RapidIO™ Interconnect Specification Part 5: Globally Shared Memory Logical Specification

Rev. 2.2, 06/2011

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	Incorporate comment review changes	03/08/2001
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 Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005
2.0	No technical changes	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	Technical changes: errata showing: 10-08-00001.005, Consolidated Comments on 11-01-00000.000	05/05/2011
2.2	Removed confidentiality markings for public release	06/06/2011

NO WARRANTY.THE RAPIDIO TRADE ASSOCIATION PUBLISHES THE SPECIFICATION “AS IS”. THE RAPIDIO TRADE ASSOCIATION MAKES NO WARRANTY, REPRESENTATION OR COVENANT, EXPRESS OR IMPLIED, OF ANY KIND CONCERNING THE SPECIFICATION, INCLUDING, WITHOUT LIMITATION, NO WARRANTY OF NON INFRINGEMENT, NO WARRANTY OF MERCHANTABILITY AND NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE. USER AGREES TO ASSUME ALL OF THE RISKS ASSOCIATED WITH ANY USE WHATSOEVER OF THE SPECIFICATION. WITHOUT LIMITING THE GENERALITY OF THE FOREGOING, USER IS RESPONSIBLE FOR SECURING ANY INTELLECTUAL PROPERTY LICENSES OR RIGHTS WHICH MAY BE NECESSARY TO IMPLEMENT OR BUILD PRODUCTS COMPLYING WITH OR MAKING ANY OTHER SUCH USE OF THE SPECIFICATION.

DISCLAIMER OF LIABILITY. THE RAPIDIO TRADE ASSOCIATION SHALL NOT BE LIABLE OR RESPONSIBLE FOR ACTUAL, INDIRECT, SPECIAL, INCIDENTAL, EXEMPLARY OR CONSEQUENTIAL DAMAGES (INCLUDING, WITHOUT LIMITATION, LOST PROFITS) RESULTING FROM USE OR INABILITY TO USE THE SPECIFICATION, ARISING FROM ANY CAUSE OF ACTION WHATSOEVER, INCLUDING, WHETHER IN CONTRACT, WARRANTY, STRICT LIABILITY, OR NEGLIGENCE, EVEN IF THE RAPIDIO TRADE ASSOCIATION HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGES.

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.

RapidIO Trade Association

Table of Contents

Chapter 1 Overview

Chapter 2 System Models

Chapter 3 Operation Descriptions

Table of Contents

Chapter 4 Packet Format Descriptions

Chapter 5 Globally Shared Memory Registers

Chapter 6 Communication Protocols

Table of Contents

Chapter 7 Address Collision Resolution Tables

Table of Contents

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 5: Globally Shared Memory Logical Specification*, including a description of the relationship between this specification and the other specifications of the RapidIO interconnect.

1.2 Overview

Although RapidIO is targeted toward the message passing programming model, it supports a globally shared distributed memory (GSM) model as defined by this specification. The globally shared memory programming model is the preferred programming model for modern general-purpose multiprocessing computer systems, which requires cache coherency support in hardware. This addition of GSM enables both distributed I/O processing and general purpose multiprocessing to co-exist under the same protocol.

The *RapidIO Part 5: Globally Shared Memory Logical Specification* is one of the RapidIO logical layer specifications that define the interconnect's overall protocol and packet formats. This layer contains the information necessary for end points to process a transaction. Other RapidIO logical layer specifications include *RapidIO Part 1: Input/Output Logical Specification* and *RapidIO Part 2: Message Passing Logical Specification.*

The logical specifications do not imply a specific transport or physical interface, therefore they are specified in a bit stream format. Necessary bits are added to the logical encodings for the transport and physical layers lower in the specification hierarchy.

RapidIO is a definition of a system interconnect. System concepts such as processor programming models, memory coherency models and caching are beyond the scope of the RapidIO architecture. The support of memory coherency models, through caches, memory directories (or equivalent, to hold state and speed up remote memory access) is the responsibility of the end points (processors, memory, and possibly I/O devices), using RapidIO operations. RapidIO provides the operations to construct a wide variety of systems, based on programming models that range from strong consistency through total store ordering to weak ordering. Inter-operability between end points supporting different coherency/caching/directory models is not guaranteed. However, groups of end-points with conforming models can be linked to others conforming to different models on the same RapidIO fabric. These different groups can communicate through RapidIO messaging or I/O operations. Any reference to these areas within the RapidIO architecture specification are for illustration only.

The *RapidIO Interconnect Globally Shared Memory Logical Specification* assumes that the reader is familiar with the concepts and terminology of cache coherent systems in general and with CC-NUMA systems in specific. Further information on shared memory concepts can be found in:

Daniel E. Lenoski and Wolf-Dietrich Weber, “Scalable Shared-Memory Multiprocessing”, Morgan Kaufmann, 1995.

and

David Culler, Jaswinder Pal Singh, and Anoop Gupta: “Parallel Computer Architecture: A Hardware/Software Approach”, Morgan Kaufmann, 1998

1.2.1 Memory System

Under the globally shared distributed memory programming model, memory may be physically located in different places in the machine yet may be shared amongst different processing elements. Typically, mainstream system architectures have addressed shared memory using transaction broadcasts sometimes known as bus-based snoopy protocols. These are usually implemented through a centralized memory controller for which all devices have equal or uniform access. Figure 1-1 shows a typical bus-based shared memory system.

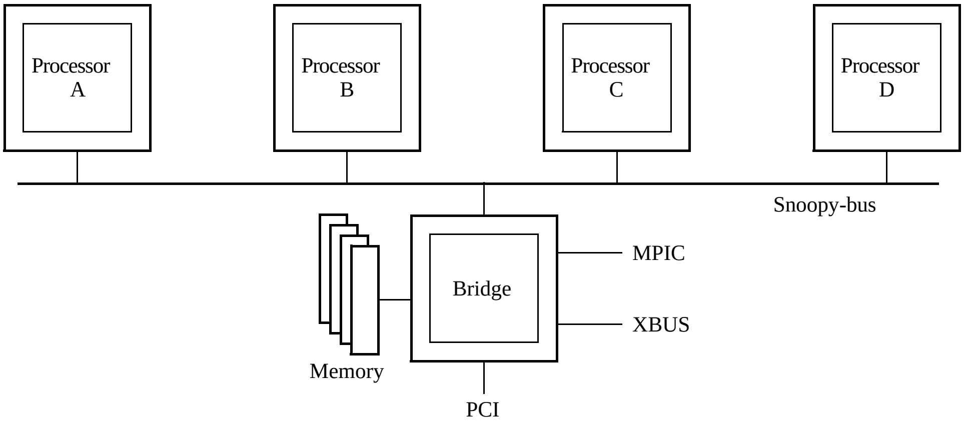


Figure 1-1. A Snoopy Bus-Based System

Super computers, massively parallel, and clustered machines that have distributed memory systems must use a different technique from broadcasting for maintaining memory coherency. Because a broadcast snoopy protocol in these machines is not efficient given the number of devices that must participate and the latency and transaction overhead involved, coherency mechanisms such as memory directories or distributed linked lists are required to keep track of where the most current copy of data resides. These schemes are often referred to as cache coherent non-uniform memory access (CC-NUMA) protocols. A typical distributed memory system architecture is shown in Figure 1-2.

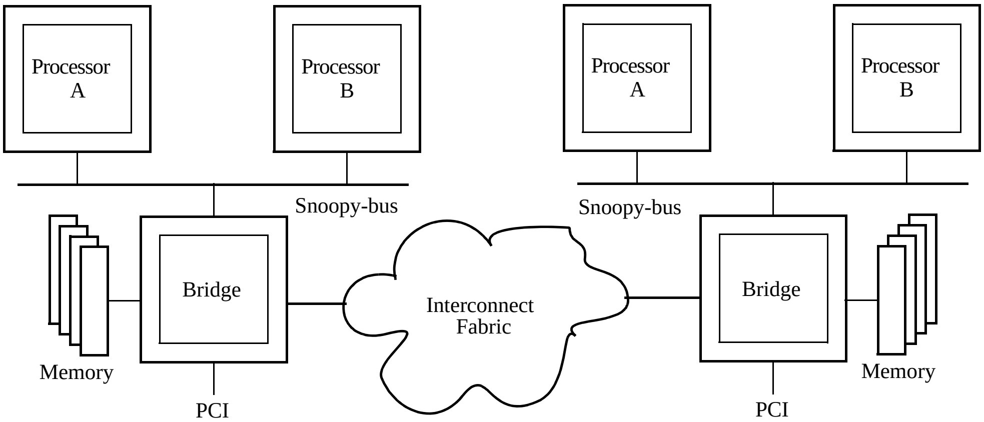


Figure 1-2. A Distributed Memory System

For RapidIO, a relatively simple directory-based coherency scheme is chosen. For this method each memory controller is responsible for tracking where the most current copy of each data element resides in the system. RapidIO furnishes a variety of ISA specific cache control and operating system support operations such as block flushes and TLB synchronization mechanisms.

