RapidIO™ Interconnect Specification Part 7: System and Device Inter-operability Specification

Rev. 2.2, 06/2011

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First public release	04/06/2001
1.2	Technical changes: incorporate Rev. 1.1 errata rev. 1.1.1, errata 3	06/26/2002
1.3	Technical changes: incorporate Rev 1.2 errata 1 as applicable, the following errata showings: 004-05-00002.002 Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005
2.0	Technical changes: errata showing 06-02-00001.005	06/14/2007
2.0	Removed confidentiality markings for public release	03/06/2008
2.1	No technical changes	MM/DD/200Y
2.1	Removed confidentiality markings for public release	08/13/2009
2.2	No technical changes	05/05/2011
2.2	Removed confidentiality markings for public release	06/06/2011

Questions regarding the RapidIO Trade Association, specifications, or membership should be forwarded to:

RapidIO Trade Association
12343 Hymeadow, Suite 2-R
(non-US mail deliveries to Suite 3-E)
Austin, TX 78750
512-401-2900 Tel.
512-401-2902 FAX.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Blank page

List of Figures

List of Figures

Blank Page

List of Tables

List of Tables

Blank Page

Chapter 1 Overview

1.1 Introduction

This chapter provides an overview of the *RapidIO Part 7: System and Device Inter-operability Specification* document. This document assumes that the reader is familiar with the RapidIO specifications, conventions, and terminology.

1.2 Overview

The RapidIO Architectural specifications set a framework to allow a wide variety of implementations. This document provides a standard set of device and system design solutions to provide for inter-operability.

Each chapter addresses a different design topic. This revision of the system and device inter-operability specification document covers the following issues:

Chapter 2, “System Exploration and Initialization”

Chapter 3, “RapidIO Device Class Requirements”

Chapter 4, “PCI Considerations”

Chapter 5, “Globally Shared Memory Devices”

Blank page

Chapter 2 System Exploration and Initialization

2.1 Introduction

There are several basic ways of exploring and initializing a RapidIO system. The simplest method is to somehow define the power-up state of the system components such that all devices have adequate knowledge of the rest of the system to communicate as needed. This is frequently accomplished by shifting initialization information into all of the devices in the machine at boot time from serial ROMs or similar devices. This method is most applicable for relatively static systems and systems where boot-up time is important. A second method, having processors explore and configure the system at boot time, requires more time but is much more flexible in order to support relatively fast changing plug-and-play or hot-swap systems. This document describes a simple form of this second method. A much more detailed multiple host exploration and configuration algorithm utilizing the same system reset requirements is specified in the *RapidIO Interconnect Specification Annex 1: Software/System Bring Up Specification*.

2.2 Boot code access

In most RapidIO applications system initialization requires software for exploring and initializing devices. This is typically done by a processor or set of processors in the system. The boot code for the processor(s) may reside in a ROM local to the processor(s) or on a remote RapidIO agent device. A method of accessing the boot code through an uninitialized system is required if the boot code is located on a remote RapidIO agent device.

After resetting, a processor typically vectors to a fixed address and issues a code fetch. The agent hardware between the processor and the RapidIO fabric is required to take this read request and map it automatically to a NREAD transaction. The transaction is also mapped to a dedicated device ID at the proper address offset to find the boot code. All devices between the processor and the agent device where the boot ROM resides shall default to a state that will route the NREAD transaction to the boot ROM device and route the response back to the processor. The device ID for the agent device where the boot ROM resides is device ID=0xFE (0x00FE for 16-bit device IDs). The processor default device IDs are assigned sequentially starting at 0x00 (0x0000 for 16-bit device IDs).

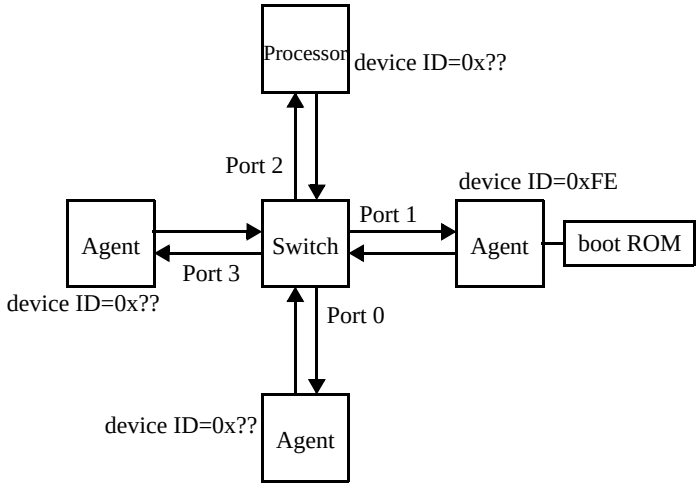

Figure 2-1. Example system with boot ROM

Figure 2-1 shows an example system with the boot ROM residing on an Agent device. The default routing state for the switch device between the processor and the agent shall allow all requests to device ID=0xFE to get to the agent device and all response packets to get from the agent device back to the processor. This means that the switch may also have to know the device ID that the processor will be using while fetching boot code (processor device IDs are assigned starting at 0x00 as described above). For the example in Figure 2-2, the system processor defaults to device ID=0x00, and the switch's default state routes device ID=0x00 to port 2.

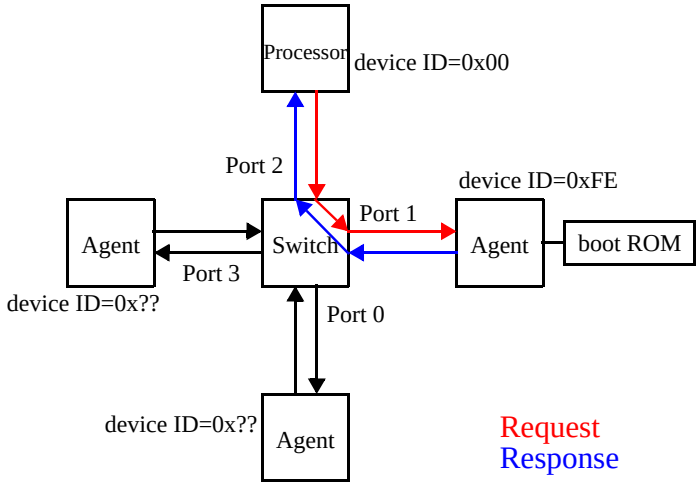

Figure 2-2. Automatically finding the boot ROM

Once the processor is able to begin running boot code, it can begin executing the exploration and initialization of the rest of the system.

2.3 Exploration and initialization

This example algorithm addresses the simple case of a system with a single processor that is responsible for exploring and initializing a system, termed a Host. The exploration and initialization process starts with a number of rules that the component and system designers shall follow.

2.3.1 Exploration and initialization rules

1. A Host shall be able to "reach" all agent devices that it is to be responsible for. This may require mechanisms to generate third party transactions to reach devices that are not transparently visible.
2. Maintenance responses generated by agent and switch devices shall be sent to the port that the maintenance request was received on. For example, consider a device that implements a 5 port switch. The system Host issues a maintenance read request to the switch device, which is received on input port 3. The switch, upon generating the maintenance response to the maintenance read request, must route it to output port 3 even though the switch may have been configured by default to route the response to a port other than port 3 (when the switch is configured it should also route the response to port 3).
3. All devices have CSRs to assist with exploration and initialization procedures. The registers used in this example contain the following information:
   - Base device ID register - This is the default device ID for the device, and it resides in a standard register in the CSR space at offset 0x60. At power-up, the base device ID defaults to logic 0xFF for all agent devices (0xFFFF for 16-bit route fields), with the exception of the boot code device and the Host device. The boot code device (if present) will have it's device ID default to 0xFE and the Host device will have it's device ID default to 0x00 as described in Section 2.2. A device may have multiple device IDs, but only this architecturally defined device ID is used in the exploration and initialization procedure.
   - Master Enable bit - the Master Enable bit is reset at power-up for agent devices and set for Host devices. The Master Enable bit is located in the *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification* or the *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification* Port General Control CSR at block offset 0x3C. If the Master Enable bit is clear the agent device is not allowed to issue requests and is only able to respond to received requests. This bit is used by the system Host to control when agents are allowed to issue transactions into the system. Switches are by default enabled and do not have a Master Enable bit.
   - Discovered bit - the Discovered bit is reset at power-up for agent devices and set for the Host device, and is located in the 8/16 LP-LVDS or 1x/4x LP-Serial physical layer Port General Control

CSR at block offset 0x3C. The system Host device sets this bit when the device has been discovered through the exploration mechanism. The Discovered bit is useful for detecting routing loops, and for hot plug or swap environments.

2.3.2 Exploration and initialization algorithm

If the above rules are followed, all agent devices are now accessible either as an end point that responds to any maintenance transaction or, for switches, via the hop_count mechanism.

The basic algorithm is to explore the system through each end point in sequence by first locating the adjacent device by sending a maintenance read to device ID=0xFF and hop count= 0x00, which is guaranteed to cause the adjacent device to respond. That device is then configured to reach the next device by assigning it a unique base device ID other than 0xFF, setting up route tables to reach the next device, etc.

When all devices in the system have been identified and have unique base device IDs assigned (no devices have a base device ID value=0xFF), the Host can then complete the final device ID assignment and configuration required for the application and enable agent devices to issue requests.

2.3.3 Exploration and initialization example

Figure 2-3 shows the previous example of a small single Host system.

Following the rules defined above, the base device ID value for all devices except the Host and boot ROM device after reset is applied is 0xFF, the Host has it's Master Enable and Discovered bits set, and the agent devices have their Master Enable and Discovered bits cleared.

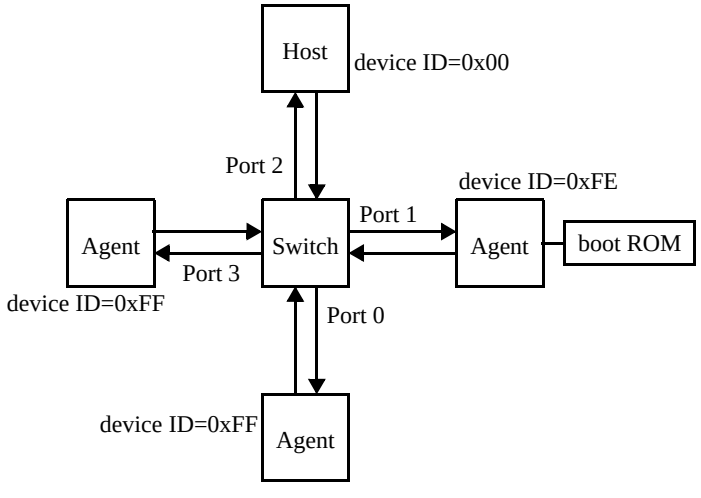

Figure 2-3. Example system

Assigning the Host's base device ID=0x00 is the first step in the process. The next step is to find the adjacent device, so the Host sends a maintenance read of offset 0x00_0000 to device ID=0xFF and hop_count=0x00. The switch consumes the request because the hop_count field is equal to zero and responds by sending the contents of it's Device Identity and Information CARs back to the port the request came from. From the returned information, the software on the Host can identify this as a switch. The Host then reads the switch port information CAR at offset 0x00_0014 to find out which port it is connected to. The response indicates a 4 port switch (which the Host may have already known from the device information register), connected to port 2.

The Host then examines the default routing tables for the switch to find the port route for the boot device ID=0xFE so it can preserve the path to the boot code (which it may still be running), and discovers that the boot device is located through port 1 of the switch. It also sets the switch's Discovered bit.

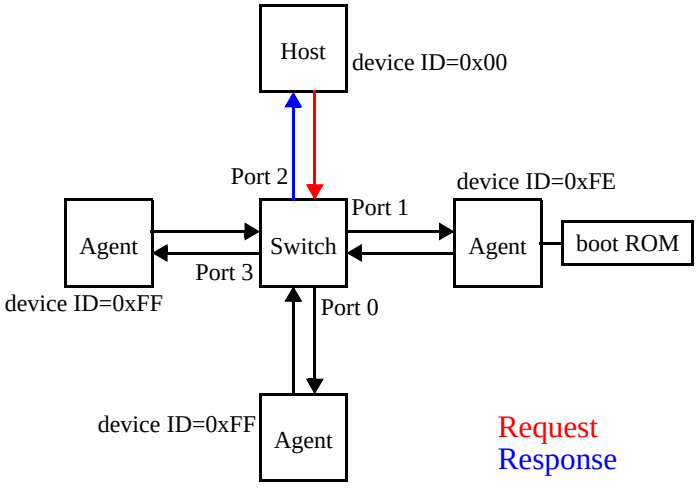

Figure 2-4. Finding the adjacent device

The next step is for the Host to configure the switch to route device ID=0xFF to port 0 and device ID=0x00 to port 2 (which it already was because of the boot device in the system) via maintenance write requests to hop_count=0x00. The Host then issues another maintenance read request, this time to device ID=0xFF and hop_count=0x01. The switch discovers that it is not the final destination of the maintenance request packet, so it decrements the hop_count and routes the packet to port 0 and on to the attached agent device. The agent device responds, and the switch routes the response packet to device ID=0x00 back through port 2 to the Host. Again, software identifies the device, sets its Discovered bit, configures it as required, and assigns the base device ID=0x01.

