the PCISLAVE_PAR_ERR_DATA_PHASE configuration parameter with pcislave_configure.

- Resolution:: The user can now properly inject PAR on the second data phase by using the PCISLAVE_PAR_ERR_DATA_PHASE configuration parameter.

-
- Reference:: 54851
 - MDL Version:: 01070
 - MDL Date:: 24-May-2002
 - SRC Version:: v3.58

- Problem:: The pcislave_fx model did not actively drive the AD[31:0], CXBE[3:0], PAR, and PAR64 signals when GNT# was parked on the device and the bus was idle.

- Resolution:: The model now drives an idle bus correctly. Note also, as a result of this problem, PCIX Error 94 is now turned on by default to recognize this error condition.

-
- Reference:: 54775
 - MDL Version:: 01069
 - MDL Date:: 26-Apr-2002
 - SRC Version:: v3.57
 - Problem:: The pcislave_fx FlexModel did not return a Split Completion under the following conditions: 1) The model issued a two short (less than two data phases) transactions, and both of them are terminated with a Split Response; 2) The GNT# signal was parked on the device.
 - Resolution:: The model was corrected.

-
- Reference:: 54763
 - MDL Version:: 01069
 - MDL Date:: 26-Apr-2002
 - SRC Version:: v3.57
 - Problem:: The pcislave_fx FlexModel issued a wrong address under the following conditions: 1) The pcimaster issued a 64-64 burst read transaction with a starting address[2] = 1; 2) The model terminated the transaction with a Split Response; 3) When the model returned the Split Completion, the master responded with wait state (TRDY# delay); 4) The master issued another burst read with two data phases crossing the first ADB, and terminated with a Split Response; 5) The model returned the Split Completion and then disconnected the transaction on the first ADB; and, 6) The model continued the Split Completion and the master terminated the transaction with a Retry.
 - Resolution:: The model was corrected.

-
- Reference:: 54751
 - MDL Version:: 01069
 - MDL Date:: 26-Apr-2002

- SRC Version:: v3.57
 - Problem:: The pcislave_fx model did not assert TRDY# under the following conditions: 1) The decode speed was set to 4 in the request command for a Master-Abort termination; and, 2) The decode speed was set to 0 in the request command.
 - Resolution:: The model was corrected.
-

- Reference:: 54694
 - MDL Version:: 01068
 - MDL Date:: 25-Mar-2002
 - SRC Version:: v3.56
 - Problem:: The pcislave_fx would adjust the byte count and set the BCM bit for Split Completion transactions that would have crossed the slave's programmed memory boundary. According to the PCI-X specification, the byte count can be modified only when the slave intends to disconnect the transaction at the first ADB. The slave is required to adjust the byte count only when that ADB is less than four data phases from the start of the transaction.
 - Resolution:: The model has been corrected to only adjust the byte count and set the BCM bit for split completion transactions that start less than four data phases from an ADB that the slave intends to disconnect on.
-

- Reference:: 54681
 - MDL Version:: 01068
 - MDL Date:: 25-Mar-2002
 - SRC Version:: v3.56
 - Problem:: The pcislave_fx model terminated Split Completions one clock cycle too late under the following conditions: 1) Two 64-bit masters issue back-to-back read transactions; 2) The models were configured to respond as a 64-bit and 32-bit devices; 3) The latter device (the 32-bit device) was configured to return first.
 - Resolution:: The model has been corrected.
-

- Reference:: 54496
 - MDL Version:: 01068
 - MDL Date:: 25-Mar-2002
 - SRC Version:: v3.56
 - Problem:: The pcislave_fx model could not be configured to disconnect at all ADB's of a transaction.
 - Resolution:: A new ctype called PCISLAVE_SPLIT_ADB_DISC_INCR has been added which allows users to disconnect on all ADB's of a transaction. Consult the section on the pcislave_fx in the datasheet for additional information.
-

- Reference:: 54166
 - MDL Version:: 01068
 - MDL Date:: 25-Mar-2002
 - SRC Version:: v3.56
 - Problem:: The pcislave_fx did not make optimal use of memory function calls to its internal memory during bus transactions that called for a read-modify-write to the slave's memory.
 - Resolution:: The model has been enhanced to make fewer calls to internal memory for read-modify-write situations.
-

- Reference:: 54661
- MDL Version:: 01067
- MDL Date:: 26-Feb-2002
- SRC Version:: v3.55
- Problem:: In PCI mode, the model would incorrectly assert SERR# to signal an address phase parity error under the following conditions: 1) The model was configured to assert SERR# (parity error emulation) for the address phase of the previous transaction. 2) The model was configured to assert PERR# (parity error emulation) for a data phase of the current transaction. 3) The model was configured for a slow decode speed on the current transaction.