To reduce the directory overhead required, the architecture is optimized around small clusters of 16 processors known as coherency domains. With the concept of domains, it is possible for multiple coherence groupings to coexist in the interconnect as tightly coupled processing clusters.

1.3 Features of the Globally Shared Memory Specification

The following are features of the RapidIO GSM specification designed to satisfy the needs of various applications and systems:

1.3.1 Functional Features

- A cache coherent non-uniform memory access (CC-NUMA) system architecture is supported to provide a globally shared memory model because physics is forcing component interfaces in many high-speed designs to be point-to-point instead of traditional bus-based.
- The size of processor memory requests are either in the cache coherence granularity, or smaller. The coherence granule size may be different for different processor families or implementations.

- Instruction sets in RapidIO support a variety of cache control and other operations such as block flushes. These functions are supported to run legacy applications and operating systems.

1.3.2 Physical Features

- RapidIO packet definition is independent of the width of the physical interface to other devices on the interconnect fabric.
- The protocols and packet formats are independent of the physical interconnect topology. The protocols work whether the physical fabric is a point-to-point ring, a bus, a switched multi-dimensional network, a duplex serial connection, and so forth.
- RapidIO is not dependent on the bandwidth or latency of the physical fabric.
- The protocols handle out-of-order packet transmission and reception.
- Certain devices have bandwidth and latency requirements for proper operation. RapidIO does not preclude an implementation from imposing these constraints within the system.

1.3.3 Performance Features

- Packet headers must be as small as possible to minimize the control overhead and be organized for fast, efficient assembly and disassembly.
- 48- and 64-bit addresses are required in the future, and must be supported initially.
- An interventionist (non-memory owner, direct-to-requestor data transfer, analogous to a cache-to-cache transfer) protocol saves a large amount of latency for memory accesses that cause another processing element to provide the requested data.
- Multiple transactions must be allowed concurrently in the system, otherwise a majority of the potential system throughput is wasted.

1.4 Contents

Following are the contents of the *RapidIO Interconnect Globally Shared Memory Logical Specification:*

- Chapter 1, “Overview,” describes the set of operations and transactions supported by the RapidIO globally shared memory protocols.
- Chapter 2, “System Models,” introduces some possible devices that could participate in a RapidIO GSM system environment. The chapter explains the memory directory-based mechanism that tracks memory accesses and maintains cache coherence. Transaction ordering and deadlock prevention are also covered.

- Chapter 3, “Operation Descriptions,” describes the set of operations and transactions supported by the RapidIO globally-shared memory (GSM) protocols.
- Chapter 4, “Packet Format Descriptions,” contains the packet format definitions for the GSM specification. The two basic types, request and response packets, with their sub-types and fields are defined. The chapter explains how memory read latency is handled by RapidIO.
- Chapter 5, “Globally Shared Memory Registers,” describes the visible register set that allows an external processing element to determine the globally shared memory capabilities, configuration, and status of a processing element using this logical specification. Only registers or register bits specific to the GSM logical specification are explained. Refer to the other RapidIO logical, transport, and physical specifications of interest to determine a complete list of registers and bit definitions.
- Chapter 6, “Communication Protocols,” contains the communications protocol definitions for this GSM specification.
- Chapter 7, “Address Collision Resolution Tables,” explains the actions necessary under the RapidIO GSM model to resolve address collisions.

1.5 Terminology

Refer to the Glossary at the back of this document.

1.6 Conventions

||	Concatenation, used to indicate that two fields are physically associated as consecutive bits
ACTIVE_HIGH	Names of active high signals are shown in uppercase text with no overbar. Active-high signals are asserted when high and not asserted when low.
$\overline{\text{ACTIVE\_LOW}}$	Names of active low signals are shown in uppercase text with an overbar. Active low signals are asserted when low and not asserted when high.
italics	Book titles in text are set in italics.
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets.
TRANSACTION	Transaction types are expressed in all caps.
operation	Device operation types are expressed in plain text.
n	A decimal value.
[*n*-*m*]	Used to express a numerical range from *n* to *m*.

0b*nn* A binary value, the number of bits is determined by the number of digits.

0x*nn* A hexadecimal value, the number of bits is determined by the number of digits or from the surrounding context; for example, 0x*nn* may be a 5, 6, 7, or 8 bit value.

x This value is a don’t care

Chapter 2 System Models

2.1 Introduction

This overview introduces some possible devices in a RapidIO system.

2.2 Processing Element Models

Figure 2-1 describes a possible RapidIO-based computing system. The processing element is a computer device such as a processor attached to a local memory and also attached to a RapidIO system interconnect. The bridge part of the system provides I/O subsystem services such as high-speed PCI interfaces and gigabit ethernet ports, interrupt control, and other system support functions. Multiple processing elements require cache coherence support in the RapidIO protocol to preserve the traditional globally shared memory programming model (discussed in Section 2.3.1, “Globally Shared Memory System Model”).

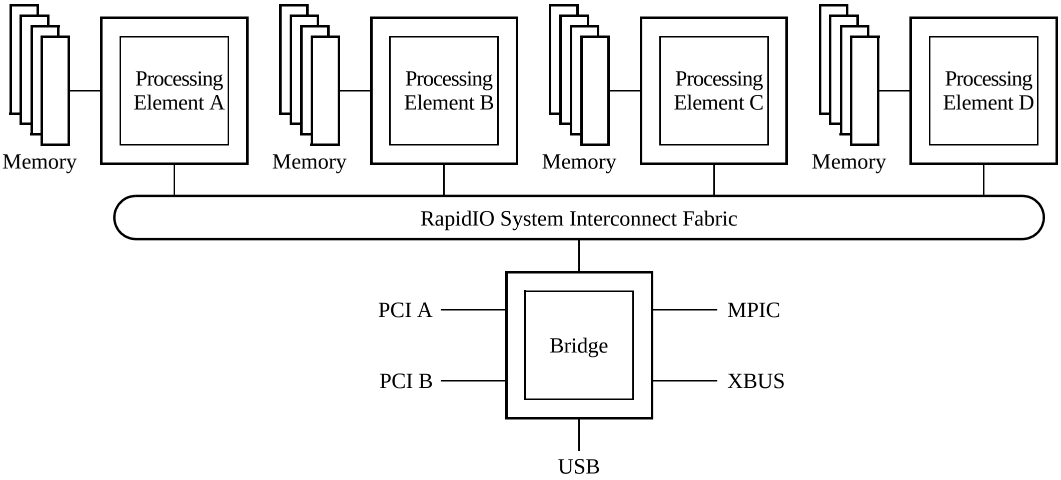


Figure 2-1. A Possible RapidIO-Based Computing System

A processing element containing a processor typically has associated with it a caching hierarchy to improve system performance. The RapidIO protocol supports a set of operations sufficient to fulfill the requirements of a processor with a caching hierarchy and associated support logic such as a processing element.

RapidIO is defined so that many types of devices can be designed for specific applications and connected to the system interconnect. These devices may participate in the cache coherency protocol, act as a DMA device, utilize the message passing facilities to communicate with other devices on the interconnect, and so forth. A bridge could be designed, for example, to use the message passing facility to pass ATM packets to and from a processing element for route processing. The following sections describe several possible processing elements.

2.2.1 Processor-Memory Processing Element Model

Figure 2-2 shows an example of a processing element consisting of a processor connected to an agent device. The agent carries out several services on behalf of the processor. Most importantly, it provides access to a local memory that has much lower latency than memory that is local to another processing element (remote memory accesses). It also provides an interface to the RapidIO interconnect to service those remote memory accesses.

