

# **LogiCORE™ IP Endpoint PIPE v1.8 for PCI Express®**

## ***User Guide***

UG167 July 23, 2010



Xilinx is providing this product documentation, hereinafter "Information," to you "AS IS" with no warranty of any kind, express or implied. Xilinx makes no representation that the Information, or any particular implementation thereof, is free from any claims of infringement. You are responsible for obtaining any rights you may require for any implementation based on the Information. All specifications are subject to change without notice.

XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE INFORMATION OR ANY IMPLEMENTATION BASED THEREON, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Except as stated herein, none of the Information may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx.

© 2005-2010 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners. PCI, PCI Express, PCIe, and PCI-X are trademarks of PCI-SIG.

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/28/05	1.1	Initial Xilinx release.
8/31/05	1.2	Made available through CORE Generator v7.1i SP3.
01/18/06	1.3	Updated release number, tools version v8.1i, and release date.
7/13/06	1.4	Updated core version to 1.4, tools version to 8.2i.
9/21/06	1.5	Updated core version to 1.5.
2/15/07	1.6	Updated core version to 1.6; ISE tools to 9.1i.
5/17/07	1.7	Updated core version to 1.7; updated for PCI-SIG compliance.
7/23/10	2.0	Updated core version to 1.8 and updated ISE to 12.2. Integrated <i>Endpoint PIPE for PCI Express Getting Started Guide</i> (GSG168) chapters into this user guide. Added <a href="#">Chapter 2, "Licensing the Core,"</a> <a href="#">Chapter 4, "Quickstart Example Design,"</a> and <a href="#">Appendix D, "Additional Design Considerations."</a>



# Table of Contents

---

Revision History .....	3
<b>Schedule of Figures</b> .....	11
<b>Schedule of Tables</b> .....	13
<b>Preface: About This Guide</b>	
Guide Contents .....	15
Additional Resources .....	16
Conventions .....	16
Typographical .....	16
Online Document .....	17
<b>Chapter 1: Introduction</b>	
About the Core .....	19
System Requirements .....	19
Recommended Design Experience .....	19
Additional Core Resources .....	20
Technical Support .....	20
Feedback .....	20
Core .....	20
Document .....	20
<b>Chapter 2: Licensing the Core</b>	
Before you Begin .....	21
License Options .....	21
Simulation Only .....	21
Full System Hardware Evaluation .....	21
Obtaining Your License Key .....	22
Simulation License .....	22
Full System Hardware Evaluation License .....	22
Full License Key .....	22
Installing Your License File .....	22
Obtaining the NXP Simulation Models .....	23
<b>Chapter 3: Core Overview</b>	
Overview .....	25
Protocol Layers .....	26
Transaction Layer .....	26
Data Link Layer .....	26
Physical Layer .....	27
Configuration Management .....	27

PCI Configuration Space . . . . .	27
Core Interfaces . . . . .	30
System Interface . . . . .	30
NXP Standalone PHY . . . . .	30
PCI Express PHY Pin Description . . . . .	30
Transaction Interface . . . . .	32
Common TRN Interface . . . . .	33
Transmit TRN Interface . . . . .	33
Receive TRN Interface . . . . .	34
Configuration Interface . . . . .	36
Interrupt Generation Signals . . . . .	38
Error Reporting Signals . . . . .	39

## Chapter 4: Quickstart Example Design

<b>Overview . . . . .</b>	<b>41</b>
Simulation Design Overview . . . . .	41
Implementation Design Overview . . . . .	42
Example Design Elements . . . . .	42
<b>Generating the Core . . . . .</b>	<b>42</b>
<b>Simulating the Example Design . . . . .</b>	<b>44</b>
Setting up for Simulation . . . . .	45
Running the Simulation . . . . .	45
<b>Implementing the Example Design . . . . .</b>	<b>45</b>
<b>Directory Structure and File Contents . . . . .</b>	<b>47</b>
<project directory> . . . . .	47
<project directory>/<component name> . . . . .	48
<component name>/doc . . . . .	48
<component name>/example_design . . . . .	48
<component name>/implement . . . . .	49
implement/results . . . . .	49
<component name>/simulation . . . . .	50
simulation/dsport . . . . .	51
simulation/functional . . . . .	51
simulation/tests . . . . .	52
<b>NXP Simulation Models . . . . .</b>	<b>52</b>

## Chapter 5: Generating and Customizing the Core

<b>Using the CORE Generator GUI . . . . .</b>	<b>53</b>
<b>Basic Parameter Settings . . . . .</b>	<b>54</b>
Component Name . . . . .	55
PIPE Interface Type . . . . .	55
ID Initial Values . . . . .	55
Class Code . . . . .	56
Cardbus CIS Pointer . . . . .	56
<b>Base Address Registers . . . . .</b>	<b>57</b>
Base Address Register Overview . . . . .	58
Managing Base Address Register Settings . . . . .	59
Disabling Unused Resources . . . . .	59
<b>Configuration Register Settings . . . . .</b>	<b>60</b>
Capabilities Register . . . . .	61

Device Capabilities Register . . . . .	61
Link Capabilities Register . . . . .	62
MSI Control Register . . . . .	62
<b>Advanced Settings . . . . .</b>	<b>63</b>
Transaction Layer Module . . . . .	64
Advanced Physical Layer . . . . .	64
Advanced User Configuration Space Settings . . . . .	65
Power Management Registers . . . . .	65
Power Consumption . . . . .	65
Power Dissipated . . . . .	65

## Chapter 6: Designing with the Core

<b>Designing with the Transaction Interface . . . . .</b>	<b>67</b>
TLP Format on the 32-bit Transaction Interface . . . . .	67
Transmitting Outbound Packets . . . . .	68
Basic TLP Transmit Operation . . . . .	68
Presenting Back-to-Back Transactions on the Transaction Interface . . . . .	72
Source Throttling on the Transmit Data Path . . . . .	73
Destination Throttling of the Transmit Data Path . . . . .	74
Discontinuing Transmission of Transaction by Source . . . . .	76
Discontinuing Transmission of Transaction by Destination . . . . .	76
Packet Data Poisoning on the Transmit Transaction Interface . . . . .	77
Appending ECRC to Protect TLPs . . . . .	78
Transmit Buffers . . . . .	78
Receiving Inbound Packets . . . . .	79
Basic TLP Receive Operation . . . . .	79
Throttling the Data Path on the Transaction Interface . . . . .	83
Receiving Back-To-Back Transactions on the Transaction Interface . . . . .	84
Packet Re-ordering on Receive Transaction Interface . . . . .	85
Packet Data Poisoning and TLP Digest on Transaction Interface . . . . .	86
Packet Base Address Register Hit on Receive Transaction Interface . . . . .	87
Packet Transfer Discontinue on Transaction Interface . . . . .	88
Receiver Flow Control Credits Available . . . . .	90
Accessing Configuration Space Registers . . . . .	90
Registers Mapped Directly onto the Configuration Interface . . . . .	90
Device Control and Status Register Definitions . . . . .	91
Accessing Additional Registers Through the Configuration Port . . . . .	95
Additional Packet Handling Requirements . . . . .	95
Generation of Completions . . . . .	95
Tracking Non-Posted Requests and Inbound Completions . . . . .	95
Reporting User Error Conditions . . . . .	96
Power Management . . . . .	100
Power Management Support . . . . .	100
Active State Power Management . . . . .	100
Programmed Power Management . . . . .	100
Generating Interrupt Requests . . . . .	102
MSI Mode . . . . .	103
Legacy Interrupt Mode . . . . .	104

## Chapter 7: Core Constraints

Optional User Constraints . . . . .	107
-------------------------------------	-----

<b>Required Constraints</b> .....	107
Device, Package, and Speedgrade Selection .....	107
I/O Location Assignment Constraints .....	108
System Reset (Input) Signal .....	108
PIPE Receive (Input) Signals .....	108
PIPE Transmit (Output) Signals .....	108
Timing Constraints .....	109
Ignore Timing on Nets .....	109
Time Names for Clock Signals .....	109
Time Specifications for Clock Signals .....	109
Time Specifications for PIPE Signals .....	109
Timing Budget for PIPE Signals .....	110
NXP PX1011B-EL1 To XC3S1000-5 .....	110
XC3S1000-5 To NXP PX1011B-EL1 .....	111

## Appendix A: Tracking Receive-buffer Space for Inbound Completions

I/O Completions .....	113
Memory Read Completions .....	114

## Appendix B: Programmed Input Output Example Design

<b>System Overview</b> .....	115
<b>PIO Hardware</b> .....	116
Base Address Register Support .....	117
Changing CORE Generator Default BAR Settings .....	117
TLP Data Flow .....	118
Memory/IO Write TLP Processing .....	118
Memory/IO Read TLP Processing .....	118
PIO File Structure .....	119
PIO Application .....	120
Receive Path .....	121
Transmit Path .....	122
Endpoint Memory .....	124
<b>PIO Operation</b> .....	125
PIO Read Transaction .....	125
PIO Write Transaction .....	126
Device Utilization .....	127
<b>Summary</b> .....	127

## Appendix C: Downstream Port Model Test Bench

<b>Architecture</b> .....	130
<b>Simulating the Design</b> .....	131
<b>Test Selection</b> .....	131
<b>Waveform Dumping</b> .....	133
<b>Output Logging</b> .....	133
<b>Parallel Test Programs</b> .....	134
<b>Test Description</b> .....	134
Test Program: pio_writeReadBack_test0 .....	136



<b>Expanding the Downstream Port Model</b> .....	137
Downstream Port Model TPI Task List .....	137
.....	143
.....	144

## **Appendix D: Additional Design Considerations**

<b>Package Constraints</b> .....	147
<b>User Constraints Files</b> .....	147
<b>Wrapper File Usage</b> .....	148



# Schedule of Figures

---

## Chapter 1: Introduction

## Chapter 2: Licensing the Core

## Chapter 3: Core Overview

*Figure 3-1: Endpoint PIPE for PCIe Top-level Functional Blocks and Interfaces* . . . . . 26

## Chapter 4: Quickstart Example Design

*Figure 4-1: Example Simulation Design Block Diagram* . . . . . 41

*Figure 4-2: Example Implementation Design Block Diagram* . . . . . 42

*Figure 4-3: New Project Dialog Box* . . . . . 43

*Figure 4-4: Project Options* . . . . . 43

*Figure 4-5: Endpoint PIPE for PCI Express Main Screen* . . . . . 44

## Chapter 5: Generating and Customizing the Core

*Figure 5-1: PCI Express Basic Parameters: Screen 1* . . . . . 54

*Figure 5-2: PCI Express Basic Parameters: Screen 2* . . . . . 55

*Figure 5-3: Endpoint PIPE BAR Options: Screen 3* . . . . . 57

*Figure 5-4: Endpoint PIPE BAR Options: Screen 4* . . . . . 58

*Figure 5-5: Endpoint PIPE Configuration Settings: Screen 5* . . . . . 60

*Figure 5-6: Endpoint PIPE Configuration Settings: Screen 6* . . . . . 61

*Figure 5-7: Endpoint PIPE Advanced Settings: Screen 7* . . . . . 63

*Figure 5-8: Endpoint PIPE Advanced Settings: Screen 8* . . . . . 64

## Chapter 6: Designing with the Core

*Figure 6-1: PCI Express Base Specification Byte Order* . . . . . 67

*Figure 6-2: TLP 3-DW Header without Payload* . . . . . 69

*Figure 6-3: TLP with 4-DW Header without Payload* . . . . . 70

*Figure 6-4: TLP with 3-DW Header with Payload* . . . . . 71

*Figure 6-5: TLP with 4-DW Header with Payload* . . . . . 71

*Figure 6-6: Back-to-back Transaction on Transaction Interface* . . . . . 72

*Figure 6-7: Source Throttling on the Transmit Data Path* . . . . . 73

*Figure 6-8: Destination Throttling of the Endpoint Transaction Interface* . . . . . 74

*Figure 6-9: Destination Throttling of the Endpoint Transaction Interface* . . . . . 75

*Figure 6-10: Source Driven Transaction Discontinue on Transaction Interface* . . . . . 76

*Figure 6-11: Destination Driven Transaction Discontinue on Transaction Interface* . . . . . 77

*Figure 6-12: Packet Data Poisoning on the Transaction Interface* . . . . . 78

*Figure 6-13: TLP 3-DW Header without Payload* . . . . . 80

<i>Figure 6-14: TLP 4-DW Header without Payload</i> . . . . .	81
<i>Figure 6-15: TLP 3-DW Header with Payload</i> . . . . .	82
<i>Figure 6-16: TLP 4-DW Header with Payload</i> . . . . .	83
<i>Figure 6-17: User Application Throttling Receive TLP</i> . . . . .	84
<i>Figure 6-18: Receive Back-To-Back Transactions</i> . . . . .	84
<i>Figure 6-19: User Application Throttling of Back-to-Back TLPs</i> . . . . .	85
<i>Figure 6-19: Packet Re-ordering on Receive Transaction Interface</i> . . . . .	86
<i>Figure 6-20: Receive Transaction Data Poisoning</i> . . . . .	87
<i>Figure 6-21: BAR Target Determination using trn_rbar_hit</i> . . . . .	88
<i>Figure 6-22: Receive Transaction Discontinue</i> . . . . .	89
<i>Figure 6-23: Example Configuration Space Access</i> . . . . .	95
<i>Figure 6-24: Signaling Unsupported Request for Non-Posted TLP</i> . . . . .	98
<i>Figure 6-25: Signaling Unsupported Request for Posted TLP</i> . . . . .	98
<i>Figure 6-27: Example of UR Completion; No Error Message Sent</i> . . . . .	99
<i>Figure 6-28: Power Management Handshaking</i> . . . . .	102
<i>Figure 6-29: Requesting Interrupt Service: MSI and Legacy Mode</i> . . . . .	105

## Chapter 7: Core Constraints

## Appendix A: Tracking Receive-buffer Space for Inbound Completions

## Appendix B: Programmed Input Output Example Design

<i>Figure B-1: PCI Express System Overview</i> . . . . .	116
<i>Figure B-2: PIO Design Components</i> . . . . .	120
<i>Figure B-3: PIO 64-bit Application</i> . . . . .	120
<i>Figure B-4: PIO 32-bit Application</i> . . . . .	121
<i>Figure B-5: Rx Engines</i> . . . . .	121
<i>Figure B-6: Tx Engines</i> . . . . .	123
<i>Figure B-7: EP Memory Access</i> . . . . .	124
<i>Figure B-8: Back-to-Back Read Transactions</i> . . . . .	126
<i>Figure B-9: Back-to-Back Write Transactions</i> . . . . .	127

## Appendix C: Downstream Port Model Test Bench

<i>Figure C-2: Downstream Port Model and Top-level Endpoint</i> . . . . .	130
---	-----

## Appendix D: Additional Design Considerations

# Schedule of Tables

---

## Chapter 1: Introduction

## Chapter 2: Licensing the Core

## Chapter 3: Core Overview

Table 3-1: PCI Express Configuration Space Header . . . . .	29
Table 3-2: System Interface Signals . . . . .	30
Table 3-3: PXPIPE Transmit Data Interface Signals . . . . .	30
Table 3-4: PXPIPE Receive Data Interface Signals . . . . .	31
Table 3-5: Clock and Reference Signals . . . . .	31
Table 3-6: PXPIPE Command Interface Signals . . . . .	31
Table 3-7: PXPIPE Status Interface Signals . . . . .	32
Table 3-8: Common Transaction Interface Signals . . . . .	33
Table 3-9: Transmit Transaction Interface Signals . . . . .	33
Table 3-10: Receive Transaction Interface Signals . . . . .	34
Table 3-11: Configuration Interface Signals . . . . .	36
Table 3-12: User Application Interrupt-Generation Signals . . . . .	38
Table 3-13: User Application Error-Reporting Signals . . . . .	39

## Chapter 4: Quickstart Example Design

Table 4-1: Project Directory . . . . .	47
Table 4-2: Component Name Directory . . . . .	48
Table 4-3: Doc Directory . . . . .	48
Table 4-4: Example Design Directory . . . . .	48
Table 4-5: Implement Directory . . . . .	49
Table 4-6: Results Directory . . . . .	49
Table 4-7: Simulation Directory . . . . .	50
Table 4-8: dsport Directory . . . . .	51
Table 4-9: Functional Directory . . . . .	51
Table 4-10: dsport Directory . . . . .	52

## Chapter 5: Generating and Customizing the Core

## Chapter 6: Designing with the Core

Table 6-1: Transmit Buffers Available . . . . .	78
Table 6-2: trn_rbar_hit_n to Base Address Register Mapping . . . . .	87
Table 6-3: Transaction Receiver Credits Available Initial Values . . . . .	90
Table 6-4: Command and Status Registers Mapped to the Configuration Port . . . . .	91

<i>Table 6-5: Bit Mapping on Header Status Register</i> .....	91
<i>Table 6-6: Bit Mapping on Header Command Register</i> .....	92
<i>Table 6-7: Bit Mapping of PCI Express Device Status Register</i> .....	93
<i>Table 6-8: Bit Mapping of PCI Express Device Control Register</i> .....	93
<i>Table 6-9: Bit Mapping of PCI Express Link Status Register</i> .....	94
<i>Table 6-10: Bit Mapping on the Link Control Register</i> .....	94
<i>Table 6-11: User-indicated Error Signaling</i> .....	97
<i>Table 6-12: Possible Error Conditions for TLPs Received by the User Application</i> ....	97
<i>Table 6-13: Power Management Handshaking Signals</i> .....	102

## Chapter 7: Core Constraints

## Appendix A: Tracking Receive-buffer Space for Inbound Completions

<i>Table A-1: Receive-buffer Completion Allocations</i> .....	113
---	-----

## Appendix B: Programmed Input Output Example Design

<i>Table B-1: TLP Traffic Types</i> .....	117
<i>Table B-2: PIO Design File Structure</i> .....	119
<i>Table B-3: Rx Engine: Read Outputs</i> .....	122
<i>Table B-4: Rx Engine: Write Outputs</i> .....	122
<i>Table B-5: Tx Engine Inputs</i> .....	123
<i>Table B-6: EP Memory: Write Inputs</i> .....	125
<i>Table B-7: EP Memory: Read Inputs</i> .....	125
<i>Table B-8: PIO Design FPGA Resources</i> .....	127

## Appendix C: Downstream Port Model Test Bench

<i>Table C-1: Tests Provided with Downstream Port Model</i> .....	131
<i>Table C-2: Simulator Dump File Format</i> .....	133
<i>Table C-3: Test Setup Tasks</i> .....	137
<i>Table C-4: TLP Tasks</i> .....	138
<i>Table C-5: BAR Initialization Tasks</i> .....	142
<i>Table C-6: Example PIO Design Tasks</i> .....	143
<i>Table C-7: Expectation Tasks</i> .....	144

## Appendix D: Additional Design Considerations

<i>Table D-1: Supported Device and Interface Combinations</i> .....	147
---	-----

# About This Guide

---

The *LogiCORE Endpoint PIPE for PCI Express® User Guide* describes the function and operation of the Xilinx Endpoint PIPE for PCI Express (PCIe®) core, including how to design, customize, and implement the core.

## Guide Contents

This manual contains the following chapters:

- [Preface, “About this Guide”](#) introduces the organization and purpose of this user guide, a list of additional resources, and the conventions used in this document.
- [Chapter 1, “Introduction,”](#) describes the core and related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- [Chapter 2, “Licensing the Core,”](#) provides information about licensing the core and acquiring and installing the NXP simulation models.
- [Chapter 3, “Core Overview,”](#) defines the main components of the Endpoint PIPE for PCIe core architecture.
- [Chapter 4, “Quickstart Example Design,”](#) provides instructions for quickly generating, simulating, and implementing the example design using the demonstration test bench.
- [Chapter 5, “Generating and Customizing the Core,”](#) describes how to modify the Endpoint PIPE for PCIe interface.
- [Chapter 6, “Designing with the Core,”](#) provides instructions for designing an Endpoint device using the Endpoint PIPE for PCIe core.
- [Chapter 7, “Core Constraints,”](#) defines the required and optional constraints for the PCI Express PIPE core.
- [Appendix A, “Tracking Receive-buffer Space for Inbound Completions,”](#) provides steps to track receive-buffer space for inbound completions.
- [Appendix B, “Programmed Input Output Example Design,”](#) describes the PCIe Programmed Input Output (PIO) example design provided with the core.
- [Appendix C, “Downstream Port Model Test Bench,”](#) describes the test bench environment, which provides a test program interface for use with the PIO example design.
- [Appendix D, “Additional Design Considerations,”](#) defines additional considerations when implementing the example design.

## Additional Resources

To find additional documentation, see the Xilinx website at:

[www.xilinx.com/support/documentation/index.htm](http://www.xilinx.com/support/documentation/index.htm).

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

[www.xilinx.com/support/mysupport.htm](http://www.xilinx.com/support/mysupport.htm).

## Conventions

This document uses the following conventions. An example illustrates each convention.

### Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
Courier font	Messages, prompts, and program files that the system displays	<code>speed grade: - 100</code>
<b>Courier bold</b>	Literal commands that you enter in a syntactical statement	<b>ngdbuild</b> <i>design_name</i>
<b>Helvetica bold</b>	Commands that you select from a menu	<b>File → Open</b>
	Keyboard shortcuts	<b>Ctrl+C</b>
<i>Italic font</i>	Variables in a syntax statement for which you must supply values	<b>ngdbuild</b> <i>design_name</i>
	References to other manuals	See the <i>User Guide</i> for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are <i>not</i> connected.
Dark Shading	Items that are not supported or reserved	This feature is not supported
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as <b>bus[ 7:0 ]</b> , they are required.	<b>ngdbuild</b> [ <i>option_name</i> ] <i>design_name</i>
Braces { }	A list of items from which you must choose one or more	<b>lowpwr</b> = { <b>on</b>   <b>off</b> }
Vertical bar	Separates items in a list of choices	<b>lowpwr</b> = { <b>on</b>   <b>off</b> }
Angle brackets < >	User-defined variable or in code samples	<directory name>



Convention	Meaning or Use	Example
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Repetitive material that has been omitted	<b>allow block</b> <i>block_name loc1 loc2 ... locn;</i>
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active low	<b>usr_teof_n</b> is active low.

## Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section " <a href="#">Additional Resources</a> " for details. Refer to " <a href="#">Title Formats</a> " in <a href="#">Chapter 1</a> for details.
<a href="#">Blue, underlined text</a>	Hyperlink to a website (URL)	Go to <a href="http://www.xilinx.com">www.xilinx.com</a> for the latest speed files.



# Chapter 1

# Introduction

This chapter introduces the Endpoint PIPE for PCIe core and provides related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.

## About the Core

For information about system requirements, installation, and licensing options, see [Chapter 2, “Licensing the Core.”](#)

## System Requirements

- Windows® XP Professional 32-bit/64-bit
- Windows Vista Business 32-bit/64-bit

- Red Hat® Enterprise Linux WS v4.0 32-bit/64-bit
- Red Hat Enterprise Desktop v5.0 32-bit/64-bit (with Workstation Option)
- SUSE Linux Enterprise (SLE) desktop and server v10.1 32-bit/64-bit

- ISE® 12.2

## Recommended Design Experience

Although the Endpoint PIPE for PCIe core is a fully verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, previous experience building high performance, pipelined FPGA designs using Xilinx implementation software and user constraints files (UCF) is recommended.

## Additional Core Resources

For detailed information and updates about the core see the following documents, located on the [Endpoint PIPE for PCI Express](#) product page.

- *LogiCORE Endpoint PIPE for PCI Express Data Sheet*
- *LogiCORE Endpoint PIPE for PCI Express Release Notes*

Additional information and resources related to PCI Express technology are available from the following web sites:

- [PCI Express at PCI-SIG](#)
- [PCI Express PHY Transceiver \(PIPE\)](#)
- [PCI Express Developer's Forum](#)

## Technical Support

For technical support, go to [www.xilinx.com/support](http://www.xilinx.com/support). Questions are routed to a team of engineers with expertise using the Endpoint PIPE for PCIe core.

Xilinx will provide technical support for use of this product as described in the *Endpoint PIPE for PCI Express User Guide*. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

## Feedback

Xilinx welcomes comments and suggestions about the core and the accompanying documentation.

### Core

For comments or suggestions about the core, please submit a WebCase to [www.xilinx.com/support/clearexpress/websupport.htm](http://www.xilinx.com/support/clearexpress/websupport.htm). Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

### Document

For comments or suggestions about this document, please submit a WebCase at [www.xilinx.com/support/clearexpress/websupport.htm](http://www.xilinx.com/support/clearexpress/websupport.htm). Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

# Licensing the Core

---

This chapter provides instructions licensing for the Endpoint PIPE for PCI Express core, which you must do before using the core in your designs. In addition, information about acquiring and installing the NXP simulation models is provided.

The Endpoint PIPE for PCIe core is provided under the terms of the [Xilinx LogiCORE Site License Agreement](#), which conforms to the terms of the [SignOnce](#) IP License standard defined by the Common License Consortium. Purchase of the core entitles you to technical support and access to updates for one year.

## Before you Begin

This chapter assumes that you have installed the required ISE Updates either by running the Xilinx Update utility, or by performing a manual installation by downloading the required update from the Xilinx.com Downloads page, [www.xilinx.com/download](http://www.xilinx.com/download).

## License Options

The Endpoint PIPE for PCI Express LogiCORE IP offers three licensing options. After installing the required Xilinx ISE and EDK Updates, choose a license option.

### Simulation Only

The Simulation Only Evaluation license key for this core is included with this core, shipped with the Xilinx ISE software. This key lets you assess core functionality with your own design. (Functional simulation is supported by a dynamically generated HDL structural model.)

### Full System Hardware Evaluation

The Full System Hardware Evaluation license is available at no cost and lets you fully integrate the core into an FPGA design, place-and-route the design, evaluate timing, and perform functional simulation of the core using the example design and demonstration test bench provided with the core. In addition, the license key lets you generate a bitstream from the placed and routed design, which can then be downloaded to a supported device and tested in hardware. The core will continue to function in the target device for a limited time before timing out (ceasing to function), at which time it can be reactivated by reconfiguring the device.

The Endpoint PIPE for PCI Express core requires installation of a Full License key and the relevant ISE Update.

The Full license key provides full access to all core functionality both in simulation and in hardware, including:

- Functional simulation support
- Full implementation support including place and route and bitstream generation
- Full functionality in the programmed device with no time outs

## Obtaining Your License Key

This section contains information about obtaining a simulation, full system hardware, and full license keys.

### Simulation License

No action is required to obtain the Simulation Only Evaluation license key; it is provided by default with the Xilinx ISE software.

### Full System Hardware Evaluation License

To obtain a Full System Hardware Evaluation license, do the following:

1. Navigate to the product page for this core: [www.xilinx.com/products/ipcenter/DO-DI-PCIE-PIPE.htm](http://www.xilinx.com/products/ipcenter/DO-DI-PCIE-PIPE.htm)
2. Click Evaluate.
3. Follow the instructions to install the required Xilinx ISE Update.

### Full License Key

To obtain a Full license key, please follow these instructions.

1. Navigate to the product page for this core: [www.xilinx.com/products/ipcenter/DO-DI-PCIE-PIPE.htm](http://www.xilinx.com/products/ipcenter/DO-DI-PCIE-PIPE.htm)
2. Click Order
3. Follow the instructions to install the required Xilinx ISE Update, and generate the required license key on the Xilinx Product Download and Licensing Site, [www.xilinx.com/getproduct](http://www.xilinx.com/getproduct).

## Installing Your License File

An email will be sent to you containing instructions for installing your license file. Additional details about IP license key installation can be found in the ISE Design Suite Installation (this needs to be verified), Licensing and Release Notes document, which is available at [www.xilinx.com/support/documentation/dt\\_ise12-1.htm](http://www.xilinx.com/support/documentation/dt_ise12-1.htm).

## Obtaining the NXP Simulation Models

Acquire and install NXP PX1011B-EL1 PHY simulation models, located at [www.standardics.nxp.com/products/pcie/phys/](http://www.standardics.nxp.com/products/pcie/phys/). When simulating your design, if the NXP simulation model has not been installed or is installed in an incorrect location, a message from NXP Semiconductors appears:

Dear Customer,

In order to simulate the Xilinx Endpoint PIPE Core for PCI Express with PXPIPE interface, a simulation model of the NXP PX1011B-EL1 PCI Express PHY is required. This model is the property of NXP (formerly Philips Semiconductors), and is not included with the Xilinx product.

Various simulation models for commonly used tools are available. Please visit our NXP web page, download the no-charge end-user license agreement, sign, scan and email back to [interface.support@nxp.com](mailto:interface.support@nxp.com):

[www.standardics.nxp.com/support/models/px/](http://www.standardics.nxp.com/support/models/px/)

Be sure to include your complete contact information in your correspondence so that we can reply to your request as quickly as possible.

For data sheets, application notes, boundary scan files and other technical support documents, please visit

[www.standardics.nxp.com/products/pcie/phys/](http://www.standardics.nxp.com/products/pcie/phys/)

Thank you,

NXP Semiconductors





# Core Overview

---

This chapter describes the main components of the Endpoint PIPE for PCI Express core architecture.

## Overview

The Endpoint PIPE for PCIe core follows the *PCI Express Base Specification v1.1* layering model, which consists of the Physical, Data Link, and Transaction Layers, and configuration space.

Figure 3-1 illustrates the interfaces to the core, as defined below:

- System (SYS) interface
- PCI Express PIPE (PXPIPE) interface
- Configuration (CFG) interface
- Transaction (TRN) interface

The core uses packets to exchange information between the various modules. Packets are formed in the Transaction and Data Link Layers to carry information from the transmitting component to the receiving component. Necessary information is added to the packet being transmitted, which is required to handle the packet at those layers. At the receiving end, each layer of the receiving element processes the incoming packet, strips the relevant information and forwards the packet to the next layer. As a result, the received packets are transformed from their Physical Layer representation to their Data Link Layer representation and the Transaction Layer representation.

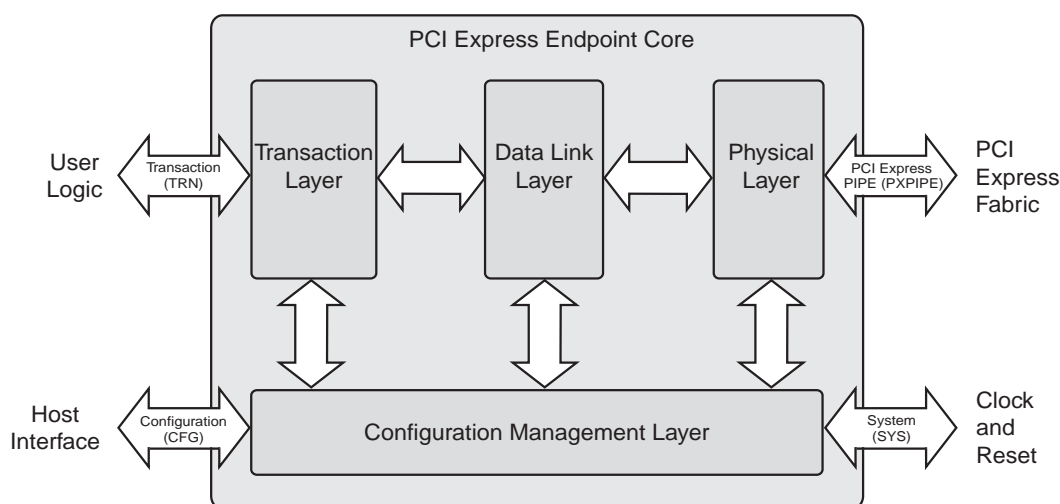


Figure 3-1: Endpoint PIPE for PCIe Top-level Functional Blocks and Interfaces

## Protocol Layers

The functions of the protocol layers, as defined by the *PCI Express Base Specification*, include generation and processing of Transaction Layer Packets (TLPs), flow control management, initialization and power management, data protection, error checking and retry, physical link interface initialization, maintenance and status tracking, serialization, de-serialization and other circuitry for interface operation. Each layer is defined below.

### Transaction Layer

The Transaction Layer is the upper layer of the PCI Express architecture, and its primary function is to accept, buffer, and disseminate Transaction Layer packets or TLPs. TLPs communicate information through the use of memory, IO, configuration, and message transactions. To maximize the efficiency of communication between devices, the Transaction Layer enforces PCI-compliant Transaction ordering rules, manages TLP buffer space via credit-based flow control and offers optional support for data poisoning.

### Data Link Layer

The Data Link Layer acts as an intermediate stage between the Transaction Layer and the Physical Layer. Its primary responsibility is to provide a reliable mechanism for the exchange of TLPs between two components on a link.

Services provided by the Data Link Layer include data exchange (TLPs), error detection and recovery, initialization services and the generation and consumption of Data Link Layer Packets (DLLPs). DLLPs are used to transfer information between Data Link Layers of two directly connected components on the link. DLLPs convey information such as Power Management, Flow Control, and TLP acknowledgments.

## Physical Layer

The Physical Layer exchanges information between the Data Link Layer and the external PHY component. Information received from the Data Link Layer is converted to an appropriate format and transmitted to the external PHY across the PXPIPE interface. The PXPIPE is an extended version of the PIPE specification and connects the Xilinx device and the NXP PHY. In the same way, incoming data is received from the external PHY through the PXPIPE, and transferred onto the Data Link Layer.

## Configuration Management

The Configuration Management layer maintains the PCI Type0 Endpoint configuration space and supports the following features:

- Implements PCI Configuration Space
- Supports Configuration Space accesses
- Power Management functions
- Implements error reporting and status functionality
- Implements packet processing functions
  - Receive
    - Configuration Reads and Writes
  - Transmit
    - Completions with or without data
    - TLM Error Messaging
    - User Error Messaging
    - Power Management Messaging/Handshake
- Implements MSI and INTx interrupt emulation
- Implements the Device Serial Number Capability in the PCI Express Extended Capability space

## PCI Configuration Space

The configuration space consists of three primary parts, illustrated in [Table 3-1](#). These include the following:

- Legacy PCI v3.0 Type 0 Configuration Header
- Legacy Extended Capability Items
  - PCI Express Capability Item
  - Power Management Capability Item
  - Message Signaled Interrupt Capability Item
- PCI Express Extended Capabilities
  - Device Serial Number Extended Capability Structure

The core implements three legacy extended capability items. The remaining legacy extended capability space from address 0x6C to 0xFF is reserved or user-definable. If the user does not use this space, the core returns 0x00000000 when this address range is read.

If the user chooses to implement registers within 0x6C to 0xFF, this space must be implemented in the User Application. The user is also responsible for returning 0x00000000 for any address within this range that is not implemented in the user appositional. For more information about enabling this feature, see *Chapter 4, Customizing the Core*.

The core also implements one PCI Express Extended Capability. The remaining PCI Express Extended Capability space from addresses 0x10C to 0x3FF is reserved and the space from addresses 0x400 to 0xFFFF is optionally available for users to implement. The core returns a Completion with UR (unsupported request) if there is a configuration access to addresses 0x10C to 0x3FF. If the user does not elect to implement addresses 0x400 to 0xFFFF, the core also returns a Completion with UR to configuration accesses in this range. If the user chooses to implement registers within 0x400 to 0xFFFF, this space must be implemented in the User Application. The user is also responsible for returning a Completion with UR for any address within this range that is not implemented in the User Application. For more information about enabling this feature, see *Chapter 4, Customizing the Core*.

Table 3-1: PCI Express Configuration Space Header

31		16 15		0
Device ID		Vendor ID		000h
Status		Command		004h
Class Code			Rev ID	008h
BIST	Header	Lat Timer	Cache Ln	00Ch
Base Address Register 0				010h
Base Address Register 1				014h
Base Address Register 2				018h
Base Address Register 3				01Ch
Base Address Register 4				020h
Base Address Register 5				024h
Cardbus CIS Pointer				028h
Subsystem ID		Subsystem Vendor ID		02Ch
Expansion ROM Base Address				030h
Reserved			CapPtr	034h
Reserved				038h
Max Lat	Min Gnt	Intr Pin	Intr Line	03Ch
PM Capability		NxtCap	PM Cap	040h
Data	BSE	PMCSR		044h
MSI Control		NxtCap	MSI Cap	048h
Message Address (Lower)				04Ch
Message Address (Upper)				050h
Reserved		Message Data		054h
PE Capability		NxtCap	PE Cap	058h
PCI Express Device Capabilities				05Ch
Device Status		Device Control		060h
PCI Express Link Capabilities				064h
Link Status		Link Control		068h
Reserved Legacy Configuration Space (Returns 0x00000000)				06Ch-0FFh
Next Cap	Capability	PCI Exp. Ext. Cap.(DSN)		100h
PCI Express Device Serial Number (1st)				104h
PCI Express Device Serial Number (2nd)				108h
Reserved Extended Configuration Space (Returns Completion with UR)				10Ch-FFFh

## Core Interfaces

The Endpoint PIPE for PCIe core includes top-level signal interfaces that have sub-groups for the receive direction, transmit direction, and signals common to both directions.

### System Interface

Table 3-2 defines the System (SYS) interface signal.