- Resolution:: The model has been corrected.

-
- Reference:: 54165
 - MDL Version:: 01067
 - MDL Date:: 26-Feb-2002
 - SRC Version:: v3.55
 - Problem:: The model performed only 32-bit operations to internal memory. This was not optimal in the case of 64-bit Memory Reads and Writes.
 - Resolution:: The model has been changed to perform 64 bit operations to internal memory when it receives a 64-bit Memory Read or Write operation from the bus.

-
- Reference:: 54651
 - MDL Version:: 01067
 - MDL Date:: 26-Feb-2002
 - SRC Version:: v3.55
 - Problem:: The pcislave_fx model was not capable of independently controlling the PAR or PAR64 signals when injecting errors on those signals.
 - Resolution:: The model has been enhanced with a new configuration ctype called PCISLAVE_PAR_PAR64_SELECT. Consult the datasheet on how to use this new configuration type.

-
- Reference:: 54569
 - MDL Version:: 01067
 - MDL Date:: 26-Feb-2002
 - SRC Version:: v3.55
 - Problem:: The pcislave_fx FlexModel did not discard a Split Completion transaction under the following conditions: 1) the Requester signaled disconnect on ADB; and, 2) the Requester signaled target-abort on the Split Completion just before an ADB.

- Resolution:: The model has been fixed.

-
- Reference:: 54559
 - MDL Version:: 01067
 - MDL Date:: 26-Feb-2002
 - SRC Version:: v3.55
 - Problem:: The pcislave_fx model did not issue target-abort as is required for an I/O transaction with an illegal combination of AD[1:0] and byte enables. Instead, the model issued target-abort on a subsequent transaction.
 - Resolution:: The model has been fixed.

-
- Reference:: 54463
 - MDL Version:: 01066
 - MDL Date:: 29-Jan-2002
 - SRC Version:: v3.54
 - Problem:: When the model received a pcislave_dump_file command with a starting address such that the address as it appears in the dump file would both exceed 32'hfffffff and contain a zero in bits [31:28], then the model would omit that 0 in the address seen in the dump file. Note that addresses in the dump file are right shifted two positions from the address given in the pcislave_dump_file command.
 - Resolution:: The model has been corrected.

-
- Reference:: 54478
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50

- Problem:: The pcislave_fx model incorrectly asserted the SERR# signal for more than one cycle for the data phase of a Special Cycle.
 - Resolution:: The model has been fixed.
-

- Reference:: 54377
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: The pcislave_fx model did not properly assert the PERR# signal under the following conditions: 1) A master drove a wrong PAR value so that the model could assert PERR#; 2) The model was configured to generated parity errors by setting the PCISLAVE_PAR_DATA_PHASE configuration parameters. This is the same STAR as 54404.
 - Resolution:: The model has been fixed.
-

- Reference:: 54373
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: A user could not configure the pcislave_fx FlexModel to assert the STOP# signal to disconnect on ADB even though the transaction would not be able to reach ADB.
 - Resolution:: The model has been enhanced with a new parameter called PCISLAVE_STOP_ON_1ST_DATA_PHASE) so that the user can force the model to assert STOP# on the first data phase of a transaction. This parameter overwrites the PCISLAVE_STOP_ASSERT_B4_ADB parameter. Consult Chapter 6 of the datasheet for additional information on PCISLAVE_STOP_ON_1ST_DATA_PHASE.
-

- Reference:: 54409
- MDL Version:: 01064

- MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: The pcislave_fx model did not “dequeue” a Split transaction under the following conditions: 1) The model terminated a burst read with a Split Response; 2) The returned Split Completion was terminated with Disconnect on ADB; 3) The number of remaining data phases before the completion of the Split transaction was one; and, 4) The master asserted STOP# when the slave tried to finish the last data phase of the split completion.
 - Resolution:: The model was fixed.
-

- Reference:: 54408
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: In PCI mode for fast back-to-back transactions, the pcislave_fx model was not driving the PERR# signal for the last write data phase when the next fast back-to-back transaction was a read transaction.
 - Resolution:: The model has been fixed.
-

- Reference:: 54404
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: The pcislave_fx model did not properly assert the PERR# signal under the following conditions: 1) A master created a data phase with bad parity (bad PAR value); 2) The model was configured to generate a parity error signal (PERR), even if the parity was good, for that data phase. This is the same STAR as 54377.
 - Resolution:: The model has been fixed.
-