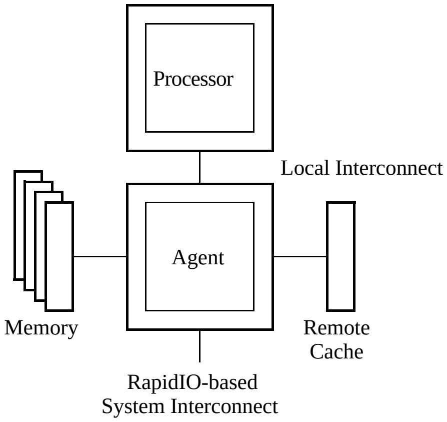


Figure 2-2. Processor-Memory Processing Element Example

In support of the remote accesses, the agent maintains a cache of remote accesses that includes all remote data currently residing in and owned by the local processor. This cache may be either external or internal to the agent device.

Agent caching is necessary due to the construction of the RapidIO cache coherence protocol combined with the cache hierarchy behavior in modern processors. Many modern processors have multiple level non-inclusive caching structures that are maintained independently. This implies that when a coherence granule is cast out of the processor, it may or may not be returning ownership of the granule to the memory system. The RapidIO protocol requires that ownership of a coherence granule be guaranteed to be returned to the system on demand and without ambiguous cache state changes as with the castout behavior. The remote cache can guarantee that a coherence granule requested by the system is owned locally and can be returned to the home memory (the physical memory containing the coherence granule) on demand. A processing element that is fully integrated would also need to support this behavior.

2.2.2 Integrated Processor-Memory Processing Element Model

Another form of a processor-memory processing element is a fully integrated component that is designed specifically to connect to a RapidIO interconnect system as shown in Figure 2-3. This type of device integrates a memory system and other support logic with a processor on the same piece of silicon or within the same package. Because such a device is designed specifically for RapidIO, a remote cache is not required because the proper support can be designed into the processor and its associated logic rather than requiring an agent to compensate for a stand alone processor’s behavior.

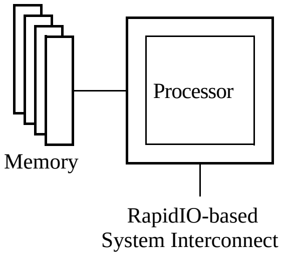


Figure 2-3. Integrated Processor-Memory Processing Element Example

2.2.3 Memory-Only Processing Element Model

A different processing element may not contain a processor at all, but may be a memory-only device as in Figure 2-4. This type of device is much simpler than a processor as it is only responsible for responding to requests from the external system, not from local requests as in the processor-based model. As such, its memory is remote for all processors in the system.

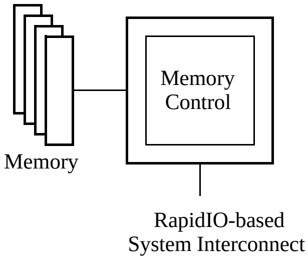


Figure 2-4. Memory-Only Processing Element Example

2.2.4 Processor-Only Processing Element

Similar to a memory-only element, a processor-only element has no local memory. A processor-only processing element is shown in Figure 2-5.

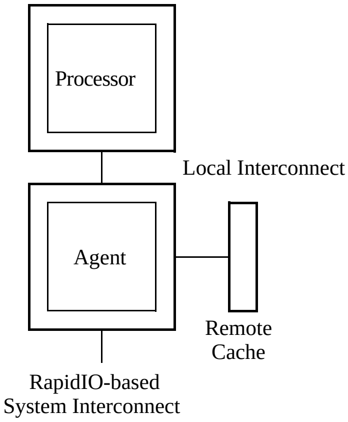


Figure 2-5. Processor-Only Processing Element Example

2.2.5 I/O Processing Element

This type of processing element is shown as the bridge in Figure 2-1. This device has distinctly different behavior than a processor or a memory. An I/O device only needs to move data into and out of local or remote memory in a cache coherent fashion. This means that if the I/O device needs to read from memory, it only needs to obtain a known good copy of the data to write to the external device (such as a disk drive or video display). If the I/O device needs to write to memory, it only needs to get ownership of the coherence granule returned to the home memory and not take ownership for itself. Both of these operations have special support in the RapidIO protocol.

2.2.6 Switch Processing Element

A switch processing element is a device that allows communication with other processing elements through the switch. A switch may be used to connect a variety of RapidIO compliant processing elements. A possible switch is shown in Figure 2-6.

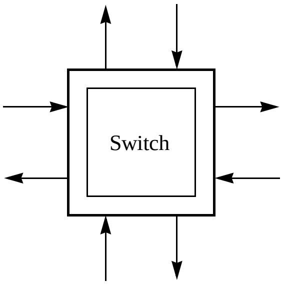


Figure 2-6. Switch Processing Element Example

2.3 Programming Models

RapidIO supports applications developed under globally shared memory and software-managed cache coherence programming models.

2.3.1 Globally Shared Memory System Model

The preferred programming model for modern computer systems provides memory that is accessible from all processors in a cache coherent fashion. This model is also known as GSM, or globally shared memory. For traditional bus-based computer systems this is not a difficult technical problem to solve because all participants in the cache coherence mechanism see all memory activity simultaneously, meaning that communication between processors is very fast and handled without explicit software control. However, in a non-uniform memory access system, this simultaneous memory access visibility is not the case.

With a distributed memory system, cache coherence needs to be maintained through some tracking mechanism that keeps records of memory access activity and explicitly notifies specific cache coherence participant processing elements when a cache coherence hazard is detected. For example, if a processing element wishes to write to a memory address, all participant processing elements that have accessed that coherence granule are notified to invalidate that address in their caches. Only when all of the participant processing elements have completed the invalidate operation and replied back to the tracking mechanism is the write allowed to proceed.

The tracking mechanism preferred for the RapidIO protocol is the memory directory based system model. This system model allows efficient, moderate scalability with a reasonable amount of information storage required for the tracking mechanism.

Cache coherence is defined around the concept of domains. The RapidIO protocol assumes a memory directory based cache coherence mechanism. Because the storage requirements for the directory can be high, the protocol was optimized assuming a 16-participant domain size as a reasonable coherence scalability limit. With this limit in mind, a moderately scalable system of 16 participants can be designed, possibly using a multicast mechanism in the transport layer for better efficiency. This size does not limit a system designer from defining a larger or a smaller coherent system such as the four processing element system in Figure 2-1 on page 17 since the number of domains and the number of participants is flexible. The total number of coherence domains and the scalability limit are determined by the number of transport bits allowed by the appropriate transport layer specification.

Table 2-1 describes an example of the directory states assumed for the RapidIO protocol for a small four-processing element cache coherent system (the table assumes that processor 0 is the local processor). Every coherence granule that is accessible by a remote processing element has this 4-bit field associated with it, so some state storage is required for each globally shared granule. The least significant bit (the right most, bit 3) indicates that a processing element has taken ownership of a coherence granule. The remaining three bits indicate that processing elements have accessed that coherence granule, or the current owner if the granule has been modified, with bit 0 corresponding to processor 3, bit 1 corresponding to processor 2, and bit 2 corresponding to processor 1. These bits are also known as the sharing mask or sharing list.

Owing to the encoding of the bits, the local processing element is always assumed to have accessed the granule even if it has not. This definition allows us to know exactly which processing elements have participated in the cache coherency protocol for each shared coherence granule at all times. Other state definitions can be implemented as long as they encompass the MSL (modified, shared, local) state functionality described here.

Table 2-1. RapidIO Memory Directory Definition

State	Description
0000	Processor 0 (local) shared
0001	Processor 0 (local) modified
0010	Processor 1, 0 shared
0011	Processor 1 modified
0100	Processor 2, 0 shared
0101	Processor 2 modified
0110	Processor 2, 1, 0 shared
0111	Illegal
1000	Processor 3, 0 shared
1001	Processor 3 modified
1010	Processor 3, 1, 0 shared
1011	Illegal
1100	Processor 3, 2, 0 shared
1101	Illegal

Table 2-1. RapidIO Memory Directory Definition (Continued)

1110	Processor 3, 2, 1, 0 shared
1111	Illegal

When a coherence granule is referenced, the corresponding 4-bit coherence state is examined by the memory controller to determine if the access can be handled in memory, or if data must be obtained from the current owner (a shared granule is owned by the home memory). Coherence activity in the system is started using the cache coherence protocol, if it is necessary to do so, to complete the memory operation.