**Table 3-2: System Interface Signals**

Function	Signal Name	Direction	Description
System Reset	sys_reset_n	Input	Asynchronous, active low signal.

The system reset signal is an active-low asynchronous input. The assertion of `sys_reset_n` causes a hard reset of the entire core, including the NXP PHY. In typical endpoint applications, a sideband reset signal is present and should be connected to `sys_reset_n`. For endpoint applications that do not have a sideband system reset signal, the initial hardware reset should be generated locally. In either case, subsequent resets can be communicated in-band using the PCI Express protocol.

Note that systems designed to the PCI Express electro-mechanical specification provide a sideband reset signal which uses 3.3V signaling levels; carefully read the FPGA device data sheet to understand the requirements for interfacing to such signals.

### NXP Standalone PHY

The Endpoint PIPE for PCIe core uses an external NXP PX1011B-EL1. The interconnect between the core and the NXP PHY uses an extension of the PIPE architecture defined by Intel Corporation called the PXPIPE. Using another PHY with the PIPE interface is not compatible. Please see the NXP PX1011B data sheet for more information and the latest errata items related to the NXP PHY.

### PCI Express PHY Pin Description

Table 3-3 through Table 3-7 define the PXPIPE input and output signals. Note that input/output is defined from the perspective of the FPGA—a signal defined as an Output is *driven* by the FPGA and a signal defined as an Input is *received* by the FPGA. These signals are driven and received by the Endpoint PIPE for PCIe core; the user does not have to actively control these signals.

**Table 3-3: PXPIPE Transmit Data Interface Signals**

Symbol	Type	Description
TXDATA[7:0]	Output	8-bit transmit data from the FPGA to the NXP PHY.
TXDATAK	Output	Data/Control for the symbols of transmit data. A value of 0 indicates a data byte, a value of 1 indicates a control byte.

Table 3-4: PXPIPE Receive Data Interface Signals

Symbol	Type	Description
RXDATA[7:0]	Input	8-bit receive data from the NXP PHY to the FPGA.
RXDATAK	Input	Data/Control for the symbols of receive data. A value of 0 indicates a data byte, a value of 1 indicates a control byte.

Table 3-5: Clock and Reference Signals

Symbol	Type	Description
TXCLK	Output	Source synchronous 250 MHz clock output clock from the FPGA. All the data and input signals to the PHY from the FPGA are synchronized to this clock.
RXCLK	Input	Source synchronous 250 MHz clock input to the FPGA. All the data and input signals to the FPGA from the PHY are synchronized to this clock.

Table 3-6: PXPIPE Command Interface Signals

Symbol	Type	Description
TXDETECTRX_LOOPBACK	Output	Enable the NXP PHY to begin a receiver detection operation or to begin loopback.
TXELECIDLE	Output	Forces NXP PHY Tx output to electrical idle when asserted in all power states. When deasserted while in P0 (as indicated by the <i>PowerDown</i> signals), indicates that there is valid data present on the TXDATA[7:0] and TXDATAK pins and that the data should be transmitted. When deasserted in P2 the signal has no function. It would be used to indicate that the PHY should begin transmitting beacon signaling however beacon is not supported in this version of the PHY. TXELECIDLE must always be asserted while in power states P0s and P1.
TXCOMPLIANCE	Output	When high, sets the running disparity to negative. Used when transmitting the compliance pattern.
RXPOLARITY	Output	Active high, signals the PHY to perform a polarity inversion on the receive data.
RESETN	Output	Active low PHY reset from FPGA.
POWERDOWN[1:0]	Output	Power up or down the transceiver. Power states: 00 - P0, normal operation 01 - P0s, low recovery time latency, power saving state 10 - P1, longer recovery time (64us max) latency, lower power state 11 - reserved for P2, lowest power state.

Table 3-7: PXPIPE Status Interface Signals

Symbol	Type	Description
RXVALID	Input	Indicates symbol lock and valid data on RXDATA and RXDATAK.
PHYSTATUS	Input	Used to communicate completion of several NXP PHY functions including power management state transitions, and receiver detection.
RXELECIDLE	Input	Indicates receiver detection of an electrical idle. This is an asynchronous signal.
RXSTATUS[2:0]	Input	Encodes receiver status and error codes for the received data stream and receiver detection. 000 - Received data OK 001 - 1 SKP added 010 - 1 SKP removed 011 - Receiver detected 100 - 8B/10B decode error 101 - Elastic Buffer overflow 110 - Elastic Buffer underflow 111 - Receive disparity error

## Transaction Interface

Table 3-8 through Table 3-10 define the Transaction (TRN) interface signals, which provide a mechanism for the user design to generate and consume TLPs.



## Common TRN Interface

Table 3-8 defines the common Transaction interface signals.

Table 3-8: Common Transaction Interface Signals

Name	Direction	Description
trn_clk	Output	<b>Transaction Clock:</b> 62.50 MHz. Transaction and Configuration interface operations are referenced-to and synchronous-with the rising edge of this clock. trn_clk is unavailable when the core sys_reset_n is held asserted. trn_clk is guaranteed to be stable at the nominal operating frequency once the core deasserts trn_reset_n.
trn_reset_n	Output	<b>Transaction Reset:</b> Active low. User logic interacting with the Transaction and Configuration interfaces must use trn_reset_n to return to their quiescent states. trn_reset_n is deasserted synchronously with respect to trn_clk, sys_reset_n is deasserted and is asserted asynchronously with sys_reset_n assertion. Note that trn_reset_n will not be asserted for the core in-band reset events like <i>Hot Reset</i> or <i>Link Disable</i> . trn_reset_n is asserted due to loss of input receive clock from the external PHY device (used by the core's internal DCM to generate trn_clk).
trn_lnk_up_n	Output	<b>Transaction Link Up:</b> Active low. Transaction link-up is asserted when the core and the connected upstream link partner port are ready and able to exchange data packets. Transaction link-up is deasserted when the core and link partner are attempting to establish communication, and when communication with the link partner is lost due to errors on the transmission channel. When the core is driven to Hot Reset and Link Disable states by the link partner, then trn_lnk_up_n will be deasserted and all TLPs stored in the endpoint core will be lost.

## Transmit TRN Interface

Table 3-9 defines the transmit (Tx) Transaction interface signals.

Table 3-9: Transmit Transaction Interface Signals

Name	Direction	Description
trn_tsof_n	Input	<b>Transmit Start-of-Frame (SOF):</b> Active low. Signals the start of a packet.
trn_teof_n	Input	<b>Transmit End-of-Frame (EOF):</b> Active low. Signals the end of a packet.
trn_td[31:0]	Input	<b>Transmit Data:</b> Packet data to be transmitted.
trn_terrfdw_n	Input	<b>Transmit Error Forward:</b> Active low. Marks the packet in progress as error poisoned. Can be asserted any time between SOF and EOF, inclusive.
trn_tsrc_rdy_n	Input	<b>Transmit Source Ready:</b> Active low. Indicates that the User Application is presenting valid data on trn_td[31:0].

Table 3-9: Transmit Transaction Interface Signals (Cont'd)

Name	Direction	Description
trn_tdst_rdy_n	Output	<b>Transmit Destination Ready:</b> Active low. Indicates that the core is ready to accept data on trn_td[31:0]. The simultaneous assertion of trn_tsrc_rdy_n and trn_tdst_rdy_n marks the successful transfer of one DWORD on trn_td[31:0].
trn_tsrc_dsc_n	Input	<b>Transmit Source Discontinue:</b> Can be asserted any time starting on the first cycle after SOF to EOF, inclusive.
trn_tdst_dsc_n	Output	<b>Transmit Destination Discontinue:</b> Active low. Indicates that the core is aborting the current packet.
trn_tbuf_av[4:0]	Output	<b>Transmit Buffers Available:</b> Number of transmit buffers available in the core. Each transmit buffer can hold one packet with up to 512 bytes of payload.

## Receive TRN Interface

Table 3-10 defines the receive (Rx) Transaction interface signals.

Table 3-10: Receive Transaction Interface Signals

Name	Direction	Description
trn_rsdf_n	Output	<b>Receive Start-of-Frame (SOF):</b> Active low. Signals the start of a packet.
trn_reof_n	Output	<b>Receive End-of-Frame (EOF):</b> Active low. Signals the end of a packet.
trn_rd[31:0]	Output	<b>Receive Data:</b> Packet data being received.
trn_rerrfwd_n	Output	<b>Receive Error Forward:</b> Active low. Marks the packet in progress as error poisoned. Asserted by the core for the entire length of the packet.
trn_rsrc_rdy_n	Output	<b>Receive Source Ready:</b> Active low. Indicates the core is presenting valid data on trn_rd[31:0].
trn_rdst_rdy_n	Input	<b>Receive Destination Ready:</b> Active low. Indicates the User Application is ready to accept data on trn_rd[31:0]. The simultaneous assertion of trn_rsrc_rdy_n and trn_rdst_rdy_n marks the successful transfer of one DWORD on trn_rd[31:0].
trn_rsrc_dsc_n	Output	<b>Receive Source Discontinue:</b> Active low. Indicates the core is aborting the current packet. Asserted when the physical link is going into reset.

Table 3-10: Receive Transaction Interface Signals (Cont'd)

Name	Direction	Description
trn_rnp_ok_n	Input	<b>Receive Non-Posted OK:</b> Active low. The User Application asserts trn_rnp_ok_n when it is ready to accept a Non-Posted Request packet, allowing Posted and Completion packets to bypass Non-Posted packets in the inbound queue, if necessitated by the User Application. When the User Application approaches a state where it is unable to service Non-Posted Requests, it must deassert trn_rnp_ok_n one clock cycle before the core presents EOF of the last Non-Posted TLP the User Application can accept.
trn_rbar_hit_n[6:0]	Output	<b>Receive BAR Hit:</b> Active low. Indicates BAR(s) targeted by the current receive transaction. trn_rbar_hit_n[0] => BAR0 trn_rbar_hit_n[1] => BAR1 trn_rbar_hit_n[2] => BAR2 trn_rbar_hit_n[3] => BAR3 trn_rbar_hit_n[4] => BAR4 trn_rbar_hit_n[5] => BAR5 trn_rbar_hit_n[6] => Expansion ROM Address Note that if two BARs are configured into a single 64-bit address, both corresponding trn_rbar_hit_n bits are asserted.
trn_rfc_ph_av[7:0] <sup>a</sup>	Output	<b>Receive Posted Header Flow Control Credits Available:</b> The number of Posted Header FC credits available to the remote link partner.
trn_rfc_pd_av[11:0] <sup>a,b</sup>	Output	<b>Receive Posted Data Flow Control Credits Available:</b> The number of Posted Data FC credits available to the remote link partner.
trn_rfc_nph_av[7:0] <sup>a,b</sup>	Output	<b>Receive Non-Posted Header Flow Control Credits Available:</b> Number of Non-Posted Header FC credits available to the remote link partner.
trn_rfc_npd_av[11:0] <sup>a</sup>	Output	<b>Receive Non-Posted Data Flow Control Credits Available:</b> Number of Non-Posted Data FC credits available to the remote link partner.
trn_rfc_cplh_av[7:0] <sup>a,b</sup>	Output	<b>Receive Completion Header Flow Control Credits Available:</b> Number of Completion Header FC credits available to the remote link partner.
trn_rfc_cpld_av[11:0] <sup>a,b</sup>	Output	<b>Receive Completion Data Flow Control Credits Available:</b> Number of Completion Data FC credits available to the remote link partner.

- a. Credit values provided to the user are instantaneous quantities, not the cumulative (from time zero) values seen by the remote link partner.
- b. Completion credit values provided to the user reflect the actual state of the core's receive FIFO, even though the Base Specification requires an endpoint to advertise infinitive completion credits to its link partner regardless of actual receive capacity.

## Configuration Interface

Table 3-11 defines the Configuration (CFG) interface signals. The Configuration interface enables the user design to inspect the state of the configuration space. The user provides a 10-bit configuration address, which selects one of the 1024 configuration space double word (DWORD) registers. The endpoint returns the state of the selected register over the 32-bit data output port. See “Accessing Configuration Space Registers,” page 90 for usage.

Table 3-11: Configuration Interface Signals

Name	Direction	Description
cfg_do[31:0]	Output	<b>Configuration Data Out:</b> A 32-bit data output port used to obtain read data from the configuration space inside the core.
cfg_rd_wr_done_n	Output	<b>Configuration Read Write Done:</b> Active low. The read-write done signal indicates a successful completion of the user configuration register access operation. For a user configuration register read operation, the signal validates the cfg_do[31:0] data-bus value. Currently, writes to the configuration space through the configuration port are not supported. <sup>a</sup>
cfg_di[31:0]	Input	<b>Configuration Data In:</b> 32-bit data input port used to provide write data to the configuration space inside the core. Not supported. <sup>a</sup>
cfg_dwaddr[9:0]	Input	<b>Configuration DWORD Address:</b> A 10-bit address input port used to provide a configuration register DWORD address during configuration register accesses.
cfg_wr_en_n	Input	<b>Configuration Write Enable:</b> Active-low write-enable for configuration register access. Not supported. <sup>a</sup>
cfg_rd_en_n	Input	<b>Configuration Read Enable:</b> Active low, read-enable for configuration register access.
cfg_turnoff_ok_n	Input	<b>Configuration Turnoff OK:</b> Active-low power turn-off ready signal. The User Application can assert this to notify the core that it is safe for power to be turned off.
cfg_to_turnoff_n	Output	<b>Configuration To Turnoff:</b> Notifies the user that a PME_TURN_Off message has been received and the CMM will start polling the cfg_turnoff_ok_n input coming in from the user. After cfg_turnoff_ok_n is asserted, CMM sends a PME_To_Ack message to the upstream device.
cfg_byte_en_n[3:0]	Input	<b>Configuration Byte Enable:</b> Active-low byte enables for configuration register access signal. Not supported. <sup>a</sup>
cfg_bus_number[7:0]	Output	<b>Configuration Bus Number:</b> This output provides the assigned bus number for the device. The User Application must use this information in the Bus Number field of outgoing TLP request. Default value after reset is 00h. Refreshed whenever a Type 0 Configuration packet is received.

Table 3-11: Configuration Interface Signals (Cont'd)

Name	Direction	Description
cfg_device_number[4:0]	Output	<b>Configuration Device Number:</b> This output provides the assigned device number for the device. The User Application must use this information in the Device Number field of outgoing TLP request. Default value after reset is 00000b. Refreshed whenever a Type 0 Configuration packet is received.
cfg_function_number[2:0]	Output	<b>Configuration Function Number:</b> Provides the function number for the device. The User Application must use this information in the Function Number field of outgoing TLP request. Function number is hard-wired to 000b.
cfg_status[15:0]	Output	<b>Configuration Status:</b> The status register from the Configuration Space Header.
cfg_command[15:0]	Output	<b>Configuration Command:</b> The command register from the Configuration Space Header.
cfg_dstatus[15:0]	Output	<b>Configuration Device Status:</b> The device status register from the PCI Express Extended Capability Structure.
cfg_dcommand[15:0]	Output	<b>Configuration Device Command:</b> The control register from the PCI Express Extended Capability Structure.
cfg_lstatus[15:0]	Output	<b>Configuration Link Status:</b> Link status register from the PCI Express Extended Capability Structure.
cfg_lcommand[15:0]	Output	<b>Configuration Link Command:</b> Link control register from the PCI Express Extended Capability Structure.
cfg_pm_wake_n	Input	<b>Configuration Power Management Wake:</b> A one-clock cycle active low assertion signals the core to generate and send a Power Management Wake Event (PM_PME) Message TLP to the upstream link partner. <b>Note:</b> The user is required to assert this input only under stable link conditions as reported on the cfg_pcie_link_state[2:0]. Assertion of this signal when the PCI Express link is in transition results in incorrect behavior on the PCI Express link.
cfg_pcie_link_state_n[2:0]	Output	<b>PCI Express Link State:</b> This one-hot encoded bus reports the Link State Information to the user. 110b - PCI Express Link State is "L0" 101b - PCI Express Link State is "L0s" 011b - PCI Express Link State is "L1" 111b - PCI Express Link State is "in transition"
cfg_trn_pending_n	Input	<b>User Transaction Pending:</b> If asserted, sets the Transactions Pending bit in the Device Status Register. <b>Note:</b> The user is required to assert this input if the User Application has not received a completion to an upstream request.
cfg_dsn[63:0]	Input	<b>Configuration Device Serial Number:</b> Serial Number Register fields of the PCI Express Device Serial Number extended capability.

Table 3-11: Configuration Interface Signals (Cont'd)

Name	Direction	Description
fast_train_simulation_only	Input	<b>Fast Train:</b> Should only be asserted for simulation. Training counters are lowered when this input is asserted (set to "1") to allow the simulation to train faster. Do not assert this input when using the core in hardware. Doing so will cause the core to fail link training.
two_plm_auto_config	Input	<b>Two-PLM Auto-Config:</b> Used only for simulation; forces device to act as a down stream device for link training purposes. This signal should only be set to "1", if in simulation two Xilinx Endpoint cores are connected back-to-back. When using the provided downstream port model for simulation, this signal should be set to "0" on the endpoint core under test. This signal should be set to "0" when using the core in hardware as an endpoint device. Any other setting in hardware is not supported.

- a. Writing to the configuration space through the user configuration port is not supported at this time. These ports are on the Endpoint core to allow for future enhancement of this feature.

## Interrupt Generation Signals

Table 3-12 defines the User Application interrupt-generation signals.

Table 3-12: User Application Interrupt-Generation Signals

Port Name	Direction	Description
cfg_interrupt_n	Input	<b>Configuration Interrupt:</b> Active-low interrupt-request signal. The User Application may assert this to cause selected interrupt message-type to be transmitted by the core. The signal should be held low until <code>cfg_interrupt_rdy_n</code> is asserted.
cfg_interrupt_rdy_n	Output	<b>Configuration Interrupt Ready:</b> Active-low interrupt grant signal. The simultaneous assertion of <code>cfg_interrupt_rdy_n</code> and <code>cfg_interrupt_n</code> indicates that the core has successfully transmitted the requested interrupt message.
cfg_interrupt_mmenable[2:0]	Output	<b>Configuration Interrupt Multiple Message Enable:</b> This is the value of the Multiple Message Enable field. Values range from 000b to 101b. A value of 000b indicates that single vector MSI is enabled, while other values indicate the number of bits that may be used for multi-vector MSI.
cfg_interrupt_msienable	Output	<b>Configuration Interrupt MSI Enabled:</b> Indicates that the Message Signaling Interrupt (MSI) messaging is enabled. If 0, then only Legacy (INTx) interrupts may be sent.

Table 3-12: User Application Interrupt-Generation Signals (Cont'd)

Port Name	Direction	Description										
cfg_interrupt_di[7:0]	Input	<p><b>Configuration Interrupt Data In:</b> For Message Signaling Interrupts (MSI), the portion of the Message Data that the Endpoint must drive to indicate MSI vector number, if Multi-Vector Interrupts are enabled. The value indicated by cfg_interrupt_mmenable[2:0] determines the number of lower-order bits of Message Data that the Endpoint provides; the remaining upper bits of cfg_interrupt_di[7:0] are not used. For Single-Vector Interrupts, cfg_interrupt_di[7:0] is not used. For Legacy interrupt messages (Assert_INTx, Deassert_INTx), this indicates which message-type will be sent, where:</p> <table><tr><th>Value</th><th>Legacy Interrupt</th></tr><tr><td>00h</td><td>INTA</td></tr><tr><td>01h</td><td>INTB</td></tr><tr><td>02h</td><td>INTC</td></tr><tr><td>03h</td><td>INTD</td></tr></table>	Value	Legacy Interrupt	00h	INTA	01h	INTB	02h	INTC	03h	INTD
Value	Legacy Interrupt											
00h	INTA											
01h	INTB											
02h	INTC											
03h	INTD											
cfg_interrupt_do[7:0]	Output	<p><b>Configuration Interrupt Data Out:</b> This is the value of the lowest 8 bits of the Message Data field in the Endpoint's MSI capability structure. This value is used in conjunction with cfg_interrupt_mmenable[2:0] to drive cfg_interrupt_di[7:0].</p>										
cfg_interrupt_assert_n	Input	<p><b>Configuration Legacy Interrupt Assert/Deassert Select:</b> Selects between Assert and Deassert messages for Legacy interrupts when cfg_interrupt_n is asserted. Not used for MSI interrupts.</p> <table><tr><th>Value</th><th>Message Type</th></tr><tr><td>0</td><td>Assert</td></tr><tr><td>1</td><td>Deassert</td></tr></table>	Value	Message Type	0	Assert	1	Deassert				
Value	Message Type											
0	Assert											
1	Deassert											

## Error Reporting Signals

Table 3-13 defines the User Application error-reporting signals.

Table 3-13: User Application Error-Reporting Signals

Port Name	Direction	Description
cfg_err_ecrc_n	Input	<p><b>ECRC Error Report:</b> The user can assert this signal to report an ECRC error (end-to-end CRC).</p>
cfg_err_ur_n	Input	<p><b>Configuration Error Unsupported Request:</b> The user can assert this signal to report that an unsupported request was received.</p>

Table 3-13: User Application Error-Reporting Signals (Cont'd)

Port Name	Direction	Description
cfg_err_cpl_timeout_n	Input	<b>Configuration Error Completion Timeout:</b> The user can assert this signal to report a completion timed out. <b>Note:</b> The user should assert this signal only if the device power state is D0. Asserting this signal in non-D0 device power states might result in an incorrect operation on the PCI Express link. For additional information, see the <i>PCI Express Base Specification</i> , Rev.1.1, Section 5.3.1.2.
cfg_err_cpl_unexpect_n	Input	<b>Configuration Error Completion Unexpected:</b> The user can assert this signal to report that an unexpected completion was received.
cfg_err_cpl_abort_n	Input	<b>Configuration Error Completion Aborted:</b> The user can assert this signal to report that a completion was aborted.
cfg_err_posted_n	Input	<b>Configuration Error Posted:</b> This signal is used to further qualify any of the cfg_err_* input signals. When this input is asserted concurrently with one of the other signals, it indicates that the transaction which caused the error was a posted transaction.
cfg_err_cor_n	Input	<b>Configuration Error Correctable Error:</b> The user can assert this signal to report that a correctable error was detected.
cfg_err_tlp_cpl_header[47:0]	Input	<b>Configuration Error TLP Completion Header:</b> Accepts the header information from the user when an error is signaled. This information is required so that the core can issue a correct completion, if required. The following information should be extracted from the received error TLP and presented in the format below: [47:41] Lower Address [40:29] Byte Count [28:26] TC [25:24] Attr [23:8] Requester ID [7:0] Tag



## Quickstart Example Design

This chapter provides an overview of the Endpoint PIPE for PCI Express example design and instructions for generating the core. It also covers simulating and implementing the example design using the provided demonstration test bench.

### Overview

The example simulation design consists of two discrete parts:

- The Endpoint for PCI Express Downstream Port Model, a test bench that generates, consumes, and checks PCIe bus traffic.
- The Programmed Input-Output (PIO) example design, a completer application. The PIO example design responds to PCIe Read and Write requests to its memory space and can be synthesized for testing in hardware.

### Simulation Design Overview

For the example simulation design, transactions are sent from the Downstream Port Model to the core and processed by the PIO example design. [Figure 4-1](#) illustrates the simulation design provided with the core. For more information about the Downstream Port Model, see [Appendix C, “Downstream Port Model Test Bench.”](#)

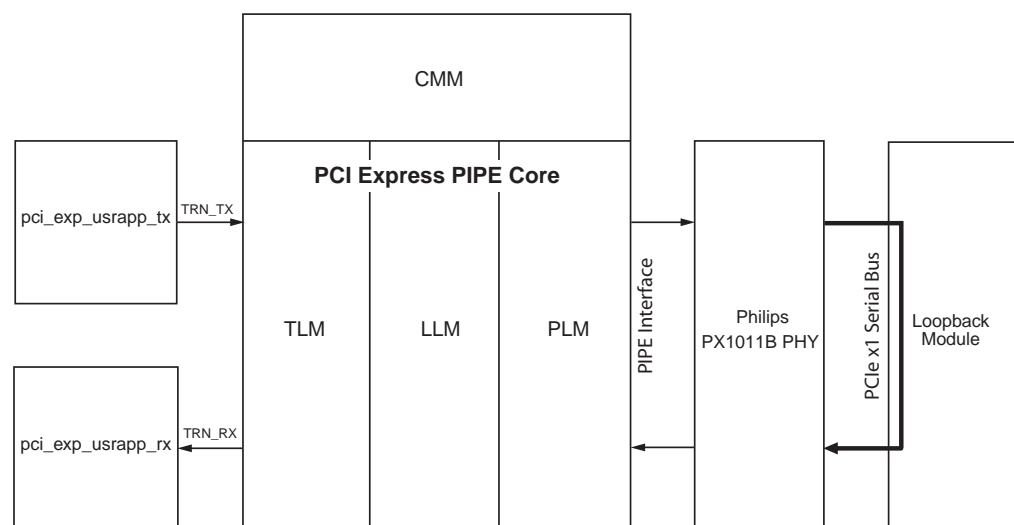


Figure 4-1: Example Simulation Design Block Diagram

## Implementation Design Overview

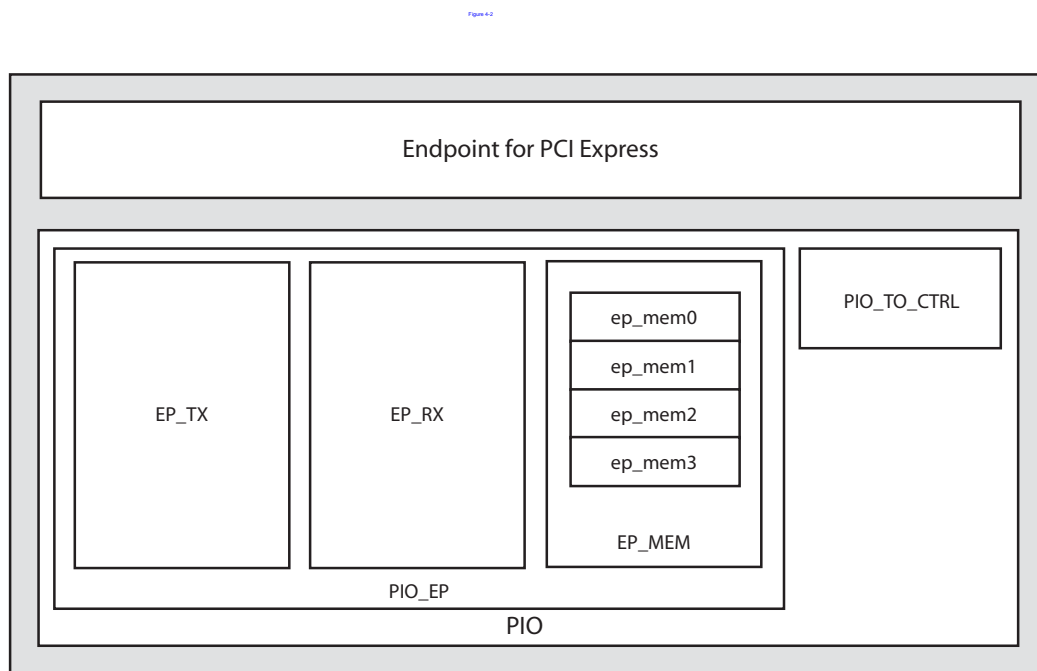


Figure 4-2: Example Implementation Design Block Diagram

## Example Design Elements

The PIO example design elements include the following:

- Core netlists
- Core simulation models
- An example Verilog HDL wrapper (instantiates the core and example design)
- A customizable demonstration test bench to simulate the example design

The example design has been tested with Xilinx ISE v12.2 and the following simulators:

- Cadence™ Incisive Enterprise Simulator (IES) v9.2 and above
- Synopsys® VCS and VCS MX 2009.12 and above
- Mentor Graphics® ModelSim® v6.5c and above

## Generating the Core

To generate an Endpoint PIPE for PCIe core using the default values in the CORE Generator graphical user Interface (GUI), do the following:

1. Start the CORE Generator.

For help starting and using the CORE Generator, see the Xilinx CORE Generator Guide, available from the [ISE documentation](http://www.xilinx.com/ise/documentation/core_generator/) web page.

2. Choose File > New Project. The New Project dialog box appears.

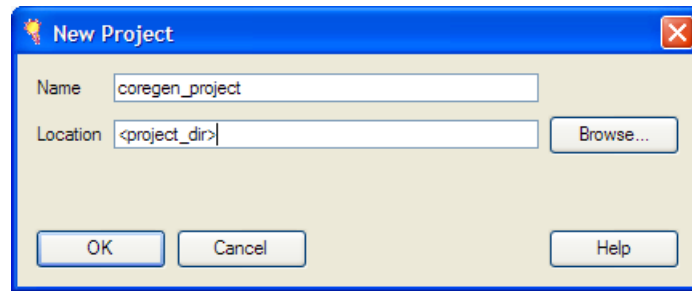


Figure 4-3: New Project Dialog Box

3. Enter a project name and location; then click OK. <project\_dir> is used in this example. The Project Options dialog box appears.

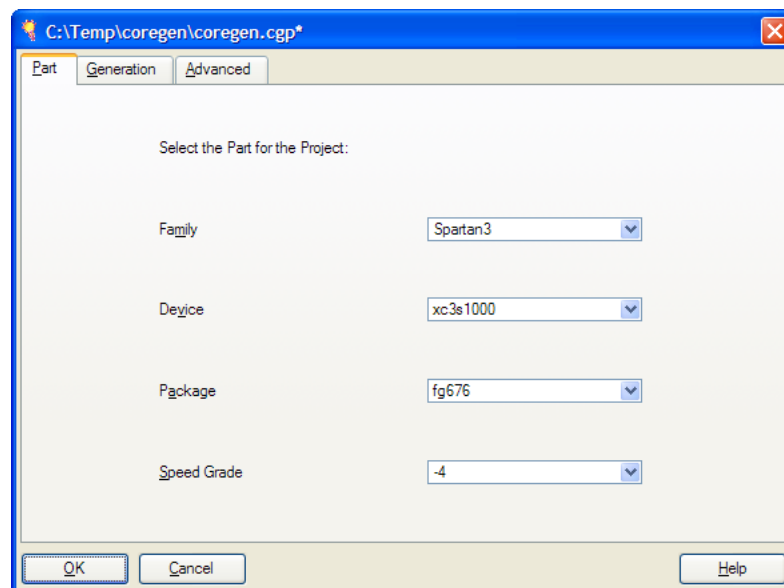


Figure 4-4: Project Options

4. Set the project options:

From the Part tab, select the following options:

- **Family:** Spartan3
- **Device:** xc3s1000
- **Package:** fg676
- **Speed Grade:** -4

**Note:** If an unsupported silicon family is selected, the core is not available for customization and is dimmed in the list of cores.

From the Generation tab, select the following parameters:

- **Design Entry:** Select either VHDL or Verilog. (Note that the example design is provided for Verilog only.)
- **Vendor:** Select Synplicity® or ISE (for XST).

5. Click OK.
6. Locate the Endpoint PIPE for PCI Express core in the core selection tree under Standard Bus Interfaces/PCI Express. Double-click the core name to display the main GUI screen. (Figure 4-5).

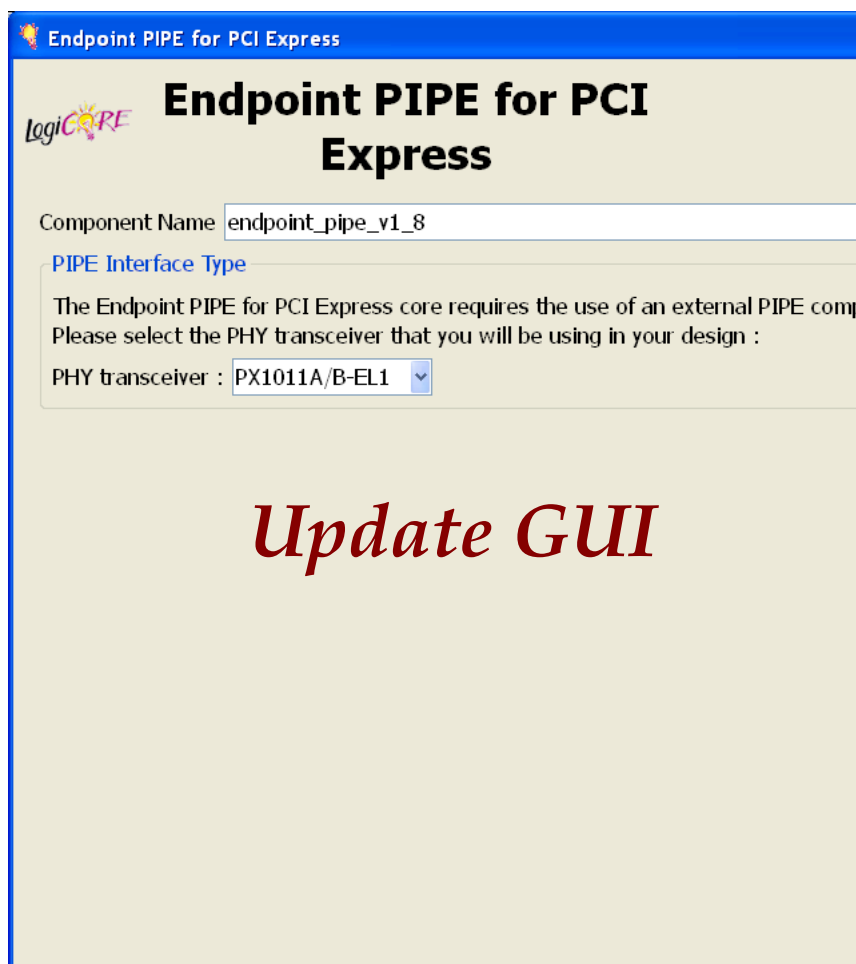


Figure 4-5: Endpoint PIPE for PCI Express Main Screen

7. In the Component Name field, enter a name for the core. <component\_name> is used to represent the project name in this example.
8. Click Finish to generate the core using the default parameters. The core and its supporting files, including the PIO example design and Downstream Port Model test bench, are generated in the project directory.

For detailed information about the example design directories and files, see [“Directory Structure and File Contents,”](#) page 47.

## Simulating the Example Design

The example design provides a quick way to simulate and observe the behavior of the core. The simulation environment provided with Endpoint PIPE for PCIe core performs simple memory access tests on the PIO example design. Transactions are generated by the Downstream Port Model and responded to by the PIO example design.

- PCI Express Transaction Layer Packets (TLPs) are generated by the test bench transmit user application (`pci_exp_usrapp_tx`). As it transmits TLPs it also generates a log, `tx.dat`.
- PCI Express TLPs are received by the test bench receive user application (`pci_exp_usrapp_rx`). As the user application receives the TLPs, it generates a log file, `rx.dat`.

For more information about the test bench, see Appendix C, “Downstream Port Model Test Bench,” in the *LogiCORE Endpoint PIPE for PCI Express User Guide*.

## Setting up for Simulation

Simulation scripts are provided for Cadence IUS (VNC), Synopsys VCS, and Mentor Graphics ModelSim. Set your environment to run the simulation tool of your choice.

## Running the Simulation

The test bench provided with the example design supports pre-implementation mode (RTL) simulations:

- The test bench, along with RTL model of the example design
  - The Verilog HDL model of the Endpoint PIPE for PCIe core, created by the CORE Generator
  - The PCI Express PHY models (which must be obtained from NXP and installed) are simulated. For additional information about obtaining the NXP model, see [“Obtaining the NXP Simulation Models,” page 9](#).
1. To run the simulation, go to the following directory:  
`<project_dir>/<component_name>/simulation/functional`
  2. Run the script that corresponds to your simulation tool using one of the following:
    - **VCS:** `simulate_vcs.sh`
    - **Cadence IES:** `simulate_ncsim.sh`
    - **ModelSim:** `simulate_mti.do`

## Implementing the Example Design

After generating the core, the netlists and the example design can be processed by the Xilinx implementation tools. The generated output files include scripts to assist the user in running the Xilinx software.

To implement the example design, open a command prompt or terminal window and type the following commands:

### Windows

```
ms-dos> cd <project_dir>\<component_name>\implement
ms-dos> implement.bat
```

### UNIX

```
unix-shell% cd <project_dir>/<component_name>/implement
unix-shell% ./implement.sh
```

These commands execute a script that synthesizes, builds, maps, and place-and-routes the example design. The script then generates a post-par simulation model for use in timing

simulation. The resulting files are placed in the `results` directory and execute the following processes:

1. Remove data files from the previous runs.
2. Synthesizes the example design using either Synplicity Synplify or XST.
  - The core is instanced as a black box within the example design.
3. `ngdbuild` builds a Xilinx design database for the example design.

Inputs:

**Part-Package-Speed Grade selection:**

XC3S1000-FG676-4

**Example design UCF:**

`xilinx_1_lane_epipe_ep-XC3S1000-FG676-4.ucf`

4. `map`: Maps the design to the selected FPGA using the constraints provided.
5. `par`: Places cells onto FPGA resources and routes connectivity.
6. `trce`: Performs static timing analysis on design using specified constraints.
7. `netgen`: Generates a logical Verilog HDL representation of the design and an SDF file, for post-layout verification.
8. `bitgen`: Generates a bitstream file for programming the FPGA.