- Reference:: 54495
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: The pcislave_fx model did not drive the PERR# signal for the last data phase under the following conditions: 1) The master issued a block write transaction; 2) The model asserted STOP# to disconnect on ADB; and, 3) The model was configured to abort (Target-Abort) the transaction before or just after the ADB disconnection.
 - Resolution:: The model was fixed.
-

- Reference:: 54321
 - MDL Version:: 01064
 - MDL Date:: 24-Jan-2002
 - SRC Version:: v3.50
 - Problem:: Simulators were crashing on Linux systems. The problem was due to name collision problems with a model's object file and other simulator files. A common symptom of this problem was for the simulator to issue a message about "Bad pointer access" when it halted.
 - Resolution:: Name collision problems have been fixed on Linux platforms which were causing simulator crashes.
-

- Reference:: 54489
- MDL Version:: 01063
- MDL Date:: 14-Jan-2002
- SRC Version:: v3.49
- Problem:: Internal testing detected a fatal error on Linux platforms due to a memory management problems.
- Resolution:: The model has been corrected.

-
- Reference:: 54370
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The following changes occurred in the datasheets: 1) table added listing out all model pins and their purpose; 2) pcislave_fx ctype list updated with new parameters for SCEM functionality; 3) new pcimonitor_fx error messages added; 4) modeltype references deleted in the PCI test suite documentation; 5) information on command queues added to FAQ and command reference pages; 6) new test scenarios added in PCI test suite documentation for the purpose of testing bridge-like devices; 7) PCIX Error Messages now reference appropriate section of the PCIX Specification.
 - Resolution:: Documentation has been updated.
-

- Reference:: 54343
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: In PCI-X mode, devices checking parity should only do so when the data is transferred. The pcislave_fx was checking parity (and potentially driving PERR) on data phases that it had signaled target abort.
 - Resolution:: The model will no longer check parity for data phases in which it has signaled Target Abort.
-

- Reference:: 54319
- MDL Version:: 01062
- MDL Date:: 19-Dec-2001
- SRC Version:: v3.48

- Problem:: The pcislave_fx did not terminate a Split Completion transaction under the following conditions:
 - A master issued a read transaction two data phases from ADB and terminated with a Split Response
 - Split Completion was terminated with a Retry
 - Resolution:: The model was corrected for this problem.
-

- Reference:: 54164
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The memory management code in the model contained redundant function calls, possibly effecting performance.
 - Resolution:: Redundant function calls in the memory management code have been removed.
-

- Reference:: 54339
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The model would not recognize the expression PCISLAVE_SCEM_ADDR in VHDL, VERA, or C code.
 - Resolution:: The wrappers have been updated to recognize PCISLAVE_SCEM_ADDR in VHDL, VERA, and C code.
-

- Reference:: 54368

- MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The pcislave_fx FlexModel did not allow users adequate control over Split Completion messages while using the configuration parameters PCISLAVE_SCEM_ADDR, PCISLAVE_SCEM_INDEX, and PCISLAVE_SCEM_CLASS.
 - Resolution:: To give users adequate control over Split Completion messages, Synopsys has created for pcislave_fx two new configuration types. These configuration types are called PCISLAVE_SC_MSG and PCISLAVE_SC_ERR. When a customer uses the PCISLAVE_SC_MSG option, the model will return only a Split Completion message regardless which transaction was terminated with Split Response. The PCISLAVE_SC_ERR will allow a user to turn on/off the Split Completion Error bit. The PCISLAVE_SC_ERR is only used when PCISLAVE_SC_MSG is on. The parameters PCISLAVE_SCEM_ADDR, PCISLAVE_SCEM_INDEX, and PCISLAVE_SCEM_CLASS are still functional, although their use is discouraged. They will no longer be listed in the datasheet.
-

- Reference:: 54337
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The pcislave_fx FlexModel did not allow users adequate control over Split Completion messages while using the configuration parameters PCISLAVE_SCEM_ADDR, PCISLAVE_SCEM_INDEX, and PCISLAVE_SCEM_CLASS.
 - Resolution:: To give users adequate control over Split Completion messages, Synopsys has created for pcislave_fx two new configuration types. These configuration types are called PCISLAVE_SC_MSG and PCISLAVE_SC_ERR. When a customer uses the PCISLAVE_SC_MSG option, the model will return only a Split Completion message regardless which transaction was terminated with Split Response. The PCISLAVE_SC_ERR will allow a user to turn on/off the Split Completion Error bit. The PCISLAVE_SC_ERR is only used when PCISLAVE_SC_MSG is on. The parameters PCISLAVE_SCEM_ADDR, PCISLAVE_SCEM_INDEX, and PCISLAVE_SCEM_CLASS are still functional, although their use is discouraged. They will no longer be listed in the datasheet.
-

