RapidIO MegaCore Function

User Guide

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com

MegaCore Version: 7.1
Document Date: May 2007

UG-MC_RIOPHY-2.5

Contents

Chapter 3. Physical Layer—Serial Specifications

Chapter 4. Variations with Physical, Transport, and Logical Layers

About This User Guide

Revision History

The table below displays the revision history for the chapters in this user guide.

Chapter	Date	Version	Changes Made
1	May 2007	7.1	• Added Arria™GX device support • Updated the performance information.
	December 2006	7.0	• Added Cyclone® III device support.
	December 2006	6.1	• Added Doorbell Message support. • Added Stratix® III support.
	April 2006	3.1.0	• Updated the release information and device family support tables. • Updated the performance information.
	January 2006	3.0.1	• Updated the release information and device family support tables. • Updated the features. • Updated the performance information.
	October 2005	3.0.0	• Updated the release information and device family support tables. • Removed all references to the AIRbus interface. • Updated the performance information.
	April 2005	2.2.2	• Updated the release information and device family support tables.
	January 2005	2.2.1	• Updated the release information.
	December 2004	2.2.0	• Updated the release information. • Updated the features. • Updated the performance information.
	March 2004	2.1.0	• Updated the release information and device family support tables. • Moved the Configuration Options table to Chapters 3 and 4. • Moved Interfaces and Protocols descriptions to Chapters 3 and 4. • Added OpenCore® Plus description. • Updated the performance information.
2	May 2007	7.1	• Updated MegaWizard® Plug-In Manager Design Flow and added the SOPC Builder Design Flow

Chapter	Date	Version	Changes Made
2	December 2006	7.0	• No change.
	December 2006	6.1	• Revised the MegaWizard Getting Started section
	April 2006	3.1.0	• Updated format.
	January 2006	3.0.1	• Updated the system requirements. • Updated the walkthrough instructions and screen captures. • Updated the demonstration testbench description.
	October 2005	3.0.0	• Updated the system requirements. • Updated the walkthrough instructions. • Removed all references to the AIRbus interface.
	April 2005	2.2.2	• Updated the system requirements.
	January 2005	2.2.1	• No change.
	December 2004	2.2.0	• Updated the system requirements. • Added IP CD installation instructions. • Updated the walkthrough instructions. • Added the 4× serial parameter. • Added the receive priority retry threshold parameters. • Removed the receive buffer control interface parameter. • Removed the Set Constraints section.
	March 2004	2.1.0	• Added Linux instructions. • Moved the configuration parameters description to Chapters 3 and 4. • Updated the walkthrough instructions. • Added IP functional simulation models information.
3	May 2007	7.1	• Chapter reorganization, which now contains only a serial interface that uses only the Physical layer. • Removed parallel interface information.

Chapter	Date	Version	Changes Made
3	December 2006	7.0	• No change.
	December 2006	6.1	• Revised the section on clock inputs.
	April 2006	3.1.0	• Updated the clock domains section and modified Figure 3–2. • Updated parameters table and a few signals.
	January 2006	3.0.1	• Added description of Avalon-MM interface and signals. • Added description of external transceiver interface and signals. • Updated the clock information. • Updated the Parameters table. • Added the MegaCore Verification section.
	October 2005	3.0.0	• Removed all references to the AIRbus interface.
	April 2005	2.2.2	• Updated the Error Handling section. • Added the Forced Compensation Sequence Insertion section. • Added more description to the `atxovf` signal.
	January 2005	2.2.1	• No change.
	December 2004	2.2.0	• Updated the Functional Description section, to add the 4× serial feature and description. • Removed the receive buffer control interface. • Updated the description of sublayers 1 and 3. • Updated the Parameters, Signals, and some Registers tables.
	March 2004	2.1.0	• Added OpenCore Plus time-out behavior description. • Added Interfaces and Protocols descriptions. • Added Parameters description (table). • Updated, renamed, and deleted some signals. • Updated the register set.
4	May 2007	7.1	• Reorganized chapter to discuss a variation that uses the Physical, Transport, and Logical layers. • Improved description of the error detection and management, Maintenance module, software interface, and Avalon™-ST Pass-Through port. • Added IO slave interface example.

Chapter	Date	Version	Changes Made
4	December 2006	7.0	• No change.
	December 2006	6.1	• Revised the section on clock inputs.
	April 2006	3.1.0	• No change.
	January 2006	3.0.1	• Added description of Avalon-MM interface and signals. • Updated the Parameters table. • Updated the MegaCore Verification section.
	October 2005	3.0.0	• Removed all references to the AIRbus interface. • Removed the MegaCore Verification section.
	April 2005	2.2.2	• Updated the Error Handling section. • Added more description to the `atxovf` signal.
	January 2005	2.2.1	• No change.
	December 2004	2.2.0	• Updated the features, Figure 4-1, and Figure 4-2. • Removed the receive buffer control interface. • Updated the description of sublayers 2 and 3. • Updated the Parameters, Signals, and some Registers tables. • Added the MegaCore Verification section.
	March 2004	2.1.0	• Removed 16-bit port width feature, related description and figures. • Added OpenCore Plus time-out behavior description. • Added Interfaces and Protocols descriptions. • Added Parameters description (table). • Updated, renamed, and deleted some signals. • Updated the register set.
5	May 2007	7.1	• New chapter contains SOPC Builder Design Example. Former chapter 5 is now chapter 4.
	December 2006	7.0	• No change.
	December 2006	6.1	• Added the doorbell feature information. • Updated the MegaCore and RapidIO version numbers. • Revised the section on clock inputs.
	April 2006	3.1.0	• Updated content throughout this chapter.
	January 2006	3.0.1	• Updated content throughout this chapter.
	October 2005	3.0.0	• Added this new Transport & Input/Output Logical layer Specifications chapter.

Chapter	Date	Version	Changes Made
A	May 2007	7.1	• Previous Appendix A has been removed. New Appendix A, Initialization Sequence, was Appendix C in previous releases.
	December 2006	7.0	• No change.
	April 2006	3.1.0	• No change.
	January 2006	3.0.1	• No change.
	October 2005	3.0.0	• No change.
	December 2004	2.2.0	• Added Stratix® II references.
B	May 2007	7.1	• Former Appendix B has been removed from this release. • New Appendix B is the XGMII interface.
	December 2006	7.0	• No change.
	April 2006	3.1.0	• No change.
	January 2006	3.0.1	• No change.
	October 2005	3.0.0	• No change.
	December 2004	2.2.0	• Added Stratix II references. • Removed Stratix timing information.
	March 2004	2.1.0	• Removed all references to APEX II device family, including static timing information.
C	May 2007	7.1	• Former Appendix C is now appendix A.
	December 2006	7.0	• No change.
	April 2006	3.1.0	• No change.
	January 2006	3.0.1	• Added this Initialization Sequence appendix.
	October 2005	3.0.0	• Removed the Compliance appendix.
	December 2004	2.2.0	• Removed the packet retry transmission order compliance issue from Table C3. • Added the port link time-out control issue to Table C2.
	March 2004	2.1.0	• Added this Compliance appendix.

How to Contact Altera

For the most up-to-date information about Altera® products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/
Technical training services	custrain@altera.com
Product literature	www.altera.com/literature
Product literature services	literature@altera.com
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (for example, the VHDL keyword `BEGIN`), as well as logic function names (for example, `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

1. About This MegaCore Function

Release Information

Table 1–1 provides information about this release of the RapidIO® MegaCore® function.

Table 1–1. RapidIO Release Information

Item	Description
Version	7.1
Release Date	May 2007
Ordering Code	IP-RIOPHY
Product ID	0095
Vendor ID	6AF7

New in RapidIO MegaCore Function Version 7.1

- Support for Arria™ GX device family
- SOPC Builder support
- Avalon™ Streaming (Avalon-ST)
- Improved error detection and recovery
- Multicast event reception

Device Family Support

MegaCore functions provide either full or preliminary support for target Altera® device families:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 1–2 shows the level of support offered by the RapidIO MegaCore function to each Altera device family.

Table 1–2. Device Family Support

Device Family	Support
Arria GX	Preliminary
Cyclone® II	Full
Cyclone III	Preliminary
HardCopy® II	Full
Stratix® II	Full
Stratix II GX	Full
Stratix III	Preliminary
Stratix GX	Full
Other device families	No support

Performance and Size

Table 1–3 lists the resources and expected performance for a selection of variations using the Serial Physical layer with the I/O Master and Slave Avalon-MM and Maintenance Master modules of the Logical layer enabled. These results were obtained using the Quartus® II software version 7.1 for the following devices:

- Cyclone II (EP2C35F484C6)
- Cyclone III (EP3C55F780C6)
- Stratix GX (EP1SGX40DF1020C5)

Table 1–3. Serial RapidIO Utilization

Device	Parameters				LEs	Memory	
	Lane	Baud Rate (Gbaud)	Atlantic Port Width (bits)	Buffer Size (Kbytes)		M4K	M512
Cyclone II	1x	3.125 with external SERDES	32	8 TX, 4 RX	12,119	87	–
	4x	1.250 with external SERDES	64	8 TX, 4 RX	15,962	105	–
Cyclone III	1x	3.125 with external SERDES	32	8 TX, 4 RX	12,909	78	–
	4x	1.250 with external SERDES	64	8 TX, 4 RX	15,823	86	–
Stratix GX	1x	3.125	32	8 TX, 4 RX	13,321	79	21
	4x	1.250	64	8 TX, 4 RX	16,103	73	17

Table 1–4 lists the resources for additional selections of variations using the Serial Physical layer with the I/O Master and Slave Avalon-MM and Maintenance Master modules of the Logical layer enabled. These results were obtained using the Quartus II software version 7.1 for the following devices:

- Arria GX (EP1AGX60DF780C6)
- Stratix II (EP2S30F672C3)
- Stratix II GX (EP2SGX30D780C3)
- Stratix III (EP3SE260F1508)

Table 1–4. Serial RapidIO Utilization

Device	Parameters				Combinational ALUTs	Logic Registers	Memory	
	Mode	Baud Rate (Gbaud)	Atlantic Port Width (bits)	Buffer Size (Kbytes)			M4K or M9K *(1)*	M512
Arria GX	1x	2.5	32	8 TX, 4 RX	7,764	11,506	84	13
	4x	2.5	64	8 TX, 4 RX	15,783	15,783	82	13
Stratix II	1x	3.125 with external SERDES	32	8 TX, 4 RX	7,757	11,511	85	14
	4x	1.250 with external SERDES	64	8 TX, 4 RX	9,152	14,507	80	12
Stratix II GX	1x	3.125	32	8 TX, 4 RX	7,696	8,433	84	13
	4x	3.125	64	8 TX, 4 RX	10,343	11,539	81	14
Stratix III	1x	3.125 with external SERDES	32	8 TX, 4 RX	8,862	12,741	50	–
	4x	3.125 with external SERDES	64	8 TX, 4 RX	10,654	16,202	53	–

Notes for Table 1–4
(1) M9K for Stratix III, M4K for all others.

Table 1–5 shows the recommended device family speed grades for the supported link widths and internal clock frequencies. In all cases it is recommended that the Quartus II Analysis & Synthesis Optimization Technique be set to **Speed**. See the "Compile the Design" section for information on how to apply this setting.

Table 1–5. Recommended Device Family and Speed Grades

Device Family	Mode	1x			4x		
	Rate	1.25 Gbaud	2.5 Gbaud	3.125 Gbaud	1.25 Gbaud	2.5 Gbaud	3.125 Gbaud
	Fmax	31.25MHz	62.50MHz	78.125MHz	62.5MHz	125MHz	156.25MHz
Arria GX *(3)*		-6	-6	–	-6	-6	–
Stratix II, Stratix II GX		-3, -4, -5	-3, -4, -5	-3, -4, -5	-3, -4, -5	-3, -4	-3*(1)*
Stratix III		-2, -3, -4	-2, -3, -4	-2, -3, -4	-2, -3, -4	-2, -3, -4	-2, -3
Stratix GX		-5, -6, -7	-5, -6, -7	-5, -6, -7	-5	-5*(2)*	–
Cyclone II, Cyclone III		-6, -7, -8	-6, -7, -8	-6, -7	-6, -7, -8	–	–

Notes for Table 1–5:

(1) 4x 3.125 Gbaud is possible in a -4 Stratix II and Stratix II GX only with the smallest Rx and Tx buffer sizes.

(2) 4x 2.5 Gbps may be possible in Stratix GX with the use of multiple seeds when using the Quartus II Design Space Explorer.

(3) Only the -6 speed grade is available for the Arria GX device family.

Features

- Physical layer features
  - 1×/4× Serial
    - Arria GX support, including 1x and 4x up to 2.5 Gbaud
    - Stratix II GX and Stratix GX support, including 1× and 4× up to 3.125 Gbaud
    - Cyclone II, Cyclone III, Stratix II, Stratix III, and HardCopy II support with an XGMII-like interface to an external high-speed full-duplex, serializer/deserializer (SERDES) transceiver
  - Packet buffering for receiver and transmitter, flow control, error detection, packet assembly and delineation
- Transport layer features
  - Supports multiple Logical layer modules
    - A round robin outgoing scheduler chooses packets to transmit from various Logical layer modules
  - Supports 8-bit device IDs
- Capability Registers (CARs) and Command and Status Registers (CSRs)
  - 32-bit Avalon® Memory-Mapped (Avalon-MM) interface bus slave supporting local single-word access
- Maintenance Master Logical layer module
  - 32-bit Avalon-MM bus master supporting the reception of single-word access
- Maintenance Slave Logical layer module
  - 32-bit Avalon-MM bus slave supporting single-word access
  - 32-bit wide Avalon-MM bus slave interface to access remote CARs and CSRs
- Input/Output Avalon-MM Master Logical layer module
  - Avalon-MM bus masters support burst transfers of up to 256 bytes (64 32-bit words or 32 64-bit words)
- Input/Output Avalon-MM Slave Logical layer module
  - Avalon-MM bus slaves support burst transfers of up to 256 bytes (64 32-bit words or 32 64-bit words)
- Avalon streaming interface allows custom implementation of Message module and SAR (Segmentation and Reassembly) functions
- Doorbell Module
  - 32-bit Avalon-MM slave, supporting single-word access
  - Supports 16 outstanding doorbell packets with timeout mechanism
- Compliant with all applicable standards, including:
  - RapidIO Trade Association, *RapidIO Interconnect Specification*, Revision 1.3, February 2005.
  - Altera Corporation, *FS-13: Atlantic™ Interface*.
  - Altera Corporation, *Avalon Memory-Mapped Interface Specification*.
  - Altera Corporation, *Avalon Streaming Interface Specification*
- Easy-to-use MegaWizard interface

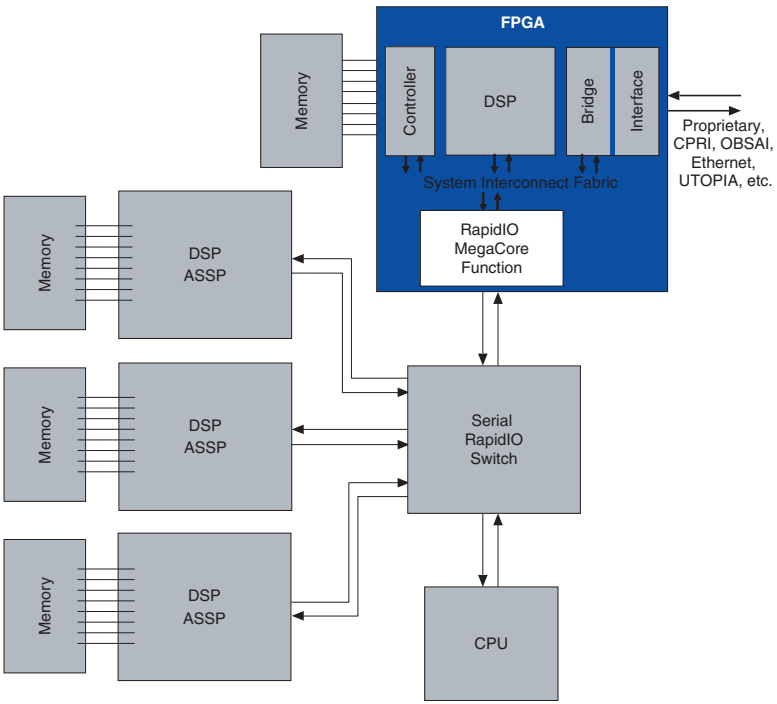
- SOPC Builder support
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore Plus evaluation

General Description

The RapidIO interconnect—an open standard developed by the RapidIO Trade Association—is a high-performance packet-switched interconnect technology designed to pass data and control information between microprocessors, digital signal processors (DSPs), communications and network processors, system memories, and peripheral devices.

The RapidIO MegaCore function targets high-performance, multicomputing, high-bandwidth I/O applications. Figure 1–1 on page 1–6 shows an example system implementation.

Figure 1–1. Typical Application



MegaCore Function Design Flows

Optimized for Altera devices, the RapidIO MegaCore function can be customized to support a wide variety of applications. You can use either the MegaWizard® Plug-In Manager or SOPC Builder interfaces to customize the MegaCore function.

MegaWizard Plug-In Manager Design Flow

You can use the MegaWizard Plug-In Manager interface in the Quartus II software to parameterize and manually instantiate a custom MegaCore function variation. The wizards guide you as you set parameter values and select optional ports. This flow is best for manual instantiation of the MegaCore function in a higher-level design.

SOPC Builder Design Flow

The SOPC Builder flow enables integration of a RapidIO endpoint into an SOPC Builder system with an automatically generated system interconnect. The SOPC Builder design flow automatically connects user instantiated components with system interconnect fabric, eliminating the requirement to design low-level interfaces and significantly reducing design time.

OpenCore Plus Evaluation

With the Altera free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP™ megafunction) within your system using the Quartus® II software and Altera supported VHDL and Verilog HDL simulators
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include MegaCore functions
- Program a device and verify your design in hardware

You only need to purchase a license for the MegaCore function when you are completely satisfied with its functionality and performance, and want to take your design to production.

For more information on OpenCore Plus hardware evaluation using the RapidIO MegaCore function, see "OpenCore Plus Time-Out Behavior" on page 4–53, and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

2. Getting Started

Design Flow

To evaluate the RapidIO MegaCore function and use the OpenCore Plus feature include these steps in your design flow:

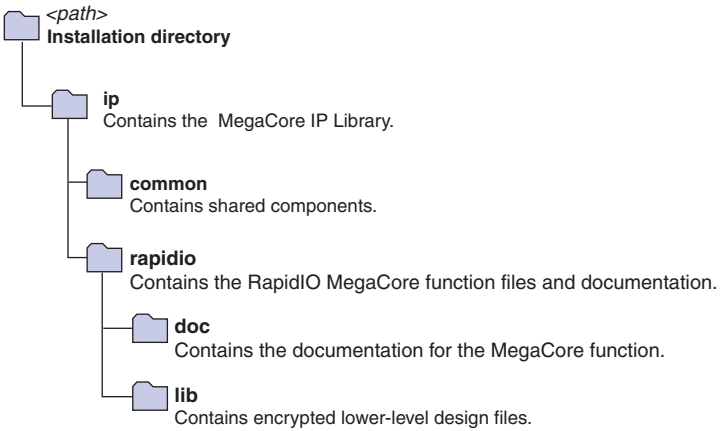
1. Obtain and install the RapidIO MegaCore function.

   The RapidIO MegaCore Function is part of the MegaCore IP Library, which is distributed with the Quartus II software and downloadable from the Altera website, **www.altera.com**.

For system requirements and installation instructions, refer to *Quartus II Installation & Licensing for Windows* or *Quartus II Installation & Licensing for UNIX & Linux Workstations* on the Altera website.

Figure 2–1 shows the directory structure for the RapidIO MegaCore function, where <*path*> is the installation directory. The default installation directory on Windows is **c:\altera**<*version number*>; on UNIX and Solaris it is **/opt/altera/**<*version number*>.

Figure 2–1. Directory Structure



2. Decide whether to use the MegaWizard® Plug-In Manager or SOPC Builder design flow.

- If using the MegaWizard Plug-In Manager flow, create a custom variation of the RapidIO MegaCore function.
- If using SOPC Builder flow, instantiate and parameterize the RapidIO SOPC Builder component.

3. Implement the rest of your design using SOPC Builder or the design entry method of your choice.
4. Use the IP functional simulation model to verify the operation of your design.

   For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in Volume 3 of the *Quartus II Handbook*.

5. Use the Quartus II software to compile your design.

   You can also generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

6. Purchase a license for the RapidIO MegaCore function.

After you have purchased a license for the RapidIO MegaCore function, follow these additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera device(s) on your board.
3. Program the Altera device(s) with the completed design.

MegaWizard Plug-In Manager Design Flow Walkthrough

This walkthrough explains how to create a RapidIO MegaCore function using the MegaWizard Plug-In Manager and the Quartus II software. When you finish generating a custom variation of the RapidIO MegaCore function, you can incorporate it into your overall project.

You also can generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

This walkthrough consists of the following steps:

- Create a New Quartus II Project
- Launch the MegaWizard Plug-In Manager
- Parameterize
- Set Up Simulation and Generate the Function
- Simulate the Design
- Compile the Design

Create a New Quartus II Project

You start the design process by creating a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity. To create a new project follow these steps:

1. Choose **Programs > Altera > Quartus II** <*version*> from the Windows Start menu to run the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.

2. On the FIle Menu, click **New Project Wizard**.

3. Click **Next** in the **New Project Wizard Introduction** (the introduction does not display if you turned it off previously).

4. In the **Name, Top-Level Entity** page, enter the following information:

   a. Specify the working directory for your project. For example, this walkthrough uses the **c:\altera\projects\rio_project** directory.

   b. Specify the name of the project. This walkthrough uses **rio_example** for the project name.

The Quartus II software automatically specifies a top-level design entity that has the same name as the project. Do not change this name.

5. Click **Next** to close this page and display the **Add Files** page.

   When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

6. If you installed the MegaCore IP Library in a different directory from where you installed the Quartus II software, you must add the user libraries manually:

   a. Click **User Libraries**.

   b. Type <*path*>**\ip** into the **Library name** box, where <*path*> is the directory in which you installed the RapidIO MegaCore function.

   c. Click **Add** to add the path to the Quartus II project.

   d. Click **OK** to save the library path in the project.

7. Click **Next** to close this page and display the **Family & Device Settings** page.

8. On the **Family & Device Settings** page, select the target device family in the **Family** list.

9. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

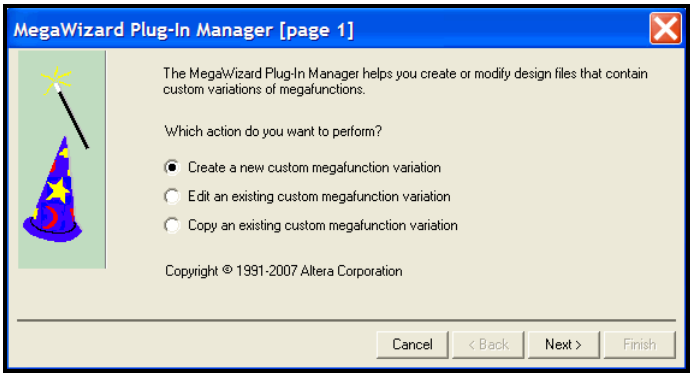
Launch the MegaWizard Plug-In Manager

To launch the MegaWizard Plug-In Manager in the Quartus II software, follow these steps:

1. Start the MegaWizard Plug-In Manager. On the Tools menu, select **MegaWizard Plug-In Manager**. The **MegaWizard Plug-In Manager** dialog box displays (see Figure 2–2).

   Refer to Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

Figure 2–2. MegaWizard Plug-In Manager



2. Click **Create a new create a new custom megafunction variation** and then click **Next**.

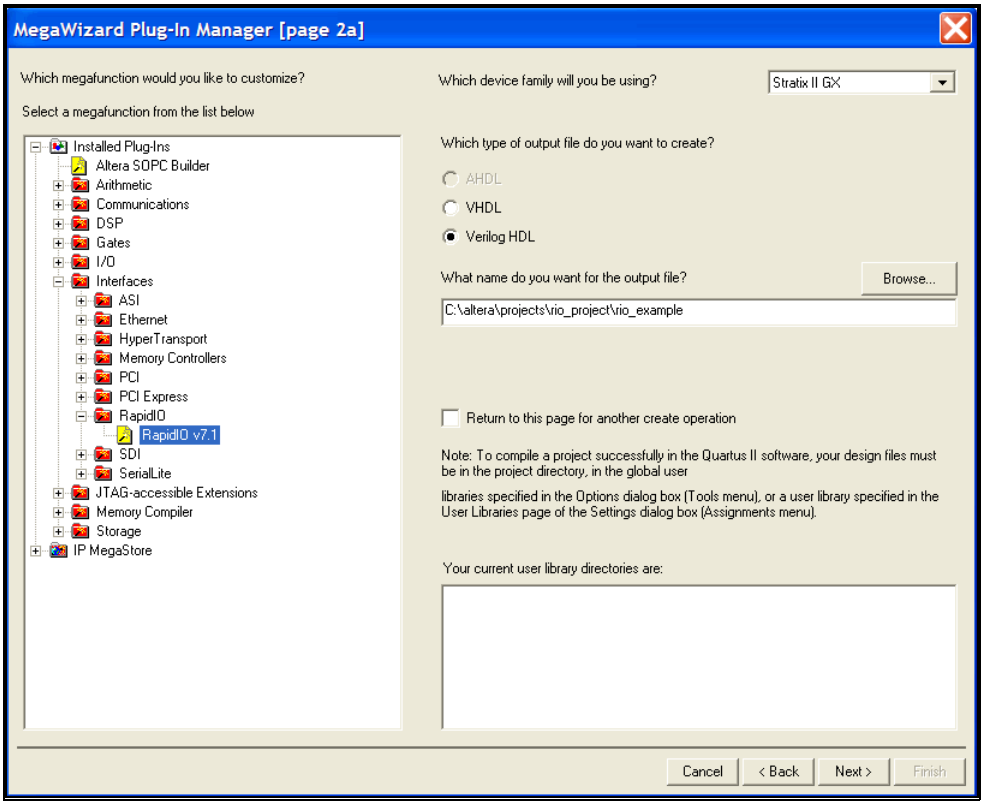
3. Expand the **Installed Plug-Ins** and **Interfaces** lists, click **RapidIO**, and then click **RapidIO v7.1**. See Figure 2–3.

4. In the **Which device family will you be using?** box, select the device family you want to use for this MegaCore function variation. For this example, select **Stratix II GX**.

5. In the **Which type of output file do you want to create**? box, click the output file type for your design; the MegaWizard interface supports VHDL and Verilog HDL. In this example, click **Verilog HDL**.

6. In the **What name do you want for the output file?** box, The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files <*project path*>\<*variation name*>. For this example, specify **rio_example** for the output file name. Figure 2–3 shows the MegaWizard interface after you have made these settings.

Figure 2–3. Select the MegaCore Function



7. Click **Next** to display the **Parameter Settings** page for the RapidIO MegaCore function (see Figure 2–4).

You can change the page that the MegaWizard interface displays by clicking **Next** or **Back** at the bottom of the dialog box. You can move directly to a specific page by clicking on the page name: **Physical Layer**, **Transport and Maintenance**, **I/O and Doorbell**, or **Capability Registers**.

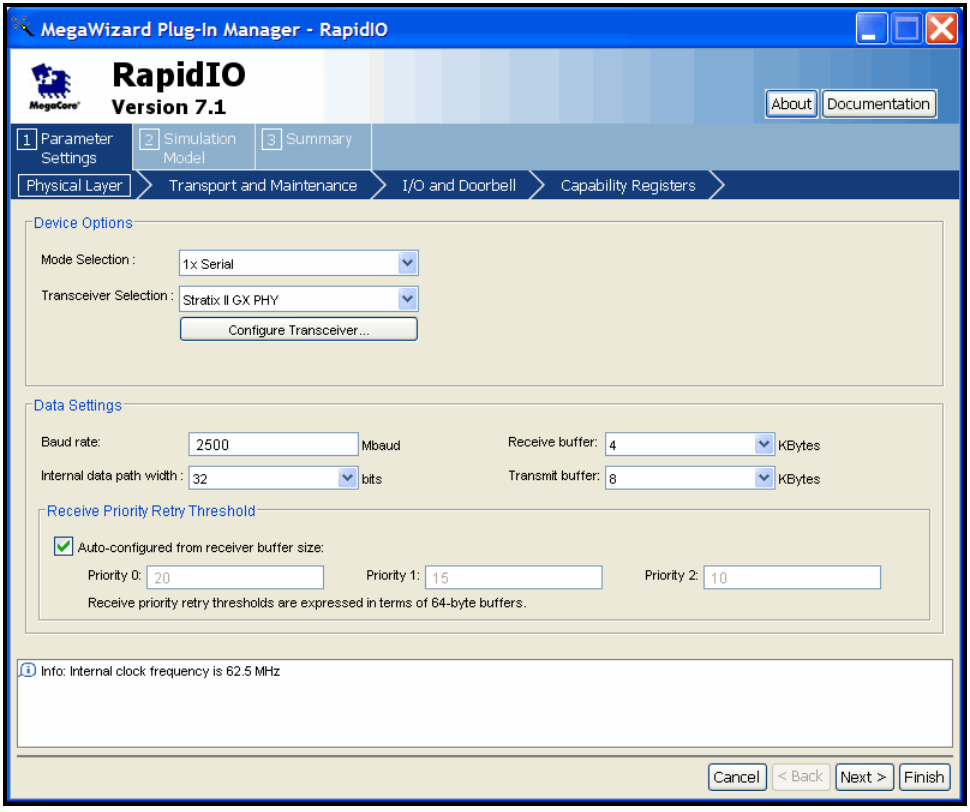
Parameterize

This section describes the parameters available for the RapidIO MegaCore function, and the benefits of different options.

Not all parameters are supported by, or are relevant for every MegaCore function variation.

To parameterize your MegaCore function, follow these steps (see Figure 2–4):

Figure 2–4. Physical Layer Parameters



1. On the **Physical Layer** page, under **Device Options**, select options for **Mode Selection** and **Transceiver Selection**.

Table 2–1 shows the baud rates supported by the serial RapidIO MegaCore function for each device family.

Table 2–1. Serial RapidIO Device Support

Device Family	Lane	Serial 1x			Serial 4x		
	Baud Rate	1.250	2.500	3.125	1.250	2.500	3.125
Arria GX		✓	✓	(1)	✓	✓	(1)
HardCopy II with External Transceiver		✓	✓	✓	✓	✓	✓
Stratix II GX		✓	✓	✓	✓	✓	✓
Stratix III or Stratix II with External Transceiver		✓	✓	✓	✓	✓	✓
Stratix GX		✓	✓	✓	✓	✓	(1)
Cyclone III or Cyclone II with External Transceiver		✓	✓	✓	✓	(2)	(1)

Note:
(1) This device does not support 3.125 Gbaud.
(2) This device does not support 2.500 Gbaud 4x lanes

2. From the **Transceiver Selection** list, select the transceiver.

   Select the appropriate PHY option for the Arria GX, Stratix GX, or Stratix II GX device.

   For devices without transceivers, selecting **External Transceiver** allows your design to use the serial RapidIO MegaCore function with any supported device.

3. Click the **Configure Transceiver...** button to configure the transceiver for Arria GX, Stratix GX, or the Stratix II GX PHY transceiver.

   The **Configure Transceiver** dialog box (see Figure 2–5) is displayed. This walkthrough uses alt2gxb as the example.

   The Arria GX, Stratix GX, or Stratix II GX PHY transceiver requires that you configure the altgxb or alt2gxb megafunction.

   a. Under the **Transmitter Functionality** in the **Voltage Output Differential (VOD)** panel for the **Specify VOD control setting**, select a value of **0**, **1**, **2**, **3**, **4**, or **5**.

b. Under the **Transmitter Functionality** for the **Specify a pre-emphasis control setting**, select a value of **0**,**1**, **2**, **3**, **4 or 5**.

For Stratix II GX devices, the pre-emphasis control values supported are **0**,**1**,**2**,**3**,**4**, and **5**. For **0**, pre-emphasis is off. For **1**, the pre-emphasis will be the maximum negative value. For **2**, pre-emphasis will be the medium negative value. The value **3** is a special value in which only the first posttap is set (set to the maximum), while the other taps are off. A value of **4** yields a medium positive value, while **5** sets the pre-emphasis values to the maximum positive supported values.

c. Under the **Transmitter Functionality** in the **Bandwidth mode** box, select **low** or **high**.

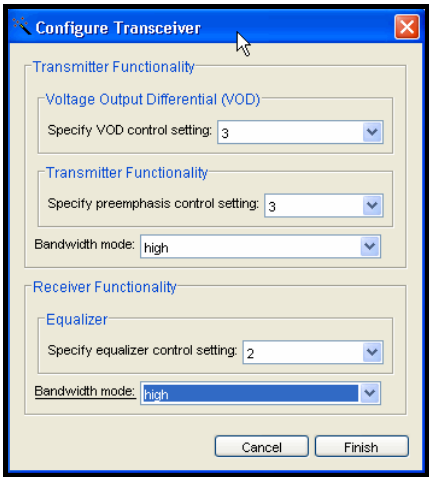
d. Under the **Receiver Functionality** in the **Equalizer Specify equalizer control setting**, the receiver, choose an equalizer control setting value of **0**,**1**, **2**, **3**, **4**, or **5**.

For Stratix II GX devices the supported equalizer control setting values are **0**,**1**, **2**, **3**, and 4. These values correspond to **0** for the lowest/off, **1** for a value between medium and lowest, 2 is medium, a **3** is between medium and high, and a **4** is high.

e. Under the **Receiver Functionality** in the **Bandwidth mode** box, select **low** or **high**.

f. Click **Finish** to return to the previous page.

Figure 2–5. Configure Transceiver for altgxb and alt2gxb Megafunctions



4. Under **Data Settings, set the following options**:

   a. For baud rate, specify a baud rate (Mbaud).

   b. For **Internal data path width**, select a datapath width. For this example, select **32**.

   c. For the **Receive buffer** and **Transmit buffer**, select a buffer size (KBytes).

5. For the **Receive Priority Retry Threshold**, set preferences. Turn on or off **Auto-configured from receiver buffer size**. If you turn off the check box, specify a value for each of the priority boxes:

   a. Enter a value for **Receive Priority 0 Retry Threshold**.

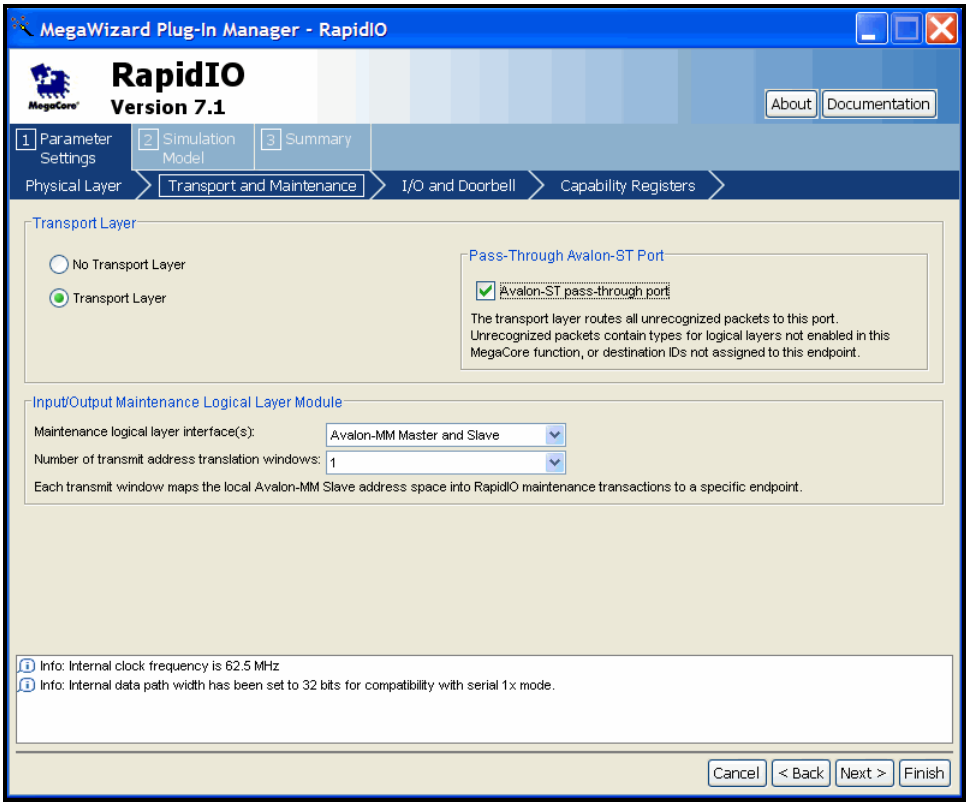
   b. Enter a value for **Receive Priority 1 Retry Threshold**.

   c. Enter a value for **Receive Priority 2 Retry Threshold**.

   Receiver priority retry thresholds are expressed in terms of 64-byte buffers. Each maximum size packet requires five buffers.

6. Click **Next** to display the **Transport and Maintenance** page (see Figure 2–6).

Figure 2–6. Transport Layer & Maintenance Parameters



7. Under Transport layer, click **No Transport Layer** or **Transport Layer.**

   a. If click **No transport Layer**, you are finished parameterizing your custom Physical layer-only variation. Click **Finish** and go to "Set Up Simulation and Generate the Function" on page 2–15.

   b. If you click **Transport Layer**, turn on or off the **Avalon-ST pass-through port**.

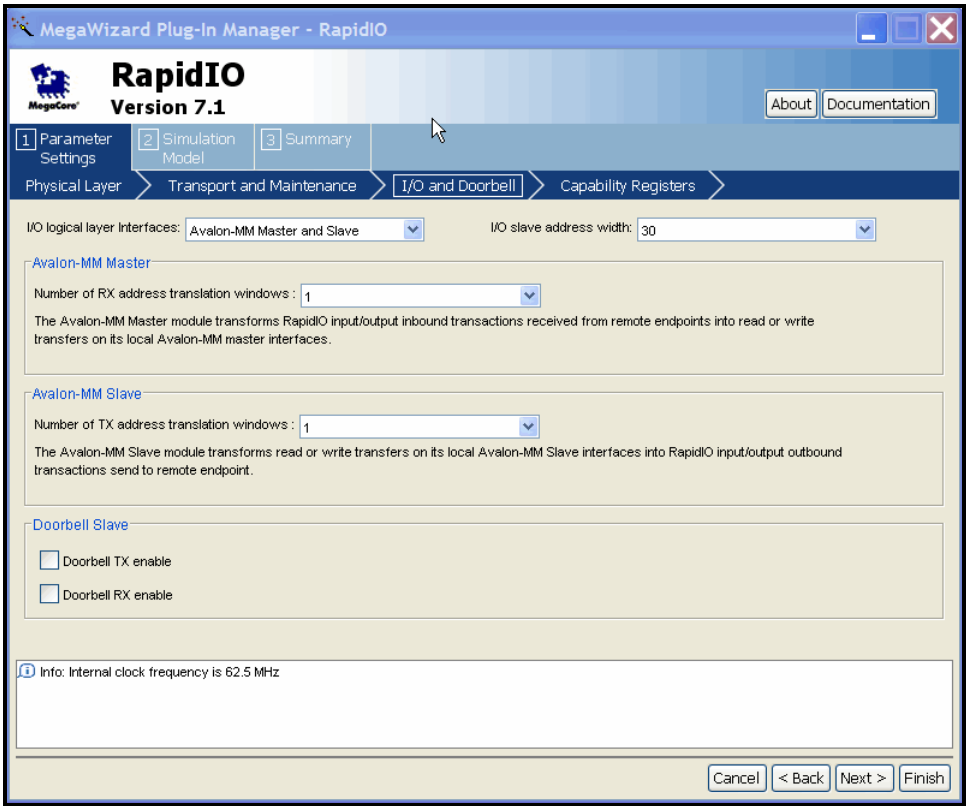
   The Transport layer routes all unrecognized packets to the Avalon-ST pass-through port. Unrecognized packets contain `ftypes` for Logical layers not enabled in this MegaCore function, or destination IDs not assigned to this endpoint.

8. Under the **Input/Output** Maintenance **Logical Layer Module,** select the M**aintenance Logical Layer interfaces**.

If you select an **Avalon-MM Master and Slave** interface or just an **Avalon-MM Slave** interface, also select the **Number of transmit address translation windows**. You can select from **1** to **16** windows.

9. Click **Next** to display the **I/O and Doorbell** page; see Figure 2–7.

Figure 2–7. I/O Logical Doorbell Layer Parameters



10. Select one of the **I/O logical layer interface(s)**.

11. If your selection in Step 10 included an Avalon-MM Slave interface, select an **I/O slave address width**. The value can be an integer in the range of **25** through **32**.

The **I/O slave address width** is set to **30** by default. However, to avoid over-allocating Avalon-MM memory space, Altera recommends setting this value to the lowest value for your system.

12. If your selection in Step 10 included an Avalon-MM Master interface, select the **Number of RX address translation windows**. You can choose from **1** to **16** windows.

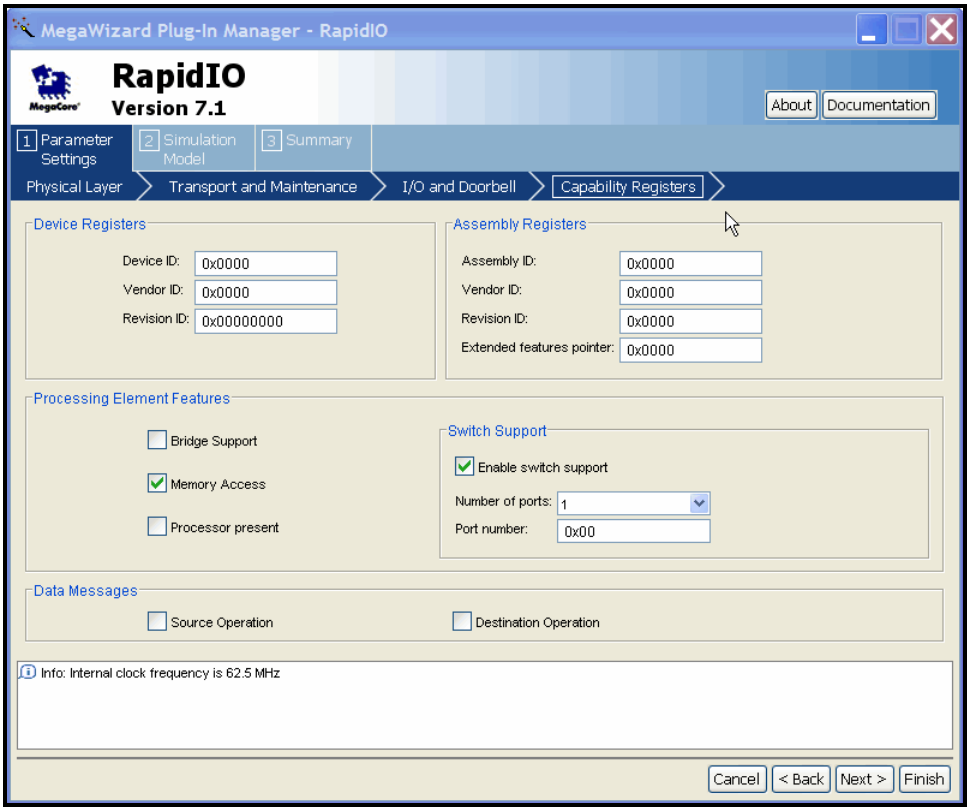
13. If in Step 10 you selected an Avalon-MM Slave interface, select the **Number of TX address translation windows**. You can choose from **1** to **16** windows.

14. Under **Doorbell Slave**, click **Doorbell TX enable** and/or **Doorbell RX enable** to turn on transmitting and/or receiving doorbell messages. This option determines if the rapidIO MegaCore function can transmit or receive doorbell messages from the user's custom software interface.

15. Click **Next** to display the **Capability Registers** page; see Figure 2–8.

If you want to set up a simulation model, do not click **Finish** now because clicking **Finish** now bypasses simulation setup.

Figure 2–8. Capability Registers



16. Under **Device Registers,** specify the device registers values: **Device ID**, **Vendor ID**, and **Revision ID**.

17. Under the **Assembly Registers values:** specify the **Assembly ID**, **Vendor ID**, **Revision ID**, and **Extended features pointer**.

18. Under Processing Element Features, turn on or off the **Bridge support**, **Memory access**, and **Processor present**.

19. Under **Switch Support**, turn on or off the **Enable switch support**.

    a. If you turn on **Enable switch support**, select the **Number of ports** and enter a **Port number**. You can select from **1** to **16** ports.

20. For **Data Messages**, turn on or turn off the **Source Operation** and/or **Destination Operation** messages.

    Turning on or off these options allows host user logic to implement message passing that can be used by the RapidIO MegaCore function and that reports user logic errors through standard error management.

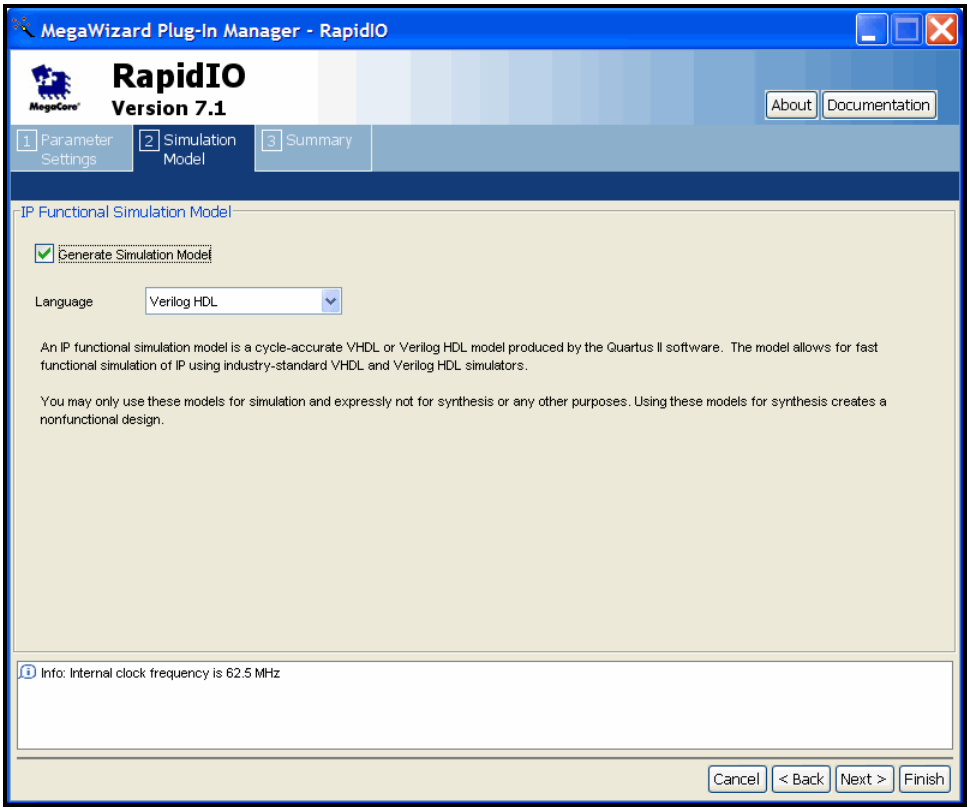
21. Click the **Simulation Model** tab.