The following FPGA implementation related files are generated in the `results` directory:

- `routed.bit`  
FPGA configuration information.
- `routed.v`  
Verilog functional Model.
- `routed.sdf`  
Timing model Standard Delay File.
- `mapped.mrp`  
Xilinx map report.
- `routed.par`  
Xilinx place and route report.
- `routed.twr`  
Xilinx timing analysis report.

The script file starts from an EDIF/NGC file and results in a bitstream file. Although it is possible to use the Xilinx ISE GUI to implement the example design, the GUI flow is not included in this document.

## Directory Structure and File Contents

The PCI Express PIPE core directories and their associated files are defined in the sections that follow. Click a directory name to go to the desired directory and its associated files.



### <project directory>

Top-level project directory; name is user-defined



#### <project directory>/<component name>

Core release notes file



#### <component name>/implement

Core implementation script files



#### <component name>/doc

Supporting core documentation



#### <component name>/example\_design

Verilog design files and UCF



#### <component name>/implement

Implementation script files



#### implement/results

Results directory, created after implementation scripts are run, and contains implement script results



#### <component name>/simulation

Simulation scripts



#### simulation/dsport

Simulation files



#### simulation/functional

Functional simulation files



#### simulation/tests

Test command files

### <project directory>

The project directory contains all the CORE Generator project files.

Table 4-1: Project Directory

Name	Description
<project_dir>	
<component_name>.ngc	Top-level netlist.
<component_name>.v[hd]	Verilog or VHDL simulation model.
<component_name>.xco	CORE Generator project-specific option file; can be used as an input to the CORE Generator.
<component_name>_flist.txt	List of files delivered with core.

Table 4-1: Project Directory (Cont'd)

Name	Description
<component_name>.{veo vho}	VHDL or Verilog instantiation template.

[Back to Top](#)

## <project directory>/<component name>

The component name directory contains the release notes file provided with the core, which may include last-minute changes and/or updates.

Table 4-2: Component Name Directory

Name	Description
<project_dir>/<component_name>	
pci_express_pipe_release_notes.txt	Release notes file for the Endpoint PIPE for PCIe core.

[Back to Top](#)

## <component name>/doc

The doc directory contains the PDF documentation provided with the core.

Table 4-3: Doc Directory

Name	Description
<project_dir>/<component_name>/doc	
pcie_pipe_ds321.pdf	<i>LogiCORE Endpoint PIPE for PCI Express Data Sheet</i>
pcie_pipe_ug167.pdf	<i>LogiCORE Endpoint PIPE for PCI Express User Guide</i>

[Back to Top](#)

## <component name>/example\_design

The example design directory contains the example design files provided with the core.

Table 4-4: Example Design Directory

Name	Description
<project_dir>/<component_name>/example_design	
pci_exp_1_lane_epipe_ep.v	Verilog top-level example design.
xilinx_pci_exp_1_lane_epipe_ep.v	Example design wrapper file.
xilinx_pci_exp_1_lane_epipe_endpoint_product.v	Enables Endpoint PIPE for PCIe version of the test bench.
<filename>.ucf	Example design UCF. Filename varies by family, part, and package selected.



Table 4-4: Example Design Directory (Cont'd)

Name	Description
EP_MEM.v impl_test.v pci_exp_32b_app.v PIO.v PIO_32.v PIO_32_RX_ENGINE.v PIO_32_TX_ENGINE.v PIO_EP.v PIO_EP_MEM_ACCESS.v PIO_TO_CTRL.v	PIO example design files.

[Back to Top](#)

## <component name>/implement

The implement directory contains the core implementation script files.

Table 4-5: Implement Directory

Name	Description
<project_dir>/<component_name>/implement	
implement.bat implement.sh	DOS and UNIX/Linux implementation scripts.
synplify.prj	Synplify synthesis script.
xilinx_pci_exp_1_lane_epipe_ep_inc.xst	XST project file.
xst.scr	XST synthesis script.

[Back to Top](#)

## implement/results

The results directory is created by the implement script, after which the implement script results are placed in the results directory.

Table 4-6: Results Directory

Name	Description
<project_dir>/<component_name>/implement/results	
The results directory is created by the implement script, after which the implement script results are placed in the results directory.	

[Back to Top](#)

## <component name>/simulation

The simulation directory contains the source files for the provided test bench.

**Table 4-7: Simulation Directory**

Name	Description
<project_dir>/<component_name>/simulation	
board_common.v	Contains test bench definitions.
board.v	Top-level simulation module and loop-back.
pxl011a_pli.v	NXP PX1011A-EL1 PCI Express Standalone PHY dummy model for Synopsys VCS and ModelSim simulator. This file must be replaced with the actual model acquired from the PHY vendor. See <a href="#">“Obtaining the NXP Simulation Models,” page 9</a> for information.
pxl011a.v	NXP PX1011A-EL1 PCI Express Standalone PHY dummy model, for Cadence IUS simulator. This file must be replaced with the actual model acquired from the PHY vendor. See <a href="#">“Obtaining the NXP Simulation Models,” page 9</a> for information.
sys_clk_gen_ds.v	System differential clock source.
sys_clk_gen.v	System clock source.
xilinx_pci_exp_32bit_cor_ep.v	Endpoint PIPE for PCIe application module. This module instances FPGA example design and PCI Express PHY, along with application models.
xilinx_pci_exp_cor_ep.f	List of files comprising the design being tested.
xilinx_pci_exp_defines.v	Endpoint PIPE for PCIe application macro definitions.

[Back to Top](#)

## simulation/dsport

The dsport directory contains files for the Downstream Port Model test bench.

**Table 4-8: dsport Directory**

Name	Description
<project_dir>/<component_name>/simulation/dsport	
dsport_cfg.v	Downstream Port Model files.
pci_exp_usrapp_cfg.v	
pci_exp_usrapp_tx.v	
pci_exp_usrapp_rx.v	
pci_exp_usrapp_com.v	
pci_exp_1_lane_64b_dsport.v	
xilinx_pci_exp_downstream_port.v	
pci_exp_expect_tasks.v	
xilinx_pci_exp_dsport.v	

[Back to Top](#)

## simulation/functional

The functional directory contains input files for functional simulations.

**Table 4-9: Functional Directory**

Name	Description
<project_dir>/<component_name>/simulation/functional	
board_rtl_x01_epipe.f board_rtl_x01_epipe_ncv.f	List of files for RTL simulations.
simulate_ncsim.sh	Cadence IUS simulation script.
simulate_vcs.sh	Synopsys VCS simulation script.
simulate_mti.do	ModelSim simulation script
wave_ncsim.sv	Cadence IUS waveform script.
wave_mti.do	ModelSim waveform script.
xilinx_lib.f xilinx_lib_mti_v4fx.f xilinxn_lib_vcs_v4fs.f	Xilinx libraries; requires modification to point to the user location of Xilinx libraries.

[Back to Top](#)

## simulation/tests

The tests directory contains sample tests to drive the downstream port model

Table 4-10: **dsport Directory**

Name	Description
<project_dir>/<component_name>/simulation/tests	
sample_tests1.v	Sample tests to drive the Downstream Port Model.
tests.v	
pio_tests.v	

[Back to Top](#)

## NXP Simulation Models

To simulate the Endpoint PIPE for PCI Express core with the PXPIPE interface, a simulation model of the NXP PX1011B-EL1 PCI Express PHY is required. This model is the property of NXP and is not included with this Xilinx product.

Various simulation models for commonly used tools are available. Please visit the NXP web page at [www.standardics.nxp.com/support/models/px/](http://www.standardics.nxp.com/support/models/px/) to obtain a license agreement and register for the simulation models. Be sure to include your contact information in your request so that NXP can respond to you as quickly as possible.

# *Generating and Customizing the Core*

---

The Endpoint PIPE for PCI Express core is fully configurable and highly customizable, yielding a resource-efficient implementation tailored to your design requirements. To generate and customize the core, a graphical user interface (GUI) is provided through the CORE Generator.

## **Using the CORE Generator GUI**

The Endpoint PIPE for PCIe CORE Generator GUI consists of eight screens:

- Screens 1 and 2: [“Basic Parameter Settings”](#)
- Screens 3 and 4: [“Base Address Registers”](#)
- Screens 5 and 6: [“Configuration Register Settings”](#)
- Screens 7 and 8: [“Advanced Settings”](#)

## Basic Parameter Settings

The initial screens (Figure 5-1 and Figure 5-2) are used to define basic parameters for the core, including component name, interface type, ID initial values, class code, and Cardbus CIS pointer information.

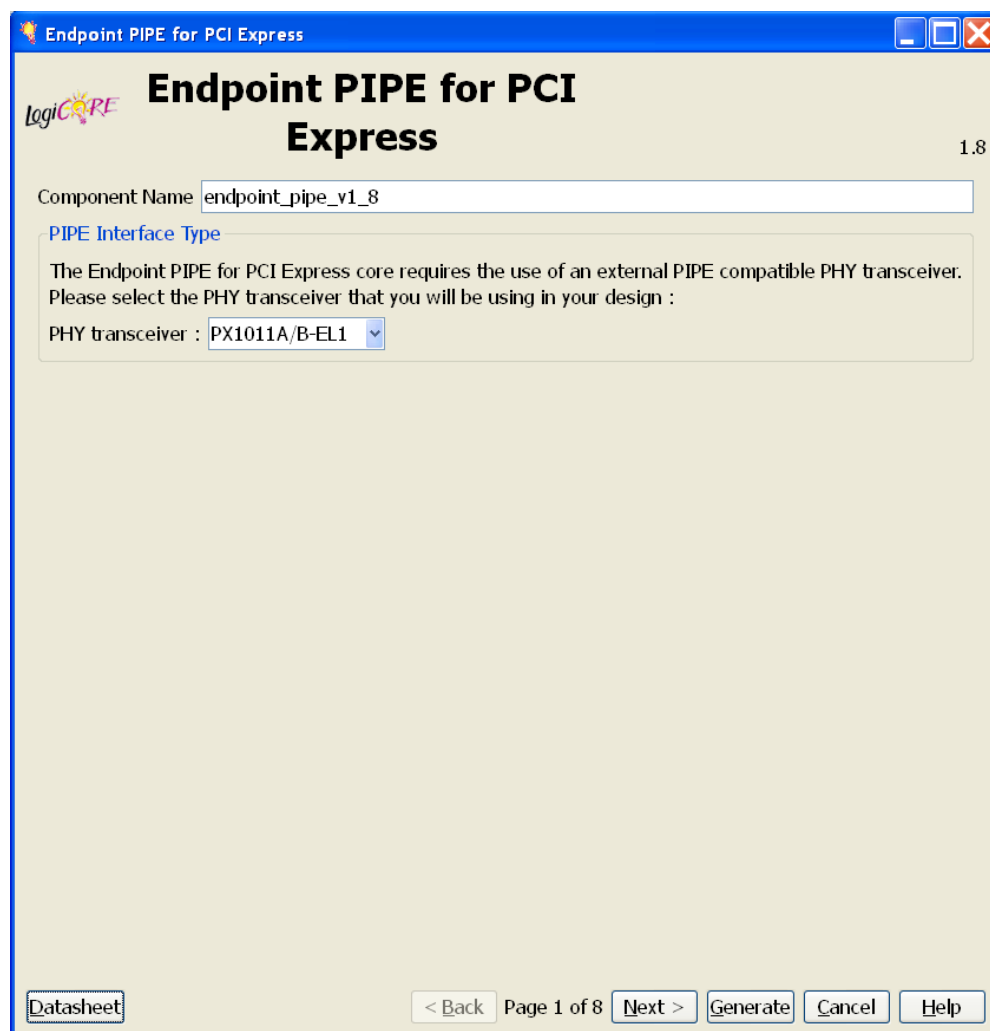


Figure 5-1: PCI Express Basic Parameters: Screen 1

**Endpoint PIPE for PCI Express** 1.8

**ID Initial Values**

Vendor ID :  Range: 0000..FFFF (Hex)

Device ID :  Range: 0000..FFFF (Hex)

Revision ID :  Range: 00..FF (Hex)

Subsystem Vendor ID :  Range: 0000..FFFF (Hex)

Subsystem ID :  Range: 0000..FFFF (Hex)

**Class Code**

Base Class :  Range: 00..FF (Hex)

Sub-Class :  Range: 00..FF (Hex)

Interface :  Range: 00..FF (Hex)

Class Code :  (Hex)

**Cardbus CIS Pointer**

Cardbus CIS Pointer :  (Hex)

[Datasheet](#) [< Back](#) Page 2 of 8 [Next >](#) [Generate](#) [Cancel](#) [Help](#)

Figure 5-2: PCI Express Basic Parameters: Screen 2

## Component Name

Base name of the output files generated for the core. The name must begin with a letter and be composed of the following characters: a to z, 0 to 9, and "\_."

## PIPE Interface Type

The PHY Device selection is used to select from different devices provided by PHY vendors; currently, only one is available.

## ID Initial Values

- **Vendor ID.** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, 10EEh, is the Vendor ID for Xilinx. Enter a vendor identification number here. FFFFh is reserved.

- **Device ID.** A unique identifier for the application; the default value is 0007h. This field can be any value; change this value for the application.
- **Revision ID.** Indicates the revision of the device or application; an extension of the Device ID. The default value is 00h; enter values appropriate for the application.
- **Subsystem Vendor ID.** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EEh. Typically, this value is the same as Vendor ID. Note that setting the value to 0000h can cause compliance testing issues.
- **Subvendor ID.** Further qualifies the manufacturer of the device or application. This value is typically the same as the Vendor ID; default value is 0007h. Note that setting the value 0000h can cause compliance testing issues.

## Class Code

The Class Code identifies the general function of a device, and is divided into three byte-size fields:

- **Base Class.** Broadly identifies the type of function performed by the device.
- **Sub-Class.** More specifically identifies the device function.
- **Interface.** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

Class code encoding can be found at [www.pcisig.com](http://www.pcisig.com).

## Cardbus CIS Pointer

Used in cardbus systems and points to the Card Information Structure for the cardbus card. If this field is non-zero, an appropriate Card Information Structure must exist in the correct location. The default value is 0000\_0000h; value range is 0000\_0000h-FFFF\_FFFFh.



# Base Address Registers

The Base Address Register (BAR) screens(Figure 5-3 andFigure 5-4) let you set the base address register space. Each Bar (0 through 5) represents a 32-bit parameter.

**Endpoint PIPE for PCI Express**

LogiCORE 1.8

**Base Address Registers (1 of 2)**

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

**BAR 0 Options**

☒ BAR 0 Type : IO ☐ 64 bit (consumes BAR 1)  
 Size : 64 Kilobytes ☐ Prefetchable  
 Value : FFFF0001 (Hex)

**BAR 1 Options**

☒ BAR 1 Type : Memory ☐ 64 bit (consumes BAR 2)  
 Size : 64 Kilobytes ☐ Prefetchable  
 Value : FFFF0000 (Hex)

**BAR 2 Options**

☒ BAR 2 Type : Memory ☒ 64 bit (consumes BAR 3)  
 Size : 1 Megabytes ☐ Prefetchable  
 Value : FFF00004 (Hex)

**BAR 3 Options**

☐ BAR 3 Type : IO ☐ 64 bit (consumes BAR 4)  
 Size : 64 Kilobytes ☐ Prefetchable  
 Value : FFFFFFFF (Hex)

Datasheet < Back Page 3 of 8 Next > Generate Cancel Help

Figure 5-3: Endpoint PIPE BAR Options: Screen 3

**Endpoint PIPE for PCI Express**

LogiCORE

**Endpoint PIPE for PCI Express**

1.8

Base Address Registers (2 of 2)

**BAR 4 Options**

☐ BAR 4 Type : IO ☐ 64 bit (consumes BAR 5)

Size : 64 Kilobytes ☐ Prefetchable

Value : 00000000 (Hex)

**BAR 5 Options**

☐ BAR 5 Type : IO ☐ 64 bit (consumes BAR 5)

Size : 64 Kilobytes ☐ Prefetchable

Value : 00000000 (Hex)

**Expansion ROM Base Address Register**

☒ Expansion Rom Size : 4 Kilobytes

Value : FFFFFFF01 (Hex)

Datasheet < Back Page 4 of 8 Next > Generate Cancel Help

Figure 5-4: Endpoint PIPE BAR Options: Screen 4

## Base Address Register Overview

The Endpoint PIPE for PCIe core supports up to six 32-bit Base Address Registers (BARs) or three 64-bit BARs, and the Expansion ROM BAR. BARs can be one of two sizes:

- **32-bit BARs.** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for Memory or I/O.
- **64-bit BARs.** The address space can be as small as 128 bytes or as large as 8 exabytes. Used for Memory only.

All BAR registers share the following options:

- **Checkbox.** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type.** BARs can either be I/O or Memory.

- **I/O.** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs.
- **Memory.** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible to the user.
- **Size**
  - **Memory:** When Memory and 64-bit are *not selected*, the size can range from 128 bytes to 2 gigabytes. When Memory and 64-bit are *selected*, the size can range between 128 bytes and 8 exabytes.
  - **I/O.** When selected, the size can range from 128 bytes to 2 gigabytes.
- **Prefetchable.** Identifies the ability of the memory space to be prefetched.
- **Value.** The value assigned to the BAR based on the current selections.

For more information about managing the Base Address Register settings, see [“Managing Base Address Register Settings.”](#)

### Expansion ROM Base Address Register

If selected, the Expansion ROM is activated and can be a value from 2 kilobytes to 4 gigabytes.

## Managing Base Address Register Settings

Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate GUI settings for the desired base address register.

Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4Kbytes in size should be avoided. The maximum I/O space allowed is 256 bytes. The use of I/O space should be avoided in all new designs.

Prefetchability is the ability of memory space to be prefetched. A memory space is prefetchable if there are no side-effects on reads (that is, data will not be destroyed by reading, as from a RAM). Byte write operations can be merged into a single double-word write, when applicable.

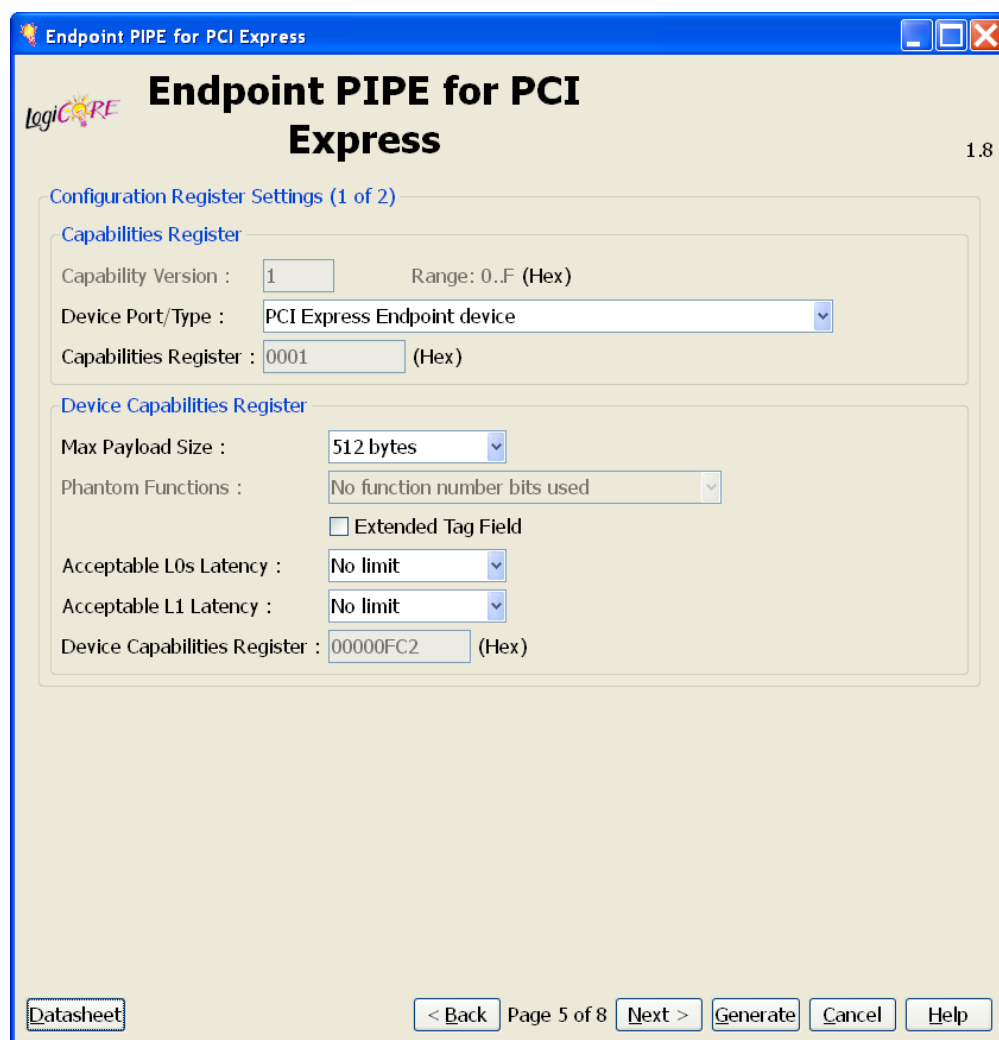
When configuring the core as a PCI Express Endpoint (non-Legacy), 64-bit addressing must be supported for all BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all BARs that do not have the prefetchable bit set. The prefetchable bit related requirement does not apply to a Legacy Endpoint. In either of the above cases (PCI Express Endpoint or a Legacy Endpoint), the minimum memory address range supported by a BAR is 128 bytes.

## Disabling Unused Resources

For best results, disable unused base address registers to trim associated logic. A base address register is disabled by deselecting unused BARs in the GUI.

## Configuration Register Settings

The Configuration Registers screens (Figure 5-5 and Figure 5-6) let you set options for the Capabilities register, Device Capabilities register, and Link Capabilities register.



**Endpoint PIPE for PCI Express** 1.8

Configuration Register Settings (1 of 2)

**Capabilities Register**

Capability Version : 1 Range: 0..F (Hex)

Device Port/Type : PCI Express Endpoint device

Capabilities Register : 0001 (Hex)

**Device Capabilities Register**

Max Payload Size : 512 bytes

Phantom Functions : No function number bits used

☐ Extended Tag Field

Acceptable L0s Latency : No limit

Acceptable L1 Latency : No limit

Device Capabilities Register : 0000FC2 (Hex)

[Datasheet](#) < Back Page 5 of 8 Next > Generate Cancel Help

Figure 5-5: Endpoint PIPE Configuration Settings: Screen 5

Figure 5-6: Endpoint PIPE Configuration Settings: Screen 6

## Capabilities Register

- **Capability Version.** Indicates PCI-SIG defined PCI Express capability structure version number; this value can not be changed.
- **Device Port Type.** Indicates the PCI Express logical device type. Note that some values are allowed even though the core does not support this functionality, as denoted by the word *UNSUPPORTED* next to the value.
- **Capabilities Register.** Displays the value of the Capabilities register sent to the core and is not editable.

## Device Capabilities Register

- **Max Payload Size:** Indicates the maximum payload size that the device/function can support for TLPs.
- **Phantom Functions:** This field is not supported.

- **Extended Tag Field.** Indicates the maximum supported size of the Tag field as a Requester. When selected, indicates 8-bit Tag field support. When deselected, indicates 5-bit Tag field support.
- **Acceptable L0s Latency:** Indicates the acceptable total latency that an Endpoint can withstand due to the transition from L0s state to the L0 state.
- **Acceptable L1 Latency:** Indicates the acceptable latency that an Endpoint can withstand due to the transition from L1 state to the L0 state.
- **Device Capabilities Register:** Displays the value of the Device Capabilities register sent to the core and is not editable.

## Link Capabilities Register

This section is used to set the Link Capabilities register.

- **Maximum Link Speed:** Indicates the maximum link speed of the given PCI Express Link. This value is set to 2.5 Gbps and is not editable.
- **Maximum Link Width:** Indicates the maximum link width implemented by the core. This value is set to X1 and is not editable.
- **Enable ASPM L1 Support:** Indicates the level of ASPM supported on the given PCI Express Link. When selected, L0s and L1 are supported; when deselected, only L0s entry is supported.
- **Link Capabilities Register:** Displays the value of the Link Capabilities register sent to the core and is not editable.

## MSI Control Register

This section is used to set the MSI Control register.

- **Multiple Message Capable.** Selects the number of MSI vectors to request from the Root Complex.

## Advanced Settings

The Advanced Settings screens (Figure 5-7 and Figure 5-8) includes settings for the Transaction layer, Physical layer, user configuration space, power consumption, and power dissipation options.

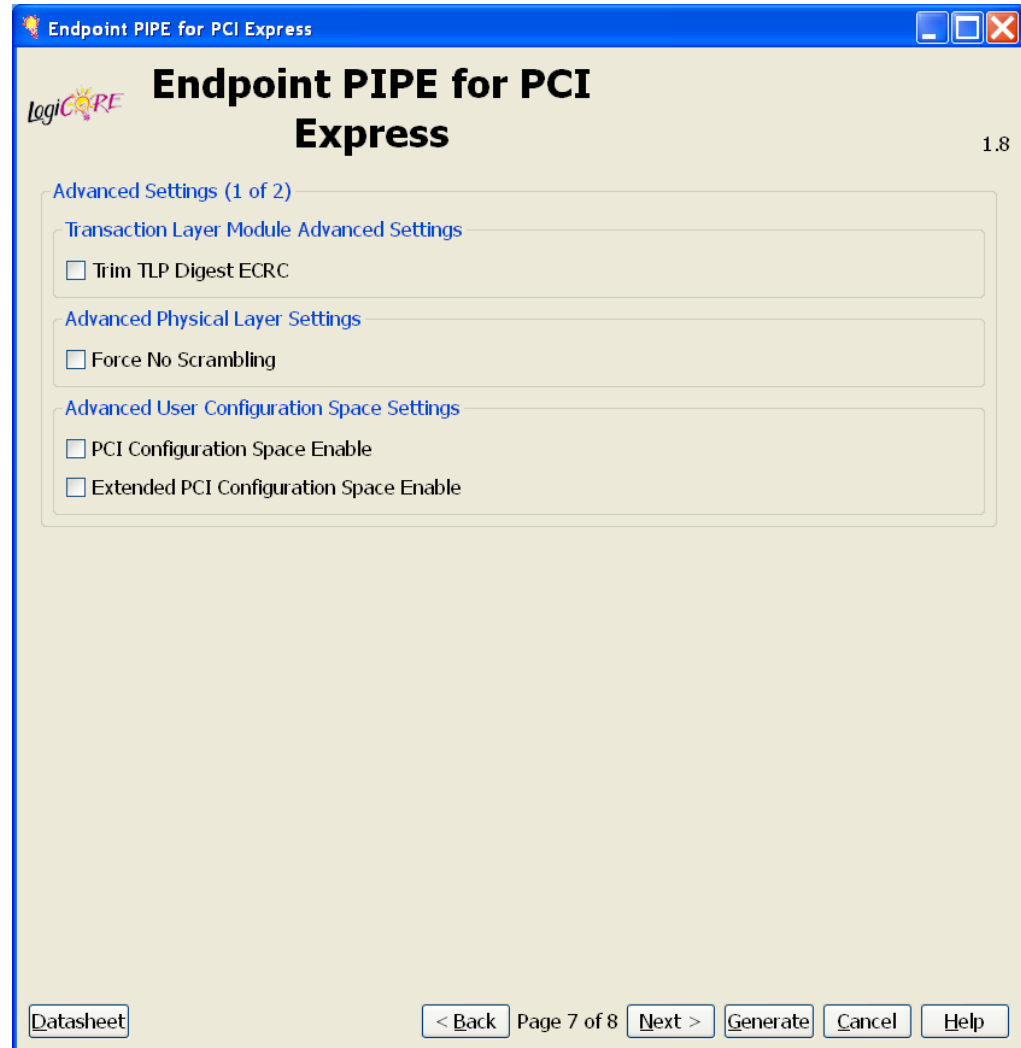
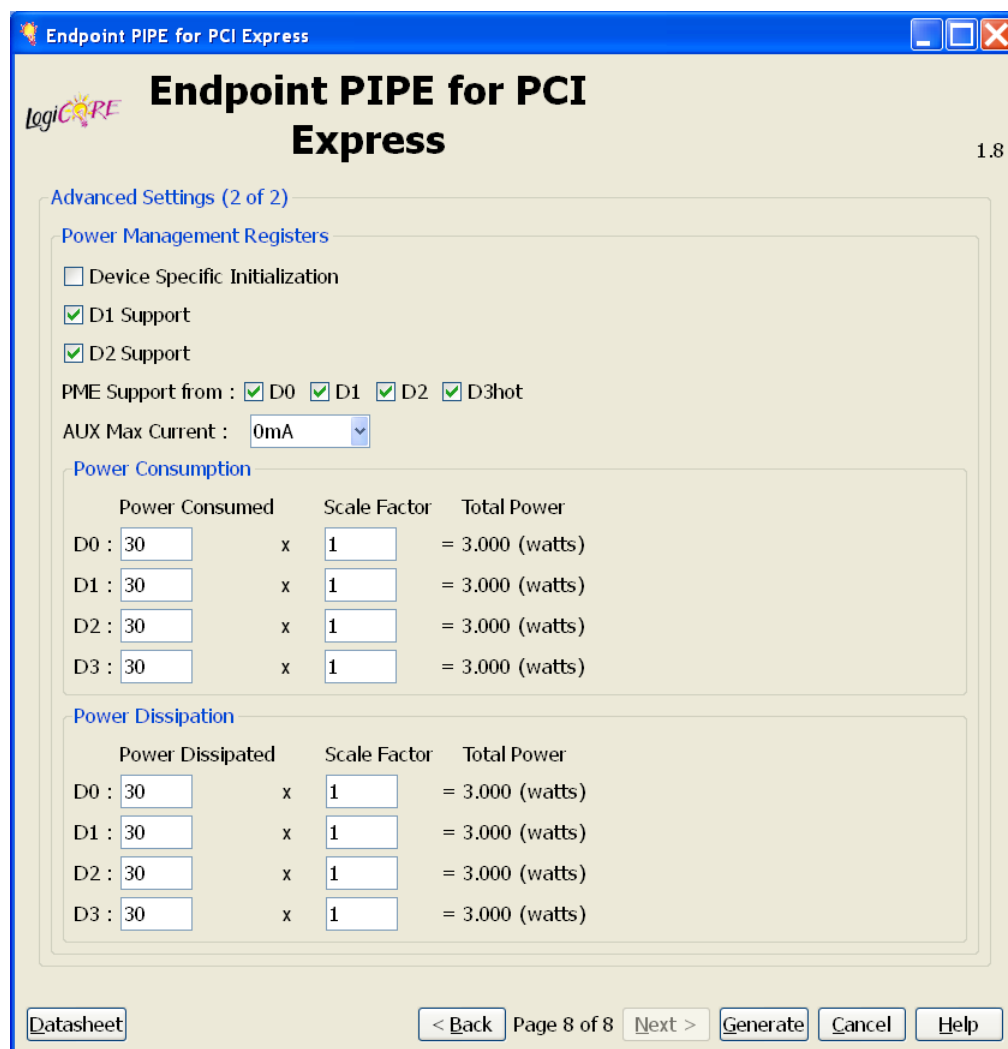


Figure 5-7: Endpoint PIPE Advanced Settings: Screen 7



Endpoint PIPE for PCI Express 1.8

logiCORE

## Endpoint PIPE for PCI Express

Advanced Settings (2 of 2)

### Power Management Registers

☐ Device Specific Initialization

☒ D1 Support

☒ D2 Support

PME Support from : ☒ D0 ☒ D1 ☒ D2 ☒ D3hot

AUX Max Current : 0mA

### Power Consumption

	Power Consumed		Scale Factor	Total Power
D0 :	30	x	1	= 3.000 (watts)
D1 :	30	x	1	= 3.000 (watts)
D2 :	30	x	1	= 3.000 (watts)
D3 :	30	x	1	= 3.000 (watts)

### Power Dissipation

	Power Dissipated		Scale Factor	Total Power
D0 :	30	x	1	= 3.000 (watts)
D1 :	30	x	1	= 3.000 (watts)
D2 :	30	x	1	= 3.000 (watts)
D3 :	30	x	1	= 3.000 (watts)

[Datasheet](#) [< Back](#) Page 8 of 8 [Next >](#) [Generate](#) [Cancel](#) [Help](#)

Figure 5-8: Endpoint PIPE Advanced Settings: Screen 8

## Transaction Layer Module

**Trim TLP Digest ECRC.** Causes the core to trim any TLP digest from an inbound packet before presenting it to the user.

## Advanced Physical Layer

**Force No Scrambling.** Used for diagnostic purposes only and should never be enabled in a working design. Setting this bit results in the data scramblers being turned off so that the serial data stream can be analyzed.



## Advanced User Configuration Space Settings

- **PCI Configuration Space Enable.** Allows the User Application to add/implement PCI Legacy capability registers beyond address BFh. This option should be selected if the User Application implements such a legacy capability configuration space, starting at C0h.
- **Extended PCI Configuration Space Enable.** Allows the User Application to add/implement PCI-Express extended capability registers beyond address 1FFh. This box should be checked if the user-application implements such an extended capability configuration space starting at 200h.

## Power Management Registers

- **Device Specific Initialization.** This bit indicates whether special initialization of this function is required (beyond the standard PCI configuration header) before the generic class device driver is able to use it. When selected, this option indicates that the function requires a device specific initialization sequence following transition to the D0 uninitialized state. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **D1 Support.** When selected, this option indicates that the function supports the D1 Power Management State. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **D2 Support.** When selected, this option indicates that the function supports the D2 Power Management State. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **PME Support From:** When an option is selected, indicates the power states in which the function can assert PME#. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **AUX Max Current.** Indicates the 3.3 V auxiliary current requirements for the PCI function. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.

## Power Consumption

For information about power consumption, see section 3.2.6 of the *PCI Bus Power Management Interface Specification Revision 1.2*

## Power Dissipated

For information about power dissipation, see section 3.2.6 of the *PCI Bus Power Management Interface Specification Revision 1.2*.



## Designing with the Core

This chapter provides instructions for designing a device using the Endpoint PIPE for PCI Express core, and assumes knowledge of the PCI Express Transaction Layer Packet (TLP) header fields. Header fields are defined in the *PCI Express Base Specification v1.1*, “Chapter 2, Transaction Layer Specification.”

### Designing with the Transaction Interface

This section describes designing with the PCI Express 32-bit Endpoint PIPE Transaction interface, including

- ◆ “Transmitting Outbound Packets”
- ◆ “Receiving Inbound Packets”
- ◆ “Accessing Configuration Space Registers”
- ◆ “Additional Packet Handling Requirements”
- ◆ “Power Management”
- ◆ “Generating Interrupt Requests”

### TLP Format on the 32-bit Transaction Interface

Data is transmitted and received in Big-Endian order as required by the *PCI Express Base Specification*. See Chapter 2 of the *PCI Express Base Specification* for detailed information about TLP packet ordering. [Figure 6-1](#) represents a typical 32-bit addressable Memory Write Request TLP (as illustrated in Chapter 2 of the specification).

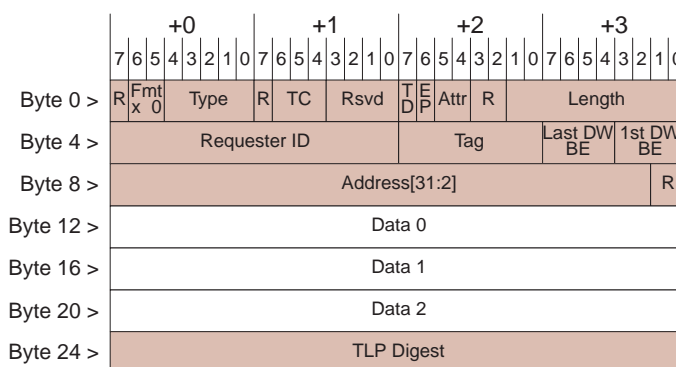


Figure 6-1: PCI Express Base Specification Byte Order

When using the 32-bit Transaction interface, packets are arranged on the 32-bit data path in the same order as that shown in [Figure 6-1](#). Byte 0 of the packet appears on `trn_td[31:24]` (outbound) or `trn_rd[31:24]` (inbound) of the first DWORD, byte 1 on `trn_td[23:16]` or `trn_rd[23:16]`, and so forth. Byte 4 of the packet then appears on `trn_td[31:24]` or `trn_rd[31:24]` of the second DWORD. The Header section of the packet consists of either three or four DWORDs, determined by the TLP format and type as described in section 2.2 of the *PCI Express Base Specification*.

Packets sent to the core for transmission must follow the formatting rules for Transaction Layer Packets (TLPs) as specified in Chapter 2 of the *PCI Express Base Specification*. The User Application is responsible for ensuring its packets' validity, as the core does not check packet validity or validate packets. The exact fields of a given TLP vary depending on the packet type being transmitted.