2.3.1.1 Software-Managed Cache Coherence Programming Model

The software-managed cache coherence programming model depends upon the application programmer to guarantee that the same coherence granule is not resident in more than one cache in the system simultaneously if it is possible for that coherence granule to be written by one of the processors. The application software allows sharing of written data by using cache manipulation instructions to flush these coherence granules to memory before they are read by another processor. This programming model is useful in transaction and distributed processing types of systems.

2.4 System Issues

The following sections describe transaction ordering and system deadlock considerations in a RapidIO GSM system.

2.4.1 Operation Ordering

Operation completion ordering in a globally shared memory system is managed by the completion units of the processing elements participating in the coherence protocol and by the coherence protocol itself.

2.4.2 Transaction Delivery

There are two basic types of delivery schemes that can be built using RapidIO processing elements: unordered and ordered. The RapidIO logical protocols assume that all outstanding transactions to another processing element are delivered in an arbitrary order. In other words, the logical protocols do not rely on transaction interdependencies for operation. RapidIO also allows completely ordered delivery systems to be constructed. Each type of system puts different constraints on the implementation of the source and destination processing elements and any intervening hardware. The specific mechanisms and definitions of how RapidIO enforces transaction ordering are discussed in the appropriate physical layer specification.

2.4.3 Deadlock Considerations

A deadlock can occur if a dependency loop exists. A dependency loop is a situation where a loop of buffering devices is formed, in which forward progress at each device is dependent upon progress at the next device. If no device in the loop can make progress then the system is deadlocked.

The simplest solution to the deadlock problem is to discard a packet. This releases resources in the network and allows forward progress to be made. RapidIO is designed to be a reliable fabric for use in real time tightly coupled systems, therefore, discarding packets is not an acceptable solution.

In order to produce a system with no chance of deadlock it is required that a deadlock free topology be provided for response-less operations. Dependency loops to single direction packets can exist in unconstrained switch topologies. Often the dependency loop can be avoided with simple routing rules. Topologies like hypercubes or three-dimensional meshes, physically contain loops. In both cases, routing is done in several dimensions (x,y,z). If routing is constrained to the x dimension, then y, then z (dimension ordered routing), then topology related dependency loops are avoided in these structures.

In addition, a processing element design shall not form dependency links between its input and output port. A dependency link between input and output ports occurs if a processing element is unable to accept an input packet until a waiting packet can be issued from the output port.

RapidIO supports operations, such as coherent read-for-ownership operations, that require responses to complete. These operations can lead to a dependency link between an processing element’s input port and output port.

As an example of an input to output port dependency, consider a processing element where the output port queue is full. The processing element can not accept a new request at its input port since there is no place to put the response in the output port queue. No more transactions can be accepted at the input port until the output port is able to free entries in the output queue by issuing packets to the system.

A further consideration is that of the read-for-ownership operation colliding with a castout of the requested memory address by another processing element. In order for the read-for-ownership operation to complete the underlying castout operation must complete. Therefore the castout must be given higher preference in the system in order to move ahead of other operations in order to break up the dependency.

The method by which a RapidIO system maintains a deadlock free environment is described in the appropriate Physical Layer specification.

Chapter 3 Operation Descriptions

3.1 Introduction

This chapter describes the set of operations and transactions supported by the RapidIO globally-shared memory (GSM) protocols. The opcodes and packet formats are described in Chapter 4, “Packet Format Descriptions.” The complete protocols are described in Chapter 6, “Communication Protocols.”

The RapidIO operation protocols use request/response transaction pairs through the interconnect fabric. A processing element sends a request transaction to another processing element if it requires an activity to be carried out. The receiving processing element responds with a response transaction when the request has been completed or if an error condition is encountered. Each transaction is sent as a packet through the interconnect fabric. For example, a processing element that requires data from home memory in another processing element sends a READ_HOME transaction in a request packet. The receiving element then reads its local memory at the requested address and returns the data in a DONE transaction via a response packet. Note that not all requests require responses; some requests assume that the desired activity will complete properly.

A number of possible response transactions can be received by a requesting processing element:

- A DONE response indicates to the requestor that the desired transaction has completed and also returns data for read-type transactions as described above.
- The INTERVENTION, DONE_INTERVENTION, and DATA_ONLY responses are generated as part of the processing element-to-processing element (as opposed to processing element-to-home memory) data transfer mechanism defined by the cache coherence protocol. The INTERVENTION and DONE_INTERVENTION responses are abbreviated as INTERV and DONE_INTERV in this chapter.
- The NOT_OWNER and RETRY responses are received when there are address conflicts within the system that need resolution.
- An ERROR response means that the target of the transaction encountered an unrecoverable error and could not complete the transaction.

Packets may contain additional information that is interpreted by the interconnect fabric to route the packets through the fabric from the source to the destination, such as a device number. These requirements are described in the appropriate RapidIO transport layer specification and are beyond the scope of this specification.

Depending upon the interconnect fabric, other packets may be generated as part of the physical layer protocol to manage flow control, errors, etc. Flow control and other fabric-specific communication requirements are described in the appropriate RapidIO physical layer specification and are beyond the scope of this document.

Each request transaction sent into the system is marked with a transaction ID that is unique for each requestor and responder processing element pair. This transaction ID allows a response to be easily matched to the original request when it is returned to the requestor. An end point cannot reuse a transaction ID value to the same destination until the response from the original transaction has been received by the requestor. The number of outstanding transactions that may be supported is implementation dependent.

The transaction behaviors are also described as state machine behavior in Chapter 6, “Communication Protocols”.

3.2 GSM Operations Cross Reference

Table 3-1 contains a cross reference of the GSM operations defined in this RapidIO specification and their system usage.

Table 3-1. GSM Operations Cross Reference

Operation	Transactions Used	Possible System Usage	Description	Packet Format	Protocol
Read	READ_HOME, READ_OWNER, RESPONSE	CC-NUMA operation	Section 3.3.1 page 28	Types 1 and 2: Section 4.2.5 page 50 and Section 4.2.6 page 51	Section 6.4 page 68
Instruction read	IREAD_HOME, READ_OWNER, RESPONSE	Combination of CC-NUMA and software-maintained coherence of instruction caches	Section 3.3.2 page 29	Type 2 Section 4.2.6 page 51	Section 6.4 page 68
Read-for-ownership	READ_TO_OWN_HOME, READ_TO_OWN_OWNER, DKILL_SHARER RESPONSE	CC-NUMA operation	Section 3.3.3 page 31	Types 1 and 2: Section 4.2.5 page 50 and Section 4.2.6 page 51	Section 6.6 page 75
Data cache invalidate	DKILL_HOME, DKILL_SHARER, RESPONSE	CC-NUMA operation; software-maintained coherence operation	Section 3.3.4 page 33	Type 2 Section 4.2.6 page 51	Section 6.7 page 79
Castout	CASTOUT, RESPONSE	CC-NUMA operation	Section 3.3.5 page 34	Type 5 Section 4.2.8 page 52	Section 6.8 page 82

Table 3-1. GSM Operations Cross Reference (Continued)

Operation	Transactions Used	Possible System Usage	Description	Packet Format	Protocol
TLB invalidate-entry	TLBIE, RESPONSE	Software-maintained coherence of page table entries	Section 3.3.6	Type 2 Section 4.2.6	Section 6.9
TLB invalidate-entry synchronize	TLBSYNC, RESPONSE	Software-maintained coherence of page table entries	Section 3.3.7	Type 2 Section 4.2.6	Section 6.9
Instruction cache invalidate	IKILL_HOME, IKILL_SHARER, RESPONSE,	Software-maintained coherence of instruction caches	Section 3.3.8	Type 2 Section 4.2.6	Section 6.7

Data cache flush	FLUSH, DKILL_SHARER, READ_TO_OWN_OWNER, RESPONSE	CC-NUMA flush instructions;
		CC-NUMA write-through cache support;
		CC-NUMA DMA I/O device support;
		software-maintained coherence operation.
			Section 3.3.9	Types 2 and 5:
				Section 4.2.6
				and
				Section 4.2.8
					Section 6.10

I/O read	IO_READ_HOME, IO_READ_OWNER, INTERV, RESPONSE	CC-NUMA DMA, I/O DMA device support	Section 3.3.10	Types 1 and 2: Section 4.2.5 and Section 4.2.6	Section 6.11

3.3 GSM Operations

A set of transactions are used to support GSM (cache coherence) operations to cacheable memory space. The following descriptions assume that all requests are to system memory rather than to some other type of device.