Set Up Simulation and Generate the Function

1. To set up a simulation model, turn on **Generate Simulation Model** (see Figure 2–9).

2. Select the **Language** for the simulation model. You can select **VHDL** or **Verilog HDL**.

Figure 2–9. Simulation Model



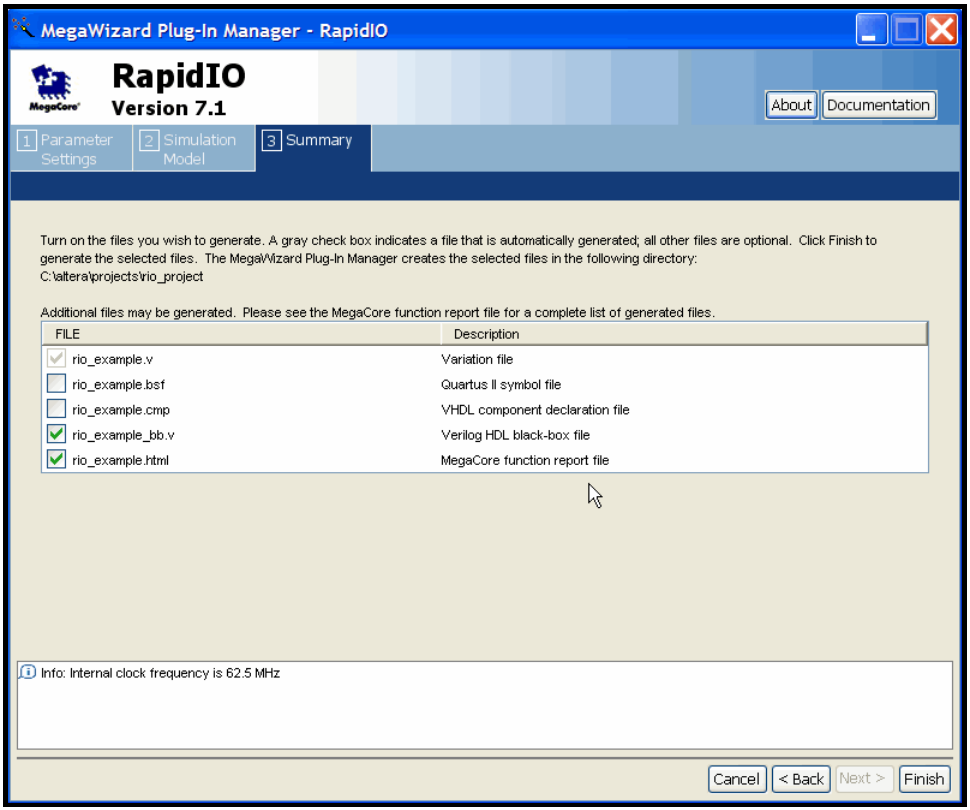
3. Click **Next** to continue to the **Summary** page. On this page you can you specify the files that are generated (see Figure 2–10). After clicking on the files to be generated, click **Finish** to generate the function and simulation model files.

The IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model file produced by the Quartus II software. This model supports fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

CAUTION

Only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

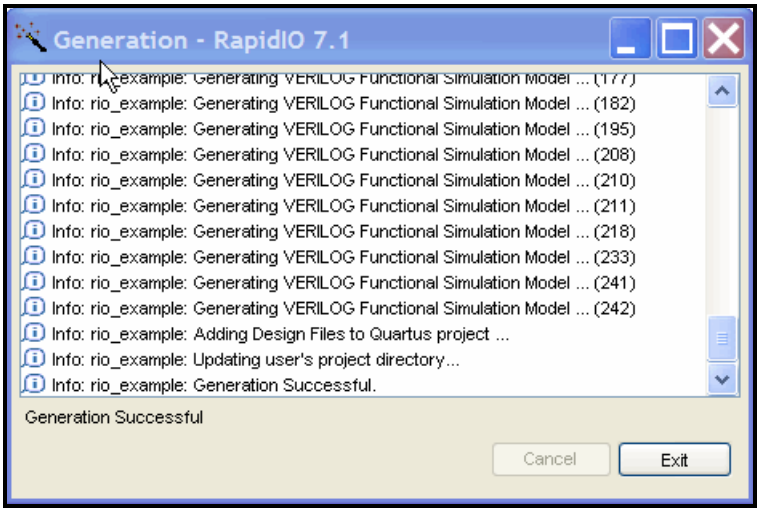
Figure 2–10. Summary Page



4. To generate the specified files and close the MegaWizard Plug-In Manager, click **Finish**.

The Generation panel displays file generation status. When all files have been generated, the Generation panel returns a Generation Successful status message as Figure 2–11 illustrates. Click **Exit** to close the panel. The generation phase can take several minutes to complete. A generation report, written to the project directory and named <*variation name*>**.html**, lists the files and ports generated.

Figure 2–11. Generated Files Message



You can view the generated files by opening the file <*variation_name*>**.html**.

Table 2–2 describes the generated files and other files that may be in your project directory. The names and types of files specified in the report vary based on whether you created your design with VHDL or Verilog HDL.

Table 2–2. Project Files Note (1) (Part 1 of 4)

Filename *(2)*	Description
<*variation_name*>.bsf	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation_name*>.html	The MegaCore function report file.
<*variation_name*>.ppf	This XML file describes the MegaCore pin attributes to the Quartus II Pin Planner. MegaCore pin attributes include pin direction, location, I/O standard assignments, and drive strength.
<*variation_name*>.v	A MegaCore function variation file which defines a Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<*variation_name*>.vo	Verilog HDL IP functional simulation model.
<*variation_name*>_avalon_bfm_master.v	Verilog Avalon-MM master bus functional model for use in simulation with the demo testbench.

Table 2–2. Project Files Note (1) **(Part 2 of 4)**

Filename (2)	Description
<*variation_name*>_avalon_bfm_slave.v	Verilog Avalon-MM slave bus functional model for use in simulation with the demo testbench.
<*variation_name*>_bb.v	Verilog HDL black-box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
<*variation_name*>_concentrator.v	Encrypted Verilog RTL for the Avalon-MM Concentrator module. This is used to access all the registers from a common Avalon interface. Required for Place and Route.
<*variation_name*>_concentrator_sister.v	Encrypted Verilog RTL for the Avalon-MM Concentrator module. Required for running the demo testbench.
<*variation_name*>_constraints.tcl	Tcl script to apply required constraints and assignments. Required for place and route. Run this script before you compile the MegaCore function.
<*variation_name*>_drbell.v	Encrypted Verilog RTL for the Doorbell Logical layer module. Required for place and route.
<*variation_name*>_drbell_sister.v	Encrypted Verilog RTL for the Doorbell Logical layer module of the sister RapidIO MegaCore. Required for running the demo testbench.
<*variation_name*>_hookup.iv	Verilog include file instantiating the RapidIO MegaCore function and utilities in the demo testbench. Used for simulation.
<*variation_name*>_hutil.iv	Verilog include file containing various utilities used for simulation with the demo testbench.
<*variation_name*>_io_master.v	Encrypted Verilog RTL for Input/Output Logical layer Avalon-MM master module. Required for Place and Route.
<*variation_name*>_io_master_sister.v	Encrypted Verilog RTL for Input/Output Logical layer Avalon-MM master module of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_io_slave.v	Encrypted Verilog RTL for Input/Output Logical layer Avalon-MM slave module. Required for Place and Route.
<*variation_name*>_io_slave_sister.v	Encrypted Verilog RTL for Input/Output Logical layer Avalon-MM slave module of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_maintenance.v	Encrypted Verilog RTL for Maintenance Logical layer module. Required for Place and Route.
<*variation_name*>_maintenance_sister.v	Encrypted Verilog RTL for Maintenance Logical layer module of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_phy_mnt.v	Encrypted Verilog RTL for the physical layer's CAR and CSR access module. Required for Place and Route.

Table 2–2. Project Files Note (1) **(Part 3 of 4)**

Filename (2)	Description
<*variation_name*>_phy_mnt_sister.v	Encrypted Verilog RTL for the physical layer's CAR and CSR access module of the sister RapidIO MegaCore function. Required for running the demo testbench
<*variation_name*>_reg_mnt.v	Encrypted Verilog RTL for main CAR and CSR access module, including error management registers. Required for Place and Route.
<*variation_name*>_reg_mnt_sister.v	Encrypted Verilog RTL for main CAR and CSR access module of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_rio.v	Verilog RTL of the top level module, in clear text. Required for Place and Route.
<*variation_name*>>_rio_sister.v	Verilog RTL of the top level module of the sister RapidIO MegaCore function, in clear text. Required for running the demo testbench.
<*variation_name*>_sister_rio.vo	Verilog HDL IP functional simulation model.
<*variation_name*>_riophy_dcore.ocp	OpenCore Plus description file. Required to generate time-limited device programming files for OpenCore Plus hardware evaluation of the RapidIO MegaCore function without a license.
<*variation_name*>_riophy_dcore.v	Encrypted Verilog RTL for the main portion of the physical layer of the RapidIO MegaCore function. Required for Place and Route.
<*variation_name*>_riophy_dcore_sister.v	Encrypted Verilog RTL for the main portion of the physical layer of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_riophy_gxb.v	Clear text Verilog instantiation of the altgxb or alt2gxb or alt2gxb SerDes MegaFunction. Required for place and route.
<*variation_name*>_riophy_sister_gxb.v	Clear text Verilog instantiation of the altgxb or alt2gxb or alt2gxb SerDes MegaFunction of the sister RapidIO MegaCore. Required for running the demo testbench.
<*variation_name*>_riophy_reset.v	Clear text Verilog reset controller module for serial RapidIO MegaCore. Required for place and route.
<*variation_name*>_riophy_reset_sister.v	Clear text Verilog reset controller module for the sister RapidIO MegaCore. Required for running the demo testbench.
<*variation_name*>_riophy_xcvr.v	Clear text Verilog RTL wrapper for altgxb MegaFunction or XGMII interface for SerDes. Required for Place and Route.
<*variation_name*>_riophy_xcvr_sister.v	Clear text Verilog RTL wrapper for altgxb MegaFunction or XGMII interface for SerDes. Required for running the demo testbench.
<*variation_name*>_run_modelsim.tcl	Tcl script to run the demo testbench under ModelSIM and report the PASS/FAIL status of the testbench.
<*variation_name*>_tb.v	Verilog demo testbench for the RapidIO MegaCore function. Used in simulation.

Table 2–2. Project Files Note (1) (Part 4 of 4)	
Filename (2)	**Description**
<*variation_name*>_transport.v	Encrypted Verilog RTL for the Transport layer module. Required for Place and Route.
<*variation_name*>_transport_sister.v	Encrypted Verilog RTL for the Transport layer module of the sister RapidIO MegaCore function. Required for running the demo testbench.
<*variation_name*>_xgmii.v	Verilog RTL for the Serial RapidIO MegaCore function, in clear text, for the XGMII interface module. Required for Place and Route and for simulation with the demo testbench.
<*variation_name*>_xgmii_sister.v	Verilog RTL for the sister RapidIO MegaCore function's XGMII interface module. Required for running the demo testbench.

Notes to Table 2–2:
(1) These files are variation dependent, some may be absent or their names may change.
(2) <*variation name*> is a prefix variation name supplied automatically by the MegaWizard interface.

5. After you review the generation report, click **Exit** to close the MegaWizard.

You can now integrate your custom MegaCore function variation into your design, simulate, and compile.

Constraints are automatically set by the MegaWizard Plug-In Manager.

Simulate the Design

You can simulate your design using the MegaWizard interface-generated VHDL or Verilog HDL functional simulation model.

For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in Volume 3 of the *Quartus II Handbook*.

The RapidIO MegaCore function automatically generates a Verilog HDL demonstration testbench to match your specific variation. Scripts to compile and run the demonstration testbench using a variety of simulators and models also are provided. This testbench demonstrates how to instantiate a model in a design, and includes some simple stimulus to control the user interfaces of the RapidIO interface.

For a complete list of models or libraries required to simulate the RapidIO MegaCore function, refer to the *<variation name>*_**run_modelsim.tcl** script provided with the demonstration testbench.

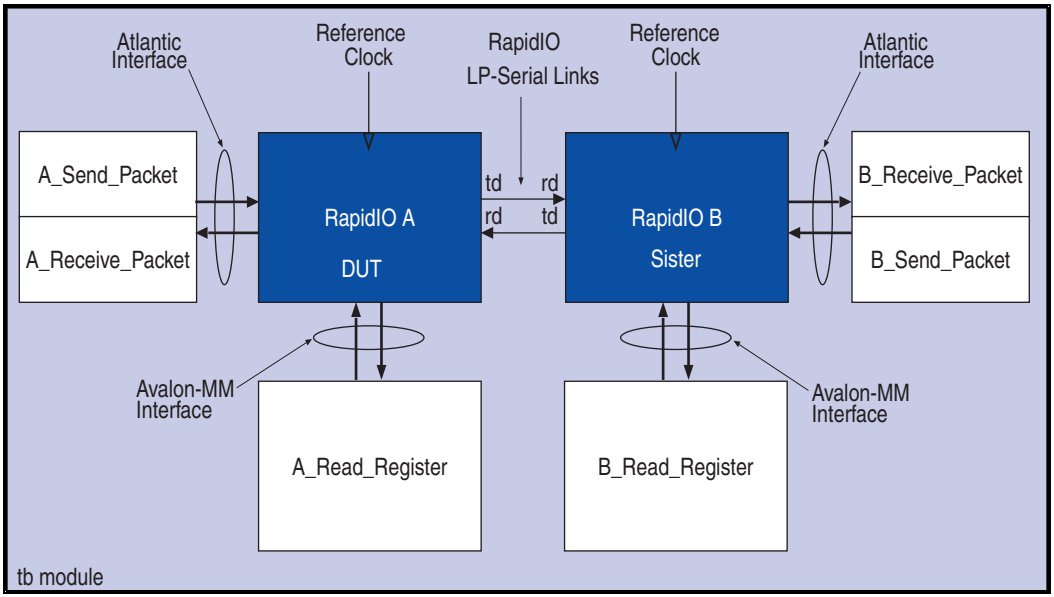
Demonstration Testbench for Variations with a Physical Layer Only

The demonstration testbench that is generated when you use only the physical layer tests the following functions:

- Port initialization process
- Transmission, reception, and acknowledgment of packets with 8 to 256 bytes of data payload
- Writing to and reading from the Atlantic interfaces
- Reading from the software interface registers

The testbench consists of two RapidIO MegaCore functions interconnected through their high-speed serial interfaces, see Figure 2–12. (Each MegaCore function's `td` output is connected to the other MegaCore function's `rd` input.) The testbench module provides clocking and reset control along with tasks to write to and read from the MegaCore function's Atlantic interfaces, and a task to read from the command and status register (CSR) set. For variations with external transceivers, these MegaCore functions are interconnected through their XGMII interfaces.

Figure 2–12. Serial RapidIO Physical Layer Demonstration Testbench *Note (1)*



Note to Figure 2–12:
(1) The external blocks, shown in white, are Verilog HDL tasks.

The testbench starts with the MegaCore functions in a reset state. A common reference clock is provided to all clock inputs. After coming out of reset, the MegaCore functions start the port initialization process to detect the presence of a partner and establish bit synchronization and code group boundary alignment. After the MegaCore functions have asserted their `port_initialized` output signals, the testbench checks that the port initialization process completed successfully by reading the `Error` and `Status` CSR to confirm the expected values of the `PORT_OK` and `PORT_UNINIT` register bits.

Packets with 8 to 256 bytes of data payload are then transmitted from one MegaCore function to the other. The receiving MegaCore function sends the proper acknowledgment symbols and the received packets are checked in the expected sequence for data integrity.

The format of the transmitted packets is described in Table 2–3.

Table 2–3. Serial Packets Format

Packet Byte	Format	Description
First Header word	*{AckID[4:0],Reserved[2:0], prio[1:0],tt[1:0],ftype[3:0]}*	AckID is set to zero and is replaced by the transmitting MegaCore function. The prio field is used by the receiver to select the output queue. The tt and ftype fields are used by the transport and logical layers and are ignored by the physical layer MegaCore functions, except I/O logical maintenance packet type.
DestinationID	*{DestinationID[15:0]}*	These fields are used by the Transport and Logical layers and are transferred unchanged by the physical layer MegaCore functions.
SourceID	*{SourceID[15:0]}*	
Last Header word	*{Transaction[3:0],Size[3:0],TID[7:0]}*	
Payload bytes	8 to 256 bytes	The payload bytes in the packet are set to an incrementing sequence starting at 0.

The received packet format is similar, but cyclic redundancy codes (CRCs) and padding (when required) are appended to the packet and an intermediate CRC is inserted in the packets after the first 80 bytes, when the packet size exceeds 80 bytes.

Table 2–4 lists the tasks used to write packets to a MegaCore function for transmission, read and check a received packet, and read the value from a register and compare it to an expected value.

Table 2–4. Serial Tasks

Function	Prototype	Comments
Write Packet to an Atlantic slave sink.	task send_packet; input [1:0] prio; input [1:0] tt; input [3:0] ftype; input [8:0] payload_sizes;	The payload_size should be an even number between 8 and 256 inclusive. The actual name of the task is pre-pended with A_ or B_ depending on which MegaCore function it should act.
Read and check a packet from an Atlantic slave source.	task receive_packet; input [1:0] prio; input [1:0] tt; input [3:0] ftype; input [8:0] payload_size;	prio—packet priority tt—transport type ftype—packet format type payload size—size of the packet payload
Read from Register	task read_register; input [15:0]address; input [31:0]expected;	The read value is compared to the expected value, any difference is flagged as an error. “don’t care” values can be specified by putting ‘x’s in the corresponding bit position.

All of the packets are sent contiguously, in sequence. After all packets have been sent, the idle sequence is transmitted until the end of the simulation.

The testbench concludes by checking that all of the packets have been received. If no error is detected and all packets are received, the testbench issues a **TESTBENCH PASSED** message stating that the simulation was successful.

If an error is detected, a **TESTBENCH FAILED** message is issued to indicate that the testbench has failed. A **TESTBENCH INCOMPLETE** message is issued if the expected number of checks is not made. For example, if not all packets are received before the testbench is terminated. The variable tb.exp_chk_cnt determines the number of checks done to insure completeness of the testbench.

To get a value change dump file called **dump.vcd** for all viewable signals, uncomment the line `//`define MAKEDUMP` in the <*variation name*>**_tb.v** file.

Demonstration Testbench for Variations with a Transport and Logical Layers.

When the variation includes a Transport layer and a Logical layer, the generated testbench tasks that generate and monitor packets on the Atlantic interfaces for the Physical-layer-only variations are replaced by tasks that generate and monitor transactions on the Avalon-MM and Avalon-ST interfaces.

If the Maintenance module is present, the testbench causes a few maintenance type read and write request packets to be sent from the main RapidIO MegaCore device under test (DUT) to a sister RapidIO MegaCore function. Transaction are initiated by doing Avalon-MM transactions on the DUT's Maintenance Slave Avalon-MM interface, and are checked on the sister's Maintenance Master Avalon-MM interface. Similarly, Input/Output or Doorbell, transactions are generated if the corresponding module has been selected when the MegaCore was parameterized. Avalon-ST packets are transferred through the Avalon-ST pass-through port if it is present.

IP Functional Simulation Model

To use the IP functional simulation model that you created, follow these steps:

For Solaris or Linux operating systems, type these commands:

1. Turn the <*variation name*>**_run_modelsim.tcl** script into executable files and change the permissions by typing:

```
chmod +x <variation name>_run_modelsim.tcl ↵
```

2. Run the script by typing:

```
./<variation name>_run_modelsim.tcl ↵
```

For Windows operating systems, run the script by typing this command:

```
vish <variation name>_run_modelsim.tcl ↵
```

In all cases, the testbench itself is in Verilog HDL, therefore a license to run mixed language simulations is required to run the testbench with the VHDL model.

In addition to the specified model, the scripts use a few clear-text source files: (See Table 2–2 on page 2–18.)

- <*variation name*>**_tb.v** is the top level testbench file
- <*variation name*>**_hutil.iv** defines a few general purpose testing utilities
- <*variation name*>**_demo_hookup.iv** connects the two instantiations of the MegaCore functions together and generates the required clock and reset signals
- <*variation name*>**_demo_util.iv** defines the tasks to read and write on the Avalon-MM or Atlantic interfaces.

Compile the Design

You can use the Quartus II software to compile your design. Refer to Quartus II Help for instructions on compiling your design.

The script file <*variation name*>**_constraints.tcl** sets required constraints for the compilation place and route. Run this script in the Quartus II software before you compile the variation.

SOPC Builder Design Flow Walkthrough

When building a system with SOPC Builder, you use the SOPC Builder interface to instantiate a RapidIO MegaCore function component and other available SOPC Builder components. The software automatically generates HDL files that include all of the specified components and interconnections. The HDL files are ready to be compiled in the Quartus II software to generate hardware for your system. A testbench module also is generated that includes basic transactions to validate the correctness of the generated HDL files.

For more information, refer to these topics and documents:

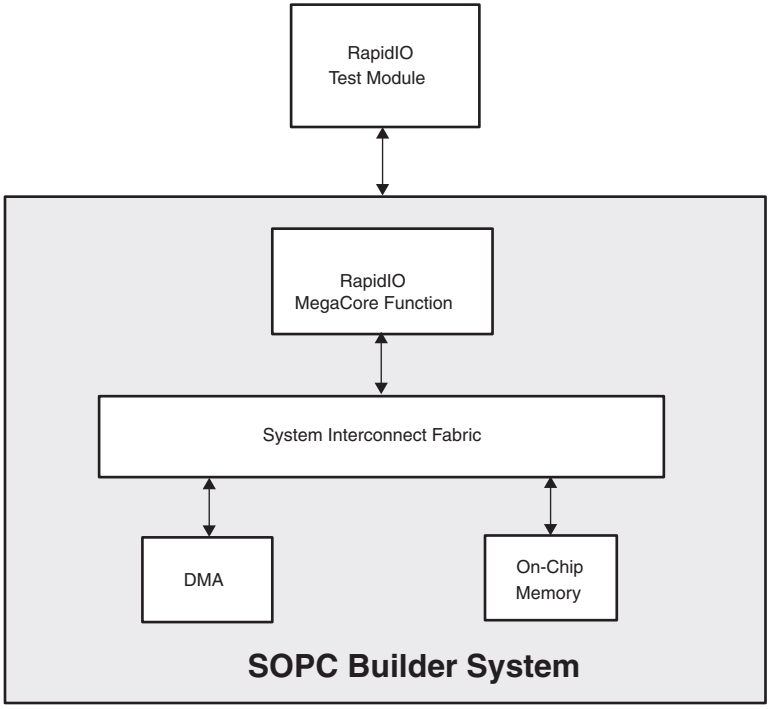
- The system interconnect fabric, refers to the System Interconnect Fabric chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder, refer to Section 1, SOPC Builder Features, and Section II, Building Systems with SOPC Builder in volume 4 of the *Quartus II Handbook*.

This walkthrough explains how to use SOPC Builder and the Quartus II software to generate a system containing the following components:

- RapidIO MegaCore function
- DMA controller
- On-chip memory

Figure 2–13 shows a block diagram of the system you create in this walkthrough.

Figure 2–13. Example SOPC Builder System



In this walkthrough, follow these steps:

- Create a New Quartus II Project
- Launch the SOPC Builder from Quartus II
- Instantiate and Parameterize the RapidIO Component
- Generate and Simulate the System

This example walkthrough does not use all available parameters and options. The RapidIO SOPC Builder design flow supports a subset of the RapidIO Megawizard Plug-In Manager flow parameters. The following parameters are supported in SOPC Builder flow as outlined below:

- The Transport layer is automatically enabled
- The Avalon-ST pass-through port is unavailable

For more information on specific parameters used in this walkthrough, refer to the MegaWizard Plug-In Manager Design Flow Walkthrough or Table 4–19 on page 4–78, Transport & I/O Logical Layer Parameters.

Create a New Quartus II Project

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the system name. To create a new project follow these steps:

1. On the Windows Start menu, click **Programs > Altera> Quartus II** *<version>* to start the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.

2. On the File menu, click **New Project Wizard...**

3. On the **New Project Wizard: Introduction** page, click **Next**.

   This introduction page does not display if you turned it off previously.

4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:

   a. Specify the following working directory for your project:

      c:\altera\project_rio\rapidio_sopc

   b. Specify **rio_sys** for the project name. You must specify the same name for both the project and the top-level design entity.

      The Quartus II software automatically specifies the top-level system that has the same name as the project. Do not change it.

   c. Click **Next** twice (on this and the next page) to display **New Project Wizard: Family & Device Settings**.

5. On the **Family & Device Settings** page, select the following target device family and options:

   a. In the **Family** list, select **Stratix II GX**.

   This walkthrough creates a design targeting the Stratix II GX device family. You also can use these procedures for other supported device families.

   b. In the **Target device** box, select **Specific device selected in Available devices list**.

c. Under **Show in 'Available device' list**, all fields should have the default value of **Any**.

d. In the **Available devices** list, select **EP2SGX90EF1152C3**.

6. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

7. Before starting SOPC Builder, check to ensure the RapidIO library is in the Quartus II Global User Libraries. If the RapidIO library is not listed in the Global User Libraries, add the RapidIO library by following these steps:

a. On the Quartus II Tools menu, select **Options**.

b. Select **Global User Library**.

c. In the **Library name** box, you can use the browser to locate the library for the RapidIO MegaCore function and add the RapidIO library path to the Global User Library or you can specify the path using the format:

<ip installed path> **/rapidio/lib**

d. Click **Ok** after specifying the RapidIO MegaCore function installation path to include this path in the Quartus II Global User Library.

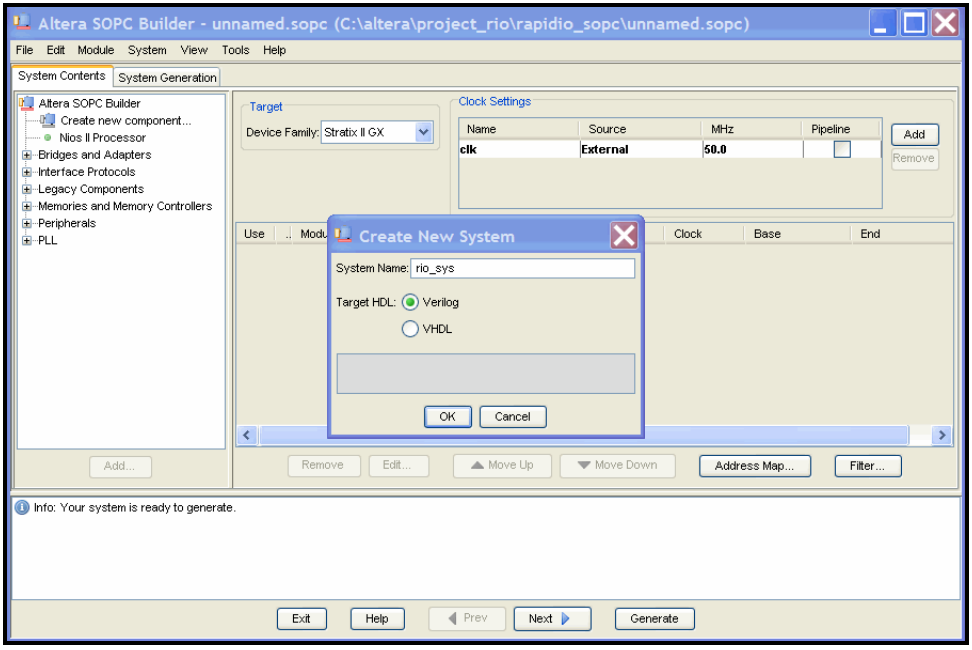
Launch SOPC Builder from Quartus II

Launch SOPC Builder from the Quartus II software.

1. On the Tools menu, click **SOPC Builder...** to start SOPC Builder. The **Altera SOPC Builder** window appears. See Figure 2–14.

Refer to Quartus II Help for more information on how to use SOPC Builder.

Figure 2–14. SOPC Builder Interface System Name Request Dialog



2. In the **System Name** box, type **rio_sys** for the project top-level system name. Under **Target HDL**, select **Verilog** and click **OK.** See Figure 2–14.

In this example, you are choosing the SOPC Builder-generated system file to be the same as the project's top level file. This is not required for your own design.

For your own project you can use Verilog or VHDL, however the RapidIO SOPC Builder testbench is only provided in Verilog. Additionally, the SOPC Builder simulation scripts only support a single HDL and thus can only run the RapidIO SOCP Builder testbench when the Target HDL is Verilog. To complete this walkthrough you must use Verilog as the Target HDL.

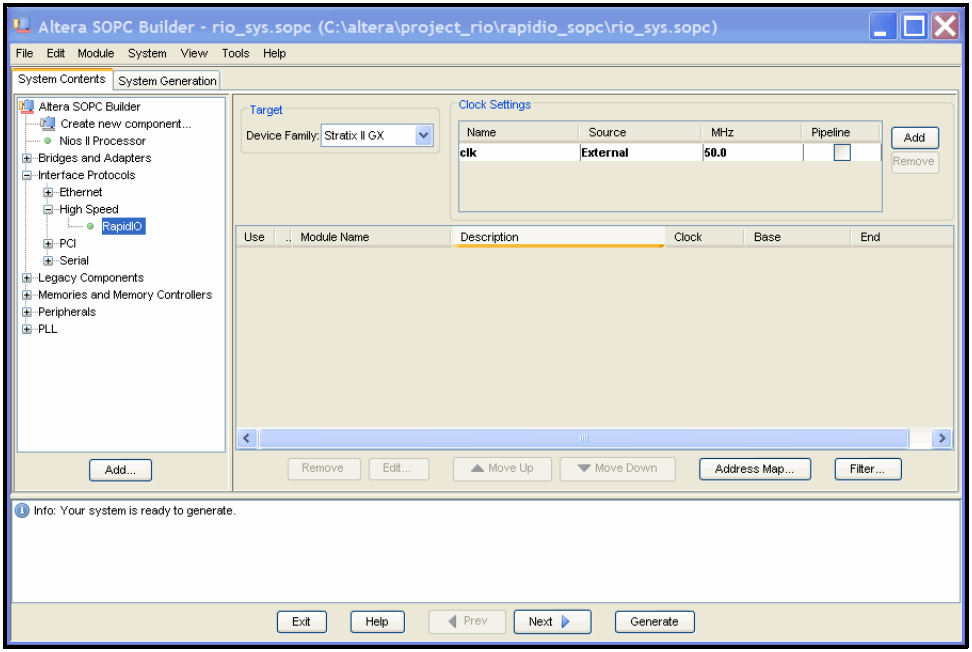
Instantiate and Parameterize the RapidIO Component

To instantiate and parameterize the RapidIO MegaCore component in your system, follow these steps:

1. Under **Interface Protocols** in the **High Speed** directory, click the **RapidIO** MegaCore component. See Figure 2–15

2. Click **Add...**. The SOPC Builder interface displays showing the RapidIO component. See Figure 2–15.

Figure 2–15. Add RapidIO MegaCore Function Component



In this section, you parameterize the RapidIO MegaCore function component.

Not all parameters are supported by and are relevant for every MegaCore function variation.

In SOPC Builder, the RapidIO MegaCore function automatically enables the Transport layer but the Avalon-ST pass-through port is not available.

Figure 2–16. Physical Layer Parameters

To parameterize your MegaCore function, follow these steps (see Figure 2–4):

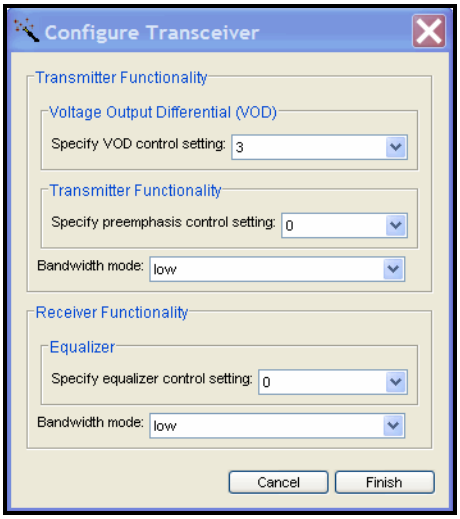
1. On the **Physical Layer** page, in **Mode Selection:** select **Serial 4x**.

Table 2–1 shows the baud rates supported by the serial RapidIO MegaCore function for each device family.

2. In the **Transceiver Selection**, select the **Stratix II GX PHY** transceiver.

   The Arria GX, Stratix GX or Stratix II GX PHY transceiver requires that you configure the altgxb or alt2gxb megafunction. For this example, the default settings are used. See Figure 2–17.

Figure 2–17. Configure Transceiver Settings



3. Under **Data Settings** set the following options:

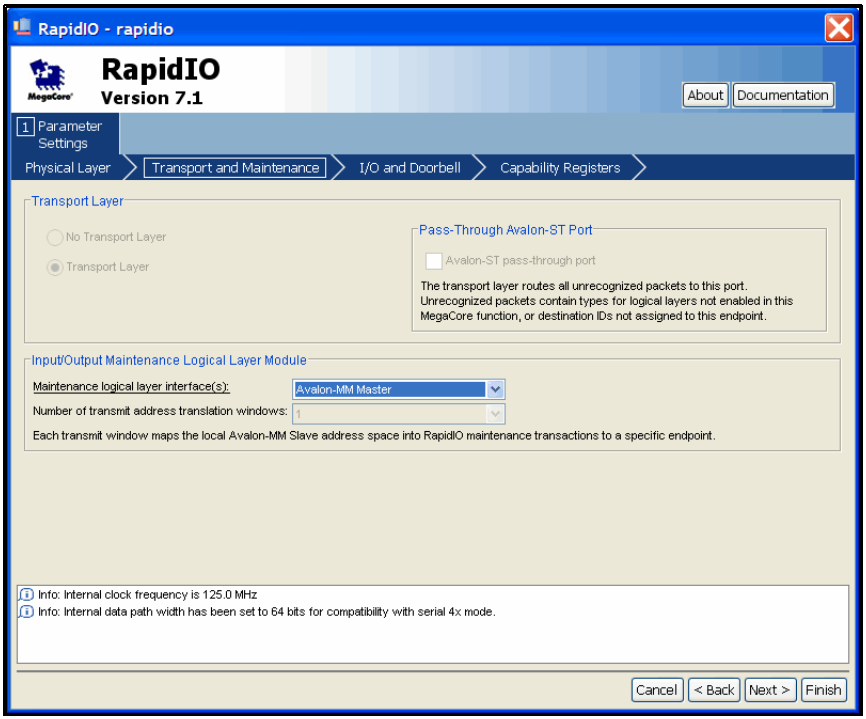
   - For **Baud rate**, specify `2500`.
   - For **Internal datapath width**, specify `64` bits.

4. Under receive **Priority Retry Threshold**, turn on the **Auto-configured from receiver buffer size**.

   Receiver priority retry thresholds are expressed in terms of 64-byte buffers. Each maximum size packet requires five buffers.

5. Click the **Next** to display **Transport and Maintenance** page; see Figure 2–18.

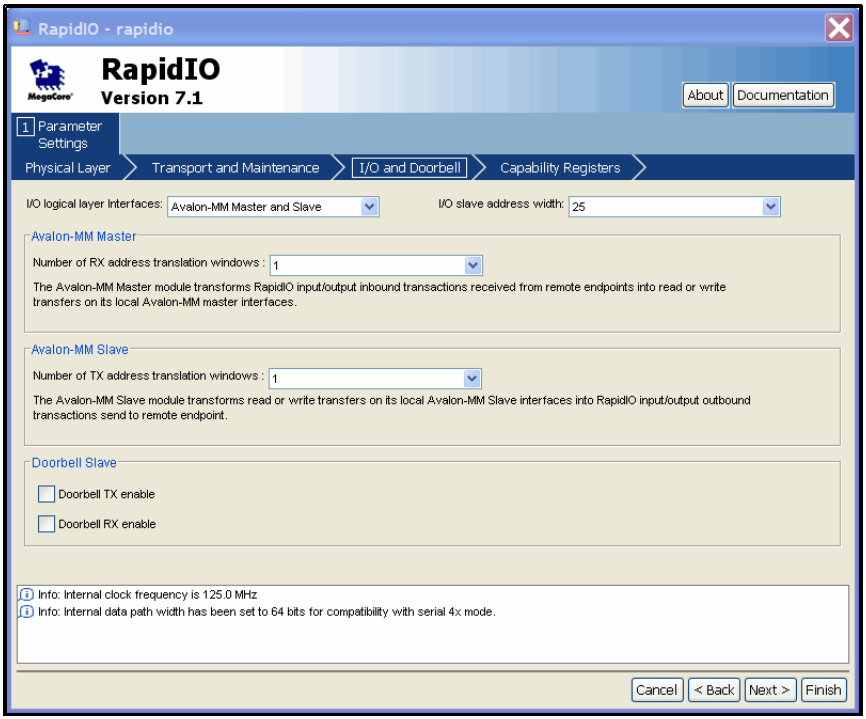
Figure 2–18. Transport Layer & Maintenance Parameters



For SOPC Builder, the Transport layer is always enabled and the Avalon-ST pass-through port is always disabled.

6. Under **Input/Output** Maintenance **Logical Layer Module**, select the following options:
   - For the **Maintenance logical layer interface(s):** select **Avalon-MM Master**.
   - For the **Number of transport address translation windows:** select **1**.
7. Click **Next** to display the **I/O and Doorbell** page; see Figure 2–19.

Figure 2–19. I/O Logical Layer and Doorbell Parameters



8. For the **I/O Logical layer Interfaces**, select **Avalon-MM Master and Slave**.

9. For the **I/O slave address width**, select **25**.

   The Input/Output Slave address width is set to **30** by default. However, to avoid over-allocating Avalon-MM memory space, setting this value to the lowest value for your system is highly recommended.

10. For the **Number of RX address translation windows**, select **1**.

11. For the **Number of TX address translation windows**, select **1**.

12. **Doorbell Slave** messaging is not turned on for this example.

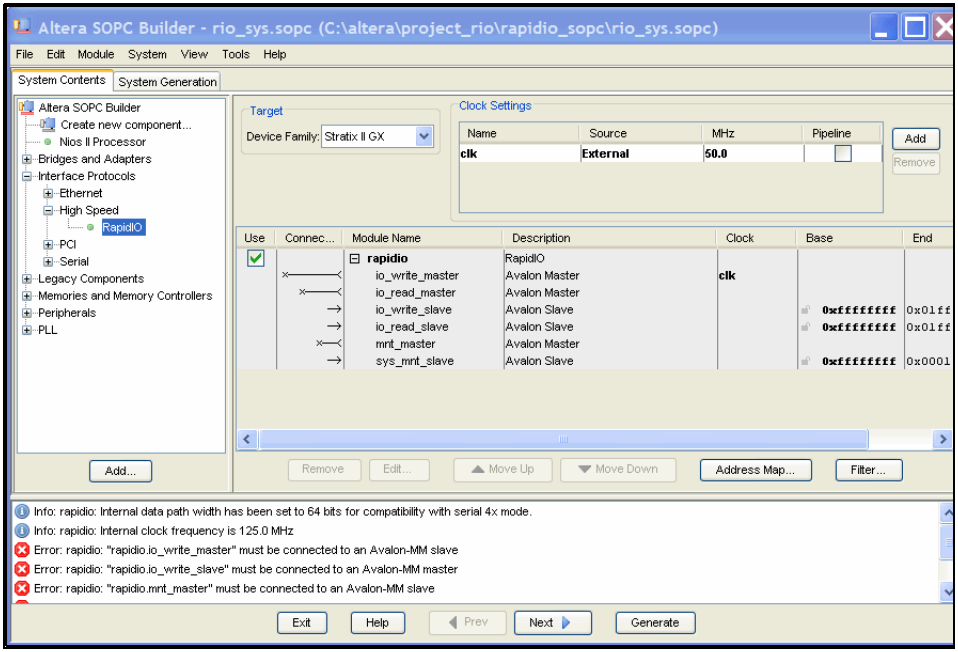
    When doorbell messaging is turned on, a 32- bit Avalon slave port enables doorbell messaging from the user application to the MegaCore function. Turning off doorbell messaging reduces resource usage and may be desirable for some applications.

13. Click **Finish** to complete parameterization and add the RapidIO MegaCore function to the SOPC Builder system

Add the RapidIO Component

After adding the RapidIO MegaCore function component to your system, various Avalon-MM ports are created and shown as connection points in the SOPC Builder interface. Error messages indicate that these ports are not connected. See Figure 2–20.

Figure 2–20. RapidIO MegaCore Function Component Added and Avalon-MM Ports Created



These errors are resolved after you add the remaining components to your system and make all of the appropriate connections. The default instance name of the RapidIO MegaCore function component is **rapidio**.

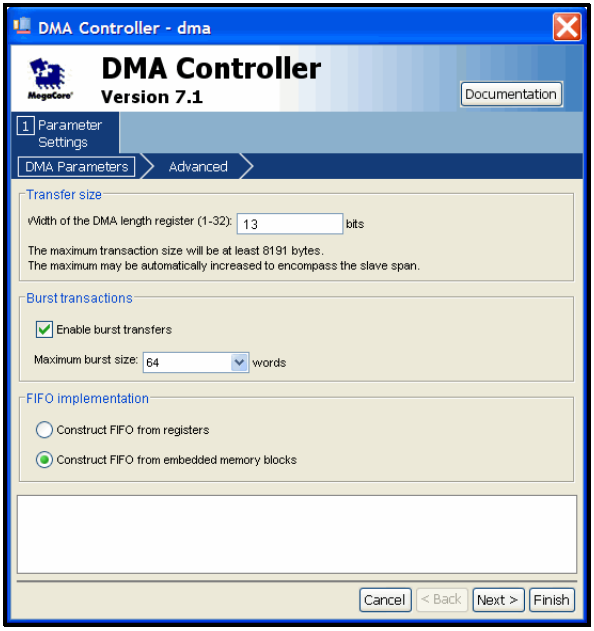
You can change the default name by right-clicking on the component name and click **Rename**. The component name must be unique; it cannot be the name of the system name.

Add the DMA Controller

Expand the **Memories and Memory Controllers** directory to add the **DMA controller** to your system.

1. Apply the DMA controller settings shown in Figure 2–21.
2. Click **Finish** to add the DMA controller to your SOPC Builder system.

Figure 2–21. Add the DMA Controller



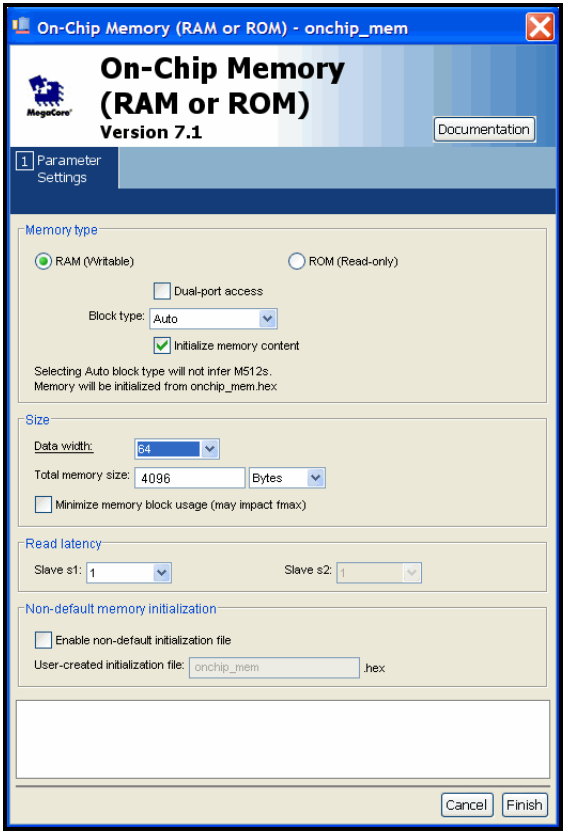
Add the On-Chip Memory

Expand the **On-Chip** directory to locate and add the **On-Chip Memory (RAM or ROM)** to your system.

1. Apply the On-Chip memory settings shown in Figure 2–22.

2. Click **Finish** to add the memory to your SOPC Builder system.

Figure 2–22. Add the On-Chip Memory (RAM or ROM)



Connect the System Components

For the external RapidIO processing elements to access the internal registers of the RapidIO variation, your created system must meet the following criteria:

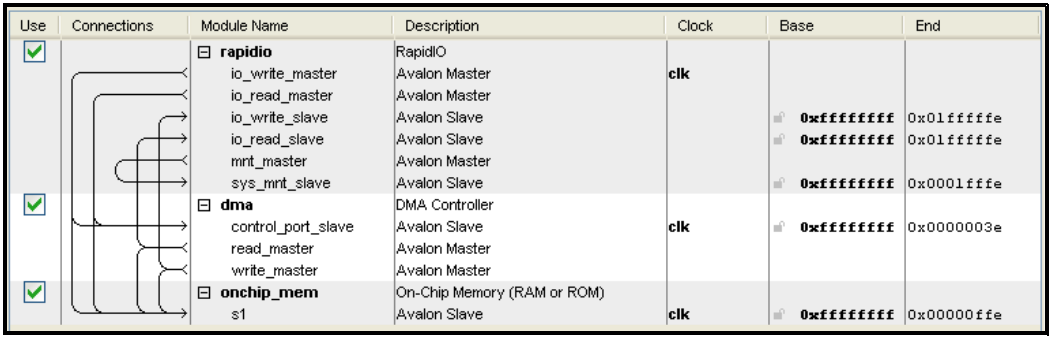
- The Maintenance Master port must be connected to System Maintenance Slave port
- The System Maintenance Slave port Base address must be assigned to address 0x0.

With all the system components added, in this step, connect the components as follows:

1. Connect rapidio `mnt_master` to `sys_mnt_slave`.
2. Connect rapidio `io_read_master` and `io_write_master` to onchip_mem `s1`.
3. Connect the rapidio `io_read_master` and `io_write_master` to the dma `control_port_slave`.
4. Connect the dma `read_master` to rapidio `io_read_slave`.
5. Connect dma `write_master` to rapidio `io_write_slave`.
6. Connect dma `read_master` and `write_master` to onchip_mem `s1`.

Refer to Figure 2–23 to ensure you correctly connected the above ports.

Figure 2–23. Complete System Connections



Assign Addresses and Set the Clock Frequency

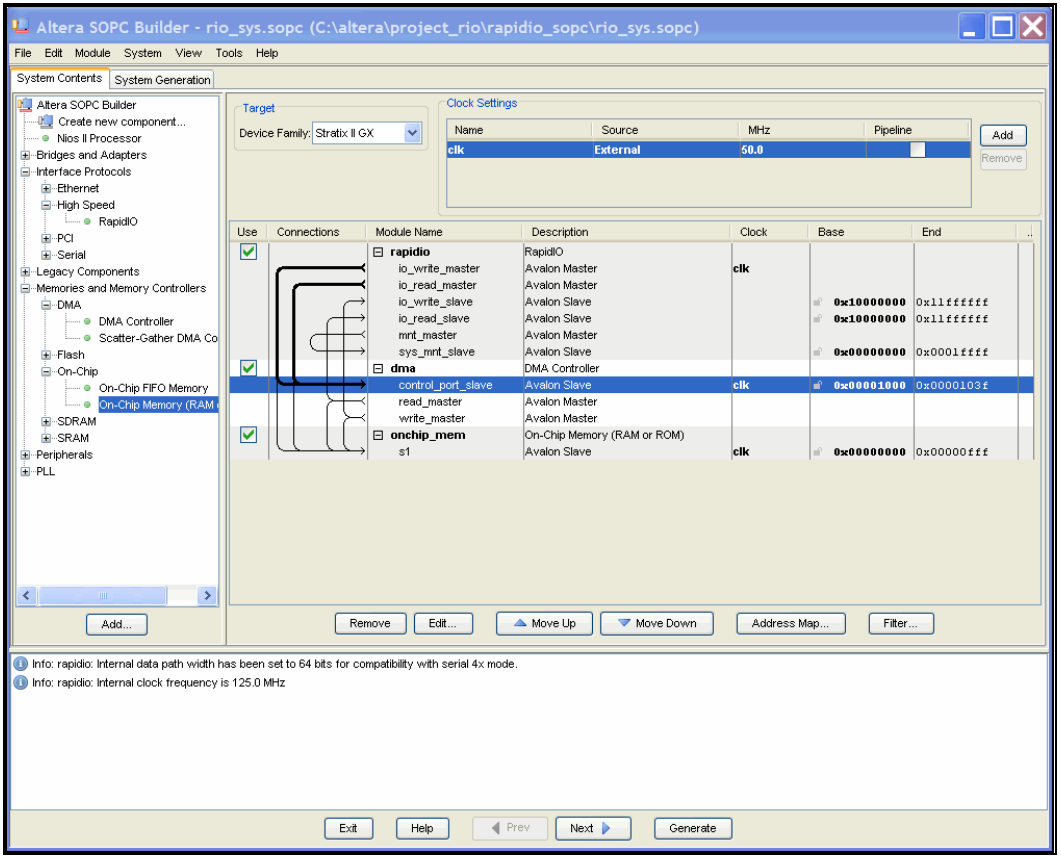
Make the following address assignments:

1. Assign rapidio `sys_mnt_slave` base address to `0x00000000`.
2. Assign rapidio `io_read_slave` and `io_write_slave` base address to `0x10000000`.
3. Assign dma `control_port_slave` base address to `0x00001000`.