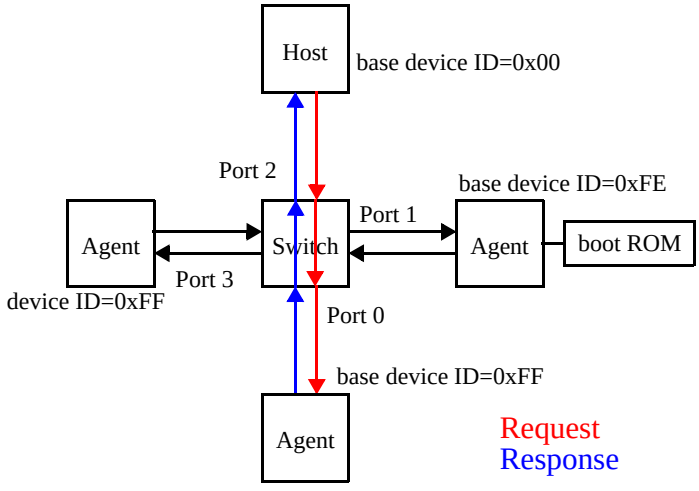

Figure 2-5. Finding the device on switch port 0

The Host then modifies the routing tables to now route device ID=0x01 to port 0. Since the boot device is located through port 1, instead of modifying the routing tables to route device ID=0xFF to port 1, the Host issues a maintenance read of device ID=0xFE (the boot device) and hop_count=0x01. The response identifies the agent on port 1, sets the agent's Discovered bit, and configures it as necessary, leaving the base device ID=0xFE so the Host can continue to execute the boot code.

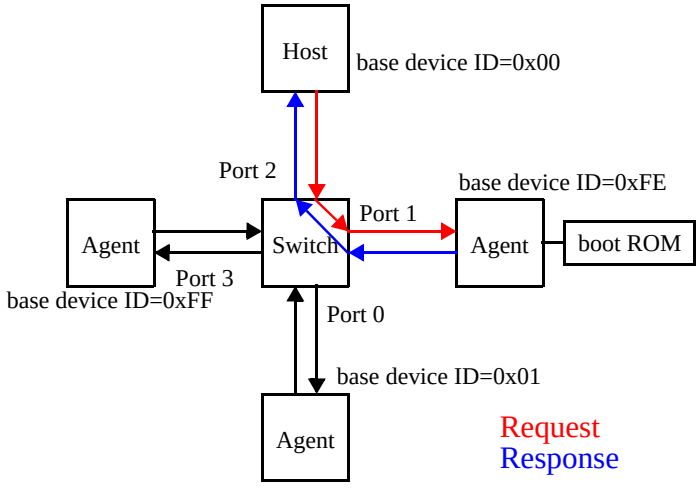

Figure 2-6. Finding the device on switch port 1

For the next iteration, the Host sets the switch device routing table entry for device ID=0xFF to route to port 3 (the Host already knows it is directly connected to port 2), and issues the maintenance read transaction as before.

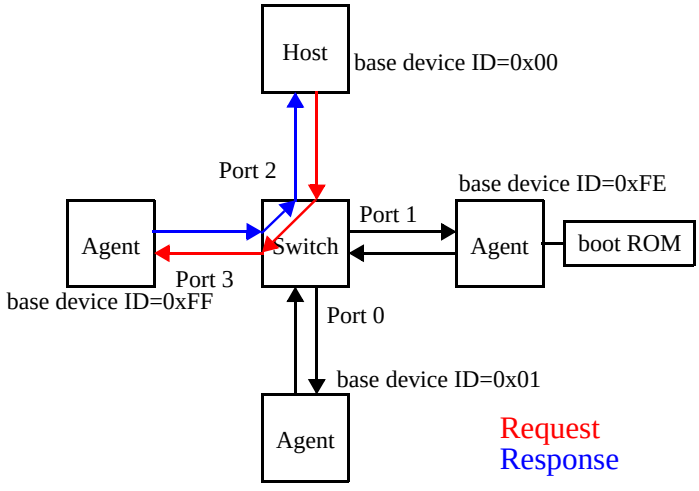

Figure 2-7. Finding the device on switch port 3

When the end point only agent responds with the requested CAR information the Host now knows that exploration is completed (there are no other paths to follow through the fabric), and can finalize configuring the system as shown in Figure 2-8. The agent devices can then have their Master Enable bits set so they can begin to issue transactions into the initialized system. The boot device ID can be changed, if desired, when the Host completes executing code from the boot ROM.

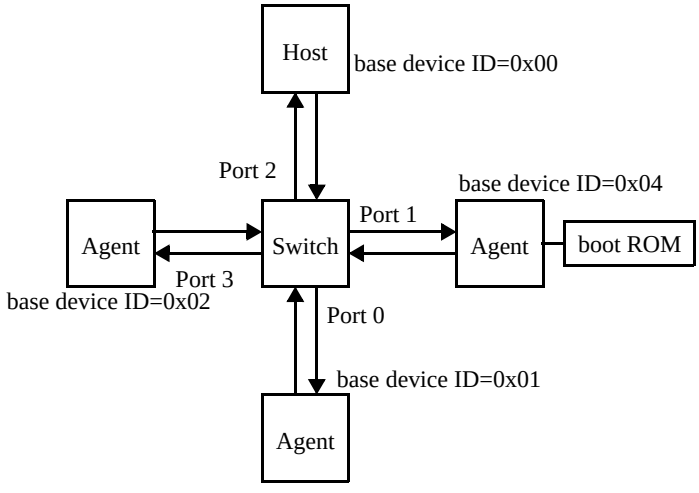

Figure 2-8. Final initialized system state

Variants to this procedure may be desirable. For example, a system may wish to enable some devices before exploration has been completed.

More complex systems with multiple Hosts, failed Host recovery, and hot swap requirements can be addressed with more complex algorithms utilizing the Host

base device ID Lock Register and the Component Tag Register in standard registers in the CSR space at offsets 0x68 and 0x6C.

Chapter 3 RapidIO Device Class Requirements

3.1 Introduction

The RapidIO Architecture specifications allow for a variety of implementations. In order to form standard points of support for RapidIO, this chapter describes the requirements for RapidIO devices adhering to the *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification* or the *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification* and corresponding to different measures of functionality. Three device "classes" are defined, each with a minimum defined measure of support. The first class defines the functionality of the least capable device, with subsequent classes expanding the measure of support, in order to establish levels of inter-operability.

3.2 Class Partitioning

Each class includes the functionality defined in all previous class devices and defines the minimum additional functionality for that class. A device is not required to comply *exactly* with a class, but may optionally supply additional features as a value-add for that device. All functions that are not required in any class list are also optional value-adds for a device.

First is a set of requirements that are applicable to all RapidIO compliant devices, including switch devices without end point functionality.

3.2.1 Generic: All devices

3.2.1.1 General requirements

- One or more 8/16 LP-LVDS and/or 1x/4x LP-Serial ports
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification* and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*)
- Support for small (8-bit) transport device ID fields
  - (refer to *RapidIO Part 3: Common Transport Specification*, Section 2.4)
- Ability to accept requests with all sourceID and destinationID values on exit from reset
  - (refer to *RapidIO Part 3: Common Transport Specification*, Section 2.3)
- Support for recovery from a single corrupt packet or control symbol
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*,

Section 1.3.5) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.10.2)

- Support for packet retry protocol
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.2.4) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.6)
- Support for throttle based flow control on 8/16 LP-LVDS physical layer ports
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 2.3)
- Support for transaction ordering for flowID B
  - (end point programmability for all flow levels is recommended)
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.2.2) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.3.3)
- Switch devices maintain error coverage internally
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.3.6) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.5)
- Support for maximum size (276 byte) packets for switch devices
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.4) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 2.4)
- Support for maximum size (256 byte) data payloads for end point devices
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 3.1.2)
- Device must contain the following registers:
  - Device Identity CAR
  - Device Information CAR
  - Assembly Identity CAR
  - Assembly Information CAR
  - Processing Element Features CAR
  - Source Operations CAR
  - Destination Operations CAR
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 4.4)

3.2.1.2 Operation support as target

- Maintenance read
  - (switch targeted by hop_count transport field)
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section

2.3.1, Section 3.1.10)

- Maintenance write
  - (switch targeted by hop_count transport field)
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.3.1, Section 3.1.10)

3.2.1.3 Operation support as source

- <none>

3.2.2 Class 1: Simple target device

3.2.2.1 General requirements

- all Generic requirements
- Support for 34-bit address packet formats
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 4.4.5)

3.2.2.2 Operation support as target

- all Generic requirements
- Write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.7)
- Streaming-write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.8)
- Write-with-response
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.3, Section 3.1.7)
- Read
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.1, Section 3.1.5)

3.2.2.3 Operation support as source

- all Generic requirements

3.2.3 Class 2: Simple mastering device

3.2.3.1 General requirements

- all Class 1 requirements

3.2.3.2 Operation support as target

- all Class 1 requirements

3.2.3.3 Operation support as source

- all Class 1 requirements
- Maintenance read
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.3.1, Section 3.1.10)
- Maintenance write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.3.1, Section 3.1.10)
- Write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.7)
- Streaming-write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.8)
- Write-with-response
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.3, Section 3.1.7)
- Read
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.1, Section 3.1.5)

3.2.4 Class 3: Complex mastering device

3.2.4.1 General requirements

- all Class 2 requirements

3.2.4.2 Operation support as target

- all Class 2 requirements
- Atomic set
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.4, Section 3.1.7)
- Maintenance port-write
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.3.1, Section 3.1.10)
- Data message mailbox 0, letter 0, single segment, 8 byte payload
  - (refer to *RapidIO Part 2: Message Passing Logical Specification*, Section 2.2.2, Section 3.1.5)

3.2.4.3 Operation support as source

- all Class 2 requirements
- Atomic set
  - (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.4, Section 3.1.7)
- Data message mailbox 0, letter 0, single segment, 8 byte payload
  - (refer to *RapidIO Part 2: Message Passing Logical Specification*, Section 2.2.2, Section 3.1.5)

Blank page

Chapter 4 PCI Considerations

4.1 Introduction

RapidIO contains a rich enough set of operations and capabilities to allow transport of legacy interconnects such as PCI[1]. While RapidIO and PCI share similar functionality, the two interconnects have different protocols thus requiring a translation function to move transactions between them. A RapidIO to PCI bridge processing element is required to make the necessary translation between the two interconnects. This chapter describes architectural considerations for an implementation of a RapidIO to PCI bridge processing element. This chapter is not intended as an implementation instruction manual, rather, it is to provide direction to the bridge processing element architect and aid in the development of interoperable devices. For this chapter it is assumed that the reader has a thorough understanding of the PCI 2.2 and/or the PCI-X 1.0 specifications.

Figure 4-1 shows a typical system with devices connected using various RapidIO and PCI bus segments. A host bridge is connected to various peripherals via a PCI bus. A RapidIO bridge is used to translate PCI formatted transactions to the equivalent RapidIO operations to allow access to the rest of the system, including additional subordinate PCI bus segments.

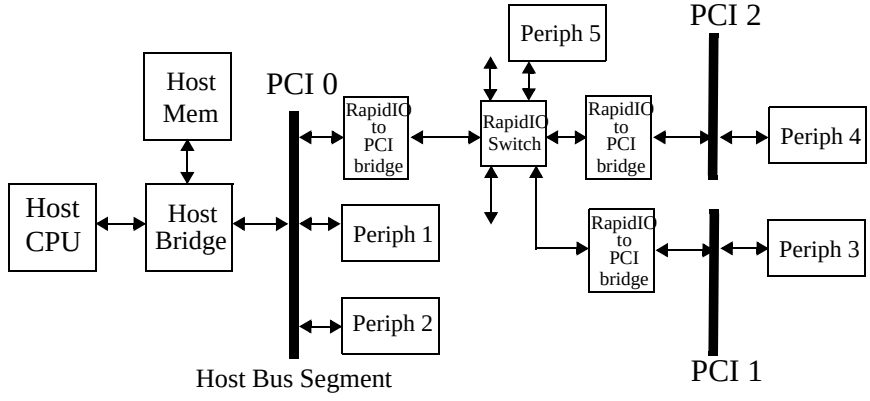

Figure 4-1. Example System with PCI and RapidIO

[1]For additional information on the Peripheral Component Interconnect PCI refer to the PCI 2.2 and the PCI-X 1.0 specifications.