GSM operations occur based on the size of the coherence granule. Changes in the coherence granule for a system do not change any of the operation protocols, only the data payload size. The only exceptions to this are flush and I/O read operations, which may request (in the case of an I/O read), or have (in the case of a flush) a sub-coherence granule to support coherent I/O and write-through caches. Flush operations may also have no data payload in order to support cache manipulation instructions.

Some transactions are sent to multiple recipients in the process of completing an operation. These transactions can be sent either as a number of directed transactions or as a single transaction if the transport layer has multicast capability. Multicast capability and operation is defined in the appropriate RapidIO transport layer specification.

3.3.1 Read Operations

The READ_HOME, READ_OWNER, and RESPONSE transactions are used during a read operation by a processing element that needs a shared copy of cache-coherent data from the memory system. A read operation always returns one coherence granule-sized data payload.

The READ_HOME transaction is used by a processing element that needs to read a shared copy of a coherence granule from a remote home memory on another processing element.

The READ_OWNER transaction is used by a home memory processing element that needs to read a shared copy of a coherence granule that is owned by a remote processing element.

The following types of read operations are possible:

- If the requested data exists in the memory directory as shared, the data can be returned immediately from memory with a DONE RESPONSE transaction and the requesting processing element's device ID is added to the sharing mask as shown in Figure 3-1.

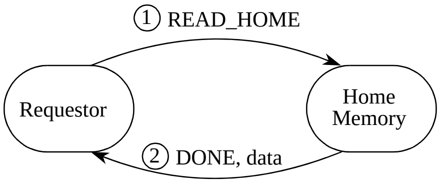


Figure 3-1. Read Operation to Remote Shared Coherence Granule

- If the requested data exists in the memory directory as modified, the up-to-date (current) data must be obtained from the owner. The home memory then sends a READ_OWNER request to the processing element that owns the coherence granule. The owner passes a copy of the data to the original requestor and to memory, memory is updated, and the directory state is changed from modified and owner to shared by the previous owner and the requesting processing element's device ID as shown in Figure 3-2.

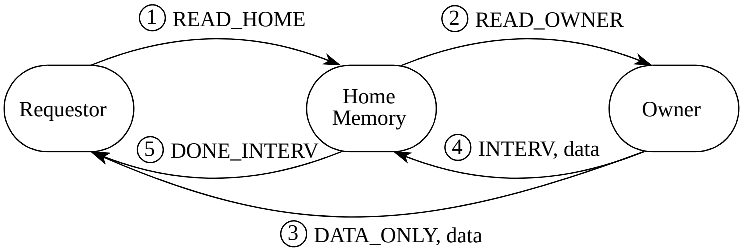


Figure 3-2. Read Operation to Remote Modified Coherence Granule

- If the processing element requesting a modified coherence granule happens to be the home for the memory, some of the transactions can be eliminated as shown in Figure 3-3.

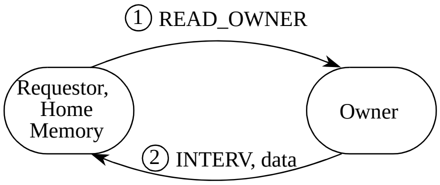


Figure 3-3. Read Operation to Local Modified Coherence Granule

3.3.2 Instruction Read Operations

Some processors have instruction caches that do not participate in the system cache coherence mechanism. Additionally, the instruction cache load may also load a shared instruction and data cache lower in the cache hierarchy. This can lead to a situation where the instruction cache issues a shared read operation to the system for a coherence granule that is owned by that processor’s data cache, resulting in a cache coherence paradox to the home memory directory.

Due to this situation, an instruction read operation must behave like a coherent shared read relative to the memory directory and as a non-coherent operation relative to the requestor. Therefore, the behavior of the instruction read operation is nearly identical to a data read operation with the only difference being the way that the apparent coherence paradox is managed.

The IREAD_HOME and RESPONSE transactions are used during an instruction read operation by a processing element that needs a copy of sharable instructions from the memory system. An instruction read operation always returns one coherence granule-sized data payload. Use of the IREAD_HOME transaction rather than the READ_HOME transaction allows the memory directory to properly handle the paradox case without sacrificing coherence error detection in the system. The IREAD_HOME transaction participates in address collision detection at the home memory but does not participate in address collision detection at the requestor.

The following types of instruction read operations are possible:

- If the requested instructions exists in the memory directory as shared, the instructions can be returned immediately from memory and the requesting processing element’s device ID is added to the sharing mask as shown in Figure 3-4.

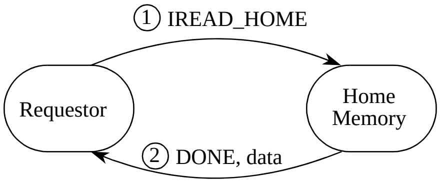


Figure 3-4. Instruction Read Operation to Remote Shared Coherence Granule

- If the requested data exists in the memory directory as modified, the up-to-date (current) data must be obtained from the owner. The home memory then sends a READ_OWNER request to the processing element that owns the coherence granule. The owner passes a copy of the data to the original requestor and to memory, memory is updated, and the directory state is changed from modified and owner to shared by the previous owner and the requesting processing element's device ID as shown in Figure 3-5.

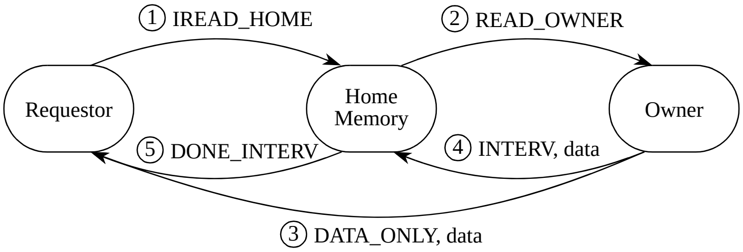


Figure 3-5. Instruction Read Operation to Remote Modified Coherence Granule

- If the processing element requesting a modified coherence granule happens to be the home for the memory the READ_OWNER transaction is used to obtain the coherence granule as shown in Figure 3-6.

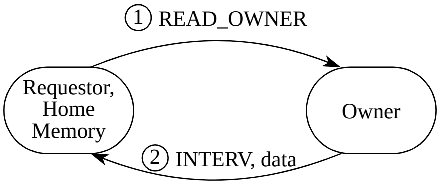


Figure 3-6. Instruction Read Operation to Local Modified Coherence Granule

- The apparent paradox case is if the requesting processing element is the owner of the coherence granule as shown in Figure 3-7. The home memory sends a READ_OWNER transaction back to the requesting processing element with the source and secondary ID set to the home memory ID, which indicates that the response behavior should be an INTERVENTION transaction rather than an INTERVENTION and a DATA_ONLY transaction as shown in Figure 3-5.

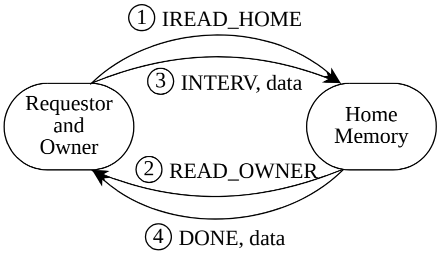


Figure 3-7. Instruction Read Operation Paradox Case

3.3.3 Read-for-Ownership Operations

The READ_TO_OWN_HOME, READ_TO_OWN_OWNER, DKILL_SHARER, and RESPONSE transactions are used during read-for-ownership operations by a processing element that needs to write to a coherence granule that does not exist in its caching hierarchy. A read-for-ownership operation always returns one coherence granule-sized data payload. These transactions are used as follows:

- The READ_TO_OWN_HOME transaction is used by a processing element that needs to read a writable copy of a coherence granule from a remote home memory on another processing element. This transaction causes a copy of the data to be returned to the requestor, from memory if the data is shared, or from the owner if it is modified.
- The READ_TO_OWN_OWNER transaction is used by a home memory processing element that needs to read a writable copy of a coherence granule that is owned by a remote processing element.
- The DKILL_SHARER transaction is used by the home memory processing element to invalidate shared copies of the coherence granule in remote processing elements.

Following are descriptions of the read-for-ownership operations:

- If the coherence granule is shared, DKILL_SHARER transactions are sent to the participants indicated in the sharing mask, which results in a cache invalidate operation for the recipients as shown in Figure 3-8.