4. Assign `onchip_mem s1` base address to `0x00000000`.
5. In the **Clock Settings** box, select the **50.0**, type **125** for the external clock source `clk` and type ↵.
6. On the File menu, click **Save** to save the SOPC Builder system.

Figure 2–24 shows the completed SOPC Builder example system.

Figure 2–24. Complete SOPC Builder Example System



Generate the System

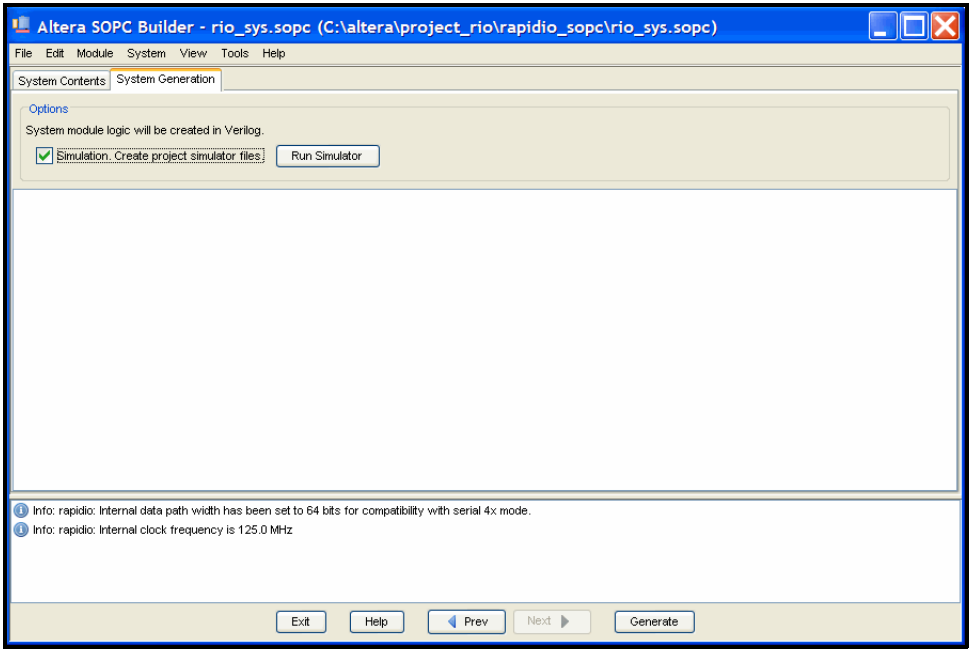
After you create your system with all the required components and connections and you have resolved any errors, generate the system by following these steps:

1. Click on the System Generation tab.
2. Turn on **Simulation. Create project simulator files.** This enables the generation of the testbench and simulation model files for your SOPC Builder system. See Figure 2–25.
3. Click **Generate** to start the generation process. See Figure 2–25.

Generating the system files, the simulation models, and environment takes a few minutes.

When the SOPC Builder system has been generated successfully, the system HDL files are added to your project directory and are ready to be simulated and compiled with the Quartus II software.

Figure 2–25. System Generation



Simulate the System

To simulate your system with the sample testbench, follow these steps:

1. In the project directory, for the testbench example, you must edit the **rapidio_sopc_tb.v file**. Open this file and search for the `SOPC_EXAMPLE_DESIGN` parameter. When you find this parameter, change the value from `0` to `1`.

2. Start the ModelSim simulator and change the directory to the **rio_sys_sim** directory under the project directory.

3. Type the following command at the simulator command prompt:

   ```
   source setup_sim.do ↵
   ```

   This simulation script will only work for RapidIO when the Target HDL language is Verilog.

4. To compile all the files and load the design, type the following command at the simulator prompt:

   ```
   s ↵
   ```

5. To simulate the design, type the following command at the simulator prompt:

   ```
   vsim4> run -all ↵
   ```

The RapidIO sample testbench performs the following transactions:

- Sends a sequence of read requests to the internal registers of the MegaCore function
- Sets up the address translation register within the MegaCore for maintenance and I/O transactions
- Programs the DMA transfer data between the test module and on-chip memory
- Verifies data integrity

When simulation finishes, exit ModelSim and return to the Quartus II software to compile your system.

Compile the System

The SOPC Builder generated HDL system files are ready to be compiled in the Quartus II software to generate the programming file for your system hardware. To compile your system in the Quartus II software, follow these steps:

1. Open the Quartus II project created in the "Create a New Quartus II Project" on page 2–29.
2. Source the generated Tcl script by typing:

```
source rapidio_constraints.tcl ↵
```

The **rapidio_constraints.tcl** script file sets the required constraints for compilation. The Fmax constraint on the Avalon domain defaults to 125MHz. Modify this constraint if the Avalon clock domain of your system operates at a different speed than the default setting.

3. From the Quartus II Processing menu, click **Start Compilation** to compile your system.

Program a Device

After you have compiled your design, program your targeted Altera device, and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the RapidIO MegaCore function before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model, and produce a time-limited programming file.

For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in Volume 3 of the *Quartus II Handbook*.

You can simulate the RapidIO MegaCore function in your design, and perform a time-limited evaluation of your design in hardware.

For more information on OpenCore Plus hardware evaluation using the RapidIO MegaCore function, see "OpenCore Plus Time-Out Behavior" on page 4–53, and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance, and want to take your design to production.

Set Up Licensing

After you purchase all licenses for the MegaCore function, you can request a license file from the Altera website at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

If your system instantiates other MegaCore functions that require licensing, you need to purchase appropriate licenses as required.

3. Physical Layer—Serial Specifications

Functional Description

This section describes the 1× or 4× serial Physical layer of the RapidIO™ MegaCore® function. The Physical layer is divided into three sublayers. The following sections describe these sublayers.

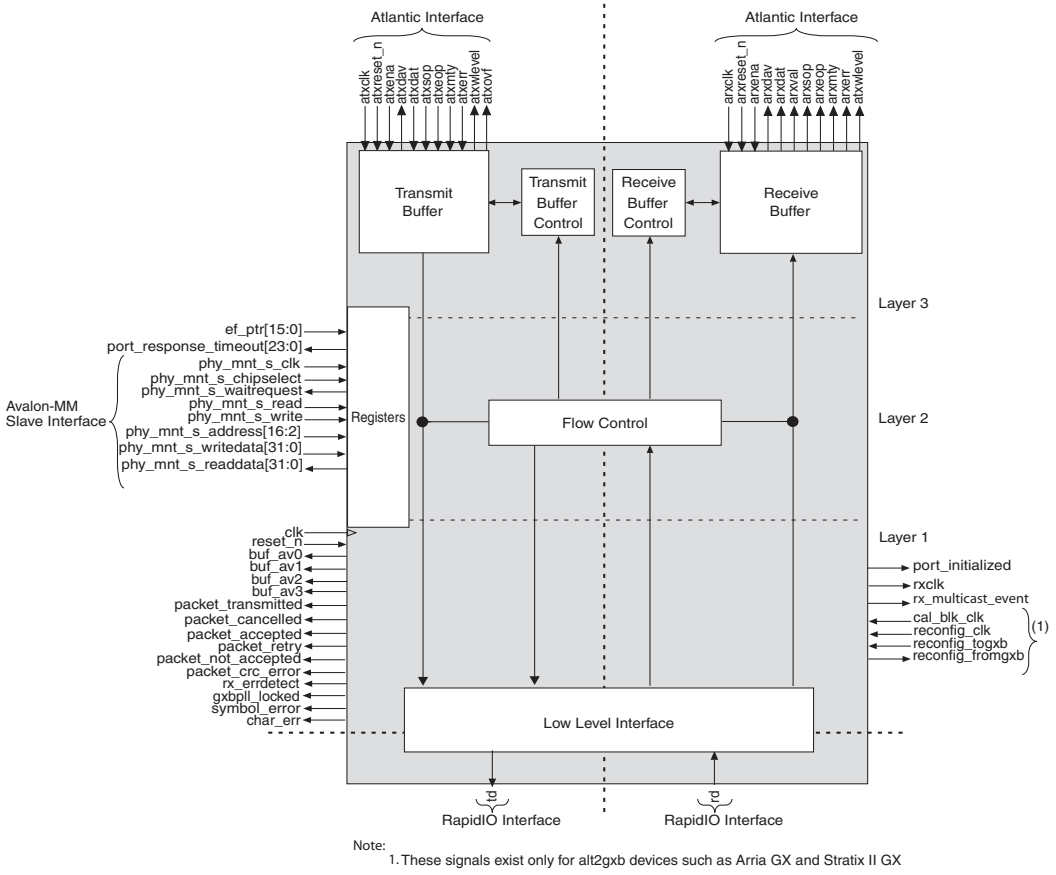
Features

- Layer 1
  - Port initialization
  - Receiver
    - One or four lane high-speed data deserialization (up to 3.125 Gbaud for 1× serial with 32-bit Atlantic™ interface; up to 4× 3.125 Gbaud for 4× serial with 64-bit Atlantic interface)
    - Clock and data recovery
    - 8B/10B decoding
    - Lane synchronization
    - Packet/control symbol delineation
    - Cyclic redundancy code (CRC) checking on packets
    - Control symbol CRC-5 checking
    - Error detection
    - Idle sequence removal
  - Transmitter
    - One or four lane high-speed data serialization (up to 3.125 Gbaud for 1× serial with 32-bit Atlantic interface; up to 4× 3.125 Gbaud for 4× serial with 64-bit Atlantic interface)
    - 8B/10B encoding
    - Packet/control symbol assembly
    - CRC generation on packets
    - Control symbol CRC-5 generation
    - Pseudo-random idle sequence generation
- Layer 2
  - Processor access (registers)
    - Status/control
  - Flow control (`AckID` window tracking)
    - Time-out on acknowledgements
  - Order of retransmission maintenance, and acknowledgements
  - `AckID` assignment
  - Error management

- Layer 3
  - Atlantic interface with clock decoupling
  - First-in first-out (FIFO) buffer with level output port
  - Adjustable buffer sizes
  - Transmitter
    - Four transmission queues, and four retransmission queues to handle packet prioritization
    - Up to 32-Kbytes of buffering
  - Receiver
    - Up to 32-Kbytes of buffering

Figure 3–1 shows a high-level block diagram of the serial RapidIO MegaCore function physical layer.

Figure 3–1. Serial RapidIO Block Diagram Parameters



Note:
1. These signals exist only for alt2gxb devices such as Arria GX and Stratix II GX

Interfaces

The Altera RapidIO MegaCore function physical layer supports the following interfaces:

- RapidIO interface
- Atlantic interface
- Avalon-MM Slave Interface
- XGMII interface

RapidIO Interface

The RapidIO interface is the serial interface described by the RapidIO Trade Association. The protocol is divided into a three-layer hierarchy: Physical layer, Transport layer, and Logical layer. This chapter focuses on the Physical layer implementation. The serial RapidIO interface can be used to communicate to another device that supports a RapidIO serial interface. It can also be used to communicate to multiple devices through a RapidIO switch. The communicating devices must operate in the same mode, data rate, and lane configuration.

More detailed information on the RapidIO interface is available from the RapidIO Trade Association's website at **www.rapidio.org**.

Atlantic Interface

The Atlantic interface, an Altera protocol, is the data plane that allows access to the physical layer. A user implemented Transport layer function uses the Atlantic interface to communicate with the Physical layer. For 1× serial variations, the Atlantic interface is always 32 bits. For 4× serial variations supporting up to 1.25 GBaud of throughput, the Atlantic interface is 32 bits. For 4× serial variations, the Atlantic interface is 64 bits.

The Atlantic interface is a full-duplex synchronous protocol. The transmit Atlantic interface supports 32- and 64-bit packet data transfers. It works as a slave-sink interface. The receive Atlantic interface supports 32- and 64-bit packet data transfers. It works as a slave-source interface.

The `arxdav` signal is asserted when a full packet is available to be read from the receive buffer.

If the `arxena` signal is asserted when the `arxdav` signal is not asserted, the first word becomes available on the Atlantic interface, and the `arxval` signal is asserted as soon as the first 64-byte block of a packet (or the full packet if it is smaller than 64 bytes) is ready to be read out of the receive buffer. Thus, the MegaCore function does not wait for the full packet before reading out the first block.

Atlantic Interface Error Management

For variations that do not implement the transport layer, to minimize latency the MegaCore function can start transmitting a packet before it is completely received on the transmit Atlantic interface. The MegaCore function also can start outputting the packet on the receive Atlantic interface before the packet is completely received from the link partner on the RapidIO interface. In this case, if a packet error is detected after transmission starts from the Atlantic link but before the entire packet has been received, the `arxerr` and `arxeop` signals are asserted and the packet is terminated. User logic should drop and ignore packets that have the `arxerr` signal asserted during transmission because the content of these packets is not reliable.

Similarly, if the user logic needs to abort the transmission of a packet that it has started to transfer to the MegaCore function through the transmit Atlantic interface, the user logic needs to assert the `atxerr` and `atxeop` signals. If the packet transmission has already started on the RapidIO port, the packet is aborted with a `stomp` control symbol.

The transmit Atlantic interface has an additional output signal, `atxovf`, that indicates a transmit buffer overflow condition. The `atxovf` output signal is asserted and the packet is dropped. If an attempt to start transmitting a new packet is made by asserting `atxena` and `atxsop` three clock cycles or more after `atxdav` is de-asserted, the packet will be dropped and the `atxovf` is output.

Atlantic Interface Error Handling Signals

The `arxerr` signal can be asserted for a variety of reasons, listed below. As an Atlantic signal, it is synchronous to `arxclk` and is only valid when `arxval` is asserted. Once asserted, `arxerr` stays asserted until the end of the packet when `arxeop` is asserted.

- CRC error—When a CRC error is detected, the `packet_crc_error` signal is asserted for one `rxclk` clock period. The `packet_not_accepted` signal is asserted when the `packet_not_accepted` symbol is transmitted. The `arxerr` signal is also asserted when the packet is read out of the Atlantic interface.

- Stomp—The `arxerr` signal is asserted if a `stomp` control symbol is received in the midst of a packet, causing it to be prematurely terminated. The `arxerr` signal is also asserted for any packet received between the `stomp` symbol and the following `restart-from-retry` symbol.

- Packet size—If a received packet exceeds the allowable size, it is cut short to the maximum allowable size (276 bytes total), and `arxerr` and `arxeop` are asserted on the last word.
- Outgoing symbol buffer full—Under some congestion conditions, there may be no space in the outgoing symbol buffer for the `packet_accepted` symbol. If this happens, the packet cannot be acknowledged and will have to be retried. Thus, `arxerr` is asserted to indicate to the downstream circuit that the received packet should be ignored because it will be retried.
- Symbol error —If an embedded symbol is errored, `arxerr` is asserted and the packet in which it is embedded is retried.
- Character error—If an errored character (an invalid 10-bit code, or a character of wrong disparity) or an invalid character (any control character other than the non-delimiting SC control character inside a packet) is received within a packet, the `arxerr` and `arxeop` signals are asserted and the rest of the packet is dropped.

For more information on these interfaces, refer to the *FS13: Atlantic Interface*, and the *Avalon Memory-Mapped Interface Specification* available at **www.altera.com**.

Avalon-MM Slave Interface

The 32-bit Avalon-MM slave interface is a control plane that allows access to the RapidIO MegaCore's internal registers. The CARs and CSRs defined in the RapidIO *Physical Layer Specification* are a subset of the registers that this interface supports. The full register set that is accessible through this interface is described in the section, "Software Interface" on page 3–32. The Avalon transactions supported by this interface are single reads and writes with variable latency.

XGMII External Transceiver Interface

The XGMII interface is the external transceiver interface that connects the RapidIO MegaCore function to the external physical coding sublayer (PCS) and physical media attachment (PMA) sublayer devices when an internal serial transceiver is not used.

The external transceiver interface provides 8-bit transmit and receive data paths per serial lane, plus the necessary control and clocking signals to allow bidirectional data transfers. This interface is similar to the 10 gigabit media independent interface (XGMII) for the 4× serial RapidIO protocol and gigabit media independent interface (GMII) for the 1× serial RapidIO protocol using either HSTL Class 1 or SSTL Class 2 IO drivers. The XGMII

supports one control signal per 8 bits for the external transceiver encoder, and one control and one error signal per 8 bits from the external transceiver decoder.

On the transmit side, the 8-bit data (`td`) and 1-bit control (`tc`) signals per lane are transmitted on the rising and falling edges of a center aligned clock, `tclk`. The external transmitter can be disabled by asserting the `phy_dis` signal high to force line errors. The external transmitter can be disabled when the Initialization State Machine (described in paragraph 4.7.3 of *Part 6: Physical Layer 1×/4× LP Serial Physical Layer Specification*, Revision 1.3) transitions to the SILENT state and drives the `phy_dis` signal high to simulate turning off the output driver.

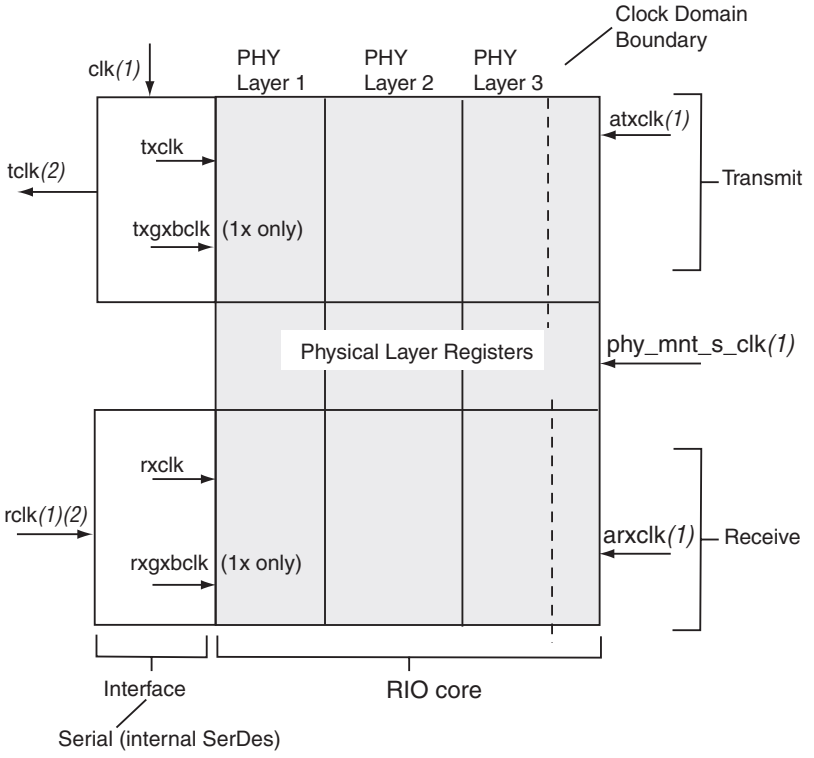
On the receive side, the 8-bit data (`rd`) and 1-bit control (`rc`) signals per lane are received and sampled on the rising and falling edges of a center aligned clock, `rclk`. Separate error (`rerr`) and `rclk` signals are associated with each lane.

See "Signals" on page 3–27 for further details. Additionally, Appendix C describes the timing requirements for the XGMII interface.

Clock Domains

In addition to the high-speed clock domains inside the Arria GX, Stratix GX or Stratix II GX transceiver, the 1× serial RapidIO MegaCore function comprises six clock domains: two transceiver clocks, two internal global clocks, and two Atlantic interface clocks. There is an additional clock, the clock domain for the `phy_mnt_s` Avalon-MM interface. See Figure 3–2 for the clock signals in a 1x serial RapidIO MegaCore function.

Figure 3–2. 1× Serial Clock Domains



Clock descriptions:

`txgxbclk`: Transmitter transceiver clock
`rxgxbclk`: Receiver transceiver clock
`txclk`: Transmitter internal global clock (same as clk)
`rxclk`: Receiver internal global clock
`atxclk`: Atlantic interface clock—greater than, or equal to txclk
`arxclk`: Atlantic interface clock—greater than, or equal to rxclk
`phy_mnt_s_clk`: Avalon-MM interface clock for register access
`rclk`: XGMII receive clock
`tclk`: XGMII transmit clock

Notes to Figure 3–2
(1) Input clocks (user supplied)
(2) Clocks rclk and tclk exist only when using an external SerDes with XGMII

The 4x serial RapidIO MegaCore function has multiple clock domains: a main system clock (txclk); a recovered clock, rxclk, two Atlantic interface clocks (atxclk and arxclk), two high-speed transceiver clocks, and the phy_mnt_s_clk Avalon-MM interface clock.

Stratix GX has multiple recovered clocks within the core for 4x variants.

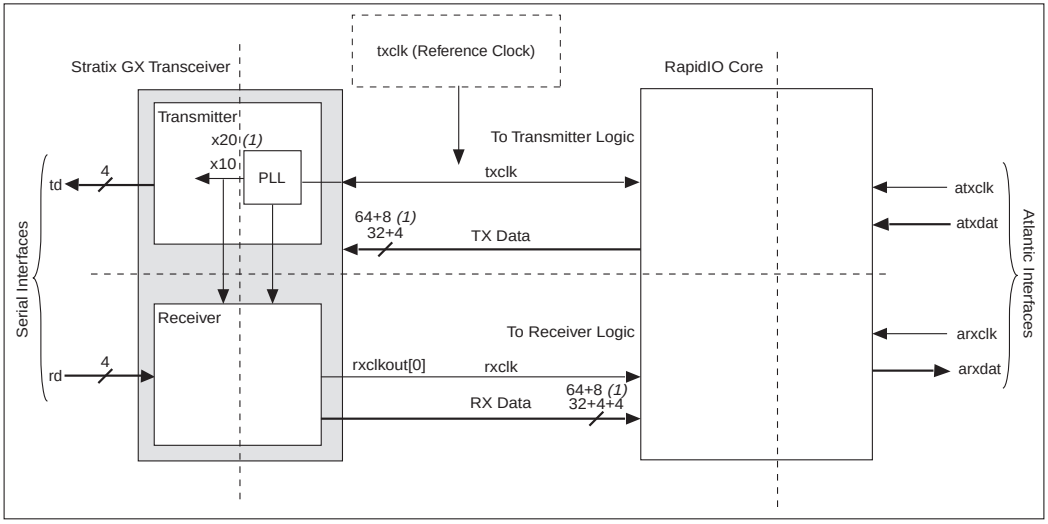
Stratix II GX uses 0ppm Quartus II settings as specified in the **constraints.tcl** file

The txclk clock is the internal system clock of the Physical layer. It is derived from the clk reference clock by division by one, two, or four, depending on the configuration of the MegaCore IP. Division is performed by a flip-flop-based circuit and does not require a dedicated PLL.

Figure 3–3 shows a top-level view of the clock domains and how they relate to each other.

The main system clock drives the transmit-side logic, and serves as a reference clock for the Arria GX, Stratix GX, or Stratix II GX transceiver's PLL. The PLL generates the high-speed transmit clock and the reference clocks for the receive-side high-speed deserializer clock and recovery unit (CRU). The CRU generates the recovered clock (rxclk) that drives the receive-side logic.

Figure 3–3. 4× Serial Clock Domains



Note to Figure 3–3:
(1) For 64-bit Atlantic interface.

Baud Rates

The serial RapidIO specification specifies baud rates of 1.25, 2.5, and 3.125 Gbaud. Table 3–1 shows the relationship between baud rates and internal clock rates for 1× serial.

Table 3–1. Baud Rates and Internal Clock Rates for 1× Serial

Baud Rates (Gbaud)	Transceiver Clocks (MHz) (clk/txgxbclk/rxgxbclk)	Internal Clocks (MHz) (txclk/rxclk)
3.125 (2)	156.25(2)	78.125(2)
2.5	125	62.5
1.25	62.5/125(1)	31.25

Note:
(1) 62.5 MHz for Arria GX or Stratix II GX devices; 125 MHz for Stratix GX devices.
(2) This rate is not supported for Arria GX devices.

Table 3–2 shows the relationship between baud rates and internal clock rates for 4× serial.

Table 3–2. Baud Rates and Internal Clock Rates for 4× Serial

Baud Rates (Gbaud)	Internal Clocks (MHz) (clk/txclk/rxclk)
	64-Bit Atlantic Interface
3.125 *(1)*	156.25 *(1)*
2.5	125
1.25	62.5

Note:
(1) This rate is not supported for Arria GX devices.

For more information on using high-speed transceiver blocks, refer to Volume 2, Section I, of the *Arria GX Device Handbook, Stratix GX Device Handbook* or the *Stratix II GX Device Handbook*.

Resets

All reset signals can be asserted asynchronously to any clock. However, most reset signals must be deasserted synchronously to a specific clock. The Atlantic interface resets, for example, should be deasserted on the rising edge of the corresponding clock.

See "Signals" on page 3–27 for further details.

Variations of the serial RapidIO MegaCore function that use the internal transceiver have a dedicated reset control module called riophy_reset to handle the specific requirements of the internal transceiver module. This reset control module is provided in the riophy_reset.v clear-text Verilog HDL source file, and is instantiated inside the top-level module found in the clear text <*variation_name*>_rio.v Verilog HDL source file.

Variations of the serial RapidIO MegaCore function that use an external transceiver do not require this special reset control module.

The riophy_reset module controls all of the RapidIO MegaCore function's internal reset signals. In particular, it generates the recommended reset sequence for the `altgxb` or `alt2gxb` high-speed serial I/O megafunction.

The sequence of events when reset_n is asserted, and as the MegaCore function comes out of reset after reset_n is deasserted, are described in the following steps.

When reset_n is asserted:

1. The internal signals rxreset_n and txreset_n are asserted to keep the riophy_dcore module in reset until the clocks it relies on are stable.
2. The gxbpll_areset, txdigitalreset, rxdigitalreset and rxanalogreset signals are asserted.

When reset_n is deasserted:

1. Wait at least 1 millisecond (ms); Stratix GX only.
2. Deassert gxbpll_areset; Stratix GX only.
3. Wait for gxbpll_locked to be asserted.
4. Deassert txdigitalreset and rxanalogreset.
5. Wait for rx_freqlocked to be asserted.
6. Wait for at least 2 ms.
7. Deassert rxdigitalreset.
8. Deassert rxreset_n and txreset_n.

The MegaCore function is now operating normally.

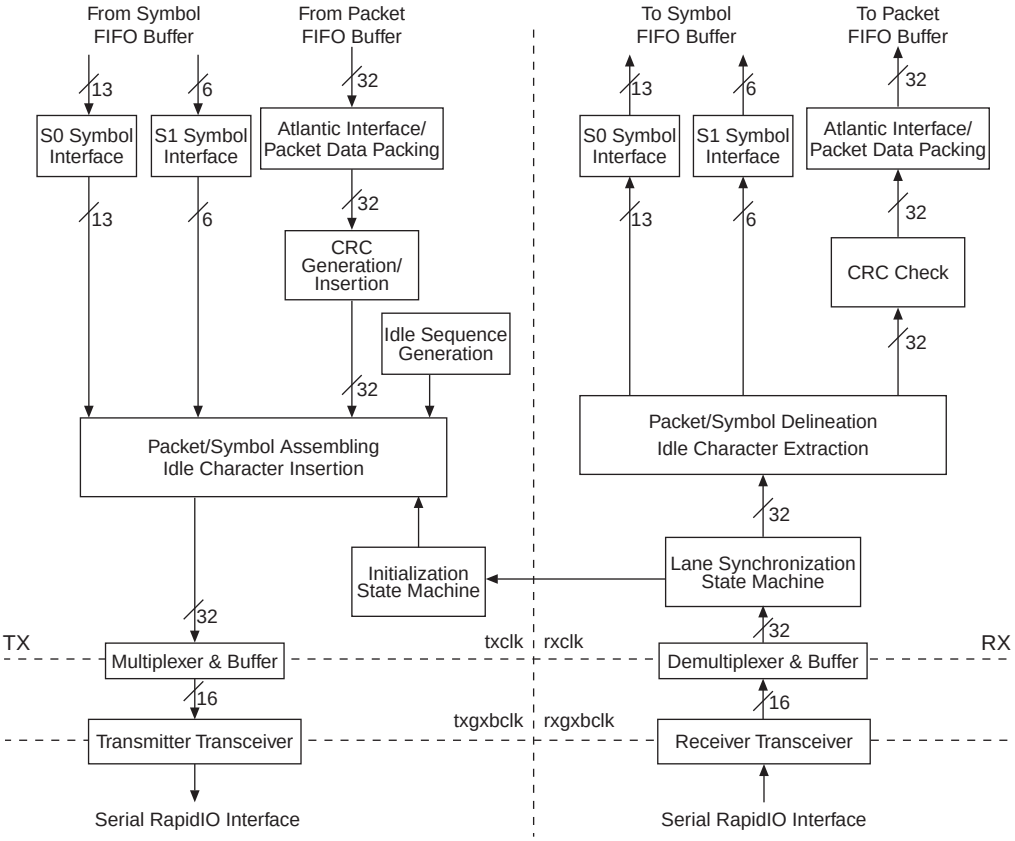
When, as part of its normal operation, the Initialization State Machine (described in paragraph 4.7.3 of *Part 6: Physical Layer 1×/4× LP Serial Physical Layer Specification, Revision 1.3*) transitions to the SILENT state and drives the link_drvr_oe signal low, the txdigitalreset signal of the altgxb or alt2gxb megafunction is asserted to simulate turning off the output driver. This causes a steady stream of K28.5 idle characters all of identical disparity to be transmitted. This, in turn, causes the receiving end to detect several disparity errors and forces the state machine to re-initialize, thus achieving the desired result of the SILENT state.

If two adjacent MegaCore functions are reset one after the other, one of the MegaCore functions may enter the Input Error Stopped state because one of the MegaCore functions is in the SILENT state while the other is already initialized. The initialized MegaCore function is the one to enter the Input Error Stopped state, and subsequently recover.

Layer 1

The layer 1 sublayer is designed to be a full-duplex interface with serial differential ports to a serial RapidIO device or MegaCore function. This section gives a block-by-block description of the layer 1 functions. Figure 3–4 shows a detailed block diagram of the layer 1.

Figure 3–4. Layer 1 Data Flow Block Diagram



Receiver

The layer 1 receiver sublayer receives and passes packets to the layer 3, and passes control symbols to the layer 2.

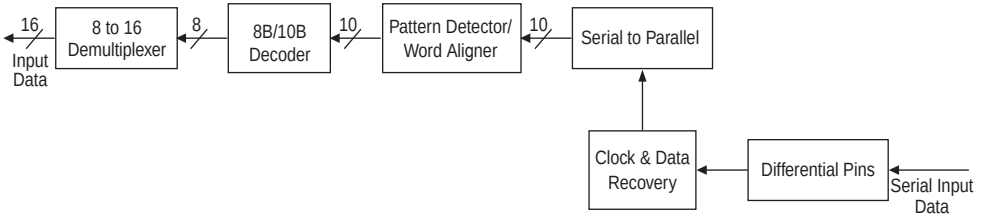
Clock & Data

The layer 1 receiver requires two clock domains: an Arria GX, Stratix GX, or Stratix II GX megafunction clock (`rxgxbclk`), and an internal global clock (`rxclk`).

Receiver Transceiver

The receiver transceiver is an embedded megafunction within the Arria GX, Stratix GX, or Stratix II GX FPGA. Serial data from differential input pins is fed into the clock and recovery unit (CRU) to detect clock and data. Recovered data is deserialized into 10-bit code groups and sent to the pattern detector and word aligner block to detect word boundaries. Properly aligned 10-bit code groups are then 8B/10B decoded into 8-bit characters and converted to 16-bit data via the 8-to-16 demultiplexer. Figure 3–5 shows the structure and the data flow of the receiver transceiver.

Figure 3–5. Receiver Transceiver Structure



Lane Synchronization State Machine

The lane synchronization state machine monitors the lane synchronization status. If the signal `lane_sync` is asserted, the lane is synchronized and the valid data is presented at the input path.

Packet/Symbol Delineation & Idle Character Extraction

The packet/symbol delineation and idle character extraction block delineates the input data into two data streams. One goes into the packet FIFO buffer, and the other goes into the symbol FIFO buffer.

This block also extracts idle characters from the data stream. It detects stomp symbol and packet size error, and asserts the corresponding error signals to layer 2. This block checks the 5-bit CRC at the end of the 24-bit symbol that covers the first 19 bits. The polynomial $x^5+x^4+ x^2+ 1$ is used. If the CRC is incorrect, the error signal `symbol_error` is asserted.

CRC Check

The RapidIO specification specifies that the Physical layer must add a 16-bit CRC to the end of all packets. The size of the packet determines how many CRCs are required.

- For packets of 80 bytes or fewer—header and payload data included—a single 16-bit CRC is appended to the end of the packet.
- For packets longer than 80 bytes—header and payload data included—two 16-bit CRCs are inserted after the 80^{th} transmitted byte. Two null padding bytes are appended to the packet before transmission if the resulting packet size is not an integer multiple of 4 bytes.

The middle CRC of a received packet is not removed in variations of the MegaCore function that include only the Physical layer. The Physical layer cannot determine whether the last bytes of a received packet are CRC or padding without looking at the transport and logical fields of the packet; the final CRC and padding bytes (if present) are not removed from the received packet and remain on the output of the receive Atlantic interface.

However, in variations of the MegaCore function that include Logical layers, the Transport layer removes the middle CRC after the 80^{th} byte (if present), and the Logical layer modules remove the final CRC and padding bytes (if present).

The CRC Check block uses the CCITT polynomial $x^{16} + x^{12} + x^5 + 1$ to check the 16-bit CRCs that cover all packet header bits (except the first six bits) and all data payload, and flags CRC and packet size errors.

Atlantic Interface/Packet Data Packing

This block sends 32- or 64-bit data to the upper layer via a 32- or 64-bit master-source Atlantic interface. It generates all required handshake signals for the interface.

S0 & S1 Symbol Interface

These blocks receive `stype0` control symbols and `stype1` control symbols, respectively. These blocks send control symbols to the upper layer via a simple dual-port FIFO interface.

Transmitter

The layer 1 transmitter sub-layer assembles packets and control symbols, received over a slave-source Atlantic interface, into one message and passes it to the serial RapidIO interface.

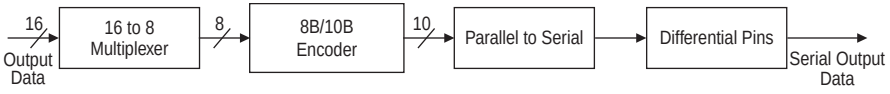
Clock and Data

For non-XGMII modes, the layer 1 transmitter uses two clocks: an Arria GX, Stratix GX, or Stratix II GX megafunction clock (`txgxbclk`), and an internal global clock (`txclk`).

Transmitter Transceiver (NonXGMII Mode)

The transmitter transceiver is an embedded megafunction within the Arria GX, Stratix GX or Stratix II GX FPGA. The 16-bit parallel output data is internally multiplexed to 8-bit data and 8B/10B encoded. The 10-bit encoded data is then serialized and sent to differential output pins. Figure 3–6 shows the transmitter transceiver structure and data flow direction.

Figure 3–6. Transmitter Transceiver Structure



Initialization State Machine

The serial port must be initialized before it can receive valid data. This state machine works closely with the lane synchronization state machine to monitor the `lane_sync` signal. When the `lane_sync` signal is asserted, the state machine enters the 1× or 4× mode state. The `port_initialized` signal is also asserted.

Packet/Symbol Assembling & Idle Character Insertion

The packet/symbol assembling and idle character insertion block assembles packet data and control symbol into a proper output format, with corresponding delimiting symbols and special characters. It generates 5 bit CRCs to cover the 19-bit symbol and appends the CRC at the end of the symbol. The polynomial $x^5+x^4+ x^2+ 1$ is used. It inserts an idle sequence if both the packet FIFO and symbol FIFO buffers are empty. During port initialization, it continues to send idle characters until the port is initialized. This module is also responsible for inserting status control symbols at least once every 1,024 transmitted code groups and the rate compensation sequence at least every 5,000 code groups or columns.

Idle Sequence Generation

When there is no data to transmit, layer 1 automatically inserts idle characters to transmit. As stated in *Part 6: Physical Layer 1×/4× LP Serial Physical Layer Specification* of the *RapidIO Interconnect Specification, Revision 1.3, February 2005:*

"The 1x idle sequence consists of a sequence of the code-groups /K/, /A/, and /R/ (the idle code-groups) and shall be used by ports in operating is 1x mode. The 4x idle sequence consists of a sequence of the columns ||K||, ||A||, ||R|| (the idle columns) and shall be used by ports operating in 4x mode. Both sequences shall comply with the following requirements:

1. The first code-group (column) of an idle sequence generated by a port operating in 1x mode (4x mode) shall be a /K/ (||K||). The first code-group (column) shall be transmitted immediately following the last code-group (column) of a packet or delimited control symbol.

2. At least once every 5000 code-groups (columns) transmitted by operating in 1x mode (4x mode), an idle sequence containing the /K/R/R/R/ code-group sequence (||K||R||R||R|| column sequence) shall be transmitted by the port. This sequence is referred to as the "compensation sequence".

3. When not transmitting the compensation sequence, all code-groups (columns) following the first code-group (column) of an idle sequence generated by a port operating in 1x mode (4x mode) shall be a pseudo-randomly selected sequence of /A/, /K/, and /R/ (||A||, ||K||, and ||R||) based on a pseudo-random sequence generator of 7th order or greater and subject to the minimum and maximum /A/ (||A||) spacing requirements.

4. The number of non /A/ code-groups (non ||A|| columns) between /A/ code-groups (||A|| columns) in the idle sequence of a port operating in 1x mode (4x mode) shall be no less than 16 and no more than 32. The number shall be pseudo-randomly selected, uniformly distributed across the range and based on a pseudo-random sequence generator of 7th order or greater."

CRC Generation & Insertion

The RapidIO specification specifies that the Physical layer must add a 16-bit CRC to the end of all packets. The CCITT polynomial $x^{16} + x^{12} + x^5 + 1$ is used for CRC generation. This block generates a CRC that covers all packet header bits, (except the first six bits) and all data payload. The size of the packet determines how many CRCs are required.

- For packets of 80 bytes or fewer—header and payload data included—a single 16-bit CRC is appended to the end of the packet.
- For packets longer than 80 bytes—header and payload data included—two 16-bit CRCs are inserted after the 80^{th} transmitted byte. Two null padding bytes are appended to the packet before transmission if the resulting packet size is not an integer multiple of 4 bytes.

Atlantic Interface/Packet Data Packing

The transmitter receives packet data from upper layers via a 32- or 64-bit master-sink Atlantic interface. It generates all required handshake signals for the interface.

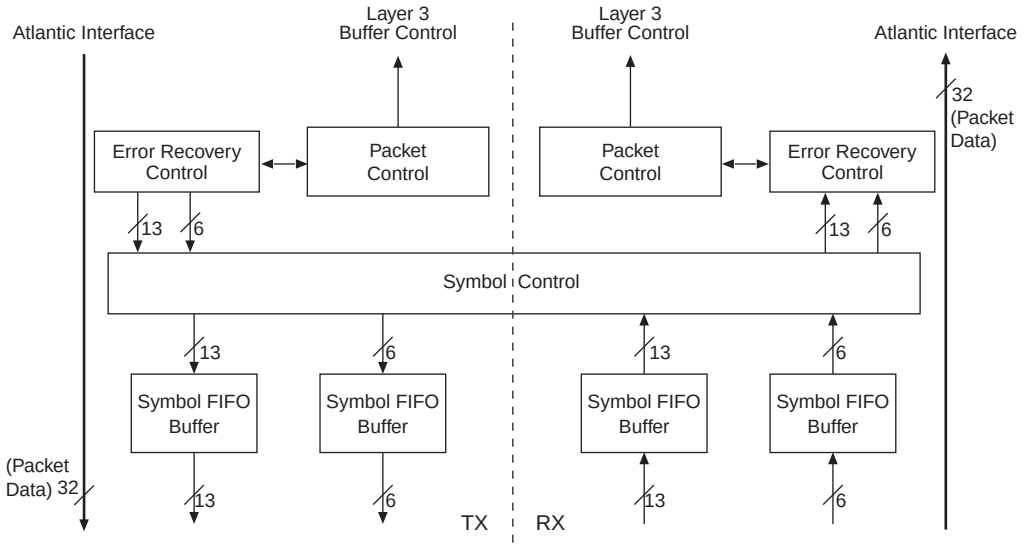
S0 & S1 Symbol Interface

The transmitter receives control symbols from upper layers via 13-bit and 6-bit FIFO interfaces. The 13-bit interface is for stype0 control symbols, and the 6-bit interface is for stype1 control symbols. It also decodes packet termination symbols: stomp, restart from retry, and link request.

Layer 2

The layer 2 sublayer provides flow control for the serial RapidIO Physical layer. This section gives a block-by-block description of the layer 2. Figure 3–7 shows a detailed block diagram of the layer 2.

Figure 3–7. Layer 2 Data Flow Block Diagram



Receiver

The layer 2 receiver sublayer is responsible for processing incoming control symbols. It also monitors incoming packet `ackIDs` to maintain proper flow.

Clock and Data

The layer 2 receiver comprises one clock domain: an internal global clock (`rxclk`).

Symbol FIFO Buffer

Incoming 13-bit stype0 control symbols and 6-bit stype1 control symbols are stored in their respective symbol FIFO buffers by the layer 1. These symbols are retrieved by the layer 2 for further processing.

Symbol Control

On the receive side, the layer 2 keeps track of the sequence of `ackIDs` and tells the layer 3 which packets have been acknowledged, and which packets to drop.

Packet Control

The packet control block uses a sliding window protocol to handle incoming and outgoing packets. Each incoming and outgoing packet has an attached 5-bit `ackID` in the header field. The value of `ackID` is zero at reset. It increments after each packet is sent out, and rolls over to zero after it has reached 31. All packets can only be accepted by the receiver in the sequential order specified by the `ackID`. If a packet cannot be accepted by the receiver due to buffer congestion, a packet retry request control symbol with the lost `ackID` is sent to the transmitter. The sender then retransmits all packets starting from the lost `ackID`.

Error Recovery Control

A packet or control symbol corrupted by an incorrect CRC, or by a CRC-5 error, must be recovered. During the error recovery process, two interdependent state machines are required to operate the input and output ports, respectively.

When an incoming packet or control symbol is corrupted, the receiver sends a `packet not accepted` symbol to the sender. A link-request link-response control symbol pair is then exchanged between the link partners and the sender then retransmits all packets starting from the `ackID` of the corrupted packet.

Transmitter

The layer 2 transmitter sublayer is responsible for creating and transmitting outgoing control symbols. It also monitors outgoing packet `ackID`s to maintain proper flow.

Clock and Data

The layer 2 transmitter comprises one clock domain: an internal global clock (`txclk`).

Symbol FIFO Buffer

The layer 2 provides these symbol FIFO buffers to store outgoing 13-bit stype0 control symbols and 6-bit stype1 control symbols. These symbols are retrieved by the layer 1, and sent out via the serial RapidIO interface.

Symbol Control

On the transmit side, the layer 2 keeps track of the sequence of ackIDs and tells the layer 3 which packet to send with what ackID. The layer 2 also tells the layer 3 which packet has been acknowledged, and thus can be discarded in the buffers.

Packet Control

The packet control block uses a sliding window mechanism to handle incoming and outgoing packets. This block also sets the time-out counters for each outgoing packet. When a time-out occurs for an outgoing packet, the packet control block treats it as an unexpected acknowledge control symbol, and starts the recovery process.

Error Recovery Control

Protocol violations or detection of a corrupted control symbol or packet cause the error recovery process to be initiated. During the error recovery process, two interdependent state machines are required to operate the input and output ports, respectively.

For error recovery, transmitted packets are held by the output port for possible retransmission in case an error is detected by the receiving device. The packets are held until the sending device receives a packet-accepted control symbol for that packet. If a packet is retransmitted, the time-out counter is reset for that retransmitted packet.

Layer 3

The layer 3 sublayer provides buffers, and buffer management for packet data. This section briefly describes the layer 3 functions.

Receiver

The layer 3 receiver sublayer accepts packet data from the layer 1 sublayer, and stores it in its buffers for the user. The receiver buffer is partitioned in 64-byte blocks that are allocated from a free queue as required, and returned to the free queue when no longer needed. Up to five 64-byte blocks can be used to store a packet.

The RapidIO Specification requires that at least one packet, of higher priority than all previously transmitted packets, always be able to pass through.

To meet this requirement, the layer 3 receiver sublayer accepts or retries received packets based on their priority and the receive buffer's fill level.

The receiver bases its decision to accept or retry packets on three programmable threshold levels: Threshold_2, Threshold_1, and Threshold_0.

- Packets of priority 3 (highest priority) are retried only if the receiver buffer is full.
- Packets of priority 2 are retried only if the number of available free 64-byte blocks is less than Threshold_2.
- Packets of priority 1 are retried only if the number of available free 64-byte blocks is less than Threshold_1.
- Packets of priority 0 (lowest priority) are retried only if the number of available free 64-byte blocks is less than Threshold_0.

The default threshold values are:

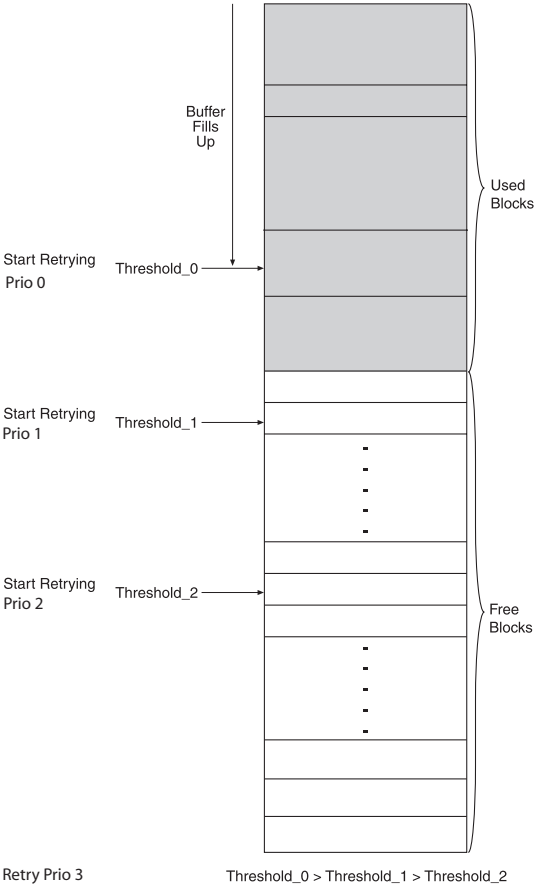
- Threshold_2 = 10
- Threshold_1 = 15
- Threshold_0 = 20

The threshold values are programmed through the MegaWizard interface by turning off **Auto-configured from receiver buffer size** on the Physical layer page.

To comply with the RapidIO specification, the threshold values must increase monotonically by at least the size of one packet (see Figure 3–8 on page 3–22). The MegaWizard Plug-In enforces the consistency checks.