Where RapidIO is introduced into a legacy system, it is desirable to limit changes to software. For transactions which must travel between RapidIO and PCI it is necessary to map address spaces defined on the PCI bus to those of RapidIO, translate PCI transaction types to RapidIO operations, and maintain the producer/consumer requirements of the PCI bus. This chapter will address each of these considerations for both PCI version 2.2 and PCI-X.

4.2 Address Map Considerations

PCI defines three physical address spaces, specifically, the memory, I/O memory, and configuration spaces. RapidIO, on the other hand, only addresses memory and configuration space. This section discusses memory space. Configuration space is discussed in Section 4.4. Figure 4-2 shows a simple example of the PCI memory and I/O address spaces for a host bus segment. In order for devices on the PCI bus to communicate with those connected through RapidIO, it is necessary to provide a memory mapping function. The example PCI host memory map uses a 32-bit physical address space resulting in 4 Gbytes of total address space. Host memory is shown at the bottom of the address map and peripheral devices at the top. Consider that the RapidIO to PCI bridge processing element contains a specified window(s) of address space mapped to it using the PCI base address register(s)[1]. The example shown in Figure 4-2 illustrates the RapidIO bridge address window located in an arbitrary software defined location. Likewise, if it was desired to communicate with PCI legacy I/O devices over RapidIO an I/O window would be assigned to the RapidIO to PCI bridge as shown.

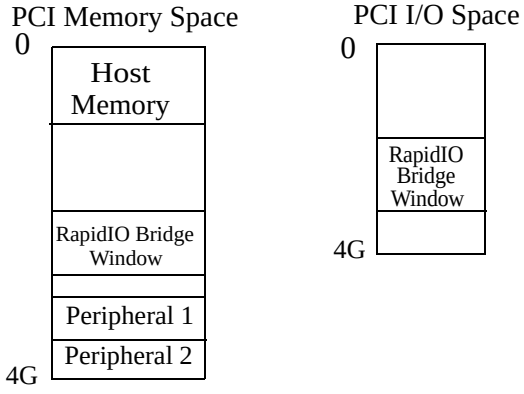

Figure 4-2. Host segment PCI Memory Map Example

Any transactions issued to the bus segment with an address that matches the RapidIO bridge window will be captured by the RapidIO to PCI bridge for forwarding. Once the transaction has been accepted by the RapidIO to PCI bridge processing element it must be translated to the proper RapidIO context as shown in

[1]Refer to the PCI 2.2 Specification Chapter 6 for a discussion on PCI address maps and configuration registers

Figure 4-3. For the purposes of this discussion this function is called the Address Mapping and Translation function (AMT). The AMT function is responsible for translating PCI addresses to RapidIO addresses as well as the translation and assignment of the respective PCI and RapidIO transaction types. The address space defined by the RapidIO bridge window may represent more than one subordinate RapidIO target device. A device on PCI bus segment 0 shown in Figure 4-1 may require access to a peripheral on PCI bus 1, bus 2, or RapidIO Peripheral 5. Because RapidIO uses source addressing (device IDs), the AMT is responsible for translating the PCI address to both a target device ID and associated offset address. In addition to address translation, RapidIO attributes, transaction types, and other necessary delivery information are established.

Similarly, transactions traveling from a RapidIO bus to a PCI bus must also pass through the AMT function. The address and transaction type are translated back into PCI format, and the AMT selects the appropriate address for the transaction. Memory mapping is relied upon for all transactions bridged between PCI and RapidIO.

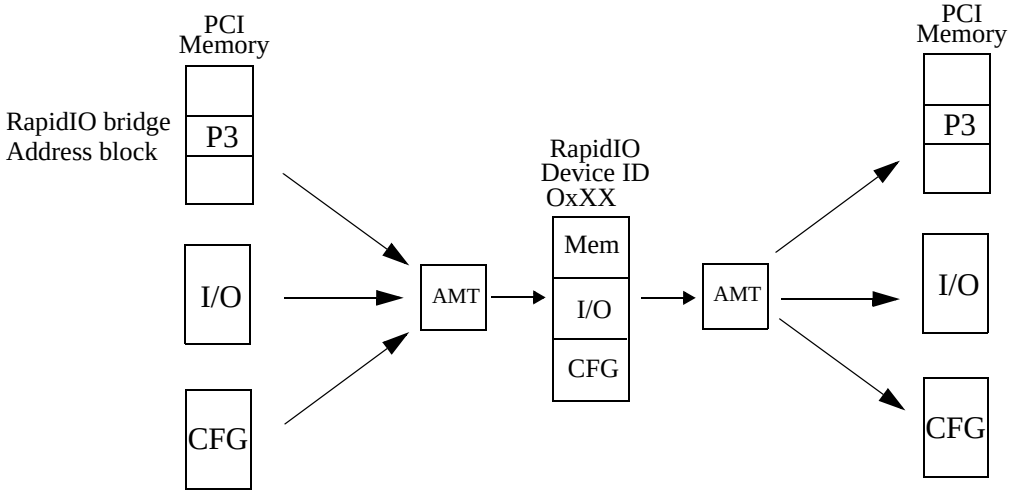

Figure 4-3. AMT and Memory Mapping

4.3 Transaction Flow

In considering the mapping of the PCI bus to RapidIO it is important to understand the transaction flow of PCI transactions through RapidIO.

4.3.1 PCI 2.2 Transaction Flow

The PCI 2.2 specification defines two classes of transaction types, posted and non-posted. Figure 4-4 shows the route taken by a PCI-RapidIO posted write transaction. Once the request is sent from the PCI Master on the bus, it is claimed by

the bridge processing element which uses the AMT to translate it into a RapidIO request. Only when the transaction is in RapidIO format can it be posted to the RapidIO target. In some cases it may be desirable to guarantee end to end delivery of the posted write transaction. For this case the RapidIO NWRITE_R transaction is used which results in a response as shown in the figure.

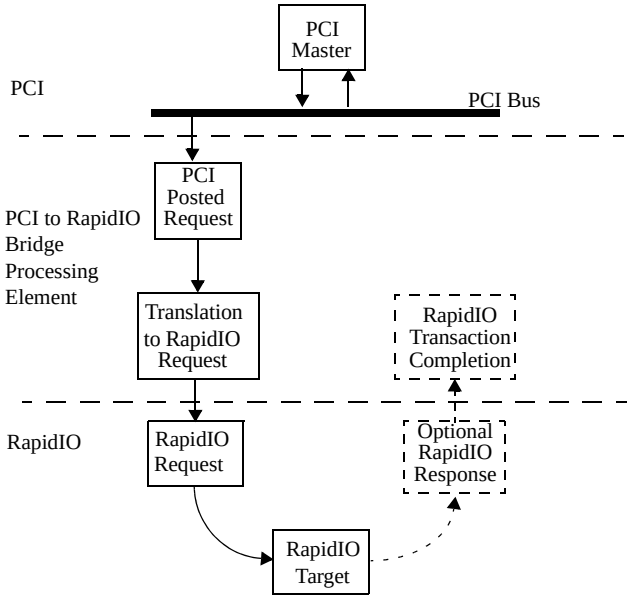

Figure 4-4. PCI Mastered Posted Write Transaction Flow Diagram

A non-posted PCI transaction is shown in Figure 4-5. The transaction is mastered by the PCI agent on the PCI bus and accepted by the RapidIO to PCI bridge. The transaction is retried on the PCI bus if the bridge is unable to complete it within the required timeout period. In this case the transaction is completed as a delayed transaction. The transaction is translated to the appropriate RapidIO operation and issued on the RapidIO port. At some time later a RapidIO response is received and the results are translated back to PCI format. When the PCI master subsequently retries the transaction, the delayed results are returned and the operation is completed.

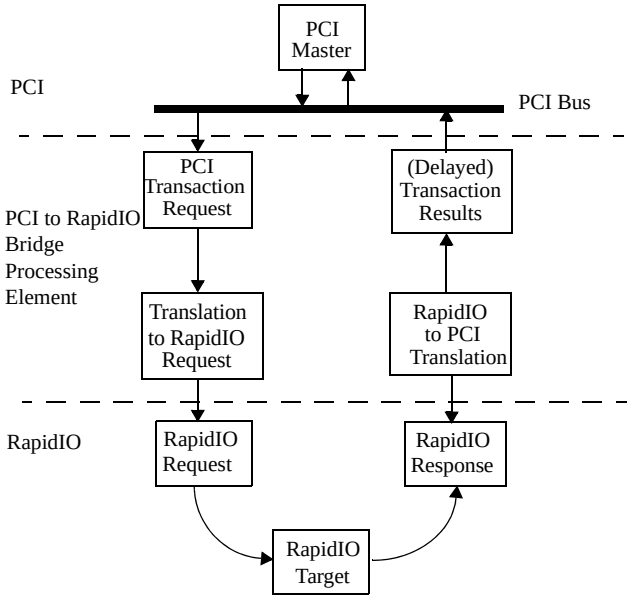

Figure 4-5. PCI Mastered non-posted (delayed) Transaction Flow Diagram

Because PCI allows unbounded transaction data tenures, it may be necessary for the RapidIO to PCI bridge to break the single PCI transaction into multiple RapidIO operations. In addition, RapidIO does not have byte enables and therefore does not support sparse byte transactions. For this case the transaction must be broken into multiple operations as well. “Section 4.7, Byte Lane and Byte Enable Usage” on page 41 describes this situation in more detail.

A RapidIO mastered operation is shown in Figure 4-6. For this case the RapidIO request transaction is received at the RapidIO to PCI bridge. The bridge translates the request into the appropriate PCI command which is then issued to the PCI bus. The PCI target may complete the transaction as a posted, non-posted, or delayed non-posted transaction depending on the command type. Once the command is successfully completed on the PCI bus the results are translated back into the RapidIO format and a response transaction is issued back to the RapidIO Master.

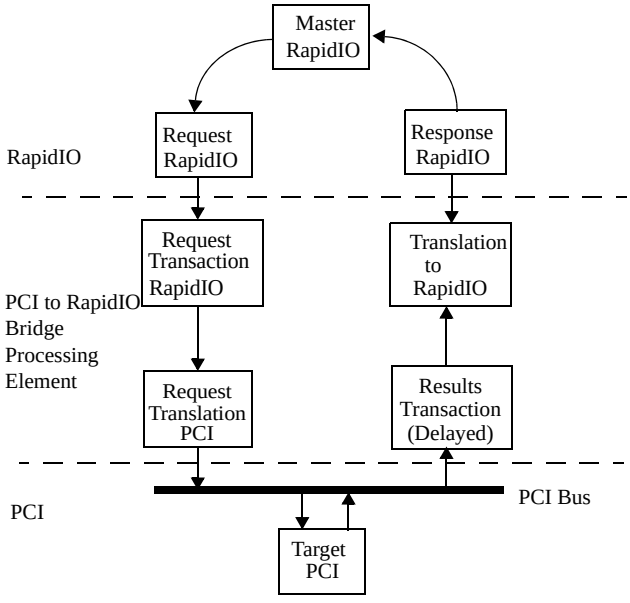

Figure 4-6. RapidIO Mastered Transaction

4.3.2 PCI-X Transaction Flow

The flow of transactions described in the previous section applies to the PCI-X bus as well. PCI-X supports split transactions instead of delayed transactions. The example shown in Figure 4-7 illustrates a transaction completed with a PCI-X split completion. The PCI-X master issues a transaction. The RapidIO to PCI-X bridge determines that it must complete the transaction as a split transaction, and responds with a split response. The transaction is translated to RapidIO and a request is issued on the RapidIO port. The RapidIO target returns a response transaction which is translated to a PCI-X Split Completion transaction completing the operation. PCI-X allows up to a 4 Kilobyte request. Larger PCI-X requests must be broken into multiple RapidIO operations. The RapidIO to PCI-X bridge may return the results back to the PCI-X Master using multiple Split Completion transactions in a pipelined fashion. Since PCI-X only allows devices to disconnect on 128 byte boundaries it is advantageous to break the large PCI-X request into either 128 or 256 byte RapidIO operations.