- Reference:: 54025

- MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The pcislave_dump_file command took a long time to execute when large spans of memory were dumped.
 - Resolution:: The speed of the pcislave_dump_file command has been improved.
-

- Reference:: 53886
 - MDL Version:: 01062
 - MDL Date:: 19-Dec-2001
 - SRC Version:: v3.48
 - Problem:: The memory management code in the pcislave_fx was incompatible with VCS version 6.0 running on Microsoft Windows NT platform. This was causing program crashes.
 - Resolution:: The pcislave_fx memory management code was changed to prevent program crashes when used with VCS version 6.0.
-

- Reference:: 54225
 - MDL Version:: 01061
 - MDL Date:: 30-Nov-2001
 - SRC Version:: v3.47
 - Problem:: PCI Documentation Changes: 1) all references to file_gen have been removed from the Test Suite Manual with concurrent reorganization of the appropriate section; 2) Tables on parity differences between master and slave updated; 3) PCI errors added under pcimonitor_fx section; 4) Known Problems section moved into the "addendum.pdf" or History section at the rear of the datasheet; 5) Links added from Test Suite Manual to web-based versions of the datasheets
 - Resolution:: Documentation has been updated.
-

- Reference:: 54222
 - MDL Version:: 01061
 - MDL Date:: 30-Nov-2001
 - SRC Version:: v3.47
 - Problem:: The pcislave_fx model terminated a Split Completion early (before a data transfer) under the following conditions:
 - Issued a Split Completion that terminated at an ADB with data remaining to be transferred
 - Bus was parked on the model. When the Split Completion continued, the transaction terminated early without transferring data. The model then re-issued the continued Split Completion a second time. This transaction completed successfully. After the second try, all the data was successfully transferred.
 - Resolution:: The model was corrected for this problem.
-

- Reference:: 54169
- MDL Version:: 01061
- MDL Date:: 30-Nov-2001
- SRC Version:: v3.47
- Problem:: The model violated the PCI-X protocol by hanging the bus with FRAME# and IRDY# asserted under the following conditions:
 - The model was configured for a Split Completion
 - The model was configured to Disconnect on the Second ADB
 - A 64-bit to 64-bit memory read command was issued with a starting address one 32-bit data phase from an ADB; that is, 32'hXXXXXXXXC
 - When the Split Completion was executed, the master device signaled a Disconnect on ADB on the first data phase.

- Resolution:: The model has been corrected for this problem.

-
- Reference:: 54146

- MDL Version:: 01061

- MDL Date:: 30-Nov-2001

- SRC Version:: v3.47

- Problem:: The pcislave_fx incorrectly issued a target abort message when a transaction that was one data phase from a boundary was terminated with a single data phase disconnect.

- Resolution:: The model has been fixed for this problem.

-
- Reference:: 54145

- MDL Version:: 01061

- MDL Date:: 30-Nov-2001

- SRC Version:: v3.47

- Problem:: The model did not take out of the queue a Split Completion that came within one data phase of the absolute upper memory limit of 64'hFFFFFFFFFFFFFFFF. Because this situation was unexpected, the model instead attempted to continue the transaction in a subsequent Split Completion, which typically receives a Master Abort action.

- Resolution:: The model has been corrected for this problem.

-
- Reference:: 54140

- MDL Version:: 01061

- MDL Date:: 30-Nov-2001

- SRC Version:: v3.47

- Problem:: The pcislave_fx model did not issue a split completion error message (SCEM) or process any further transactions after receiving a command under the following conditions:

- A Split Completion was issued, which crossed a memory boundary – either user-specified or the upper PCI boundary of physical memory 2. The Split Completion received a Disconnect at ADB termination

3) The ADB coincided with the memory boundary

- Resolution:: The model was corrected for the listed problems.

-
- Reference:: 54130

- MDL Version:: 01060

- MDL Date:: 30-Oct-2001

- SRC Version:: v3.46

- Problem:: Documentation Changes: 1) Datasheet now has diagram of PCI model pins (PCI DATASHEET AND PCI TEST SUITE MANUALS); 2) Parity error descriptions improved (PCI DATASHEETS); 3) IDSEL usage clarified (PCI DATASHEETS); 4) Added for user hints and tips on how to debug issues flagged by the PCI Test Suite (PCI TEST SUITE MANUAL).

- Resolution:: Documentation updates.

-
- Reference:: 54124

- MDL Version:: 01060

- MDL Date:: 30-Oct-2001

- SRC Version:: v3.46

- Problem:: TThe pcislave_fx model was issuing incorrect data.