- threshold_2 > 9
- threshold_1 > threshold_2 + 4
- threshold_0 >threshold_1 + 4
- threshold_0 < Number of available buffers

Figure 3–8. Receiver Threshold Levels



Clock & Data

The layer 3 receiver sublayer comprises two clock domains: an internal global clock (`rxclk`), and an Atlantic interface clock (`arxclk`). The buffer provides clock decoupling.

Receiver Buffers

The buffer size can be configured to 4, 8, 16, or 32 Kilobytes.

The following fatal errors cause the receiver buffer to be flushed, and any stored packets to be lost:

- Reception of a port-response control symbol with the `port_status` set to `Error`.
- Reception of a port-response control symbol with the `port_status` set to `OK` but the `ackid_status` set to an ackid that is not pending (transmitted but not acknowledged yet).
- Transmitter times out while waiting for link-response.
- Receiver times out while waiting for link-request.
- Reception of four link-request control symbols with the `cmd` set to `reset-device` in a row.

See Table 1–3 on page 1–2 and Table 1–4 on page 1–3 for examples of memory usage depending on buffer size.

Transmitter

The layer 3 transmitter sub-layer accepts packet data from the Atlantic interface, and stores it into its buffer for the layer 1 sublayer.

The RapidIO Specification requires that newly arrived, higher priority packets be transmitted ahead of the retransmission of previously transmitted, but not acknowledged, (retried) lower priority packets. The Specification also requires that at least one packet, of higher priority than all previously transmitted packets, always be able to pass through. To meet these requirements, the layer 3 transmitter sublayer includes four transmit queues and four retransmit queues, one for each priority level.

Transmit & Retransmit Queues

As packets are written to the transmitter's Atlantic interface, they are added to the tail end of the appropriate priority transmit queue. The transmitter always transmits the packet at the head of the highest priority non-empty queue. Once transmitted, the packet is moved to the corresponding priority retransmit queue.

When a `packet-accepted` control symbol is received for a non-acknowledged transmitted packet, the accepted packet is removed from its retransmit queue.

If a `packet-retry` control symbol is received, all of the packets in the re-transmit queues are returned to the head of the corresponding transmit queues. The transmitter sends a restart-from-retry symbol, and the transmission resumes with the highest priority packet available, possibly not the same packet that was originally transmitted and retried. If higher priority packets have been written to the Atlantic interface since the retried packet was originally transmitted, they will automatically be chosen to be transmitted before lower priority packets are retransmitted.

The layer 2 ensures that no more than 31 unacknowledged packets are transmitted, and that the AckIDs are used and acknowledged in incrementing order.

Transmit Buffer

The transmit buffer is the main memory in which the packets are stored. The buffer is partitioned into 64-byte blocks to be used on a first-come, first-served basis by the transmit and retransmit queues.

The following fatal errors cause the transmit buffer to be flushed, and any stored packets to be lost:

- Reception of a port-response control symbol with the `port_status` set to `Error`.
- Reception of a port-response control symbol with the `port_status` set to `OK` but the `ackid_status` set to an ackid that is not pending (transmitted but not acknowledged yet).
- Transmitter times out while waiting for link-response.
- Receiver times out while waiting for link-request.
- Reception of four link-request control symbols with the `cmd` set to `reset-device` in a row.

Clock & Data

The layer 3 transmitter sublayer requires two clock domains: an internal global clock (`txclk`), and an Atlantic interface clock (`atxclk`). The buffer provides clock decoupling.

Forced Compensation Sequence Insertion

As packet data is written to the transmit Atlantic interface, it is stored in 64-byte blocks. To minimize the latency introduced by the RapidIO MegaCore function, transmission of the packet starts as soon as the first 64-byte block is available (or the end of the packet is reached, for packets shorter than 64 bytes). Should the next 64-byte block not be available by the time the first one has been completely transmitted, status control symbols are inserted in the middle of the packet in lieu of idles as the true idle sequence can only be inserted between packets, and cannot be embedded inside a packet. This, along with other embedded symbols, such as packet-accepted symbols, causes the transmission of the packet to be stretched in time.

Compensation sequences must be inserted every 5,000 code groups or columns, and must be inserted between packets. The serial RapidIO MegaCore function checks whether the 5,000 code group deadline is approaching before the transmission of every packet, and inserts a

compensation sequence when the number of code groups or columns, left before the required compensation sequence insertion, falls below a specified threshold.

That threshold is chosen to allow time for the transmission of a packet of maximum legal size (276 bytes), even when it is stretched by the insertion of a significant number of embedded symbols. (Up to 37 embedded symbols, or 148 bytes, theoretically need to be embedded should the traffic in the other direction consist of minimum-sized packets.)

In some cases, for example when using an extremely slow transmit Atlantic clock, the transmission of a packet can be stretched beyond the point where a compensation sequence must be inserted. When this occurs, the packet transmission is aborted with a `stomp` control symbol, the compensation sequence is inserted, and normal transmission resumes.

When the receive side receives the stomped packet, it simply marks it as errored by asserting `arxerr`.

No traffic is lost and no protocol violation occurs, but an unexpected `arxerr` assertion occurs.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation can support the following two modes of operation:

- *Untethered*—the design runs for a limited time
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.

For MegaCore functions, the untethered timeout is 1 hour; the tethered timeout value is indefinite.

Your design stops working after the hardware evaluation time expires. After that time, the RapidIO MegaCore function behaves as if its Atlantic interface signals `atxena` and `arxena` are tied low. All Tx and Rx packet transfers through the physical layer are suppressed.

As a result, it is impossible for the RapidIO MegaCore function to transmit new packets (it will only transmit idles and status control symbols), or read packets out of the Atlantic interface. If the far end

continues to transmit packets, the RapidIO MegaCore function starts refusing new packets by sending `packet_retry` control symbols as soon as its receiver buffer fills up beyond the corresponding threshold.

For more information on OpenCore Plus hardware evaluation using the RapidIO MegaCore function, see *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Parameters

Table 3–3 shows the RapidIO Physical layer function parameters, which can only be set in the MegaWizard interface (see "Parameterize" on page 2–7).

Table 3–3. Serial RapidIO Physical Layer Parameters (Part 1 of 2)

Parameter	Value	Description
`Mode selection`	Serial 1× or 4×	One or four lane high-speed data serialization or deserialization (up to 3.125 Gbps)
`PHY selection`	Arria GX PHY, Stratix GX PHY, Stratix II GX PHY, or External transceiver	The Arria GX PHY, Stratix GX PHY, and Stratix II GX PHY options enable serial RapidIO variations using the built-in transceiver blocks of the respective device families. You can use the **Configure Transceiver...** button in the Device panel of the Physical layer page to set the analog parameters for the transceiver block. The External Transceiver mode enables serial RapidIO variations with Stratix GX, Stratix II GX, Arria GX, and any supported device family.
`Baud rate`	From 500 to 3.125 Mbaud *(2)*	The baud rate is the external device's serial data rate. The serial RapidIO specification specifies baud rates of 1.25 to 3.125 Mbaud. Table 3–1 on page 3–9 shows the relationship between baud rates and internal clock rates.
`Receive buffer`	4, 8, 16, or 32 Kbytes *(1)*	The Receive buffer parameter allows you to select the receiver buffer size.
`Transmit buffer`	8, 16, or 32 Kbytes *(1)*	The Transmit buffer parameter allows you to select the transmitter buffer size.

Table 3–3. Serial RapidIO Physical Layer Parameters (Part 2 of 2)

Parameter	Value	Description
Receive priority 0/1/2 retry threshold	Threshold_2 > 9 Threshold_1 > Threshold_2 + 4 Threshold_0 > Threshold_1 + 4 Threshold_0 < (2 ** p_rxbuf_addr_width)	When the number of available free 64-byte blocks in the receive buffer is less than one of these thresholds, the receiver refuses incoming packets of the corresponding priority level by sending packet-retry symbols. These priorities can be set automatically by turning on the **Auto-configured from receiver buffer size:** option of the Receive Priority Retry Threshold panel on the Physical layer page.

Notes to Table 3–3:
(1) Buffers are implemented in embedded RAM blocks. Depending on the size of the device used, the maximum buffer size may be limited by the number of available RAM blocks.
(2) Arria GX does not support 3.125 Mbaud.

Signals

Tables 3–4 through 3–14 list the pins used by the serial RapidIO MegaCore function, with the I/Os shown in Figure 3–1 on page 3–2. The active-low signals are indicated by _n.

For signals and bus widths specific to your variation, refer to the HTML file generated by the MegaWizard interface.

Tables 3–4 through 3–6 list the signals used in the serial layer 1.

Table 3–4. Serial RapidIO Interface Layer 1 Signals

Signal	Direction	Description
rd	Input	Receive data—a unidirectional data receiver. It is connected to the td bus of the transmitting device.
td	Output	Transmit data—a unidirectional point-to-point driver to transmit the packet information. The td bus of one device is connected to the rd bus of the receiving device.

Table 3–5. External Transceiver Interface Signals (Part 1 of 2)

Signal	Direction	Description
td	Output	Transmit data. 8-bit (1×) or 32-bit (4×) parallel data interface.
tc	Output	Transmit control. 1 bit for 1×; 4 bits for 4×.
tclk	Output	Transmit DDR center aligned clock.
phy_dis	Output	Transmit external transceiver PHY disable.

Table 3–5. External Transceiver Interface Signals (Part 2 of 2)

Signal	Direction	Description
rd	Input	Receive data. 8-bit (1×) or 32-bit (4×) parallel data interface.
rc	Input	Receive control. 1 bit for 1×; 4 bits for 4×.
rclk	Input	Recovered DDR center aligned clock. 1 bit for 1×; 4 bits for 4×.
rerr	Input	Receive error. 1 bit for 1×; 4 bits for 4×.

Table 3–6. Serial Layer 1 Global Signals

Signal	Direction	Description
clk	Input	Reference clock. See Table 3–1 and Table 3–2 for frequency requirements.
reset_n	Input	Active-low reset. reset_n can be asserted asynchronously but must be deasserted synchronously with clk.
rxclk	Output	Receive-side recovered clock.
txclk	Output	The internal system clock of the Physical layer. It is derived from the clk reference by division of one, two, or four, depending on the configuration of the MegaCore. Division is performed by a flip-flop-based circuit and thus does not require a dedicated PLL.
port_initialized	Output	This signal indicates that the serial RapidIO initialization sequence has completed successfully. This is a level signal asserted high while the initialization state machine is in the 1X_MODE or 4X_MODE state, as described in *paragraph 4.6 of Part VI of the RapidIO Specification*.

Table 3–7 lists the signals used in the layer 2.

Table 3–7. Physical Layer Slave Avalon-MM Interface Signals

Signal	Direction	Description
phy_mnt_s_clk	Input	Clock
phy_mnt_s_chipselect	Input	Slave chip select
phy_mnt_s_waitrequest	Output	Wait request
phy_mnt_s_read	Input	Read enable
phy_mnt_s_write	Input	Write enable
phy_mnt_s_address[16:0]	Input	Address bus
phy_mnt_s_writedata[31:0]	Input	Write data bus
phy_mnt_s_readdata[31:0]	Output	Read data bus

Tables 3–8 and 3–9 list the signals used in the layer 3 Atlantic receive and transmit interfaces.

Table 3–8. Serial Layer 3 Atlantic Receive Interface Signals

Signal	Direction	Description
arxclk	Input	Receive clock, which is greater than or equal to rx_clk.
arxreset_n	Input	Receive active-low reset. arxreset_n can be asserted asynchronously but should be deasserted synchronously to arxclk.
arxena	Input	Receive enable.
arxdav	Output	Receive data available. The arxdav signal is asserted when at least one complete packet is available to be read from the receive buffer. It is deasserted when the receive buffer does not have at least one complete packet available.
arxdat	Output	Receive data bus.
arxval	Output	Receive data valid.
arxsop	Output	Receive start of packet.
arxeop	Output	Receive end of packet.
arxmty	Output	Number of invalid bytes on arxdat.
arxerr	Output	Receive data error.
arxwlevel *(1)*	Output	Receive buffer write level (number of free 64-byte blocks in the receive buffer).

Note to Table 3–8:

(1) The following equation: log2(size of the receive buffer in bytes/64)+1 determines the number of bits. For example, a receive buffer size of 16 KBytes would give: log2(16×1024/64)+1= 9 bits (i.e., [8:0]).

Table 3–9. Serial Layer 3 Atlantic Transmit Interface Signals (Part 1 of 2)

Signal	Direction	Description
atxclk	Input	Transmit clock, which is greater than or equal to tx_clk.
atxreset_n	Input	Transmit active-low reset. atxreset_n can be asserted asynchronously but should be deasserted on the rising edge of atxclk.
atxena	Input	Transmit enable.
atxdav	Output	Transmit data available. atxdav is asserted when the transmit buffer has space to accept at least one maximum size packet (i.e., 276 bytes). It is deasserted when it does not have space to accept at least one maximum size packet.
atxdat	Input	Transmit data bus.
atxsop	Input	Transmit start of packet.

Table 3–9. Serial Layer 3 Atlantic Transmit Interface Signals (Part 2 of 2)		
Signal	**Direction**	**Description**
atxeop	Input	Transmit end of packet.
atxmty	Input	Number of invalid bytes on atxdat.
atxerr	Input	Transmit data error.
atxwlevel *(1)*	Output	Transmit buffer write level (number of free 64-byte blocks in the transmit buffer).
atxovf	Output	Transmit buffer overflow. If a new packet is started by asserting atxena and atxsop three or more atxclk clock cycles after atxdav is deasserted, atxovf is asserted and the packet is ignored.

Note to Table 3–9:
(1) The following equation: log2(size of the transmit buffer in bytes/64) determines the number of bits. For example, a transmit buffer size of 16 KBytes would give: log2(16x1024/64)= 8 bits (i.e., [7:0]).

Table 3–10 shows the packet and error monitoring signals for the serial RapidIO MegaCore function.

Table 3–10. Packet and Error Monitoring Signals			
Signal	**Direction**	**Clock Domain**	**Description**
packet_transmitted	Output	txclk	Pulsed high for one clock cycle when a packet's transmission completes normally.
packet_cancelled	Output	txclk	Pulsed high for one clock cycle when a packet's transmission is cancelled by sending a stomp, a restart-from-retry, or a link-request symbol.
packet_accepted	Output	txclk	Pulsed high for one clock cycle when a packet-accepted symbol is being transmitted.
packet_retry	Output	txclk	Pulsed high for one clock cycle when a packet-retry symbol is being transmitted.
packet_not_accepted	Output	txclk	Pulsed high for one clock cycle when a packet-not-accepted symbol is being transmitted.
packet_crc_error	Output	rxclk	Pulsed high for one clock cycle when a CRC error is detected in a received packet.
symbol_error	Output	rxclk	Pulsed high for one clock cycle when a corrupted symbol is received.
char_err	Output	rxclk	Pulsed for one clock cycle when an invalid character or a valid but illegal character is detected.

Table 3–11 shows the multicast event signal which is toggled when it receives a Multicast Event control symbol.

Table 3–11. Multicast Event Signal

Signal	Direction	Clock Domain	Description
multicast_event_rx	Output	rxclk	A top-level output port that is toggled for one clock cycle when a Multicast Event control symbol is received.

Table 3–12 shows the receive priority retry threshold-related signals for the serial RapidIO MegaCore function.

Table 3–12. Priority Retry Threshold-Related Signals

Signal	Direction	Description
buf_av0	Output	Buffer available signal; relates to priority retry threshold 0.
buf_av1	Output	Buffer available signal; relates to priority retry threshold 1.
buf_av2	Output	Buffer available signal; relates to priority retry threshold 2.
buf_av3	Output	Buffer available signal; relates to priority retry threshold 3.

Table 3–13 lists the transceiver signals, which exist when a device such as Arria GX or Stratix II GX is used.

Table 3–13. Transceiver Signals

Signal	Direction	Description
cal_blk_clk	Input	The Stratix II GX transceiver's on-chip termination resistors in the transceiver channels are calibrated by a single calibration block. This circuitry requires a calibration clock. The frequency range of the cal_blk_clk is 10 Mhz to 125 Mhz. For more information, refer to Chapter 2 of *Stratix II GX Device Handbook*, Volume 2. If external termination is being used, this signal can be tied low.
reconfig_clk	Input	Reference clock for the dynamic reconfiguration controller. The frequency range for this clock is 2.5 Mhz to 50 Mhz. If you choose to use a ALT2GXB_RECONFIG block in your design to dynamically control the ALT2GXB, then this clock is required by the ALT2GXB_RECONFIG and the RapidIO MegaCore function.

Table 3–13. Transceiver Signals		
Signal	**Direction**	**Description**
reconfig_togxb	Input	Driven from an external ALT2GXB_RECONFIG block. Supports the selection of multiple transceiver channels for dynamic reconfiguration. If no external ALT2GXB_RECONFIG block is used, then you can tie this bus to ground.
reconfig_fromgxb	Output	Driven to an external ALT2GXB_RECONFIG block. The bus identifies the transceiver channel whose settings are being transmitted to the ALT2GXB_RECONFIG block. If no external ALT2GXB_RECONFIG block is used, then this bus can be left untied.

Table 3–14 shows the register-related signals for the serial RapidIO MegaCore function.

Table 3–14. Register-Related Signals		
Signal	**Direction**	**Description**
ef_ptr[15:0]	Input	Most significant bits [31:16] of the PHEAD0 register.
port_response_timeout[23:0]	Output	Most significant bits [31:8] of PRTCTRL register.

Software Interface

All addresses access 32-bit registers and are shown as hexadecimal values. The access addresses for each register increment by units of 4. Table 3–15 shows the memory map for the serial RapidIO Physical layer function.

Table 3–15. Master Memory Map		
Address	**Name**	**Description**
'h100	PHEAD0	Port Maintenance Block Header 0
'h104	PHEAD1	Port Maintenance Block Header 1
'h120	PLTCTRL	Port Link Time-out Control CSR
'h124	PRTCTRL	Port Response Time-out Control CSR
'h13C	PGCTRL	Port General Control CSR
'h158	ERRSTAT	Port 0 Error and Status CSR
'h15C	PCTRL0	Port 0 Control CSR

Table 3–16 lists the access codes used to describe the type of register bits.

Table 3–16. Register Access Codes

Code	Description
RW	Read/write
RO	Read-only
RW1C	Read/write 1 to clear
RW0S	Read/write 0 to set
RTC	Read to clear
RTS	Read to set
RTCW	Read to clear/write
RTSW	Read to set/write
RWTC	Read/write any value to clear
RWTS	Read/write any value to set
RWSC	Read/write self-clearing
RWSS	Read/write self-setting
UR0	Unused bits/read as 0
UR1	Unused bits/read as 1

Physical Layer Registers

Tables 3–17 through 3–23 describe the registers for physical layer of the serial RapidIO MegaCore function. The offset values are as defined by the RapidIO standard.

Table 3–17. PHEAD0—Port Maintenance Block Header 0—'h100

Field	Bits	Access	Function	Default
EF_PTR	31:16	RO	Hard-wired pointer to the next block in the data structure, if one exists. The value is set from the ef_ptr input port.	ef_ptr
EF_ID	15:0	RO	Hard-wired extended features ID.	'h0004

Table 3–18. PHEAD1—Port Maintenance Block Header 1—'h104

Field	Bits	Access	Function	Default
RSRV	31:0	UR0	Reserved.	0

Table 3–19. PLTCTRL—Port Link Time-Out Control CSR—'h120

Field	Bits	Access	Function	Default
VALUE	31:8	RW	Time-out interval value.	'hffffff
RSRV	7:0	UR0	Reserved.	0

Table 3–20. PRTCTRL—Port Response Time-Out Control CSR—'h124

Field	Bits	Access	Function	Default
VALUE	31:8	RW	Time-out internal value. PHY layer-only variations: This value is not used by the RapidIO MegaCore function. The contents of this register are brought out to the port_response_timeout output signal. Variations using logical layers: The duration of the port response timeout is equal to the 24-bit number contained in this field, multiplied by the sysclk_timeout_prescaler, multiplied by the period of the sysclk. Note: avoid timeouts less than 24'h00_0010 as they may not be reliable. The sysclk_timeout_prescalar value is set by the MegaWizard based on the clk clock's period such that the maximum time value 24'hFF_FFFF corresponds to approximately 4.5 seconds.	'hffffff
RSRV	7:0	UR0	Reserved.	0

Table 3–21. PGCTRL—Port General Control CSR—'h13C (Part 1 of 2)

Field	Bits	Access	Function	Default
HOST	31	RW	A host device is a device that is responsible for system exploration, initialization, and maintenance. Agent or slave devices are typically initialized by host devices. 'b0—agent or slave device. 'b1—host device.	0
ENA	30	RW	The master enable bit controls whether or not a device is allowed to issue requests into the system. If the master enable is not set, the device may only respond to requests. 'b0—processing element cannot issue requests. 'b1—processing element can issue requests.	0

Table 3–21. PGCTRL—Port General Control CSR—'h13C (Part 2 of 2)

Field	Bits	Access	Function	Default
DISCOVERED	29	RW	This device has been located by the processing element responsible for system configuration. 'b0—The device has not been previously discovered. 'b1—The device has been discovered by another processing element.	0
RSRV	28:0	UR0	Reserved.	0

Table 3–22. ERRSTAT—Port 0 Error and Status CSR—'h158 (Part 1 of 2)

Field	Bits	Access	Function	Default
RSRV	31:21	UR0	Reserved.	0
OUT_RTY_ENC	20	RW1C	Output port has encountered a retry condition. This bit is set when bit 18 is set.	0
OUT_RETRIED	19	RO	Output port has received a packet-retry control symbol and cannot make forward progress. This bit is set when bit 18 is set. This bit is cleared when a packet-accepted or a packet-not-accepted control symbol is received.	0
OUT_RTY_STOP	18	RO	Output port has received a packet-retry control symbol and is in the output retry-stopped state.	0
OUT_ERR_ENC	17	RW1C	Output port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 16 is set.	0
OUT_ERR_STOP	16	RO	Output port is in the output error-stopped state.	0
RSRV1	15:11	UR0	Reserved.	0
IN_RTY_STOP	10	RO	Input port is in the input retry-stopped state.	0
IN_ERR_ENC	9	RW1C	Input port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 8 is set.	0
IN_ERR_STOP	8	RO	Input port is in the input error-stopped state.	0
RSRV2	7:5	UR0	Reserved.	0
PWRITE_PEND	4	UR0	This register is not implemented and is reserved. It is always set to zero.	0
RSRV3	3	UR0	Reserved.	0
PORT_ERR	2	RW1C	Input or output port has encountered an error from which hardware was unable to recover.	0

Table 3–22. ERRSTAT—Port 0 Error and Status CSR—'h158 (Part 2 of 2)

Field	Bits	Access	Function	Default
PORT_OK	1	RO	Input and output ports are initialized and the port is exchanging error-free control symbols with the adjacent device.	0
PORT_UNINIT	0	RO	Input and output ports are not initialized. This bit and bit 1 are mutually exclusive.	'b1

Table 3–23. PCTRL0—Port 0 Control CSR—'h15C (Part 1 of 3)

Field	Bits	Access	Function	Default
PORT_WIDTH *(1)*	31:30	R0	Hardware width of the port: 'b00—Single-lane port. 'b01—Four-lane port. 'b10–'b11—Reserved.	0
INIT_WIDTH *(1)*	29:27	R0	Width of the ports after initialized: 'b000—Single lane port, lane 0. 'b001—Single lane port, lane 2. 'b010—Four lane port. 'b011–'b111—Reserved.	
PWIDTH_OVRIDE *(1)*	26:24	R0	Soft port configuration to override the hardware size: 'b000—No override. 'b001—Reserved. 'b010—Force single lane, lane 0. 'b011—Force single lane, lane 2. 'b100–'b111—Reserved.	0

Table 3–23. PCTRL0—Port 0 Control CSR—'h15C (Part 2 of 3)

Field	Bits	Access	Function	Default
PORT_DIS	23	RW	Port disable: 'b0—port receivers/drivers are enabled. 'b1—port receivers are disabled, causing the drivers to send out idles. ● When this bit transitions from one to zero, the initialization state machines' force_reinit state variable is asserted, as described in *Part 6: Physical Layer 1×/4× LP Serial Physical Layer Specification Revision 1.3*, paragraphs 4.7.3.5 and 4.7.3.6. In turn, this assertion causes the port to enter the SILENT state and to attempt to reinitialize the link. ● When reception is disabled, the input buffers are kept empty until this bit is cleared. ● When PORT_DIS is asserted and the drivers are disabled, the transmit buffer are reset and kept empty until this bit is cleared, any previously stored packets are lost and any attempt to write a packet to the atx Atlantic interface is ignored by the Physical layer, new packets are NOT stored for later transmission.The logical layers are responsible for re-transmitting any lost packets that require a response.	0
OUT_PENA	22	RW	Output port transmit enable: 'b0—port is stopped and not enabled to issue any packets except to route or respond to I/O logical maintenance packets, depending upon the functionality of the processing element. Control symbols are not affected and are sent normally. 'b1—port is enabled to issue any packets.	1
IN_PENA	21	RW	Input port receive enable: 'b0—port is stopped and only enabled to respond I/O logical maintenance packets, depending upon the functionality of the processing element. Other packets generate packet-not-accepted control symbols to force an error condition to be signalled by the sending device. Control symbols are not affected and are received and handled normally. 'b1—port is enabled to respond to any packet.	1
ERR_CHK_DIS	20	RW	This bit disables all RapidIO transmission error checking: 'b0—Error checking and recovery is enabled. 'b1—Error checking and recovery is disabled. Device behavior when error checking and recovery is disabled and an error condition occurs is undefined.	0
MULTICAST	19	RW	Send incoming multicast-event control to this port (multiple port devices only) **(Not currently implemented in the core.)**	0

Table 3–23. PCTRL0—Port 0 Control CSR—'h15C (Part 3 of 3)

Field	Bits	Access	Function	Default
RSRV2	18:1	UR0	Reserved.	0
PORT_TYPE	0	UR1	This indicates the port-type, parallel or serial. 'b0—Parallel port. 'b1—Serial port.	1

Notes to (3–23)

(1) The 4x Serial RapidIO variations of the RapidIO MegaCore IP function do not support falling back to 1x mode, so PWIDTH_OVRIDE will always read back as 'b000.

MegaCore Verification

Before releasing a version of the RapidIO MegaCore function, Altera runs a comprehensive regression test, which executes the wizard to create the instance files. These files are tested in simulation and hardware to confirm functionality.

The RapidIO MegaCore function was also subjected to interoperability testing. Interoperability tests verify the performance of the MegaCore function in real-life applications, and ensure compliance with ASSP devices.

Simulation Testing

The RapidIO core is verified using industry-standard simulators ModelSim, and VCS in combination with Vera. The test suite contains testbenches that use the RapidIO bus functional model (BFM) from the RapidIO Trade Association to verify the functionality of the IP core.

The regression suite tests various functionalities, including:

- Link initialization
- Packet format
- Packet priority
- Error handling
- Throughput
- Flow control

Constrained random techniques are used to generate appropriate stimulus for the functional verification of the IP core. Functional coverage metrics are used to measure the quality of the random stimulus, and to ensure that all important features have been verified.

Hardware Testing

The RapidIO MegaCore function is tested and verified in hardware for different platforms and environments.

The hardware tests cover serial ×1 and ×4 variations running at 1.25 and 3.125 gigabits per second (Gbps), and processing the following traffic types:

- NReads of various size payloads—4 bytes to 256 bytes—millions of packets
- NWrites of various size payloads—4 bytes to 256 bytes—millions of packets
- NWrite_R of a few different size packets—hundreds of packets
- PortWrites—hundreds of packets
- Maintenance—hundreds to thousands of packets

The hardware tests also cover the following control symbol types: Packet Accepted, Packet Retry, Packet Not Accepted, Packet Control Symbol, and Link Maintenance Control Symbol.

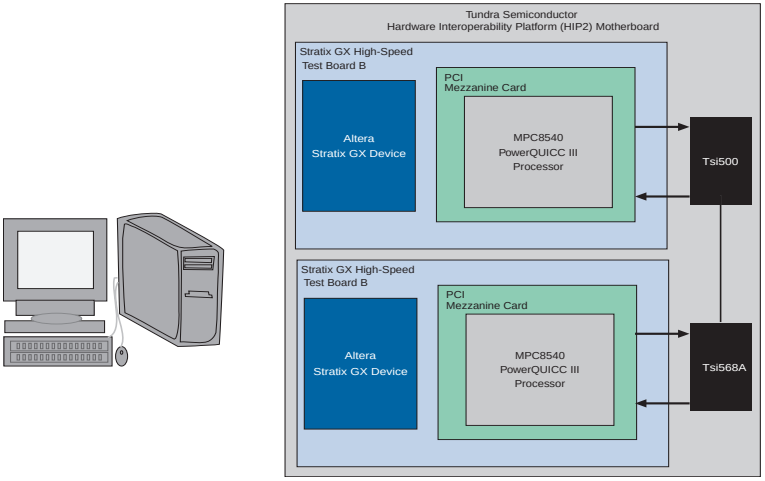
Interoperability Testing

The interoperability tests performed on the RapidIO MegaCore function certify that the serial RapidIO MegaCore function has been tested for the following functionality:

- Compatibility with commercial RapidIO devices
- Device compatibility with the Stratix® GX/Stratix II GX family of devices
- Endurance
- Operation at speed

The interoperability tests were conducted on a hardware interoperability platform (HIP) provided by Tundra Semiconductor. Two variations of the HIP were used: one was used to test the Altera RapidIO MegaCore function against itself and the other was used to test the Altera RapidIO MegaCore function's interoperability with the Tsi568A switch. The RapidIO MegaCore functions were implemented in Stratix devices; each device on its own Altera Stratix RapidIO test board. The RapidIO MegaCore functions were interconnected by the HIP motherboard, connected through intervening RapidIO Tundra Tsi568A switches. Figure 3–9 on page 3–40 shows a block diagram of the test environment.

Figure 3–9. Interoperability Test Block Diagram



The Serial Physical layer of the RapidIO MegaCore function was also tested on the Stratix II GX device along with the Transport and Logical layers. For more information, see "Interoperability Testing" on page 4–118.

4. Variations with Physical, Transport, and Logical Layers

Functional Description

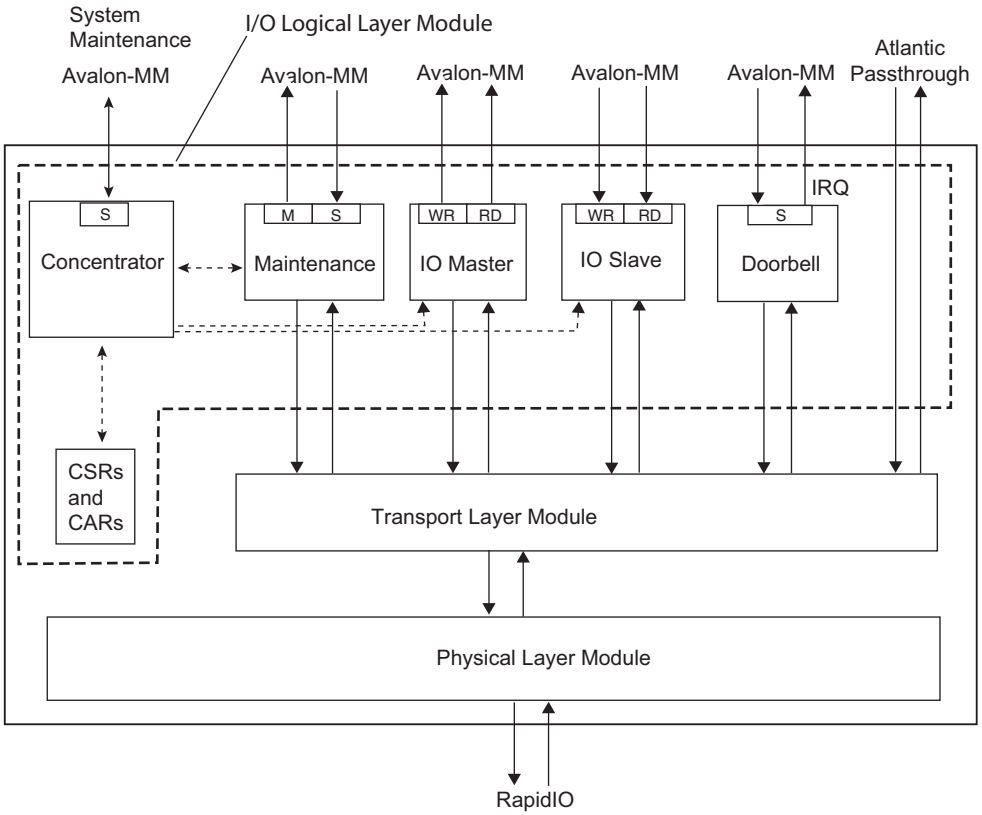
Information in this chapter is relevant only if your custom RapidIO® MegaCore® function variation contains a Transport layer.

In the MegaWizard Plug-In Manager design flow, you add a Transport layer by turning on the **Transport Layer** option on the **Transport and Maintenance** page during parameterization. In the SOPC Build Design flow, the Transport layer is automatically enabled.

This chapter describes the features of the Transport and Logical layers, and how they integrate and interact with the existing Physical layer to create the three-layer RapidIO MegaCore function. Figure 4–1 shows an example high-level block diagram, with the following modules and layers:

- A Concentrator module, used to consolidate register access.
- Maintenance module used for initiating and terminating maintenance transactions
- Input/Output Master module, used for initiating and terminating NREADs, NWRITEs, SWRITEs, NWRITEs_R
- A doorbell module for transacting RapidIO doorbell messages.
- An Avalon-ST pass-through port that provides direct access to the Transport layer offering the user access to a complete or unparsed RapidIO packet.
- Transport layer module, with at least one Logical layer module or the the Avalon-ST pass-through port. used for processing the TRansport layer fields of a RapidIO packet.
- A RapidIO Physical layer module which implements the complete RapidIO to Physical layer specification 1.3. See Chapter 3, Physical Layer—Serial Specifications for details.

Figure 4–1. RapidIO MegaCore Functional Block Diagram Note (1)



Note to Figure 4–1:
(1) The dashed lines represent internal Avalon-MM interfaces.

Interfaces

The user interface to the RapidIO MegaCore function is supported by the following interfaces:

- Avalon Memory-Mapped (Avalon-MM) Interface
- Avalon Streaming (Avalon-ST) Interface

Avalon-MM Interface

The Avalon-MM interface provides user access to the modules in the Logicial layer:

- Concentrator
- Maintenance
- I/O Master
- Input/Output Slave
- Doorbell

Both slave and master variations are used. The interface supports a 32-or 64-bit datapath. width and 32-bit wide address. The interface can support simple single read/write transactions as well as burst reads/writes. Subsequent sections describe these modules and the details of how they support the Avalon -MM interface.

For more information on this interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Avalon- ST Interface

The Avalon-ST interface provides user access to the Transport layer. The Avalon-ST pass-through port interface is a full-duplex synchronous protocol that supports 32- or 64-bit datapath. RapidIO packets received that specify an FTYPE not supported by this MegaCore function or that have a DestinationID that does not match the Base Device ID of this core, will be routed to the RX port of the Avalon-ST pass-through port, if it is enabled.

You enable the Avalon-ST pass-through port in the MegaWizard Plug-In Manager flow by turning on the Avalon-ST pass-through port on the Transport and Maintenance page of the MegaWizard interface during parameterization of the MegaCore function. For additional control, you also can turn on the **Source Operation** and/or **Destination Operation** to allow the local RapidIO endpoint to generate (**Source Operation**) or transfer (**Destination Operation**) data messages through the Avalon-ST pass-through port.

For more information, refer to the section, "Avalon-ST Pass-Through Interface" on page 4–45.

The Avalon-ST pass-through port is not supported in the SOPC Builder flow.

For more information on this interface, refer to the *Avalon Streaming Interface Specification*.

Clock & Reset

This section describes the clock and reset signals for variations with Physical, Transport, and Logical layer modules.

Clock

Variations with Physical, Transport, and Logical layer modules have two clock inputs. The `clk` clock is the reference clock for the Physical layer. Its frequency is determined by the desired baud rate.

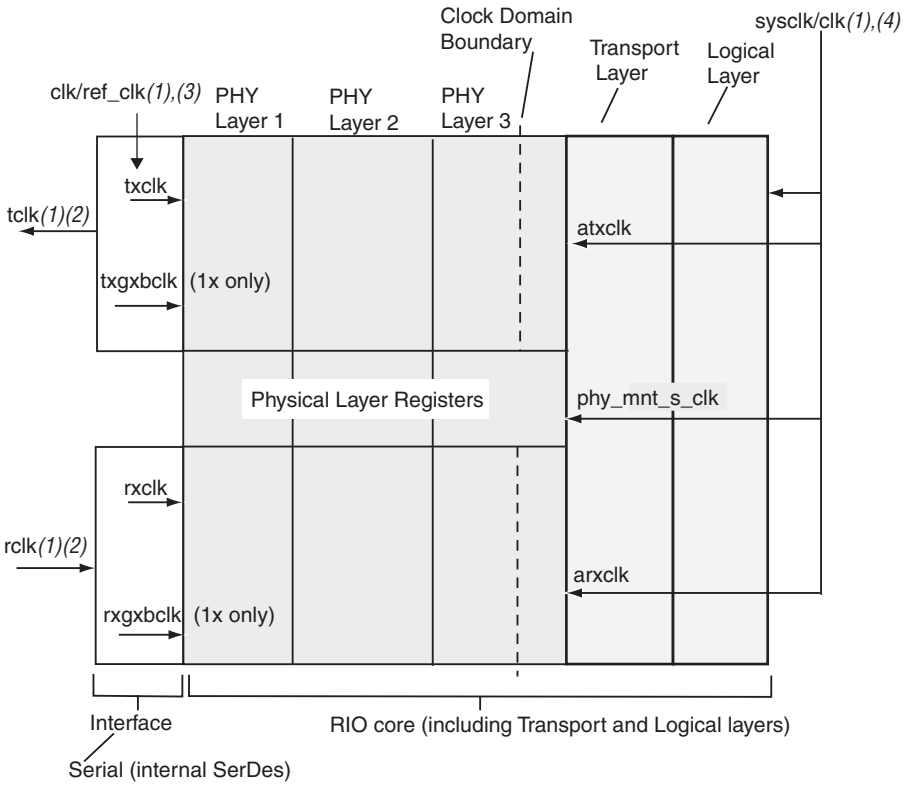
The `sysck` clock drives the transport and logical layer modules, its frequency is nominally the same frequency as `clk` but can differ by up to ±50%.

Clock domain crossing between the sysclk clock domain and the physical layer's clock domains is done in the physical layer's buffers.In the Maintenance module, clock domain crossing for a local processor with a different clock is handled by an external Avalon-MM system interconnect fabric.

The system interconnect fabric manages clock domain crossing if some of the components of the Avalon system run off of a different clock. For optimal throughput, it is recommended to run all the Avalon components in the datapath on the same clock.

All of the Avalon-MM clock inputs for the Logical layer modules must be connected to the same clock source as `sysclk` for Non SOPC Builder mode or `clk` for SOPC Builder mode. See Figure 4–2 for a block diagram of the clock structure of variations with Physical, Transport and Logical layer clock structure. See Table 4–1 for information on clock rates used.

Figure 4–2. Clock Domains



Clock descriptions:

`txgxbclk`: Transmitter transceiver clock
`rxgxbclk`: Receiver transceiver clock
`txclk`: Transmitter internal global clock (same as system clock)
`rxclk`: Receiver internal global clock
`atxclk`: Atlantic interface clock—greater than, or equal to txclk
`arxclk`: Atlantic interface clock—greater than, or equal to rxclk
`phy_mnt_s_clk`: Avalon-MM interface clock for register access
`rclk`: XGMII receive clock
`tclk`: XGMII transmit clock

Notes to Figure 4–2:

(1) Input clocks (user supplied)
(2) Clocks `rclk` and `tclk` exist only when using an external SerDes with XGMII
(3) `clk` exists for nonSOPC Builder mode. `ref_clk` exists for SOPC Builder mode.
(4) `sysclk` exists for nonSOPC Builder modes. `clk` exists for SOPC Builder mode.

Table 4–1. Clock Frequencies Used With Serial Transport and Logical Layers

Baud Rate (GBaud)	1x Serial/32-bit				x4 Serial/64-bit
	Stratix GX transceiver		Arria GX, Stratix IIGX, or XGMII		Arria GX, Stratix GX, Stratix IIGX, XGMII
	clk (MHz)	sysclk *(3)* (MHz)	clk *(4)*, tclk, *(2)* rclk (MHz)	sysclk *(3)*, txclk, rxclk (MHz)	clk, txclk, sysclk *(1)*, *(3)*, rclk, tclk *(2)* (MHz)
1.25	125 MHz	31.25 MHz	62.5 MHz	31.25 MHz	62.5 MHz
2.5	125 MHz	62.5 MHz	125 MHz	62.5 MHz	125 MHz
3.125	156.25 MHz	78.25 MHz	156.25 MHz *(5)*	78.25 MHz *(5)*	156.25 MHz *(5)*

Notes for Table 4–1:
(1) Nominal frequency shown for `sysclock`.
(2) `tclk` and `rclk` are for external transceivers with XGMII interfaces.
(3) `sysclk` exists in nonSOPC Builder mode; `clk` exists in SOPC Builder mode.
(4) `clk` exists in nonSOPC Builder mode; `ref_clk` exists in SOPC Builder mode
(5) Arria GX does not support 3.125 Gbaud.

Reset

The transport and Logical layer modules use a single main system, active-low reset input signal (`reset_n`).

The reset input signal can be asserted asynchronously, but must last at least one `sysclk` clock period and be deasserted synchronously with the rising edge of `sysclk`. The assertion of `reset_n` causes the whole module to reset. While held in reset, all outputs are driven low. Upon coming out of reset, all buffers are empty. See "Software Interface" on page 4–86 for the reset value of the registers.

Transport Layer Module

The Transport layer module is an optional module of the RapidIO MegaCore function, that is intended for use in an end-point processing element with at least one Logical layer module or the Avalon-ST pass-through port.

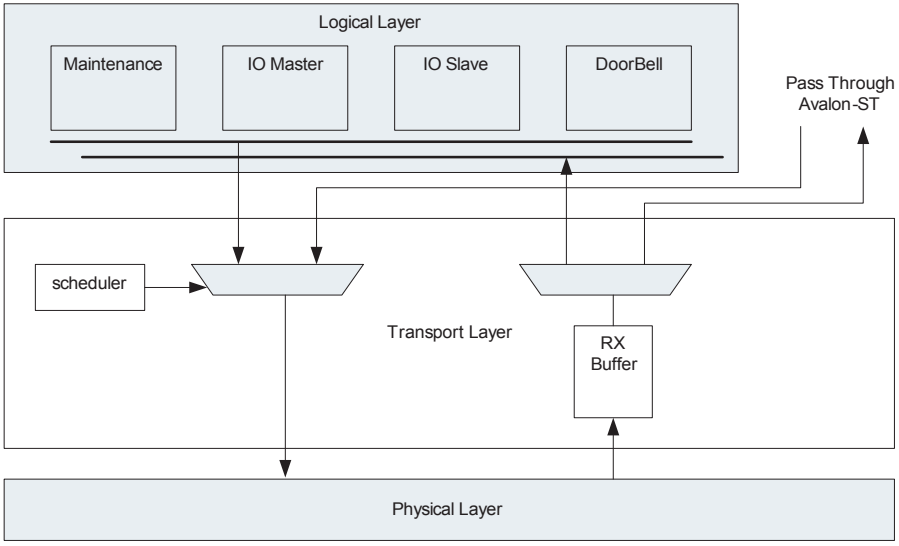
When you create your custom RapidIO MegaCore function variation in the MegaWizard interface (see "Parameterize" on page 2–7), you can select **No Transport Layer** or **Transport Layer**.

If you select **No Transport Layer**, you choose to use a Physical layer-only variation. If you create a variation without a Transport layer, refer to Chapter 3, Physical Layer—Serial Specifications for more information instead of this chapter.

If you select **Transport Layer**, you also have the choice to turn on the Avalon-ST pass-through port parameter. If you turn on this parameter, the Transport layer routes all unrecognized packets to the Avalon-ST pass-through port. Unrecognized packets contain `ftypes` for Logical layers not enabled in this MegaCore function, or destination IDs not assigned to this endpoint.

The Transport layer module is divided into receiver and transmitter submodules. Figure 4–3 on page 4–7 shows a block diagram of the Transport layer module.

Figure 4–3. Transport Layer Block Diagram



Receiver

On the receive side, the Transport layer module receives packets from the Physical layer. The packet is sent to one of the Logical layer modules or Avalon-ST pass-through port according to the packet format type (`ftype`) and transaction type (`ttype`) header fields. The

receiver reads the `ftype` from the header of each packet and routes packets to respective Logical layer modules. Packets with unsupported `ftype` and invalid `ttype` fields or invalid destination IDs are routed to the Avalon-ST pass-through port if it is present, otherwise the packets are dropped. Packets that are marked as errored by the Physical layer by the assertion of `arxerr` (for example, packets with a CRC error or that were stomped) are filtered and dropped from the stream being sent to the Logical layer modules.

Transmitter

On the transmit side, the Transport layer module uses a scheduler to select which Logical layer module is to transmit packets. The scheduler is round-robin based by default. For the round-robin based scheduler, the Transport layer polls the various Logical layer modules in turn to determine whether a packet is available. When a packet is available, the Transport layer transmits the whole packet, and then continues polling the next logical modules.

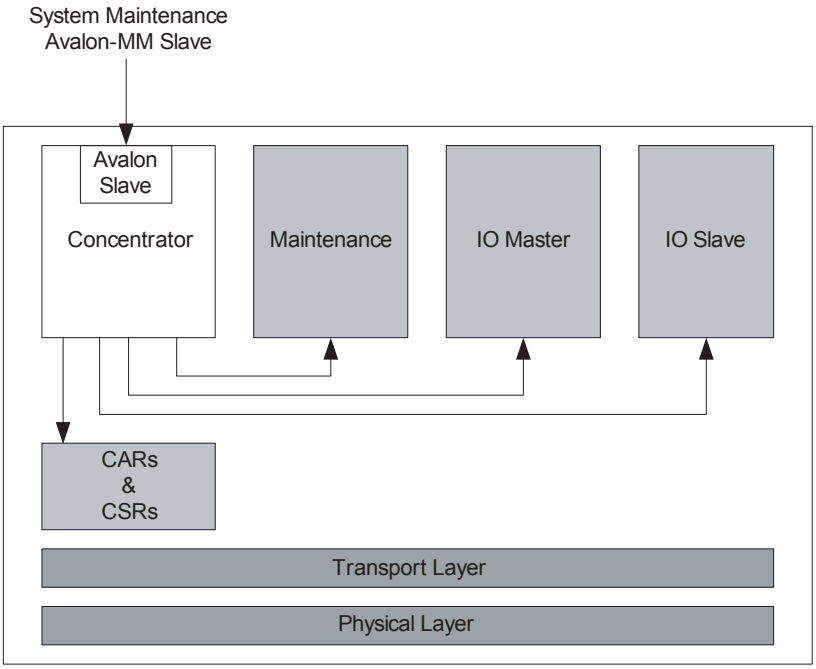
Under normal conditions, all errored packets are dropped in the Logical layer modules prior to the transfer to the Transport layer. However, the Pass-through port's `gen_tx_error` signal can be used to terminate any errored packet that is missed. In that case, the `gen_tx_endofpacket` signal should also be asserted.The `gen_tx_error` signal is useful when a user-defined Logical layer module is present at the Avalon-ST pass-through port because it can be used to abort the packet transmission.

For more information on the Transport layer, refer to *Part 3: Common Transport Specification* of the *RapidIO Interconnect Specification,* Revision 1.3.