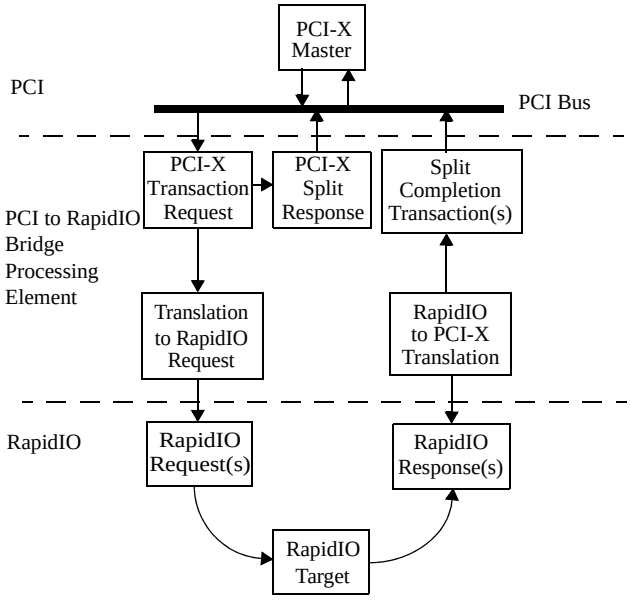

Figure 4-7. PCI-X Mastered Split Response Transaction

4.4 RapidIO to PCI Transaction Mapping

The RapidIO I/O and GSM specifications include the necessary transactions types to map all PCI transactions. Table 4-1 lists the map of transactions between PCI and RapidIO. A mapping mechanism such as the AMT function described in Section 4.2 is necessary to assign the proper transaction type based on the address space for which the transaction is targeted.

Table 4-1. PCI 2.2 to RapidIO Transaction Mapping

PCI Command	RapidIO Transaction	Comment
Interrupt-acknowledge	NREAD	
Special-cycle	NWRITE	
I/O-read	NREAD	
I/O-write	NWRITE_R	
Memory-read, Memory-Read-Line, Memory-Read-Multiple	NREAD or IO_READ_HOME	The PCI memory read transactions can be represented by the NREAD operation. If the operation is targeted to hardware maintained globally coherent memory address space then the I/O Read operation must be used (see “Section 4.6, Interactions with Globally Shared Memory” on page 37.)
Memory-write, Memory-write-and-invalidate	NWRITE, NWRITE_R, or FLUSH	The PCI Memory Write and Memory-Write-and-Invalidate can be represented by the NWRITE operation. If reliable delivery of an individual write transaction is desired then the NWRITE_R is used. If the operation is targeted to hardware maintained globally coherent memory address space then the Data Cache Flush operation must be used (refer to “Section 4.6, Interactions with Globally Shared Memory” on page 37.)

Table 4-1. PCI 2.2 to RapidIO Transaction Mapping

PCI Command	RapidIO Transaction	Comment
Configuration-read	NREAD	
Configuration-write	NWRITE_R	

PCI 2.2 memory transactions do not specify a size. It is possible for a PCI master to read a continuous stream of data from a target or to write a continuous stream of data to a target. Because RapidIO is defined to have a maximum data payload of 256 bytes, PCI transactions that are longer than 256 bytes must be broken into multiple RapidIO operations.

Table 4-2 shows the transaction mapping between PCI-X and RapidIO.

Table 4-2. PCI-X to RapidIO Transaction Mapping

PCI-X Command	RapidIO Transaction	Comment
Interrupt-acknowledge	NREAD	
Special-cycle	NWRITE	
I/O-read	NREAD	
I/O-write	NWRITE_R	
Memory-read DWORD	NREAD or IO_READ_HOME	The PCI-X memory read DWORD transactions can be represented by the NREAD operation. If the operation is targeted to hardware maintained coherent memory address space then the I/O Read operation must be used (refer to “Section 4.6, Interactions with Globally Shared Memory” on page 37.) This is indicated in PCI-X using the No Snoop (NS) bit described in Section 2.5 of the PCI-X 1.0 specification.
Memory-write	NWRITE, NWRITE_R, or FLUSH	The PCI-X Memory Write and Memory-Write-and-Invalidate can be represented by the NWRITE operation. If reliable delivery of an individual write transaction is desired then the NWRITE_R is used. If the operation is targeted to hardware maintained coherent memory address space then the Data Cache Flush operation must be used (refer to “Section 4.6, Interactions with Globally Shared Memory” on page 37.) This is indicated in PCI-X using the No Snoop (NS) bit described in Section 2.5 of the PCI-X 1.0 specification.
Configuration-read	NREAD	
Configuration-write	NWRITE_R	
Split Completion	--	The Split Completion transaction is the result of a request on the PCI-X bus that was terminated by the target with a Split Response. In the case of the RapidIO to PCI-X bridge this would be the artifact of a transaction that either the bridge mastered and received a split response or was the target and issued a split response. This command is equivalent to a RapidIO response transaction and does not traverse the bridge.

Table 4-2. PCI-X to RapidIO Transaction Mapping

PCI-X Command	RapidIO Transaction	Comment
Memory-read-block	NREAD or IO_READ_HOME	The PCI-X memory read transactions can be represented by the NREAD operation. If the operation is targeted to hardware maintained globally coherent memory address space then the I/O Read operation must be used (refer to "Section 4.6, Interactions with Globally Shared Memory" on page 37.) This is indicated in PCI-X using the No Snoop (NS) bit described in Section 2.5 of the PCI-X 1.0 specification.
Memory-write-block	NWRITE, NWRITE_R, or FLUSH	The PCI-X Memory Write and Memory-Write-and-Invalidate can be represented by the NWRITE operation. If reliable delivery of an individual write transaction is desired then the NWRITE_R is used. If the operation is targeted to hardware maintained globally coherent memory address space then the Data Cache Flush operation must be used (refer to "Section 4.6, Interactions with Globally Shared Memory" on page 37.) This is indicated in PCI-X using the No Snoop (NS) bit described in Section 2.5 of the PCI-X 1.0 specification.

The PCI-X addendum to the PCI specification adds the ability to do split operations. This results in an operation being broken into a Split Request and one or more Split Completions. As a target of a PCI-X Split Request, the RapidIO to PCI bridge may reply with a Split Response and complete the request using multiple RapidIO operations. The results of these operations are issued on the PCI-X bus as Split Completions. If the RapidIO to PCI-X bridge is the initiator of a Split Request, the target may also indicate that it intends to run the operation as a split transaction with a Split Response. In this case the target would send the results to the RapidIO to PCI-X bridge using Split Completions.

4.5 Operation Ordering and Transaction Delivery

This section discusses what the RapidIO to PCI bridge must do to address the requirements of the ordering rules of the PCI specifications.

4.5.1 Operation Ordering

Section 1.2.1 of the *RapidIO Part 1: Input/Output Logical Specification* describes a set of ordering rules. The rules guarantee ordered delivery of write data and that results of read operations will contain any data that was previously written to the same location.

For bridge devices, the PCI 2.2 specification has the additional requirement that the results of a read command push ahead posted writes in both directions.

In order for the RapidIO to PCI bridge to be consistent with the PCI 2.2 ordering rules it is necessary to follow the transaction ordering rules listed in section 1.2.1 of the I/O logical specification. In addition, the RapidIO to PCI bridge is required to adhere to the following RapidIO rule:

Read responses must push ahead all write requests and write responses.

4.5.2 Transaction Delivery Ordering

The RapidIO 8/16 LP-LVDS and 1x/4x LP-Serial physical layer specifications describe the mechanisms by which transaction ordering and delivery occur through the system. When considering the requirements for the RapidIO to PCI bridge it is first necessary to follow the transaction delivery ordering rules in section 1.2.4.1 of the 8/16 LP-LVDS specification and/or Section 5.8 of the 1x/4x LP-Serial specification. Further, it is necessary to add additional constraints to maintain programming model compatibility with PCI.

As described in Section 4.5.1 above, PCI has an additional transaction ordering requirement over RapidIO. In order to guarantee inter-operability, transaction ordering, and deadlock free operation, it is recommended that devices be restricted to utilizing transaction request flow level 0. In addition, it is recommended that response transactions follow a more strict priority assignment. Table 4-3 illustrates the priority assignment requirements for transactions in the PCI to RapidIO environment.

Table 4-3. Packet priority assignments for PCI ordering

RapidIO packet type	priority	comment
read request	0	This will push write requests and responses ahead
write request	1	Forces writes to complete in order, but allows write requests to bypass read requests
read response	1	Will force completion of preceding write requests and allows bypass of read requests
write response	2	Will prevent NWRITE_R request based deadlocks

The PCI transaction ordering model requires that a RapidIO device not issue a read request into the system unless it has sufficient resources available to receive and process a higher priority write or response packet in order to prevent deadlock. PCI 2.2 states that read responses cannot pass write transactions. The RapidIO specification provides PCI ordering by issuing priority 0 to read requests, and priority 1 to read responses and PCI writes. Since read responses and writes are issued at the same priority, the read responses will not pass writes.

4.5.3 PCI-X Relaxed Ordering Considerations

The PCI-X specification defines an additional ordering feature called relaxed ordering. If the PCI-X relaxed ordering attribute is set for a read transaction, the results for the read transaction are allowed to pass posted write transactions. PCI-X read transactions with this bit set allow the PCI-X to RapidIO bridge to ignore the rule described in Section 4.5.1. Table 4-4 shows the results of this additional

function.

Table 4-4. Packet priority assignments for PCI-X ordering

RapidIO packet type	priority	comment
read request	0	This will push write requests and responses ahead
write request	1	Forces writes to complete in order, but allows write requests to bypass of read requests
read response	1	When PCI-X Relaxed Ordering attribute is set to 0. Will force completion of preceding write requests and allows bypass of read requests
read response	2, 3	When PCI-X Relaxed Ordering attribute is set to 1. The endpoint may promote the read response to higher priority to allow it to move ahead of posted writes.
write response	2	

4.6 Interactions with Globally Shared Memory

Traditional systems have two notions of system or subsystem cache coherence. The first, non-coherent, means that memory accesses have no effect on the caches in the system. The memory controller reads and writes memory directly, and any cached address becomes incoherent in the system. This behavior requires that all cache coherence with I/O be managed using software mechanisms, as illustrated in Figure 4-8.

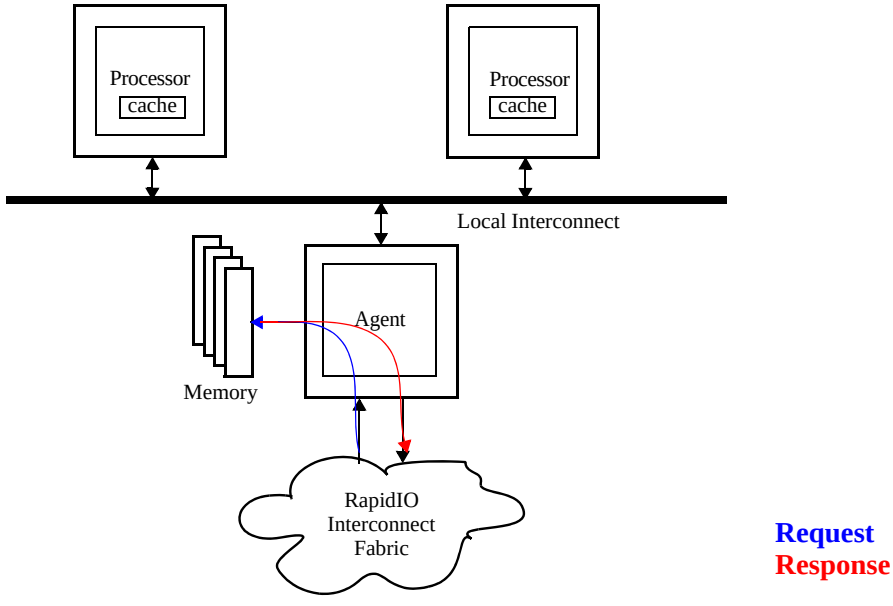

Figure 4-8. Traditional Non-coherent I/O Access Example

The second notion of system cache coherence is that of global coherence. An I/O access to memory causes a snoop cycle to be issued on the processor bus, keeping all of the system caches coherent with the memory, as illustrated in Figure 4-9.