The data appropriate for the second data phase of a Split Completion was issued during both the first and second data phases when the following problems occurred:

- The model was configured to Disconnect at ADB
- The model received a Retry twice to a Split Completion

- The transaction began one data phase away from the intended ADB for disconnect
 - Resolution:: The model was corrected for the listed problems.
-

- Reference:: 54117
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: When the model received a Target Abort to a Split Completion, the model would repeat the address and attribute information for the discarded Split Completion at the next Split Completion.
 - Resolution:: The model was corrected.
-

- Reference:: 54077
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would issue incorrect data. The data appropriate for the second data phase of a Split Completion would be issued during both the first and second data phases, when the following occurred: 1) The model was configured to Disconnect at ADB; and, 2) The model received Retry twice to a Split Completion; and 3) The transaction began one data phase away from the intended ADB for disconnect. This is the same problem as in STAR 54124.
 - Resolution:: The model was corrected.
-

- Reference:: 54076
- MDL Version:: 01060
- MDL Date:: 30-Oct-2001
- SRC Version:: v3.46

- Problem:: The slave would incorrectly issue the message: "FlexModelId: '3'. Generated burst memory address out of boundary of defined memory address space – target abort next access" under the following conditions: 1) The upper memory boundary of the model was configured to the PCI limit of 64'hFFFFFFFFFFFFFFFF; 2) A 64_64 read transaction was issued to the model that crossed the memory boundary. The message and the subsequent Target Abort signal were each issued one cycle too early.
 - Resolution:: The model was corrected.
-

- Reference:: 54068
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would incorrectly issue the message: "FlexModelId: '3'. Generated burst memory address out of boundary of defined memory address space – target abort next access" under the following conditions: 1) A transaction was issued to the slave that would cross a defined memory boundary; 2) The memory boundary was an ADB, and the slave was configured to disconnect at that ADB.
 - Resolution:: The model now accounts for disconnects on ADB when evaluating memory boundary target aborts.
-

- Reference:: 54062
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would mistakenly create a Split Completion Error Message under the following conditions: 1) Two memory spaces were set up to be contiguous (that is, PCISLAVE_MEM_L_1 = PCISLAVE_MEM_U_0 + 1); 2) A Split Completion Read crossed that boundary.
 - Resolution:: The model was corrected.
-

- Reference:: 54055
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: If the slave received a Master Abort termination to a Split Completion while the bus was parked on the slave, the slave would repeat the Split Completion rather than discard it. Same issue as STAR 54015.
 - Resolution:: The model was corrected.
-

- Reference:: 54050
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The PCISLAVE_PCI_ERROR configuration parameter can be used to cause the model to assert PERR during write transactions, even though there was not a parity error. Previously, it was only possible to select which data phase the model would assert PERR. PERR would stay asserted until the end of the transaction.
 - Resolution:: The model has been enhanced to use the PCISLAVE_PAR_ERR_DATA_PHASE parameter in conjunction with PCISLAVE_PCI_ERR to select a single data phase for PERR assertion.
-

- Reference:: 54034
- MDL Version:: 01060
- MDL Date:: 30-Oct-2001
- SRC Version:: v3.46
- Problem:: The model signaled Disconnect at ADB one cycle too early under the following conditions:
1) PCISLAVE_STOP_ASSERT_B4_ADB was set to 31; 2) a 32_32 transaction was issued starting one data phase from an ADB. The model would assert STOP for 32 data phases instead of 31.

- Resolution:: The model was corrected.

-
- Reference:: 54024
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would signal Disconnect at ADB one ADB too late under the following conditions: 1) The model was configured to disconnect on ADB; and, 2) a 64-64 bit transaction began one cycle from an ADB. Same as STAR 54018.
 - Resolution:: The model was corrected.

-
- Reference:: 54018
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would signal Disconnect at ADB one ADB too late under the following conditions: 1) The model was configured to disconnect on ADB; and, 2) A 64-64 bit transaction began one cycle from an ADB. Same as STAR 54024.
 - Resolution:: The model was corrected.

-
- Reference:: 53994
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The pcislave_fx FlexModel was not be able to signal Disconnect on ADB when the model was configured through the STOP_ASSERT_B4_ADB parameter to signal Disconnect on ADB 16 data phases before ADB (for 64 bit), or 32 data phases away from ADB (for 32 bit).