The presence of a TLP Digest or ECRC is indicated by the value of TD field in the TLP Header section. When TD=1, a correctly computed CRC32 remainder DWORD is expected to be presented as the last DWORD of the packet. The CRC32 remainder DWORD is not included in the length field of the TLP header. The User Application must calculate and present the TLP Digest as part of the packet when transmitting packets. Upon receiving packets with a TLP Digest present, the User Application must check the validity of the CRC32 based on the contents of the packet. The PCI Express Endpoint PIPE for PCIe core does not check the TLP Digest for incoming packets.

## Transmitting Outbound Packets

### Basic TLP Transmit Operation

The PCI Express Endpoint PIPE for PCIe core automatically transmits the following types of packets:

- Completions to a remote device in response to Configuration Space requests
  - Error-message responses to inbound requests malformed or unrecognized by the core
- Note:** Certain unrecognized requests, for example, unexpected completions, can only be detected by the User Application, which is responsible for generating the appropriate response.

The User Application is responsible for constructing the following types of outbound packets:

- Memory and I/O Requests to remote devices.
- Completions in response to requests to the User Application, for example, a Memory Read Request.
- Completions in response to user-implemented Configuration Space requests when enabled. These requests include PCI Legacy capability registers beyond address BFh and PCI Express extended capability registers beyond address 1FFh.

**Note:** For important information about accessing user-implemented Configuration Space while in a low-power state, see [“Power Management,” page 100](#).

Table 3-9, page 33 defines the transmit Transaction interface signals. To transmit a TLP, the User Application must perform the following sequence of events on the Transaction interface:

1. The User Application logic asserts `trn_tsrc_rdy_n`, `trn_tsof_n` and presents the first TLP DWORD on `trn_td[31:0]` when it is ready to transmit data.
2. The User Application asserts `trn_tsrc_rdy_n` and presents the remainder of the TLP DWORDs on `trn_td[31:0]` for subsequent clock cycles (for which the core asserts `trn_tdst_rdy_n`).
3. The User Application asserts `trn_tsrc_rdy_n` and `trn_teof_n` together with the last DWORD of data.
4. At the next clock cycle, the User Application deasserts `trn_tsrc_rdy_n` to signal the end of valid transfers on `trn_td[31:0]`.

Figure 6-2 illustrates a 3-DW TLP header without a data payload; an example is a 32-bit addressable Memory Read request.

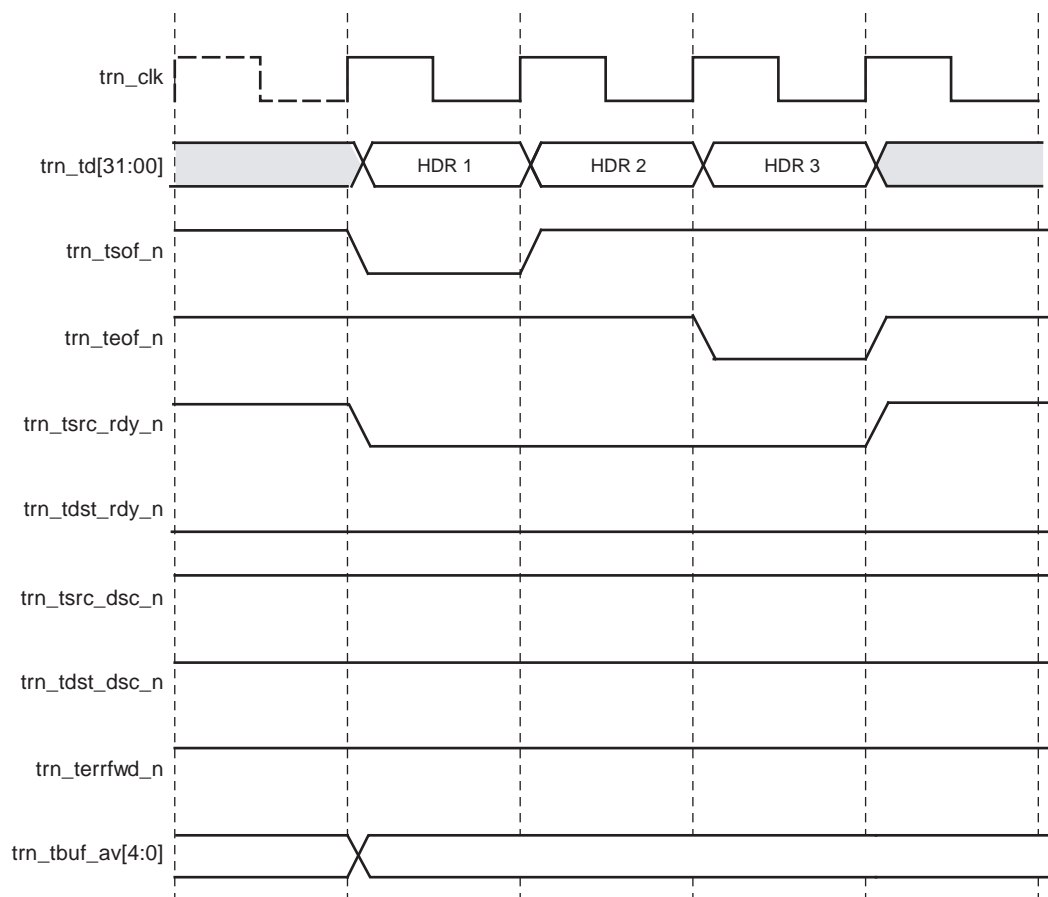


Figure 6-2: TLP 3-DW Header without Payload

Figure 6-3 illustrates a 4-DW TLP header without a data payload; an example is a 64-bit addressable Memory Read request.

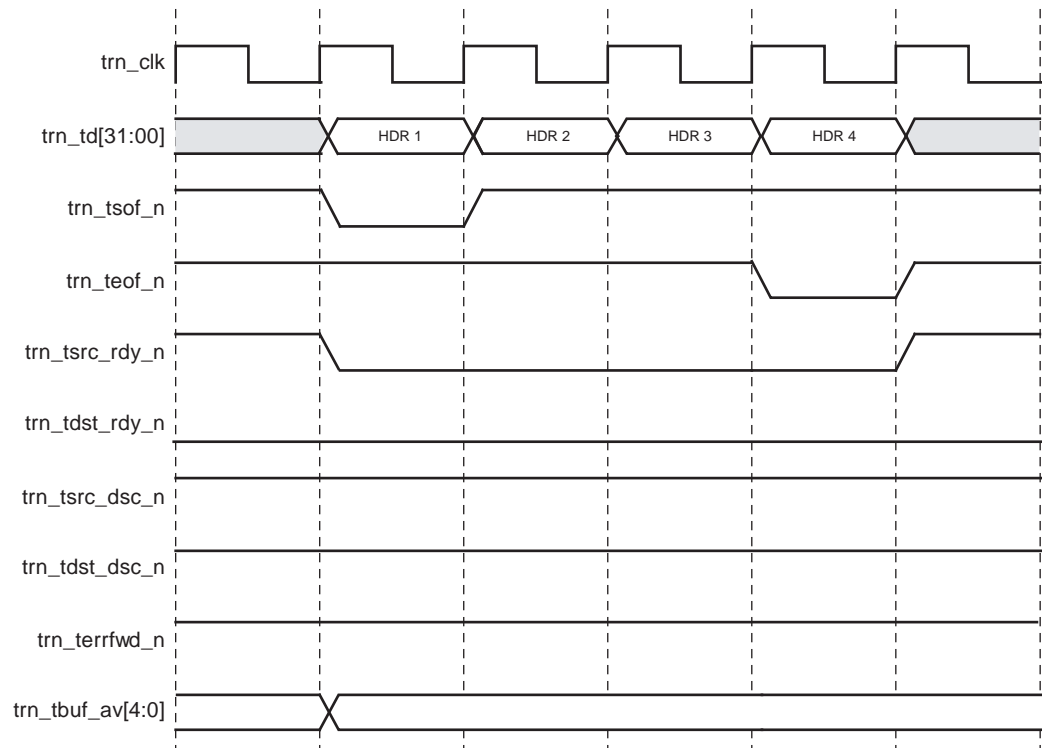


Figure 6-3: TLP with 4-DW Header without Payload

Figure 6-4 illustrates a 3-DW TLP header with a data payload; an example is a 32-bit addressable Memory Write request.

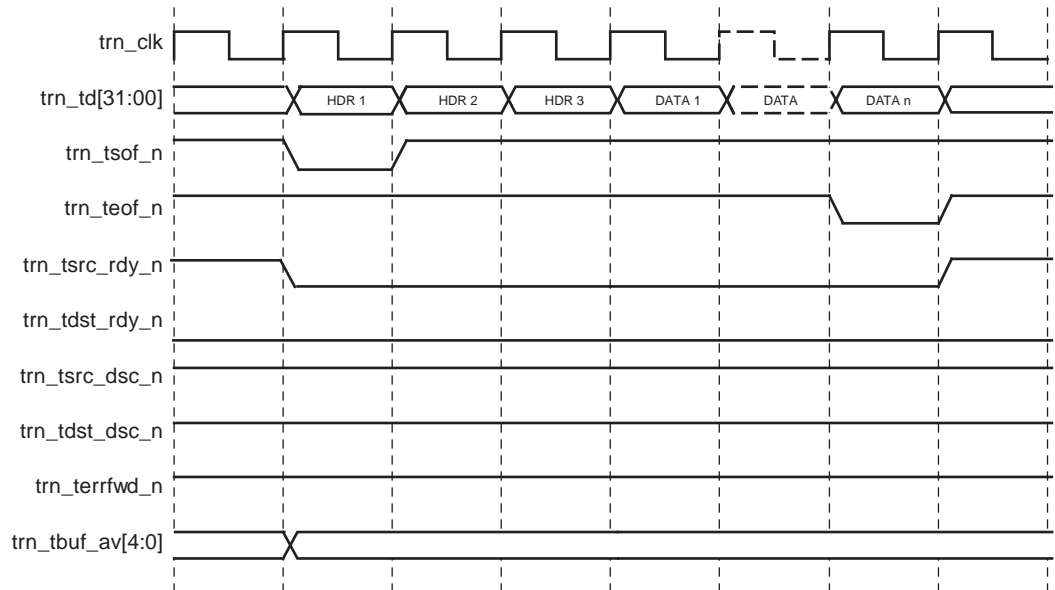


Figure 6-4: TLP with 3-DW Header with Payload

Figure 6-5 illustrates a 4-DW TLP header with a data payload; an example is a 32-bit addressable Memory Write request.

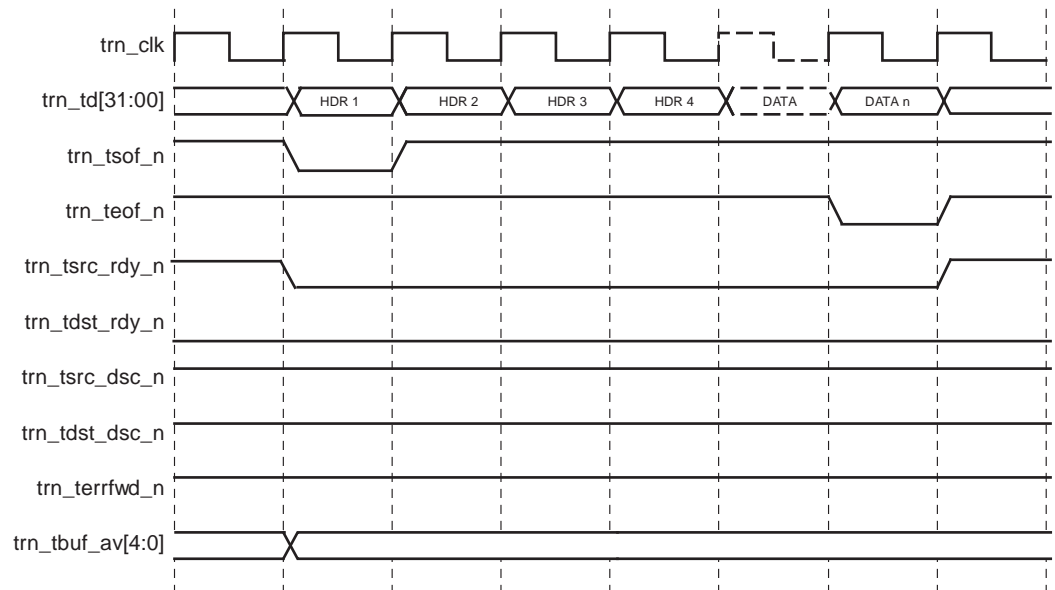


Figure 6-5: TLP with 4-DW Header with Payload

## Presenting Back-to-Back Transactions on the Transaction Interface

The User Application can present back-to-back TLPs on the Transaction interface to maximize bandwidth utilization. [Figure 6-6](#) illustrates back-to-back TLPs presented on the Transaction interface. The User Application asserts `trn_tsof_n` and presents a new TLP on the next clock cycle after asserting `trn_teof_n` for the previous TLP.

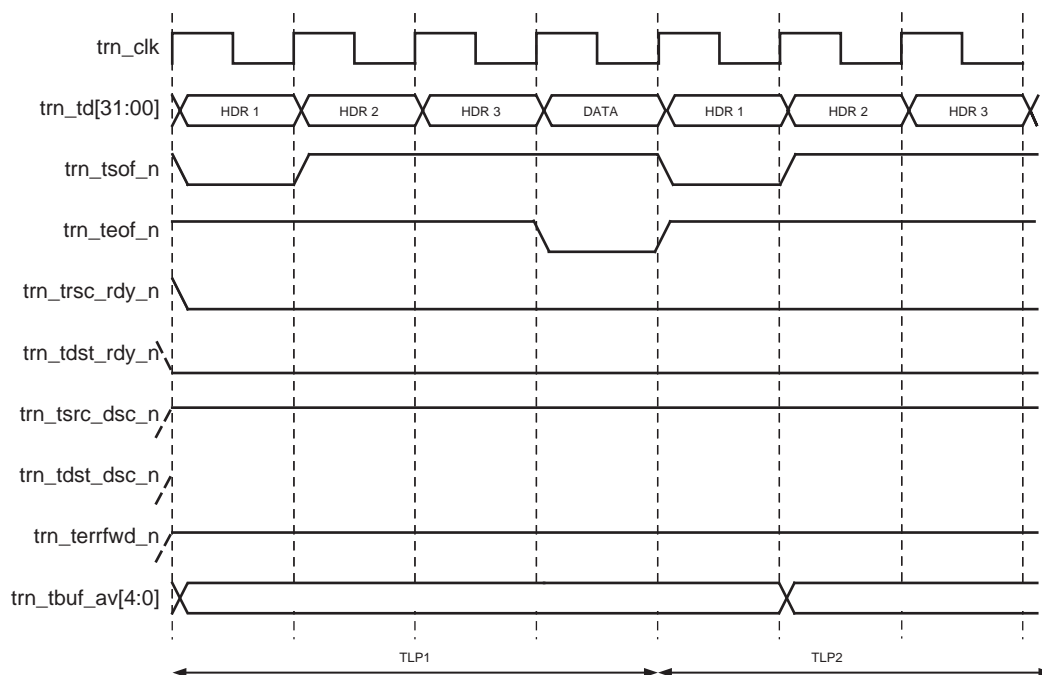


Figure 6-6: Back-to-back Transaction on Transaction Interface



## Source Throttling on the Transmit Data Path

The Endpoint core Transaction interface lets the User Application throttle back if it has no data present on `trn_td[31:0]`. When this condition occurs, the User Application deasserts `trn_tsrc_rdy_n`, which instructs the Endpoint core Transaction interface to disregard data presented on `trn_td[31:0]`. Figure 6-7 illustrates the source throttling mechanism, where the User Application does not have data to present every clock cycle, and for this reason must deassert `trn_tsrc_rdy_n` during these cycles.

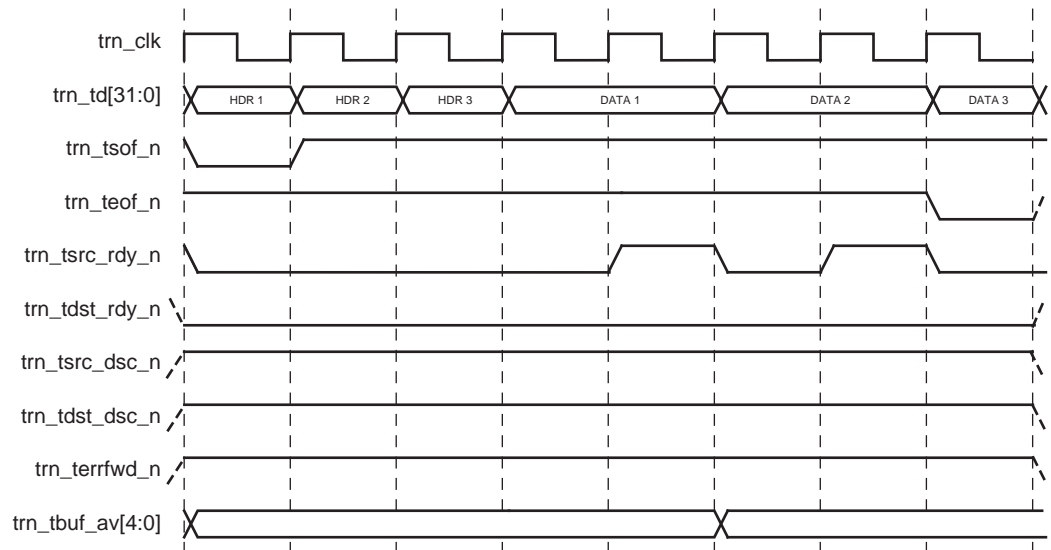


Figure 6-7: Source Throttling on the Transmit Data Path

## Destination Throttling of the Transmit Data Path

The Endpoint core Transaction interface throttles the User Application if there is no space left for a new TLP in its transmit buffer pool. This can occur if the link partner is not processing incoming packets at a rate equal to or greater than the rate at which the User Application is presenting TLPs. Figure 6-8 illustrates the deassertion of `trn_tdst_rdy_n` to throttle the User Application when the core's internal transmit buffers are full.

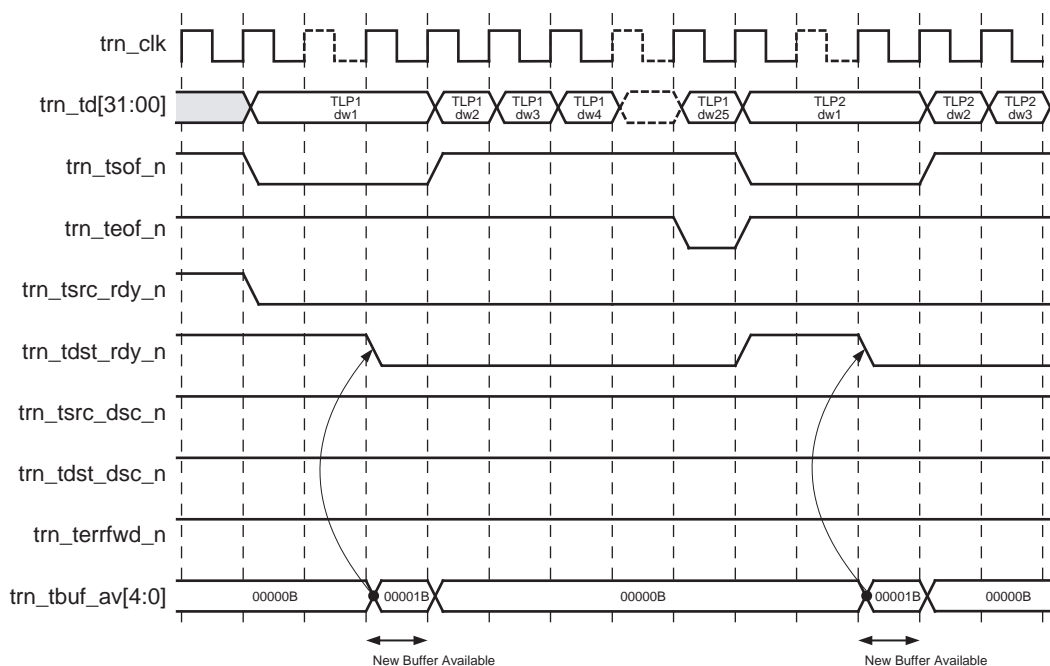
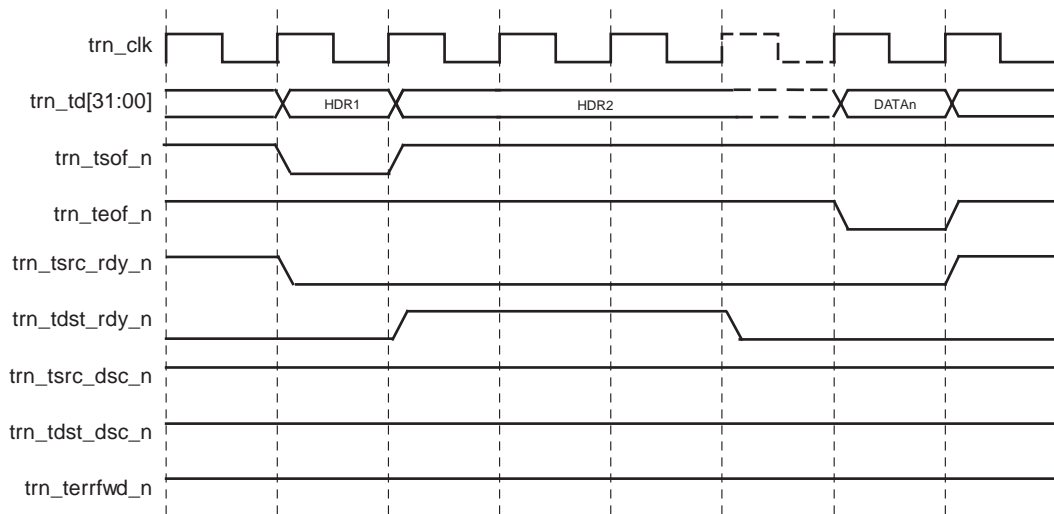


Figure 6-8: Destination Throttling of the Endpoint Transaction Interface

If the core Transaction interface accepts the start of a TLP by asserting `trn_tdst_rdy_n`, it is guaranteed to accept the complete TLP with a size up to the value contained in the `Max_Payload_Size` field of the PCI Express Device Control Register (Offset 08H). The core Transaction interface deasserts `trn_tdst_rdy_n` *only* after it has accepted the TLP completely and has no buffer space available for a new TLP with one exception: the core can deassert `trn_tdst_rdy_n` after accepting `trn_tsof_n` (first DWORD) if it is transmitting a configuration Completion TLP or an error Message TLP response. The core subsequently asserts `trn_tdst_rdy_n` after transmitting the internal TLP and keeps it asserted for duration of the TLP, as displayed in Figure 6-9.



**Figure 6-9: Destination Throttling of the Endpoint Transaction Interface**

The Endpoint core Transaction interface throttles the User Application when the Power State field in Power Management Control/Status Register (Offset 0x4) of the PCI Power Management Capability Structure is changed to a non-D0 state. When this occurs, any on-going TLP is accepted completely and **trn\_tdst\_rdy\_n** is subsequently deasserted, disallowing the User Application from initiating any new transactions—for the duration that the core is in the non-D0 power state.

## Discontinuing Transmission of Transaction by Source

The Endpoint core Transaction interface lets the User Application terminate transmission of a TLP by asserting `trn_tsrc_dsc_n`. Both `trn_tsrc_rdy_n` and `trn_tdst_rdy_n` must be asserted together with `trn_tsrc_dsc_n` for the TLP to be discontinued. The signal `trn_tsrc_dsc_n` must not be asserted together with `trn_tsof_n`. It can be asserted on any cycle after `trn_tsof_n` deasserts to and including the assertion of `trn_teof_n`. Asserting `trn_tsrc_dsc_n` has no effect if no TLP transaction is in progress on the Transaction interface. Figure 6-10 illustrates the User Application discontinuing a packet using `trn_tsrc_dsc_n`. Asserting `trn_teof_n` together with `trn_tsrc_dsc_n` is optional.

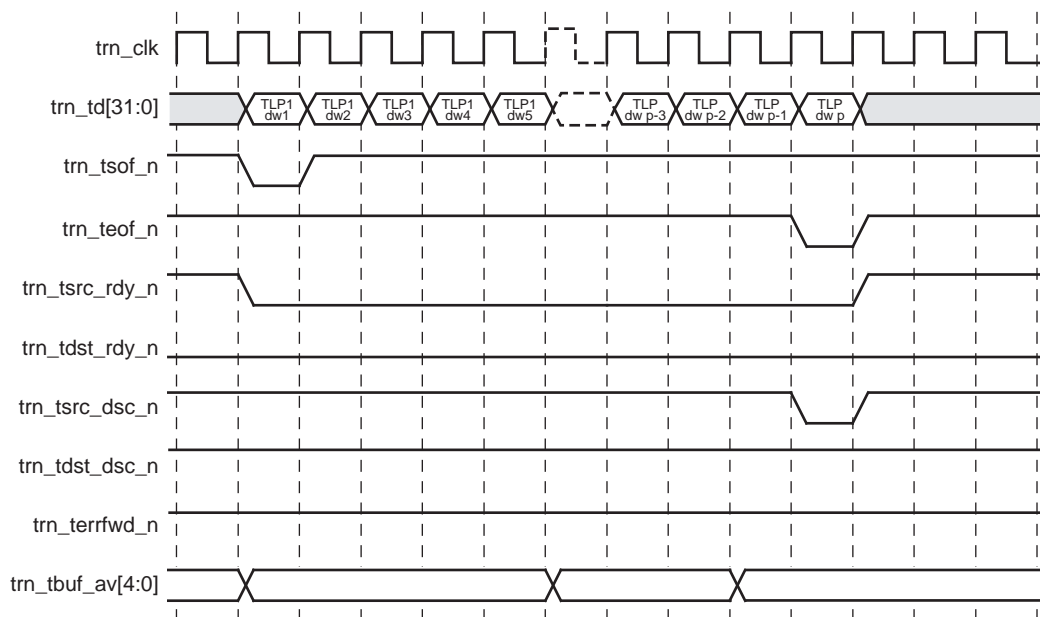


Figure 6-10: Source Driven Transaction Discontinue on Transaction Interface

## Discontinuing Transmission of Transaction by Destination

The Endpoint core Transaction interface discontinues an ongoing TLP transfer if the presented TLP payload size exceeds the value contained in the Max\_Payload\_Size field of the PCI Express Device Control Register (Offset 08H) by asserting `trn_tdst_dsc_n` for one clock cycle. After `trn_tdst_dsc_n` is asserted, the Transaction interface discards already presented TLP data and disregards the remainder of the presented TLP. The User Application can monitor `trn_tdst_dsc_n`, and stop presenting the remainder of the TLP, to efficiently utilize transmit interface bandwidth. Figure 6-11 illustrates the Endpoint core discontinuing a packet using `trn_tdst_dsc_n`.

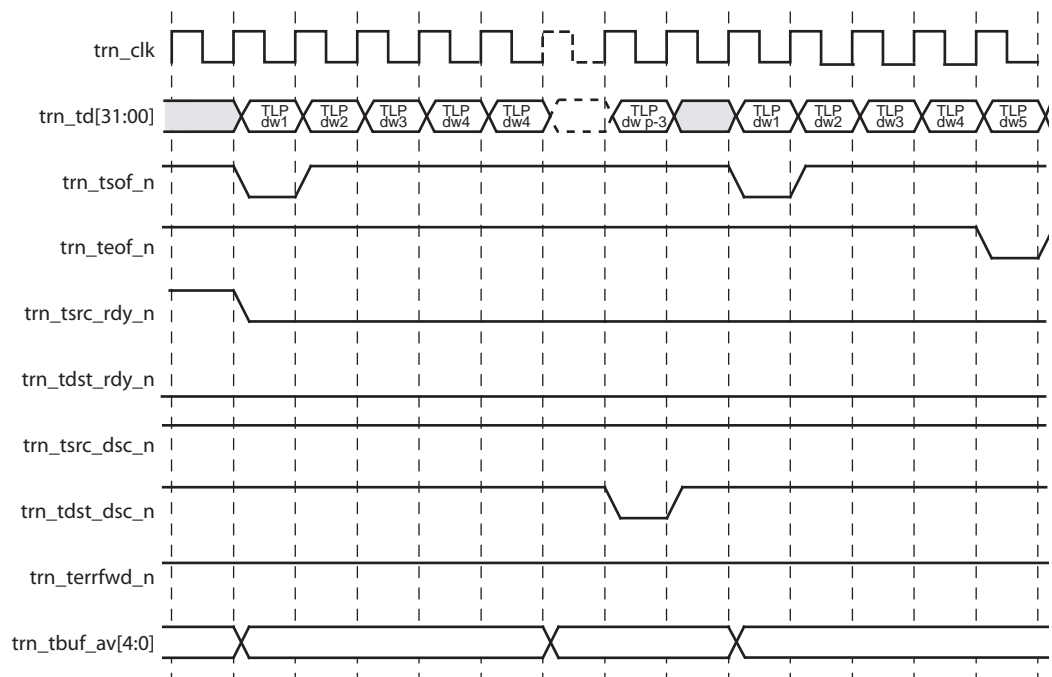


Figure 6-11: Destination Driven Transaction Discontinue on Transaction Interface

## Packet Data Poisoning on the Transmit Transaction Interface

The User Application can use one of the following mechanisms to mark the data payload of a transmitted TLP as poisoned:

- Set EP=1 in the TLP header. This mechanism can be used if the payload is known to be poisoned when the first DWORD of the header is presented to the Endpoint core on the TRN interface.
- Assert trn\_terr\_fwd\_n for at least 1 valid data transfer cycle any time during the packet transmission, as shown in Figure 6-12. This causes the Endpoint core to set EP=1 in the TLP header when it transmits the packet onto the PCI Express fabric. This mechanism can be used if the User Application does not know whether a packet can be poisoned at the start of packet transmission.

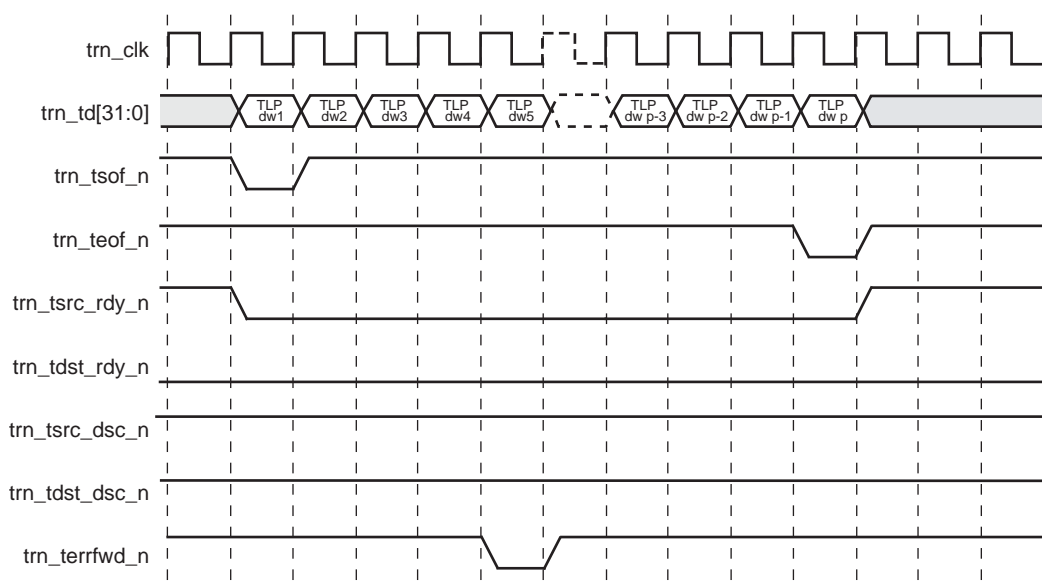


Figure 6-12: Packet Data Poisoning on the Transaction Interface

## Appending ECRC to Protect TLPs

If the User Application needs to send a TLP Digest associated with a TLP, it must construct the TLP header such that the TD bit is set and the 1-DWORD TLP Digest is properly computed and appended after the last valid TLP payload section (if applicable).

## Transmit Buffers

The Endpoint core Transaction interface provides `trn_tbuf_av`, an instantaneous indication of the number of Max\_Payload\_Size buffers available for use in the transmit buffer pool. Table 6-1 defines the number of transmit buffers available and maximum supported payload size for a specific core. Cores from the entire Endpoint for PCI Express family are included for reference.

Table 6-1: Transmit Buffers Available

Core Name	Buffers	Supported Max Payload Size
1-lane 64-bit Endpoint	16	512 bytes
4-lane 64-bit Endpoint	16	512 bytes
8-lane 64-bit Endpoint	32	256 bytes
1-lane 32-bit Endpoint	8	512 bytes
4-lane 32-bit Endpoint	16	512 bytes
1-lane Endpoint PIPE	8	512 bytes

Each buffer can hold one maximum sized TLP. A maximum sized TLP is a TLP with a 4-DWORD header plus a data payload equal to the MAX\_PAYLOAD\_SIZE of the core (as defined in the Device Capability register) plus a TLP Digest. Note that after the link is trained, the root complex sets the MAX\_PAYLOAD\_SIZE value in the Device Control register. This value is equal to or less than the value advertised by the core's Device Capability register. For more information about these registers, see section 7.8 of the *PCI Express Base Specification*. A TLP is held in the core's transmit buffer until the link partner acknowledges receipt of the packet, at which time the buffer is released and a new TLP can be loaded into it by the User Application.

For example, if the 8-lane 64-bit Endpoint core is being used, there are 32 total transmit buffers. Each of these buffers can hold at a maximum one 64-bit Memory Write Request (4 DWORD header) plus 256 bytes of data (64 DWORDs) plus TLP Digest (1 DWORD) for a total of 69 DWORDs. Note that this example assumes the root complex set the MAX\_PAYLOAD\_SIZE register of the Device Control register to 256 bytes, which is the maximum capability advertised by this core. For this reason, at any given time, this core could have 32 of these 69 DWORD TLPs awaiting transmittal. There is no sharing of buffers among multiple TLPs, so even if user is sending smaller TLPs such as 32-bit Memory Read request with no TLP Digest totaling 3 DWORDs only per TLP, each transmit buffer still holds only one TLP at any time.

The internal transmit buffers are shared between the User Application and the core's Configuration Management Module (CMM). Due to this, the `trn_tbuf_av` bus may fluctuate even if the User Application is not transmitting packets. The CMM generates completion TLPs in response to configuration reads or writes, interrupt TLPs at the request of the user application, and message TLPs when needed.

The Transmit Buffers Available indication enables the User Application to completely utilize the PCI transaction ordering feature of the core transmitter. The transaction ordering rules allow for Posted and Completion TLPs to bypass Non-Posted TLPs. See section 2.4 of the *PCI Express Base Specification* for more information about ordering rules.

The core supports the transaction ordering rules and promotes Posted and Completion packets ahead of blocked Non-Posted TLPs. Non-Posted TLPs can become blocked if the link partner is in a state where it momentarily has no Non-Posted receive buffers available, which it advertises through Flow Control updates. In this case, the core promotes Completion and Posted TLPs ahead of these blocked Non-Posted TLPs. However, this can only occur if the Completion or Posted TLP has been loaded into the core by the User Application. By monitoring the `trn_tbuf_av` bus, the User Application can ensure there is at least one free buffer available for any Completion or Posted TLP. Promotion of Completion and Posted TLPs only occurs when Non-Posted TLPs are blocked; otherwise packets are sent on the link in the order they are received from the User Application.

## Receiving Inbound Packets

### Basic TLP Receive Operation

Table 3-10, page 34 defines the receive Transaction interface signals. The following sequence of events must occur on the receive Transaction interface for the Endpoint core to present a TLP to the User Application logic:

1. When the User Application is ready to receive data, it asserts `trn_rdst_rdy_n`.

2. When the core is ready to transfer data, the core asserts `trn_rsrc_rdy_n` with `trn_rsof_n` and presents the first complete TLP DWORD on `trn_rd[31:0]`.
3. The core then deasserts `trn_rsof_n`, asserts `trn_rsrc_rdy_n`, and presents TLP DWORDs on `trn_rd[31:0]` for subsequent clock cycles, for which the User Application logic asserts `trn_rdst_rdy_n`.
4. The core then asserts `trn_reof_n` and presents either the last DWORD on `trn_td[31:0]`.
5. If no further TLPs are available, at the next clock cycle, the core deasserts `trn_rsrc_rdy_n` to signal the end of valid transfers on `trn_rd[31:0]`.

Figure 6-13 illustrates a 3-DW TLP header without a data payload; an example is a 32-bit addressable Memory Read request.