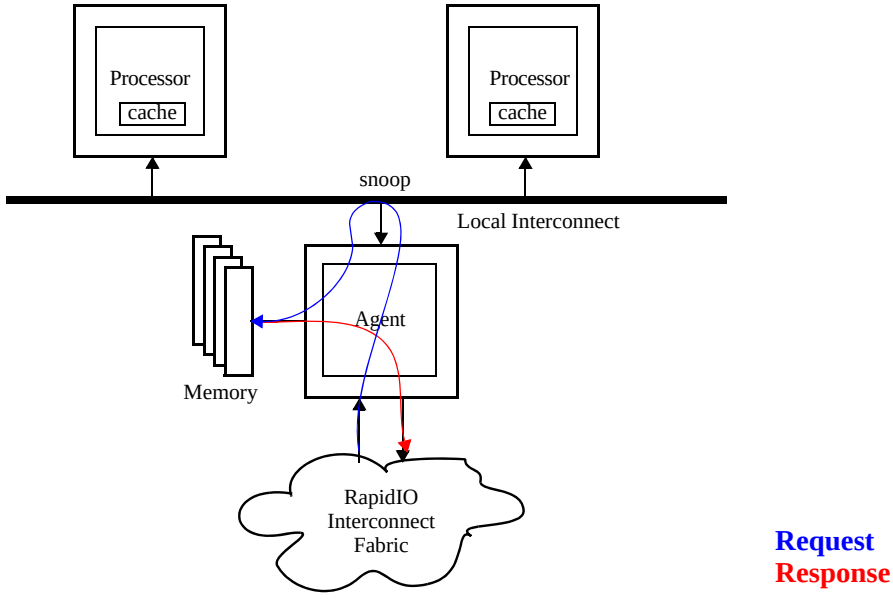

Figure 4-9. Traditional Globally Coherent I/O Access Example

With RapidIO globally shared systems, there is no common bus that can be used in order to issue the snoop, so global coherence requires special hardware support beyond simply snooping the bus. This leads to a third notion of cache coherence, termed local coherence. For local coherence, a snoop on a processor bus local to the targeted memory controller can be used to keep those caches coherent with that part of memory, but not caches associated with other memory controllers, as illustrated in Figure 4-10. Therefore, what once was regarded in a system as a "coherent access" is no longer globally coherent, but only locally coherent. Typically, deciding to snoop or not snoop the local processor caches is either determined by design or system architecture policy (always snoop or never snoop), or by an attribute associated with the physical address being accessed. In PCI-X, this attribute is the No Snoop (NS) bit described in Section 2.5 of the PCI-X 1.0 specification.

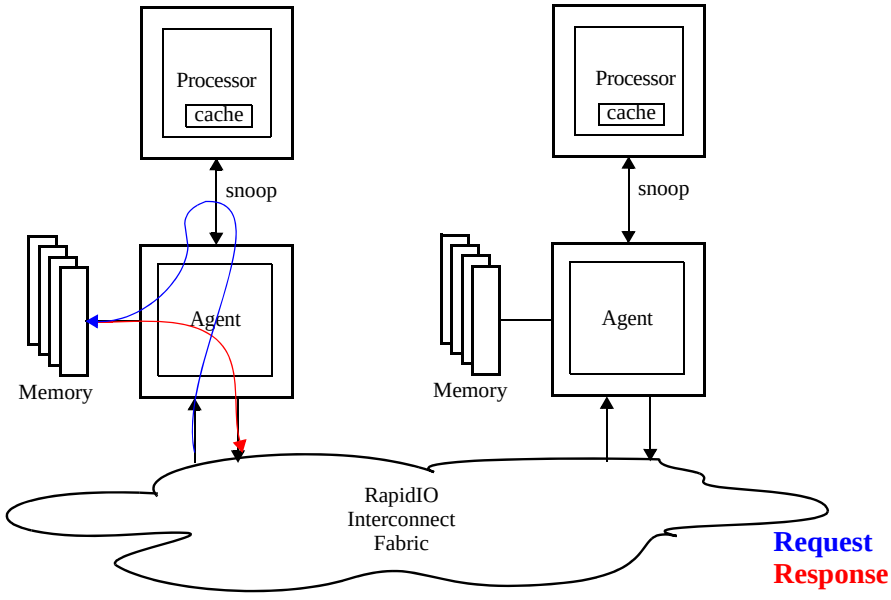

Figure 4-10. RapidIO Locally Coherent I/O Access Example

In order to preserve the concept of global cache coherence for a system, the *RapidIO Part 5: Globally Shared Memory Logical Specification* defines several operations that allow a RapidIO to PCI bridge processing element to access data in the globally shared space without having to implement all of the cache coherence protocol. These operations are the I/O Read and Data Cache Flush operations (globally shared memory specification, sections 3.2.9 and 3.2.10). For PCI-X bridging, these operations can also be used as a way to encode the NO SNOOP attribute for locally as well as globally coherent transactions. The targeted memory controller can be designed to understand the required behavior of such a transaction. These encodings also are useful for tunneling PCI-X transactions between PCI-X bridge devices.

The data payload for an I/O Read operation is defined as the size of the coherence granule for the targeted globally shared memory domain. However, the Data Cache Flush operation allows coherence granule, sub-coherence granule, and sub-double-word writes to be performed.

The IO_READ_HOME transaction is used to indicate to the GSM memory controller that the memory access is globally coherent, so the memory controller finds the latest copy of the requested data within the coherence domain (the requesting RapidIO to PCI bridge processing element is, by definition, not in the coherence domain) without changing the state of the participant caches. Therefore, the I/O Read operation allows the RapidIO to PCI bridge to cleanly extract data from a coherent portion of the system with minimal disruption and without having to be a full participant in the coherence domain.

The Data Cache Flush operation has several uses in a coherent part of a system. One

such use is to allow a RapidIO to PCI bridge processing element to write to globally shared portions of the system memory. Analogous to the IO_READ_HOME transaction, the FLUSH transaction is used to indicate to the GSM memory controller that the access is globally coherent. The memory controller forces all of the caches in the coherence domain to invalidate the coherence granule if they have a shared copy (or return the data to memory if one had ownership of the data), and then writes memory with the data supplied with the FLUSH request. This behavior allows the I/O device to cleanly write data to the globally shared address space without having to be a full participant in the coherence domain.

Since the RapidIO to PCI bridge processing element is not part of the coherence domain, it is never the target of a coherent operation.

4.6.1 I/O Read Operation Details

Most of the complexity of the I/O Read operation resides in the memory controller. For the RapidIO to PCI Bridge processing element the I/O Read operation requires some additional attention over the non-coherent read operation. The necessary portions of the I/O Read state machine description in Section 6.10 of the globally shared memory specification are extracted below. Refer to Chapter 6 of the GSM specification for state machine definitions and conventions. The GSM specification takes precedence in the case of any discrepancies between the corresponding portions of the GSM specification and this description.

4.6.1.1 Internal Request State Machine

This state machine handles requests to the remote globally shared memory space.

```
remote_request(IO_READ_HOME, mem_id, my_id);
```

4.6.1.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect.

```
switch(remote_response)
case DONE:
        return_data();
        free_entry();
case DONE_INTERVENTION:                 // must be from third party
        set_received_done_message();
        if (received_data_only_message)
                free_entry();
        else
                // wait for a DATA_ONLY
        endif;
case DATA_ONLY:                         // this is due to an intervention, a
                                        // DONE_INTERVENTION should come
                                        // separately
        set_received_data_only_message();
        if (received_done_message)
                return_data();
                free_entry();
        else
                return_data();          // OK for weak ordering
        endif;
```

```
case RETRY:
        remote_request(IO_READ_HOME, received_srcid, my_id);
default
        error();
```

4.6.2 Data Cache Flush Operation Details

As with the I/O Read operation, the complexity for the Data Cache Flush operation resides in the memory controller. The necessary portions of the Data Cache Flush state machine description from Section 6.10 of the GSM logical specification are extracted below. Refer to Chapters 2 and 3 of the GSM specification to determine the size of data payloads for the FLUSH transaction. The GSM specification takes precedence in the case of any discrepancies between the corresponding portions of the GSM specification and this description.

4.6.2.1 Internal Request State Machine

This state machine handles requests to the remote globally shared memory space.

```
remote_request(FLUSH, mem_id, my_id, data);
```

4.6.2.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect.

```
switch (received_response)
case DONE:
        local_response(OK);
        free_entry();
case RETRY:
        remote_request(FLUSH, received_srcid, my_id, data);
default:
        error();
```

4.7 Byte Lane and Byte Enable Usage

PCI makes use of byte enables and allows combining and merging of transactions. This may have the result of write transactions with sparse valid bytes. In order to save on transaction overhead, RapidIO does not include byte enables. RapidIO does, however, support a set of byte encodings defined in Chapter 3 of the *RapidIO Part 1: Input/Output Logical Specification*. PCI to RapidIO operations may be issued with sparse bytes. Should a PCI write transaction with byte enables that do not match a RapidIO byte encoding be issued to a RapidIO to PCI bridge, that operation must be broken into multiple valid RapidIO operations.

4.8 Error Management

Errors that are detected on a PCI bus are signaled using side band signals. The treatment of these signals is left to the system designer and is outside of the PCI specifications. Likewise, this document does not recommend any practices for the delivery of error interrupts in the system.

Blank page

Chapter 5 Globally Shared Memory Devices

5.1 Introduction

Different processing elements have different requirements when participating in a RapidIO GSM environment. The GSM protocols and address collision tables are written from the point of view of a fully integrated processing element comprised of a local processor, a memory controller, and an I/O controller. Obviously, the complexity and implementation requirements for this assumed device are much greater than required for a typical design. This chapter assumes that the reader is familiar with the *RapidIO Part 5: Globally Shared Memory Logical Specification*.

Additionally, this chapter contains the 8/16 LP-LVDS and 1x/4x LP-Serial physical layer transaction to priority mappings to guarantee that a system maintains cache coherence and is deadlock free.

5.2 Processing Element Behavior

In Chapter 2 of the globally shared memory specification are a number of examples of possible processing elements:

- A processor-memory processing element
- A memory-only processing element
- A processor-only processing element
- An I/O processing element
- A switch processing element

Of all of these, only the switch processing element does not have to implement anything additional to exist in a GSM system or sub-system. All of the remaining processing element types are of interest, and all are likely to exist in some form in the marketplace. This chapter is intended to define the portions of the protocol necessary to implement each of these devices. Other processing elements are allowed by the globally shared memory specification, for example, a memory-I/O processing element. The portions of the protocol necessary to implement these devices are not addressed in this chapter.

The behaviors described in this chapter have been extracted directly from revision 1.1 of the globally shared memory specification, and may be out of date with respect to the latest revision of that document. The GSM specification takes precedence in

the case that there are discrepancies between it and this chapter.

5.2.1 Processor-Memory Processing Element

This processing element is very nearly the same as the assumed processing element used for the state machine description in Chapter 6, and requires nearly all of the described functionality. The following operation behavior is not changed from the Chapter 6 descriptions:

- Read
- Instruction read
- Read for ownership
- Data cache and instruction cache invalidate
- Castout
- TLB invalidate entry and TLB invalidate entry synchronize
- Data cache flush

This leaves the I/O Read operation. Since the processor-memory processing element does not contain an I/O device, this processing element will not generate the I/O read operation, but is required to respond to it. This removes the internal request state machine and portions of the response state machine, requiring the behavior described in Section 2.1.1 below. The only exception to this is the special case where there exists multiple coherence domains. It is possible that a processor in one coherence domain may wish to read data in another coherence domain and thus would require support of the I/O Read operation.

5.2.1.1 I/O Read Operations

This operation is used for I/O reads of globally shared memory space.