- Resolution:: The model was corrected.
-

- Reference:: 53983
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would accept lower memory bounds that were not double QWORD aligned and lower IO bounds that were not DWORD aligned as is required by the PCI spec (see 6.2.5.1 of PCI2.2). This would cause the model to ignore (Master Abort seen on the bus) subsequent non QWORD aligned Memory transactions or non DWORD aligned IO transactions that were issued with a starting address within one QWORD/DWORD respectively of the lower memory bound.
 - Resolution:: The model now issues a warning if lower memory bounds do not follow the PCI spec: "WARNING: Address is not double QWORD aligned (see 6.2.5.1 in PCI spec). Discarding lower four bits of mem_lower_0." The lower memory bound is then set up correctly.
-

- Reference:: 53675
 - MDL Version:: 01060
 - MDL Date:: 30-Oct-2001
 - SRC Version:: v3.46
 - Problem:: The model would incorrectly issue the message: "FlexModelId: '3'. Generated burst memory address out of boundary of defined memory address space – target abort next access" under the following condition: A 64-64 bit transaction was issued to the model that ran up to, but did not cross, the memory boundary.
 - Resolution:: The model was corrected.
-

- Reference:: 53084
- MDL Version:: 01060
- MDL Date:: 30-Oct-2001

- SRC Version:: v3.46
 - Problem:: The model would violate the PCI-X protocol by asserting TRDY after a transaction had already ended with a Master Abort when the decode speed was set for Subtractive plus one (DEVSEL asserts six cycles after the address phase). This decode speed was set with the pcislave_request command (decode=4).
 - Resolution:: The model was corrected.
-

- Reference:: 54005
 - MDL Version:: 01059
 - MDL Date:: 10-Oct-2001
 - SRC Version:: v3.45
 - Problem:: If the slave received a Target Abort to a Split Completion after at least one successful data phase, then the model would incorrectly repeat the address used for that Split Completion in a subsequent Split Completion transaction.
 - Resolution:: The model has been fixed.
-

- Reference:: 54003
 - MDL Version:: 01058
 - MDL Date:: 10-Oct-2001
 - SRC Version:: v3.44
 - Problem:: In PCI mode, during back to back transactions, if the model drove PERR during the final data phase of a transaction, it would erroneously continue to drive PERR during the address phase of the next transaction.
 - Resolution:: The model has been fixed.
-

- Reference:: 53997
- MDL Version:: 01058

- MDL Date:: 10-Oct-2001
 - SRC Version:: v3.44
 - Problem:: The pcislave_fx FlexModel terminated a burst transaction prematurely under the following conditions: 1) a master issued a transaction with a starting address one data phase from ADB; and, 2) the model asserted STOP# on the first or second cycle after ADB.
 - Resolution:: The model was fixed.
-

- Reference:: 53987
 - MDL Version:: 01058
 - MDL Date:: 10-Oct-2001
 - SRC Version:: v3.44
 - Problem:: The slave would signal Disconnect at ADB erroneously during a DWORD transaction when the following conditions were true: (1) the starting address of the DWORD transaction and the byte count of the previous transaction combined to form a would-be transaction that would have crossed an ADB; and, (2) the slave was configured to disconnect on ADB.
 - Resolution:: The model has been corrected.
-

- Reference:: 53979
 - MDL Version:: 01058
 - MDL Date:: 10-Oct-2001
 - SRC Version:: v3.44
 - Problem:: The model would issue a Split Completion with an incorrect address field (the address was repeated from a previous Split Completion) when the following conditions were met: 1) a Split Completion occurred that transferred data for more than one data phase; 2) a subsequent Split Completion received a master abort termination.
 - Resolution:: The model has been corrected.
-

- Reference:: 53952
 - MDL Version:: 01057
 - MDL Date:: 28-Sep-2001
 - SRC Version:: v3.43
 - Problem:: The licensing packages for this model did not support the new DesignWare Regression feature.
 - Resolution:: The licensing software packages have been updated and the DesignWare-Regression feature can now be used with this model.
-

- Reference:: 53972
 - MDL Version:: 01056
 - MDL Date:: 25-Sep-2001
 - SRC Version:: v3.42
 - Problem:: This latest release of the model included a memory leak fix that would limit the number of outstanding pcislave_addr_req commands without corresponding pcislave_read_rslt commands. If a user issued more than 20 pcislave_addr_req commands without the same number of corresponding pcislave_read_rslt commands, later pcislave_read_rslt commands for the matched addr_req commands (which had exited the queue-20 deep) would then return data of zero.
 - Resolution:: The data queue size has been increased to 9999.
-

- Reference:: 53965
- MDL Version:: 01055
- MDL Date:: 20-Sep-2001
- SRC Version:: v3.41
- Problem:: Users would run out of process memory when simulating for an extended length of time (48 hrs or more) due to memory leaks in several flexmodel utilities. This sometimes lead to core dumps.