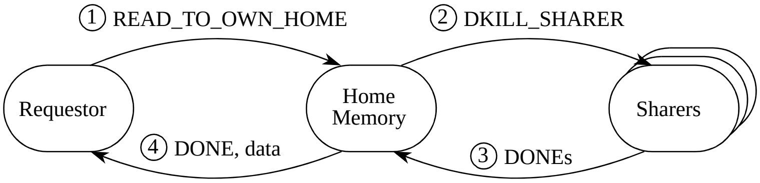


Figure 3-8. Read-for-Ownership Operation to Remote Shared Coherence Granule

- If the coherence granule is modified, a READ_TO_OWN_OWNER transaction is sent to the owner, who sends a copy of the data to the requestor (intervention) and marks the address as invalid as shown in Figure 3-9. The final memory directory state shows that the coherence granule is modified and owned by the requestor's device ID.

Because the coherence granule in the memory directory was marked as modified, home memory does not necessarily need to be updated. However, the RapidIO protocol requires that a processing element return the modified data and update the memory, allowing some attempt for data recovery if a coherence problem occurs.

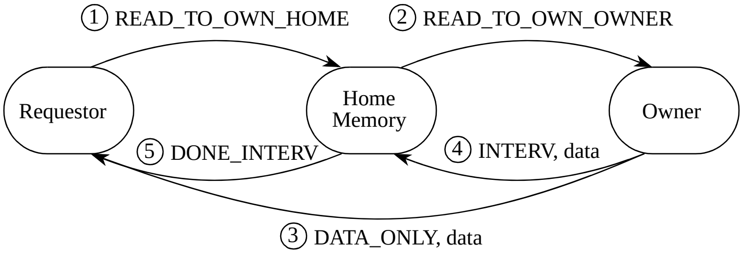


Figure 3-9. Read-for-Ownership Operation to Remote Modified Coherence Granule

- If the requestor is on the same processing element as the home memory and the coherence granule is shared, a DKILL_SHARER transaction is sent to all sharing processing elements (see Figure 3-10). The final directory state is marked as modified and owned by the local requestor.

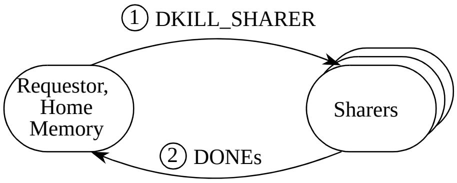


Figure 3-10. Read-for-Ownership Operation to Local Shared Coherence Granule

- If the requestor is on the same processing element as the home memory and the coherence granule is owned by a remote processing element, a READ_TO_OWN_OWNER transaction is sent to the owner (see Figure 3-11). The final directory state is marked as modified and owned by the local requestor.

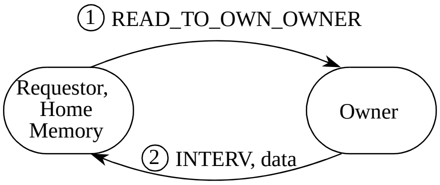


Figure 3-11. Read-for-Ownership Operation to Local Modified Coherence Granule

3.3.4 Data Cache Invalidate Operations

The DKILL_HOME, DKILL_SHARER, and RESPONSE transactions are requests to invalidate a coherence granule in all of the participants in the coherence domain as follows:

- The DKILL_HOME transaction is used by a processing element to invalidate a data coherence granule that has home memory in a remote processing element.
- The DKILL_SHARER transaction is used by the home memory processing element to invalidate shared copies of the data coherence granule in remote processing elements.

Data cache invalidate operations are also useful for systems that implement software-maintained cache coherence. In this case, a requestor may send DKILL_HOME and DKILL_SHARER transactions directly to other processing elements without going through home memory as in a CC-NUMA system. The transactions used for the data cache invalidate operation depend on whether the requestor is on the same processing element as the home memory of the coherence granule as follows:

- If the requestor is not on the same processing element as the home memory of the coherence granule, a DKILL_HOME transaction is sent to the remote home memory processing element. This causes the home memory for the shared coherence granule to send a DKILL_SHARER to all processing elements marked as sharing the granule in the memory directory state except for the requestor (see Figure 3-12). The final memory state shows that the coherence granule is modified and owned by the requesting processing element's device ID.

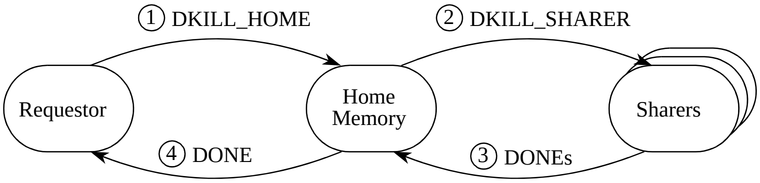


Figure 3-12. Data Cache Invalidate Operation to Remote Shared Coherence Granule

- If the requestor is on the same processing element as the home memory of the coherence granule, the home memory sends a DKILL_SHARER transaction to all processing elements marked as sharing the coherence granule in the memory directory. The final memory state shows the coherence granule modified and owned by the local processor (see Figure 3-13).

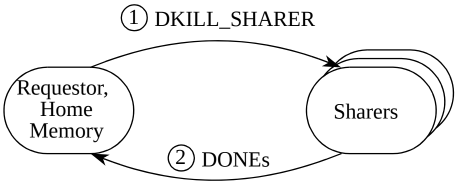


Figure 3-13. Data Cache Invalidate Operation to Local Shared Coherence Granule

3.3.5 Castout Operations

The CASTOUT and RESPONSE transactions are used in a castout operation by a processing element to relinquish its ownership of a coherence granule and return it to the home memory. The CASTOUT can be treated as a low-priority transaction unless there is an address collision with an incoming request, at which time it must become a high-priority transaction. The CASTOUT causes the home memory to be updated with the most recent data and changes the directory state to owned by home memory and shared (or owned, depending upon the default directory state) by the local processing element (see Figure 3-14).

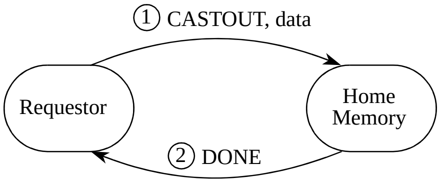


Figure 3-14. Castout Operation on Remote Modified Coherence Granule

A CASTOUT transaction does not participate in address collision detection at the home memory to prevent deadlocks or cache paradoxes caused by packet-to-packet timing in the interconnect fabric. For example, consider a case where processing element A is performing a CASTOUT that collides with an incoming READ_OWNER transaction. If the CASTOUT is not allowed to complete at the home memory, the system will deadlock. If the read operation that caused the READ_OWNER completes (through intervention) before the CASTOUT transaction is received at the home memory, the CASTOUT will appear to be illegal because the directory state will have changed.

3.3.6 TLB Invalidate-Entry Operations

The TLBIE and RESPONSE transactions are used for TLB invalidate-entry operations. If the processor TLBs do not participate in the cache coherence protocol, the TLB invalidate-entry operation is used when page table translation entries need to be modified. The TLBIE transaction is sent to all participants in the coherence domain except for the original requestor. A TLBIE transaction has no effect on the memory directory state for the specified address and does not participate in address collisions (see Figure 3-15).

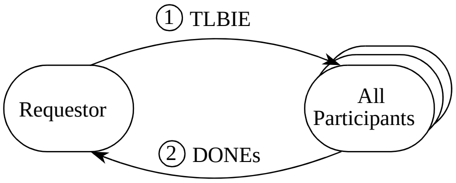


Figure 3-15. TLB Invalidate-Entry Operation

3.3.7 TLB Invalidate-Entry Synchronization Operations

The TLBSYNC and RESPONSE transactions are used for TLB invalidate-entry synchronization operations. It is used to force the completion of outstanding TLBIE transactions at the participants. The DONE response for a TLBSYNC transaction is only sent when all preceding TLBIE transactions have completed. This operation is necessary due to possible indeterminate completion of individual TLBIE transactions when multiple TLBIE transactions are being executed simultaneously. The TLBSYNC transaction is sent to all participants in the coherence domain except for the original requestor. The transaction has no effect on the memory directory state for the specified address and does not participate in address collisions (see Figure 3-16).