5.2.1.1.1 Response State Machine

This machine handles responses to requests made to the RapidIO interconnect made on behalf of a third party.

```
switch(remote_response)
case INTERVENTION:
        update_memory();
        remote_response(DONE_INTERVENTION, original_srcid, my_id);
        free_entry();
case NOT_OWNER,                                  // data comes from memory, mimic
                                                 // intervention
case RETRY:
        switch(directory_state)
        case LOCAL_MODIFIED,
        case LOCAL_SHARED:
                remote_response(DATA_ONLY, original_srcid, my_id,
                        data);
                remote_response(DONE_INTERVENTION, original_srcid,
                        my_id);
                free_entry();
        case REMOTE_MODIFIED:                    // spin or wait for castout
```

```
                        remote_request(IO_READ_OWNER, received_srcid, my_id,
                                my_id);
                default:
                        error();
default:
                error();
```

5.2.1.1.2 External Request State Machine

This machine handles requests from the system to the local memory or the local processor. This may require making further external requests.

```
if (address_collision)                                  // use collision tables in
                                                        // Chapter 7, "Address Collision Resolution
Tables"
elseif (IO_READ_HOME)                                   // remote request to our local memory
                assign_entry();
                switch (directory_state)
                case LOCAL_MODIFIED:
                        local_request(READ_LATEST);
                        remote_response(DONE, received_srcid, my_id, data);
                                                        // after push completes
                        free_entry();
                case LOCAL_SHARED:
                        remote_response(DONE, received_srcid, my_id, data);
                        free_entry();
                case REMOTE_MODIFIED:
                        remote_request(IO_READ_OWNER, mask_id, my_id, received_srcid);
                case SHARED:
                        remote_response(DONE, received_srcid, my_id, data);
                        free_entry();
                default:
                        error();
else                                                    // IO_READ_OWNER request to our caches
                assign_entry();
                local_request(READ_LATEST);             // spin until a valid response from
                                                        // the caches
                switch (local_response)
                case MODIFIED:                          // processor indicated a push;
                                                        // wait for it
                        if (received_srcid == received_secid)
                                                        // original requestor is also home
                                                        // module
                                remote_response(INTERVENTION, received_srcid, my_id,
                                        data);
                        else
                                remote_response(DATA_ONLY, received_secid, my_id,
                                        data);
                                remote_response(INTERVENTION, received_srcid, my_id);
                        endif;
                case INVALID:                           // must have cast it out during
                                                        // an address collision
                        remote_response(NOT_OWNER, received_srcid, my_id);
                default:
                        error();
                free_entry();
endif;
```

5.2.2 Memory-only Processing Element

This processing element is simpler than the assumed processing element used in Chapter 6, removing all of the internal request state machines and portions of all of the external request and response state machines. A memory-only processing element does not receive TLB invalidate entry or TLB invalidate synchronize operations. The required behavior for each operation is described below.

5.2.2.1 Read Operations

This operation is a coherent data cache read.

5.2.2.1.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(remote_response)
case INTERVENTION:
        update_memory();
        update_state(SHARED, original_srcid);
        remote_response(DONE_INTERVENTION, original_srcid, my_id);
        free_entry();
case NOT_OWNER,                                 // data comes from memory,
                                                // mimic intervention
case RETRY:
        switch(directory_state)
        case LOCAL_SHARED:
                update_state(SHARED, original_srcid);
                remote_response(DATA_ONLY, original_srcid,
                        my_id, data);
                remote_response(DONE_INTERVENTION, original_srcid,
                        my_id);
                free_entry();
        case LOCAL_MODIFIED:
                update_state(SHARED, original_srcid);
                remote_response(DATA_ONLY, original_srcid,
                        my_id, data);
                remote_response(DONE_INTERVENTION, original_srcid,
                        my_id);
                free_entry();
        case REMOTE_MODIFIED:                   // spin or wait for castout
                remote_request(READ_OWNER, received_srcid,
                        my_id, my_id);
        default:
                error();
default:
        error();
```

5.2.2.1.2 External Request State Machine

This state machine handles read requests from the system to the local memory. This may require making further external requests.

```
if (address_collision)                          // use collision tables in
                                                // Chapter 7, “Address Collision Resolution
Tables”
else                                            // READ_HOME
        assign_entry();
        switch (directory_state)
        case LOCAL_MODIFIED:
                local_request(READ);
```

```
                        update_state(SHARED, received_srcid);
                                                        // after possible push completes
                        remote_response(DONE, received_srcid, my_id, data);
                        free_entry();
                case LOCAL_SHARED,
                case SHARED:
                        update_state(SHARED, received_srcid);
                        remote_response(DONE, received_srcid, my_id, data);
                        free_entry();
                case REMOTE_MODIFIED:
                        if (mask_id ~= received_srcid)
                                                        // intervention case
                                remote_request(READ_OWNER, mask_id,
                                        my_id, received_srcid);
                        else
                                error();                // he already owned it;
                                                        // cache paradox (or I-fetch after d-
                                                        // store if not fixed elsewhere)
                        endif;
                default:
                        error();
endif;
```

5.2.2.2 Instruction Read Operations

This operation is a partially coherent instruction cache read.

5.2.2.2.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(remote_response)
case INTERVENTION:
        update_memory();
        update_state(SHARED, original_srcid);
        remote_response(DONE, original_srcid, my_id);
        free_entry();
case NOT_OWNER,                                         // data comes from memory,
                                                        // mimic intervention
case RETRY:
        switch(directory_state)
        case LOCAL_SHARED:
                update_state(SHARED, original_srcid);
                remote_response(DONE, original_srcid, my_id);
                free_entry();
        case LOCAL_MODIFIED:
                update_state(SHARED, original_srcid);
                remote_response(DONE, original_srcid, my_id);
                free_entry();
        case REMOTE_MODIFIED:                   // spin or wait for castout
                remote_request(READ_OWNER, received_srcid,
                        my_id, my_id);
        default:
                error();
default:
        error();
```

5.2.2.2.2 External Request State Machine

This state machine handles instruction read requests from the system to the local memory. This may require making further external requests.

```
if (address_collision)                                  // use collision tables in
                                                        // Chapter 7, “Address Collision Resolution
Tables”
else                                                    // IREAD_HOME
        assign_entry();
        switch (directory_state)
        case LOCAL_MODIFIED:
                local_request(READ);
                update_state(SHARED, received_srcid);
                                                        // after possible push completes
                remote_response(DONE, received_srcid, my_id, data);
                free_entry();
        case LOCAL_SHARED,
        case SHARED:
                update_state(SHARED, received_srcid);
                remote_response(DONE, received_srcid, my_id, data);
                free_entry();
        case REMOTE_MODIFIED:
                if (mask_id ~= received_srcid)
                                                        // intervention case
                        remote_request(READ_OWNER, mask_id,
                                my_id, received_srcid);
                else                                    // he already owned it in his
                                                        //data cache; cache paradox case
                        remote_request(READ_OWNER, mask_id, my_id, my_id);
                endif;
        default:
                error();
endif;
```

5.2.2.3 Read for Ownership Operations

This is the coherent cache store miss operation.

5.2.2.3.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(received_response)
case DONE:                                              // invalidates for shared
                                                        // directory states
        if ((mask ~= (my_id OR received_id)) == 0)
                                                        // this is the last DONE
                update_state(REMOTE_MODIFIED, original_srcid);
                remote_response(DONE, original_srcid, my_id, data);
                free_entry();
        else
                mask <= (mask ~= received_srcid);
                                                        // flip the responder’s shared bit
        endif;                                          // and wait for next DONE
case INTERVENTION:
                                                        // remote_modified case
        update_memory();                                // for possible coherence error
                                                        // recovery
        update_state(REMOTE_MODIFIED, original_id);
        remote_response(DONE_INTERVENTION, original_id, my_id);
        free_entry();
case NOT_OWNER:                                         // data comes from memory, mimic
```

```
                                                        // intervention
        switch(directory_state)
        case LOCAL_SHARED:
        case LOCAL_MODIFIED:
                update_state(REMOTE_MODIFIED, original_srcid);
                remote_response(DATA_ONLY, original_srcid, my_id,
                        data);
                remote_response(DONE, original_srcid, my_id);
                free_entry();
        case REMOTE_MODIFIED:
                remote_request(READ_TO_OWN_OWNER, received_srcid,
                        my_id, original_srcid);
        default:
                error();
case RETRY:
        switch (directory_state)
        case LOCAL_MODIFIED,
        case LOCAL_SHARED:
                update_state(REMOTE_MODIFIED, original_srcid);
                remote_response(DATA_ONLY, original_srcid, my_id,
                        data);
                remote_response(DONE, original_srcid, my_id);
                free_entry();
        case REMOTE_MODIFIED:                   // mask_id must match received_srcid
                                                // or error condition
                remote_request(READ_TO_OWN_OWNER, received_srcid,
                        my_id, my_id);
        case SHARED:
                remote_request(DKILL_SHARER, received_srcid, my_id,
                        my_id);
        default:
                error();
default:
        error();
```

5.2.2.3.2 External Request State Machine

This state machine handles requests from the interconnect to the local memory. This may require making further external requests.

```
if (address_collision)                          // use collision tables
                                                // in Chapter 7, “Address Collision Resolution
Tables”
else                                            // READ_TO_OWN_HOME
        assign_entry();
        switch (directory_state)
        case LOCAL_MODIFIED,
        case LOCAL_SHARED:
                local_request(READ_TO_OWN);
                remote_response(DONE, received_srcid, my_id, data);
                                                // after possible push
                update_state(REMOTE_MODIFIED, received_srcid);
                free_entry();
        case REMOTE_MODIFIED:
                if (mask_id ~= received_srcid)
                                                //intervention case
                        remote_request(READ_TO_OWN_OWNER, mask_id, my_id,
                                received_srcid);
                else
                        error();                // he already owned it!
                endif;
        case SHARED:
                local_request(READ_TO_OWN);
                if (mask == received_srcid)
                                                //requestor is only remote sharer
                        update_state(REMOTE_MODIFIED, received_srcid);
```

```
                                        remote_response(DONE, received_srcid, my_id, data);
                                                                // from memory
                                        free_entry();
                                else                            //there are other remote sharers
                                        remote_request(DKILL_SHARER, (mask ~= received_srcid),
                                                my_id, my_id);
                                endif;
                        default:
                                error();
endif;
```

5.2.2.4 Data Cache and Instruction Cache Invalidate Operations

This operation is used with coherent cache store-hit-on-shared, cache operations.

5.2.2.4.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(received_response)
case DONE:                                                      // invalidates for shared
                                                                // directory states
        if ((mask ~= (my_id OR received_id)) == 0)
                                                                // this is the last DONE
                update_state(REMOTE_MODIFIED, original_srcid);
                remote_response(DONE, original_srcid, my_id);
                free_entry();
        else
                mask <= (mask ~= received_srcid);
                                                                // flip the responder's shared bit
        endif;                                                  // and wait for next DONE
case RETRY:
        remote_request({DKILL_SHARER, IKILL_SHARER}, received_srcid,
                my_id);                                         // retry
default:
        error();
```

5.2.2.4.2 External Request State Machine

This state machine handles requests from the system to the local memory. This may require making further external requests.

```
if (address_collision)                                          // use collision tables in
                                                                //  Chapter 7, "Address Collision Resolution
Tables"
else                                                            // DKILL_HOME or IKILL_HOME
        assign_entry();
        if (DKILL_HOME)
                switch (directory_state)
                case LOCAL_MODIFIED,                            // cache paradoxes; DKILL is
                                                                // write-hit-on-shared
                case LOCAL_SHARED,
                case REMOTE_MODIFIED:
                        error();
                case SHARED:                                    // this is the right case, send
                                                                // invalidates to the sharing list
                        local_request(DKILL);
                        if (mask == received_srcid
                                                                // requestor is only remote sharer
                                update_state(REMOTE_MODIFIED, received_srcid);
                                remote_response(DONE, received_srcid, my_id);
                                free_entry();
                        else                                    // there are other remote sharers
                                remote_request(DKILL_SHARER,
```

```
                                                    (mask ~= received_srcid), my_id, NULL);
                        endif;
                default:
                        error();
        else                                        // IKILL goes to everyone except the
                                                    // requestor
                remote_request(IKILL_SHARER,
                        (mask <= (participant_list ~=
                        (received_srcid AND my_id), my_id);
endif;
```

5.2.2.5 Castout Operations

This operation is used to return ownership of a coherence granule to home memory, leaving it invalid in the cache.

5.2.2.5.1 External Request State Machine

This machine handles requests from the system to the local memory. This may require making further external requests.

```
assign_entry();
update_memory();
state_update(LOCAL_SHARED, my_id);          // may be LOCAL_MODIFIED if the
                                            // default is owned locally
remote_response(DONE, received_srcid, my_id);
free_entry();
```

5.2.2.6 Data Cache Flush Operations

This operation returns ownership of a coherence granule to home memory and performs a coherent write.