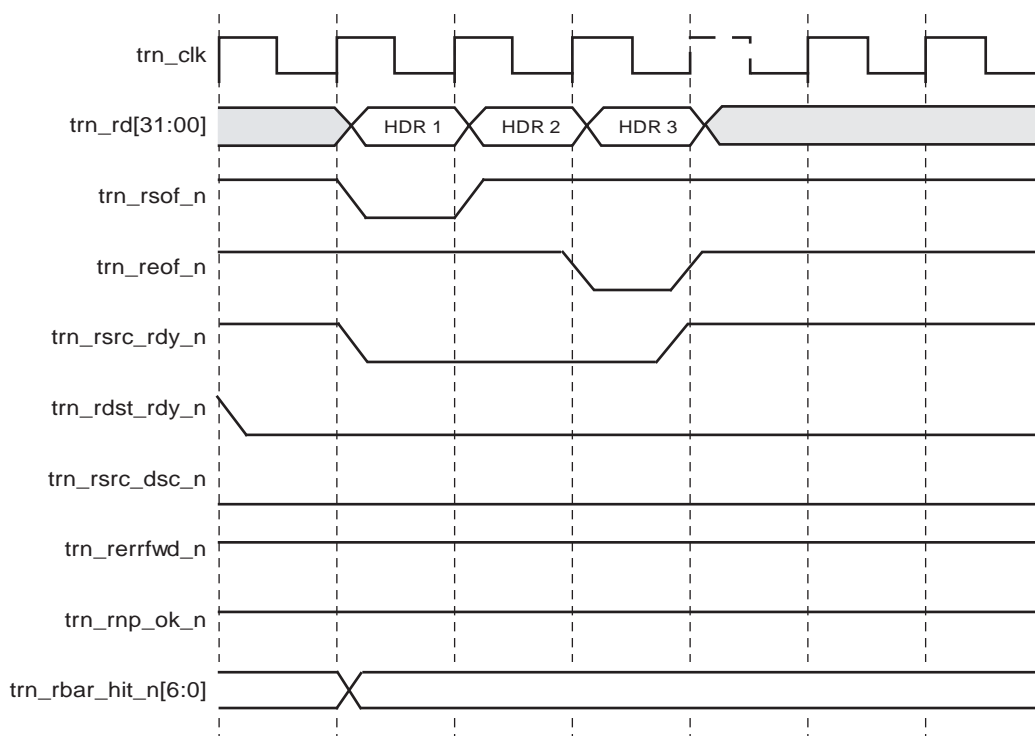


Figure 6-13: TLP 3-DW Header without Payload



Figure 6-14 illustrates a 4-DW TLP header without a data payload; an example is a 64-bit addressable Memory Read request.

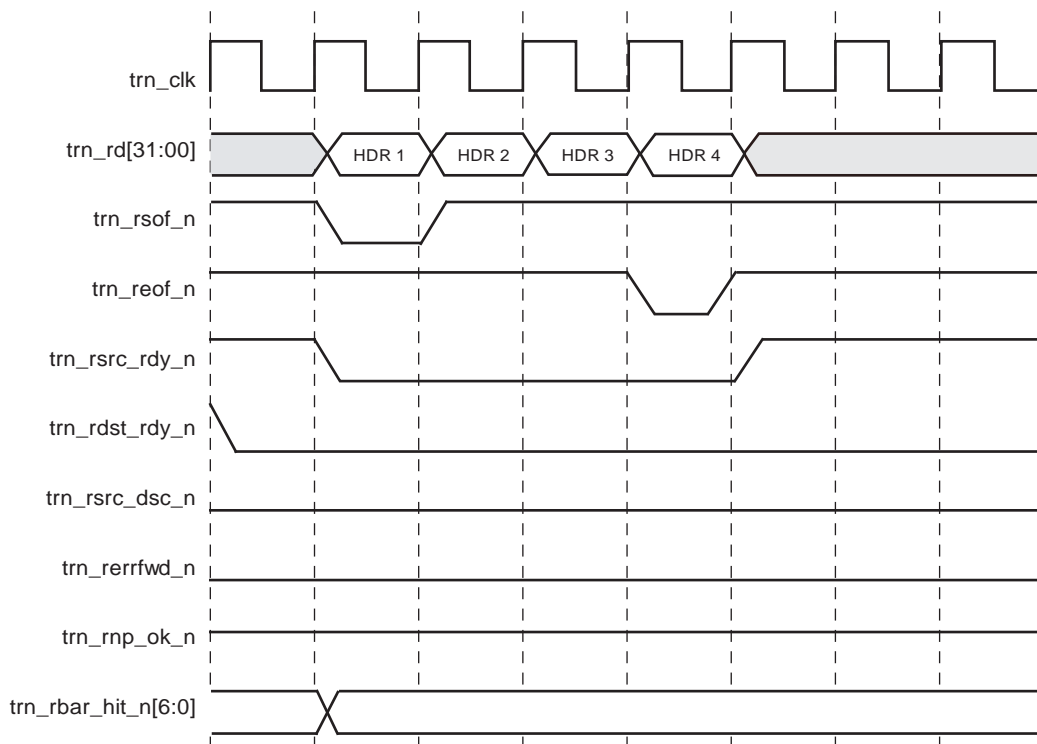


Figure 6-14: TLP 4-DW Header without Payload

Figure 6-15 illustrates a 3-DW TLP header with a data payload; an example is a 32-bit addressable Memory Write request.

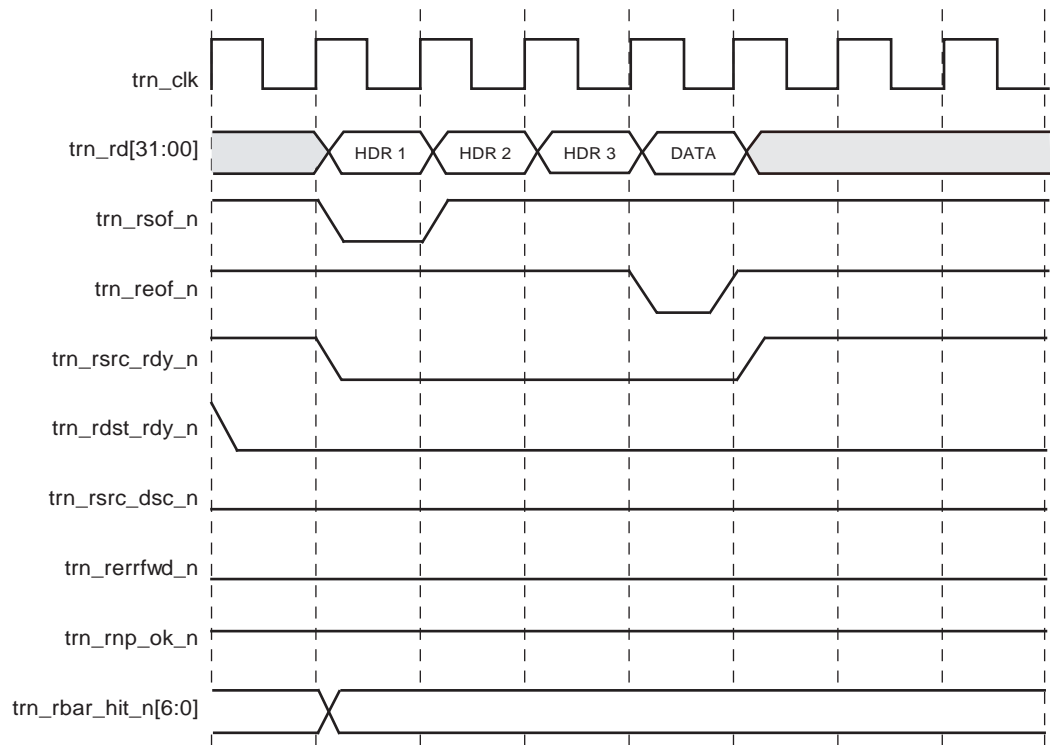


Figure 6-15: TLP 3-DW Header with Payload

Figure 6-16 illustrates a 4-DW TLP header with a data payload; an example is a 64-bit addressable Memory Write request.

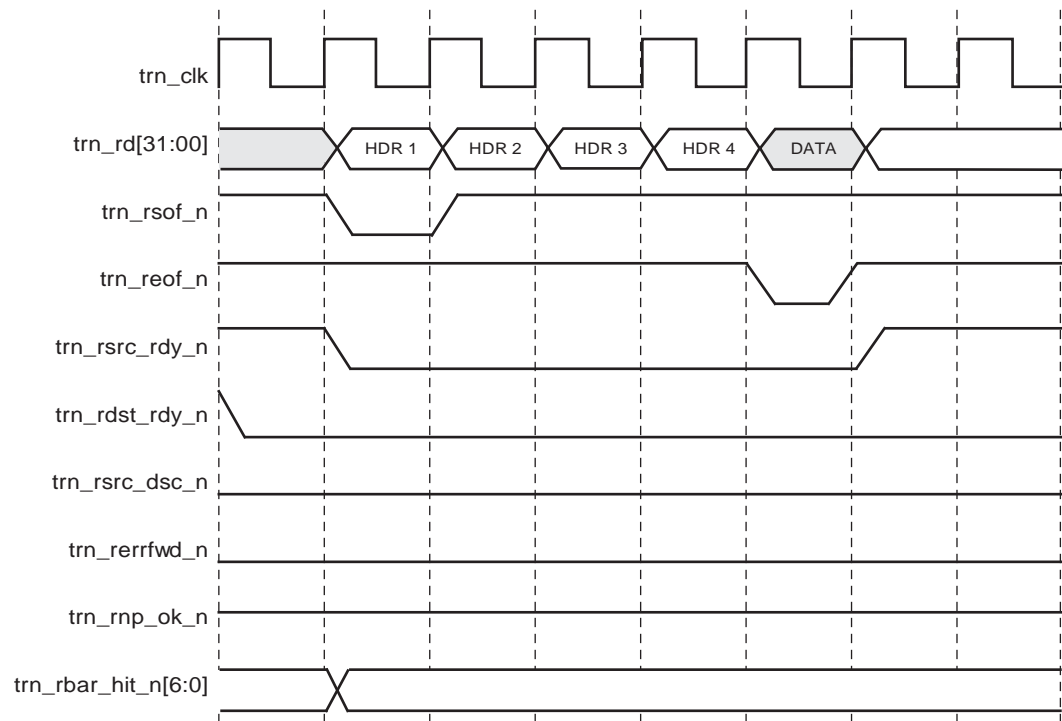


Figure 6-16: TLP 4-DW Header with Payload

## Throttling the Data Path on the Transaction Interface

The User Application can stall the transfer of data from the core at any time by deasserting `trn_rdst_rdy_n`. If the user deasserts `trn_rdst_rdy_n` while no transfer is in progress and if a TLP becomes available, the core asserts `trn_rsrc_rdy_n` and `trn_rsof_n` and presents the first TLP DWORD on `trn_rd[31:0]`. The core remains in this state until the user asserts `trn_rdst_rdy_n` to signal the acceptance of the data presented on `trn_rd[31:0]`. At that point, the core presents subsequent TLP DWORDs as long as `trn_rdst_rdy_n` remains asserted. If the user deasserts `trn_rdst_rdy_n` during the middle of a transfer, the core stalls the transfer of data until the user asserts `trn_rdst_rdy_n` again. There is no limit to the number of cycles the user can keep `trn_rdst_rdy_n` deasserted. The core will pause until the user is again ready to receive TLPs.

Figure 6-17 illustrates the core asserting `trn_rsrc_rdy_n` and `trn_rsof_n` along with presenting data on `trn_rd[31:0]`. The User Application logic inserts wait states by deasserting `trn_rdst_rdy_n`. The Endpoint core will not present the next TLP DWORD until it detects `trn_rdst_rdy_n` assertion. The User Application logic asserts or deasserts `trn_rdst_rdy_n` as required to balance receipt of new TLP transfers with the rate of TLP data processing inside the application logic.

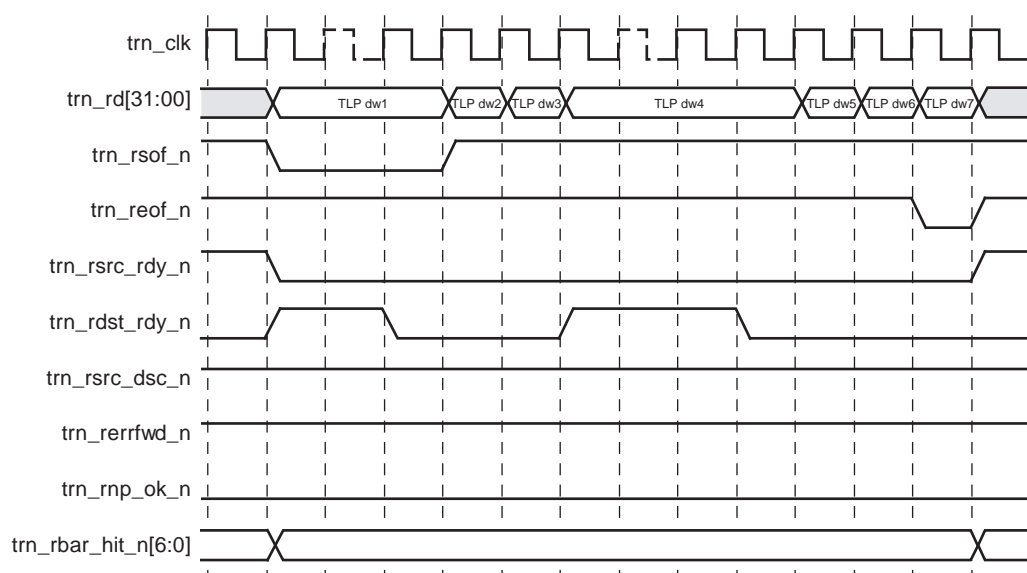


Figure 6-17: User Application Throttling Receive TLP

## Receiving Back-To-Back Transactions on the Transaction Interface

The User Application logic must be designed to handle presentation of back-to-back TLPs on the receive Transaction interface by the core. The core can assert `trn_rsof_n` for a new TLP at the clock cycle after `trn_reof_n` assertion for the previous TLP. Figure 6-18 illustrates back-to-back TLPs presented on the receive interface.

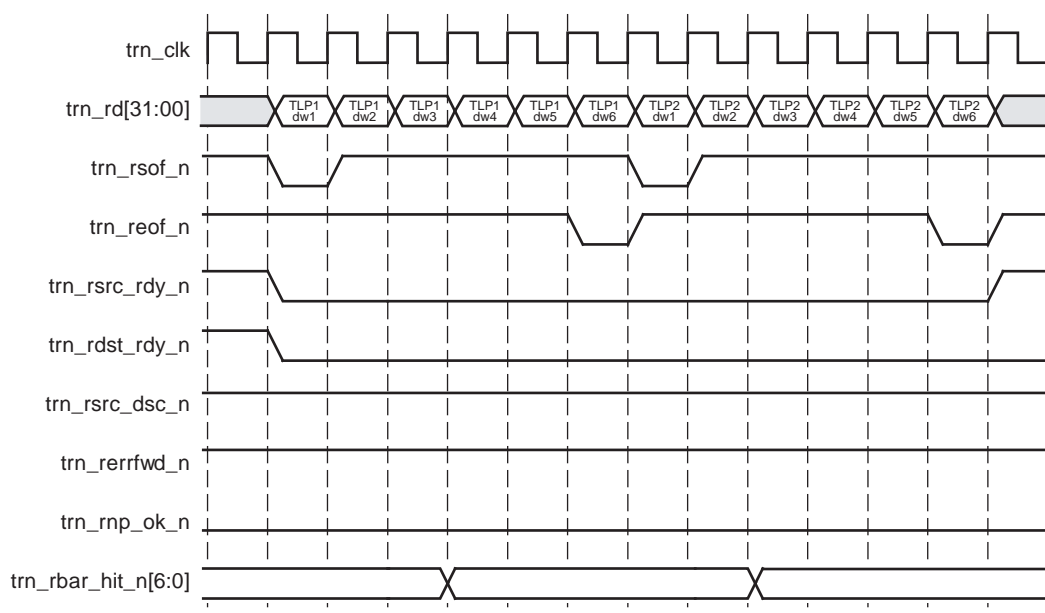


Figure 6-18: Receive Back-To-Back Transactions

If the User Application cannot accept back-to-back packets, it can stall the transfer of the TLP by deasserting `trn_rdst_rdy_n` as discussed in the previous section. Figure 6-19 shows an example of using `trn_rdst_rdy_n` to pause the acceptance of the second TLP.

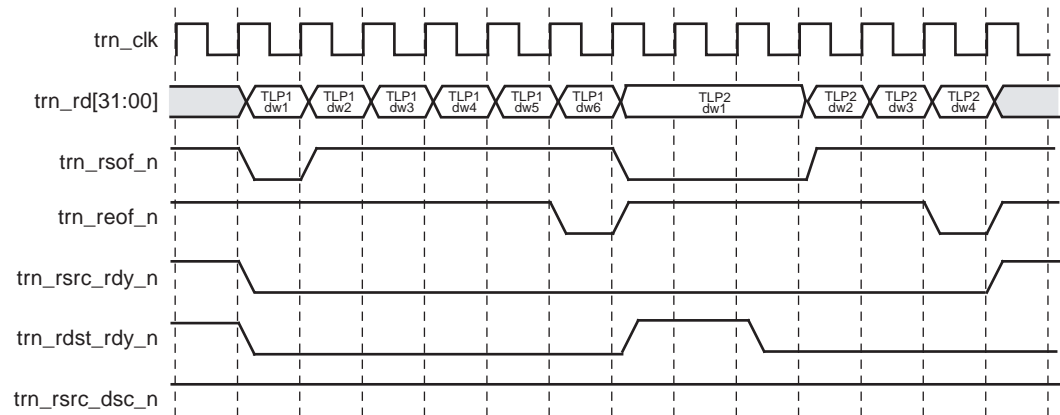


Figure 6-19: User Application Throttling of Back-to-Back TLPs

## Packet Re-ordering on Receive Transaction Interface

Transaction processing in the core receiver is fully compliant with the PCI transaction ordering rules. The transaction ordering rules allow for Posted and Completion TLPs to bypass Non-Posted TLPs. See section 2.4 of the *PCI Express Base Specification* for more information about the ordering rules.

The User Application can deassert the signal `trn_rnp_ok_n` if it is not ready to accept Non-Posted Transactions from the core, as shown in Figure 6-19 but can receive Posted and Completion Transactions. The User Application must deassert `trn_rnp_ok_n` at least one clock cycle before `trn_eof_n` of the last non-posted packet the user can accept. While `trn_rnp_ok_n` is deasserted, received Posted and Completion Transactions pass Non-Posted Transactions. After the User Application is ready to accept Non-Posted Transactions, it must reassert `trn_rnp_ok_n`. Previously bypassed Non-Posted Transactions are presented to the User Application before other received TLPs.

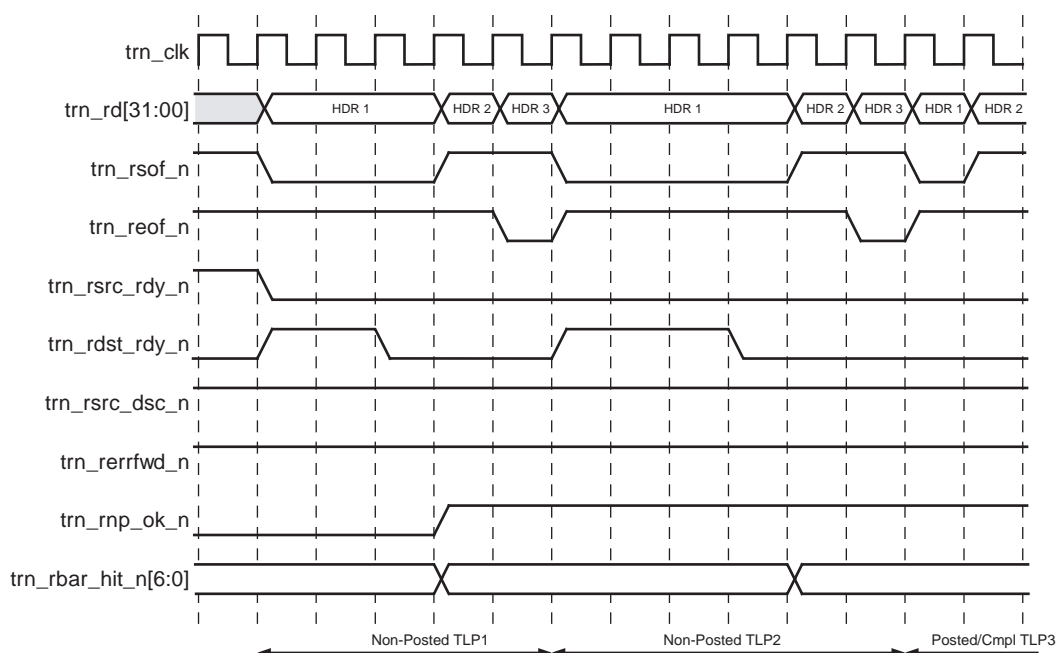


Figure 6-19: Packet Re-ordering on Receive Transaction Interface

To ensure that there is no overflow of Non-Posted storage space available in the user's logic, the following algorithm must be used:

```

For every clock cycle, do {
  if (Valid transaction Start-Of-Frame accepted by User Application) {
    Extract TLP_type from the 1st TLP DW
    if (TLP_type == Non-Posted) {
      if (Non-Posted_Buffers_Available <= 2)
        Deassert trn_rnp_ok_n on the following clock cycle.
      else if (Other user policies stall Non-Posted transactions)
        Deassert trn_rnp_ok_n on the following clock cycle.
      else
        Assert trn_rnp_ok_n on the following clock cycle.
        Decrement Non-Posted_Buffers_Available in User Application
    }
  }
}

```

## Packet Data Poisoning and TLP Digest on Transaction Interface

To simplify logic within the User Application, the Endpoint core performs automatic pre-processing based on values of TLP Digest (TD) and Data Poisoning (EP) header bit fields on the received TLP.

All received TLPs with the Data Poisoning bit in the header set (EP=1) are presented to the user. The Endpoint core asserts the `trn_rerrfwd_n` signal for the duration of each poisoned TLP, as illustrated in Figure 6-20.

If the TLP Digest bit field in the TLP header is set (TD=1), the TLP contains an End-to-End CRC (ECRC). The Endpoint core performs the following operations based on how the user configured the core during core generation:

- If the Trim TLP Digest option is on, the Endpoint core removes and discards the ECRC field from the received TLP and clears the TLP Digest bit in the TLP header.
- If the Trim TLP Digest option is off, the Endpoint core does not remove the ECRC field from the received TLP and presents the entire TLP including TLP Digest to the User Application receiver interface.

See [Chapter 5, “Generating and Customizing the Core,”](#) for more information about how to enable the Trim TLP Digest option during core generation.

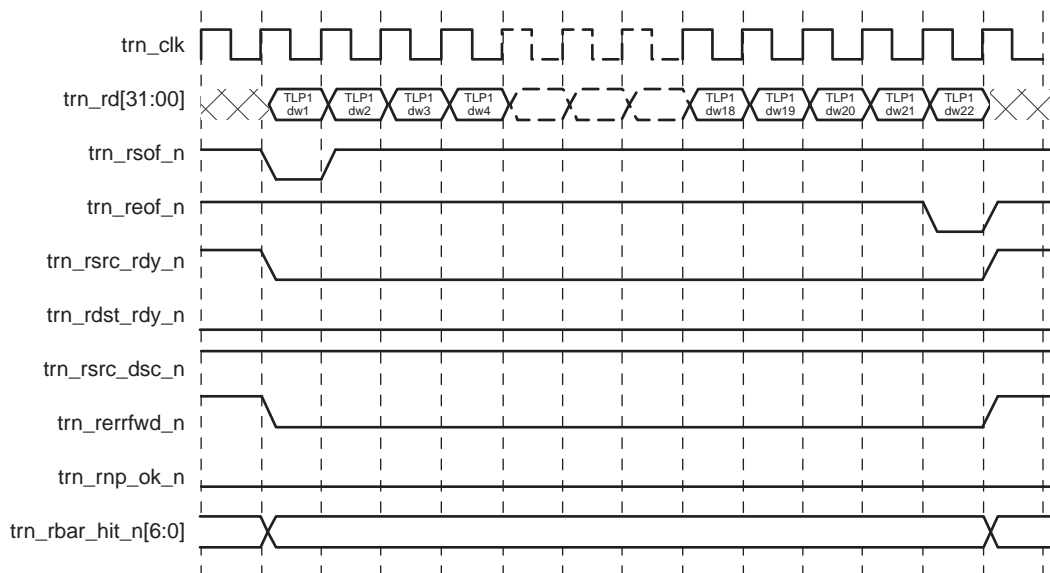


Figure 6-20: Receive Transaction Data Poisoning

## Packet Base Address Register Hit on Receive Transaction Interface

The Endpoint core decodes incoming Memory and IO TLP request addresses to determine which Base Address Register (BAR) in the Endpoint's Type0 configuration space is being targeted, and indicates the decoded base address on `trn_rbar_hit_n[6:0]`. For each received Memory or IO TLP, a minimum of one and a maximum of two (adjacent) bit(s) are set to 0. If the received TLP targets a 32-bit Memory or IO BAR, only one bit is asserted. If the received TLP targets a 64-bit Memory BAR, two adjacent bits are asserted. If the core receives a TLP that is not decoded by one of the BARs (that is, a misdirected TLP), then the core will drop it without presenting it to the user and it will automatically generate an Unsupported Request message. Note that even if the core is configured for a 64-bit BAR, the system may not always allocate a 64-bit address, in which case only one `trn_rbar_hit_n[6:0]` signal will be asserted.

[Table 6-2](#) illustrates mapping between `trn_rbar_hit_n[6:0]` and the BARs, and the corresponding byte offsets in the core Type0 configuration header.

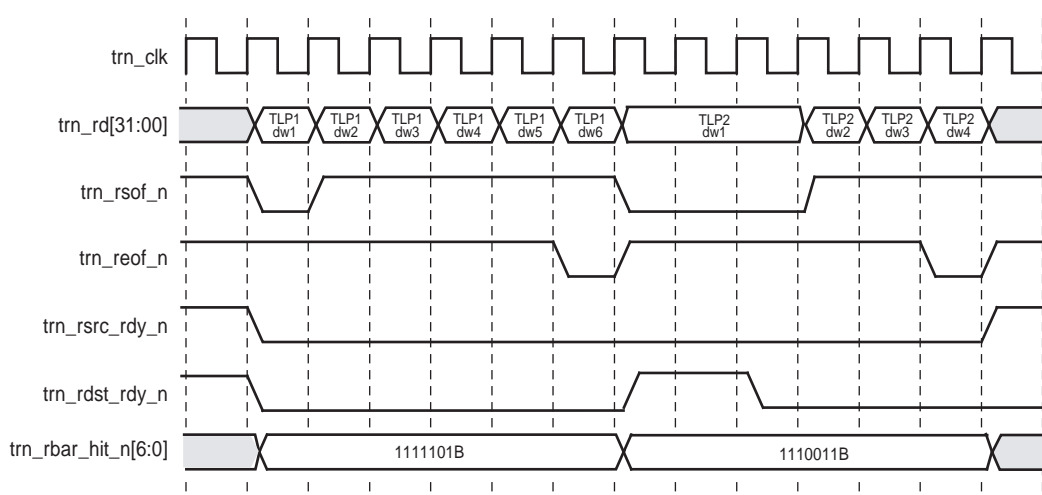
Table 6-2: `trn_rbar_hit_n` to Base Address Register Mapping

<code>trn_rbar_hit_n[x]</code>	BAR	Byte Offset
0	0	10h
1	1	14h
2	2	18h

Table 6-2: `trn_rbar_hit_n` to Base Address Register Mapping (Cont'd)

<code>trn_rbar_hit_n[x]</code>	BAR	Byte Offset
3	3	1Ch
4	4	20h
5	5	24h
6	Expansion ROM BAR	30h

For a Memory or IO TLP Transaction on the receive interface, `trn_rbar_hit_n[6:0]` is valid for the entire TLP, starting with the assertion of `trn_rsof_n`, as shown in Figure 6-21. When receiving non-Memory and non-IO transactions, `trn_rbar_hit_n[6:0]` is undefined.

Figure 6-21: BAR Target Determination using `trn_rbar_hit`

The signal `trn_rbar_hit_n[6:0]` enables received Memory/IO Transactions to be directed to the appropriate destination Memory/IO apertures in the User Application. By utilizing `trn_rbar_hit_n[6:0]`, application logic may inspect only the lower order Memory/IO address bits within the address aperture to simplify decoding logic.

## Packet Transfer Discontinue on Transaction Interface

The Endpoint PIPE for PCIe core asserts `trn_rsrc_dsc_n` if communication with the link partner is lost, which results in the termination of an *in-progress* TLP. The loss of communication with the link partner is signaled by deassertion of `trn_lnk_up_n`. When `trn_lnk_up_n` is deasserted, it effectively acts as a Hot Reset to the entire core. For this reason, all TLPs stored inside the core or being presented to the receive interface are irrecoverably lost. Figure 6-22 illustrates packet transfer discontinue scenario.



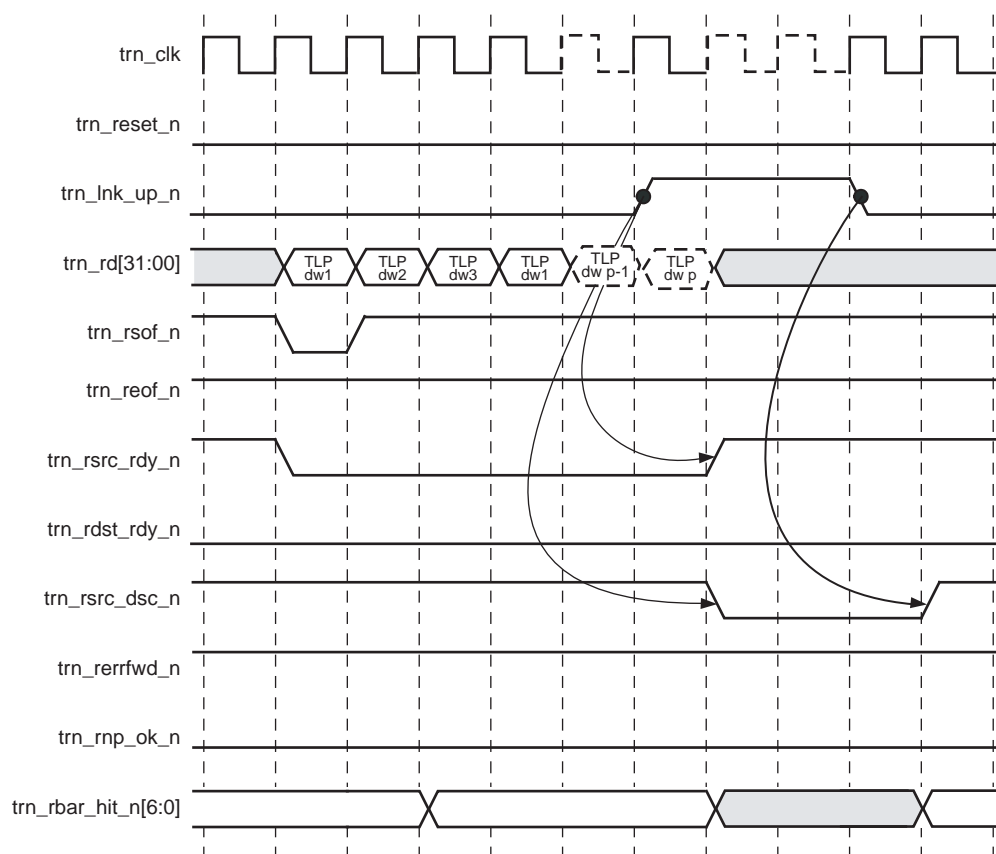


Figure 6-22: Receive Transaction Discontinue

## Receiver Flow Control Credits Available

The Endpoint PIPE for PCIe core provides the User Application information about the state of the receiver buffer pool queues. This information represents the current space available for the Posted, Non-Posted, and Completion queues.

One Header Credit is equal to either a 3 or 4 DWORD TLP Header and one Data Credit is equal to 16 bytes of payload data. [Table 6-3](#) provides values on credits available immediately after `trn_lnk_up_n` assertion but before the reception of any TLP. If space available for any of the above categories is exhausted, the corresponding credit available signals will indicate a value of zero. Credits available return to initial values after the receiver has drained all TLPs.

**Table 6-3: Transaction Receiver Credits Available Initial Values**

Credit Category	Signal Name	Maximum Credits (Hex)	Max Size (Decimal)
Non-Posted Header	<code>trn_rfc_nph_av[7:0]</code>	0C	12Headers
Non-Posted Data	<code>trn_rfc_npd_av[11:0]</code>	00C	192Bytes
Posted Header	<code>trn_rfc_ph_av[7:0]</code>	20	32Headers
Posted Data	<code>trn_rfc_pd_av[11:0]</code>	120	4608 Bytes
Completion Header	<code>trn_rfc_cplh_av[7:0]</code>	24	33-36 Headers <sup>a</sup>
Completion Data	<code>trn_rfc_cpld_av[11:0]</code>	090	2176-2304 Bytes <sup>a</sup>

a. Variable depending on the Trim TLP Digest option.

The User Application can use the `trn_rfc_ph_av[7:0]`, `trn_rfc_pd_av[11:0]`, `trn_rfc_nph_av[7:0]`, and `trn_rfc_npd_av[11:0]` signals to efficiently utilize and manage receiver buffer space available in the core and the core application.

Endpoint cores have a unique requirement where the User Application must use advanced methods to prevent buffer overflows while requesting Non-Posted Read Requests from an upstream component. According to the specification, an Endpoint is required to advertise infinite storage credits for Completion Transactions in its receivers. This means that endpoints must internally manage Memory Read Requests transmitted upstream and not overflow the receiver when the corresponding Completions are received. The User Application must use Completion credit information presented to modulate the rate and size of Memory Read requests, to stay within the instantaneous Completion space available in the Endpoint receiver. For additional information, see [Appendix A, "Tracking Receive-buffer Space for Inbound Completions."](#)

## Accessing Configuration Space Registers

### Registers Mapped Directly onto the Configuration Interface

The Endpoint PIPE for PCIe core provides direct access to select command and status registers in its Configuration Space. Values in these registers are modified by Configuration Writes received from the Root Complex and cannot be modified by the User

Application. Table 6-4 defines the command and status registers mapped to the configuration port.

**Table 6-4: Command and Status Registers Mapped to the Configuration Port**

Port Name	Direction	Description
cfg_bus_number[7:0]	Output	<b>Bus Number:</b> Default value after reset is 00h. Refreshed whenever a Type 0 Configuration Write packet is received.
cfg_device_number[4:0]	Output	<b>Device Number:</b> Default value after reset is 00000b. Refreshed whenever a Type 0 Configuration Write packet is received.
cfg_function_number[2:0]	Output	<b>Function Number:</b> Function number of the core, hard wired to 000b.
cfg_status[15:0]	Output	<b>Status Register:</b> Status register from the Configuration Space Header.
cfg_command[15:0]	Output	<b>Command Register:</b> Command register from the Configuration Space Header.
cfg_dstatus[15:0]	Output	<b>Device Status Register:</b> Device status register from the PCI Express Capability Structure.
cfg_dcommand[15:0]	Output	<b>Device Command Register:</b> Device control register from the PCI Express Capability Structure.
cfg_lstatus[15:0]	Output	<b>Link Status Register:</b> Link status register from the PCI Express Capability Structure.
cfg_lcommand[15:0]	Output	<b>Link Command Register:</b> Link control register from the PCI Express Capability Structure.

## Device Control and Status Register Definitions

### cfg\_bus\_number[7:0], cfg\_device\_number[4:0], cfg\_function\_number[2:0]

Together, these three values comprise the core ID, which the core captures from the corresponding fields of inbound Type 0 Configuration accesses. The User Application is responsible for using this core ID as the Requestor ID on any requests it originates, and using it as the Completer ID on any Completion response it sends. Note that the Endpoint PIPE for PCIe core supports only one function; for this reason, the function number is hard wired to 000b.

### cfg\_status[15:0]

This bus allows the User Application to read the Status register in the PCI Configuration Space Header. Table 6-5 defines these bits. See the *PCI Express Base Specification* for detailed information.

**Table 6-5: Bit Mapping on Header Status Register**

Bit	Name
cfg_status[15]	Detected Parity Error
cfg_status[14]	Signaled System Error

Table 6-5: Bit Mapping on Header Status Register (Cont'd)

Bit	Name
cfg_status[13]	Received Master Abort
cfg_status[12]	Received Target Abort
cfg_status[11]	Signaled Target Abort
cfg_status[10:9]	DEVSEL Timing (hard-wired to 00b)
cfg_status[8]	Master Data Parity Error
cfg_status[7]	Fast Back-to-Back Transactions Capable (hard-wired to 0)
cfg_status[6]	Reserved
cfg_status[5]	66 MHz Capable (hard-wired to 0)
cfg_status[4]	Capabilities List Present (hard-wired to 1)
cfg_status[3]	Interrupt Status
cfg_status[2:0]	Reserved
cfg_command[1]	Memory Address Space Decoder Enable
cfg_command[0]	IO Address Space Decoder Enable

**cfg\_command[15:0]**

This bus reflects the value stored in the Command register in the PCI Configuration Space Header. Table 6-6 provides the definitions for each bit in this bus. See the *PCI Express Base Specification* for detailed information.