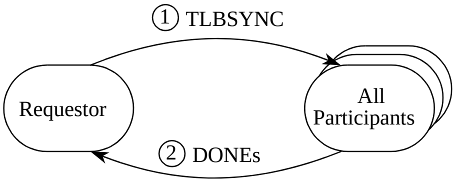


Figure 3-16. TLB Invalidate-Entry Synchronization Operation

3.3.8 Instruction Cache Invalidate Operations

The IKILL_HOME, IKILL_SHARER, and RESPONSE transactions are used during instruction cache invalidate operations to invalidate shared copies of an instruction coherence granule in remote processing elements. Instruction cache invalidate operations are needed if the processor instruction caches do not participate in the cache coherence protocol, requiring instruction cache coherence to be maintained by software.

An instruction cache invalidate operation has no effect on the memory directory state for the specified address and does not participate in address collisions. Following are descriptions of the instruction cache invalidate operations:

- If the requestor is not on the same processing element as the home memory of the coherence granule, an IKILL_HOME transaction is sent to the remote home memory processing element. This causes the home memory for the shared coherence granule to send an IKILL_SHARER to all processing element participants in the coherence domain because the memory directory state only properly tracks data, not instruction, accesses. (See Figure 3-17.)

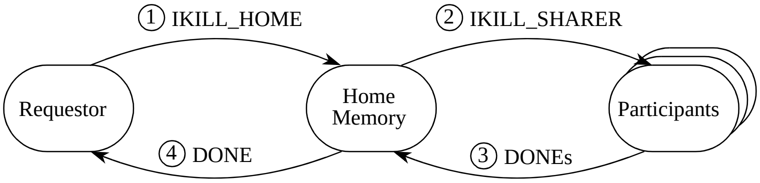


Figure 3-17. Instruction Cache Invalidate Operation to Remote Sharable Coherence Granule

- If the requestor is on the same processing element as the home memory of the coherence granule, the home memory sends an IKILL_SHARER transaction to all processing element participants in the coherence domain as shown in Figure 3-18.

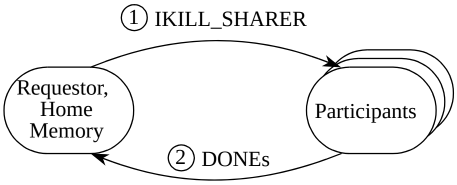


Figure 3-18. Instruction Cache Invalidate Operation to Local Sharable Coherence Granule

3.3.9 Data Cache Flush Operations

The FLUSH, DKILL_SHARER, READ_TO_OWN_OWNER, and RESPONSE transactions are used for data cache flush operations, which return ownership of a coherence granule back to the home memory if it is modified and invalidate all copies if the granule is shared. A flush operation with associated data can be used to implement an I/O system write operation and to implement processor write-through and cache manipulation operations. These transactions are used as follows:

- The FLUSH transaction is used by a processing element to return the ownership and current data of a coherence granule to home memory. The data payload for the FLUSH transaction is typically the size of the coherence granule for the system but may be multiple double-words or one double-word or less. FLUSH transactions without a data payload are used to support cache manipulation operations. The memory directory state is changed to owned by home memory and shared (or modified, depending upon the processing element's normal default state) by the local processing element.

- The DKILL_SHARER transaction is used by the home memory processing element to invalidate shared copies of the data coherence granule in remote processing elements.
- The READ_TO_OWN_OWNER transaction is used by a home memory processing element that needs to retrieve ownership of a coherence granule that is owned by a remote processing element.

The FLUSH transaction is able to specify multiple double-word and sub-double-word data payloads; however, they must be aligned to byte, half-word, word, or double-word boundaries. Multiple double-word FLUSH transactions cannot exceed the number of double-words in the coherence granule. The write size and alignment for the FLUSH transaction are specified in Table 4-8. Unaligned and non-contiguous operations are not supported and must be broken into multiple FLUSH transactions by the sending processing element.

A flush operation internal to a processing element that would cause a FLUSH transaction for a remote coherence granule owned by that processing element (for example, attempting a cache write-through operation to a locally owned remote coherence granule) must generate a CASTOUT rather than a FLUSH transaction to properly implement the RapidIO protocol. Issuing a FLUSH under these circumstances generates a memory directory state paradox error in the home memory processing element.

Following are descriptions of the flush operations:

- If a flush operation is to a remote shared coherence granule, the FLUSH transaction is sent to the home memory, which sends a DKILL_SHARER transaction to all of the processing elements marked in the sharing list except for the requesting processing element. The processing elements that receive the DKILL_SHARER transaction invalidate the specified address if it is found shared in their caching hierarchy (see Figure 3-19).

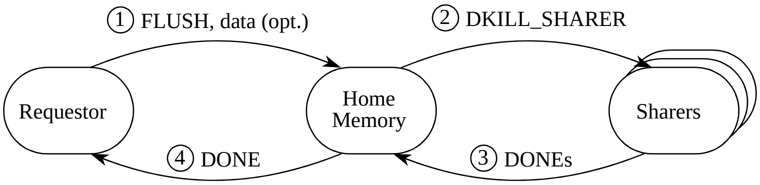


Figure 3-19. Flush Operation to Remote Shared Coherence Granule

- If the coherence granule is owned by a remote processing element, the home memory sends a READ_TO_OWN_OWNER transaction to it with the secondary (intervention) ID set to the home memory ID instead of the requestor ID. The owner then invalidates the coherence granule in its caching hierarchy and returns the coherence granule data (see Figure 3-20).

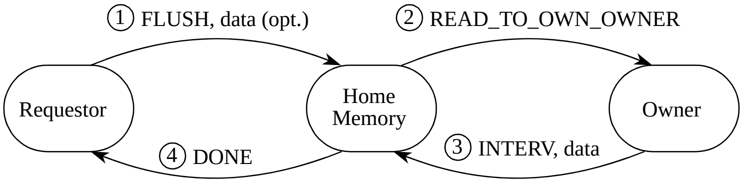


Figure 3-20. Flush Operation to Remote Modified Coherence Granule

- If the requestor and the home memory for the coherence granule are in the same processing element, DKILL_SHARER transactions are sent to all participants marked in the sharing list (see Figure 3-21).

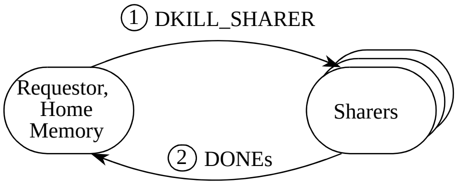


Figure 3-21. Flush Operation to Local Shared Coherence Granule

- If the requestor and the home memory for the coherence granule are in the same processing element but the coherence granule is owned by a remote processing element, a READ_TO_OWN_OWNER transaction is sent to the owner (see Figure 3-22).

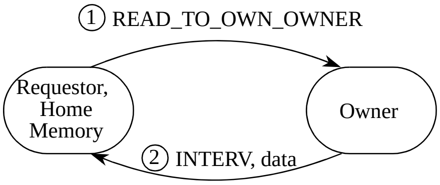


Figure 3-22. Flush Operation to Local Modified Coherence Granule

3.3.10 I/O Read Operations

The IO_READ_HOME, IO_READ_OWNER, and RESPONSE transactions are used during I/O read operations by a processing element that needs a current copy of cache-coherent data from the memory system, but does not need to be added to the sharing list in the memory directory state. The I/O read operation is most useful for DMA I/O devices. An I/O read operation always returns the requested size data payload. The requested data payload size can not exceed the size of the coherence granule. These transactions are used as follows:

- The IO_READ_HOME transaction is used by a requestor that is not in the same processing element as the home memory for the coherence granule.
- The IO_READ_OWNER transaction is used by a home memory processing element that needs to read a copy of a coherence granule owned by a remote processing element.

Following are descriptions of the I/O operations:

- If the requested data exists in the memory directory as shared, the data can be returned immediately from memory and the sharing mask is not modified (see Figure 3-24).

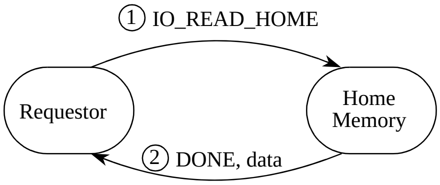


Figure 3-23. I/O Read Operation to Remote Shared Coherence Granule

- If the requested data exists in the memory directory as modified, the home memory sends an IO_READ_OWNER transaction to the processing element that owns the coherence granule. The owner passes a copy of the data to the requesting processing element (intervention) but retains ownership of and responsibility for the coherence granule (see Figure 3-24 and Figure 3-25).