5.2.2.6.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(received_response)
case DONE:                                  // invalidates for shared directory
                                            // states
        if ((mask ~= (my_id OR received_id)) == 0)
                                            // this is the last DONE
                remote_response(DONE, original_srcid, my_id, my_id);
                if (received_data)
                                            // with original request or response
                        update_memory();
                endif;
                update_state(LOCAL_SHARED); // or LOCAL_MODIFIED
                free_entry();
        else
                mask <= (mask ~= received_srcid);
                                            // flip responder's shared bit
        endif;                              // and wait for next DONE
case NOT_OWNER:
        switch(directory_state)
        case LOCAL_SHARED,
        case LOCAL_MODIFIED:
                remote_response(DONE, original_srcid, my_id);
                if (received_data)
                                            // with original request
                        update_memory();
                endif;
```

```
                free_entry();
        case REMOTE_MODIFIED:
                remote_request(READ_TO_OWN_OWNER, received_srcid,
                        my_id, original_srcid);
        default:
                error();
case RETRY:
        switch(directory_state)
        case LOCAL_SHARED,
        case LOCAL_MODIFIED:
                remote_response(DONE, original_srcid, my_id);
                if (received_data)
                                                // with original request
                        update_memory();
                endif;
                free_entry();
        case REMOTE_MODIFIED:
                remote_request(READ_TO_OWN_OWNER, received_srcid,
                        my_id, original_srcid);
        case SHARED:
                remote_request(DKILL_SHARER, received_srcid, my_id);
        default:
                error();
default:
        error();
```

5.2.2.6.2 External Request State Machine

This state machine handles requests from the system to the local memory. This may require making further external requests.

```
if (address_collision)                          // use collision tables in
                                                //  Chapter 7, “Address Collision Resolution
Tables”
else                                            // FLUSH
        assign_entry();
        switch (directory_state)
        case LOCAL_MODIFIED,
        case LOCAL_SHARED:
                local_request(READ_TO_OWN);
                remote_response(DONE, received_srcid, my_id);
                                                // after snoop completes
                if (received_data)              // from request or local response
                        update_memory();
                endif;
                update_state(LOCAL_SHARED, my_id);
                                                // or LOCAL_MODIFIED
                free_entry();
        case REMOTE_MODIFIED:
                if (mask_id ~= received_srcid)  // owned elsewhere
                        remote_request(READ_TO_OWN_OWNER, mask_id, my_id,
                                received_srcid);
                else                            // requestor owned it; shouldn’t
                                                // generate a flush
                        error();
                endif;
        case SHARED:
                local_request(READ_TO_OWN);
                if (mask == received_srcid)     // requestor is only remote sharer
                        remote_response(DONE, received_srcid, my_id);
                                                // after snoop completes
                        if (received_data)      // from request or response
                                update_memory();
                        endif;
                        update_state(LOCAL_SHARED, my_id); // or LOCAL_MODIFIED
                        free_entry();
```

```
                    else                                    //there are other remote sharers
                        remote_request(DKILL_SHARER, (mask ~= received_srcid), my_id,
                            my_id);
                    endif;
            default:
                    error();
endif;
```

5.2.2.7 I/O Read Operations

This operation is used for I/O reads of globally shared memory space.

5.2.2.7.1 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of a third party.

```
switch(remote_response)
case INTERVENTION:
            update_memory();
            remote_response(DONE_INTERVENTION, original_srcid, my_id);
            free_entry();
case NOT_OWNER,                                     // data comes from memory, mimic
                                                    // intervention
case RETRY:
            switch(directory_state)
            case LOCAL_MODIFIED,
            case LOCAL_SHARED:
                    remote_response(DATA_ONLY, original_srcid, my_id,
                            data);
                    remote_response(DONE_INTERVENTION, original_srcid,
                            my_id);
                    free_entry();
            case REMOTE_MODIFIED:                   // spin or wait for castout
                    remote_request(IO_READ_OWNER, received_srcid, my_id,
                            my_id);
            default:
                    error();
default:
            error();
```

5.2.2.7.2 External Request State Machine

This machine handles requests from the system to the local memory. This may require making further external requests.

```
if (address_collision)                              // use collision tables in
                                                    // Chapter 7, “Address Collision Resolution
Tables”
else                                                // IO_READ_HOME
            assign_entry();
            switch (directory_state)
            case LOCAL_MODIFIED:
                    local_request(READ_LATEST);
                    remote_response(DONE, received_srcid, my_id, data);
                                                    // after push completes
                    free_entry();
            case LOCAL_SHARED:
                    remote_response(DONE, received_srcid, my_id, data);
                    free_entry();
            case REMOTE_MODIFIED:
                    remote_request(IO_READ_OWNER, mask_id, my_id, received_srcid);
            case SHARED:
                    remote_response(DONE, received_srcid, my_id, data);
                    free_entry();
```

```
                default:
                        error();
        endif;
```

5.2.3 Processor-only Processing Element

A processor-only processing element is much simpler than the assumed combined processing described in Chapter 6. Much of the internal request, response, and external request state machines are removed.

5.2.3.1 Read Operations

This operation is a coherent data cache read.

5.2.3.1.1 Internal Request State Machine

This state machine handles requests to remote memory from the local processor.

```
if (address_collision)                                  // this is due to an external request
                                                        // in progress or a cache
        local_response(RETRY);                          // index hazard from a previous request
else                                                    // remote - we've got to go
                                                        // to another module
        assign_entry();
        local_response(RETRY);                          // can't guarantee data before a
                                                        // snoop yet
        remote_request(READ_HOME, mem_id, my_id);
endif;
```

5.2.3.1.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch(remote_response)
case DONE:
        local_response(SHARED);                         // when processor re-requests
        return_data();
        free_entry();
case DONE_INTERVENTION:                                 // must be from third party
        set_received_done_message();
        if (received_data_only_message)
                free_entry();
        else
                                                        // wait for a DATA_ONLY
        endif;
case DATA_ONLY:                                         // this is due to an intervention, a
                                                        // DONE_INTERVENTIONshould come
                                                        // separately
        local_response(SHARED);
        set_received_data_only_message();
        if (received_done_message)
                return_data();
                free_entry();
        else
                return_data();                          // OK for weak ordering
        endif;
case RETRY:
        remote_request(READ_HOME, received_srcid, my_id);
default
        error();
```

5.2.3.1.3 External Request State Machine

This state machine handles read requests from the system to the local processor. This may require making further external requests.

```
if (address_collision)                                  // use collision tables in
                                                        // Chapter 7, "Address Collision Resolution
Tables"
else                                                    // READ_OWNER
        assign_entry();
        local_request(READ);                            // spin until a valid response
                                                        // from caches
        switch (local_response)
        case MODIFIED:                                  // processor indicated a push;
                                                        // wait for it
                cache_state(SHARED or INVALID);
                                                        // surrender ownership
                if (received_srcid == received_secid)
                                                        // original requestor is also home
                        remote_response(INTERVENTION, received_srcid,
                                my_id, data);
                else
                        remote_response(DATA_ONLY, received_secid,
                                my_id, data);
                        remote_response(INTERVENTION, received_srcid,
                                my_id, data);
                endif;
        case INVALID:                                   // must have cast it out
                remote_response(NOT_OWNER, received_srcid, my_id);
        default:
                error();
        free_entry();
endif;
```

5.2.3.2 Instruction Read Operations

This operation is a partially coherent instruction cache read.

5.2.3.2.1 Internal Request State Machine

This state machine handles requests to remote memory from the local processor.

```
if (address_collision)                                  // this is due to an external
                                                        // request in progress or a cache
        local_response(RETRY);                          // index hazard from a previous request
else                                                    // remote - we've got to go
                                                        // to another module
        assign_entry();
        local_response(RETRY);
                                                        // can't guarantee data before a
                                                        // snoop yet
        remote_request(IREAD_HOME, mem_id, my_id);
endif;
```

5.2.3.2.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch(remote_response)
case DONE:
        local_response(SHARED);                         // when processor re-requests
        return_data();
        free_entry();
```

```
case DONE_INTERVENTION:                          // must be from third party
        set_received_done_message();
        if (received_data_only_message)
                free_entry();
        else
                                                 // wait for a DATA_ONLY
        endif;
case DATA_ONLY:                                  // this is due to an intervention; a
                                                 // DONE_INTERVENTION should come
                                                 // separately
        local_response(SHARED);
        set_received_data_only_message();
        if (received_done_message)
                return_data();
                free_entry();
        else
                return_data();                   // OK for weak ordering
        endif;
case RETRY:
        remote_request(IREAD_HOME, received_srcid, my_id);
default
        error();
```

5.2.3.2.3 External Request State Machine

This state machine handles instruction read requests from the system to the local processor.

```
if (address_collision)                           // use collision tables in
                                                 //  Chapter 7, “Address Collision Resolution
Tables”
else                                             // READ_OWNER request to our caches
        assign_entry();
        local_request(READ);                     // spin until a valid response
                                                 // from caches
        switch (local_response)
        case MODIFIED:                           // processor indicated a push;
                                                 // wait for it
                cache_state(SHARED or INVALID);
                                                 // surrender ownership
                if (received_srcid == received_secid)
                                                 // original requestor is also home
                        remote_response(INTERVENTION, received_srcid,
                                my_id, data);
                else
                        remote_response(DATA_ONLY, received_secid,
                                my_id, data);
                        remote_response(INTERVENTION, received_srcid,
                                my_id, data);
                endif;
        case INVALID:                            // must have cast it out
                remote_response(NOT_OWNER, received_srcid, my_id);
        default:
                error();
        free_entry();
endif;
```

5.2.3.3 Read for Ownership Operations

This is the coherent cache store miss operation.

5.2.3.3.1 Internal Request State Machine

This state machine handles requests to remote memory from the local processor.

```
if (address_collision)                                  // this is due to an external request
                                                        // in progress or a cache index
        local_response(RETRY);                          // hazard from a previous request
else                                                    // remote - we've got to go to another
                                                        // module
        assign_entry();
        local_response(RETRY);
        remote_request(READ_TO_OWN_HOME, mem_id, my_id);
endif;
```

5.2.3.3.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch (received_response)
case DONE:
        local_response(EXCLUSIVE);
        return_data();
        free_entry();
case DONE_INTERVENTION:
        set_received_done_message();
        if (received_data_message)
                free_entry();
        else
                                                        // wait for DATA_ONLY
        endif;
case DATA_ONLY:
        set_received_data_message();
        local_response(EXCLUSIVE);
        if (received_done_message)
                return_data();
                free_entry();
        else
                return_data();                          // OK for weak ordering
        endif;                                          // and wait for a DONE
case RETRY:                                             // lost at remote memory so retry
        remote_request(READ_TO_OWN_HOME, mem_id, my_id);
default:
        error();
```

5.2.3.3.3 External Request State Machine

This state machine handles requests from the interconnect to the local processor.

```
if (address_collision)                                  // use collision tables
                                                        // in Chapter 7, "Address Collision Resolution
Tables"
elseif(READ_TO_OWN_OWNER                                // request to our caches
        assign_entry();
        local_request(READ_TO_OWN);                     // spin until a valid response from
                                                        // the caches
        switch (local_response)
        case MODIFIED:                                  // processor indicated a push
                cache_state(INVALID);
                                                        // surrender ownership
                if (received_srcid == received_secid)
```

```
                                                    //the original request is from the home
                        remote_response(INTERVENTION, received_srcid, my_id,
                                data);
                    else                            // the original request is from a
                                                    // third party
                        remote_response(DATA_ONLY, received_secid, my_id,
                                data);
                        remote_response(INTERVENTION, received_srcid, my_id,
                                data);
                    endif;
                    free_entry();
            case INVALID:                           // castout address collision
                    remote_response(NOT_OWNER, received_srcid, my_id);
            default:
                    error();
else                                                // DKILL_SHARER request to our caches
            assign_entry();
            local_request(READ_TO_OWN);
                                                    // spin until a valid response from the
                                                    // caches
            switch (local_response)
            case SHARED,
            case INVALID:                           // invalidating for shared cases
                    cache_state(INVALID);           // surrender copy
                    remote_response(DONE, received_srcid, my_id);
                    free_entry();
            default:
                    error();

endif;
```

5.2.3.4 Data Cache and Instruction Cache Invalidate Operations

This operation is used with coherent cache store-hit-on-shared, cache operations.

5.2.3.4.1 Internal Request State Machine

This state machine handles requests to remote memory from the local processor.

```
if (address_collision)                              // this is due to an external request in
                                                    // progress or a cache index
            local_response(RETRY);                  // hazard from a previous request
else                                                // remote - we've got to go to another
                                                    // module
            assign_entry();
            local_response(RETRY);
            remote_request({DKILL_HOME, IKILL_HOME}, mem_id, my_id);
endif;
```

5.2.3.4.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch (received_response)
case DONE:
            local_response(EXCLUSIVE);
            free_entry();
case RETRY:
            remote_request({DKILL_HOME, IKILL_HOME}, received_srcid,
                    my_id);                 // retry the transaction
default:
            error();
```

5.2.3.4.3 External Request State Machine

This state machine handles requests from the system to the local processor.