Table 6-6: Bit Mapping on Header Command Register

Bit	Name
cfg_command[15:11]	Reserved
cfg_command[10]	Interrupt Disable
cfg_command[9]	Fast Back-to-Back Transactions Enable (hardwired to 0)
cfg_command[8]	SERR Enable
cfg_command[7]	IDSEL Stepping/Wait Cycle Control (hardwired to 0)
cfg_command[6]	Parity Error Enable
cfg_command[5]	VGA Palette Snoop (hardwired to 0)
cfg_command[4]	Memory Write and Invalidate (hardwired to 0)
cfg_command[3]	Special Cycle Enable (hardwired to 0)
cfg_command[2]	Bus Master Enable
cfg_command[1]	Memory Address Space Decoder Enable
cfg_command[0]	IO Address Space Decoder Enable

The User Application must monitor the Bus Master Enable bit (`cfg_command[2]`) and refrain from transmitting requests while this bit is not set. This requirement applies only to requests; completions can be transmitted regardless of this bit.

### cfg\_dstatus[15:0]

This bus reflects the value stored in the Device Status register of the PCI Express Extended Capabilities Structure. Table 6-7 defines each bit in the `cfg_dstatus` bus. See the *PCI Express Base Specification* for detailed information.

**Table 6-7: Bit Mapping of PCI Express Device Status Register**

Bit	Name
cfg_dstatus[15:6]	Reserved
cfg_dstatus[5]	Transaction Pending
cfg_dstatus[4]	AUX Power Detected
cfg_dstatus[3]	Unsupported Request Detected
cfg_dstatus[2]	Fatal Error Detected
cfg_dstatus[1]	Non-Fatal Error Detected
cfg_dstatus[0]	Correctable Error Detected

### cfg\_dcommand[15:0]

This bus reflects the value stored in the Device Control register of the PCI Express Extended Capabilities Structure. Table 6-8 defines each bit in the `cfg_dcommand` bus. See the *PCI Express Base Specification* for detailed information.

**Table 6-8: Bit Mapping of PCI Express Device Control Register**

Bit	Name
cfg_dcommand[15]	Reserved
cfg_dcommand[14:12]	Max_Read_Request_Size
cfg_dcommand[11]	Enable No Snoop
cfg_dcommand[10]	Auxiliary Power PM Enable
cfg_dcommand[9]	Phantom Functions Enable
cfg_dcommand[8]	Extended Tag Field Enable
cfg_dcommand[7:5]	Max_Payload_Size
cfg_dcommand[4]	Enable Relaxed Ordering
cfg_dcommand[3]	Unsupported Request Reporting Enable
cfg_dcommand[2]	Fatal Error Reporting Enable
cfg_dcommand[1]	Non-Fatal Error Reporting Enable
cfg_dcommand[0]	Correctable Error Reporting Enable

### cfg\_lstatus[15:0]

This bus reflects the value stored in the Link Status register in the PCI Express Extended Capabilities Structure. [Table 6-9](#) defines each bit in the `cfg_lstatus` bus. See the *PCI Express Base Specification* for details.

**Table 6-9: Bit Mapping of PCI Express Link Status Register**

Bit	Name
cfg_lstatus[15:13]	Reserved
cfg_lstatus[12]	Slot Clock Configuration
cfg_lstatus[11]	Reserved
cfg_lstatus[10]	Reserved
cfg_lstatus[9:4]	Negotiated Link Width
cfg_lstatus[3:0]	Link Speed

### cfg\_lcommand[15:0]

This bus reflects the value stored in the Link Control register of the PCI Express Extended Capabilities Structure. [Table 6-10](#) provides the definition of each bit in `cfg_lcommand` bus. See the *PCI Express Base Specification* for more details.

**Table 6-10: Bit Mapping on the Link Control Register**

Bit	Name
cfg_lcommand[15:8]	Reserved
cfg_lcommand [7]	Extended Synch
cfg_lcommand [6]	Common Clock Configuration
cfg_lcommand [5]	Retrain Link (Reserved for an endpoint device)
cfg_lcommand [4]	Link Disable
cfg_lcommand [3]	Read Completion Boundary
cfg_lcommand[2]	Reserved
cfg_lcommand [1:0]	Active State Link PM Control

## Accessing Additional Registers Through the Configuration Port

Configuration registers that are not directly mapped to the user interface can be accessed by configuration-space address using the ports shown in [Table 3-11, page 36](#).

The User Application must supply the read address as a DWORD address, not a byte address. To calculate the DWORD address for a register, divide the byte address by four. For example:

- The DWORD address of the Command/Status Register in the PCI Configuration Space Header is 01h. (The byte address is 04h.)
- The DWORD address for BAR0 is 04h. (The byte address is 10h.)

To read any register in this address space, the User Application drives the register DWORD address onto `cfg_dwaddr[9:0]`. The Endpoint core drives the content of the addressed register onto `cfg_do[31:0]`. The value on `cfg_do[31:0]` is qualified by signal assertion on `cfg_rd_wr_done_n`. [Figure 6-23](#) illustrates an example with two consecutive reads from the Configuration Space.

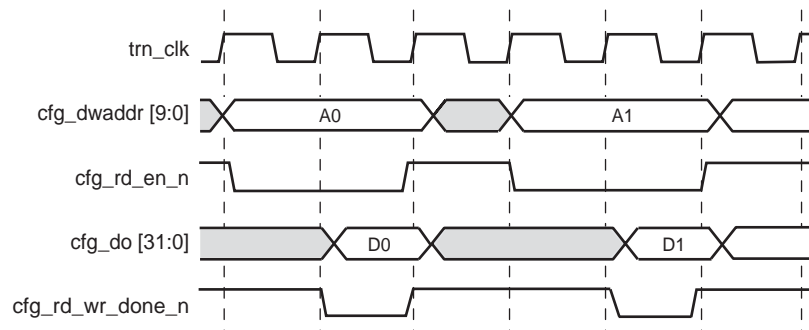


Figure 6-23: Example Configuration Space Access

Note that writing to the Configuration Space using `cfg_di[31:0]` and `cfg_wr_en_n` is not supported in this release. The Endpoint core ignores any writes to the Configuration Space by the User Application.

## Additional Packet Handling Requirements

The User Application must manage the following mechanisms to ensure protocol compliance, because the core does not manage them automatically.

### Generation of Completions

The Endpoint core does not generate Completions for Memory Reads or I/O requests made by a remote device. The user is expected to service these completions according to the rules specified in the *PCI Express Base Specification*.

### Tracking Non-Posted Requests and Inbound Completions

The Endpoint core does not track transmitted I/O requests or Memory Reads that have yet to be serviced with inbound Completions. The User Application is required to keep track of such requests via the Tag ID or other information.

Keep in mind that one Memory Read request can be answered by several Completion packets. The User Application must accept all inbound Completions associated with the original Memory Read until all requested data has been received.

The *PCI Express Base Specification* requires that an endpoint advertise infinite Completion Flow Control credits as a receiver; the endpoint can only transmit Memory Reads and I/O requests if it has enough space to receive subsequent Completions.

The Endpoint core does not keep track of receive-buffer space for Completion. Rather, it sets aside a fixed amount of buffer space for inbound Completions. The User Application must keep track of this buffer space to know if it can transmit requests requiring a Completion response.

See [Appendix A, “Tracking Receive-buffer Space for Inbound Completions”](#) for information about implementing this mechanism.

## Reporting User Error Conditions

The User Application must report errors that occur during Completion handling using dedicated error signals on the Endpoint core interface, and must observe the Device Power State before signaling an error to the core. If the User Application detects an error (for example, a Completion Timeout) while the device has been programmed to a non-D0 state, the User Application is responsible to signal the error after the device is programmed back to the D0 state.

After the User Application signals an error, the core reports the error on the PCI Express Link and also sets the appropriate status bit(s) in the Configuration Space. Because status bits must be set in the appropriate Configuration Space register, the User Application cannot generate error reporting packets on the transmit interface. The type of error-reporting packets transmitted depends on whether or not the error resulted from a Posted or Non-Posted Request. All user-reported errors cause Message packets to be sent to the Root Complex, unless the error is regarded as an Advisory Non-Fatal Error. For more information about Advisory Non-Fatal Errors, see Chapter 6 of the *PCI Express Base Specification*. Errors on Non-Posted Requests can result in either Messages to the Root Complex or Completion packets with non-Successful status sent to the original Requester.

### Error Types

The User Application triggers six types of errors using the signals defined in [Table 3-13, page 39](#).

- End-to-end CRC ECRC Error
- Unsupported Request Error
- Completion Timeout Error
- Unexpected Completion Error
- Completer Abort Error
- Correctable Error

Multiple errors can be detected in the same received packet; for example, the same packet can be an Unsupported Request and have an ECRC error. If this happens, only one error should be reported. Because all user-reported errors have the same severity, the User Application design can determine which error to report. The `cfg_err_posted_n` signal, combined with the appropriate error reporting signal, indicates what type of error-



reporting packets are transmitted. The user can signal only one error per clock cycle. See [Figure 6-24](#) and [Figure 6-25](#), and [Table 6-11](#) and [Table 6-12](#).

The `cfg_err_posted_n` signal, combined with the appropriate error reporting signal, indicates what type of error-reporting packets are transmitted. See [Table 6-11](#) and [Figure 6-24](#).

**Table 6-11: User-indicated Error Signaling**

Reported Error <sup>a</sup>	cfg_err_posted_n	Action
None	Don't care	No Action Taken
cfg_err_ur_n	0 or 1	0: If enabled, a Non-Fatal Error Message will be sent 1: A Completion with a non-Successful status will be sent
cfg_err_cpl_abort_n	0 or 1	0: If enabled, a Non-Fatal Error Message will be sent 1: A Completion with a non-Successful status will be sent
cfg_err_cpl_timeout_n	Don't care	If enabled, a Non-Fatal Error Message will be sent
cfg_err_ecrc_n	Don't care	If enabled, a Non-Fatal Error Message will be sent
cfg_err_cor_n	Don't care	If enabled, a Correctable Error Message will be sent
cfg_err_cpl_unexpected_n	Don't care	Regarded as an Advisory Non-Fatal Error (ANFE); no action taken

a. Indicates whether one of the other five `cfg_err` signals is asserted.

**Table 6-12: Possible Error Conditions for TLPs Received by the User Application**

Received TLP Type	Possible Error Condition						Error Qualifying Signal Status	
		Unsupported Request (cfg_err_ur_n)	Completion Abort (cfg_err_cpl_abort_n)	Correctable Error (cfg_err_cor_n)	ECRC Error (cfg_err_ecrc_n)	Unexpected Completion (cfg_err_cpl_unexpect_n)	Value to Drive on (cfg_err_posted_n)	Drive Data on (cfg_err_tlp_cpl_header[47:0])
Memory Write		4	5	N/A	4	5	0	No
Memory Read		4	4	N/A	4	5	1	Yes
I/O		4	4	N/A	4	5	1	Yes
Configuration TLP (Extended Space)		4	4	N/A	4	5	1	Yes
Completion		5	5	N/A	4	4	0	No

Whenever an error is detected in a Non-Posted Request, the User Application deasserts `cfg_err_posted_n` and provides header information on `cfg_err_tlp_cpl_header[47:0]` during the same clock cycle the error is reported, as illustrated in [Figure 6-24](#). The additional header information is necessary to construct the required Completion with non-successful status. Additional information about when to assert or deassert `cfg_err_posted_n` is provided in the following sections.

If an error is detected on a Posted Request, the User Application instead asserts `cfg_err_posted_n`, but otherwise follows the same signaling protocol. This will result in a Non-Fatal Message to be sent, if enabled.

The Endpoint core's ability to generate error messages can be disabled by the Root Complex issuing a configuration write to the Endpoint core's Device Control register and the PCI Command register setting the appropriate bits to 0. For more information about these registers, see Chapter 7 of the *PCI Express Base Specification*. However, error-reporting status bits are always set in the Configuration Space whether or not their Messages are disabled.

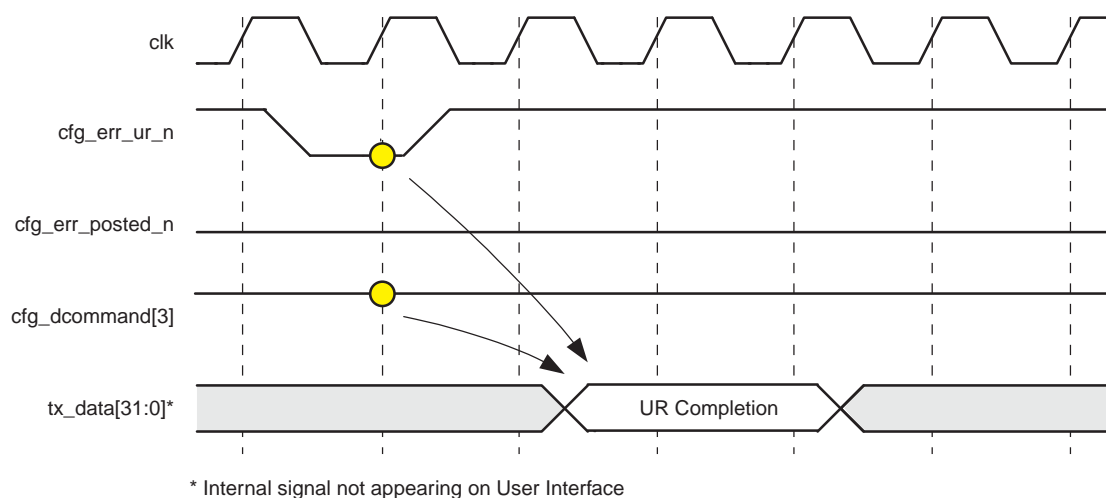


Figure 6-24: Signaling Unsupported Request for Non-Posted TLP

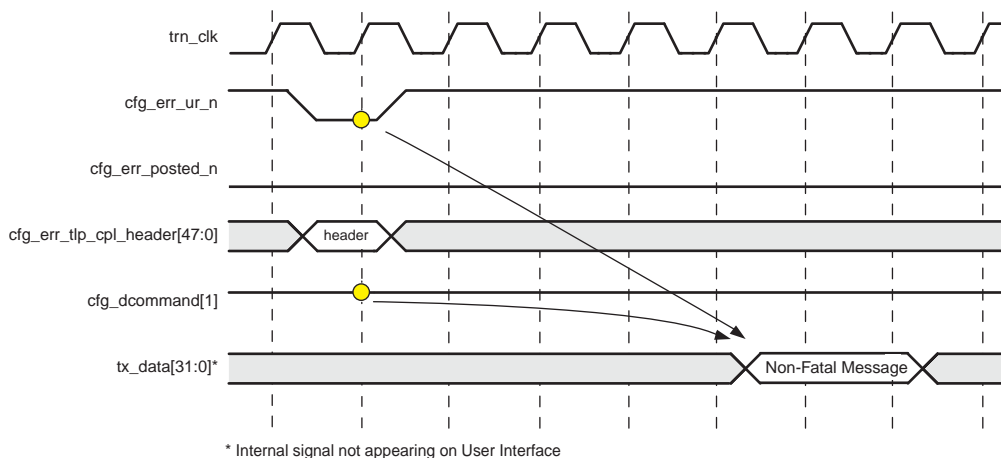
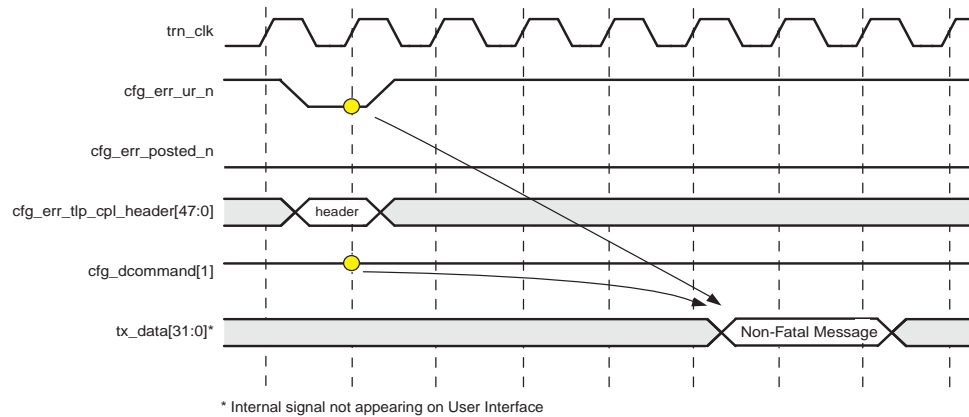


Figure 6-25: Signaling Unsupported Request for Posted TLP



**Figure 6-27: Example of UR Completion; No Error Message Sent**

### Completion Timeouts

The Endpoint PIPE for PCIe core does not implement Completion timers; for this reason, the User Application must track how long its pending Non-Posted Requests have each been waiting for a Completion and trigger timeouts on them accordingly. The core has no method of knowing when such a timeout has occurred, and for this reason does not filter out inbound Completions for expired requests.

If a request times out, the User Application must assert `cfg_err_cpl_timeout_n`, which causes an error message to be sent to the Root Complex.

If a Completion is later received after a request times out, the User Application must treat it as an Unexpected Completion.

### Unexpected Completions

The Endpoint PIPE for PCIe core automatically reports Unexpected Completion in response to inbound Completions whose Requestor ID is different than the Endpoint ID programmed in the Configuration Space. These completions are not passed to the User Application. The current version of the Endpoint PIPE for PCIe core regards an Unexpected Completion to be an Advisory Non-Fatal Error (ANFE), and no message is sent.

### Completer Abort

If the User Application is unable to transmit a normal Completion in response to a Non-Posted Request it receives, it must signal `cfg_err_cpl_abort_n`. The `cfg_err_posted_n` signal can also be set to 1 simultaneously to indicate Non-Posted and the appropriate request information placed on `cfg_err_tlp_cpl_header[47:0]`. This sends a Completion with non-Successful status to the original Requester, but does not send an Error Message. If the `cfg_err_posted_n` signal is set to 0 (to indicate a Posted transaction), no Completion is sent, but a Non-Fatal Error Message will be sent (if enabled).

### Unsupported Request

If the User Application receives an inbound Request it does not support or recognize, it must assert `cfg_err_ur_n` to signal an Unsupported Request. The `cfg_err_posted_n` signal must also be asserted or deasserted depending on whether the packet in question is a Posted or Non-Posted Request. If the packet is Posted, a Non-Fatal Error Message will be

sent out (if enabled); if the packet is Non-Posted, a Completion with a non-Successful status is sent to the original Requester.

The Unsupported Request condition can occur for several reasons, including the following:

- An inbound Memory Write packet violates the User Application's programming model, for example, if the User Application has been allotted a 4 kB address space but only uses 3 kB, and the inbound packet addresses the unused portion. (Note: If this occurs on a Non-Posted Request, the User Application should use `cfg_err_cpl_abort_n` to flag the error.)
- An inbound packet uses a packet Type not supported by the User Application, for example, an I/O request to a memory-only device.

### ECRC Error

The Endpoint PIPE for PCIe core does not check the ECRC field for validity. If the User Application chooses to check this field, and finds the CRC is in error, it can assert `cfg_err_ecrc_n`, causing a Non-Fatal Error Message to be sent.

## Power Management

The Endpoint PIPE for PCIe core supports the following power management modes:

- Active State Power Management (ASPM)
- Programmed Power Management (PPM)

Implementing these power management functions as part of the PCI Express design enables the PCI Express hierarchy to seamlessly exchange power-management messages to save system power. All power management message identification functions are implemented. The sections below describe the user logic definition to support the above modes of power management.

For additional information on ASPM and PPM implementation, see the *PCI Express Base Specification*.

### Power Management Support

The Endpoint PIPE for PCIe core fully supports power management modes.

#### Active State Power Management

The Active State Power Management (ASPM) functionality is autonomous and transparent from a user-logic function perspective. The Endpoint PIPE for PCIe core supports the conditions required for ASPM.

#### Programmed Power Management

To achieve considerable power savings on the PCI Express hierarchy tree, the Endpoint PIPE for PCIe core supports the following link states of Programmed Power Management (PPM):

- L0: Active State (data exchange state)
- L1: Higher Latency, lower power standby state

- L3: Link Off State

All PPM messages are always initiated by an upstream link partner. Programming the Endpoint PIPE for PCIe core to a non-D0 state, results in PPM messages being exchanged with the upstream link-partner. The PCI Express Link transitions to a lower power state after completing a successful exchange.

### PPM L0 State

The PPM L0 state represents *normal* operation and is transparent to the user logic. The Endpoint PIPE for PCIe core reaches the L0 (active state) after a successful initialization and training of the PCI Express Link as per the protocol.

### PPM L1 State

The following steps outline the transition of the Endpoint PIPE for PCIe core to the PPM L1 state.

1. The transition to a lower power PPM L1 state is always initiated by an upstream device, by programming the PCI Express device power state to D1/D2/D3-hot.
2. The core then throttles/stalls the user logic from initiating any new transactions on the user interface by deasserting `trn_tdst_rdy_n`. Any pending transactions on the user interface are however accepted fully and can be completed later.  
**Note:** If user-implemented extended Configuration Space is enabled (see [“Advanced User Configuration Space Settings,” page 65](#)) the core will not deassert `trn_tdst_rdy_n` during a non-D0 power state. In this case it is up to the user application to ensure only legal TLPs are transmitted during low-power states.
3. The core exchanges appropriate power management messages with its link partner to successfully transition the link to a lower power PPM L1 state. This action is transparent to the user logic.
4. All user transactions are stalled for the duration of time when the device power state is non-D0.
5. The device power state is communicated to the user logic through the user configuration port interface. The user logic is responsible for performing a successful read operation to identify the device power state.
6. The user logic, after identifying the device power state as D1, can initiate a request through the `cfg_pm_wake_n` to the upstream link partner to transition the link to a higher power state L0.

### PPM L3 State

The following steps outline the transition of the Endpoint PIPE for PCIe core to the PPM L3 state.

1. The Endpoint PIPE for PCIe core moves the link to the PPM L3 power state upon the upstream device programming the PCI Express device power state to D3.
2. During this duration, the Endpoint can receive a Power-Management Turn-Off (PME-turnoff) Message from its upstream partner.
3. The Endpoint core initiates a handshake with the user logic through `cfg_to_turnoff_n` (see [Table 6-13](#)) and expects a `cfg_turnoff_ok_n` back from the user logic.

4. A successful handshake results in a transmission of the Power Management Turn-off Acknowledge (PME-turnoff\_ack) Message by the Endpoint core to its upstream link partner.
5. The Endpoint core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for *removal* of power to the core.

Table 6-13: Power Management Handshaking Signals

Port Name	Direction	Description
cfg_to_turnoff_n	Output	Asserted if a power-down request TLP is received from the upstream device. After assertion <code>cfg_to_turnoff_n</code> remains asserted until the user asserts <code>cfg_turnoff_ok_n</code> .
cfg_turnoff_ok_n	Input	Asserted by the User Application when it is safe to power down

Power-down negotiation follows these steps:

1. Before power and clock are turned off, the Root Complex or the Hot-Plug controller in a downstream switch issues a PME\_TO\_Ack broadcast message.
2. When the Endpoint PIPE for PCIe core receives this TLP, it asserts `cfg_to_turnoff_n` to the User Application and starts polling the `cfg_turnoff_ok_n` input.
3. When the User Application detects the assertion of `cfg_to_turnoff_n`, it must complete any packet in progress and stop generating any new packets. After the User Application is ready to be turned off, it asserts `cfg_turnoff_ok_n` to the core. After assertion or of `cfg_turnoff_ok_n`, the User Application has committed to being turned off.
4. The Endpoint core sends a PME\_TO\_Ack when it detects assertion of `cfg_turnoff_ok_n`, as displayed in Figure 6-28.

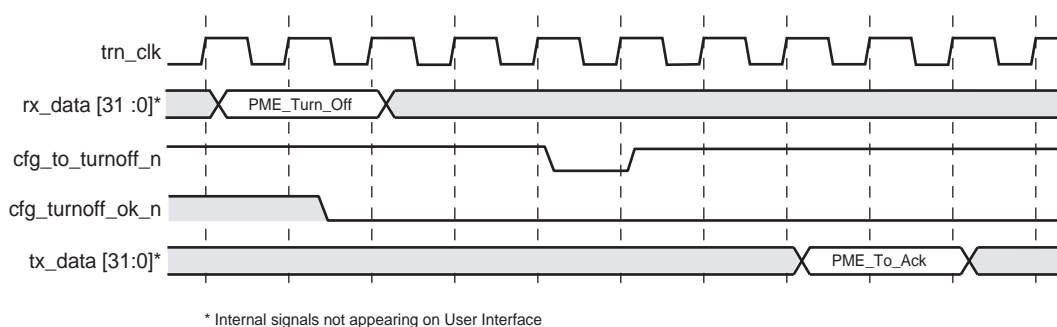


Figure 6-28: Power Management Handshaking

## Generating Interrupt Requests

The Endpoint PIPE for PCIe core supports sending interrupt request as either legacy interrupts or Message Signaled Interrupts (MSI). The mode is programmed using the MSI Enable bit in the Message Control Register of the MSI Capability Structure. For more information on the MSI capability structure please refer to section 6.8 of the PCI Local Base Specification v3.0. The state of the MSI Enable bit is reflected by the `cfg_interrupt_msienable` output:

- `cfg_interrupt_msienable = 0`: Legacy Interrupt (INTx) mode
- `cfg_interrupt_msienable = 1`: MSI mode

If the MSI Enable bit is set to a 1, then the core will generate MSI requests by sending Memory Write TLPs. If the MSI Enable bit is set to 0, the core generates legacy interrupt messages as long as the Interrupt Disable bit in the PCI Command Register is set to 0:

- `cfg_command[10] = 0`: interrupts enabled
- `cfg_command[10] = 1`: interrupts disabled (request are blocked by the core)

Note that the MSI Enable bit in the MSI control register and the Interrupt Disable bit in the PCI Command register are programmed by the Root Complex. The User Application has no direct control over these bits. The User Application does not typically need to poll the MSI Enable bit unless it needs to know the method in which its interrupt requests are being conveyed to the Root Complex. Regardless of the type of interrupts being used, the user initiates interrupt request through the use the interrupt signals listed in [Table 3-12, page 38](#).

The User Application requests interrupt service in one of two ways, each of which are described below. The User Application must determine which method to use based on the value of the `cfg_interrupt_msienable` output. When 0, the Legacy Interrupt method must be used; when 1, the MSI method.

## MSI Mode

- As shown in [Figure 6-29](#), the User Application first asserts `cfg_interrupt_n`. Additionally the User Application supplies a value on `cfg_interrupt_di[7:0]` if Multi-Vector MSI is enabled (see below).
- The core asserts `cfg_interrupt_rdy_n` to signal that the interrupt has been accepted and the core sends a MSI Memory Write TLP. On the following clock cycle, the User Application deasserts `cfg_interrupt_n` if no further interrupts are to be sent.

The MSI request is either a 32-bit addressable Memory Write TLP or a 64-bit addressable Memory Write TLP. The address is taken from the Message Address and Message Upper Address fields of the MSI Capability Structure, while the payload is taken from the Message Data field. These values are programmed by system software through configuration writes to the MSI Capability structure. When the core is configured for Multi-Vector MSI, system software may permit Multi-Vector MSI messages by programming a non-zero value to the Multiple Message Enable field.

The type of MSI TLP sent (32-bit addressable or 64-bit addressable) depends on the value of the Upper Address field in the MSI capability structure. By default, MSI messages are sent as 32-bit addressable Memory Write TLPs. MSI messages use 64-bit addressable Memory Write TLPs only if the system software programs a non-zero value into the Upper Address register.

When Multi-Vector MSI messages are enabled, the User Application may override one or more of the lower-order bits in the Message Data field of each transmitted MSI TLP to differentiate between the various MSI messages sent upstream. The number of lower-order bits in the Message Data field available to the User Application is determined by the lesser of the value of the Multiple Message Capable field, as set in the CORE Generator, and the Multiple Message Enable field, as set by system software and available as the `cfg_interrupt_mmenable[2:0]` core output. The core masks any bits in `cfg_interrupt_di[7:0]` which are not configured by system software via Multiple Message Enable.

The following pseudo-code shows the processing required:

```
// Value MSI_Vector_Num must be in range: 0 <= MSI_Vector_Num <=
(2^cfg_interrupt_mmenable)-1

if (cfg_interrupt_msienable) {           // MSI Enabled
    if (cfg_interrupt_mmenable > 0) {    // Multi-Vector MSI Enabled
        cfg_interrupt_di[7:0] = {Padding_0s, MSI_Vector_Num};
    } else {                             // Single-Vector MSI Enabled
        cfg_interrupt_di[7:0] = Padding_0s;
    }
} else {
    // Legacy Interrupts Enabled
}
```

Here for example:

1. If `cfg_interrupt_mmenable[2:0] == 000b` i.e 1 MSI Vector Enabled,  
then `cfg_interrupt_di[7:0] = cfg_interrupt_do[7:0];`
2. if `cfg_interrupt_mmenable[2:0] == 101b` i.e 32 MSI Vectors Enabled,  
then `cfg_interrupt_di[7:0] = {{cfg_interrupt_do[7:5]}, {MSI_Vector#}};`

where `MSI_Vector#` is a 5 bit value and is allowed to be `00000b <= MSI_Vector# <= 11111b`

## Legacy Interrupt Mode

- As shown in [Figure 6-29](#), the User Application first asserts `cfg_interrupt_n` and `cfg_interrupt_assert_n` to assert the interrupt. The User application should select a specific interrupt (INTA, INTB, INTC, or INTD) using `cfg_interrupt_di[7:0]` as shown in [Table 3-12, page 38](#).
- The core then asserts `cfg_interrupt_rdy_n` to indicate the interrupt has been accepted. On the following clock cycle, the User Application deasserts `cfg_interrupt_n` and, if the Interrupt Disable bit in the PCI Command register is set to 0, the core sends an assert interrupt message (Assert\_INTA, Assert\_INTB, and so forth).
- After the User Application has determined that the interrupt has been serviced, it asserts `cfg_interrupt_n` while deasserting `cfg_interrupt_assert_n` to deassert the interrupt. The appropriate interrupt must be indicated via `cfg_interrupt_di[7:0]`.
- The core then asserts `cfg_interrupt_rdy_n` to indicate the interrupt deassertion has been accepted. On the following clock cycle, the User Application deasserts `cfg_interrupt_n` and the core sends a deassert interrupt message (Deassert\_INTA, Deassert\_INTB, and so forth).



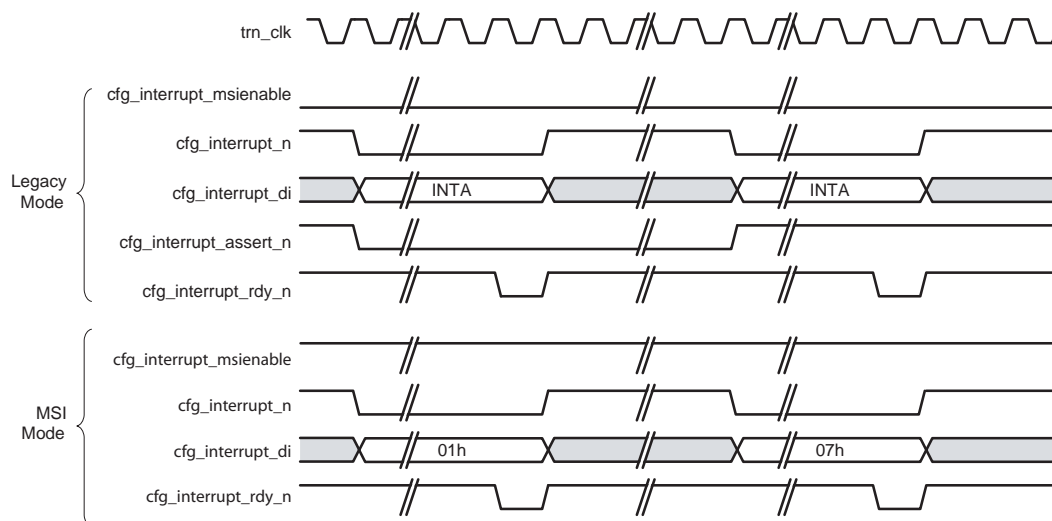


Figure 6-29: Requesting Interrupt Service: MSI and Legacy Mode



# Core Constraints

---

The Endpoint PIPE for PCI Express core requires the specification of timing and other physical implementation constraints to meet specified PCI Express performance requirements. These constraints are provided with the core in a User Constraints File (UCF).

To achieve consistent implementation results, a UCF file containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of a UCF file or specific constraints, see the *Xilinx Libraries Guide* and/or *Development System Reference Guide*.

Constraints provided with Endpoint PIPE for PCIe solution have been tested in hardware and provide consistent results. Constraints can be modified, but modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

## Optional User Constraints

It is acceptable to add additional constraints to cover other logic implemented by the user. While the UCF file shipped with the core is designed to adequately constrain the Endpoint PIPE for PCIe core itself, it cannot adequately constrain user-implemented logic interfaced to the core; this is the user's responsibility.

## Required Constraints

The design constraints for the core have been organized in the UCF file to let users clearly identify required constraints and their purpose. The following sections describe various parts of a typical UCF file. The UCF file included in the product may differ from what is defined in this document; do not edit these constraints.

### Device, Package, and Speedgrade Selection

Identifies the device, part, and package to be used for this implementation.

```
CONFIG PART = XC3S1000-FG676-5 ;
```

Many of the constraints that follow this declaration are part and package specific.

## I/O Location Assignment Constraints

### System Reset (Input) Signal

The `sys_reset_n` signal should be obtained from the PCI Express edge connector. For slot-based form factors, a system reset signal is generally present on the connector. For cable based form factors, a system reset signal may not be available. In this case, the system reset signal must be generated locally by some form of supervisory circuit.

```
NET "sys_reset_n" LOC = "AE4" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
```

### PIPE Receive (Input) Signals

These inputs must be properly terminated SSTL2\_I signals. Here, the DCI version of SSTL2\_I is used. This eliminates the need for external termination at the FPGA receivers. This does NOT eliminate the need for external termination at the PHY drivers. You must properly set the VRN and VRP reference resistors for the banks in use. If you elect not to use the DCI version of SSTL2\_I, you can change the IOSTANDARDS and you must then include the required termination at both the FPGA and the PHY. Please consult the core documentation and the FPGA device data sheet for additional information. When routing these signals on the board, they should be length matched to minimize skew.

```
NET "rxclk" LOC = "AE13" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "phystatus" LOC = "AF12" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxvalid" LOC = "AD12" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxstatus<2>" LOC = "AC11" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxstatus<1>" LOC = "AD10" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxstatus<0>" LOC = "AC10" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<7>" LOC = "AF8" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<6>" LOC = "AE8" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<5>" LOC = "AC7" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<4>" LOC = "AF6" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<3>" LOC = "AE6" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<2>" LOC = "AD6" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<1>" LOC = "AC6" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<0>" LOC = "AE5" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<7>" LOC = "AD5" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
NET "rxdata<6>" LOC = "AF4" | IOSTANDARD = SSTL2_I_DCI | NODELAY ;
```

### PIPE Transmit (Output) Signals

These outputs must be properly terminated SSTL2\_I signals. Here, the DCI version of SSTL2\_I is used. This eliminates the need for external termination at the FPGA drivers. This does not eliminate the need for external termination at the PHY receivers. You must properly set the VRN and VRP reference resistors for the banks in use. If you elect not to use the DCI version of SSTL2\_I, you can change the IOSTANDARD and you must then include the required termination at both the FPGA and the PHY. Please consult the core documentation and the FPGA device data sheet for additional information. When routing these signals on the board, they should be length matched to minimize skew.

```
NET "resetsn" LOC = "AF24" | IOSTANDARD = SSTL2_I_DCI ;
NET "rxpolarity" LOC = "AE24" | IOSTANDARD = SSTL2_I_DCI ;
NET "txelecidle" LOC = "AF23" | IOSTANDARD = SSTL2_I_DCI ;
NET "txcompliance" LOC = "AE23" | IOSTANDARD = SSTL2_I_DCI ;
NET "powerdown<1>" LOC = "AD23" | IOSTANDARD = SSTL2_I_DCI ;
NET "powerdown<0>" LOC = "AF22" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<7>" LOC = "AE22" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<6>" LOC = "AD21" | IOSTANDARD = SSTL2_I_DCI ;
```

```

NET "txclk" LOC = "AE21" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<7>" LOC = "AD21" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<6>" LOC = "AF20" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<5>" LOC = "AE20" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<4>" LOC = "AF19" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<3>" LOC = "AE19" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<2>" LOC = "AF15" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<1>" LOC = "AE15" | IOSTANDARD = SSTL2_I_DCI ;
NET "txdata<0>" LOC = "AD15" | IOSTANDARD = SSTL2_I_DCI ;

```

## Timing Constraints

### Ignore Timing on Nets

Some non-critical asynchronous reset nets are ignored for static timing analysis.

```

NET ep/BU2/U0/pci_exp_1_lane_epipe_ep0/plm/kh2_mgt/reg_rst* TIG ;

NET sys_reset_n TIG ;

```

### Time Names for Clock Signals

Endpoint for PCI Express receives 250 MHz forwarded clock signal from external PHY device.

```

NET "rxclk" TNM_NET = "TNM_NET_RXCLK" ;

```

### Time Specifications for Clock Signals

```

TIMESPEC "TS_RXCLK" = PERIOD "TNM_NET_RXCLK" 250 MHz HIGH 50 % PRIORITY 0 ;

```

Note that a PRIORITY value has been assigned to the TIMESPEC for timing analysis purposes. This constraint specifies the clock frequency and duty cycle of the clock signal.