- Resolution:: The following flexmodel utilities were corrected: fastm_cmdcore, VMC process, and cpipe_hdl.

-
- Reference:: 53852
 - MDL Version:: 01055
 - MDL Date:: 20-Sep-2001
 - SRC Version:: v3.41
 - Problem:: The model was not driving the upper half of the AD bus to all ones during the data phase of a Split Completion as is required by section 2.11.2.4 of the PCI-X specification.
 - Resolution:: The model has been corrected.

-
- Reference:: 53804
 - MDL Version:: 01055
 - MDL Date:: 20-Sep-2001
 - SRC Version:: v3.41
 - Problem:: When the model issues Split Completion to a master that (1) delays TRDY/STOP and (2) issues Retry to the Split Completion and (3) delays TRDY to the retried Split Completion, the model will terminate the retried Split Completion one cycle early.
 - Resolution:: The model has been corrected.

-
- Reference:: 53753
 - MDL Version:: 01055
 - MDL Date:: 20-Sep-2001
 - SRC Version:: v3.41
 - Problem:: The model was not driving the upper half of the AD bus to all ones during the data phase of a Split Completion, as is required by section 2.11.2.4 of the PCI-X specification. Note, this is the same error as STAR 53852.

- Resolution:: The model has been corrected.

-
- Reference:: 53230

- MDL Version:: 01055

- MDL Date:: 20-Sep-2001

- SRC Version:: v3.41

- Problem:: The PCISLAVE_PAR_ERR_DATA phase parameter, used to generate bad parity on specific data phases, produced inconsistent results when configured to the first or second data phase. PERR# would assert for more than one data phase or on the wrong data phase (multiple errors were indicated instead of just one).

- Resolution:: The model has been corrected.

-
- Reference:: 53188

- MDL Version:: 01055

- MDL Date:: 20-Sep-2001

- SRC Version:: v3.41

- Problem:: When the upper memory bound of the slave was set to 64'hF and the slave received a transaction (read or write) that ran up to, but did not cross, the upper memory boundary, the slave would incorrectly issue this message: "FlexModelId: '3'. Generated burst memory address out of boundary of defined memory address space – target abort next access". The model would assert STOP# on the cycle after the last data phase. This was not legal PCI-X protocol.

- Resolution:: The model has been corrected.

-
- Reference:: 53645

- MDL Version:: 01054

- MDL Date:: 23-Aug-2001

- SRC Version:: v3.40

- Problem:: Previously, the model could not generate unexpected split completions. These transactions are useful for testing master devices which are required to handle unexpected split completions gracefully.
 - Resolution:: The model has been enhanced to create unexpected split completions by the use of the PCISLAVE_SCEM_ADDR configure parameter. This lets the user alter the address of an outstanding split completion.
-

- Reference:: 53601
 - MDL Version:: 01054
 - MDL Date:: 23-Aug-2001
 - SRC Version:: v3.40
 - Problem:: If a master device (acting as the slave during a split completion) signals Disconnect at ADB (by asserting STOP#) while the pcislave_fx has been configured to Disconnect at ADB as well, then the pcislave_fx will not honor a Disconnect on the ADB signal (STOP# assertion) on the first data phase of a subsequent split completion.
 - Resolution:: The model has been corrected.
-

- Reference:: 53330
 - MDL Version:: 01054
 - MDL Date:: 23-Aug-2001
 - SRC Version:: v3.40
 - Problem:: Repeated target aborts to split completions would cause the model to send incorrect data, and hang on a subsequent split completion (the first one that did not issue target abort).
 - Resolution:: The model has been corrected.
-

- Reference:: 53184
- MDL Version:: 01054

- MDL Date:: 23-Aug-2001
 - SRC Version:: v3.40
 - Problem:: A split completion transaction that requires the model to cross a memory boundary (created with pcislave_configure commands) should have the "Byte Count Modified" bit set in the attributes: a modified byte count such that the transaction does not cross the memory boundary and a subsequent split completion message. If the slave's upper memory boundary had been configured to 64'F, such a transaction would show the full byte count in the attributes. Furthermore, the slave would attempt to continue the transaction past the byte count (FRAME# and IRDY# would not de-assert, and the bus would hang).
 - Resolution:: The model has been corrected.
-