```
if (address_collision)                          // use collision tables in
                                                // Chapter 7, “Address Collision Resolution Tables”
else                                            // DKILL_SHARER or IKILL_SHARER request to our
caches
        assign_entry();
        local_request({READ_TO_OWN, IKILL});
                                                        // spin until a valid response from the
                                                        // caches
        switch (local_response)
        case SHARED,
        case INVALID:                           // invalidating for shared cases
                cache_state(INVALID);           // surrender copy
                remote_response(DONE, received_srcid, my_id);
                free_entry();
        default:
                error();
endif;
```

5.2.3.5 Castout Operations

This operation is used to return ownership of a coherence granule to home memory, leaving it invalid in the cache. A processor-only processing element is never the target of a castout operation.

5.2.3.5.1 Internal Request State Machine

A castout may require local activity to flush all caches in the hierarchy and break possible reservations.

```
assign_entry();
local_response(OK);
remote_request(CASTOUT, mem_id, my_id, data);
```

5.2.3.5.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch (received_response)
case DONE:
        free_entry();
default:
        error();
```

5.2.3.6 TLB Invalidate Entry, TLB Invalidate Entry Synchronize Operations

These operations are used for software coherence management of the TLBs.

5.2.3.6.1 Internal Request State Machine

The TLBIE and TLBSYNC transactions are always sent to all domain participants except the sender and are always to the processor, not home memory.

```
assign_entry();
remote_request({TLBIE, TLBSYNC}, participant_id, my_id);
endif;
```

5.2.3.6.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor. The responses are always from a coherence participant, not a home memory.

```
switch (received_response)
case DONE:
        if ((mask ~= (my_id OR received_id)) == 0)
                                                    // this is the last DONE
                free_entry();
        else
                mask <= (mask ~= received_srcid);
                                                    // flip the responder's participant
                                                    // bit and wait for next DONE
        endif;
case RETRY:
        remote_request({TLBIE, TLBSYNC}, received_srcid, my_id, my_id);
default
        error();
```

5.2.3.6.3 External Request State Machine

This state machine handles requests from the system to the local memory or the local processor. The requests are always to the local caching hierarchy.

```
assign_entry();
local_request({TLBIE, TLBSYNC});                    // spin until a valid response
                                                    // from the caches
remote_response(DONE, received_srcid, my_id);
free_entry();
```

5.2.3.7 Data Cache Flush Operations

This operation returns ownership of a coherence granule to home memory and performs a coherent write.

5.2.3.7.1 Internal Request State Machine

This state machine handles requests to remote memory from the local processor.

```
if (address_collision)                              // this is due to an external
                                                    // request in progress or a cache index
        local_response(RETRY);                      // hazard from a previous request
else                                                // remote - we've got to go to
                                                    // another module
        assign_entry();
        remote_request(FLUSH, mem_id, my_id, data);
```

```
                                                    // data is optional
endif;
```

5.2.3.7.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local processor.

```
switch (received_response)
case DONE:
        local_response(OK);
        free_entry();
case RETRY:
        remote_request(FLUSH, received_srcid, my_id, data);
                                                    // data is optional
default:
        error();
```

5.2.3.7.3 External Request State Machine

This state machine handles requests from the system to the local processor.

```
if (address_collision)                              // use collision tables in
                                                    //  Chapter 7, “Address Collision Resolution
Tables”
elseif (READ_TO_OWN_OWNER)                          // remote request to our caches
        assign_entry();
        local_request(READ_TO_OWN);                 // spin until a valid response
                                                    // from the caches
        switch (local_response)
        case MODIFIED:                              // processor indicated a push,
                                                    // wait for it
                cache_state(INVALID);               // surrender ownership
                remote_response(DONE, received_srcid, my_id, data);
        case INVALID:
                                                    // must have cast it out during an
                                                    // address collision
                remote_response(NOT_OWNER, received_srcid, my_id);
        default:
                error();
        free_entry();
else                                                // DKILL_SHARER remote request
                                                    // to our caches
        assign_entry();
        local_request(DKILL);                       // spin until a valid response from
                                                    // the caches
        switch (local_response)
        case MODIFIED:                              // cache paradox
                remote_response(ERROR, received_srcid, my_id);
        case INVALID:
                remote_response(DONE, received_srcid, my_id);
        default:
                error();
        free_entry();
endif;
```

5.2.3.8 I/O Read Operations

This operation is used for I/O reads of globally shared memory space. A processor-only processing element never initiates an I/O read operation.

5.2.3.8.1 External Request State Machine

This machine handles requests from the system to the local memory or the local

processor. This may require making further external requests.

```
if (address_collision)                                  // use collision tables in
                                                        //  Chapter 7, “Address Collision Resolution
Tables”
else                                                    // IO_READ_OWNER request to our caches
        assign_entry();
        local_request(READ_LATEST);                     // spin until a valid response from
                                                        // the caches
        switch (local_response)
        case MODIFIED:                                  // processor indicated a push;
                                                        // wait for it
                if (received_srcid == received_secid)
                                                        // original requestor is also home
                                                        // module
                        remote_response(INTERVENTION, received_srcid, my_id,
                                data);
                else
                        remote_response(DATA_ONLY, received_secid, my_id,
                                data);
                        remote_response(INTERVENTION, received_srcid, my_id);
                endif;
        case INVALID:                                   // must have cast it out during
                                                        // an address collision
                remote_response(NOT_OWNER, received_srcid, my_id);
        default:
                error();
        free_entry();
endif;
```

5.2.4 I/O Processing Element

The simplest GSM processing element is an I/O device. A RapidIO I/O processing element does not actually participate in the globally shared memory environment (it is defined as not in the coherence domain), but is able to read and write data into the GSM address space through special I/O operations that provide for this behavior. These operations are the I/O Read and Data Cache Flush operations. Other than the ability to read and write into the GSM address space, an I/O device has no other operational requirements. Since the I/O processing element is not part of the coherence domain, it is never the target of a coherence transaction and thus does not have to implement any of the related behavior, including the address collision tables.

Requirements for a specific I/O processing element, a RapidIO to PCI/PCI-X bridge, is discussed in Chapter 4, “PCI Considerations,” on page 4-27.

5.2.4.1 I/O Read Operations

This operation is used for I/O reads of globally shared memory space.

5.2.4.1.1 Internal Request State Machine

This state machine handles requests to remote memory.

```
remote_request(IO_READ_HOME, mem_id, my_id);
```

5.2.4.1.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect.

```
switch(remote_response)
case DONE:
        return_data();
        free_entry();
case DONE_INTERVENTION:                 // must be from third party
        set_received_done_message();
        if (received_data_only_message)
                free_entry();
        else
                // wait for a DATA_ONLY
        endif;
case DATA_ONLY:                         // this is due to an intervention, a
                                        // DONE_INTERVENTION should come
                                        // separately
        set_received_data_only_message();
        if (received_done_message)
                return_data();
                free_entry();
        else
                return_data();          // OK for weak ordering
        endif;
case RETRY:
        remote_request(IO_READ_HOME, received_srcid, my_id);
default
        error();
```

5.2.4.2 Data Cache Flush Operations

This operation returns ownership of a coherence granule to home memory and performs a coherent write.

5.2.4.2.1 Internal Request State Machine

This state machine handles requests to remote memory.

```
remote_request(FLUSH, mem_id, my_id, data);
```

5.2.4.2.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect.

```
switch (received_response)
case DONE:
        local_response(OK);
        free_entry();
case RETRY:
        remote_request(FLUSH, received_srcid, my_id, data);
default:
        error();
```

5.2.5 Switch Processing Element

A switch processing element is required to be able to route all defined packets. Since it is not necessary for a switch to analyze a packet in order to determine how it should be treated outside of examining the priority and the destination device ID, a switch processing element does not have any additional requirements to be used in a globally shared memory environment.

5.3 Transaction to Priority Mappings

The Globally Shared Memory model does not have the concept of an end point to end point request transaction flow like the I/O programming model. Instead, all transaction ordering is managed by the load-store units of the processors participating in the globally shared memory protocol. The GSM logical specification behaviors assume an unordered and resource unconstrained communication fabric. The ordered fabric of the 8/16 LP-LVDS and the 1x/4x LP-Serial physical layers requires the proper transaction to priority mappings to mimic the effect of an unordered fabric to suit the GSM model. These mappings leverage the physical layer ordering and deadlock avoidance rules that are required by the I/O Logical layer. In addition, it is assumed that the latency-critical GSM operations are of necessity higher priority than non-coherent I/O traffic, therefore I/O operations are recommended to be assigned to the lowest system priority flow.

Table 5-1 shows the GSM transaction to priority mappings.

Table 5-1. Transaction to Priority Mapping

Request transaction	Request Packet Priority	Response Packet Priority
READ_TO_OWN_HOME	1	2 or 3
READ_HOME	1	2 or 3
IO_READ_HOME	1	2 or 3
IREAD_HOME	1	2 or 3
DKILL_HOME	1	2 or 3
IKILL_HOME	1	2 or 3
FLUSH (without data)	1	2 or 3
FLUSH (with data)	1	2 or 3
TLBIE	1	2 or 3
TLBSYNC	1	2 or 3
READ_OWNER	2	3
READ_TO_OWN_OWNER	2	3
IO_READ_OWNER	2	3
DKILL_SHARER	2	3
IKILL_SHARER	2	3
CASTOUT	2	3

Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book.

A

Agent. A processing element that provides services to a processor.

B

Bridge. A processing element that connects one computer bus to another, allowing a processing element on one bus to access an processing element on the other.

C

Cache. High-speed memory containing recently accessed data and/or instructions (subset of main memory) associated with a processor.

Cache coherence. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache. In other words, a write operation to an address in the system is visible to all other caches in the system.

Capability registers (CARs). A set of read-only registers that allows a processing element to determine another processing element's capabilities.

Command and status registers (CSRs). A set of registers that allows a processing element to control and determine the status of another processing element's internal hardware.

Control symbol. A quantum of information transmitted between two linked devices to manage packet flow between the devices.

D

Deadlock. A situation in which two processing elements that are sharing resources prevent each other from accessing the resources, resulting in a halt of system operation.

Delayed transaction. The process of the target of a transaction capturing the transaction and completing it after responding to the source with a retry.

Destination. The termination point of a packet on the RapidIO interconnect, also referred to as a target.

Device. A generic participant on the RapidIO interconnect that sends or receives RapidIO transactions, also called a processing element.

Device ID. The identifier of an end point processing element connected to the RapidIO interconnect.

Double-word. An eight byte quantity, aligned on eight byte boundaries.

E

End point. A processing element which is the source or destination of transactions through a RapidIO fabric.

End point device. A processing element which contains end point functionality.

F

Field or Field name. A sub-unit of a register, where bits in the register are named and defined.

G

Globally shared memory (GSM). Cache coherent system memory that can be shared between multiple processors in a system.

H

Host. A processing element responsible for exploring and initializing all or a portion of a RapidIO based system.

I

Initiator. The origin of a packet on the RapidIO interconnect, also referred to as a source.

I/O. Input-output.

L

Local memory. Memory associated with the processing element in question.

LVDS. Low voltage differential signaling.

M

Mailbox. Dedicated hardware that receives messages.

Message passing. An application programming model that allows processing elements to communicate via messages to mailboxes instead of via GSM. Message senders do not write to a memory address in the target.

N

Non-coherent. A transaction that does not participate in any system globally shared memory cache coherence mechanism.

O

Operation. A set of transactions between end point devices in a RapidIO system (requests and associated responses) such as a read or a write.

P

Packet. A set of information transmitted between devices in a RapidIO system.

Peripheral component interface (PCI). A bus commonly used for connecting I/O devices in a system.

Port-write. An address-less maintenance write operation.

Priority. The relative importance of a transaction or packet; in most systems a higher priority transaction or packet will be serviced or transmitted before one of lower priority.

Processing Element (**PE**). A generic participant on the RapidIO interconnect that sends or receives RapidIO transactions, also called a device.

Processor. The logic circuitry that responds to and processes the basic instructions that drive a computer.

R

Remote memory. Memory associated with a processing element other than the processing element in question.

ROM. Read-only memory.

S

Sender. The RapidIO interface output port on a processing element.

Source. The origin of a packet on the RapidIO interconnect, also referred to as an initiator.

Switch. A multiple port processing element that directs a packet received on one of its input ports to one of its output ports.

Symbol. A 16-bit quantity.

T

Target. The termination point of a packet on the RapidIO interconnect, also referred to as a destination.

Transaction. A specific request or response packet transmitted between end point devices in a RapidIO system.

Transaction request flow. A sequence of transactions between two processing elements that have a required completion order at the destination processing element. There are no ordering requirements between transaction request flows.

Blank page