To assist timing analysis process, the following timing ignore property is attached to the following instance of a BUFGMUX inside the core. Please change the design path to the instance as it exists in your design.

```

PIN ep/BU2/U0/pci_exp_1_lane_epipe_ep0/plm/kh2_mgt/kh2_bufg.I0 TIG ;

```

### Time Specifications for PIPE Signals

The following constraints are specified to verify that PIPE input and output registers are packing into IOBs. If you encounter timing failures associated with these constraints, check that you are using map with the correct option, *-pr b*. You should manually verify in the .mrp report file that the rxclk input register is properly packed into the IOB. It is illegal to specify an OFFSET constraint on a clock input with respect to itself, so no constraint is present for rxclk. The OFFSET constraint for txclk is larger than the other outputs to account for the fact that it is clocked by both edges of rxclk; the timing tools report this as an extra half-cycle worth of delay.

```

NET "phystatus" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxvalid" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxstatus<2>" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxstatus<1>" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxstatus<0>" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdatak" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<0>" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<1>" OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;

```

```

NET "rxdata<2>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<3>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<4>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<5>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<6>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxdata<7>"          OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;
NET "rxelecidle"         OFFSET = IN 1 ns VALID 2 ns BEFORE "rxclk" ;

NET "resetsn"            OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "rxpolarity"         OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txelecidle"         OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txcompliance"       OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "powerdown<1>"       OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "powerdown<0>"       OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdatak"            OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdetectrx_loopback" OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<7>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<6>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<5>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<4>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<3>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<2>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<1>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txdata<0>"          OFFSET = OUT 4 ns AFTER "rxclk" ;
NET "txclk"              OFFSET = OUT 6 ns AFTER "rxclk" ;

```

## Timing Budget for PIPE Signals

### NXP PX1011B-EL1 To XC3S1000-5

#### Output Data Valid Window for PX1011B-EL1

All output signals from PX1011B-EL1 are synchronous to rxclk. The rxclk is a source synchronous clock center-aligned with the data. This creates a data valid window of 3.0 ns, with the data valid 1.5 ns before the rising edge of rxclk and 1.5 ns after the rising edge. This timing information is from the device data sheet and is assumed to account for all skew associated with the device.

#### Input Data Setup/Hold Window for XC3S1000-5

All input signals to the FPGA are synchronous to rxclk. The rxclk is a source synchronous clock center-aligned with the data. This design uses an implementation of active phase alignment from Xilinx Application Note 268.

The required input data setup/hold window at the device is based on the following parameters:

- Tsamp, 750 ps

This parameter indicates the total sampling error at the input registers across voltage, temperature, and process. For a single data rate interface, this number includes the DCM jitter, the DCM phase shift resolution, and DCM phase offset. This value is an estimate based on source synchronous characterization data.

- package skew, +/- 150 ps

This is the worst case package skew between pins used for rxclk and the inputs. This value is a conservative estimate based on published package skew data for another device family.

- clock skew, +/- 200 ps

This is the worst case clock skew between pins used for rxclk and the inputs. This value is measured for the clk2x signal using the delay function in FPGA Editor for this specific device and pinout.

- The input data valid window computed as the result:

$$R_x = [750 + 150 + 200] = 1100 \text{ ps}$$

### Timing Budget from PX1011B-EL to XC3S1000-5

The remaining slack is sufficient to cover skew in the signal routing, inter-symbol interference, and other analog considerations. Xilinx is not responsible for board-related failures and for this reason strongly recommends that all users perform simulations.

## XC3S1000-5 To NXP PX1011B-EL1

### Output Data Valid Window XC3S1000-5

All output signals from XC3S1000-5 are synchronous to txclk. The txclk is a source synchronous clock center-aligned with the data. The internal clock signals are generated from a 250 MHz reference (rxclk) that enters the DCM using the divide by two option. Three DCM outputs are used; clk2x is 250 MHz, clk0 is 125 MHz, and clkdv is 62.5 MHz. The clk0 is used as the DCM feedback clock. The clk2x signal is used to forward txclk via a DDR output register. The txclk yields an ideal data valid window of 4.0 ns, with the data valid 2.0 ns before the rising edge and 2.0 ns after the rising edge. This ideal window must then be adjusted for sources of skew:

- DCM output jitter, +/- 200 ps

This output jitter figure is for the 2x output, obtained from the device data sheet. Jitter is the only number in the timing budget defined with absolute magnitude; in this case, the jitter magnitude used in calculations is twice the value shown above.

- duty cycle distortion, +/- 400 ps

In this design, clk2x is used to forward txclk via a DDR output register using local clock inversion. The negative edge of clk2x causes the rising edge on txclk. For this reason, duty cycle distortion is an important consideration. This value is an estimate based on source synchronous characterization data.

- package skew, +/- 150 ps

This is the worst case package skew between pins used for txclk and the outputs. This value is a conservative estimate based on published package skew data for another device family.

- clock skew, +/- 150ps

This is the worst case clock skew between pins used for txclk and the outputs. This value is measured for the clk2x signal using the delay function in FPGA Editor for this specific device and pinout.

The clk\_in\_clkfb\_phase defines the amount of phase offset between the clock input and the feedback input of the DCM.

- The output data valid window computed as the result:  
$$T_x = 4000 - [400 + 400 + 150 + 150] = 2900 \text{ ps}$$
- Input data setup/hold window, PX1011B-EL1:  
All input signals to the PX1011B-EL are synchronous to txclk. The txclk is a source synchronous clock center-aligned with the data. The required input data setup/hold window at the device is 1.0 ns, with the data setup 0.5 ns before the rising edge of txclk and data hold 0.5 ns after the rising edge. This timing information is from the device data sheet and is assumed to account for all skew associated with the device.

#### Timing Budget from XC3S1000-5 to PX1011B-EL

The remaining slack is sufficient to cover skew in the signal routing, inter-symbol interference, and other analog considerations. Xilinx is not responsible for board-related failures and for this reason strongly recommends that all users perform simulations.



## Tracking Receive-buffer Space for Inbound Completions

The Endpoint PIPE for PCI Express core sets aside 1,536 double words of receive-buffer space for inbound Completions. All data in a Completion packet uses this space, including the header and payload.

Follow the steps below to manage the inbound Completion space:

- Determine the amount of Completion space needed for a given request.
- Make sure that there is enough Completion space available for the request.
- Allocate Completion space for the request when it is transmitted.
- De-allocate the same amount of Completion space as Completion data is read from the receive data path or if a Completion timeout occurs.

[Table A-1](#) defines how much receive-buffer space must be allocated for each type of request that can be answered with a Completion.

**Table A-1: Receive-buffer Completion Allocations**

Type	Receive-Buffer Allocation	Comments
I/O Read	4 DW	Non-burst only – Completion header + payload = 4 DW
I/O Write	3 DW	No Completion payload, header = 3 DW rounded-up
Memory Read (RCB <sup>a</sup> = 64B, default)	19 DW per aligned block of 64 bytes	Returned data can require multiple Completions, each of which must start and end at a naturally aligned 64-byte or 128-byte boundary (based on the RCB setting in the Configuration Space) except for the start and end addresses
Memory Read (RCB = 128B)	35 DW per aligned block of 128 bytes	

a. RCB stands for Read Completion Boundary.

### I/O Completions

Allocating and de-allocating space for I/O Completions is relatively easy because each request can only be answered with a single Completion packet.

Follow the steps below to allocate and de-allocate space for I/O Completions:

1. Assuming receive-buffer space is available, allocate 4 DW upon transmitting an I/O Read or I/O Write.
2. When the corresponding Completion is read from the receive data path or when the Completion times out, de-allocate 4 DW.

## Memory Read Completions

A Memory Read can be answered with multiple Completions that when combined return all the requested data. To make room for packet-header overhead, the User Application must allocate enough space for the maximum number of Completions that may be returned.

To make this process easier, the *PCI Express Base Specification* quantizes the length of all Completion packets such that each must start and end on a naturally aligned Read Completion Boundary (RCB) unless it services the starting or ending address of the original request. The value of RCB is determined by Configuration bit `cfg_lcommand[3]`:

- `cfg_lcommand[3] = 0`: RCB = 64 bytes
- `cfg_lcommand[3] = 1`: RCB = 128 bytes

If the User Application chooses not to read the RCB value, it must default to 64 bytes.

To manage Completion space for Memory Reads, the steps are:

1. Calculate the number of RCB-blocks that the returned data requires. Any fraction must be rounded up. For example, if a Memory Read requested 8 bytes of data from address 7Ch, the returned data requires two RCB blocks: 7Ch-7Fh and 80h-83h.
2. Calculate the total amount of receive-buffer space required by multiplying the number of RCB blocks by 19 DW (RCB = 64B) or 35 DW (RCB = 128B). These are the sizes of packets with 64 bytes and 128 bytes of payload, respectively, regardless of TLP Digest.
3. Assuming receive-buffer space is available, allocate the calculated amount.
4. As corresponding Completions are received, de-allocate 19 DW (RCB = 64B) or 35 DW (RCB = 128B) each time a packet crosses an RCB boundary, ends at an RCB boundary, or finishes the balance of the original request.<sup>(1)</sup>
5. If a timeout occurs before the original request is completely serviced, de-allocate only the buffer space that represents that part of the original request not yet received.

**Note:** To maximize receive-buffer usage, a designer can choose to allocate smaller amounts of space for quanta that do not fill an entire RCB block, if he is willing to design more complicated tracking logic. This can be of most benefit if the User Application only issues Memory Reads for single DWORDs.

---

1. Only one de-allocation is necessary if a Completion both ends at an RCB boundary and finishes the balance of the original Request.

# Programmed Input Output Example Design

---

Programmed Input Output (PIO) transactions are generally used by a PCI Express system host CPU to access Memory Mapped Input Output (MMIO) and Configuration Mapped Input Output (CMIO) locations in the PCI Express fabric. Endpoints for PCI Express accept Memory and IO Write transactions and respond to Memory and IO Read transactions with Completion with Data transactions.

The PIO example design (PIO design) is included with the PCIe core generated by the CORE Generator. This design allows users to easily bring up their system board with a known established working design to verify the link and functionality of the board.

**Note:** The PIO example design is shared by the Endpoint for PCI Express, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This appendix represents all the solutions generically using the name Endpoint for PCI Express.

## System Overview

The PIO design is a simple target-only application that interfaces with the Endpoint for PCI Express core's Transaction (TRN) interface and is provided as a starting point for customers to build their own designs. The following features are included:

- Four transaction-specific 2 kB target regions using the internal Xilinx FPGA block RAMs, providing a total target space of 8192 bytes
- Supports single DWORD payload Read and Write PCI Express transactions to 32/64 bit address memory spaces and IO space with support for completion TLPs
- Utilizes the core's `trn_rbar_hit_n[6:0]` signals to differentiate between TLP destination Base Address Registers
- Provides separate implementations optimized for 32-bit and 64-bit TRN interfaces

Figure B-1 illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and an Endpoint for PCI Express. PIO operations move data *downstream* from the Root Complex (CPU register) to the Endpoint, and/or *upstream* from the Endpoint to the Root Complex (CPU register). In either case, the PCI Express protocol request to move the data is initiated by the host CPU.

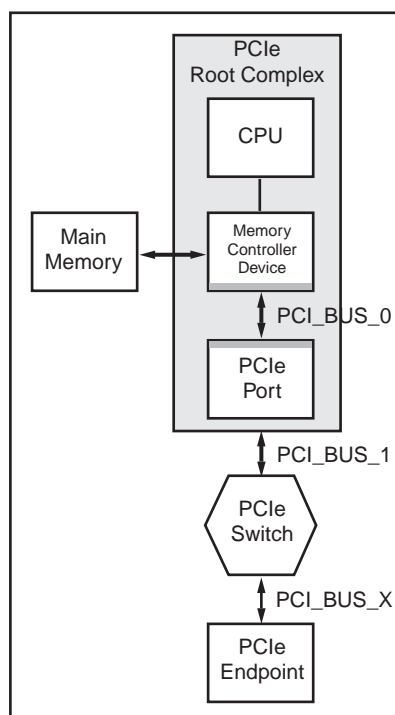


Figure B-1: **PCI Express System Overview**

Data is moved downstream when the CPU issues a store register to a MMIO address command. The Root Complex typically generates a Memory Write TLP with the appropriate MMIO location address, byte enables and the register contents. The transaction terminates when the Endpoint receives the Memory Write TLP and updates the corresponding local register.

Data is moved upstream when the CPU issues a load register from a MMIO address command. The Root Complex typically generates a Memory Read TLP with the appropriate MMIO location address and byte enables. The Endpoint generates a Completion with Data TLP once it receives the Memory Read TLP. The Completion is steered to the Root Complex and payload is loaded into the target register, completing the transaction.

## PIO Hardware

The PIO design implements a 8192 byte target space in FPGA block RAM, behind the Endpoint for PCI Express core. This 32-bit target space is accessible through single DWORD IO Read, IO Write, Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs.

The PIO design generates a completion with 1 DWORD of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or IO Read TLP request presented to it by the Endpoint for PCI Express core. In addition, the PIO design returns a completion without data with successful status for IO Write TLP request.

The PIO design processes a Memory or IO Write TLP with 1 DWORD payload by updating the payload into the target address in the FPGA block RAM space.

## Base Address Register Support

The PIO design supports four discrete target spaces, each consisting of a 2 kB block of memory represented by a separate Base Address Register (BAR). Using the default parameters, the CORE Generator produces a core configured to work with the PIO design defined in this section, consisting of the following:

- One IO Space BAR
- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR
- One Expansion ROM BAR.

Users may change the default parameters used by the PIO design; however, in some cases they may need to change the back-end user application depending on their system. See [“Changing CORE Generator Default BAR Settings”](#) for information about changing the default CORE Generator parameters and the affect on the PIO design.

Each of the four 2 kB address spaces represented by the BARs corresponds to one of four 2 kB address regions in the PIO design. Each 2 kB region is implemented using a 2 kB dual-port block RAM. As transactions are received by the Endpoint for PCI Express core, the core decodes the address and determines which of the four regions is being targeted. The core presents the TLP to the PIO design and asserts the appropriate bits of `trn_rbar_hit_n[6:0]`, as defined in [Table B-1](#).

**Table B-1: TLP Traffic Types**

Block RAM	TLP Transaction Type	Default BAR	trn_rbar_hit_n[6:0]
ep_mem0	IO TLP transactions	0	111_1110b
ep_mem1	32-bit address Memory TLP transactions	1	111_1101b
ep_mem2	64-bit address Memory TLP transactions	2-3	111_0011b
ep_mem3	32-bit address Memory TLP transactions destined for EROM	Exp. ROM	011_1111b

## Changing CORE Generator Default BAR Settings

Users can change the CORE Generator parameters and continue use the PIO design to create customized PIO design Verilog source to match the selected BAR settings. However, because the PIO design parameters are more limited than the core parameters, the following example design limitations should be considered when changing the default CORE Generator parameters:

- The example design supports one IO space BAR, two 32-bit Memory spaces (one of which must be the Expansion ROM space), and one 64-bit Memory space. If these limits are exceeded, only the first space of a given type will be active—accesses to the other spaces will not result in completions.
- Each space is implemented with a 2 kB memory. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 kB limit wrap around and overlap the 2 kB memory space.

Although there are limitations to the PIO design, Verilog source code is provided so the user can tailor the example design to their specific needs.

## TLP Data Flow

This section defines the data flow of a TLP successfully processed by the PIO design. For detailed information about the interface signals within the sub-blocks of the PIO design, see “Receive Path,” page 121 and “Transmit Path,” page 122.

The PIO design successfully processes single DWORD payload Memory Read and Write TLPs and IO Read and Write TLPs. Memory Read or Memory Write TLPs of lengths larger than one DWORD are not processed correctly by the PIO design; however, the PCI Express core *does* accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than 1 DWORD, the TLP is received completely from the core and discarded. No corresponding completion is generated.

### Memory/IO Write TLP Processing

When the core receives a Memory or IO Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the Endpoint for PCI Express core passes the TLP to the Receive TRN interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Receive TRN interface also asserts the appropriate `trn_rbar_hit_n[6:0]` signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design’s RX State Machine processes the incoming Write TLP and extracts the TLPs data and relevant address fields so that it can pass this along to the PIO design’s internal block RAM write request controller.

Based on the specific `trn_rbar_hit_n[6:0]` signal asserted, the RX State Machine indicates to the internal write controller the appropriate 2 kB block RAM to use prior to asserting the write enable request. For example, if an IO Write Request is received by the core targeting BAR0, the core passes the TLP to the PIO design and asserts `trn_rbar_hit_n[0]`. The RX State machine extracts the lower address bits and the data field from the IO Write TLP and instructs the internal Memory Write controller to begin a write to the block RAM.

In this example, the assertion of `trn_rbar_hit_n[0]` instructed the PIO memory write controller to access `ep_mem0` (which by default represents 2 kB of IO space). While the write is being carried out to the FPGA block RAM, the PIO design RX state machine deasserts the `trn_rdst_rdy_n` signal, causing the receive TRN interface to stall receiving any further TLPs until the internal Memory Write controller completes the write to the block RAM. Note that deasserting `trn_rdst_rdy_n` in this way is not required for all designs using the core—the PIO design uses this method to simplify the control logic of the RX state machine.

### Memory/IO Read TLP Processing

When the Endpoint for PCI Express core receives a Memory or IO Read TLP, the TLP destination address and transaction type are compared with the values programmed in the core BARs. If the TLP passes this comparison check, the Endpoint for PCI Express core passes the TLP to the Receive TRN interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Receive TRN interface also asserts the appropriate `trn_rbar_hit_n[6:0]` signal to indicate to the

PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design's state machine processes the incoming Read TLP and extracts the relevant TLP information and passes it along to the PIO design's internal block RAM read request controller.

Based on the specific `trn_rbar_hit_n[6:0]` signal asserted, the RX state machine indicates to the internal read request controller the appropriate 2 kB block RAM to use prior to asserting the read enable request. For example, if a Memory Read 32 Request TLP is received by the core targeting the Expansion ROM BAR, the core passes the TLP to the PIO design and asserts `trn_rbar_hit_n[6]`. The RX State machine extracts the lower address bits from the Memory 32 Read TLP and instructs the internal Memory Read Request controller to start a read operation.

In this example, the assertion of `trn_rbar_hit_n[6]` instructs the PIO memory read controller to access the EROM space, which by default represents 2 kB of memory space. A notable difference in handling of memory write and read TLPs is the requirement of the receiving device to return a Completion with Data TLP in the case of memory or IO read request.

While the read is being processed, the PIO design RX state machine deasserts the `trn_rdst_rdy_n` signal, causing the receive TRN interface to stall receiving any further TLPs until the internal Memory Read controller completes the read to the block RAM and generates the completion. Note that deasserting `trn_rdst_rdy_n` in this way is not required for all designs using the core—the PIO design uses this method to simplify the control logic of the RX state machine.

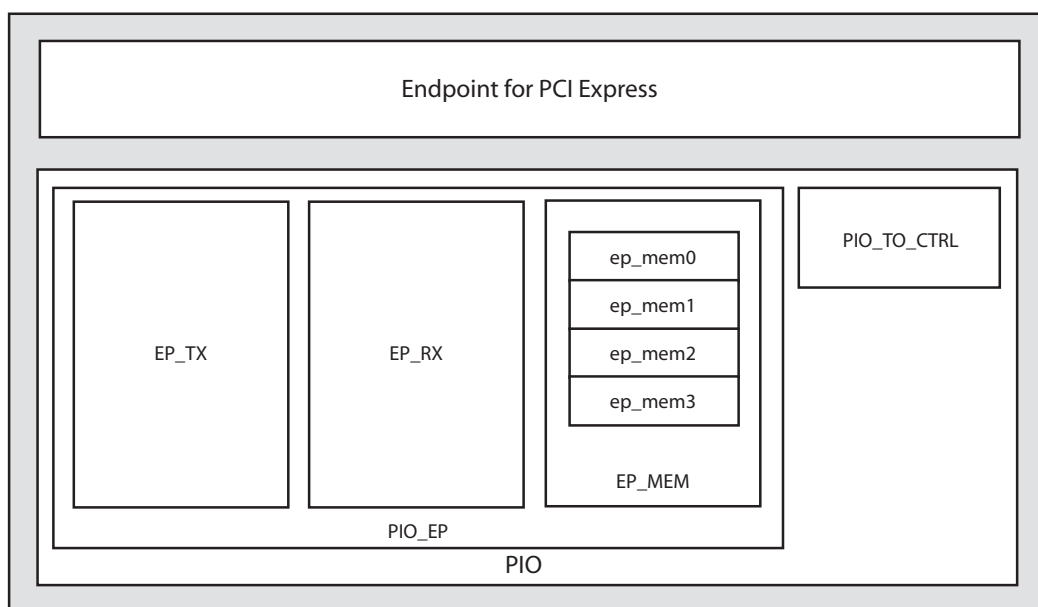
## PIO File Structure

Table B-2 defines the PIO design file structure. Note that based on the specific core targeted, not all files delivered by CORE Generator are necessary, and some files may or may not be delivered. The major difference is that some of the cores use a 32-bit user data path, others use a 64-bit data path, and the PIO design works with both. The width of the data path depends on the specific core being targeted.

**Table B-2: PIO Design File Structure**

File	Description
PIO.v	Top-level design wrapper
PIO_EP.v	PIO application module
PIO_TO_CTRL.v	PIO turn-off controller module
PIO_32.v	32b interface macro define
PIO_64.v	64b macro define
PIO_32_RX_ENGINE.v	32b Receive engine
PIO_32_TX_ENGINE.v	32b Transmit engine
PIO_64_RX_ENGINE.v	64b Receive engine
PIO_64_TX_ENGINE.v	64b Transmit engine
PIO_EP_MEM_ACCESS.v	Endpoint memory access module
EP_MEM.v	Endpoint memory

Figure B-2 shows the various components of the PIO design, which is separated into four main parts: the TX Engine, RX Engine, Memory Access Controller, and Power Management Turn-Off Controller.

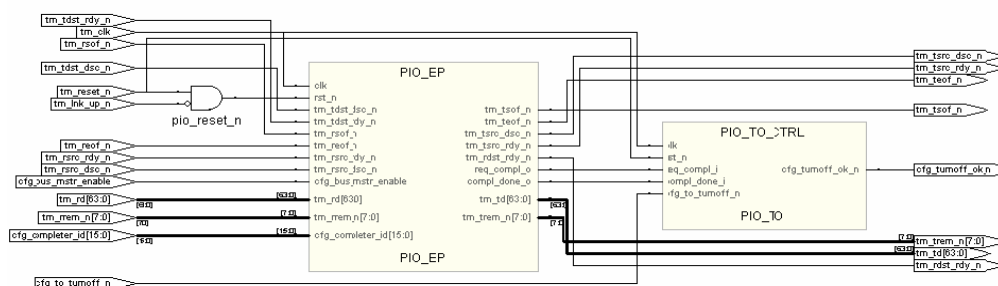


**Figure B-2: PIO Design Components**

## PIO Application

Figures B-3 and B-4 depict 64-bit and 32-bit PIO application top-level connectivity, respectively. The data path width, either 32-bits or 64-bits, depends on which core is used. The PIO\_EP module contains the PIO FPGA block RAM memory modules and the transmit and receive engines. The PIO\_TO\_CTRL module is the Endpoint Turn-Off controller unit, which responds to power turn-off message from the host CPU with an acknowledgement.

The PIO\_EP module connects to the core's Transaction (TRN) and Configuration (CFG) interfaces.



**Figure B-3: PIO 64-bit Application**



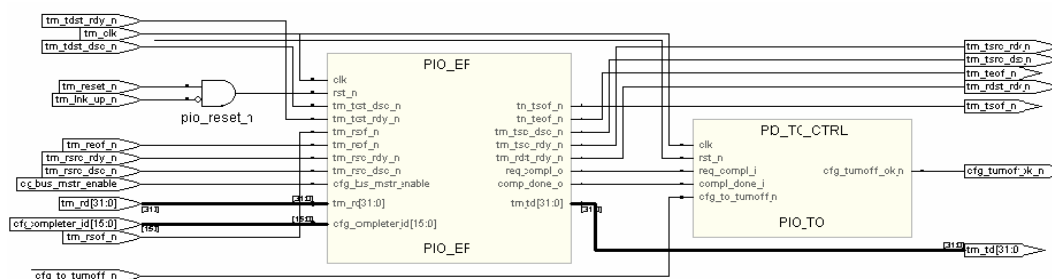


Figure B-4: PIO 32-bit Application

## Receive Path

Figure B-5 illustrates the PIO\_32\_RX\_ENGINE and PIO\_64\_RX\_ENGINE modules. The data path of the module must match the data path of the core being used. These modules connect with receive Transaction (trn\_r\*) interface.

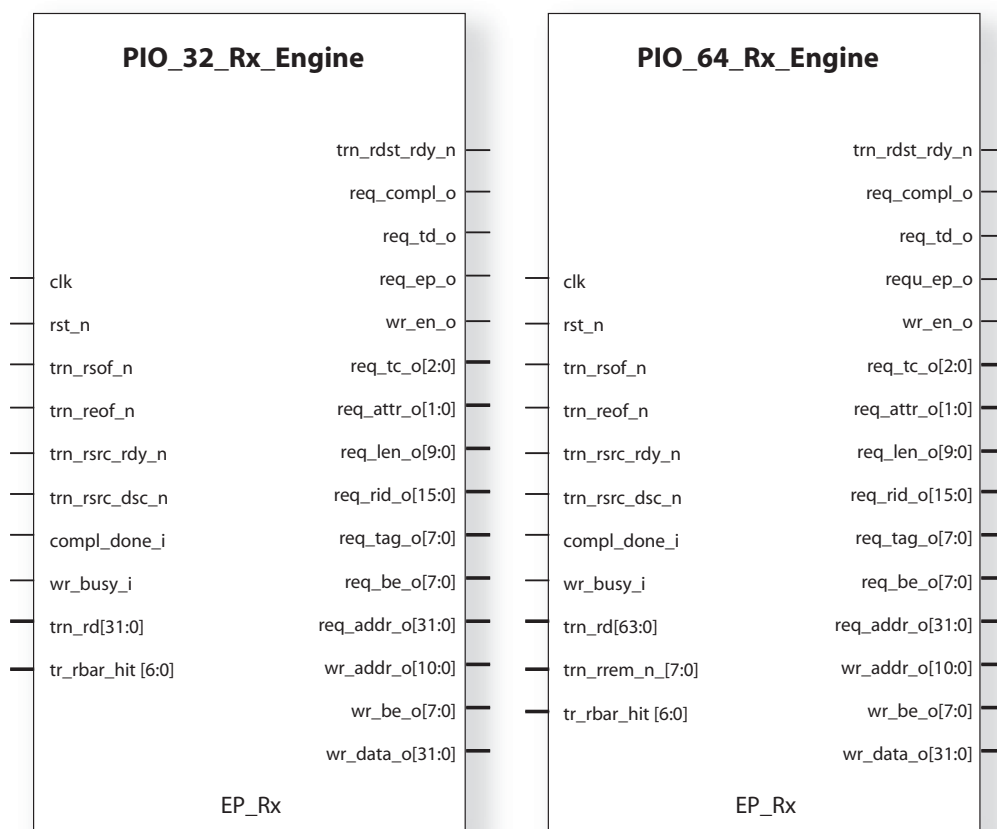


Figure B-5: Rx Engines

The PIO\_32\_RX\_ENGINE and PIO\_64\_RX\_ENGINE modules receive and parse incoming read and write TLPs.

The RX engine parses 1 DWORD 32 and 64-bit addressable memory and IO read requests. The RX state machine extracts needed information from the TLP and passes it to the memory controller, as defined in [Table B-3](#).

**Table B-3: Rx Engine: Read Outputs**

Port	Description
req_compl_o	Completion request (active high)
req_td_o	Request TLP Digest bit
req_ep_o	Request Error Poisoning bit
req_tc_o[2:0]	Request Traffic Class
req_attr_o[1:0]	Request Attributes
req_len_o[9:0]	Request Length
req_rid_o[15:0]	Request Requester Identifier
req_tag_o[7:0]	Request Tag
req_be_o[7:0]	Request Byte Enable
req_addr_o[10:0]	Request Address

The RX Engine parses 1 DWORD 32- and 64-bit addressable memory and IO write requests. The RX state machine extracts needed information from the TLP and passes it to the memory controller, as defined in [Table B-4](#).

**Table B-4: Rx Engine: Write Outputs**

Port	Description
wr_en_o	Write received
wr_addr_o[10:0]	Write address
wr_be_o[7:0]	Write byte enable
wr_data_o[31:0]	Write data

The read data path stops accepting new transactions from the core while the application is processing the current TLP. This is accomplished by `trn_rdst_rdy_n` deassertion. For an ongoing Memory or IO Read transaction, the module waits for `compl_done_i` input to be asserted before it accepts the next TLP, while an ongoing Memory or IO Write transaction is deemed complete after `wr_busy_i` is deasserted.

## Transmit Path

[Figure B-6](#) shows the `PIO_32_TX_ENGINE` and `PIO_64_TX_ENGINE` modules. The data path of the module must match the data path of the core being used. These modules connect with core transmit Transaction (`trn_r*`) interface.

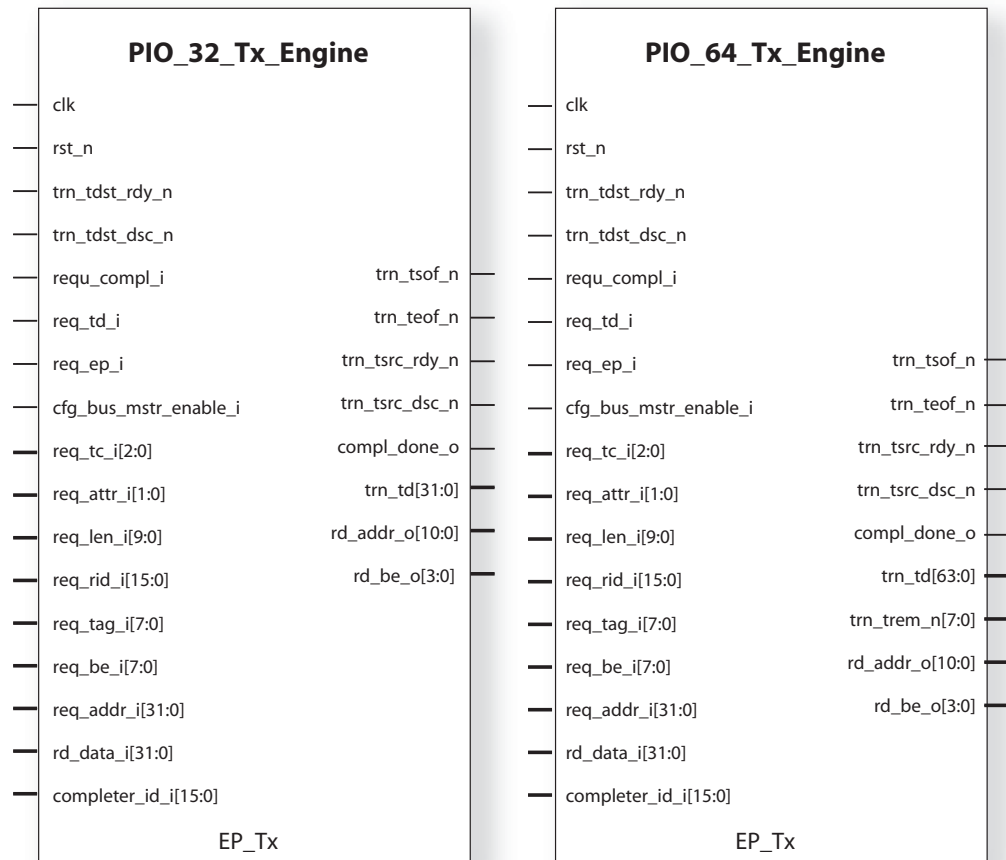


Figure B-6: Tx Engines

The PIO\_32\_TX\_ENGINE and PIO\_64\_TX\_ENGINE modules generate completions for received memory and IO read TLPs. The PIO design does not generate outbound read or write requests. However, users can add this functionality to further customize the design.

The PIO\_32\_TX\_ENGINE and PIO\_64\_TX\_ENGINE modules generate completions in response to 1 DWORD 32 and 64-bit addressable memory and IO read requests. Information necessary to generate the completion is passed to the TX Engine, as defined in Table B-5.

Table B-5: Tx Engine Inputs

Port	Description
req_compl_i	Completion request (active high)
req_td_i	Request TLP Digest bit
req_ep_i	Request Error Poisoning bit
req_tc_i[2:0]	Request Traffic Class
req_attr_i[1:0]	Request Attributes

Table B-5: Tx Engine Inputs (Cont'd)

Port	Description
req_len_i[9:0]	Request Length
req_rid_i[15:0]	Request Requester Identifier
req_tag_i[7:0]	Request Tag
req_be_i[7:0]	Request Byte Enable
req_addr_i[10:0]	Request Address

After the completion is sent, the TX engine asserts the `compl_done_i` output indicating to the RX engine that it can assert `trn_rdst_rdy_n` and continue receiving TLPs.

## Endpoint Memory

Figure B-7 displays the `PIO_EP_MEM_ACCESS` module. This module contains the Endpoint memory space.

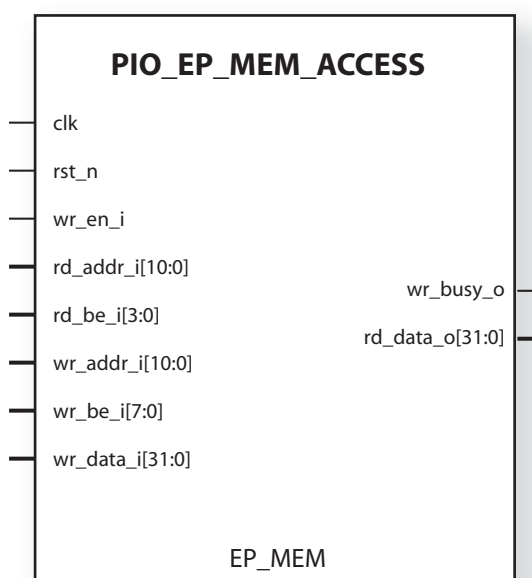


Figure B-7: EP Memory Access

The `PIO_EP_MEM_ACCESS` module processes data written to the memory from incoming Memory and IO Write TLPs and provides data read from the memory in response to Memory and IO Read TLPs.

The `EP_MEM` module processes 1 DWORD 32- and 64-bit addressable Memory and IO Write requests based on the information received from the RX Engine, as defined in

**Table B-6.** While the memory controller is processing the write, it asserts the `wr_busy_o` output indicating it is busy.

**Table B-6: EP Memory: Write Inputs**

Port	Description
<code>wr_en_i</code>	Write received
<code>wr_addr_i[10:0]</code>	Write address
<code>wr_be_i[7:0]</code>	Write byte enable
<code>wr_data_i[31:0]</code>	Write data

Both 32 and 64-bit Memory and IO Read requests of one DWORD are processed based on the following inputs, as defined in **Table B-7**. After the read request is processed, the data is returned on `rd_data_o[31:0]`.

**Table B-7: EP Memory: Read Inputs**

Port	Description
<code>req_be_i[7:0]</code>	Request Byte Enable
<code>req_addr_i[31:0]</code>	Request Address

## PIO Operation

### PIO Read Transaction

**Figure B-8** depicts a Back-To-Back Memory Read request to the PIO design. The receive engine deasserts `trn_rdst_rdy_n` as soon as the first TLP is completely received. The next Read transaction is accepted only after `compl_done_o` is asserted by the transmit engine, indicating that Completion for the first request was successfully transmitted.