- Reference:: 53679
 - MDL Version:: 01053
 - MDL Date:: 8-Aug-2001
 - SRC Version:: v3.39
 - Problem:: The pcislave_fx FlexModel would indefinitely repeat a Split Completion transaction under the following conditions: 1) An arbiter was asserting the slave's GNT pin continuously (bus parking) and, 2) the pcislave_fx initiated a Split Completion transaction. The pcislave_fx would successfully complete the Split Completion, but after that it would repeat the Split Completion transaction indefinitely. The slave in this case never reverts back to being a slave.
 - Resolution:: The pcislave_fx FlexModel functions properly if an arbiter asserts GNT# continuously.
-

- Reference:: 53574
- MDL Version:: 01052
- MDL Date:: 19-Jul-2001
- SRC Version:: v3.38
- Problem:: The model was not tolerant of changes to the configuration parameter PCISLAVE_SPLIT_DISC_ON_ADB during a split completion. Changes to this parameter while a split completion transaction was occurring would result in the model returning incorrect data on subsequent split completions.

- Resolution:: The model has been corrected.

-
- Reference:: 53544
 - MDL Version:: 01052
 - MDL Date:: 19-Jul-2001
 - SRC Version:: v3.38
 - Problem:: The model reacted incorrectly to master abort or target abort during a split completion. The correct response is to de-queue the split completion and await the next transaction. Instead, the model would attempt to issue a split completion message.
 - Resolution:: The model has been corrected.

-
- Reference:: 53450
 - MDL Version:: 01051
 - MDL Date:: 26-Jun-2001
 - SRC Version:: v3.36
 - Problem:: When the model (correctly) sets the BCM in the attributes of a split completion (because the address for the read command was 3 or less data phases from an ADB) the model will erroneously continue to assert BCM for the subsequent split completion.
 - Resolution:: The model will be corrected in an upcoming release.

-
- Reference:: 53442
 - MDL Version:: 01051
 - MDL Date:: 26-Jun-2001
 - SRC Version:: v3.36
 - Problem:: A memory write block command for a maximum size data transfer to a non-aligned address will write data on the first data phase as though the address were aligned. Data previously written to the bytes from the aligned address, up to the starting address for the memory write block command, will be overwritten.

- Resolution:: Problem will be fixed in an upcoming release.

-
- Reference:: 53440
 - MDL Version:: 01051
 - MDL Date:: 26-Jun-2001
 - SRC Version:: v3.36
 - Problem:: A 64 to 32 split completion will transmit 32'h00000000 for the even numbered data phases starting with the fourth data phase, rather than the correct data.
 - Resolution:: Will be fixed in an upcoming release.
-

- Reference:: 53397
 - MDL Version:: 01051
 - MDL Date:: 26-Jun-2001
 - SRC Version:: v3.36
 - Problem:: When the model is in locked state (from a previous locked transaction) a new locked transaction while the model is configured for any decode speed other than the fastest (DECODE_A in PCI-X, Fast in PCI) will cause the model to issue this message: "FlexModelId: '2'. pci_slave: no path out of back-off state; resetting bus to idle state". The model will stop driving the interface signals immediately.
 - Resolution:: The model will be fixed in an upcoming release.
-

- Reference:: 53389
- MDL Version:: 01051
- MDL Date:: 26-Jun-2001
- SRC Version:: v3.36

- Problem:: The datasheet for the model describes a configure parameter PCISLAVE_MAX_SPLIT_TRAN but the model actually uses PCISLAVE_SPLIT_TRAN.
- Resolution:: The model will be corrected in an upcoming release. To maintain backwards compatibility, both parameters will be recognized.

-
- Reference:: 53318
 - MDL Version:: 01051
 - MDL Date:: 26-Jun-2001
 - SRC Version:: v3.36

- Problem:: The model did not give the user the ability to control the Split Completion Message attribute and data fields.
- Resolution:: Configuration parameters have been added (PCISLAVE_SCEM_CLASS and PCISLAVE_SCEM_INDEX) to allow control over those fields in a Split Completion Message.

-
- Reference:: 53380
 - MDL Version:: 01050
 - MDL Date:: 11-Jun-2001
 - SRC Version:: v3.36

- Problem:: 1) STAR 53325. The model could not generate out of order split completions correctly. In the case of two outstanding split completions that should have been order-swapped, the model would issue the one scheduled first correctly, but the second split completion (from the first split response) was issued with an incorrect address, and possibly uninitialized data

2) STAR 53380. The model would not update memory when a write command to a non-aligned address was terminated with a single data phase disconnect response

3) STAR 53320. The model would terminate 64 bit, 4K byte writes one data transfer too early; 4) STAR 53321. If the model received a write to the final address of a non-aligned, maximum size (4K byte), 64 bit write the model would not update internal memory correctly on an ensuing write.

- Resolution:: Model has been corrected for all listed problems.