Concentrator Register Module

The Concentrator module provides access to all of the configuration registers in the RapidIO MegaCore function, including the CARs and CSRs. The configuration registers are distributed among the implemented Logical layer modules. Figure 4–4 shows how the Concentrator module provides access to all the registers which are implemented in different Logical layer modules. The Concentrator module automatically is implemented when you include the Transport layer.

Figure 4–4. Concentrator Module Provides Configuration Register Access



The Concentrator module provides an Avalon-MM Slave interface, which lets you access the RapidIO MegaCore function register set. The interface supports simple reads and writes with variable latency. Accesses are to 32-bit words addressed by a 17-bit wide byte address. When accessed, the lower 2 bits of the address are ignored, which aligns the transactions to 4-byte words. The interface supports an interrupt line, `sys_mnt_s_irq`. When enabled, the following interrupts assert the `sys_mnt_s_irq` signal.

- Received Port-Write
- IO Read Out of Bounds
- IO Write Out of Bounds
- Invalid Write
- Invalid Write Burtscount

For details on these and other interrupts, see Table 4–75 and Table 4–76

Figure 4–5 and Figure 4–6 show different ways to access the RapidIO registers.

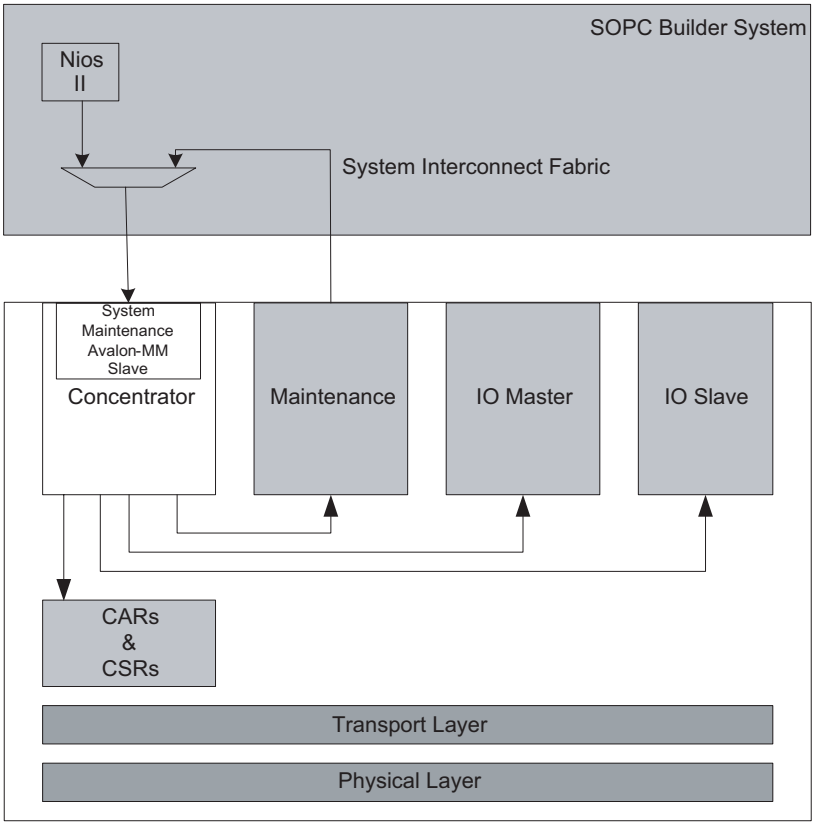
There are two ways to access registers by local host:

- SOPC Builder system interconnect fabric
- Custom logic

A local host can access the RapidIO registers from an SOPC Builder system as illustrated in Figure 4–5. In this figure, NIOS II is part of the SOPC System and is configured as an Avalon-MM Master that accesses the RapidIO MegaCore function registers through the System Maintenance Avalon-MM Slave.

See the *SOPC Builder* chapter in Volume 4 of the *Quartus II Handbook* for implementation details.

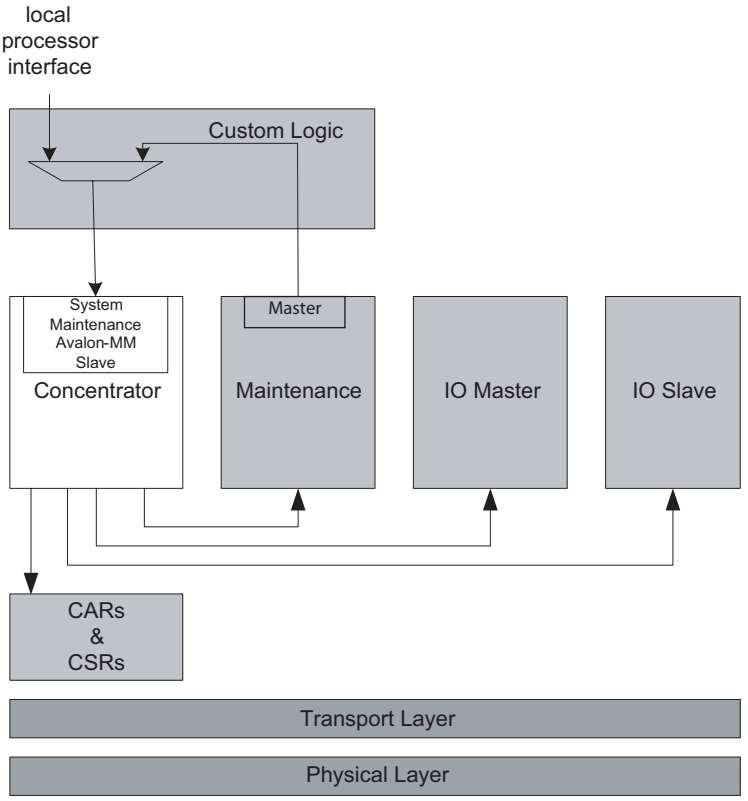
Figure 4–5. Local Host Accesses RapidIO Registers from an SOPC Builder System



Alternately, you can implement custom logic to access the RapidIO registers as shown in Figure 4–6.

A remote host can access the RapidIO registers by sending Maintenance transactions targeted to this local RapidIO MegaCore function. The Maintenance transactions are processed by the Maintenance module. If the transaction is a Read or Write, the operation is presented on the Maintenance Avalon-MM Master interface. This interface must be routed to the System Maintenance Avalon-MM Slave interface. This routing can be done with an SOPC Builder System shown by the routing to the Concentrator's system maintenance Avalon-MM Slave in Figure 4–5. If you do not use an SOPC Builder System, you can create custom logic as shown in Figure 4–6.

Figure 4–6. Custom Logic Accesses RapidIO MegaCore Function Registers



All of the registers described in the section "Software Interface" on page 4–86, with the exception of the Doorbell Registers, can be accessed by using the System Maintenance Avalon-MM Slave interface.

Maintenance Module

The Maintenance module is an optional component of the Input/Output Logical layer. The Maintenance module processes Maintenance Type transactions, including the following:

- Type 8 – Maintenance Reads and Writes
- Type 8 – Port Write Packets

When you create your custom RapidIO MegaCore function variation in the MegaWizard® interface (see the section, "Parameterize" on page 2–7), you have four choices for this module:

- **Avalon-MM Master and Slave**
- **Avalon-MM Master**
- **Avalon-MM Slave**
- **None**

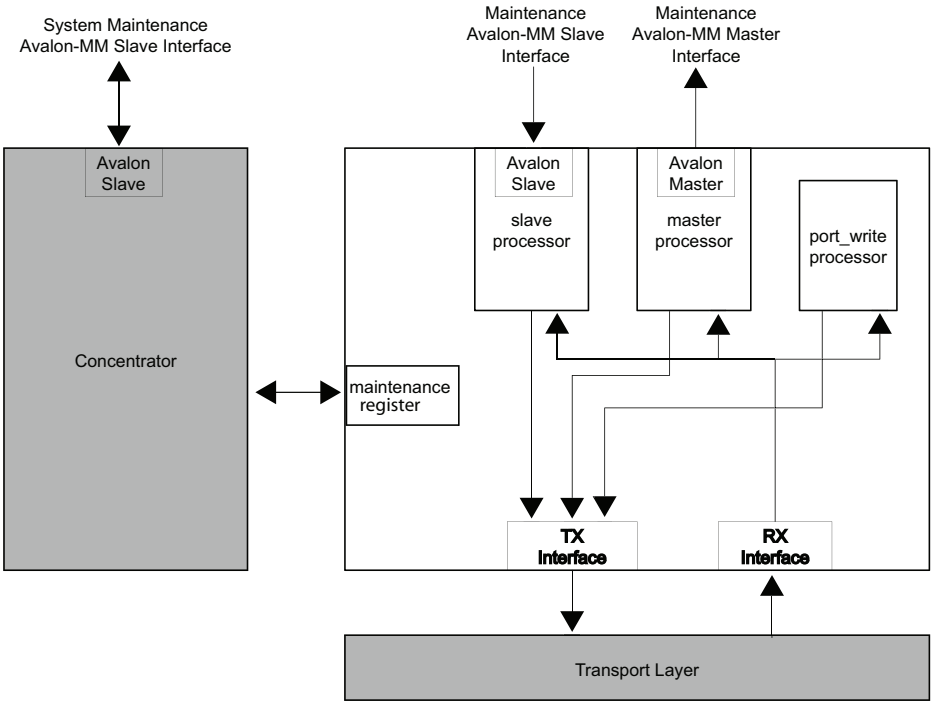
Selecting the **Avalon-MM Master and Slave** option, allows your MegaCore function to initiate and terminate Maintenance Type transactions. Selecting only the **Avalon-MM Master** restricts your MegaCore function to only terminate Maintenance Type transactions. Selecting only the **Avalon-MM Slave** also restricts your MegaCore function to initiate only Maintenance Type transactions. If you select neither, then your MegaCore function can neither initiate nor terminate Maintenance Type transactions.

If you add this module to your variation and select an Avalon-MM Slave interface, you must also choose a number of transmit address translation windows. A minimum of one window is required and a maximum of 16 windows are available.

Figure 4–7 shows a high level block diagram of the Maintenance module and the interfaces to other supporting modules. The Maintenance module can be segmented into four major submodules: maintenance_register, maintenance_slave_processor, maintenance_master_processor, and port_write_processor. The following interfaces are supported:

- Avalon-MM Slave Interface – User exposed interface
- Avalon-MM Master Interface – User exposed interface
- TX Interface – Internal used to communicate with the Transport layer
- RX Interface – Internal used to communicate with the Transport layer
- Register Interface – Internal used to communicate with the Concentrator Module

Figure 4–7. Maintenance Module Block Diagram



Maintenance Register

The maintenance register module implements all of the control and status registers needed by this module to perform its functions. These include registers described in Table 4–55 through Table 4–67 in the section "Software Interface" on page 4–86. These registers are accessible to you through the Concentrator module. A read or write request is presented across the System Maintenance Avalon-MM interface. The request is decoded by the Concentrator module and is sent to the maintenance register submodule. For a write request, the maintenance register submodule writes the addressed register and for a read request, the register is read and the value returned to the Concentrator module. The Concentrator module then sends the read value to the user across the System Maintenance Avalon-MM interface.

Maintenance Slave Processor

The Maintenance Slave Processor module does the following.

- For an Avalon Read, composes the RapidIO Logical Header Fields of a Maintenance Read transaction.
- For an Avalon Write, composes the RapidIO Logical Header Fields of a Maintenance Write transaction.
- Maintain status related to the composed Maintenance Packet.
- Presents the composed Maintenance Packet to the Transport layer for transmission.

The Avalon-MM Slave interface allows you to initiate a Maintenance Read or Write operation. The Avalon-MM Slave interface supports the following Avalon transfers.

- Single Slave Write Transfer with Variable Wait-States
- Pipelined Read Transfers with Variable Latency

At any time, there can be a maximum of 64 outstanding maintenance requests, either maintenance reads, maintenance writes, or port write requests.

Refer to the *Avalon Memory-Mapped Interface Specification* for more details on the supported transfers.

Figure 4–8 shows the signal relationships for 4 write transfers on the Avalon-MM slave interface.

Figure 4–8. Write Transfers on the Avalon-MM Slave Interface

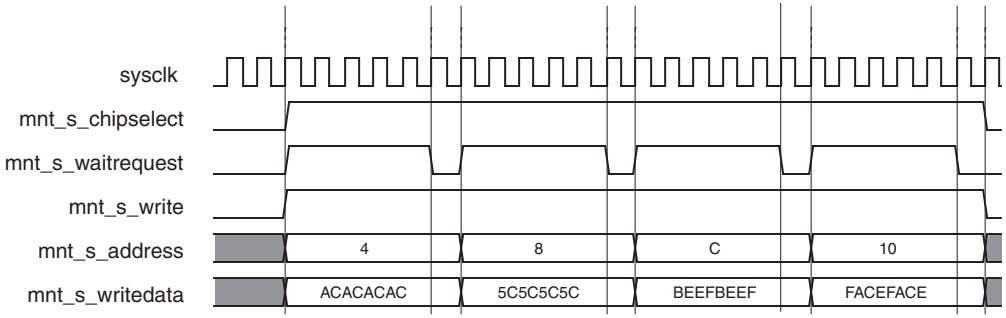
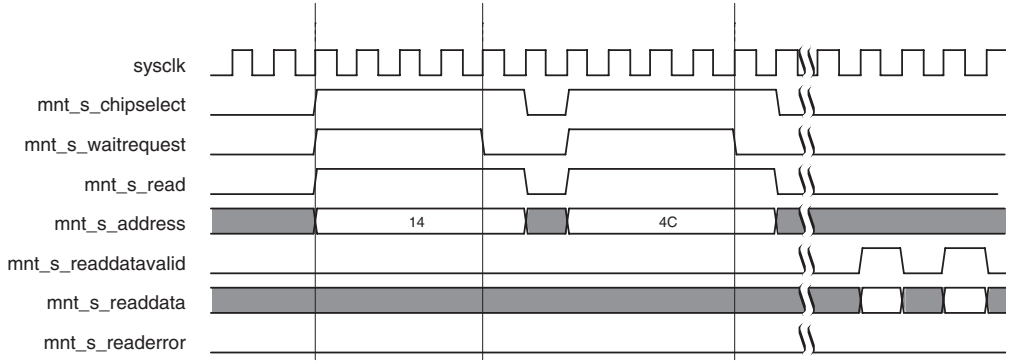


Figure 4–9 shows the signal relationships for 4 read transfers on the Avalon-MM Slave interface.

Figure 4–9. Read Transfers on the Avalon-MM Slave Interface



Reads and Writes on the Avalon-MM Slave Interface are converted into RapidIO Maintenance Reads and Writes as follows. The following are the fields of a Maintenance Type packet that are assigned by the Maintenance Module.

- PRIO
- FTYPE
- DEST_ID
- SRC_ID
- TTYPE
- RDSIZE/WRSIZE
- SOURCE_TID
- HOP_COUNT
- CONFIG_OFFSET
- WDPTR

The FTYPE field is assigned a value of 4'b1000. The TTYPE field is assigned a value of 4'b0000 for Reads and a value of 4'b0001 for Writes. The RDSIZE/WRSIZE field is fixed at 4'b1000, since only 4-byte reads and writes are supported.

The CONFIG_OFFSET is generated by using the values programmed in the TX Maintenance Address Translation Windows described in tables Table 4–68 through Table 4–74. You determine the number of mapping windows supported when you generate the MegaCore function using the MegaWizard interface. Up to 16 maximum windows are allowed.

Each window is enabled if the window enable (WEN) bit of the Tx Maintenance Window n Mask register is set. Each window is defined by three registers:

- A base register: Tx Maintenance Mapping Window n Base
- A mask register: Tx Maintenance Mapping Window n Mask
- A control register: Tx Maintenance Mapping Window n Control
- Tx Maintenance Mapping Window n Offset

For each defined and enabled window, the Avalon-MM address's least significant bits are masked out by the Window Mask and the resulting address is compared to the Window Base. If the addresses match, CONFIG_OFFSET is created based on the following equation:

If

```
(mnt_s_address & mask) = base),
config_offset = (offset[23:3] & mask[23:3]) |
(mnt_s_address[23:3] & ~mask[23:3])
```

where:

- mnt_s_address[31:0]: Avalon-MM Slave Interface Address
- config_offset[20:0]: Outgoing RapidIO register double-word offset
- base[31:0]: base address register
- mask[31:0]: mask register
- offset[23:0]: window offset register

If the address matches multiple windows, the lowest number window register set is used.

The following fields are inserted from the control register of the Mapping Window that matched.

- PRIO
- DEST_ID
- HOP_COUNT

The SOURCE_TID is generated internally and the WDPTR is assigned the negation of mnt_s_address[3].

For a Maintenance Avalon-MM Slave write, the value of the mnt_s_wridata[31:0] bus is inserted in the PAYLOAD field of the Maintenance Write packet.

Maintenance Master Processor

This module does the following tasks:

- For a Maintenance Read, converts the transaction into an Avalon read and presents it across the Maintenance Avalon-MM Master Interface.
- For a Maintenance Write, converts the transaction into an Avalon write and presents it across the Maintenance Avalon-MM Master Interface.
- Accounting related to the received RapidIO Maintenance Read/Write operation.
- For all received Maintenance request Packets, from remote ends, generates a Type 8 Response Packet and presents it to the Transport layer for transmission.

The Avalon-MM Master Interface supports the following Avalon transfers.

- Single Master Write Transfer
- Pipelined Master Read Transfers

Refer to *Avalon Memory-Mapped Interface Specification* for details on the supported transfers.

Figure 4–10 shows the signal relationships for a sequence of 4 write transfers on the Maintenance Avalon-MM Master interface.

Figure 4–10. Write Transfers on the Maintenance Avalon-MM Master Interface

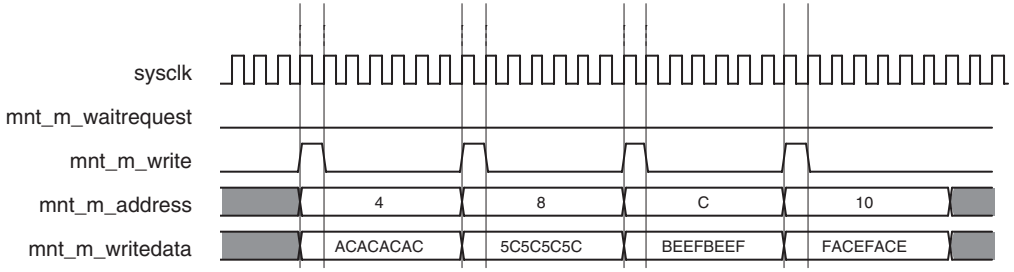
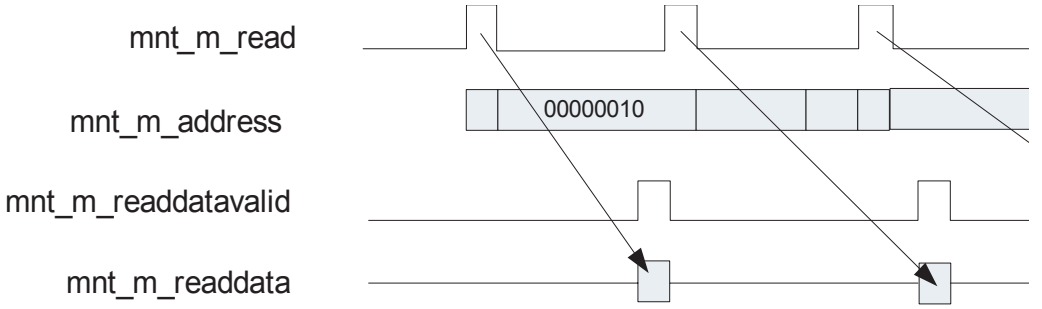


Figure 4–11 shows the signal relationships for a sequence of three read requests presented on the Maintenance Avalon-MM Master interface.

Figure 4–11. Timing of a Read Request on the Maintenance Avalon-MM Master Interface



A Maintenance Type Packet source by a remote device is first received by the Physical layer. After the Physical layer processes the packet, it is sent to the Transport layer. The Transport layer checks the FTYPE and the Destination_ID. If the FTYPE indicates that it is a Maintenance Type packet, (4'b1000), and if the Destination_ID field matches the Base Device ID programmed into register 'h60, the packet is sent to the Maintenance Module. The Maintenance Modules receives the packet on the RX Interface. The Maintenance module extracts the following fields of the packet header and uses them to compose the read or write transfer on the Maintenance Avalon-MM Master Interface.

- TTYPE
- RDSIZE/WRSIZE
- WDPTR
- CONFIG_OFFSET
- PAYLOAD

The maintenance module only supports single 32-bit word transfers, that is, RDSIZE and WRSIZE = 4'b1000; other values cause an error response packet to be sent.

The WDPTR and CONFIG_OFFSET values are used to generate the Avalon Address. It is composed as follows.

```
mnt_m_address = {rx_base, CONFIG_OFFSET, WDPTR, 2'b00}
```

where rx_base is the value programmed into register 'h10088, Figure 4–57.

The PAYLOAD is presented on the mnt_m_writedata[31:0] bus.

Port Write Processor

The Port Write Processor does the following.

- Composes the RapidIO Logical Header of a Maintenance Port Write request packet
- Presents the Port Write request packet to the Transport layer for transmission.
- Processes Port Write request packets received from a remote device.
- Alerts the user of a received Port Write using the `sys_mnt_s_irq` signal.

The port write processor is controlled through the use of registers `'h10200` through `'h1029C`.

To send a Port Write to a remote device, you must program the `Tx Port Write` data and control registers. These are located at addresses `'h10200` through `'h1024C`. The registers are accessed by using the System Maintenance Avalon-MM Slave interface. The following header fields are supplied by the values stored at the `TX Port Write Control` register at `'h10200`.

- `DESTINATION_ID`
- `PRIORITY`
- `WRSIZE`

The other fields of the Maintenance Port Write Packet are assigned as follows. The `FTYPE` is assigned a value of `4'b1000` and the `TTYPE` field is assigned a value of `4'b0100`. The `WDPTR` and `WRSIZE` fields of the transmitted packet are calculated from the size of the payload to be sent as defined by the `SIZE` field of the `TX_PORT_WRITE_CONTROL` register. The `SOURCE_TID` and `CONFIG_OFFSET` are reserved.

The payload is written into a `TX Port Write Buffer` starting at address `'h10210`. This buffer can store a maximum of 64 bytes. The Port Write Processor starts the packet composition and transmission process after the `PACKET_READY` bit in register `'h10200` has been set. The composed Maintenance Port Write packet is sent to the Transport layer for transmission.

The Maintenance Module receives a Maintenance Type packet on the RX Atlantic interface from the Transport layer. The Port Write Processor handles Maintenance Type Packets with a TTYPE value set to 4'b0100. The Port Write Processor extracts the following fields from the packet header and uses them to write the appropriate content to registers 'h10250 through 'h0129C.

- WRSIZE
- WDPTR
- PAYLOAD

The WRSIZE and the WDPTR determine the value of PAYLOAD_SIZE field in register 'h10254. The PAYLOAD is written to the RX Port Write Buffer starting at address 'h10260. A maximum of 64 bytes can be written. While the PAYLOAD is written into the buffer, the PORT_WRITE_BUSY bit of register 'h10254 is kept set. After the PAYLOAD is completely written into the buffer, the interrupt signal sys_mnt_s_irq is asserted by the Concentrator on behalf of the Port Write Processor. The interrupt is asserted only if RX_PACKET_STORED bit of Maintenance Interrupt Enable register 'h10084 is set.

Maintenance Module Error Handling

See the Maintenance Interrupt 'h10080 and Maintenance Interrupt Enable 'h10084 registers in Table 4–55 and Table 4–56. These register describe the error handling and reporting for Maintenance packets. Additionally, the following errors also apply to Maintenance packets.

- Maintenance read or Maintenance write request time out occurs and an PKT_RSP_TIMEOUT interrupt (bit 24 of register Maintenance Interrupt) is generated if a response packet is not received within the time specified by the Port Response Time-out Control port response timeout register.
- The IO_ERROR_RSP (bit 31) of the Error Management register is set when and ERROR response is received for a transmitted maintenance packet.

See Table 4–77 for more details on these error management registers.

Input/Output Logical Layer Modules

The Input/Output Logical layer module are optional components of the RapidIO MegaCore function.

When you create your custom RapidIO MegaCore function variation in the MegaWizard interface (see "Parameterize" on page 2–7), you have four choices:

- Avalon-MM Master and Slave
- Avalon-MM Master
- Avalon-MM Slave
- None

If you choose None, no Input/Output Logical layer module is added to your variation.

If you choose to add these modules to your variation, you must also choose a number of receive and/or transmit address translation (or mapping) windows. A minimum of one window is required per direction, and a maximum of 16 windows are available for each direction. For more information on address translation, see Table 4–68 through Table 4–74. Table 4–2 summarizes the functions of these address translation tables.

Table 4–2. Address Translation Tables

Address Translation Table Number	Function
Table 4–68	Input/Output Master base address for address translation in Input/Output Avalon-MM Master module.
Table 4–69	Input/Output Master address mask for address translation in Input/Output Avalon-MM Master module.
Table 4–70	Input/Output Master address offsetv for address translation in Input/Output Avalon-MM Master module.
Table 4–71	Input/Output Slave base address for address translation in Input/Output Avalon-MM Slave module.
Table 4–72	Input/Output Slave address mask for address translation in Input/Output Avalon-MM Slave module.
Table 4–73	Input/Output Slave address offset for address translation in Input/Output Avalon-MM Slave module.
Table 4–74	Input/Output Slave packet control information (for RapidIO packet header) for address translation in Input/Output Avalon-MM Slave module.

For more information on the Input/Output Logical layer, refer to *Part 1: Input/Output Logical Specification* of the *RapidIO Interconnect Specification*, Revision 1.3.

The control information is used by the Input/Output Avalon-MM slave module to construct the outgoing request packet's header. As the Input/Output Avalon-MM master module does not construct packet headers, it does not have control information registers.

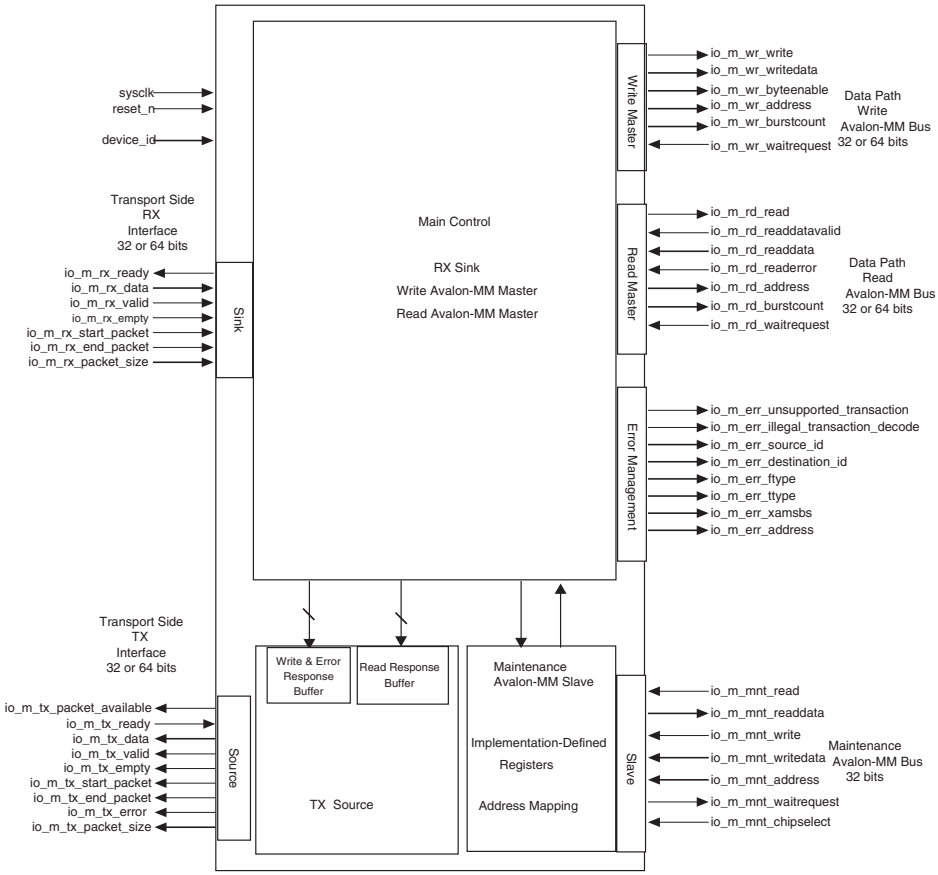
Input/Output Avalon-MM Master Module

The Input/Output Avalon-MM Master Logical layer module receives RapidIO read and write request packets from a remote end point through the Transport layer module. The Input/Output Avalon-MM Master module translates the request packets into Avalon-MM transactions, and creates and returns RapidIO response packets to the source of the request through the Transport layer. Figure 4–12 shows a block diagram of the Input/Output Avalon-MM Master Logical module and its interfaces.

The Input/Output Avalon-MM Master module is referred to as a master module because it is an Avalon-MM bus master.

To maintain full-duplex bandwidth, two independent Avalon-MM interfaces are used in the I/O Master and Slave modules—one for read transactions and one for write transactions.

Figure 4–12. I/O Master Logical Layer Block Diagram



Avalon-MM Interfaces Use Little Endian Byte Ordering

The Avalon-MM interfaces handle byte ordering differently than the RapidIO protocol. As shown in Table 4–3, the RapidIO protocol uses big endian byte ordering, whereas Avalon-MM interfaces use little endian byte ordering.

No byte or bit order is changed between the Avalon-MM protocol and RapidIO protocol, only the name is changed. For example, RapidIO Byte0 is Avalon-MM Byte7, and for all values of i from 0 to 63, bit i of the RapidIO 64-bit double word[0:63] of payload is bit (63-i) of the Avalon-MM 64-bit double word[63:0].

Because the RapidIO 32-bit word at RapidIO address 0x0000 is the most significant half of the 64-bit double word comprised of the two 32-words at RapidIO addresses 0x0000 and 0x0004, it actually corresponds to the Avalon-MM 32-bit word at address 0x0004 in the Avalon-MM address space. Thus, when a burst of two or more 32-bit Avalon-MM words is transported in RapidIO packets, the order of the pair of words is inverted so that the most significant words of each pair is transmitted or received first.

Table 4–3. Byte Ordering

Byte Lane	8'b1000_0000	8'b0100_0000	8'b0010_0000	8'b0001_0000	8'b0000_1000	8'b0000_0100	8'b0000_0010	8'b0000_0001
RapidIO Protocol (Big Endian)	Byte0[0:7]	Byte1[0:7]	Byte2[0:7]	Byte3[0:7]	Byte4[0:7]	Byte5[0:7]	Byte6[0:7]	Byte7[0:7]
	32-Bit Word[0:31] wdptr=0				32-Bit Word[0:31] wdptr=1			
	Double Word[0:63] RapidIO Address N = {29'hn, 3'b000}							
Avalon-MM Protocol (Little Endian)	Byte7[7:0]	Byte6[7:0]	Byte5[7:0]	Byte4[7:0]	Byte3[7:0]	Byte2[7:0]	Byte1[7:0]	Byte0[7:0]
	Address= N+7	Address= N+6	Address= N+5	Address= N+4	Address= N+3	Address= N+2	Address= N+1	Address= N
	32-Bit Word[31:0] Avalon-MM Address = N+4				32-Bit Word[31:0] Avalon-MM Address = N			
	64-bit Double Word0[63:0] Avalon-MM Address = N							

Input/Output Avalon-MM Master Address Mapping Windows

Address mapping, or translation windows, are used to map windows of 34-bit RapidIO addresses into windows of 32-bit Avalon-MM addresses. They are defined by sets of three 32-bit registers. Each window is defined by a base register and a mask register. The third register defines the offset into local memory where the RapidIO transaction gets mapped.

Your variation must have at least one translation window. You can change the values of the window defining registers at any time. You should disable the window before changing its window defining registers.

The number of mapping windows is defined by the parameter **Number of receive address translation windows**, for up to 16 sets of registers. Each set of registers supports one address mapping window.

A window is enabled if the window enable (WEN) bit of the IO Master Mapping Window n Mask register is set.

A window is defined by three 32-bit wide registers:

- a base register: IO Master Mapping Window_n_Base
- a mask register: IO Master Window n Mask
- an offset register: IO Master Window n Offset

Where n is from 0 to the number of address translation windows.

For each window that is defined and enabled, the least significant bits of the incoming RapidIO address are masked out by the Window Mask and the resulting address is compared to the Window Base. If the addresses match, the Avalon-MM address is made of the least significant bits of the RapidIO address and the window offset using the following equation:

Let rio_addr[33:0] be the 34-bit RapidIO address, and address[31:0] the local Avalon-MM address.

Let base[31:0], mask[31:0] and offset[31:0] be the three window defining registers. The least significant three bits of these registers are always 3'b000.

Starting from window 0, for the first window in which ((rio_addr & {xamm, mask}) == {xamb, base}),

where xamm and xamb are the Extended Address MSB fields of the IO_MASTER_WINDOW_n_MASK and IO_MASTER_WINDOW_n_BASE registers respectively, let address[31:3] = (offset[31:3] & mask[31:3]) | (rio_addr[31:3] & ~mask[31:3]).

address[2] is zero for variants with 64-bit wide data path Avalon-MM buses.

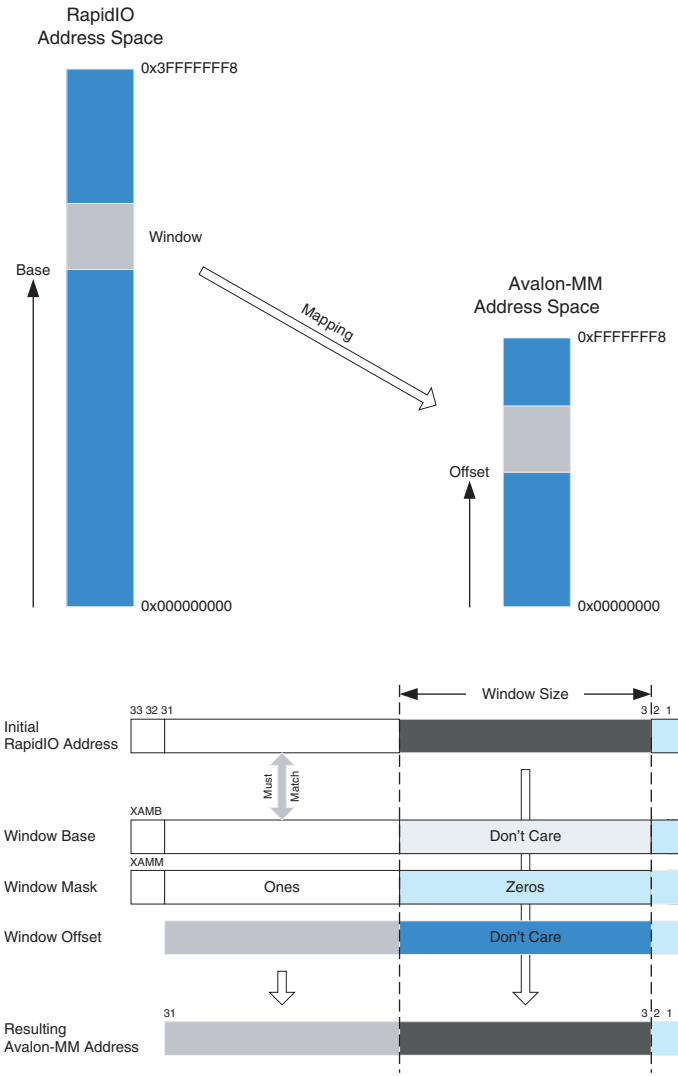
address[2] is determined by the values of wdptr and rdsize or wrsize for variants with 32-bit wide data path Avalon-MM buses.

address[1:0] are always zero.

For each received NREAD or NWRITE_R request packet that does not match any enabled window, an ERROR response packet is returned

Figure 4–13 shows a block diagram of the I/O Master Logical window translation.

Figure 4–13. I/O Master Window Translation



Input/Output Avalon-MM Slave Module

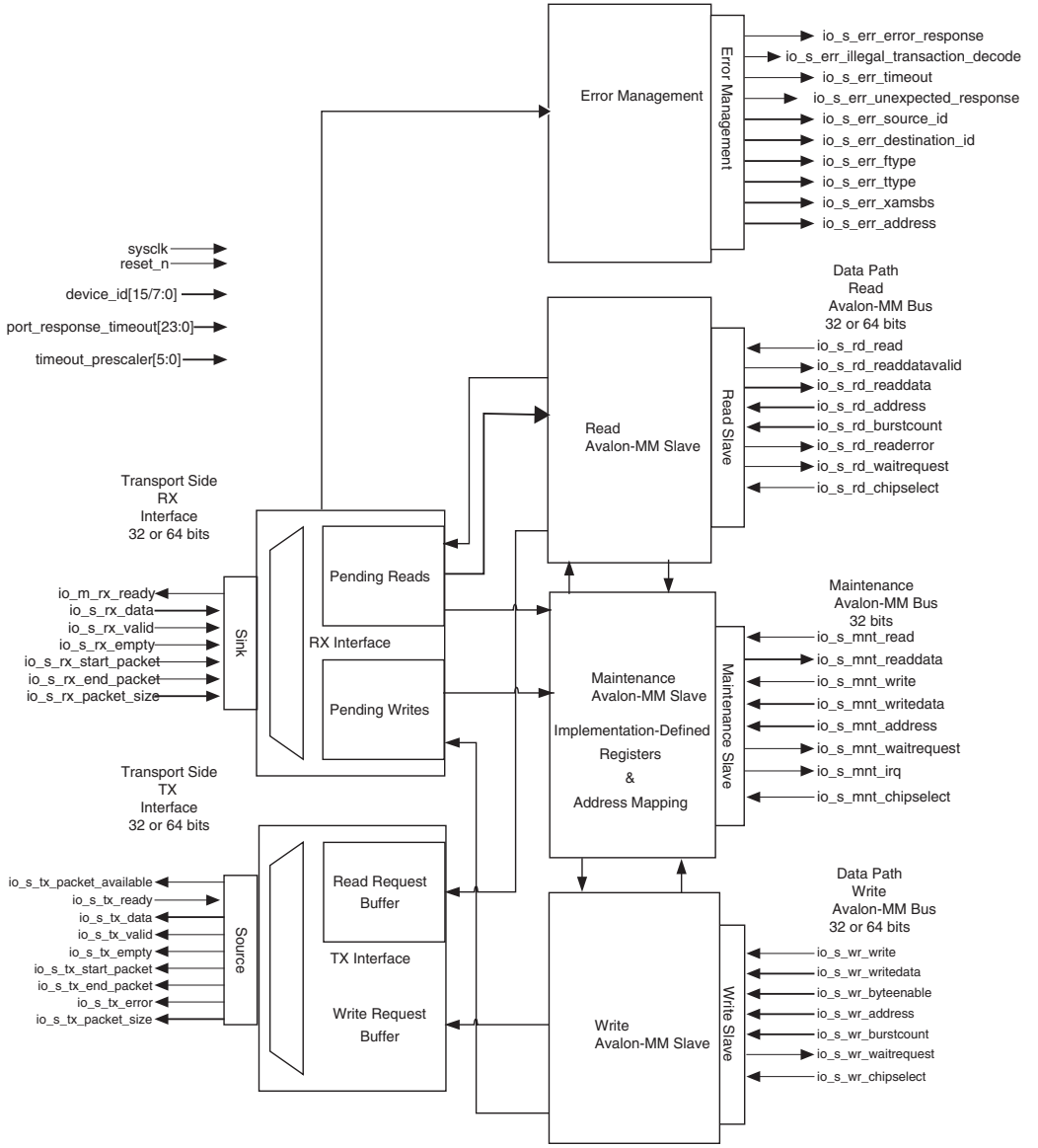
The Input/Output Avalon-MM Slave Logical layer module transforms Avalon-MM transactions into RapidIO read and write request packets that are sent through the Transport and Physical layer modules to a remote RapidIO processing element where the actual read or write

transaction takes place and response packets are created and sent back when required. Avalon-MM read transactions complete when the corresponding response packet is received. This module is referred to as a slave module because it is an Avalon-MM bus slave. Figure 4–14 shows a block diagram of the Input/Output Slave Logical module and its interfaces.

The maximum number of outstanding transactions (I/O Requests) supported is 64 (read requests + write requests).

To maintain full-duplex bandwidth, two independent Avalon-MM interfaces are used in the I/O Master and Slave modules—one for read transactions and one for write transactions.

Figure 4–14. Input/Output Avalon-MM Slave Logical Layer Block Diagram



Input/Output Avalon-MM Slave Address Mapping Windows

Address mapping, or translation windows are used to map windows of 32-bit Avalon-MM addresses into windows of 34-bit RapidIO addresses, and are defined by sets of four 32-bit registers.

Each window is defined by a base register, a mask register, and an offset register. The fourth register stores information used to prepare the packet header on the RapidIO side of the transaction, including the target device's DestinationID, the request packet's priority, and selects between the three available write request packet types: NWRITE, NWRITE_R and SWRITE.

Your variation must have at least one translation window.

You can change the values of the window defining registers at any time, even after sending a request packet and before receiving its response packet. You should disable the window before changing its window defining registers.

The number of mapping windows is defined by the parameter **Number of transmit address translation Windows**, for up to 16 sets of registers. Each set of registers supports one external host or entity at a time. A window is enabled if the window enable (`WEN`) bit of the **IO Slave Mapping Window *n* Mask** register is set, where ***n*** is the number of transmit address translation windows.

A window is defined by four registers:

- A base register: **Input/Output Slave Mapping Window *n* Base**
- A mask register: **Input/Output Slave Mapping Window *n* Mask**
- An offset register: **Input/Output Slave Mapping Window *n* Offset**
- A control register: **Input/Output Slave Mapping Window *n* Control**

For each window that is defined and enabled, the least significant bits of the Avalon-MM address are masked out by the Window Mask and the resulting address is compared to the Window Base. If the addresses match, the RapidIO address in the outgoing request packet is made of the least significant bits of the Avalon-MM address and the window offset using the following equation:

Let `avalon_address [31:0]` be the 32-bit Avalon-MM address, and `rio_addr [33:0]` the RapidIO address, where `rio_addr [33:32]` is the 2-bit wide `xamsbs` field and `rio_addr [31:3]` is the 29-bit wide `address` field in the packet and `rio_addr [2:0]` is implicitly defined by `wdptr` and `rdsize` or `wrsize`.

Let base[31:0], mask[31:0] and offset[31:0] be the values defined by the three corresponding window defining registers. The least significant three bits of base, mask and offset are fixed at 3'b000 regardless of the content of the window defining registers.

Let xamo be the Extended Address MSB Offset field in the IO_SLAVE_WINDOW_n_OFFSET register (the two least significant bits of the register).

Starting with window 0, for the first window where ((address & mask) == base),

Let rio_addr[33:3] = {xamo, (offset[31:3] & mask[31:3]) | (avalon_address[31:3]}.

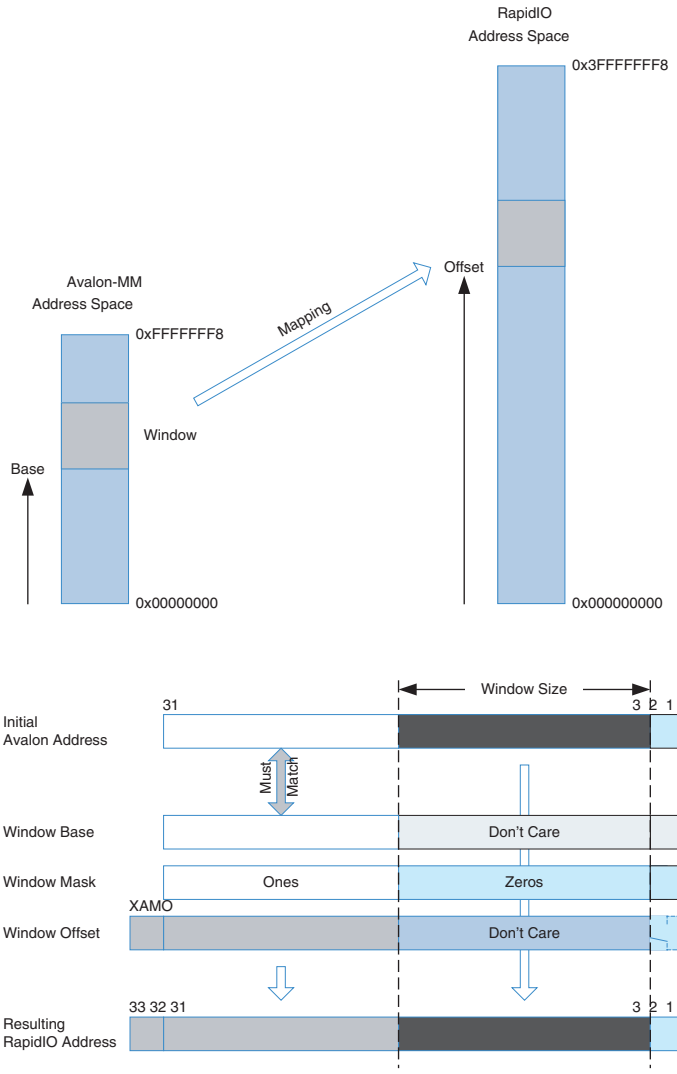
The Avalon-MM slave interfaces' burstcount and byteenable signals are used to determine the values of wdptr and rdsize or wrsize.

The bits mask[2:0] are always considered to be zero regardless of the contents of the IO_SLAVE_WINDOW_n_MASK register.

If the address matches multiple windows, the lowest number window register set is used. The following fields are inserted from the control register: PRIORITY and DESTINATION_ID. If the address does not match any window, an interrupt bit, either WRITE_OUT_OF_BOUNDS or READ_OUT_OF_BOUNDS in the Input/Output Slave Interrupt register, is set and the interrupt signal sys_mnt_s_irq is asserted if enabled by the corresponding bit in the Input/Output Slave Interrupt Enable register. An interrupt is cleared by writing one to the interrupt register.

Figure 4–15 shows a block diagram of the Input/Output Slave Logical window translation.

Figure 4–15. Input/Output Slave Window Translation

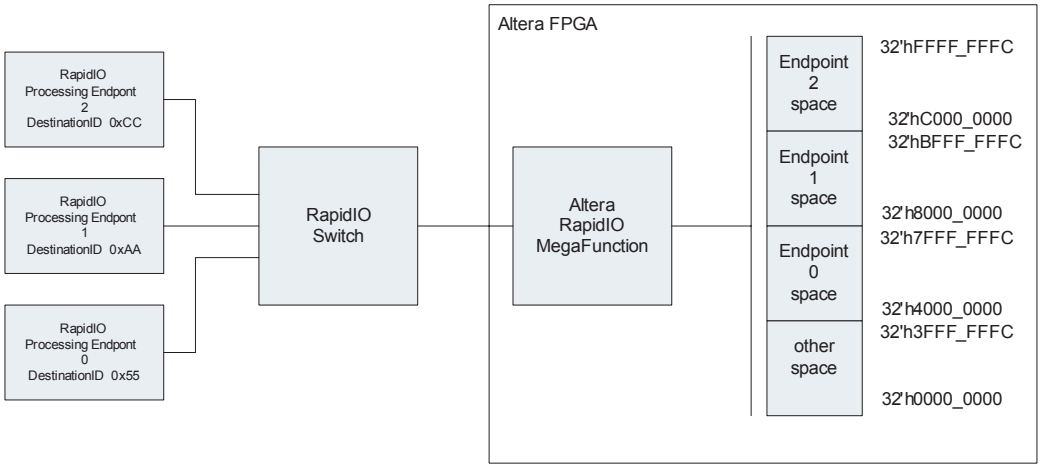


Input/Output Slave Translation Window Example

The following example shows the use of the Input/Output Slave Translation Windows. The system used in this example is shown in Figure 4–16. There are four RapidIO processing endpoints in the system. One of those processing endpoints is an Altera FPGA with a RapidIO MegaCore function. The processing endpoints can communicate with

each other through the RapidIO Switch. The Altera RapidIO has a local address mapping as shown in Figure 4–16. The mapping is based on the 32-bit Avalon-MM Address Space.