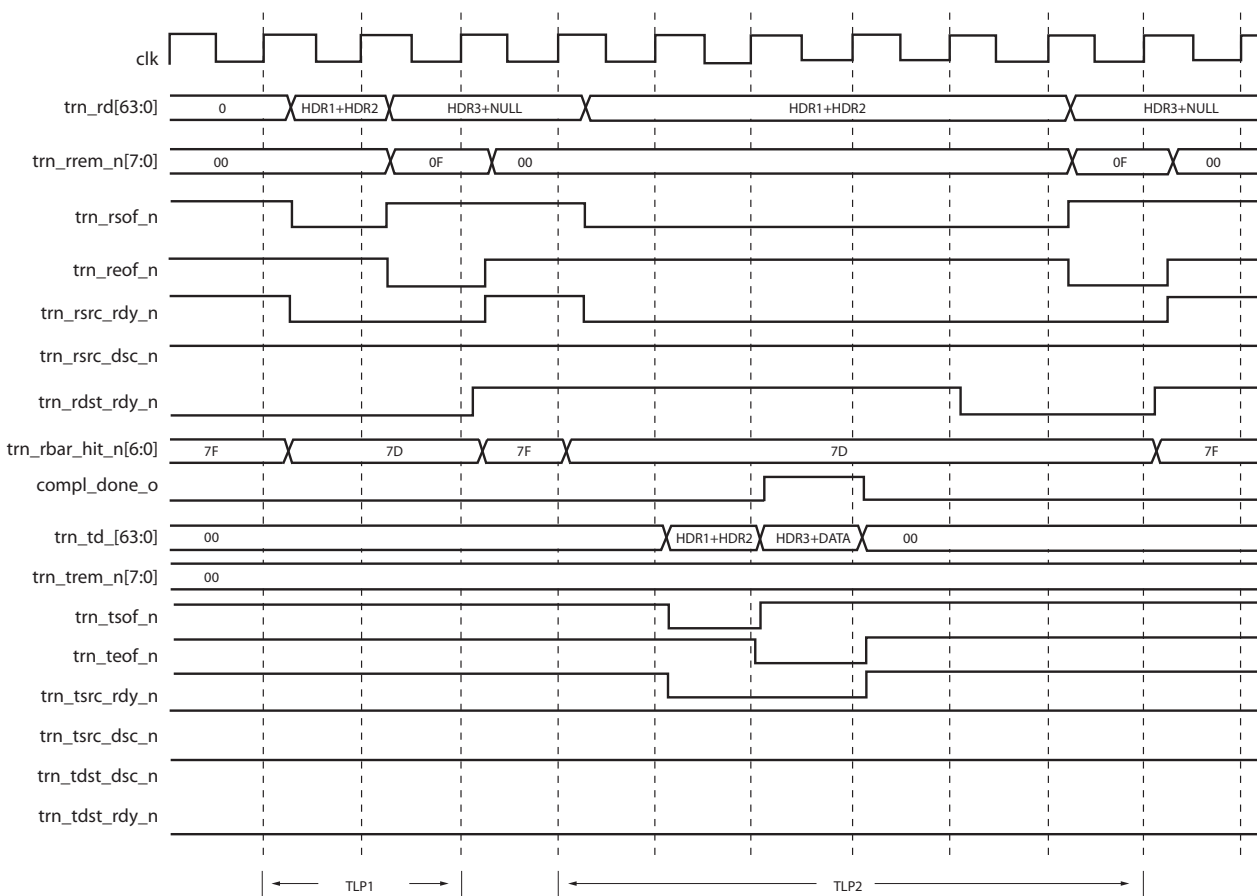


Figure B-8: Back-to-Back Read Transactions

## PIO Write Transaction

Figure B-9 depicts a back-to-back Memory Write to the PIO design. The next Write transaction is accepted only after `wr_busy_o` is deasserted by the memory access unit, indicating that data associated with the first request was successfully written to the memory aperture.

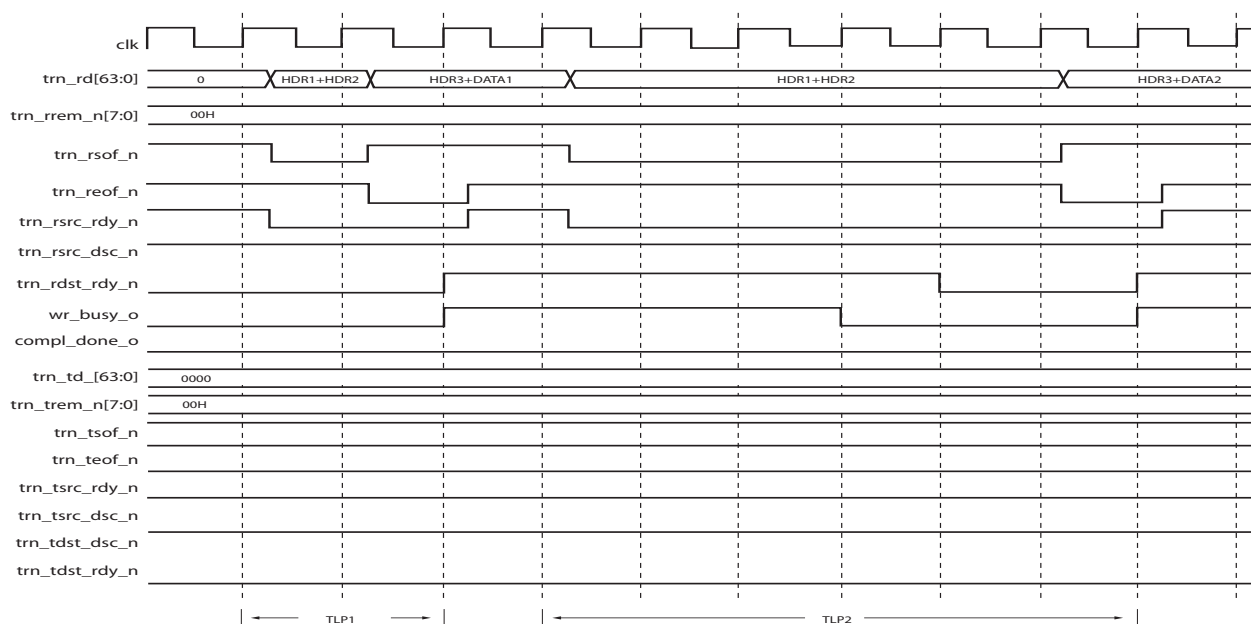


Figure B-9: Back-to-Back Write Transactions

## Device Utilization

Table B-8 shows the PIO design FPGA resource utilization.

Table B-8: PIO Design FPGA Resources

Resources	Utilization
LUTs	300
Flip-Flops	500
block RAMs	4

## Summary

The PIO design demonstrates the Endpoint for PCI Express core and its interface capabilities. In addition, it enables rapid bring-up and basic validation of end user endpoint add-in card FPGA hardware on PCI Express platforms. Users may leverage standard operating system utilities that enable generation of read and write transactions to the target space in the reference design.





## Downstream Port Model Test Bench

---

The Downstream Port Model test bench is a robust test bench environment that provides a test program interface that can be used with the provided PIO design or with your own design. The purpose of the Downstream Port Model is to provide a source mechanism for generating downstream PCI Express TLP traffic to stimulate the customer design, and a destination mechanism for receiving upstream PCI Express TLP traffic from the customer design in a simulation environment.

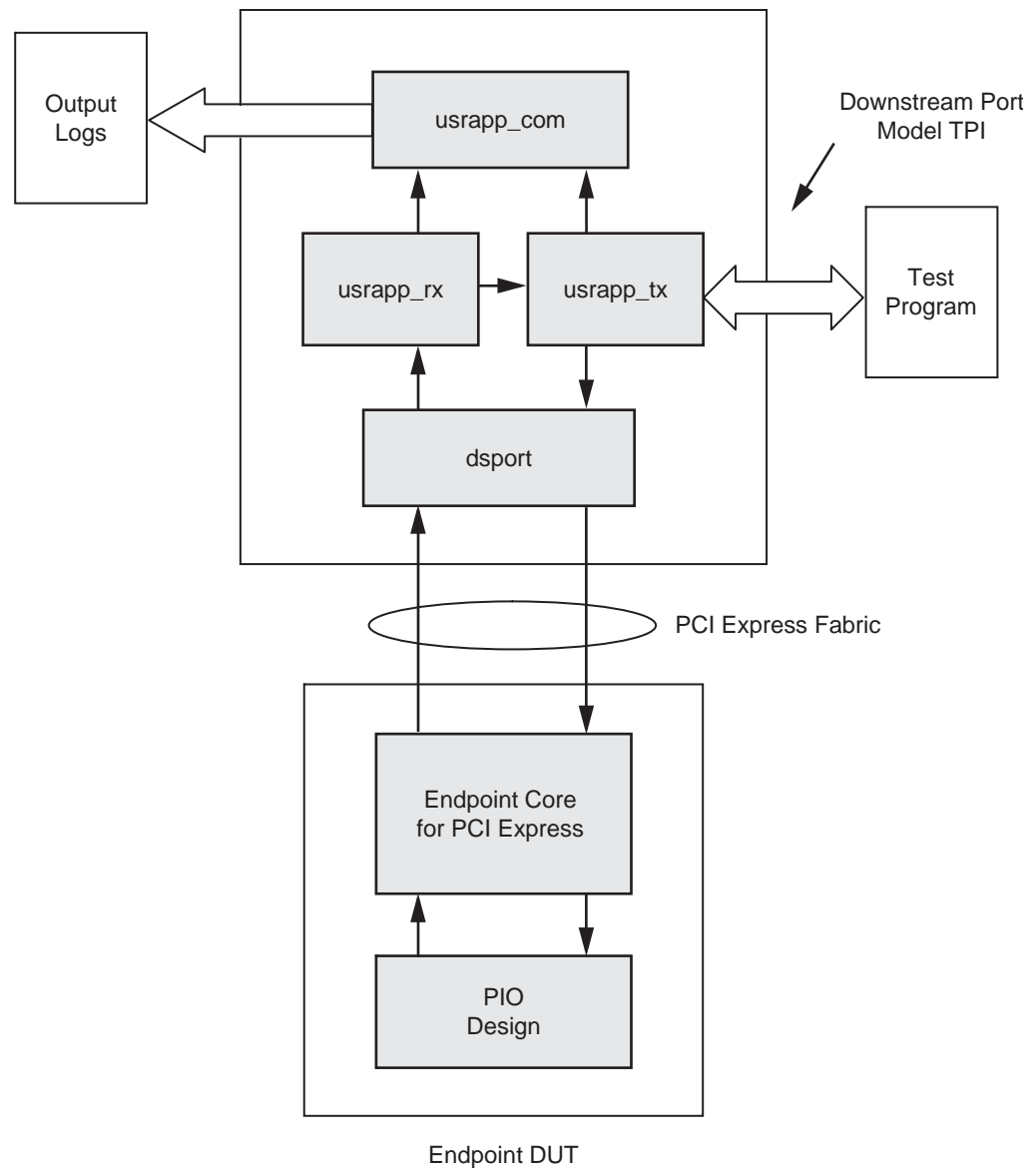
**Note:** The Downstream Port Model is shared by the Endpoint for PCI Express, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This appendix represents all the solutions generically using the name Endpoint for PCI Express.

Source code for the Downstream Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core's configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests are complete, allowing you to dedicate your efforts to verifying the correct functionality of the design rather than spending time developing a test bench infrastructure.

The Downstream Port Model consists of the following:

- Test Programming Interface (TPI), which allows the user to stimulate the Endpoint for PCI Express device
- Example tests that illustrate how to use the test program TPI
- Verilog source code for all Downstream Port Model components, which allow the user to customize the test bench

[Figure C-2](#) illustrates the Downstream Port Model coupled with the PIO design.



**Figure C-2: Downstream Port Model and Top-level Endpoint**

# Architecture

The Downstream Port Model consists of the following blocks as shown in [Figure C-2](#).

- dsport (downstream port)
- usrapp\_tx
- usrapp\_rx
- usrapp\_com

The `usrapp_tx` and `usrapp_rx` blocks interface with the `dsport` block for transmission and reception of TLPs to/from the Endpoint DUT. The Endpoint DUT consists of the Endpoint for PCI Express and the PIO design (displayed) or customer design.

The `usrapp_tx` block sends TLPs to the `dsport` block for transmission across the PCI Express link to the Endpoint DUT. In turn, the Endpoint DUT transmits TLPs across the PCI Express link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express fabric. Both the `usrapp_tx` and `usrapp_rx` utilize the `usrapp_com` block for shared functions, for example, TLP processing and log file outputting. Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the endpoint device's fabric interface. TLP responses from the endpoint device are received by the `usrapp_rx` block. Communication between the `usrapp_tx` and `usrapp_rx` blocks allow the `usrapp_tx` block to verify correct behavior and act accordingly when the `usrapp_rx` block has received TLPs from the endpoint device.

## Simulating the Design

Three simulation script files are provided with the model to facilitate simulation with VCS, Cadence IES, and ModelSim simulators:

- `simulate_vcs.sh`
- `simulate_ncsim.sh`
- `simulate_mti.do`

The example simulation script files are located in the following directory:

```
<project_dir>/<component_name>/simulation/functional
```

Instructions for simulating the PIO design using the Downstream Port Model are provided in [“Simulating the Example Design” in Chapter 4](#).

## Test Selection

The test model used for the Downstream Port Model lets you specify the name of the test to be run as a command line parameter to the simulator. For example, the `simulate_ncsim.sh` script file, used to start the Cadence IES simulator, explicitly specifies the test `sample_smoke_test0` to be run using the following command line syntax:

```
ncsim work.boardx01 +TESTNAME=sample_smoke_test0
```

You can change the test to be run by changing the value provided to `TESTNAME` defined in the test files `sample_tests1.v` and `pio_tests.v`. The same mechanism is used for VCS and ModelSim. [Table C-1](#) defines all the tests provided with Downstream Port Model.

**Table C-1: Tests Provided with Downstream Port Model**

Test Name	Description
<code>sample_smoke_test0</code>	Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.

Table C-1: Tests Provided with Downstream Port Model (Cont'd)

sample_smoke_test1	Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test form allows for confirming reception of any TLPs from the customer's design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.
pio_writeReadBack_test0	Transmits a 1 DWORD Write TLP followed by a 1 DWORD Read TLP to each of the example design's active BARs, and then waits for the Completion TLP and verifies that the write and read data match. The test will send the appropriate TLP to each BAR based on the BARs address type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).
pio_testByteEnables_test0	Issues four sequential Write TLPs enabling a unique byte enable for each Write TLP, and then issues a 1 DWORD Read TLP to confirm that the data was correctly written to the example design. The test will send the appropriate TLP to each BAR based on the BARs address-type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).
pio_memTestDataBus	Determines if the PIO design's FPGA block RAMs data bus interface is correctly connected by performing a 32-bit walking ones data test to the first available BAR in the example design.
pio_memTestAddrBus	Determines whether the PIO design's FPGA block RAM's address bus interface is correctly connected by performing a walking ones address test. This test should only be called after successful completion of pio_memTestDataBus.
pio_memTestDevice	Checks the integrity of each bit of the PIO design's FPGA block RAM by performing an increment/decrement test. This test should only be called after successful completion of pio_memTestAddrBus.

Table C-1: Tests Provided with Downstream Port Model (Cont'd)

pio_timeoutFailureExpected	Sends a Memory 32 Write TLP followed by Memory 32 Read TLP to an invalid address and waits for a Completion with data TLP. This test anticipates that waiting for the completion TLP times out and illustrates how the test programmer can gracefully handle this event.
pio_tlp_test0 (illustrative example only)	Issues a sequence of Read and Write TLPs to the example design's RX interface. Some of the TLPs, for example, burst writes, are not supported by the PIO design.

## Waveform Dumping

The Downstream Port Model provides a mechanism for outputting the simulation waveform to file by specifying the `+dump_all` command line parameter to the simulator.

For example, the script file `simulate_ncsim.sh`, is used to start the Cadence IES simulator can indicate to the Downstream Port Model that the waveform should be saved to file by using the following command line:

```
ncsim work.boardx01 +TESTNAME=sample_smoke_test0 +dump_all
```

This same mechanism is used for VCS and ModelSim. The dump file is in each simulator's native format, as defined in [Table C-2](#).

Table C-2: Simulator Dump File Format

Simulator	Dump File Format
VCS	.vpd
Cadence IES	.trn
Modelsim	.vcd

## Output Logging

When a test fails on the example or customer design, the test programmer debugs the offending test case. Typically, the test programmer inspects the wave file for the simulation and cross-reference this to the messages displayed on the standard output. Because this approach can be very time consuming, the Port Model offers an output logging mechanism to assist the tester with debugging failing test cases to speed the process.

The Downstream Port Model creates three output files (`tx.dat`, `rx.dat`, and `error.dat`) during each simulation run. Log files `rx.dat` and `tx.dat` each contain a detailed record of every TLP that was received and transmitted, respectively, by the Downstream Port Model. With an understanding of the expected TLP transmission during a specific test case, the test programmer may more easily isolate the failure.

The log file `error.dat` is used in conjunction with the expectation tasks. Test programs that utilize the expectation tasks will generate a general error message to standard output. Detailed information about the specific comparison failures that have occurred due to the expectation error is located within `error.dat`.

## Parallel Test Programs

There are two different classes of tests that are supported by the Downstream Port Model: sequential tests and parallel tests. Sequential tests are tests that exist within one process and behave similarly to sequential programs. The test depicted in “[Test Program: pio\\_writeReadBack\\_test0](#)” is an example of a sequential test. Parallel tests are tests involving more than one process thread. The test `sample_smoke_test1` is an example of a parallel test with two process threads.

Sequential tests are very useful when verifying behavior that have events with a known order. Parallel tests are very useful when verifying that a specific set of events have occurred, however the order of these events are not known.

A typical parallel test uses the form of one command thread and one or more expectation threads. These threads work together to verify a device's functionality. The role of the command thread is to create the necessary TLP transactions that cause the device to receive and generate TLPs. The role of the expectation threads is to verify the reception of an expected TLP. The Downstream Port Model TPI has a complete set of expectation tasks to be used in conjunction with parallel tests.

Note that because the example design is a target-only device, only Completion TLPs can be expected by parallel test programs while using the PIO design. However, the full library of expectation tasks can be used for expecting any TLP type when used in conjunction with the customer's design (which may include bus-mastering functionality).

## Test Description

The Downstream Port Model provides a Test Program Interface (TPI). The TPI provides the means to create tests by simply invoking a series of Verilog tasks. All Downstream Port Model tests should follow the same six steps described as follows:

1. Perform conditional comparison of a unique test name
2. Set up master timeout in case simulation hangs
3. Wait for Reset and link-up
4. Initialize the configuration space of the endpoint
5. Transmit and receive TLPs between the Downstream Port Model and the Endpoint for PCI Express device
6. Verify that the test succeeded

“[Test Program: pio\\_writeReadBack\\_test0](#)” displays the listing of a simple test program `pio_writeReadBack_test0`, written for use in conjunction with the PIO endpoint. This test program is located in the file `pio_tests.v`. As the test name implies, this test performs a one DWORD write operation to the PIO Design followed by a 1 DWORD read operation from the PIO Design, after which it compares the values to confirm that they are equal. The test is performed on the first location in each of the active Mem32 BARs of the PIO Design. For the default configuration, this test performs the write and read back to BAR1 and to the EROM space (BAR6). The following section outlines the steps performed by the test program.

- Line 1 of the sample program determines if the user has selected the test program `pio_writeReadBack_test1` when invoking the Verilog simulator.
- Line 4 of the sample program invokes the TPI call `TSK_SIMULATION_TIMEOUT` which sets the master timeout value to be long enough for the test to complete.

- Line 5 of the sample program invokes the TPI call `TSK_SYSTEM_INITIALIZATION`. This task will cause the test program to wait for the system reset to deassert as well as the endpoint's `trn_lnk_up_n` signal to assert. This is an indication that the endpoint is ready to be configured by the test program via the Downstream Port Model.
- Line 6 of the sample program uses the TPI call `TSK_BAR_INIT`. This task will perform a series of Type 0 Configuration Writes and Reads to the endpoint's PCI Configuration Space, determine the memory and IO requirements of the endpoint, and then program the endpoint's Base Address Registers so that it is ready to receive TLPs from the Downstream Port Model.
- Lines 7, 8, and 9 of the sample program work together to cycle through all the endpoint's BARs and determine whether they are enabled, and if so to determine their type, for example, Mem32, Mem64, or IO).

Note that all PIO tests provided with the Downstream Port Model are written in a form that does not assume that a specific BAR is enabled or is of a specific type (for example, Mem32, Mem64, IO). These tests perform on-the-fly BAR determination and execute TLP transactions dependent on BAR types (that is, Memory32 TLPs to Memory32 Space, IO TLPs to IO Space, and so forth). This means that if a user reconfigures the BARs of the Endpoint for PCI Express, the PIO continues to work because it dynamically explores and configures the BARs. Users are not required to follow the form used and can create tests that assume their own specific BAR configuration.

- Line 7 sets a counter to increment through all of the endpoint's BARs.
- Line 8 determines whether the BAR is enabled by checking the global array `BAR_INIT_P_BAR_ENABLED[]`. A non-zero value indicates that the corresponding BAR is enabled. If the BAR is not enabled then test program flow will move on to check the next BAR. The previous call to `TSK_BAR_INIT` performed the necessary configuration TLP communication to the endpoint device and filled in the appropriate values into the `BAR_INIT_P_BAR_ENABLED[]` array.
- Line 9 performs a case statement on the same global array `BAR_INIT_P_BAR_ENABLED[]`. If the array element is enabled (that is, non-zero), the element's value indicates the BAR type. A value of 1, 2, and 3 indicates IO, Memory 32, and Memory 64 spaces, respectively.

If the BAR type is either IO or Memory 64, then the test does not perform any TLP transactions. If the BAR type is Memory 32, program control continues to line 16 and starts transmitting Memory 32 TLPs.

- Lines 21-26 use the TPI call `TSK_TX_MEMORY_WRITE_32` and transmits a Memory 32 Write TLP with the payload DWORD '01020304' to the PIO endpoint.
- Lines 32-33 use the TPI calls `TSK_TX_MEMORY_READ_32` followed by `TSK_WAIT_FOR_READ_DATA` in order to transmit a Memory 32 Read TLP and then wait for the next Memory 32 Completion with Data TLP. In case the Downstream Port Model never receives the Completion with Data TLP, the TPI call `TSK_WAIT_FOR_READ_DATA` would locally timeout and display an error message.
- Line 34 compares the DWORD received from the Completion with Data TLP with the DWORD that was transmitted to the PIO endpoint and displays the appropriate success or failure message.

## Test Program: pio\_writeReadBack\_test0

```

1.  else if(testname == "pio_writeReadBack_test1"
2.  begin
3.  // This test performs a 32 bit write to a 32 bit Memory space and performs a read back
4.  TSK_SIMULATION_TIMEOUT(10050);
5.  TSK_SYSTEM_INITIALIZATION;
6.  TSK_BAR_INIT;
7.  for (ii = 0; ii <= 6; ii = ii + 1) begin
8.  if (BAR_INIT_P_BAR_ENABLED[ii] > 2'b00) // bar is enabled
9.  case(BAR_INIT_P_BAR_ENABLED[ii])
10. 2'b01 : // IO SPACE
11.  begin
12.  $display("[%t] : NOTHING: to IO 32 Space BAR %x", $realtime, ii);
13.  end
14. 2'b10 : // MEM 32 SPACE
15.  begin
16.  $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x",
17.  $realtime, ii);
18.  //-----
19.  // Event : Memory Write 32 bit TLP
20.  //-----
21.  DATA_STORE[0] = 8'h04;
22.  DATA_STORE[1] = 8'h03;
23.  DATA_STORE[2] = 8'h02;
24.  DATA_STORE[3] = 8'h01;
25.  P_READ_DATA = 32'hffff_ffff; // make sure P_READ_DATA has known initial value
26.  TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0] , 4'hF,
4'hF, 1'b0);
27.  TSK_TX_CLK_EAT(10);
28.  DEFAULT_TAG = DEFAULT_TAG + 1;
29.  //-----
30.  // Event : Memory Read 32 bit TLP
31.  //-----
32.  TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0], 4'hF,
4'hF);
33.  TSK_WAIT_FOR_READ_DATA;
34.  if (P_READ_DATA != {DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0] })
35.  begin
36.  $display("[%t] : Test FAILED --- Data Error Mismatch, Write Data %x != Read Data %x",
$realtime,{DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0]}, P_READ_DATA);
37.  end
38.  else
39.  begin
40.  $display("[%t] : Test PASSED --- Write Data: %x successfully received", $realtime,
P_READ_DATA);
41.  end
42.  TSK_TX_CLK_EAT(10);
43.  DEFAULT_TAG = DEFAULT_TAG + 1;
44.  end
45. 2'b11 : // MEM 64 SPACE
46.  begin
47.  $display("[%t] : NOTHING: to Memory 64 Space BAR %x", $realtime, ii);
48.  end
49.  default : $display("Error case in usrapp_tx\n");
50.  endcase
51.  end
52.  $display("[%t] : Finished transmission of PCI-Express TLPs", $realtime);
53.  $finish;
54.  end

```



## Expanding the Downstream Port Model

The Downstream Port Model was created to work with the PIO design, and for this reason is tailored to make specific checks and warnings based on the limitations of the PIO design. These checks and warnings are enabled by default when the Downstream Port Model is generated by the CORE Generator. However, these limitations can easily be disabled so that they do not affect the customer's design.

Because the PIO design was created to support at most one IO BAR, one Mem64 BAR, and two Mem32 BARs (one of which must be the EROM space), the Downstream Port Model by default makes a check during device configuration that verifies that the PCI Express Core has been configured to meet this requirement. A violation of this check will cause a warning message to be displayed as well as for the offending BAR to be gracefully disabled in the test bench. This check can be disabled by setting the `pio_check_design` variable to zero in the `pci_exp_usrapp_tx.v` file.

### Downstream Port Model TPI Task List

The Downstream Port Model TPI tasks include, which are defined in the tables that follow:

- “Test Setup Tasks”
- “TLP Tasks”
- “BAR Initialization Tasks”
- “Example PIO Design Tasks”

Table C-3: Test Setup Tasks

Name	Input(s)		Description
TSK_SYSTEM_INITIALIZATION	None		Waits for transaction interface reset and link-up between the downstream port and the end point DUT.  This task must be invoked prior to PCI Express core initialization.
TSK_USR_DATA_SETUP_SEQ	None		Initializes global 4096 byte DATA_STORE array entries to sequential values from zero to 4095.
TSK_TX_CLK_EAT	clock count	31:30	Waits clock_count transaction interface clocks.
TSK_SIMULATION_TIMEOUT	timeout	31:0	Sets master simulation timeout value in units of transaction interface clocks. This task should be used to ensure that all DUT tests complete.

Table C-4: TLP Tasks

Name	Input(s)	Description
TSK_TX_TYPE0_CONFIGURATION_READ	tag_ 7:0 reg_addr_ 11:0 first_dw_be_ 3:0	Waits for transaction interface reset and link-up between the downstream port and the end point DUT.  This task must be invoked prior to PCI Express core initialization.
TSK_TX_TYPE1_CONFIGURATION_READ	tag_ 7:0 reg_addr_ 11:0 first_dw_be_ 3:0	Sends a Type 1 PCI Express Config Read TLP from downstream port to reg_addr_ of endpoint DUT with tag_ and first_dw_be_ inputs.  CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_TYPE0_CONFIGURATION_WRITE	tag_ 7:0 reg_addr_ 11:0 reg_data_ 31:0 first_dw_be_ 3:0	Sends a Type 0 PCI Express Config Write TLP from downstream port to reg_addr_ of endpoint DUT with tag_ and first_dw_be_ inputs.  Cpl returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_TYPE1_CONFIGURATION_WRITE	tag_ 7:0 reg_addr_ 11:0 reg_data_ 31:0 first_dw_be_ 3:0	Sends a Type 1 PCI Express Config Write TLP from downstream port to reg_addr_ of endpoint DUT with tag_ and first_dw_be_ inputs.  Cpl returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_MEMORY_READ_32	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 31:0 last_dw_be_ 3:0 first_dw_be_ 3:0	Sends a PCI Express Memory Read TLP from downstream port to 32 bit memory address addr_ of endpoint DUT.  CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_MEMORY_READ_64	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 63:0 last_dw_be_ 3:0 first_dw_be_ 3:0	Sends a PCI Express Memory Read TLP from downstream port to 64 bit memory address addr_ of endpoint DUT.  CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.

Table C-4: TLP Tasks (Cont'd)

Name	Input(s)	Description
TSK_TX_MEMORY_WRITE_32	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 31:0 last_dw_be_ 3:0 first_dw_be_ 3:0 ep_ –	Sends a PCI Express Memory Write TLP from downstream port to 32 bit memory address addr_ of endpoint DUT.  CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.  The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_MEMORY_WRITE_64	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 63:0 last_dw_be_ 3:0 first_dw_be_ 3:0 ep_ –	Sends a PCI Express Memory Write TLP from downstream port to 64 bit memory address addr_ of endpoint DUT.  CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.  The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_COMPLETION	tag_ 7:0 tc_ 2:0 len_ 9:0 comp_status 2:0 –	Sends a PCI Express Completion TLP from downstream port to endpoint DUT using global COMPLETE_ID_CFG as completion ID.
TSK_TX_COMPLETION_DATA	tag_ 7:0 tc_ 2:0 len_ 9:0 byte_count 11:0 lower_addr 6:0 comp_status 2:0 ep_ –	Sends a PCI Express Completion with Data TLP from downstream port to endpoint DUT using global COMPLETE_ID_CFG as completion ID.  The global DATA_STORE byte array is used to pass completion data to task.
TSK_TX_MESSAGE	tag_ 7:0 tc_ 2:0 len_ 9:0 data 63:0 message_rtg 2:0 message_code 7:0	Sends a PCI Express Message TLP from downstream port to endpoint DUT.  Completion returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.

Table C-4: TLP Tasks (Cont'd)

Name	Input(s)	Description
TSK_TX_MESSAGE_DATA	tag_ 7:0 tc_ 2:0 len_ 9:0 data 63:0 message_rtg 2:0 message_code 7:0	<p>Sends a PCI Express Message with Data TLP from downstream port to endpoint DUT.</p> <p>The global DATA_STORE byte array is used to pass message data to task.</p> <p>Completion returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_IO_READ	tag_ 7:0 addr_ 31:0 first_dw_be_ 3:0	<p>Sends a PCI Express IO Read TLP from downstream port to IO address addr_[31:2] of endpoint DUT.</p> <p>CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_IO_WRITE	tag_ 7:0 addr_ 31:0 first_dw_be_ 3:0 data 31:0	<p>Sends a PCI Express IO Write TLP from downstream port to IO address addr_[31:2] of endpoint DUT.</p> <p>CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_BAR_READ	bar_index 2:0 byte_offset 31:0 tag_ 7:0 tc_ 2:0	<p>Sends a PCI Express 1 DWORD Memory 32, Memory 64, or IO Read TLP from the downstream port to the target address corresponding to offset byte_offset from BAR bar_index of the endpoint DUT.</p> <p>This task sends the appropriate Read TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.</p> <p>CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>

Table C-4: TLP Tasks (Cont'd)

Name	Input(s)		Description
TSK_TX_BAR_WRITE	bar_index byte_offset tag_ tc_ data_	2:0 31:0 7:0 2:0 31:0	<p>Sends a PCI Express 1 DWORD Memory 32, Memory 64, or IO Write TLP from the downstream port to the target address corresponding to offset byte_offset from BAR bar_index of the endpoint DUT.</p> <p>This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.</p>
TSK_WAIT_FOR_READ_DATA	None		<p>Waits for the next completion with data TLP that was sent by the endpoint DUT. On successful completion, the first DWORD of data from the CplD will be stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions.</p> <p>By default this task will locally time out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local time out returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.</p>

Table C-5: BAR Initialization Tasks

Name	Input(s)	Description
TSK_BAR_INIT	None	<p>Performs a standard sequence of Base Address Register initialization tasks to the Endpoint for PCI Express device using the PCI Express fabric. Performs a scan of the Endpoint's PCI BAR range requirements, performs the necessary memory and IO space mapping calculations, and finally programs the Endpoint so that it is ready to be accessed.</p> <p>On completion, the user test program may begin memory and IO transactions to the device. This function displays to standard output a memory/IO table that details how the Endpoint has been initialized. This task also initializes global variables within the Downstream Port Model that are available for test program usage. This task should only be called after TSK_SYSTEM_INITIALIZATION.</p>
TSK_BAR_SCAN	None	<p>Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express fabric in order to determine the memory and IO requirements for the Endpoint for PCI Express.</p> <p>The task stores this information in the global array BAR_INIT_P_BAR_RANGE[]. This task should only be called after TSK_SYSTEM_INITIALIZATION.</p>
TSK_BUILD_PCIE_MAP	None	<p>Performs memory/IO mapping algorithm and allocates Memory 32, Memory 64, and IO space based on the Endpoint for PCI Express requirements.</p> <p>This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of TSK_BAR_SCAN.</p>
TSK_DISPLAY_PCIE_MAP	None	<p>Displays the memory mapping information of the Endpoint's PCI Express Core's PCI Base Address Registers. For each BAR, the BAR value, the BAR range, and BAR type is given. This task should only be called after completion of TSK_BUILD_PCIE_MAP.</p>

Table C-6: Example PIO Design Tasks

Name	Input(s)		Description
TSK_TX_READBACK_CONFIG	None		<p>Performs a sequence of PCI Type 0 Configuration Reads to the Endpoint for PCI Express device's Base Address Registers, PCI Command Register, and PCIe Device Control Register using the PCI Express fabric.</p> <p>This task should only be called after TSK_SYSTEM_INITIALIZATION.</p>
TSK_MEM_TEST_DATA_BUS	bar_index	2:0	<p>Tests whether the PIO design FPGA block RAM data bus interface is correctly connected by performing a 32-bit walking ones data test to the IO or memory address pointed to by the input bar_index.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.</p>
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	<p>Tests whether the PIO design FPGA block RAM address bus interface is accurately connected by performing a walking ones address test starting at the IO or memory address pointed to by the input bar_index.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.</p>
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	<p>Tests the integrity of each bit of the PIO design FPGA block RAM by performing an increment/decrement test on all bits starting at the block RAM pointed to by the input bar_index with the range specified by input nBytes.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.</p>

Table C-7: Expectation Tasks

Name	Input(s)	Output	Description
TSK_EXPECT_CPLD	traffic_class 2:0 td - ep - attr 1:0 length 9:0 completer_id 15:0 completer_status 2:0 bcm - byte_count 11:0 requester_id 15:0 tag 7:0 address_low 6:0	expect status	Waits for a Completion with Data TLP that matches traffic_class, td, ep, attr, length, and payload. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_CPL	traffic_class 2:0 td - ep - attr 1:0 completer_id 15:0 completer_status 2:0 bcm - byte_count 11:0 requester_id 15:0 tag 7:0 address_low 6:0	Expect status	Waits for a Completion without Data TLP that matches traffic_class, td, ep, attr, and length. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_MEMRD	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 29:0	Expect status	Waits for a 32-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.



Table C-7: Expectation Tasks (Cont'd)

Name	Input(s)	Output	Description
TSK_EXPECT_MEMRD64	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 61:0	Expect status	Waits for a 64-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise.  Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 29:0	Expect status	Waits for a 32 bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise.  Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR64	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 61:0	Expect status	Waits for a 64 bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise.  Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_IOWR	td - ep - requester_id 15:0 tag 7:0 first_dw_be 3:0 address 31:0 data 31:0	Expect status	Waits for an IO Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise.  Note that this task can only be used in conjunction with Bus Master designs.



## Additional Design Considerations

### Package Constraints

This section discusses design considerations specific to the Endpoint PIPE for PCI Express targeting Spartan-3, Spartan-3E, and Spartan-3A devices. [Table D-1](#) defines the smallest supported device and interface combinations for the core.

**Table D-1: Supported Device and Interface Combinations**

Smallest Supported Device/Part Number	Data Bus Width/Speed	Wrapper File
XC3S1000 FG676-4	Width: 32-bit Port Speed: 62.5 MHz	xilinx_pci_exp_endpoint.v
XC3S500E CP132-4	Width: 32-bit Port Speed: 62.5 MHz	xilinx_pci_exp_endpoint.v
XC3S700A FG400-4	Width: 32-bit Port Speed: 62.5 MHz	xilinx_pci_exp_endpoint.v

### User Constraints Files

The UCF contains various constraints required for the Endpoint PIPE for PCIe core. The UCF is specific to the target device and must always be used while processing a design.

#### Spartan-3

```
<project_dir>/<component_name>/example_design/xilinx_1_lane_epipe_ep_XC3S1000-FG676-4.ucf
```

#### Spartan-3E

```
<project_dir>/<component_name>/example_design/xilinx_1_lane_epipe_ep_XC3S500E-CP132-4.ucf
```

#### Spartan-3A

```
<project_dir>/<component_name>/example_design/xilinx_1_lane_epipe_ep-XC3S700A-FG484-4.ucf
```

```
<project_dir>/<component_name>/example_design/xilinx_1_lane_epipe_ep-XC3S1400A-FG484-4.ucf
```

## Wrapper File Usage

The wrapper contains an instance of the Endpoint PIPE for PCI Express core. When starting a new design, modify this wrapper to include all I/O elements and modules.

```
<project_dir>/<component_name>/example_design/xilinx_pci_exp_1_lane_  
epipe_ep.v
```