Figure 4–16. Using Input/Output Slave Translation Windows



The upper two bits of the Avalon-MM Address are used to differentiate between the other three Processing endpoints and are the only bits set to `1` in the mask registers. Figures Figure 4–17 through Figure 4–19 show the values programmed into three address translation windows.

Figure 4–17. Translation Window 0

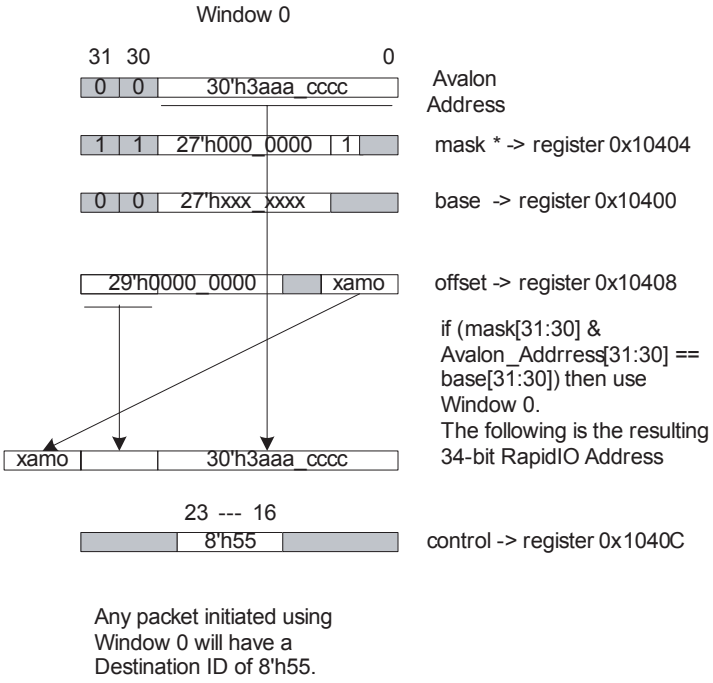


Figure 4–18. Translation Window 1

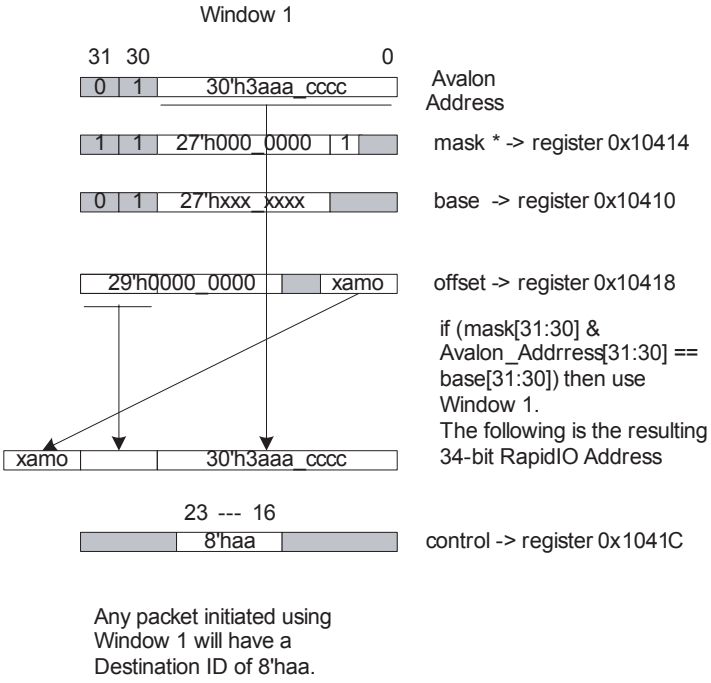
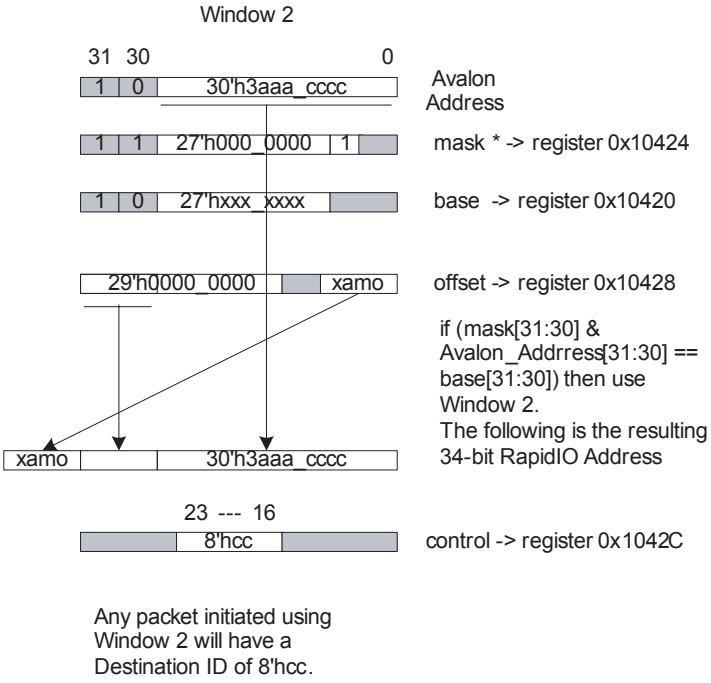


Figure 4–19. Translation Window 2



Avalon-MM Slave Address Mapping

The Avalon-MM interface and RapidIO protocol use different address spaces, so Avalon-MM burst count, byte enable, and (in 32 bit variations) address values are translated into RapidIO packet fields read size, write size, and word pointer.

Slave Request Packet Size Encoding

The Avalon-MM interface and RapidIO protocol use different address spaces. The RapidIO core converts packets between Avalon-MM and RapidIO formats. The Avalon-MM burst count, byte enable, and, in 32 bit variations, address bit 2, values are translated into the RapidIO packets' read size, write size, and word pointer fields. For more information see Figure 4–15 and Table 4–4 for packet size encoding used in the conversion

process for 32-bit datapath read requests, Table 4–5 for 32- bit datapath write requests, and Figure 4–15 and Table 4–6 for information on 64-bit datapath conversions.

Table 4–4. Slave Read Request Size Encoding (32-bit datapath)

Avalon-MM Values		RapidIO Values	
burstcount *(2)*	Address *(1)* (1 ' bx)	wdptr (1 ' bx)	rdsize *(2)* (4 ' bxxxx)
1	1	0	1000
1	0	1	1000
2	0	0	1011
3–4	0	1	1011
5–8	0	0	1100
9–16	0	1	1100
17–24	0	0	1101
25–32	0	1	1101
33–40	0	0	1110
41–48	0	1	1110
49–56	0	0	1111
57–64	0	1	1111

Notes for Table 4–4

(1) Burst transfers of more than one Avalon-MM word must start on a double-word aligned Avalon-MM address. If the Slave Read Burst Count is larger than one and io_s_rd_address[2] is not zero, the transfer completes in the same manner as a failed mapping: the READ_OUT_OF_BOUNDS bit in the IO_SLAVE_INTERRUPT register is set and `sys_mnt_s_irq` is asserted if enabled. In the case of a read transfer, the transfer is marked as errored by asserting `io_s_rd_readerror` for the duration of the burst.

(2) For read transfers, the read size of the request packet is rounded up to the next supported size, but only the number of words corresponding to the requested read burst size are returned.

Table 4–5. Slave Write Request Size Encoding (32 bit datapath) (Part 1 of 2)

Avalon-MM Values			RapidIO Values	
burstcount *(3)*	**byteenable (4'bxxxx)**	**Address** *(1)* **(1'bx)**	**wdptr (1'bx)**	**wrsize (4b'bxxxx)**
1	1000	1	0	0000
1	0100	1	0	0001
1	0010	1	0	0010
1	0001	1	0	0011
1	1000	0	1	0000
1	0100	0	1	0001
1	0010	0	1	0010
1	0001	0	1	0011
1	1100	1	0	0100
1	1110	1	0	0101
1	0011	1	0	0110
1	1100	0	1	0100
1	0111	0	1	0101
1	0011	0	1	0110
1	1111	1	0	1000
1	1111	0	1	1000

Table 4–5. Slave Write Request Size Encoding (32 bit datapath) (Part 2 of 2)

Avalon-MM Values			RapidIO Values	
burstcount *(3)*	**byteenable (4'bxxxx)**	**Address** *(1)* **(1'bx)**	**wdptr (1'bx)**	**wrsize (4b'bxxxx)**
2	1111 *(2)*	0	0	1011
4			1	1011
6 or 8			0	1100
10, 12, 14, 16			1	1100
18, 20, 22, 24			1	1101
26, 28, 30, 32			1	1101
34, 36, 38, 40			0	1110
42, 44, 46, 48			1	1110
50, 52, 54, 56			0	1111
58, 60, 62, 64			1	1111

Notes for Table 4–5:

(1) Burst transfers of more than one Avalon-MM word must start on a double-word aligned Avalon-MM address. If io_s_wr_burstcount is larger than one and io_s_wr_address is not zero, the transfer completes in the same manner as a failed mapping: the WRITE_OUT_BOUNDS bit in the IO_SLAVE_INTERRUPT register is set and `sys_mnt_s_irq` is asserted if enabled.

(2) For all Avalon-MM write transfers with burstcount larger than 1, io_s_wr_byteenable must be set to 4'b1111. It it is not, the transfer fails: the INVALID_WRITE_BYTEENABLE bit in the IO_SLAVE_INTERRUPT register is set and `io_s_mnt_irq` is asserted if enabled.

(3) For write transfers, odd burst sizes other than 1 are not supported. If one occurs the INVALID_WRITE_BURSTCOUNT bit in the IO_SLAVE_INTERRUPT register is set, causing `sys_mnt_s_irq` to be asserted if enabled.

Table 4–6. Slave Read Request Size Encoding (64 bit datapath) (Part 1 of 2)

Avalon-MM Values		RapidIO Values
burstcount *(1)*	**wdptr (1'bx)**	**rdsize** *(1)* **(4'bxxxx)**
1	1'b0	4'b1011
2	1'b1	4'b1011
3–4	1'b0	4'b1100
5–8	1'b1	4'b1100
9–12	1'b0	4'b1101
13–16	1'b1	

Table 4–6. Slave Read Request Size Encoding (64 bit datapath) (Part 2 of 2)		
Avalon-MM Values		RapidIO Values
burstcount (1)	wdptr (1'bx)	rdsize (1) (4'bxxxx)
17–20	1'b0	4'b1111
21–24	1'b1	
25–28	1'b0	
29–32	1'b1	

Notes for Table 4–6

(1) For read transfers, the read size of the request packet is rounded up to the next supported size, but only the number of words corresponding to the requested read burst size are returned.

Table 4–7. Slave Write Request Size Encoding (64 bit datapath) (Part 1 of 2)			
Avalon-MM Values		RapidIO Values	
burstcount	byteenable (8'bxxxx_xxxx)	wdptr (1'bx)	wrsize (4'bx)
1	1000_0000	0	0000
1	0100_0000	0	0001
1	0010_0000	0	0010
1	0001_0000	0	0011
1	0000_1000	1	0000
1	0000_0100	1	0001
1	0000_0010	1	0010
1	0000_0001	1	0011
1	1100_0000	0	0101
1	1110_0000	0	0110
1	0011_0000	0	0111
1	1111_1000	0	1000
1	0000_1100	1	1000
1	0000_0111	1	1001
1	0000_0011	1	1001
1	0001_1111	1	1010
1	1111_0000	0	1000
1	0000_1111	1	1000

Table 4–7. Slave Write Request Size Encoding (64 bit datapath) (Part 2 of 2)

Avalon-MM Values		RapidIO Values	
burstcount	byteenable (8'bxxxx_xxxx)	wdptr (1'bx)	wrsize (4'bx)
1	1111_1100*(1)*	0	1001
1	0011_1111	1	1001
1	1111_1110	0	1010
1	1111_1111	1	1010
1	1111_1111	0	1011
2	1111_1111	1	1011
3-4		0	1100
5-8		1	1100
9-12		1	1101
13-16			
17-20		1	1111
21-24			
25-28			
29-32			

Notes for Table 4–7:

(1) For all Avalon-MM write transfers with burstcount larger than 1, `io_s_wr_byteenable` must be set to 8'b1111_1111. If it is not, the transfer fails: the INVALID_WRITE_BYTEENABLE bit in the IO_SLAVE_INTERRUPT register is set and `io_s_mnt_irq` is asserted if enabled.

Doorbell Module

The doorbell module provides support for Type 10 packet format (doorbell class) transactions, allowing users to send and receive short software-defined messages to and from other processing elements connected to the RapidIO interface.

As shown in Figure 4–1 on page 2, the Doorbell module is connected to the Transport layer Module. In a typical application the Doorbell Module Avalon-MM slave interface is connected to Avalon-MM system interconnect fabric, allowing an Avalon-MM master to communicate with RapidIO devices by sending and receiving doorbell messages.

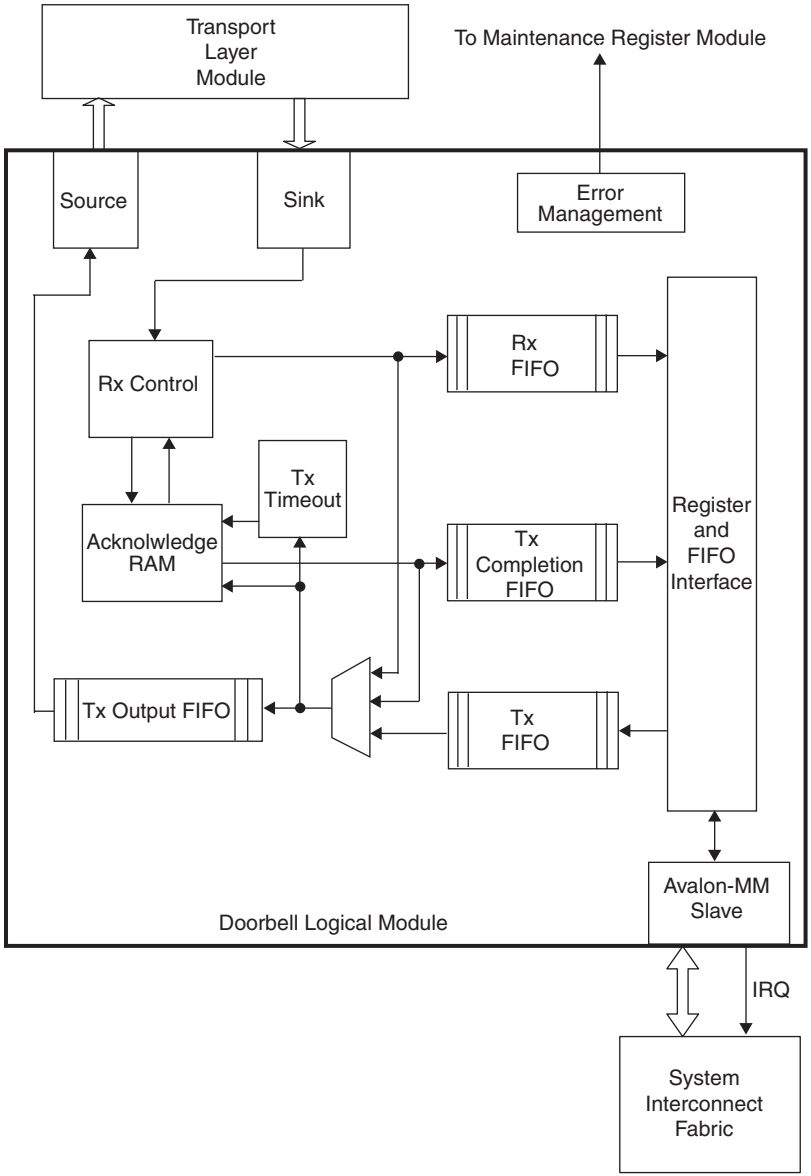
When you configure the RapidIO MegaCore function, you can enable or disable the doorbell operation feature, depending on your application requirements (if you do not need the doorbell feature, disabling it will reduce device resource usage). If you enable the feature, a 32–bit Avalon-MM slave port is created that allows the RapidIO MegaCore to receive and/or generate RapidIO doorbell messages.

Doorbell Module Block Diagram

Figure shows a block diagram of the Doorbell Module logic. This module includes a 32–bit Avalon-MM slave interface to the user interface. The Doorbell Module contains the following logic blocks:

- Register and FIFO Interface, for allowing an external Avalon-MM master to access the Doorbell Module internal registers and FIFO buffers
- Tx Output FIFO that stores the outbound doorbell and response packets waiting for transmission to the Transport layer module.
- Acknowledge RAM, used to temporarily store the transmitted doorbell packets pending responses to the packets from the target RapidIO device.
- Tx Timeout logic that checks the expiration time for each outbound Tx doorbell packet that is sent.
- Rx Control manages the Atlantic Sink protocol by fetching and processing available doorbell packets from the Transport layer module. Received packet types include:
  - Rx doorbell request
  - Rx response DONE to a successfully transmitted doorbell packet.
  - Rx response RETRY to a transmitted doorbell message
  - Rx response ERROR to a transmitted doorbell message
- Rx FIFO that stores the received doorbell messages and passes them to the external Avalon-MM master device.
- Tx FIFO that stores doorbell messages that were generated for the external Avalon-MM master device and are waiting to be transmitted.
- Tx Completion FIFO that stores the transmitted doorbell messages that have received responses. This FIFO also stores timed out TX doorbell requests.
- Error Management module that reports detected errors, including
  - Unexpected response (a response packet was received, but its TransactionID does not match any pending request that is waiting for a response.
  - Request timeout (an outbound doorbell request did not receive a response from the target device).

Figure 4–20. Doorbell Module Block Diagram



Doorbell Message Generation

To generate a doorbell request packet on the RapidIO serial interface, you perform the following steps, using a set of registers described in "Doorbell Message Registers" on page 4–112:

1. Optionally enable interrupt by writing '1' to the appropriate bit of the Tx Doorbell Interrupt Enable register.
2. Optionally enable confirmation of successful outbound messages by writing `1` to the COMPLETED bit of the `Tx Completion Control` register
3. Set up the `PRIORITY` field of the Tx Doorbell Control register.
4. Write the `Tx Doorbell` register to set up the `DESTINATION_ID` and `INFORMATION` fields of the generated doorbell packet format.

Before writing to the `Tx Doorbell` register you must be certain that the FIFO has available space to accept the write data. This is to avoid a wait request signal assertion due to a full FIFO. When wait request is asserted you can not perform other transactions on the doorbell Avalon-MM slave port until the current transaction is completed. You can determine the level of the Tx FIFO by reading the Tx Doorbell Status Register (the FIFO is 16 words deep).

Once a write to the `Tx Doorbell` register is detected, internal control logic generates and sends a Type 10 packet based on the information in the `Tx Doorbell` and `Tx Doorbell Control` registers. A copy of the outbound doorbell packet is stored in the Acknowledge RAM.

When the response to an outbound doorbell message is received, the corresponding copy of the outbound message is written into the *Tx* Doorbell Completion buffer (if enabled), and an interrupt is generated (if enabled) on the Avalon-MM slave interface by asserting the drbell_s_irq signal of the doorbell. The ERROR_CODE field in the `Tx Doorbell Completion Status` register indicates successful or error completion.

The corresponding interrupt status bit is set each time a valid response packet is received, and resets itself when the Tx Completion FIFO is empty. Software can optionally clear the interrupt status bit by writing a `1` to this specific bit location of the `Doorbell Interrupt Status` register.

Upon detecting the interrupt, software can fetch the completed message and find out its status by reading the `Tx Doorbell Completion` register and `Tx Doorbell Completion Status` register respectively.

An outbound message that times out before its response is received is treated in the same manner as an outbound message that receives error response: if enabled, an interrupt is generated by the Error Management module by asserting the sys_mnt_s_irq signal, and the ERROR_CODE field in the Tx Doorbell Completion Status register is set to indicate the error.

If interrupt is not enabled, the Avalon-MM master must periodically poll the Tx Doorbell Completion Status register to check for available completed messages before retrieving them from the Tx Completion FIFO.

Doorbell request packets for which RETRY responses are received are automatically resent by hardware. No retry limit is imposed on outbound doorbell messages.

Doorbell Message Reception

Doorbell request packets received from the Transport layer module are stored in an internal buffer, and an interrupt is generated on the doorbell Avalon-MM slave interface, if the interrupt is enabled.

The corresponding interrupt status bit is set every time a doorbell request packet is received and resets itself when the Rx FIFO is empty. Software can clear the interrupt status bit by writing a 1 to this specific bit location of the Doorbell Interrupt Status register.

If interrupt is not enabled, the external Avalon-MM master must periodically poll the Rx Doorbell Status register to check the number of available messages before retrieving them from the Rx Doorbell buffer.

Appropriate Type 13 response packets are generated internally and sent for all the received doorbell messages. A response with DONE status is generated when the received doorbell packet can be processed immediately. A response with RETRY status is generated to defer processing the received message when the internal hardware is busy, for example when the RX Doorbell buffer is full.

Avalon-ST Pass-Through Interface

The Avalon-ST pass-through interface is an optional interface that is generated when you select the Avalon-ST pass-through port in the Transport and Maintenance panel of the MegaWizard interface (see Figure 2–6 on page 2–11). All packets received by the Transport layer that do not match this MegaCore Base device_ID or which have FTYPEs that are not supported by this MegaCore function are routed to the Pass-

Through RX Avalon-ST interface. The packets can then be further examined by a local processor or parsed and processed by a custom user function.

Applications for using the Avalon-ST pass-through port include the following:

- User implementation of a RapidIO function not supported by this MegaCore function (for example, Message Passing SAR logic)
- User implementation of a custom function not specified by the RapidIO protocol, but which may be useful for the system application.

Table 4–8 provides a description of signals of the Avalon-ST pass-through interface in the transmit (TX) direction.

Table 4–8. Avalon-ST Pass-Through Port Transmission Signals

Signal	Type	Function
gen_tx_ready	Output	Indicates that the core is ready to receive data on the next clock cycle
gen_tx_valid	Input	Used to qualify all the other transmit side pass-through port input signals. On every rising edge of the clock where gen_tx_valid is high, data is sampled by the core.
gen_tx_startofpacket	Input	Marks the start of a packet transfer. This is qualified with gen_tx_valid.
gen_tx_endofpacket	Input	Marks the end of a packet transfer. This is qualified with gen_tx_valid.
gen_tx_data	Input	A 32 or 64-bit wide data bus; 1x or 4x respectively.

Table 4–8. Avalon-ST Pass-Through Port Transmission Signals

Signal	Type	Function
gen_tx_empty	Input	This bus identifies the number of empty bytes on the last data transfer of the gen_tx_endofpacket. For a 32-bit wide data bus, this bus is 2 bits wide. For a 64-bit wide data bus, this bus is 3 bits wide. The following values are supported. gen_tx_empty empty bits 2'b00 none 2'b01 [7:0] 2'b10 [15:0] 2'b11 [23:0] gen_tx_empty empty bits 3'b000 none 3'b001 [7:0] 3'b010 [15:0] 3'b011 [23:0] 3'b100 [31:0] 3'b101 [39:0] 3'b110 [47:0] 3'b111 [56:0]
gen_tx_error	Input	Indicates that the corresponding data has an error. Assertion of this signal any time during the packet transfer will cause the packet to be dropped by the core. Qualified with gen_tx_valid

Table 4–9 describes the Avalon-ST pass-through signals in the receive (RX) direction.

Table 4–9. Avalon-ST Pass-Through Port Receiver Signals

Signal	Type	Function
gen_rx_ready	Input	Indicates to the core that the user is ready to receive data on the next clock cycle
gen_rx_valid	Output	Used to qualify all the other output signals of the receive side pass-through port. On every rising edge of the clock where gen_rx_valid is high, gen_rx_data can be sampled.
gen_rx_startofpacket	Output	Marks the start of a packet transfer. *(2)*
gen_rx_endofpacket	Output	Marks the end of a packet transfer. *(2)*
gen_rx_data	Output	A 32 or 64-bit wide data bus; 1x or 4x mode respectively.

Table 4–9. Avalon-ST Pass-Through Port Receiver Signals

Signal	Type	Function
gen_rx_empty	Output	This bus identifies the number of empty bytes on the last data transfer of the gen_rx_endofpacket. For a 32-bit wide data bus, this bus is 4 bits wide. For a 64-bit wide data bus, this bus is 8 bits wide. The following values are supported. gen_rx_empty empty bits 2'b00 none 2'b01 [7:0] 2'b10 [15:0] 2'b11 [23:0] gen_rx_empty empty bits 3'b000 none 3'b001 [7:0] 3'b010 [15:0] 3'b011 [23:0] 3'b100 [31:0] 3'b101 [39:0] 3'b110 [47:0] 3'b111 [56:0]
gen_rx_error	Output	Indicates that the corresponding data has an error. This signal is never asserted by the RapidIO MegaCore function.
gen_rx_size *(1)*	Output	Identifies the number of cycles the current packet transfer will require. This signal is only valid on the start of packet cycle (gen_rx_startofpacket).

Note:
(1) This is not an Avalon-ST signal and will be exported when the RapidIO MegaCore function is used as part of an SOPC Builder system.
(2) gen_rx_valid is used to qualify all the other output signals of the receive side pass-through port.

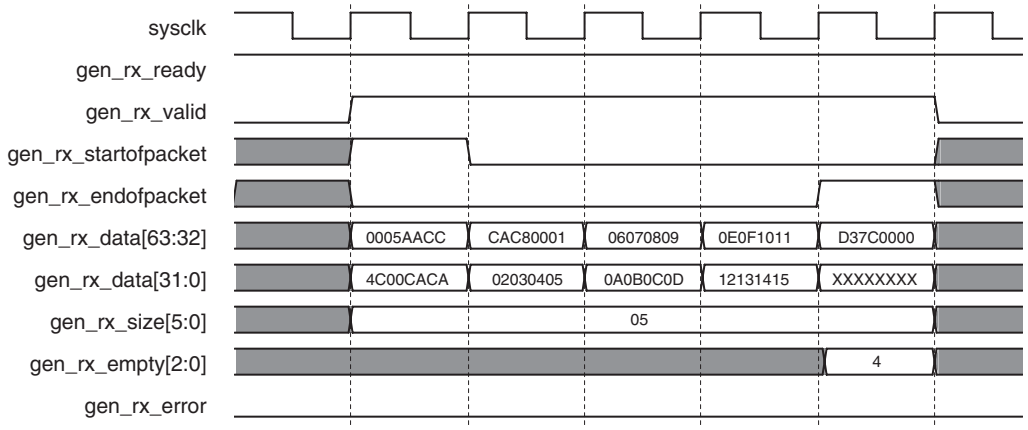
Pass-Through Port Examples

This section contains two examples, one receiving and the other transmitting a packet through the Avalon-ST pass-through port.

Packet Routed Through RX Port on Avalon-ST Pass-Through Port
The following example of a packet routed to the receiver Avalon-ST pass-through port is for a variation that only has the Maintenance module and the Avalon-ST pass-through port enabled. The interface does not have an Input/Output Logical layer interface. A packet received by the RapidIO MegaCore on the RapidIO interface whose FTYPE does not indicate a

Maintenance Type transaction is routed to the receiver port of the Avalon-ST pass-through interface. The timing diagram in Figure 4–21 shows a packet received on this interface.

Figure 4–21. Packet Received on the Avalon-ST Pass-Through Interface



In cycle 0, the user logic indicates to the MegaCore function that it is ready to receive a packet transfer by asserting `gen_rx_ready`. In cycle 1, the MegaCore function asserts `gen_rx_valid` and `gen_rx_startofpacket`. During this cycle, `gen_rx_size` is valid and indicates that it will take 5 cycles to transfer the packet. Table 4–10 shows the RapidIO header fields and the payload carried on the `gen_rx_data` bus and the corresponding cycle.

Table 4–10. RapidIO Header Fields and gen_rx_data Bus Payload (Part 1 of 2)

Cycle	Field	gen_rx_data bus	Value	Comment
1	ackID	[63:59]	5'h00	
	rsvd	[58:57]	2'h0	
	CRF	[56]	1'b0	
	prio	[55:54]	2'h0	
	tt	[53:52]	2'h0	
	ftype	[51:48]	4'h5	A value of 5 indicates a Write Class packet
	destinationID	[47:40]	8'haa	
	sourceID	[39:32]	8'hcc	
	ttype	[31:28]	4'h4	The value of 4 indicates a NWRITE transaction.

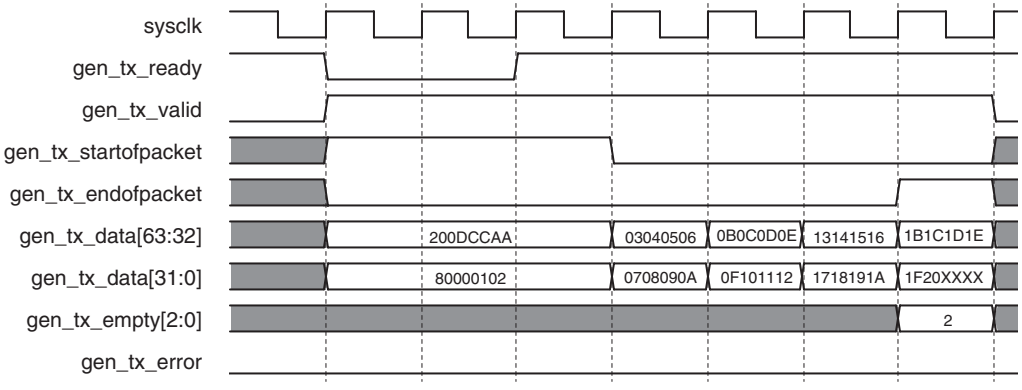
Table 4–10. RapidIO Header Fields and gen_rx_data Bus Payload (Part 2 of 2)

Cycle	Field	gen_rx_data bus	Value	Comment
1	wrsize	[27:24]	4'hc	The wrsize and wdptr values encode the maximum size of the payload.In this example it decodes to a value of 32 bytes. Refer to Table 4-4 of the *RapidIO Part 1: Input/Output Logical Specification Rev. 1.3*
	srcTID	[23:16]	8'h00	
	address[28:13]	[15:0]	16'hcaca	The 29 bit address composed is 29'h19595959. This becomes 32'hcacacac8, the double-word physical address.
2	address[12:0]	[63:51]	13'h1959	
	wdptr	[50]	1'b0	See description for the size field.
	xamsbs	[49:48]	2'h0	
	Payload Byte0,1	[47:32]	16'h0001	
	Payload Byte2,3	[31:16]	16'h0203	
	Payload Byte4,5	[15:0]	16'h0405	
3	Payload Byte6,7	[63:48]	16'h0607	
	Payload Byte8,9	[47:32]	16'h0809	
	Payload Byte10,11	[31:16]	16'h0a0b	
	Payload Byte12,13	[15:0]	16'h0c0d	
	Payload Byte14,15	[63:48]	16'h0e0f	
4	Payload Byte16,17	[47:32]	16'h1011	
	Payload Byte18,19	[31:16]	16'h1213	
	Payload Byte20,21	[15:0]	16'h1415	
5	CRC[15:0]	[63:48]	16'hd37c	For packets with a payload greater than 80 bytes, the middle crc is stripped but the last crc is not stripped. For packets smaller than 80 bytes, the crc is not stripped.
	Pad bytes	[47:32]	16'h0000	The RapidIO allows an implementation to add Pad bytes for the payload to adhere to 32-bit alignment

NREAD Example Using TX Port on Avalon-ST Pass-Through Port

The next example shows the response to an NREAD transaction. The response is presented on the TX port of the Avalon-ST pass-through interface. The timing diagram in Figure 4–22 shows the packet presented on this interface.

Figure 4–22. Packet Transmitted on the Avalon ST Pass-Through Port Interface



The timing diagram shows a response to a 32-byte NREAD request. The Xs indicate a don't care value. Table 4–11 shows the composition of the fields in the RapidIO packet header and the payload as they correspond to each clock cycle. The gen_tx_empty bits indicate a value of 2, which means that there are 2 empty bytes in the last data word.

Table 4–11. RapidIO Header Fields on the gen_rx_data Bus (Part 1 of 2)

Cycle	Field	gen_rx_data bus	Value	Comment
1	ackID	[63:59]	5'h04	
	rsvd	[58:57]	2'h0	
	CRF	[56]	1'b0	
	prio	[55:54]	2'h0	
	tt	[53:52]	2'h0	
	ftype	[51:48]	4'hd	A value of 4'hd (13 decimal) indicates a Response Class packet
	destinationId	[47:40]	8'hcc	
	sourceId	[39:32]	8'haa	
	ttype	[31:28]	4'h8	The value of 8 indicates a RESPONSE transaction with data payload.
	status	[27:24]	4'h0	A value of 0 indicates DONE. Requested transaction has been successfully completed.
	targetTID	[23:16]	8'h00	
	Payload Byte0,1	[15:0]	16'h0102	payload word 0
2	Payload Byte2,3	[63:48]	16'h0304	payload word 1
	Payload Byte4,5	[47:32]	16'h0506	
	Payload Byte6,7	[31:16]	16'h0708	
	Payload Byte8,9	[15:0]	16'h090a	

Table 4–11. RapidIO Header Fields on the gen_rx_data Bus (Part 2 of 2)

Cycle	Field	gen_rx_data bus	Value	Comment
3	Payload Byte10,11	[63:48]	16'h0b0c	
	Payload Byte12,13	[47:32]	16'h0d0e	
	Payload Byte14,15	[31:16]	16'h0f10	
	Payload Byte16,17	[15:0]	16'h1112	
4	Payload Byte18,19	[63:48]	16'h1314	
	Payload Byte20,21	[47:32]	16'h1516	
	Payload Byte22,23	[31:16]	16'h1718	
	Payload Byte24,25	[15:0]	16'h191a	
5	Payload Byte26,27	[63:48]	16'h1b1c	
	Payload Byte28,29	[47:32]	16'h1d1e	
	Payload Byte30,31	[31:16]	16'h1f10	Payload word 15

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation can support the following two modes of operation:

- *Untethered*—the design runs for a limited time
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the timeout behavior of the other megafunctions.

For MegaCore functions, the untethered timeout is 1 hour; the tethered timeout value is indefinite.

Your design stops working after the hardware evaluation time expires.

After that time, the RapidIO MegaCore function behaves as though the Physical layer's Atlantic interface signals `atxena` and `arxena` are tied low.

As a result, it is impossible for the RapidIO MegaCore function to transmit new packets (it will only transmit idles and status control symbols), or read packets out of the Atlantic interface. If the far end continues to transmit packets, the RapidIO MegaCore function starts refusing new packets by sending `packet_retry` control symbols once its receiver buffer fills up beyond the corresponding threshold.

For more information on OpenCore Plus hardware evaluation using the RapidIO MegaCore function, see *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Error Detection and Management

The error detection and management mechanisms in the RapidIO specification and built into the RapidIO MegaCore function provide a high degree of reliability. In addition to error detection, management, and recovery features, the RapidIO MegaCore function also implements a few debugging and diagnostic tools.

This section outlines the errors, error detection, and management features in the RapidIO MegaCore function.

Physical Layer Error Management

At the physical layer there are mainly two types of possible errors:

- Protocol violations
- Transmission errors

Protocol violations can be caused by a link partner that is not fully compliant to the specification, or can be a side effect of the link partner being reset.

Transmission errors can be caused by noise on the line and consist of one or more bit errors. The following mechanisms exist for checking and detecting errors:

- The receiver checks the validity of the received 8B10B encoded characters, including the running disparity.

- The receiver detects control characters changed into data characters or data characters changed into control characters, based on the context in which the character is received.
- The receiver checks the CRC of the received control symbols and packets.

The RapidIO MegaCore function physical layer transparently manages these errors for the user. The RapidIO specification defines both input and output error detection and recovery state machines that include handshaking protocols in which the receiving end signals that an error is detected by sending a `packet-not-accepted` control symbol; the transmitter then sends an `input-status link-request` control symbol that the receiver responds to with a `link-response` control symbol to indicate which packet needs to be retransmitted. The input and output error detection and recovery state machines can be monitored by software you create to read the status of the `Port 0 Error and Status` CSR.

In addition to the registers defined by the specification, the RapidIO MegaCore function provides several output signals outlined in Table 4–12 that enable user logic to monitor the error detection and recovery process.

Table 4–12. Signals for Detecting and Managing Errors

Output Signal	Comments
`char_err`	Invalid or illegal character or disparity error detected. See Table 3–10 on page 3–30, *Packet and Error Monitoring Signals,* for a more detailed description of these signals.
`symbol_error`	CRC error detected in received packet. See Table 3–10 on page 3–30, *Packet and Error Monitoring Signals,* for a more detailed description of these signals.
`packet_crc_error`	transmission error occurred
`packet_not_accepted`	marks the beginning of the error recovery process
`packet_transmitted`	allows user logic to monitor traffic exchange under error-free conditions
`packet_accepted`	allows user logic to monitor traffic exchange under error-free conditions
`packet_retry`	when asserted, this state can be symptomatic of buffer congestion
`packet_cancelled`	when asserted, this state can be symptomatic of buffer congestion
Input Signal	
`rerr`	This input signal is used by external logic to indicate 8B10B decoding errors for variations that use an external transceiver

Protocol Violations

Some protocol violations, such as a packet with an unexpected AckID or a timeout on a packet acknowledgment, can use the same error recovery mechanisms as the transmission errors described above, but other protocol violations such as a timeout on a link-request or the RapidIO MegaCore function receiving a link-response with an ackID outside the range of transmitted AckIDs, can lead to unrecoverable – or fatal errors.

Fatal Errors

Fatal errors cause a soft reset of the physical layer module, which clears all the transmit buffers and resets the transmission and expected AckID to 0. This effect also can be triggered by software that writes a one and then a zero to the `PORT_DIS` bit of the `Port 0 Control` CSR.

Fatal Error When Resetting the Link Partner

If the link partner is reset when its expected AckID is not zero, a fatal error occurs when it receives the next transmitted packet because the link partner's expected AckID is reset to zero, which causes a mismatch between the transmitted AckID and the expected AckID. The fatal error causes a soft reset of the MegaCore function. After the soft reset completes, transmitted and expected AckIDs are synchronized and normal operation resumes. Only the packets that were queued at the time of the fatal error are lost.

Logical Layer Error Management

The Logical layer manages only Logical layer errors because errors detected by the Physical layer are masked from the Logical layer module. Any packet that has the `arxerr` signal asserted is dropped in the Transport layer before it reaches the logical layer modules.

The RapidIO specification defines the following common errors and the protocols for managing them:

- malformed request or response packets
- unexpected Transaction ID
- missing response (timeout)
- response with ERROR status

The RapidIO MegaCore function implements part of the optional Error Management Extensions as defined in Part 8 of the *RapidIO Interconnect Specification Rev. 1.3* However, because not all of the registers defined in the *Error Management Extension* are implemented in the RapidIO MegaCore function, the error management registers are mapped into the Implementation Defined Space rather than in the Extended Features Space.

The Error Management registers providing the most useful information for error management are implemented in the RapidIO MegaCore function; see their description in "Error Management Registers" on page 4–110 for more information.

- Logical/Transport Layer Error Detect CSR
- Logical/Transport Layer Error Enable CSR
- Logical/Transport Layer Address Capture CSR
- Logical/Transport Layer device ID Capture CSR
- Logical/Transport Layer Control Capture CSR

When enabled, each error defined in the Error Management Extensions triggers the assertion of an interrupt on the `sys_mnt_s_irq` output signal of the System maintenance Avalon-MM slave interface and causes the capture of various packet header fields in the appropriate capture CSRs.

In addition to the errors defined by the RapidIO specification, each Logical layer module has its own set of error conditions that can be detected and managed.

Maintenance Avalon-MM Slave

The Maintenance Avalon-MM slave module creates request packets for the Avalon-MM transaction on its slave interface and processes the response packets that it receives. Anomalies are reported through one or more of the following three channels:

- Standard Error Management Registers
- Registers in the Implementation Defined Space
- Standard Error Management Registers

The following sections describe these channels.

Standard Error Management Registers

The following standard defined error types can be declared by the Input/Output Avalon-MM slave module. The corresponding error bits are then set and the required packet information is captured in the appropriate Error Management registers.

IO Error Response
This error is declared when a response with ERROR status is received for a pending maintenance read or write request.

Unsolicited Response
This error is declared when a response is received that does not correspond to any pending maintenance read or write request.

Packet response Time-out
This error is declared when a response is not received within the time specified by the Port Response Timeout CSR for a pending maintenance read or write request.

Illegal Transaction Decode
This error is declared for malformed received response packets.

- Response packet to pending maintenance read or write request with Status not DONE nor ERROR.
- Response packet with payload with a Transaction Type different from maintenance read response.
- Response packet without payload, with a Transaction Type different from maintenance write response.
- Response to a pending maintenance read request with more than 32-bit of payload. (The MegaCore only issues 32-bit read requests.)

Registers in the Implementation Defined Space

The Maintenance register module defines the Maintenance Interrupt register in which the following two bits report Maintenance Avalon-MM slave related error conditions.

- WRITE_OUT_OF_BOUNDS
- READ_OUT_OF_BOUNDS

These bits are set when the address of a write or read transfer on the Maintenance Avalon-MM slave interface falls outside of all the enabled address mapping windows. When these bits are set, the system interrupt signal `sys_mnt_s_irq` is also asserted if the corresponding bit in the Maintenance Interrupt Enable register is set.

Avalon-MM Slave Interface's Error Indication Signal

The `io_s_rd_readerror` output is asserted when a response with ERROR status is received for a maintenance read request packet when a maintenance read times out or when the Avalon-MM read address falls outside of all the enabled address mapping window.

Maintenance Avalon-MM Master

The Maintenance Avalon-MM master module processes the maintenance read and write request packets that it receives and generates response packets. Anomalies are reported by generating ERROR response packets. A response packet with ERROR status is generated in the following cases:

- Received a maintenance write request packet without payload or with more than 64 bytes of payload.
- Received a maintenance read request packet of the wrong size (too large or too small).
- Received a maintenance read or write request packet with an invalid rdsize or wrsize value.

These errors do not cause any of the standard defined errors to be declared and recorded in the Error Management registers.

Port-Write Reception Module

The Port-Write reception module processes received port-write request maintenance packets. The following bits in the Maintenance Interrupt register in the Implementation defined space are used to report any

detected anomaly. The System Maintenance Avalon-MM slave port interrupt signal sys_mnt_s_irq is asserted if the corresponding bit in the Maintenance Interrupt Enable register is set.

- The PORT_WRITE_ERROR bit is set when the packet is either too small (no payload) or too large (more than 64 bytes of payload), or if the actual size or the packet is larger than indicated by the wrsize field. These errors do not cause any of the standard defined errors to be declared and recorded in the Error Management registers.
- The PACKET_DROPPED bit is set when a port-write request packet is received but port-write reception is not enabled (by setting bit PORT_WRITE_ENA in the Rx PORT_WRITE_CONTROL register) or if a previously received port-write has not been read out from the Rx PORT_WRITE register.

Port-Write Transmission Module

There are no response packets to Port-Write requests, therefore the Port-Write transmission module does not detect or report any errors.

Input/Output Avalon-MM Slave

The Input/Output Avalon-MM slave module creates request packets for the Avalon-MM transaction on its read and write slave interfaces and processes the response packets that it receives. Anomalies are reported through one or more of the following three channels:

- Standard Error Management Registers
- Registers in the Implementation Defined Space
- The Avalon-MM slave interface's error indication signal.

Standard Error Management Registers

The following standard defined error types can be declared by the Input/Output Avalon-MM master. The corresponding bits then get set and the required packet information gets captured in the appropriate Error Management registers.

IO Error Response
This error is declared when a response with ERROR status is received for a pending NREAD or NWRITE_R request.

Unsolicited Response
This error is declared when a response is received that does not correspond to any pending NREAD or NWRITE_R request.

Packet Response Time-Out
This error is declared when a response is not received within the time specified by the `Port Response Timeout` CSR for an NREAD or NWRITE_R request.

Illegal Transaction Decode
This error is declared for malformed received response packets like those outlined below:

- NREAD or NWRITE_R response packet with Status not DONE nor ERROR.
- NWRITE_R response packet with payload or with a Transaction Type indicating the presence of a payload.
- NREAD response packet without payload, with incorrect payload size, or with a Transaction Type indicating absence of payload.

Registers in the Implementation-Defined Space

The Input/Output Avalon-MM slave module defines the Input/Output Slave Interrupt registers with the following bits. See these register descriptions in the "Input/Output Slave Mapping Registers" on page 4–107 for details on when these bits get set.

- `INVALID_WRITE_BYTEENABLE`
- `INVALID_WRITE_BURSTCOUNT`
- `WRITE_OUT_OF_BOUNDS`
- `READ_OUT_OF_BOUNDS`

When any of these bits are set, the system interrupt signal `sys_mnt_s_irq` is also asserted if the corresponding bit in the `Input/Output Slave Interrupt Enable` register is set.

The Avalon-MM slave interface's error indication signal

The `io_s_rd_readerror output` is asserted when a response with ERROR status is received for a NREAD request packet, when an NREAD request times out or when the Avalon-MM address falls outside of all the enabled address mapping window.

Input/Output Avalon-MM Master

The Input/Output Avalon-MM master module processes the request packets that it receives and generates response packets when required. Anomalies are reported through one or both of the following two channels:

- Standard Error Management Registers
- Response Packets with ERROR Status

Standard Error Management Registers

The following two standard-defined error types can be declared by the Input/Output Avalon-MM master. The corresponding bits are then set and the required packet information is captured in the appropriate error management registers.

Unsupported Transaction
This error is declared when a request packet carrying a transaction type that is not supported in the `Destination Operations CAR`: ATOMIC transaction types and reserved and implementation defined transaction types.

Illegal Transaction Decode
This error is declared when a request packet for a supported transaction is too short or if it contains illegal values in some of its fields:

- Request packet with priority = 3
- NWRITE or NWRITE_R request packets without payload
- NWRITE or NWRITE_R request packets with reserved wrsize and wdptr combination
- NWRITE, NWRITE_R, SWRITE, or NREAD request packets for which the address does not match any enabled address mapping window.
- NREAD request packet with payload
- NREAD request with rdsize that correspond to non-all-ones Byte Lanes. (Avalon-MM does not allow non-all-ones byteenable on read transfers so the Read Avalon-MM master does not have a byteenable signal.)
- Payload size does not match what is indicated by the rdsize or wrsize and wdptr fields

Response Packets with ERROR Status

An ERROR response packet is sent for NREAD and NWRITE_R and Type 5 ATOMIC request packets that cause an Illegal Transaction Decode error to be declared. It also is sent for NREAD requests for which the Avalon-MM read transfers completes with the assertion of the `io_m_rd_readerror` input signal.

Avalon-ST Pass-Through Port

Packets with valid CRCs that are not recognized as being destined to one of the implemented logical layer modules are passed to the Avalon-ST Pass-Through port for processing by user logic. In variations where the Avalon-ST Pass-Through port is not implemented, an Unsupported Transaction error is declared, if enabled, and the packet's information is captured in the error management registers.

The RapidIO MegaCore function also provides hooks for user logic to report any error detected by a user-implemented logical layer module attached to the Avalon-ST pass-through port.

The transmit side of the Avalon-ST Pass-Through port provides the `gen_tx_error` input signal that behaves essentially the same way as the `atxerr` input signal described in the Physical layer section above.

If the Avalon-ST pass-through port is enabled and at least one of the **Data Messages Source** or **Destination** operations is turned on in the MegaWizard Plug-In Manager interface, the ports in Table 4–13 are added to the MegaCore function to enable integrated error management.

Table 4–13. Message Passing Error Management Input Ports

Port	Bit	Description
Message Passing Error Management Inputs		
`error_detect_message_error_response`		Sets the `Message error response` bit in the `Logical/Transport Layer Error Detect CSR` and triggers capture into the Error Management registers of the captured fields below.

Table 4–13. Message Passing Error Management Input Ports

Port	Bit	Description
error_detect_message_format_error		Sets the Message Format Error bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_detect_message_request_timeout		Sets the Message Request Time-out bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_detect_packet_response_timeout		Sets the Message Request Time-out bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_capture_letter	[1:0]	Field captured into the Logical/Transport Layer Control Capture CSR
error_capture_mbox	[1:0]	Field captured into the Logical/Transport Layer Control Capture CSR
error_capture_msgseg_or_xmbox	[3:0]	Field captured into the Logical/Transport Layer Control Capture CSR.
Common Error Management Inputs		
error_detect_illegal_transaction_decode		Sets the Illegal transaction decode bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_detect_illegal_transaction_target		Sets the Illegal transaction target error bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_detect_packet_response_timeout		Sets the Packet response Time-out bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_detect_unsolicited_response		Sets the Unsolicited Response bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.

Table 4–13. Message Passing Error Management Input Ports

Port	Bit	Description
error_detect_unsupported_transaction		Sets the Unsupported Transaction bit in the Logical/Transport Layer Error Detect CSR and triggers capture into the Error Management registers of the captured fields below.
error_capture_ftype	[3:0]	Field captured into Control Error Management register.
error_capture_ttype	[3:0]	Field captured into Control Error Management register.
error_capture_destination_id	[15:0]	Field captured into deviceID Error Management register.
error_capture_source_id	[15:0]	Field captured into deviceID Error Management register.

Demonstration Testbench Description

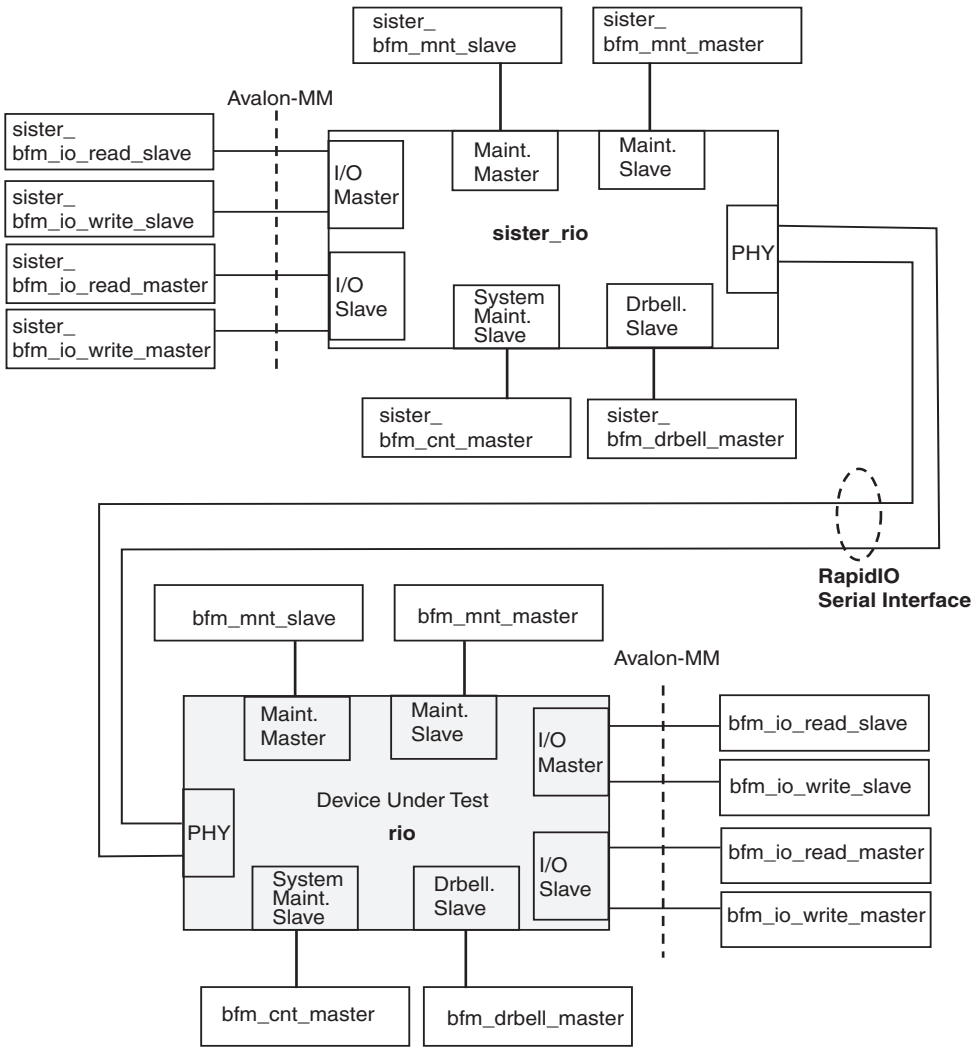
This generated testbench instantiates two symmetrical versions of a RapidIO MegaCore generated by the MegaWizard interface. One instance is the Device Under Test (DUT), referred to as rio. The other instance is the sister_rio. The sister_rio responds to transactions initiated by the rio DUT, and generates transactions to which the DUT responds. Both the sister_rio and the rio DUT have bus functional models (BFMs) that generate and monitor traffic from all the variations of Avalon-MM interfaces and generate transactions to which the DUT responds.

See Figure 4–23 for a block diagram of the testbench in which all of the available Avalon-MM interfaces are enabled. The two cores communicate with each other using the Serial RapidIO interface. The testbench initiates the following transactions at the rio DUT and targets them at the sister_rio:

- Maintenance Writes and Reads
- SWRITE
- NWRITE_R
- NWRITE
- NREAD
- Doorbell Messages
- Maintenance Port Writes and Reads

Your specific variant may not have all of the interfaces enabled. In variations with an Avalon-ST pass-through port, Type 9 (data streaming) packet traffic is generated and monitored on the DUT and the sister_rio's Avalon-ST pass-through interfaces respectively.

Figure 4–23. Transport IO Logical Layer Testbench



The diagram represents the HDL written in the file <*design_name*>_hookup.iv. Activity across the Avalon-MM interfaces is generated and checked by invoking tasks that are defined in the supplied BFMs. These models are implemented in files <*design_name*>_avalon_bfm_master.v and <*design_name*>_avalon_bfm_slave.v.

The file <*design_name*>_tb.v implements the code that performs the above mentioned transactions. The code performs a reset and initialization sequence necessary for the rio and sister_rio MegaCores to establish a link and exchange packets.

Reset, Initialization, and Configuration

The clocks that are used to drive the testbench are defined and generated in the <*design_name*>_hookup.iv file.

Refer to <*design_name*>_hookup.iv for the exact frequencies used for each of the clocks. The frequencies used depend on the configuration of the variant, for example, 1x or 4x mode, data rate selection, and width of internal datapath.

The reset sequence is simple – the main reset for the DUT and the sister core is driven low at the beginning of the simulation and kept low for a duration of 100 ns and is then deasserted.

After the reset is deasserted, the testbench waits until both the rio and the sister_rio have driven their port_initialized output signals high. This indicates that both cores have completed their initialization sequence.

The next step is to perform some basic programming of internal registers in the rio and the sister_rio. Table 4–14 shows the registers that are programmed in both rio and sister_rio cores. For a full description of each register see "Software Interface" on page 4–86.

Table 4–14. Testbench Registers

Module	Register Address	Register Name	Value	Description
rio	0x00060	Base device ID CSR	0x00AA_0000	Program the rio to have a base device ID of 0xAA
rio	0x0013C	General Control CSR	0x6000_0000	Enable Request packet generation.
sister_rio	0x00060	Base device ID CSR	0x0055_0000	Program the sister rio to have a base device ID of 0x55
sister_rio	0x0013C	General Control CSR	0x6000_0000	Enable Request packet generation for the sister_rio.
rio	0x1040C	I/O Slave Window 0 Control	0x0055_0000	Set the `destinationID` for outgoing transactions to a value 0x55. This matches the base device ID set for the sister rio.
rio	0x10404	I/O Slave Window 0 Mask	0x0000_0004	Define the Input/Output Avalon-MM Slave Window 0 to cover the whole address space (mask set to all zeros) and enable it.
sister_rio	0x10504	I/O Slave Interrupt	0x0000_000F	Enable the Input/Output Slave interrupts
sister_rio	0x10304	I/O Master window	0x0000_0004	Enable the sister_rio I/O master window 0. This will allow the sister_rio to receive I/O transactions.
rio	0x1010C	TX Maintenance Window 0 Control	0x0055_FF00	Set the `destinationID` for outgoing Maintenance Packets to 0x55. This matches the base device id set for the sister_rio. Set the hop count to 0xFF.
rio	0x10104	TX Maintenance Window 0 Mask	0x0000_0004	Enable the rio TX Maintenance window 0

Programming of the device under test RapidIO MegaCore function variation registers, in this case, rio registers, is performed by invoking read and write tasks that are defined in the BFM instance, `bfm_cnt_master`. Programming of the device under test RapidIO MegaCore function variation, in this case, sister rio, is performed by invoking the read and write tasks that are defined in the BFM instance, `sister_bfm_cnt_master`. For the exact parameters passed on to these tasks, please see the file <*design_name*>`_tb.v`. The tasks will drive either a write or read transaction across the System Maintenance Slave Avalon-MM Interface.

With the above configuration the cores can exchange basic packets across the serial link.

Maintenance Write and Read Transactions

The first set of tests performed are maintenance write and read requests. The the device under test RapidIO MegaCore function variation (the rio module) sends two maintenance write requests to the device under test RapidIO MegaCore function variation (`sister _rio` module). The writes are performed by invoking the `rw_addr_data` task defined inside the BFM instance, `bfm_mnt_master`. The `bfm_mnt_master` is an instance of the module `avalon_bfm_master` which is defined in the file <*design_name*>`_avalon_bfm_master.v`

The following parameters are passed to the task:

- WRITE – transaction type to be executed
- wr_address – address to be driven on the Avalon-MM address bus
- wr_data – write data to be driven on the Avalon-MM Write Data bus

The task performs the write transaction across the Maintenance Write Avalon-MM Slave Interface.

The device under test RapidIO MegaCore function variation then sends two maintenance read requests to the sister RapidIO MegaCore function variation. The reads are performed by invoking the `rw_data task` defined inside the BFM instance, `bfm_mnt_master`. The following parameters are passed to the task:

- READ – transaction type to be executed
- `rd_address` – address to be driven on the Avalon-MM address bus
- `rd_data` – parameter which will store the data read across the Avalon-MM Read Data bus

The task performs the read transaction across the Maintenance Read Avalon-MM Slave Interface.

The write transaction across the Avalon-MM interface will be translated into a RapidIO Maintenance Write request packet. Likewise, the read transaction across the Avalon-MM Interface will be translated into a RapidIO Maintenance Read request packet.

The Maintenance write and read request packets are received by the sister RapidIO MegaCore function variation and translated into Avalon-MM transactions that are presented across the sister RapidIO MegaCore function variation Maintenance Master Avalon-MM Interface. An instance of `avalon_bfm_slave`, the BFM for an Avalon-MM Slave, is driven by this interface. In the testbench, the write and read transactions are checked and data is returned for the read operation. The write

operation is checked by invoking the `read_writedata` task of the BFM. The tasks returns the write address and the write data. This information is then sent to `expect` functions which check for data integrity. The read operation is checked by invoking the `write_read data` task. This task returns the write address and drives the return data and read control signals on the Avalon-MM master read port of the sister RapidIO MegaCore function variation. The write address is checked again for expected values by calling an `expect` function.

SWRITE Transactions

The next set of operations performed are Streaming Writes. In order to perform SWRITE operations, one register in the core must be reconfigured, as shown in Table 4–15.

Table 4–15. SWRITE Register

Module	Register Address	Name	Value	Description
rio	0x1040C	IO Slave Window 0 Control	0x0055_0002	Sets the Destination ID for outgoing transactions to a value 0x55. This matches the base device id set for the sister_rio. Enables SWRITE operations.

With the above setting, any write operation presented across the Input/Output Slave Avalon-MM Interface on the rio will be translated into a RapidIO Streaming Write transaction.

The Avalon-MM write address must map into the Input/Output Slave Window 0. However, in this example the window is set to cover the whole Avalon-MM address space by setting the mask to all zeros.

The testbench generates a predetermined series of burst writes across the Avalon-MM Slave I/O interface on the device under test RapidIO MegaCore function variation. These write bursts are each converted into an SWRITE request packet sent on the RapidIO serial interface. Because Streaming Writes only support bursts that are a multiple of double words long, (multiple of 8-bytes), the testbench cycles from 8 to `MAX_WRITTEN_BYTES` in steps of 8 bytes. Two tasks are invoked to carry out the burst writes, `rw_addr_data` and `rw_data`. The `rw_addr_data` initiates the burst by providing the *address*, *burstcount* and the content of the first data word, while the `rw_data` executes the remaining of the burst.

At the sister RapidIO MegaCore function variation, the SWRITE request packets are received and translated into Avalon-MM transactions that are presented across the Input/Output Master Avalon-MM interface. The testbench calls the task `read_writedata` of the `sister_bfm_io_write_slave`. The task captures the written data.

The written data is then checked again the expected value by invoking an `expect` task. After completing the SWRITE tests, the testbench performs NWRITE_R operations.

NWRITE_R Transactions

In order to perform NWRITE_R operations, one register in the core must be reconfigured, as shown in Table 4–16.

Table 4–16. NWRITE_R Transactions

Core	Register Address	Name	Value	Description
rio	0x1040C	Input/Output Slave Window 0 control	0x0055_0001	Sets the Destination Id for outgoing transactions to a value 0x55. This matches the base device id set for the sister rio. Enables NWRITE_R operations.

With the above setting, any write operation presented across the Input/Output Avalon-MM Slave module's Avalon-MM write interface will be translated into a RapidIO NWRITE_R transaction. The Avalon-MM write address must map into the II/O Slave Window 0.

Initially, the testbench performs two single word bursts; writing to an even word address first and then to an odd word address. The testbench then generates a predetermined series of burst writes across the Input/Output Avalon-MM Slave module's Avalon-MM write interface on the device under test RapidIO MegaCore function variation. These write bursts are each converted into a NWRITE_R request packets sent over the RapidIO Serial Interface. The testbench cycles from 8 to MAX_WRITTEN_BYTES in steps of 8 bytes. Two tasks are invoked to carry out the burst writes, `rw_addr_data` and `rw_data`. The file `rw_addr_data` initiates the burst while the `rw_data` executes the remaining of the burst.

At the sister RapidIO MegaCore function variation, the NWRITE_R request packets are received and presented across the I/O Master Avalon-MM interface as write transactions. The testbench calls the task

read_writedata of the sister_bfm_io_write_slave. The task captures the written data. The written data is checked again the expected value by invoking an expect task.

NWRITE Transactions

In order to perform NWRITE operations, one register in the core must be reconfigured, as shown in Table . With these settings, any write operation presented across the Input/Output Slave Avalon-MM Interface will be translated into a RapidIO NWRITE transaction.

Table 4–17. NWRITE Transactions

Module	Register Address	Name	Value	Description
rio	0x1040C	I/O Slave Window 0 Control	0x0055_0000	Sets the destinationID for outgoing transactions to a value 0x55. This matches the base deviceID set for the device under test RapidIO MegaCore function variation. Sets the write request type back to NWRITE.

The Avalon-MM write address must map into the Input/Output Slave Window 0.

Initially, the testbench performs two single word bursts; writing to an even word address first and then to an odd word address. The testbench then generates a predetermined series of burst writes across the Input/Output Avalon-MM Slave module's Avalon-MM write interface on the device under test RapidIO MegaCore function variation. These write bursts are each converted into a NWRITE request packet that is send over the RapidIO Serial Interface. The testbench cycles from 8 to MAX_WRITTEN_BYTES in steps of 8 bytes. Two tasks are invoked to carry out the burst writes, rw_addr_data and rw_data. The rw_addr_data initiates the burst while rw_data executes the remaining of the burst.

At the sister RapidIO MegaCore function variation, the NWRITE operations are received and presented across the I/O Master Avalon-MM slave interface as write transactions. The testbench calls the task read_writedata of the sister_bfm_io_write_slave. The task captures the written data. The written data is checked again the expected value by invoking an expect task.

NREAD Transactions

The next set of transactions tested are NREADs. The device under test RapidIO MegaCore function variation sends a group of NREADs to the the sister RapidIO MegaCore function variation by cycling the read burst size from 8 to `MAX_READ_BYTES` in increments of 8 bytes. For each iteration, the `rw_addr_data` task is called. This task is defined in the `bfm_io_read_master` instance of the Avalon-MM Master BFM. The task performs the read request packets across the I/O Avalon-MM Slave Read Interface. The read transaction across the Avalon-MM Interface will be translated into a RapidIO NREAD request packets. The values of the `rd_address`, `rd_byteenable`, and `rd_burstcount` parameters determine the values for the `rdsize`, `wdptr` and `xamsbs` fields in the header of the RapidIO packet.

The NREAD request packets are received by the device under test RapidIO MegaCore function variation and translated into Avalon-MM read transaction that are presented across the sister RapidIO MegaCore function variation's I/O Master Avalon-MM Interface. An instance of `avalon_bfm_slave`, the BFM for an Avalon-MM Slave, is driven by this interface. The read operations are checked and data is returned by calling the task, `write_readdata`. This task drives the `data` and `read` datavalid control signals on the Avalon-MM master read port of the device under test RapidIO MegaCore function variation.

The returned data is expected at the device under test RapidIO MegaCore function variation I/O Avalon-MM Slave Interface. The task `rw_data` is called and it captures the read data. This task is defined inside the instance of `bfm_io_read_master`. The read data and an expected value is then sent to the `expect` task where it is checked for equality.

Doorbell Transactions

The first step in testing doorbell messages is to enable the doorbell interrupts. You do this by setting the lower three bits in the `Doorbell Interrupt Enable` register located at address 0x0000_0020. The testbench invokes the task `rw_addr_data` which is defined in the instance of `bfm_drbell_s_master`. The test also programs the core to store all of the successful and unsuccessful doorbell messages in the TX Completion FIFO. For more information, see the register"Tx Doorbell Status Control– Offset: 'h1C" on page 4–116.

Next, the test proceeds to push eight doorbell messages into the Transmit Doorbell Message FIFO of the device under test. The test increments the message payload for each transaction. To achieve this, the task `rw_addr_data` is invoked with a WRITE operation, and is passed the

TX doorbell register address, 0x0000_000C. This programs the 16-bit message, an incrementing payload in this example, as well as the destinationID, 0x55, to be used in the doorbell transaction packet.

To verify that the doorbell request packets have been sent, the test waits for the drbell_s_irq signal to be asserted. The test then reads the "Tx Doorbell Completion– Offset: 'h14" on page 4–115. This register provides the doorbell messages that have been added to the TX Completion FIFO. There should be eight successfully completed doorbell messages in that FIFO and each one should be accessible by reading the TX Doorbell Completion Control register eight times in succession. To perform this verification, the test invokes the task rw_data defined in the instance bfm_drbell_s_master.

The last part of the doorbell test programs the device under test RapidIO MegaCore function variation to send eight doorbell messages to the device under test RapidIO MegaCore function variation. The test verifies that all eight doorbell messages were received by the device under test RapidIO MegaCore function variation. The test calls the task rw_addr_data defined under the instance sister_bfm_drbell_s_master. The task performs a write to the register "Tx Doorbell – Offset: 'h0C". It programs the payload to be incrementing, starting at 0c01, and the destinationID to be 0xAA, matching the deviceID of the device under test RapidIO MegaCore function variation.

The test waits for the device under test RapidIO MegaCore function variation to assert the drbell_s_irq signal which indicates that a doorbell message has been received. The test then proceeds to read the eight received doorbell messages. To accomplish this, the task rw_data is called with a READ operation and an address of 0x0000_0000 which addresses the RX Doorbell register. The task is called eight times, once for each message. It returns the received doorbell message and the message is checked for an incrementing payload starting at 0x0c01 and for the sourceId to have a value of 0x55, which is the deviceID of the sister RapidIO MegaCore function variation .

Port Write Transactions

To test Port Writes, the test performs some basic configuration of the Port Write registers in the device under test RapidIO MegaCore function variation and the sister RapidIO MegaCore function variation. It then programs the device under test RapidIO MegaCore function variation to transmit Port Write request packets to the sister RapidIO MegaCore function variation. The Port Writes are received by the sister RapidIO MegaCore function variation and retrieved by the test program.

The configuration involves enabling the RX Packet Stored Interrupt in the sister RapidIO MegaCore function variation. This enables the sister RapidIO MegaCore function variation to assert the `sister_sys_mnt_s_irq` signal, which indicates that an interrupt has been set in either the Maintenance Interrupt Register or the Input/Output Slave Interrupt Register. Because this is testing Port Writes, the assertion of `sister_sys_mnt_s_irq` means that a Port Write has been received and that the payload can be retrieved. Enabling the interrupt is accomplished by calling the task `rw_addr_data` defined in the instance, `sister_bfm_cnt_master`.

A WRITE operation is performed by the task with the following address and data passed as parameters:`17'h10084, 32'h10`. In addition, the sister RapidIO MegaCore function variation has to be enabled to receive Port Write transactions from the `rio`. Again, the task is called and the following address and data are passed: `17'h10250, 32'h1`.

After the configuration is complete, the test performs the operations listed in Table 4–18.

Table 4–18. Port Write Test

Operation	Action
Places data into the TX_PORT_WRITE_BUFFER	write incrementing payload to registers at addresses `'h10210` to `'h1024C`
Indicates to the device under test RapidIO MegaCore function variation that Port Write Data is ready	write `Destination_ID = 0x55` and `PACKET_READY = 0x1` to `'h10200`
Waits for the sister RapidIO MegaCore function variation to receive the Port Write	monitor `sister_sys_mnt_s_irq`
Verifies that the sister RapidIO MegaCore function variation has the interrupt bit PACKET_STORED set	read register at address `'h10080`
Retrieves the Port Write payload from the sister RapidIO MegaCore function variation and checks for data integrity	read registers at addresses `'h10260` to `'h1029C`
Checks the sister RapidIO MegaCore function variation PORT_WRITE status for correct payload size	read register at address `'h10254`
Clears the PACKET_STORED interrupt in the sister RapidIO MegaCore function variation	write1 to bit 4 of register at address `'h10080`
Waits for the next interrupt at the sister RapidIO MegaCore function variation	monitor `sister_sys_mnt_s_irq`

The test iterates through these operations, each time incrementing the payload of the Port Write. The loop exits when the max payload for a Port Write has been transmitted, 64 Bytes.

All of the operations in the loop are executed by invoking the task `rw_addr_data` either in the `bfm_cnt_master` or the `sister_bfm_cnt_master` instances.

Transactions Across the Avalon-ST Pass-Through Interface

The IO Logical Layer Testbench tests the Avalon-ST pass-through interface by exchanging Type 9 (Data Streaming) traffic between the rio device under test and the sister_rio MegaCore function.

The supplied testbench is not used to perform a complete verification of the RapidIO MegaCore function. The full verification is performed by the Altera IP Verification Team using both directed and random verification methodologies. The purpose of the supplied testbench is to provide examples of how to program the MegaCore function and how to drive the Avalon-MM interfaces to generate RapidIO Transactions.

Parameters

Table 4–19 shows the RapidIO MegaCore function Transport, Maintenance, I/O, and Doorbell parameters, which can only be set in the MegaWizard interface (see "Parameterize" on page 2–7).

Table 4–19. Transport & I/O Logical Layer Parameters

Parameter	Values	Description
Transport Layer	No Transport layer or Transport layer	The Transport layer is required for the Maintenance, Input/Output, and Doorbell Logical layers, or the Avalon-ST pass-through port to be enabled. If you select **No** Transport Layer, you choose to use a Physical layer-only variation.
Pass-through Avalon-ST port	On or Off	The Transport layer routes all unrecognized packets to the Avalon-ST pass-through port. Unrecognized packets are those that contain ftypes for Logical layers not enabled in this MegaCore function, or destination IDs not assigned to this endpoint.
Maintenance logical layer interface(s)	Avalon-MM Master and Slave, Avalon-MM Master, Avalon-MM Slave, or None	Selects which parts of the Maintenance logical layer to implement.
Number of transmit address translation windows	1 to 16	Only applicable if an Avalon-MM Slave is chosen as the Maintenance logical layer interface.
I/O logical layer interface(s)	Avalon-MM Master and Slave, Avalon-MM Master, Avalon-MM Slave, or None	Selects whether or not to add a master/slave Avalon-MM interface.
Input/Output Slave address width	25 to 32	Specifies the Input/Output Slave address width. The default is **25**.
Number of RX address translation windows (for the Avalon-MM Master interface)	1 to 16	Only applicable if I/O Avalon-MM Master is chosen as the **I/O logical layer interface**.
Number of TX address translation windows (for the Avalon-MM Slave interface)	1 to 16	Only applicable if I/O Avalon-MM Slave is chosen as the **I/O logical layer interface**.
Doorbell TX enable	On or Off	Enables generation of outbound doorbell message transmission
Doorbell RX enable	On or Off	Enables processing of inbound doorbell messages. If not enabled, received doorbell messages are routed to the Avalon-ST pass-through port if it is enabled, or silently dropped if it is not.

Signals

Tables 4–20 through 4–31 list the pins used by the transport, maintenance, Input/Output, and Doorbell Logical layer modules of the RapidIO MegaCore function. For a list of descriptions of signals used and generated by the Physical Layer, see "Signals" on page 3–27. The active-low signals are indicated by the suffix underscore n (_n).

For signals and bus widths specific to your variation, refer to the HTML file generated by the MegaWizard interface (see Table 2–2 on page 2–18).

Table 4–20 lists the clock and reset signals used by all modules.

Table 4–20. Clock & Reset Signals

Signal	Direction	Description
sysclk	Input	System clock.
reset_n	Input	Active-low system reset.

Tables 4–21 through 4–28 list the Avalon-MM interface signals.

Table 4–21. Maintenance Master Avalon-MM Interface Signals

Signal	Direction	Description
mnt_m_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally to sample this interface.
mnt_m_waitrequest	Input	Wait request
mnt_m_read	Output	Read enable
mnt_m_write	Output	Write enable
mnt_m_address[31:0]	Output	Address bus
mnt_m_writedata[31:0]	Output	Write data bus
mnt_m_readdata[31:0]	Input	Read data bus
mnt_m_readdatavalid	Input	Read data valid

Table 4–22. Maintenance Slave Avalon-MM Interface Signals (Part 1 of 2)

Signal	Direction	Description
mnt_s_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface.
mnt_s_chipselect	Input	Maintenance slave chip select

Table 4–22. Maintenance Slave Avalon-MM Interface Signals (Part 2 of 2)

Signal	Direction	Description
mnt_s_waitrequest	Output	Maintenance slave Wait request
mnt_s_read	Input	Maintenance slave Read enable
mnt_s_write	Input	Maintenance slave Write enable
mnt_s_address[25:0]	Input	Maintenance slave Address bus
mnt_s_writedata[31:0]	Input	Maintenance slave Write data bus
mnt_s_readdata[31:0]	Output	Maintenance slave Read data bus
mnt_s_readdatavalid	Output	Maintenance slave Read data valid
mnt_s_readerror	Output	Maintenance slave Read error. Indicates that the read transfer did not complete successfully.

Table 4–23. System Maintenance Slave Avalon-MM Interface Signals

Signal	Direction	Description
sys_mnt_s_clk	Input	System clock
sys_mnt_s_chipselect	Input	System maintenance slave chip select
sys_mnt_s_waitrequest	Output	System maintenance slave wait request
sys_mnt_s_read	Input	System maintenance slave read enable
sys_mnt_s_write	Input	System maintenance slave write enable
sys_mnt_s_address[16:0]	Input	System maintenance slave address bus
sys_mnt_s_writedata[31:0]	Input	System maintenance slave write data bus
sys_mnt_s_readdata[31:0]	Output	System maintenance slave read data bus
sys_mnt_s_irq	Output	System maintenance slave interrupt request

Table 4–24. Input/Output Master Data Path Write Avalon-MM Interface Signals

Signal	Direction	Description
io_m_wr_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface.
io_m_wr_waitrequest	Input	Input/Output master Wait request
io_m_wr_write	Output	Input/Output master Write enable
io_m_wr_address[31:0]	Output	Input/Output master Address bus
io_m_wr_writedata[*n*:0]	Output	Input/Output master Write data bus
io_m_wr_byteenable[*m*:0]	Output	Input/Output master Byte enable
io_m_wr_burstcount[*k*:0]	Output	Input/Output master Burst count

Parameters *n*, *m*, and *k* are used in some of the following tables, as follows:
n = (**Internal data path width** -1)
m = (**Internal data path width**/8)
k = 6 for 32 bit **Internal data path width**, and 5 for 64 bit **Internal data path width**

Table 4–25. Input/Output Master Data Path Read Avalon-MM Interface Signals

Signal	Direction	Description
io_m_rd_waitrequest	Input	Input/Output master Wait request
io_m_rd_read	Output	Input/Output master Read enable
io_m_rd_address[31:0]	Output	Input/Output master Address bus
io_m_rd_readdata[*n*:0]	Input	Input/Output master Read data bus
io_m_rd_readdatavalid	Input	Input/Output master Read data valid
io_m_rd_burstcount[*k*:0]	Output	Input/Output master Burst count
io_m_rd_readerror	Input	Input/Output master Indicates that the burst read transfer did not complete successfully

Table 4–26. Input/Output Slave Data Path Write Avalon-MM Interface Signals

Signal	Direction	Description
io_s_wr_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface.
io_s_wr_chipselect	Input	Input/Output slave Chip select
io_s_wr_waitrequest	Output	Input/Output slave Wait request
io_s_wr_write	Input	Input/Output slave Wrtie enable
io_s_wr_address[31:0]	Input	Input/Output slave Address bus
io_s_wr_writedata[*n*:0]	Input	Input/Output slave Write data bus
io_s_wr_byteenable[*m*:0]	Input	Input/Output slave Byte enable
io_s_wr_burstcount[*k*:0]	Input	Input/Output slave Burst count

Table 4–27. Input/Output Slave Data Path Read Avalon-MM Interface Signals

Signal	Direction	Description
io_s_rd_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface
io_s_rd_chipselect	Input	Input/Output slave Chip select
io_s_rd_waitrequest	Output	Input/Output slave Wait request
io_s_rd_read	Output	Input/Output slave Read enable
io_s_rd_address[31:0]	Input	Input/Output slave Address bus
io_s_rd_readdata[*n*:0]	Output	Input/Output slave Read data bus
io_s_rd_readdatavalid	Output	Input/Output slave Read data valid
io_s_rd_burstcount[*k*:0]	Input	Input/Output slave Burst count
io_s_rd_readerror	Output	Input/Output slave read error indicates that the burst read transfer did not complete successfully

Table 4–28. Input/Output Slave Data Path Read Avalon-MM Interface Signals

Signal	Direction	Description
io_s_rd_clk	Input	This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface
io_s_rd_chipselect	Input	Input/Output slave Chip select
io_s_rd_waitrequest	Output	Input/Output slave Wait request
io_s_rd_read	Output	Input/Output slave Read enable
io_s_rd_address[31:0]	Input	Input/Output slave Address bus
io_s_rd_readdata[*n*:0]	Output	Input/Output slave Read data bus
io_s_rd_readdatavalid	Output	Input/Output slave Read data valid
io_s_rd_burstcount[*k*:0]	Input	Input/Output slave Burst count
io_s_rd_readerror	Output	Input/Output slave read error indicates that the burst read transfer did not complete successfully

Table 4–29. Doorbell Message Avalon-MM Slave Interface Signals

Signal	Direction	Description
drbell_s_clk	Input	Doorbell Avalon-MM clock. This signal is not used, therefore it can be left open. The sysclk signal is used internally as the clock reference for this interface
drbell_s_chipselect	Input	Doorbell chip select
drbell_s_write	Input	Doorbell write
drbell_s_read	Input	Doorbell read
drbell_s_address[2:0]	Input	Doorbell address
drbell_s_writedata	Input	Doorbell write data
drbell_s_readdata	Output	Doorbell read data
drbell_s_waitrequest	Output	Doorbell wait request
drbell_s_irq	Output	Doorbell interrupt

Tables 4–30 and 4–31 list the Avalon-ST pass-through interface signals..

Table 4–30. Avalon-ST Pass-Through Port Tx Signals (Part 1 of 2)

Signal	Direction	Description
gen_tx_ready	Output Sink to Source	This ready signal is asserted by the Avalon-ST sink to mark ready cycles, which are the cycles in which transfers may take place. If ready is asserted on cycle, the cycle N (N+READY_LATENCY) is a ready cycle. In the RapidIO MegaCore function, READY_LATENCY is equal to 1 so the cycle immediately following the cycle on which gen_tx_ready is asserted is the ready cycle.
gen_tx_valid	Input Source to Sink	The valid signal qualifies valid data on any cycle in which data is being transferred from the source to the sink. On each ready cycle for which gen_tx_valid is asserted the data signal and other source to sink signals are sampled by the sink. Ready cycles for which the valid signal is asserted are called active cycles.
gen_tx_startofpacket	Input Source to Sink	The gen_tx_startofpacket signal marks the active cycle containing the start of the packet.
gen_tx_endofpacket	Input Source to Sink	The gen_tx_endofpacket signal marks the active cycle containing the end of the packet.
gen_tx_data	Input Source to Sink	This data signal carries the bulk of the information being transferred from the source to the sink.

Table 4–30. Avalon-ST Pass-Through Port Tx Signals (Part 2 of 2)

Signal	Direction	Description
gen_tx_empty	Input Source to Sink	The empty signal indicates the number of symbols that are empty during the cycles that mark the end of a packet. The sink only checks the value of the empty signal during active cycles that have gen_tx_endofpacket asserted. The empty symbols are always the last symbols in data, those carried by the low-order bits. The empty signal is required on all packet interfaces whose data signal carries more than one symbol of data and has a variable length packet format. The size of the empty signal is log2(Symbols_Per_Beat).
gen_tx_error	Input Source to Sink	Errors are signaled with this error signal. A value of zero on any beat indicates the data on that beat is error-free. If gen_tx_error is asserted at any point in a packet, the whole packet is dropped.

Table 4–31. Avalon-ST Pass-Through Port Rx Signals (Part 1 of 2)

Signal	Direction	Description
gen_rx_ready	Input Source to Sink	This ready signal is asserted by the sink to mark ready cycles, which are cycles in which transfers can take place. If ready is asserted on cycle, the cycle (N+READY_LATENCY) is a ready cycle. The RapidIO MegaCore function is designed for a READY_LATENCY equal to 1.
gen_rx_valid	Output Source to Sink	The valid signal qualifies valid data on any cycle in which data is being transferred from the source to the sink. On each ready cycle for which the valid signal is asserted, the data signal and other source-to-sink signals are sampled by the sink. Ready cycles for which the valid signal is asserted are called active cycles.
gen_rx_startofpacket	Output Source to Sink	The gen_rx_startofpacket signal marks the active cycle containing the start of the packet.
gen_rx_endofpacket	Output Source to Sink	The gen_rx_endofpacket signal marks the active cycle containing the end of the packet.
gen_rx_data	Output Source to Sink	The data bus carries the bulk of the information being transferred from the source to the sink.

Table 4–31. Avalon-ST Pass-Through Port Rx Signals (Part 2 of 2)

Signal	Direction	Description
gen_rx_empty	Output Source to Sink	This bus indicates the number of symbols that are empty during the cycles that mark the end of a packet. The sink only checks the value of the empty signal during active cycles that have gen_rx_endofpacket asserted. The empty symbols are always the last symbols in data; those carried by the low-order bits. The empty signal is required on all packet interfaces in which the data signal carries more than one symbol of data and has a variable length packet format. The size of the empty signal is log2(SYMBOLS_PER_BEAT).
gen_rx_error	Output Source to Sink	Errors are signaled with this signal. A value of zero on any beat indicates the data on that beat is error-free. This signal is never asserted by the RapidIO MegaCore function.
gen_rx_size	Output	This signal indicates the number of beats that will be required to transfer the packet. This signal is only valid on the start of packet beat.

Table 4–32 shows a packet and error monitoring signal used only if you use the Transport and I/O Logical layer in your design. For additional packet and error monitoring signals, see Table 3–10 (serial interface application).

Table 4–32. Packet and Error Monitoring Signals

Signal	Direction	Description
rx_packet_dropped	Output	Pulsed high one sysclk clock cycle when a received packet is dropped by the Transport layer. Received packets are only dropped if the Avalon-ST pass-through port is not enabled in the variation. Examples of packets that will be dropped include: packets that have an incorrect DestinationID, are of a type not supported by the selected Logical layers, or have a transaction ID outside the range used by the selected Logical layers.
multicast_event_rx output	Output	The multicast_event_rx output port is toggled for a single clock cycle when a Multicast Event control symbol is detected on the Physical layer. Being toggled rather pulsed allows the logic using this signal to reside in a different clock domain than the logic that generates it (rxclk).

Software Interface

The RapidIO MegaCore function provides several sets of registers. The registers can be programmed to control the different functions of the RapidIO MegaCore function. The following sets of registers are supported.

- Standard RapidIO Capability Registers - CARs
- Standard RapidIO Command and Status Registers - CSRs
- Extended Features Registers
- Implementation Defined Registers
- Doorbell Specific Registers

All of the registers are 32-bits wide and are shown as hexadecimal values. The registers can only be accessed on a 32-bit (4 byte basis). The addressing for the registers therefore increment by units of 4.

The following sets of registers are accessible through the System Maintenance MM Slave Interface.

- CARs - Capability Registers
- CSRs - Command and Status Registers
- Extended Features Registers
- Implementation Defined Registers

A remote device can only access these registers by issuing read/write Maintenance Operations destined to the local device. Furthermore, the local device must route these transactions, if they are addressing these registers, from the Maintenance Master interface to the System Maintenance Slave interface. Routing can be done by an SOPC Builder system or by a user provided design. Refer to "Concentrator Register Module" on page 4–8 for more details.

The Doorbell specific registers can be accessed only with the Doorbell Avalon-MM slave interface. These registers are not implemented if you do not select the TX Doorbell function nor the RX Doorbell function in the MegaWizard interface. If you use only the RX Doorbell function, then only the RX related doorbell registers are implemented. If the you use only the TX Doorbell function, then only the TX related doorbell registers are implemented.

The following sections describe the address map and the function of each register set. Table 4–33 lists the access codes used to describe the type of register bits.

Table 4–33. Register Access Codes

Code	Description
RW	Read/write
RO	Read-only
RW1C	Read/write 1 to clear
RW0S	Read/write 0 to set
RTC	Read to clear
RTS	Read to set
RTCW	Read to clear/write
RTSW	Read to set/write
RWTC	Read/write any value to clear
RWTS	Read/write any value to set
RWSC	Read/write self-clearing
RWSS	Read/write self-setting
UR0	Unused bits/read as 0
UR1	Unused bits/read as 1

CARs, CSRs, Extended Features, and Implementation-Defined Registers

Table 4–33 shows a summary of the memory map for the CARs, CSRs, Extended Features and Implementation-Defined Registers. A more detailed description of these registers can be found in the follow-on tables, Table 4–35 throughTable 4–92.

This address space is accessible to the user through the System Maintenance MM Slave Interface.

Table 4–34. Memory Map (Part 1 of 3)

Address	Name	Used by
Capability Registers (CARs)		
'h0	Device Identity	The CARs are not used by any of the internal modules. They do not affect the functionality of the RapidIO MegaCore function. These registers are all Read Only. The values are programmed; many are set using the MegaWizard interface when generating the core. These are used to inform either a local processor or a processor on a remote end what this core's capabilities are.
'h4	Device Information	
'h8	Assembly Identity	
'hC	Assembly Information	
'h10	Processing Element Features	
'h14	Switch Port Information	
'h18	Source Operations	
'h1C	Destination Operation	
Command and Status Registers (CSRs)		
'h4C	Processing Element Logical layer Control	Input/Output Slave Logical layer
'h58	Local Configuration Space Base Address 0	I/O Master Logical layer
'h5C	Local Configuration Space Base Address 1	I/O Master Logical layer
'h60	Base Device ID	Transport layer for routing or filtering. Input/Output Slave Logical layer
'h68	Host Base Device ID Lock	Accessed via the Maintenance module
'h6C	Component Tag	Accessed via the Maintenance module
Extended Features Space		
'h100	Register Block Header	Physical layer
'h104- 'h11C	Reserved	
'h120	Port Link Time-out Control	Physical layer
'h124	Port Response Time-out Control	Logical layer modules
'h13C	Port General Control	Physical layer

Table 4–34. Memory Map (Part 2 of 3)

Address	Name	Used by
'h158	Port 0 Error and Status	Physical layer
'h15C	Port 0 Control	Physical layer
Implementation-Defined Space		
'h10000	Reserved	
'h10004		
'h10008		
'h1000C-'h1001C		
'h10020		
'h10024		
'h10028		
'h1002C-'h1007C		
'h10080	Maintenance Interrupt	Maintenance module
'h10084	Maintenance Interrupt Enable	Maintenance module
'h10088	Rx Maintenance Mapping	Maintenance module
'h1008C-'h100FC	Reserved	
'h10100	Tx Maintenance Window 0 Base	Maintenance module
'h10104	Tx Maintenance Window 0 Mask	Maintenance module
'h10108	Tx Maintenance Window 0 Offset	Maintenance module
'h1010C	Tx Maintenance Window 0 Control	Maintenance module
'h10110-'h101FC	Tx Maintenance Windows 1 to 15	Maintenance module
'h10200	Tx Port Write Control	Maintenance module
'h10204	Tx Port Write Status	Maintenance module
'h10210-1024C	Tx Port Write Buffer	Maintenance module
'h10250	Rx Port Write Control	Maintenance module
'h1010254	Rx Port Write Status	Maintenance module
'h10260-10290	Rx Port Write Buffer	Maintenance module
'h102A0-'h102FC	Reserved	
'h10300	I/O Master Window 0 Base	I/O Master Logical

Table 4–34. Memory Map (Part 3 of 3)

Address	Name	Used by
'h10304	I/O Master Window 0 Mask	I/O Master Logical
'h10308	I/O Master Window 0 Offset	I/O Master Logical
'h1030C	Reserved	
'h10310- 'h103FC	I/O Master Windows 1 to 15	I/O Master Logical
'h10400	I/O Slave Window 0 Base	I/O Slave Logical
'h10404	I/O Slave Window 0 Mask	I/O Slave Logical
'h10408	I/O Slave Window 0 Offset	I/O Slave Logical
'h1040C	I/O Slave Window 0 Control	II/O Slave Logical
'h10410- 'h104FC	I/O Slave Windows 1 to 15	I/O Slave Logical
'h10500	I/O Slave Interrupt	I/O Slave Logical
'h10504	I/O Slave Interrupt Enable	I/O Slave Logical
'h10508- 'h107FC	Reserved	
'h10800	Logical/Transport Layer Error Detect	Logical/Transport layer
'h10804	Logical/Transport Layer Error Enable	Logical/Transport layer
'h10808	Logical/Transport Layer Address	Logical/Transport layer
'h1080C	Logical/Transport Layer Device ID Capture	Logical/Transport layer
'h10810	Logical/Transport Layer Control Capture	Logical/Transport layer

Capability Registers (CARs)

Tables 4–35 through 4–42 describe the capability registers.

Table 4–35. Device Identity CAR—Offset: 'h00

Field	Bit	Access	Function	Default
DEVICE_ID	31:16	RO	Hard-wired device identifier	*(1)*
VENDOR_ID	15:0	RO	Hard-wired device vendor identifier	*(1)*

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–36. Device Information CAR—Offset: 'h04

Field	Bit	Access	Function	Default
DEVICE_REV	31:0	RO	Hard-wired device revision level	*(1)*

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–37. Assembly Identity CAR—Offset: 'h08

Field	Bit	Access	Function	Default
ASSY_ID	31:16	RO	Hard-wired assembly identifier	*(1)*
ASSY_VENDOR_ID	15:0	RO	Hard-wired assembly vendor identifier	*(1)*

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–38. Assembly Information CAR—Offset: 'h0C

Field	Bit	Access	Function	Default
ASSY_REV	31:16	RO	Hard-wired assembly revision level	*(1)*
EXT_FEATURE_PTR	15:0	RO	Hard-wired pointer to the first entry in the extended feature list. This pointer must be in the range of 16'H100 and 16'HFFFC.	*(1)*

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–39. Processing Element Features CAR—Offset: 'h10

Field	Bit	Access	Function	Default
BRIDGE	31	RO	Processing element can bridge to another interface.	*(1)*
MEMORY	30	RO	Processing element has physically addressable local address space and can be accessed as an end point through non-maintenance operations. This local address space may be limited to local configuration registers, or could be on-chip SRAM, etc.	*(1)*
PROCESSOR	29	RO	Processing element physically contains a local processor or similar device that executes code. A device that bridges to an interface that connects to a processor does not count.	1'b0
SWITCH	28	RO	Processing element can bridge to another external RapidIO interface – an internal port to a local end point does not count as a switch port.	1'b0
RSRV	27:7	RO	Reserved	21'h0
RE_TRAN_SUP	6	RO	Processing element supports suppression of error recovery on packet CRC errors 1'b0 - The error recovery suppression option is not supported 1'b1 - The error recovery suppression option is supported	1'b0
CRF_SUPPORT	5	RO	Processing element supports the Critical Request Flow (CRF) indicator 1'b0 - Critical Request Flow is not supported 1'b1 - Critical Request Flow is supported	1'b0
LARGE_TRANSPORT	4	RO	'b0 - Processing element does not support common transport large systems. 'b1 - Processing element supports common transport large systems.	1'b0
EXT_FEATURES	3	RO	Processing element has extended features list; the extended features pointer is valid	1'b1
EXT_ADDR_SPRT	2:0	RO	Indicates the number of address bits supported by the Processing element both as a source and target of an operation. All processing elements shall at minimum support 34-bit addresses. 'b111 – Processing element supports 66, 50, and 34-bit addresses 'b101 – Processing element supports 66 and 34-bit addresses 'b011 – Processing element supports 50 and 34-bit addresses 'b001 – Processing element supports 34-bit addresses	3'b001

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–40. Switch Port Information CAR—Offset: 'h14

Field	Bit	Access	Function	Default
RSRV	31:16	RO	Reserved	16'h0
PORT_TOTAL	15:8	RO	The total number of RapidIO ports on the processing element. 'h0 - Reserved 'h1 - 1 port 'h2 - 2 ports --- 'hff - 255 ports	*(1)*
PORT_NUMBER	7:0	RO	This is the port number from which the maintenance read operation accessed this register. Port are numbered starting with 'h0	*(1)*

Note:
(1) The default value is set in the MegaWizard interface.

Table 4–41. Source Operations CAR—Offset: 'h18 (Part 1 of 2)(1)

Field	Bit	Access	Function	Default
RSRV	31:16	RO	Reserved	16'h0
READ	15	RO	Processing element can support a read operation	1'b0
WRITE	14	RO	Processing element can support a write operation	*(2)*
SWRITE	13	RO	Processing element can support a streaming-write operation	*(2)*
NWRITE_R	12	RO	Processing element can support a write-with-response operation	*(2)*
Data Message	11	RO	Processing element can support data message operation	*(3)*
Doorbell	10	RO	Processing element can support a doorbell operation	*(4)*
ATM_COMP_SWP	9	RO	Processing element can support an atomic compare-and-swap operation	1'b0
ATM_TEST_SWP	8	RO	Processing element can support an atomic test-and-swap operation	1'b0
ATM_INC	7	RO	Processing element can support an atomic increment operation	1'b0
ATM_DEC	6	RO	Processing element can support an atomic decrement operation	1'b0
ATM_SET	5	RO	Processing element can support an atomic set operation	1'b0
ATM_CLEAR	4	RO	Processing element can support an atomic clear operation	1'b0

Table 4–41. Source Operations CAR—Offset: 'h18 (Part 2 of 2)(1)

Field	Bit	Access	Function	Default
ATM_SWAP	3	RO	Processing element can support an atomic swap operation	1'b0
PORT_WRITE	2	RO	Processing element can support a port-write operation	(5)
RSRV	1:0	RO	Reserved	2'b00

Note:

(1) If one of the Logical layers supported by the RapidIO MegaCore is not selected, the corresponding bits in the Source and Destination Operations CAR are forced to zero. These bits cannot be set to one, even if the corresponding operations are supported by user logic attached to the Pass-Through port.

(2) The value of the processing element is 1'b1 if the I/O Slave was selected in the MegaWizard interface. The value is 1'b0 if the I/O Slave was not selected in the MegaWizard interface.

(3) The default value is set in the MegaWizard interface.

(4) The value of the processing element is 1'b1 if **Doorbell RX enable** is turned on in the MegaWizard interface. If **Doorbell RX enable** is turned off, the value is 1'b0.

(5) The value of the processing element is 1'b1 if the Maintenance slave is selected in MegaWizard interface. If the Maintenance slave is not selected, the value is 1'b0.

Table 4–42. Destination Operations CAR—Offset: 'h1C (Part 1 of 2)(1)

Field	Bit	Access	Comment	Default
RSRV	31:16	RO	Reserved	16'h0
READ	15	RO	Processing element can support a read operation	(2)
WRITE	14	RO	Processing element can support a write operation	(2)
SWRITE	13	RO	Processing element can support a streaming-write operation	(2)
NWRITE_R	12	RO	Processing element can support a write-with-response operation	(2)
Data Message	11	RO	Processing element can support data message operation.	(3)
Doorbell	10	RO	Processing element can support a doorbell operation. The value of the processing element is 1'b1 if **Doorbell TX enable** is turned on in the MegaWizard interface. The value of the processing element is1'b if **Doorbell TX enable** is turned off in the MegaWizard interface.	(4)
ATM_COMP_SWP	9	RO	Processing element can support an atomic compare-and-swap operation	1'b0
ATM_TEST_SWP	8	RO	Processing element can support an atomic test-and-swap operation	1'b0
ATM_INC	7	RO	Processing element can support an atomic increment operation	1'b0
ATM_DEC	6	RO	Processing element can support an atomic decrement operation	1'b0

Table 4–42. Destination Operations CAR—Offset: 'h1C (Part 2 of 2)(1)

Field	Bit	Access	Comment	Default
ATM_SET	5	RO	Processing element can support an atomic set operation	1'b0
ATM_CLEAR	4	RO	Processing element can support an atomic clear operation	1'b0
ATM_SWAP	3	RO	Processing element can support an atomic swap operation	1'b0
PORT_WRITE	2	RO	Processing element can support a port-write operation	*(2)*
RSRV	1:0	RO	Reserved	2'b00

Note:
(1) If one of the Logical layers supported by the RapidIO MegaCore is not selected, the corresponding bits in the Source and Destination Operations CAR are forced to zero. These bits cannot be set to one, even if the corresponding operations are supported by user logic attached to the Pass-Through port.
(2) The value of the processing element is 1'b1 if the Maintenance master is selected in MegaWizard interface. If the maintenance master is not selected, the value is 1'b0.
(3) ***The default value is set in the MegaWizard interface.***
(4) Value varies based on MegaWizard interface option selection.

Command and Status Registers (CSRs)

Tables 4–43 through 4–48 describe the command and status registers.

Table 4–43. Processing Element Logical Layer Control CSR—Offset: 'h4C

Field	Bit	Access	Function	Default
RSRV	31:3	RO	Reserved	29'h0
EXT_ADDR_CTRL	2:0	RO	Controls the number of address bits generated by the Processing element as a source and processed by the Processing element as the target of an operation. 'b100 – Processing element supports 66 bit addresses 'b010 – Processing element supports 50 bit addresses 'b001 – Processing element supports 34 bit addresses All other encodings reserved	3'b001

Table 4–44. Local Configuration Space Base Address 0 CSR—Offset: 'h58

Field	Bit	Access	Function	Default
RSRV	31	RO	Reserved	1'b0
LCSBA	30:15	RO	Reserved for a 34-bit local physical address	16'h0
LCSBA	14:0	RO	Reserved for a 34-bit local physical address	15'h0

Table 4–45. Local Configuration Space Base Address 1 CSR—Offset: 'h5C (1)

Field	Bit	Access	Function	Default
LCSBA	31	RO	Reserved for a 34-bit local physical address	1'b0
LCSBA	30:0	RW	Bits 34:4 of a 34-bit physical address	31'h0

Notes for Table 4–45:

(1) The Local Configuration Space Base Address registers are hard coded to zero. If the Input/Output Avalon-MM master interface is connected to the System Maintenance Avalon-MM slave interface, regular read and write operations rather than maintenance operations, can be used to access the processing element's registers for configuration and maintenance.

Table 4–46. Base Device ID CSR—Offset: 'h60

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved	8'h0
DEVICE_ID	23:16	RW	This is the base ID of the device in a small common transport system.	8'hFF
LARGE_DEVICE_ID	15:0	RO	This is the base ID of the device in a large common transport system.	16'hFFFF

Table 4–47. Host Base Device ID Lock CSR—Offset: 'h68

Field	Bit	Access	Function	Default
RSRV	31:16	RO	Reserved	16'h0
HOST_BASE_DEVICE_ID	15:0	RW Write once; can be reset	This is the base device ID for the Processing element that is initializing this processing element.	16'hFFFF

Table 4–48. Component Tag CSR—Offset: 'h6C

Field	Bit	Access	Function	Default
COMPONENT_TAG	31:0	RW	This is a component tag for the processing element.	32'h0

Extended Feature Registers

Tables 4–49 through 6–50 describe the 1x/4x LP-Serial extended feature register block.

Table 4–49. 1x/4x LP-Serial Register Block Header—Offset: 'h100

Field	Bit	Access	Function	Default
EF_PTR	31:16	RO	Hard wired pointer to the next block in the data structure, if one exists	16'h0
EF_ID	15:0	RO	Hard wired Extended Features ID	16'h001

Table 4–50. Port Link Time-out Control CSR—Offset: 'h120

Field	Bit	Access	Function	Default
VALUE	31:8	RW	Time-out interval value	'hff_ffff
RSRV	7:0	RO	Reserved	8'h0

Table 4–51. Port Response Time-out Control—Offset: 'h124

Field	Bit	Access	Function	Default
VALUE	31:8	RW	Time-out interval value	'hff_ffff
RSRV	7:0	RO	Reserved	8'h0

Table 4–52. Port General Control—Offset: 'h13C (Part 1 of 2)

Field	Bit	Access	Function	Default
HOST	31	RW	A Host device is a device that is responsible for system exploration, initialization, and maintenance. Agent or slave devices are typically initialized by Host devices. 'b0 - agent or slave device 'b1 - host device	1'b0
ENA	30	RW	The Master Enable bit controls whether or not a device is allowed to issue requests into the system. If the Master Enable is not set, the device may only respond to requests. 'b0 - processing element cannot issue requests 'b1 - processing element can issue requests Variations that use only the Physical layer ignore this bit.	1'b0

Table 4–52. Port General Control—Offset: 'h13C (Part 2 of 2)

Field	Bit	Access	Function	Default
DISCOVER	29	RW	This device has been located by the processing element responsible for system configuration 'b0 - The device has not been previously discovered 'b1 - The device has been discovered by another processing element	1'b0
RSRV	28:0	RO	Reserved	29'h0

Table 4–53. Port0 Error and Status CSRs—Offset: 'h158 (1)

Field	Bit	Access	Function	Default
RSRV	31:21	RO	Reserved	11'h0
OUT_RTY_ENC	20	RW1C	Output port has encountered a retry condition	1'b0
OUT_RETRIED	19	RO	Output port has received a packet-retry control symbol and cannot make forward progress.	1'b0
OUT_RTY_STOP	18	RO	Output port has been stopped due to a retry and is trying to recover.	1'b0
OUT_ERR_ENC	17	RW1C	Output port has encountered (and possibly recovered from) a transmission error.	1'b0
OUT_ERR_STOP	16	RO	Output port has been stopped due to a transmission error and is trying to recover.	1'b0
RSRV	15:11	RO	Reserved	5'h0
IN_RTY_STOP	10	RO	Input port has been stopped due to a retry.	1'b0
IN_ERR_ENC	9	RW1C	Input port has encountered (and possibly recovered from) a transmission error.	1'b0
IN_ERR_STOP	8	RO	Input port has been stopped due to a transmission error.	1'b0
RSRV	7:5	RO	Reserved	3'h0
PWRITE_PEND	4	RO	This register is not implemented and is reserved. It is always set to zero.	1'b0
RSRV	3	RO	Reserved	1'b0
PORT_ERR	2	RW1C	Input or output port has encountered an unrecoverable error and has shut down (turned off both port enables).	1'b0
PORT_OK	1	RO	Input and output ports a initialized and can communicate with the adjacent device.	1'b0
PORT_UNINIT	0	RO	Input and output ports are not initialized and is in training mode.	1'b1

Notes for Table 4–53:
(1) Refer to "Error Detection and Management" on page 4–54 for details.

Table 4–54. Serial Port Control CSR—Offset: 'h15C (Part 1 of 2)

Field	Bit	Access	Function	Default
PORT_WIDTH	31:30	R0	Hardware width of the port: 'b00—Single-lane port. 'b01—Four-lane port. 'b10–'b11—Reserved.	2'b00 (for 1x variations), 2'b01 (for 4x variations)*(2)*
INIT_WIDTH	29:27	R0	Width of the ports after initialized: 'b000—Single lane port, lane 0. 'b001—Single lane port, lane 2. 'b010—Four lane port. 'b011–'b111—Reserved.	3'b000 (for 1x variations), 3'b010 (for 4x variations)
PWIDTH_OVRIDE	26:24	UR0	Soft port configuration to override the hardware size: 'b000—No override. 'b001—Reserved. 'b010—Force single lane, lane 0. 'b011—Force single lane, lane 2. 'b100–'b111—Reserved.	3'h0
PORT_DIS	23	RW	Port disable: 'b0—port receivers/drivers are enabled. 'b1—port receivers are disabled, causing the drivers to send out idles. ● When this bit transitions from one to zero, the initialization state machines' force_reinit state variable is asserted, as described in *Part 6: Physical Layer 1×/4× LP Serial Physical Layer Specification Revision 1.3*, paragraphs 4.7.3.5 and 4.7.3.6. In turn, this assertion causes the port to enter the SILENT state and to attempt to reinitialize the link. ● When reception is disabled, the input buffers are kept empty until this bit is cleared. ● When PORT_DIS is asserted and the drivers are disabled, the transmit buffer are reset and kept empty until this bit is cleared, any previously stored packets are lost and any attempt to write a packet to the atx Atlantic interface is ignored by the Physical layer, new packets are NOT stored for later transmission.	1'b0
OUT_PENA	22	RW	Output port transmit enable: 'b0—port is stopped and not enabled to issue any packets except to route or respond to I/O logical maintenance packets, depending upon the functionality of the processing element. Control symbols are not affected and are sent normally. 'b1—port is enabled to issue any packets.	1'b1

Table 4–54. Serial Port Control CSR—Offset: 'h15C (Part 2 of 2)

Field	Bit	Access	Function	Default
IN_PENA	21	RW	Input port receive enable: 'b0 - port is stopped and only enabled to respond I/O logical MAINTENANCE requests. Other requests return packet-not-accepted control symbols to force an error condition to be signaled by the sending device 'b1 - port is enabled to respond to any packet	1'b1
ERR_CHK_DIS	20	RW	This bit disables all RapidIO transmission error checking 'b0 - Error checking and recovery is enabled 'b1 - Error checking and recovery is disabled Device behavior when error checking and recovery is disabled and an error condition occurs is undefined.	1'b0
MULTICAST	19	RW	Send incoming Multicast-event control symbols to this port (multiple port devices only). This bit has no effect on the MegaCore function as multicast event generation is not supported.	1'b0
RSRV	18:1	RO	Reserved	18'h0
PORT_TYPE	0	RO	This indicates the port type, parallel or serial. 'b0 - Parallel port *(1)* 'b1 - Serial port	1'b1

Note Table 4–54:
(1) For parallel mode support using the RapidIO MegaCore function contact your Altera field applications representative or make a support request using www.altera.com/mysupport/
(2) Reflects the choice made in MegaWizard interface.

General Maintenance Interrupt Control Registers

Tables 4–55 through 4–56 describe the maintenance interrupt control registers. If any of these error conditions are detected and if the corresponding Interrupt Enable bit is set, the sys_mnt_s_irq signal is asserted.

Table 4–55. Maintenance Interrupt—Offset: 'h10080 (Part 1 of 2)

Field	Bit	Access	Function	Default
RSRV	31:7	RO	Reserved	25'h0
PORT_WRITE_ERROR	6	RW1C	Port-write error	1'b0

Table 4–55. Maintenance Interrupt—Offset: 'h10080 (Part 2 of 2)

Field	Bit	Access	Function	Default
PACKET_DROPPED	5	RW1C	Received port-write packet dropped. A port-write packet is dropped under the following conditions: • a port-write request packet is received but por-write reception has not been enabled by setting bit PORT_WRITE_ENABLE in the Rx Port Write Control register. • a previously received port-write hasn't been read out from the“Rx Port Write register	1'b0
PACKET_STORED	4	RW1C	Indicates that the core has received a Port Write packet and that the payload can be retrieved using the System Maintenance MM Slave interface.	1'b0
RSRV	3	RO	Reserved	1'b0
RSRV	2	RO	Reserved	1'b0
WRITE_OUT_OF_BOUNDS	1	RW1C	If the address of an Avalon-MM write transfer presented at the Maintenance Avalon-MM Slave interface does not fall within any of the TX Maintenance Address translation windows, then it is considered out of bounds and this bit is set.	1'b0
READ_OUT_OF_BOUNDS	0	RW1C	If the address of an Avalon-MM read transfer presented at the Maintenance Avalon-MM Slave interface does not fall within any of the TX Maintenance Address translation windows, then it is considered out of bounds and this bit is set.	1'b0

Table 4–56. Maintenance Interrupt Enable—Offset: 'h10084

Field	Bit	Access	Function	Default
RSRV	31:7	RO	Reserved	25'h0
PORT_WRITE_ERROR	6	RW	Port-write error interrupt enable	1'b0
RX_PACKET_DROPPED	5	RW	Rx port-write packet dropped interrupt enable	1'b0
RX_PACKET_STORED	4	RW	Rx port-write packet stored in buffer interrupt enable.	1'b0
RSRV	3:2	RO	Reserved	2'b00
WRITE_OUT_OF_BOUNDS	1	RW	Tx write request address out of bounds interrupt enable.	1'b0
READ_OUT_OF_BOUNDS	0	RW	Tx read request address out of bounds interrupt enable.	1'b0

Receive Maintenance Registers

Table 4–57 describes the receiver maintenance register.

Table 4–57. Rx Maintenance Mapping—Offset: 'h10088

Field	Bit	Access	Function	Default
RX_BASE	31:24	RW	Rx base address. The offset value carried in a received Maintenance Type packet is concatenated with this RX_BASE to form a 32-bit Avalon Address as follows. Avalon_address = {rx_base, cfg_offset, word_addr, 2'b00}	8'h0
RSRV	23:0	RO	Reserved	24'h0

Transmit Maintenance Registers

Tables 4–58 through 4–61 describe the transmitter maintenance registers. When transmitting a Maintenance Packet, an address translation process occurs, involving a Base, Mask, Offset, and Control register. There are up to sixteen groups of four registers. The 16 register address offsets are shown in the table titles. For more details on how to use these windows, see "Maintenance Slave Processor" on page 4–14

Table 4–58. Tx Maintenance Mapping Window n Base—Offset: 'h10100, 'h10110, 'h10120, 'h10130, 'h10140, 'h10150, 'h10160, 'h10170, 'h10180, 'h10190, 'h101A0, 'h101B0, 'h101C0, 'h101D0, 'h101E0, 'h101F0

Field	Bit	Access	Function	Default
BASE	31:3	RW	Start of the Avalon-MM address window to be mapped. The three least significant bits of the 32-bit base are assumed to be zero.	29'h0
RSRV	2:0	RO	Reserved	3'h0

Table 4–59. Tx Maintenance Mapping Window n Mask—Offset: 'h10104, 'h10114, 'h10124, 'h10134, 'h10144, 'h10154, 'h10164, 'h10174, 'h10184, 'h10194, 'h101A4, 'h101B4, 'h101C4, 'h101D4, 'h101E4, 'h101F4

Field	Bit	Access	Function	Default
MASK	31:3	RW	Mask for the address mapping window. The three least significant bits of the 32-bit mask are assumed to be zero.	29'h0
WEN	2	RW	Window enable. Set to one to enable the corresponding window.	1'b0
RSRV	1:0	RO	Reserved	2'h0

Table 4–60. Tx Maintenance Mapping Window n Offset—Offset: 'h10108, 'h10118, 'h10128, 'h10138, 'h10148, 'h10158, 'h10168, 'h10178, 'h10188, 'h10198, 'h101A8, 'h101B8, 'h101C8, 'h101D8, 'h101E8, 'h101F8

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved	8'h0
OFFSET	23:0	RW	Window offset	24'h0

Table 4–61. Tx Maintenance Mapping Window n Control—Offset: 'h1010C, 'h1011C, 'h1012C, 'h1013C, 'h1014C, 'h1015C, 'h1016C, 'h1017C, 'h1018C, 'h1019C, 'h101AC, 'h101BC, 'h101CC, 'h101DC, 'h101EC, 'h101FC

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved	8'h0
DESTINATION_ID	23:16	RW	Destination ID	8'h0
HOP_COUNT	15:8	RW	Hop count	8'hFF
PRIORITY	7:6	RW	Packet priority. 2'b11 is not a valid value for the PRIORITY field. An attempt to write 2'b11 to this field will be overwritten as 2'b10.	2'b00
RSRV	5:0	RO	Reserved	6'h0

Transmit Port-Write Registers

Tables 4–62 through 4–64 describes the transmit port-write registers

See "Port Write Processor" on page 4–20 for a description on how to use these registers to transmit a port write.

Table 4–62. Tx Port Write Control—Offset: 'h10200

Field	Bit	Access	Function	Default
RSRV	30:24	RO	Reserved	8'h0
DESTINATION_ID	23:16	RW	Destination ID	8'h0
RSRV	15:8	RO	Reserved	8'hFF
PRIORITY	7:6	RW	Request Packet's priority. 2'b11 is not a valid value for the PRIORITY field. An attempt to write 2'b11 to this field will be overwritten as 2'b10.	2'b00
SIZE	5:2	RW	Packet payload size in number of double words. If set to 0, the payload size is single word. If SIZE is set to a value larger than 8, the payload size is 8 double words (64 bytes).	4'h0
RSRV	1	RO	Reserved	1'b0
PACKET_READY	0	RW	Write 1 to start transmitting the port-write request. This bit is cleared internally once the packet has been transferred to the transport layer to be forwarded to the physical layer for transmission.	1'b0

Table 4–63. Tx Port Write Status—Offset: 'h10204

Field	Bit	Access	Function	Default
RSRV	31:0	RO	Reserved	31'h0

Table 4–64. Tx Port Write Buffer n—Offset: 'h10210 – 'h1024C

Field	Bit	Access	Function	Default
PORT_WRITE_DATA_n	31:0	RW	Port-write data. This buffer is implemented in memory and is not initialized at reset.	32'hx

Receive Port-Write Registers

Tables 4–65 through 4–67 describes the receive port-write registers.

See "Port Write Processor" on page 4–20 for a description on how to receive port write maintenance packets.

Table 4–65. Rx PPort Write Control—Offset: 'h10250

Field	Bit	Access	Function	Default
RSRV	31:2	RO	Reserved	30'h0
CLEAR_BUFFER	1	RW	Clear port-write buffer. Write 1 to activate. Always read 0.	1'b0
PORT_WRITE_ENA	0	RW	Port-write enable. If set to 1, port-write packets are accepted. If set to 0, port-write packet are dropped.	1'b1

Table 4–66. Rx Port Write Status—Offset: 'h10254

Field	Bit	Access	Function	Default
RSRV	31:6	RO	Reserved	26'h0
PAYLOAD_SIZE	5:2	RO	Packet payload size in number of double words. If the size is zero, the payload size is single word.	4'h0
RSRV	1	RO	Reserved	1'b0
PORT_WRITE_BUSY	0	RO	Port-write busy. Set if a packet is currently being stored in the buffer.	1'b0

Table 4–67. Rx Port Write Buffer n—Offset: 'h10260 – 'h1029C				
Field	Bit	Access	Function	Default
PORT_WRITE_DATA_n	31:0	RO	Port-write data. This buffer is implemented in memory and is not initialized at reset.	32'hx

I/O Master Address Mapping Registers

Tables 4–68 through 4–70 describe the I/O master registers. When the MegaCore function receives an NREAD, NWRITE, NWRITE_R, or SWRITE request packet, the RapidIO address has to be translated into a local Avalon-MM Address. The translation involves the Base, Mask, and Offset registers. There are up to sixteen register sets, one for each address mapping window. The 16 possible register address offsets are shown in the table titles.

See "Input/Output Avalon-MM Master Address Mapping Windows" on page 4–25 for more details.

Table 4–68. I/O Master Mapping Window n Base—Offset: 'h10300, 'h10310, 'h10320, 'h10330, 'h10340, 'h10350, 'h10360, 'h10370, 'h10380, 'h10390, 'h103A0, 'h103B0, 'h103C0, 'h103D0, 'h103E0, 'h103F0				
Field	Bit	Access	Function	Default
BASE	31:3	RW	Start of the RapidIO address window to be mapped. The three least significant bits of the 34-bit base are assumed to be zeros.	29'h0
RSRV	2	RO	Reserved	1'b0
XAMB	1:0	RW	Extended Address: two most significant bits of the 34-bit base.	2'h0

Table 4–69. I/O Master Mapping Window n Mask—Offset: 'h10304, 'h10314, 'h10324, 'h10334, 'h10344, 'h10354, 'h10364, 'h10374, 'h10384, 'h10394, 'h103A4, 'h103B4, 'h103C4, 'h103D4, 'h103E4, 'h103F4				
Field	Bit	Access	Function	Default
MASK	31:3	RW	Bits 31 to 3 of the mask for the address mapping window. The three least significant bits of the 34-bit mask are assumed to be zeros.	29'h0
WEN	2	RW	Window enable. Set to one to enable the corresponding window.	1'b0
XAMM	1:0	RW	Extended Address: two most significant bits of 34-bit mask.	3'h0

Table 4–70. I/O Master Mapping Window n Offset—Offset: 'h10308, 'h10318, 'h10328, 'h10338, 'h10348, 'h10358, 'h10368, 'h10378, 'h10388, 'h10398, 'h103A8, 'h103B8, 'h103C8, 'h103D8, 'h103E8, 'h103F8

Field	Bit	Access	Function	Default
OFFSET	31:3	RW	Starting offset into the Avalon-MM address space. The three least significant bits of the 32-bit offset are assumed to be zero.	29'h0
RSRV	2:0	RO	Reserved	3'h0

Input/Output Slave Mapping Registers

Tables 4–71 through 4–76 describe the Input/Output Slave registers. The registers are used to define windows in the Avalon-MM address space that are used to determine the outgoing request packet's ftype, DestinationID, priority, and address fields. There are up to sixteen register sets, one for each possible address mapping window. The 16 possible register address offsets are shown in the table titles.

Refer to "Input/Output Avalon-MM Slave Address Mapping Windows" on page 4–30 for a description on how to use these registers.

Table 4–71. Input/Output Slave Mapping Window n Base—Offset: 'h10400, 'h10410, 'h10420, 'h10430, 'h10440, 'h10450, 'h10460, 'h10470, 'h10480, 'h10490, 'h104A0, 'h104B0, 'h104C0, 'h104D0, 'h104E0, 'h104F0

Field	Bit	Access	Function	Default
BASE	31:3	RW	Start of the Avalon-MM address window to be mapped. The three least significant bits of the 32-bit base are assumed to be all zeros.	29'h0
RSRV	2:0	RO	Reserved	1'b0

Table 4–72. Input/Output Slave Mapping Window n Mask—Offset: 'h10404, 'h10414, 'h10424, 'h10434, 'h10444, 'h10454, 'h10464, 'h10474, 'h10484, 'h10494, 'h104A4, 'h104B4, 'h104C4, 'h104D4, 'h104E4, 'h104F4

Field	Bit	Access	Function	Default
MASK	31:3	RW	29 most significant bits of the mask for the address mapping window. The three least significant bits of the 32-bit mask are assumed to be zeros.	29'h0

Table 4–72. Input/Output Slave Mapping Window n Mask—Offset: 'h10404, 'h10414, 'h10424, 'h10434, 'h10444, 'h10454, 'h10464, 'h10474, 'h10484, 'h10494, 'h104A4, 'h104B4, 'h104C4, 'h104D4, 'h104E4, 'h104F4

Field	Bit	Access	Function	Default
WEN	2	RW	Window enable. Set to one to enable the corresponding window.	1'b0
RSRV	1:0	RO	Reserved	2'h0

Table 4–73. Input/Output Slave Mapping Window n Offset—Offset: 'h10408, 'h10418, 'h10428, 'h10438, 'h10448, 'h10458, 'h10468, 'h10478, 'h10488, 'h10498, 'h104A8, 'h104B8, 'h104C8, 'h104D8, 'h104E8, 'h104F8

Field	Bit	Access	Function	Default
OFFSET	31:3	RW	Bits [31:3] of the starting offset into the RapidIO address space. The three least significant bits of the 34-bit offset are assumed to be zeros.	29'h0
RSRV	2	RO	Reserved	1'b0
XAMO	1:0	RW	Extended Address: two most significant bits of the 34-bit offset.	2'h0

Table 4–74. Input/Output Slave Mapping Window n Control—Offset: 'h1040C, 'h1041C, 'h1042C, 'h1043C, 'h1044C, 'h1045C, 'h1046C, 'h1047C, 'h1048C, 'h1049C, 'h104AC, 'h104BC, 'h104CC, 'h104DC, 'h104EC, 'h104FC

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved. 8 MSB for 16-bit DestinationID	8'h0
DESTINATION_ID	23:16	RW	8 least significant bits of DestinationID	8'h0
RSRV	15:8	RO	Reserved	8'h0
PRIORITY	7:6	RW	Request Packet's priority 2'b11 is not a valid value for the PRIORITY field. An attempt to write 2'b11 to this field will be overwritten as 2'b10.	2'h0
RSRV	5:2	RO	Reserved	4'h0
SWRITE_ENABLE	1	RW	SWRITE enable. Set to one to generate SWRITE request packets. *(1)*	1'b0
NWRITE_R_ENABLE	0	RW	NWRITE_R enable *(1)*	1'b0

Note to Table 4–74:

(1) Bits 0 and 1 (NWRITE_R_ENABLE and SWRITE_ENABLE) are mutually exclusive. An attempt to write ones to both of these fields at the same time will be ignored, and that part of the register will keep its previous value.

Input/Output Slave Interrupts

Figure 4–75 describes the available Input/Output Slave interrupts. These interrupt bits assert the `sys_mnt_s_irq` signal if the corresponding interrupt bit is enabled.

Table 4–75. Input/Output Slave Interrupt—Offset: 'h10500

Field	Bit	Access	Function	Default
RSRV	31:4	RO	Reserved	25'h0
INVALID_WRITE_BYTEENABLE	3	RW1C	Write byte enable invalid. Asserted when io_s_wr_byteenable is set to invalid values. For information on valid values see Table 4–5 and Table 4–7.	1'b0
INVALID_WRITE_BURSTCOUNT	2	RW1C	Write burst count invalid. Asserted when io_s_wr_burstcount is set to an odd number larger than one in variations with 32-bit wide data path Avalon-MM write interfaces.	1'b0
WRITE_OUT_OF_BOUNDS	1	RW1C	Write request address out of bounds. Asserted when the Avalon-MM address does not fall within any enabled address mapping windows.	1'b0
READ_OUT_OF_BOUNDS	0	RW1C	Read request address out of bounds. Asserted when the Avalon-MM address does not fall within any enabled address mapping windows.	1'b0

Table 4–76. Input/Output Slave Interrupt Enable—Offset: 'h10504

Field	Bit	Access	Function	Default
RSRV	31:6	RO	Reserved	28'h0
INVALID_WRITE_BYTEENABLE	3	RW	Write byte enable invalid interrupt enable	1'b0
INVALID_WRITE_BURSTCOUNT	2	RW	Write burst count invalid interrupt enable	1'b0
WRITE_OUT_OF_BOUNDS	1	RW	Write request address out of bounds interrupt enable	1'b0
READ_OUT_OF_BOUNDS	0	RW	Read request address out of bounds interrupt enable	1'b0

Error Management Registers

Tables 4–77 through 4–81 describe the error management registers. These registers can be used by software to diagnose problems with packets being received by this local end point. If enabled, the detected error will trigger the assertion of `sys_mnt_s_irq`. The packet that caused the error will be captured in the capture registers. After an error condition is detected, the information is captured and the capture registers are "locked" until the Error Detect CSR is cleared. Upon being cleared, the capture registers are ready to capture a new packet that exhibits an error condition.

Table 4–77. Logical/Transport Layer Error Detect CSR—Offset: 'h10800

Field	Bit	Access	Function	Default
IO_ERROR_RSP	31	RW	Received a response of 'ERROR' for an I/O Logical Layer Request.	1'b0
MSG_ERROR_RESPONSE	30	RW	Received a response of 'ERROR' for a MSG Logical Layer Request.	1'b0
RSRV	29	RO	Reserved	1'b0
MSG_FORMAT_ERROR	28	RW	Received MESSAGE packet data payload with an invalid size or segment.	1'b0
ILL_TRAN_DECODE	27	RW	Received illegal fields in the request/response packet for a supported transaction.	1'b0
ILL_TRAN_TARGET	26	RW	Received a packet that contained a destination ID that is not defined for this end point.	1'b0
MSG_REQ_TIMEOUT	25	RW	A required message request has not been received within the specified time-out interval.	1'b0
PKT_RSP_TIMEOUT	24	RW	A required response has not been received within the specified time out interval.	1'b0
UNSOLICIT_RSP	23	RW	An unsolicited/unexpected response packet was received.	1'b0
UNSUPPORT_TRAN	22	RW	A transaction is received that is not supported in the Destination Operations CAR.	1'b0
RSRV	21:0	RO	Reserved	22'h0

Table 4–78. Logical/Transport Layer Error Enable CSR—Offset: 'h10804

Field	Bit	Access	Function	Default
IO_ERROR_RSP_EN	31	RW	Enable reporting of an I/O error response. Save and lock original request transaction information in all Logical/Transport Layer Capture CSRs.	1'b0
MSG_ERROR_RESPONSE_EN	30	RW	Enable reporting of a Message error response. Save and lock original request transaction information in all Logical/Transport Layer Capture CSRs.	1'b0
RSRV	29	RO	Reserved	1'b0
MSG_FORMAT_ERROR_EN	28	RW	Enable reporting of a message format error. Save and lock original request transaction information in all Logical/Transport Layer Capture CSRs.	1'b0
ILL_TRAN_DECODE_EN	27	RW	Enable reporting of an illegal transaction decode error Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs.	1'b0
ILL_TRAN_TARGET_EN	26	RW	Enable reporting of an illegal transaction target error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs.	1'b0
MSG_REQ_TIMEOUT_EN	25	RW	Enable reporting of a Message Request time-out error. Save and lock original request transaction information in Logical/Transport Layer Device ID and Control Capture CSRs for the last Message request segment packet received.	1'b0
PKT_RSP_TIMEOUT_EN	24	RW	Enable reporting of a packet response time-out error. Save and lock original request address in Logical/Transport Layer Address Capture CSRs. Save and lock original request Destination ID in Logical/Transport Layer Device ID Capture CSR.	1'b0
UNSOLICIT_RSP_EN	23	RW	Enable reporting of an unsolicited response error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs.	1'b0
UNSUPPORT_TRAN_EN	22	RW	Enable report of an unsupported transaction error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs.	1'b0
RSRV	21:0	RO	Reserved	22'h0

Table 4–79. Logical/Transport Layer Address Capture CSR—Offset: 'h10808

Field	Bit	Access	Function	Default
ADDRESS	31:3	RO	Bits 31 to 3 of the RapidIO address associated with the error.	29'h0
RSRV	2	RO	Reserved	1'b0
XAMSBS	1:0	RO	Extended address bits of the address associated with the error.	2'h0

Table 4–80. Logical/Transport Layer Device ID Capture CSR—Offset: 'h1080C

Field	Bit	Access	Function	Default
Reserved	31:24	RO	Reserved	8'h0
DESTINATION_ID	23:16	RO	The destination ID associated with the error.	8'h0
RSRV	15:8	RO	Reserved	8'h0
SOURCE_ID	7:0	RO	The source ID associated with the error.	8'h0

Table 4–81. Logical/Transport Layer Control Capture CSR—Offset: 'h10810

Field	Bit	Access	Function	Default
FTYPE	31:28	RO	Format type associated with the error.	4'h0
TTYPE	27:24	RO	Transaction type associated with the error.	4'h0
MSG_INFO	23:16	RO	Letter, mbox, and msgseg for the last message request received for the mailbox that had and error.	8'h0
RSRV	15:0	RO	Reserved	16'h0

Doorbell Message Registers

The RapidIO Megacore function has registers accessible by the Avalon-MM slave port in the Doorbell Module. These registers are described in the following sections

Refer to section Doorbell Module on page 5-33 for a detailed explanation of Doorbell functionality supported by this MegaCore function..

Table 4–82. Doorbell Message Module Memory Map

Address	Name	Used by
Doorbell Message Space		
'h00	Rx Doorbell	External Avalon-MM master wishing to generate or receive doorbell messages.
'h04	Rx Doorbell Status	
'h08	Tx Doorbell Control	
'h0C	Tx Doorbell	
'h10	Tx Doorbell Status	
'h14	Tx Doorbell Completion	
'h18	Tx Doorbell Completion Status	
'h1C	Tx Doorbell Status Control	
'h20	Doorbell Interrupt Enable	
'h24	Doorbell Interrupt Status	

Table 4–83. Rx Doorbell – Offset: 'h00

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved for future 16 bit Device ID support	8'b0
SOURCE_ID	23:16	RO	Device ID of the doorbell message initiator	8'b0
INFORMATION (MSB)	15:8	RO	Received doorbell message information field, MSB	8'b0
INFORMATION (LSB)	7:0	RO	Received doorbell message information field, LSB	8'b0

Table 4–84. Rx Doorbell Status – Offset: 'h04

Field	Bit	Access	Function	Default
RSRV	31:8	RO	Reserved	8'b0
FIFO_LEVEL	7:0	RO	Shows the number of available doorbell messages in the Rx FIFO. A maximum of 16 received messages is supported.	8'b0

Table 4–85. Tx Doorbell Control – Offset: 'h08

Field	Bit	Access	Function	Default
RSRV	31:2	RO	Reserved	30'b0
PRIORITY	1:0	R/W	Request Packet's priority 2'b11 is not a valid value for the PRIORITY field. An attempt to write 2'b11 to this field will be overwritten as 2'b10.	2'b0

Table 4–86. Tx Doorbell – Offset: 'h0C

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved for future support of 16–bit Device IDs	8'b0
DESTINATION_ID	23:16	R/W	Device ID of the targeted RapidIO processing element	8'b0
INFORMATION(MSB)	15:8	R/W	MSB information field of the outbound doorbell message	8'b0
INFORMATION (LSB)	7:0	R/W	LSB information field of the outbound doorbell message	8'b0

Table 4–87. Tx Doorbell Status – Offset: 'h10 (Part 1 of 2)

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved	8'b0
PENDING	23:16	RO	Number of doorbell messages that have been transmitted but for which a response hasn't been received yet. There can be a maximum of 16 pending doorbell messages.	8'b0

Table 4–87. Tx Doorbell Status – Offset: 'h10 (Part 2 of 2)

Field	Bit	Access	Function	Default
TX_FIFO_LEVEL	15:8	RO	The number of doorbell messages in the Tx FIFO waiting for transmission. This fifo can store a maximum of 16.	8'b0
TXCPL_FIFO_LEVEL	7:0	RO	The number of available completed Tx doorbell messages in the Tx Completion FIFO. The fifo can store a maximum of 16.	8'b0

Table 4–88. Tx Doorbell Completion– Offset: 'h14

Field	Bit	Access	Function	Default
RSRV	31:24	RO	Reserved for future support of 16–bit Device IDs	8'b0
DESTINATION_ID	23:16	RO	The Device ID of the targeted RapidIO processing element.	8'b0
INFORMATION	15:8	RO	MSB of the information field of an outbound doorbell message that has been confirmed as successful or unsuccessful.	8'b0
INFORMATION	7:0	RO	LSB of the information field of an outbound doorbell message that has been confirmed as successful or unsuccessful.	8'b0
Note: The completed Tx doorbell message comes directly from the Tx Doorbell Completion FIFO.				

Table 4–89. Tx Doorbell Completion Status– Offset: 'h18

Field	Bit	Access	Function	Default
RSRV	31:2	RO	Reserved	30'b0
ERROR_CODE	1:0	RO	This error code corresponds to the most recently read message from the Tx Doorbell Completion. After software reads the Tx Doorbell Completion register, a read to this register should follow to determine the status of the message. 2'b00: Response DONE status 2'b01: Response with ERROR status 2'b10: Timeout error	2'b0

Table 4–90. Tx Doorbell Status Control– Offset: 'h1C

Field	Bit	Access	Function	Default
RSRV	31:2	RO	Reserved	30'B0
ERROR	1	R/W	If set, outbound doorbell messages that received a response with Error, or were timed-out, are stored in the Tx Completion FIFO. Otherwise, no error reporting will occur.	1'b0
COMPLETED	0	R/W	If set, all of the successful outbound doorbell messages are stored in the Tx Completion FIFO.	1'b0

Table 4–91. Doorbell Interrupt Enable– Offset: 'h20

Field	Bit	Access	Function	Default
RSRV	31:3	RO	Reserved	29'b0
TX_CPL_OVERFLOW	2	R/W	Tx Doorbell Completion Buffer Overflow Interrupt Enable	1'b0
TX_CPL	1	R/W	Tx Doorbell Completion Interrupt Enable	1'b0
RX	0	R/W	Doorbell received interrupt Enable	1'b0

Table 4–92. Doorbell Interrupt Status– Offset: 'h24

Field	Bit	Access	Function	Default
RSRV	31:3	RO	Reserved	29'b0
TX_CPL_OVERFLOW	2	RW1C	Interrupt asserted due to Tx Completion buffer overflow. This bit will remain set until at least one entry is read from the TX Completion FIFO. After reading at least one entry, software should clear this bit. It is not necessary to read all of the TX Completion FIFO entries.	1'b0
TX_CPL	1	RW1C	Interrupt asserted due to Tx completion status	1'b0
RX	0	RW1C	Interrupt asserted due to received messages	1'b0

MegaCore Verification

Before releasing a version of the RapidIO MegaCore function, Altera runs a comprehensive regression test, which executes the wizard to create the instance files. These files are tested in simulation and hardware to confirm functionality.

The RapidIO MegaCore function was also subjected to interoperability testing. Interoperability tests verify the performance of the MegaCore function in real-life applications, and ensure compatibility with ASSP devices.

Simulation Testing

The RapidIO core is verified using industry-standard simulators ModelSim, and VCS in combination with Vera. The test suite contains testbenches that use the RapidIO bus functional model (BFM) from the RapidIO Trade Association to verify the functionality of the IP core.

The regression suite tests various functionalities, including:

- Link initialization
- Packet format
- Packet priority
- Error handling
- Throughput
- Flow control

Hardware Testing

The RapidIO MegaCore function is tested and verified in hardware for different platforms and environments.

The hardware tests cover serial ×1 and ×4 variations running at 1.25, 2.5, and 3.125 gigabits per second (Gbps), and processing the following traffic types:

- NReads of various size payloads—4 bytes to 256 bytes
- NWrites of various size payloads—4 bytes to 256 bytes
- NWrite_R of a few different size packets
- PortWrites
- Maintenance

The hardware tests also cover the following control symbol types: Packet Accepted, Packet Retry, Packet Not Accepted, Start of Packet and Endof Packet Control Symbol, and Link Maintenance Control Symbols.

Interoperability Testing

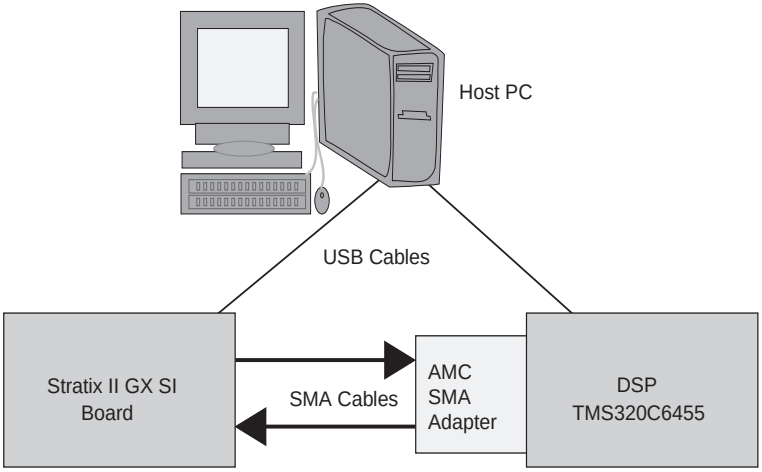
The interoperability tests performed on the RapidIO MegaCore function certify that the serial RapidIO MegaCore function has been tested for the following functionality:

- Compatibility with commercial RapidIO devices
- Device compatibility with the Stratix® GX and Stratix II GX families of devices
- Endurance
- Operation at speed

The interoperability tests were performed with the Texas Instrument DSP TMS320C6455.

The Stratix II GX test board was connected to the other RapidIO devices using SMA cables. The test sequences were executed by a NIOS II processor controlling transactions to the TMS320C6455 directly. Maintenance and I/O transactions with data integrity check were performed for each setup. Figure 4–24 shows the setup with a direct connection to the TMS320C6455.

Figure 4–24. Interoperability Test - Stratix II GX & TMS320C6455 Setup



Appendix A. Initialization Sequence

This appendix describes the most basic initialization sequence for a RapidIO™ system comprising two FPGAs, each one containing a RapidIO MegaCore® function.

To initialize the system, perform follow these steps:

1. Read the Port 0 Error and Status (ERRSTAT) command and status register (CSR) (`'h00158`) to confirm port initialization.
2. Set the following registers on the first FPGA:
    a. To set the base ID of the device to `1`, set the `DEVICE_ID` field (bits 23:16) of the `Base Device ID` register (`'h00060`) to `0x1`.
    b. To allow request packets to be issued, write 1 to the ENA field (bit 30) of the `Port General Control CSR` register (`'h13C`).
    c. To enable an all-encompassing address mapping window, write `1` to the `WEN` field (bit 4) of the `Tx Maintenance Window 0 Mask` register (`'h10104`).
    d. To set the `destination ID` (destination device) to `2`, set the `DESTINATION_ID` field (bits 23:16) of the `Tx Maintenance Window 0 Control` register (`'h1010C`) to `0x2`.
3. Set the following registers on the second FPGA:
    a. To set the base ID of the device to `2`, set the `DEVICE_ID` field (bits 23:16) of the `Base Device ID` register (`'h00060`) to `0x2`.
    b. To allow request packets to be issued, write 1 to the ENA field (bit 30) of the `Port General Control CSR` register (`'h13C`).
    c. To enable an all-encompassing address mapping window, write `1` to the WEN field (bit 4) of the `Tx Maintenance Window 0 Mask` register (`'h10104`).
    d. To set the `destination ID` (destination device) to `1`, set the `DESTINATION_ID` field (bits 23:16) of the `Tx Maintenance Window 0 Control` register (`'h1010C`) to `0x1`.

These register settings allow one FPGA to remotely access the other FPGA.

The registers follow the big endian format.

To access the registers, the system requires the following component:

- An Avalon-MM master, for example a processor, such as a Nios® embedded processor, which includes C subroutines to facilitate access to the RapidIO MegaCore functions.

You can use SOPC Builder, a Quartus® II software tool, to rapidly and easily build and evaluate your RapidIO system.

For more information on initializing a RapidIO system, refer to Fuller, Sam. 2005. *RapidIO: The Embedded System Interconnect*. John Wiley & Sons, Ltd., Chapter 10 *RapidIO Bringup and Initialization Programming*.

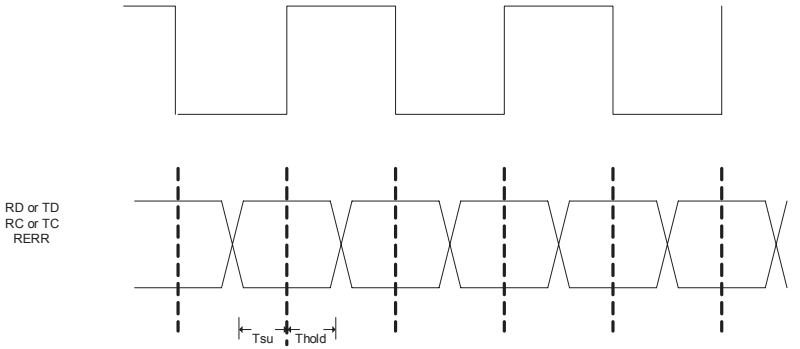
Appendix B. XGMII Interface Timing

This appendix outlines XGMII timing considerations.

Data Alignment

RapidIO transmits source-center aligned data into HSTL or SSTL pins. The clock rate required is 156.25 MHz for 3.125 Gbaud and 62.5 for 1.25 Gbaud. The following timing diagram illustrates basic timing relationships.

Figure B–1. XGMII Timing



The RapidIO XGMII interface requires the following I/O timing relationships:

- Use Fast Inputs for RD, RC and other inputs.
- Use similar clock types (for example `rclk[0]` should not be a Global clock and `rclk[1]` a regional clock).

Setting Quartus II TSU and TH Checks

You must set Quartus II TSU and TH checks. The value to use for the TSU and TH will be a function of the following:

- Effects of clock jitter, other signal integrity issues.
- Any clock phase offset on the output of the attached device
- Skew over the traces
- Typical Transmitter 960 ps for `Tsu` and `Thold` shown (3.125 Gbps)
- Receiver ideal is 480 ps for `Tsu` and `Thold` shown (3.125 Gbps)
- Adjustments for output clock phase. If not exactly center aligned, adjust the `TSU` and `TH` assignments accordingly.

Example

If the output clock was out 100 ps past the center point, add 100 ps to your `TSU` and subtract 100 ps from the `TH`. Typically, take the RX ideal and subtract the trace skew, then adjust for clock phase as shown for TAN:

- set_instance_assignment -name TSU_REQUIREMENT “400 ps” –from * -to rd
- set_instance_assignment -name TSU_REQUIREMENT “400 ps” –from * -to rc
- set_instance_assignment -name TSU_REQUIREMENT “400 ps” –from * -to rerr
- set_instance_assignment -name TH_REQUIREMENT “400 ps” –from * -to rd
- set_instance_assignment -name TH_REQUIREMENT “400 ps” –from * -to rc
- set_instance_assignment -name TH_REQUIREMENT “400 ps” –from * -to rerr

The following are for STA:

- set_max_delay -from [get_pins -hierarchical *] -to [get_ports {rd*}] 0.4
- set_max_delay -from [get_pins -hierarchical *] -to [get_ports {rc}] 0.4
- set_max_delay -from [get_pins -hierarchical *] -to [get_ports {rerr}] 0.4
- set_min_delay -from [get_pins -hierarchical *] -to [get_ports {rd*}] -0.4
- set_min_delay -from [get_pins -hierarchical *] -to [get_ports {rc}] -0.4
- set_min_delay -from [get_pins -hierarchical *] -to [get_ports {rerr}] -0.4