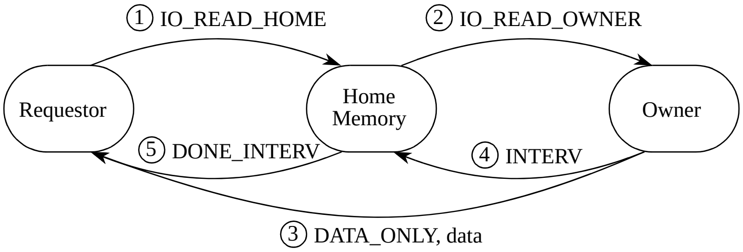


Figure 3-24. I/O Read Operation to Remote Modified Coherence Granule

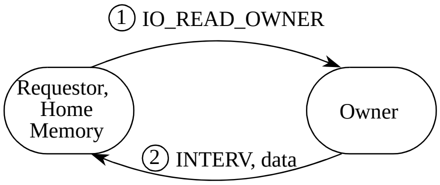
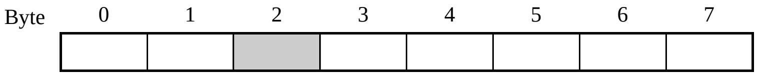


Figure 3-25. I/O Read Operation to Local Modified Coherence Granule

3.4 Endian, Byte Ordering, and Alignment

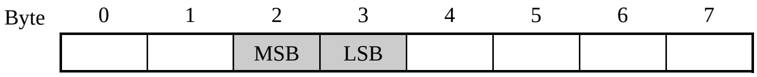
RapidIO has double-word (8-byte) aligned big-endian data payloads. This means that the RapidIO interface to devices that are little-endian shall perform the proper endian transformation to format a data payload.

Operations that specify data quantities that are less than 8 bytes shall have the bytes aligned to their proper byte position within the big-endian double-word, as in the examples shown in Figure 3-26 through Figure 3-28.



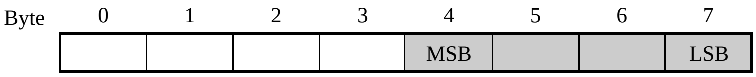
Byte address 0x0000_0002, the proper byte position is shaded.

Figure 3-26. Byte Alignment Example



Half-word address 0x0000_0002, the proper byte positions are shaded.

Figure 3-27. Half-Word Alignment Example

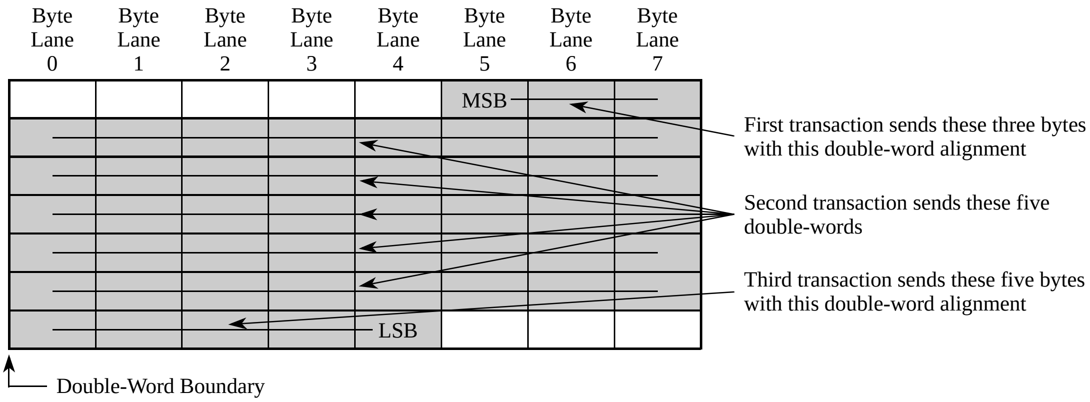


Word address 0x0000_0004, the proper byte positions are shaded.

Figure 3-28. Word Alignment Example

For write operations, a processing element shall properly align data transfers to a double-word boundary for transmission to the destination. This alignment may require breaking up a data stream into multiple transactions if the data is not naturally aligned. A number of data payload sizes and double-word alignments are defined to minimize this burden. Figure 3-29 shows a 48-byte data stream that a processing element wishes to write to another processing element through the interconnect fabric. The data displayed in the figure is big-endian and double-word aligned with the bytes to be written shaded in grey. Because the start of the stream and the end of the stream are not aligned to a double-word boundary, the sending processing element shall break the stream into three transactions as shown in the figure.

The first transaction sends the first three bytes (in byte lanes 5, 6, and 7) and indicates a byte lane 5, 6, and 7 three-byte write. The second transaction sends all of the remaining data except for the final sub-double-word. The third transaction sends the final 5 bytes in byte lanes 0, 1, 2, 3, and 4 indicating a five-byte write in byte lanes 0, 1, 2, 3, and 4.



First transaction sends these three bytes with this double-word alignment

Second transaction sends these five double-words

Third transaction sends these five bytes with this double-word alignment

Figure 3-29. Data Alignment Example

Blank page

Chapter 4 Packet Format Descriptions

4.1 Introduction

This chapter contains the packet format definitions for the *RapidIO Interconnect Globally Shared Memory Logical Specification*. There are four types of globally shared memory packet formats:

- Request
- Response
- Implementation-defined
- Reserved

The packet formats are intended to be interconnect fabric independent, so the system interconnect can be anything required for a particular application. Reserved formats, unless defined in another logical specification, shall not be used by a device.

4.2 Request Packet Formats

A request packet is issued by a processing element that needs a remote processing element to accomplish some activity on its behalf, such as a memory read operation. The request packet format types and their transactions for the *RapidIO Interconnect Globally Shared Memory Logical Specification* are shown in Table 4-1.

Table 4-1. Request Packet Type to Transaction Type Cross Reference

Request Packet Format Type	Transaction Type	Definition	Document Section No.
Type 0	Implementation-defined	Defined by the device implementation	Section 4.2.4
Type 1	READ_OWNER	Read shared copy of remotely owned coherence granule	Section 4.2.5
	READ_TO_OWN_OWNER	Read for store of remotely owned coherence granule	
	IO_READ_OWNER	Read for I/O of remotely owned coherence granule	

Table 4-1. Request Packet Type to Transaction Type Cross Reference (Continued)

Request Packet Format Type	Transaction Type	Definition	Document Section No.
Type 2	READ_TO_OWN_HOME	Read for store of home memory for coherence granule	Section 4.2.6
	READ_HOME	Read shared copy of home memory for coherence granule	
	IO_READ_HOME	Read for I/O of home memory for coherence granule	
	DKILL_HOME	Invalidate to home memory of coherence granule	
	IKILL_HOME	Invalidate to home memory of coherence granule	
	TLBIE	Invalidate TLB entry	
	TLBSYNC	Synchronize TLB invalidates	
	IREAD_HOME	Read shared copy of home memory for instruction cache	
	FLUSH	Force return of ownership of coherence granule to home memory, no update to coherence granule	
	IKILL_SHARER	Invalidate cached copy of coherence granule	
	DKILL_SHARER	Invalidate cached copy of coherence granule	
Type 3–4	—	Reserved	Section 4.2.7
Type 5	CASTOUT	Return ownership of coherence granule to home memory	Section 4.2.8
	FLUSH (with data)	Force return of ownership of coherence granule to home memory, update returned coherence granule	
Type 6–11	—	Reserved	Section 4.2.9

4.2.1 Addressing and Alignment

The size of the address is defined as a system-wide parameter; thus the packet formats do not support mixed local physical address fields simultaneously. The least three significant bits of all addresses are not specified and are assumed to be logic 0.

The coherence-granule-sized cache-coherent write requests and read responses are aligned to a double-word boundary within the coherence granule, with the specified data payload size matching that of the coherence granule. Sub-double-word data payloads must be padded and properly aligned within the 8-byte boundary. Non-contiguous or unaligned transactions that would ordinarily require a byte mask are not supported. A sending device that requires this behavior must break the operation into multiple request transactions. An example of this is shown in Section 3.4, “Endian, Byte Ordering, and Alignment.”

4.2.2 Data Payloads

Cache coherent systems are very sensitive to memory read latency. One way of reducing the latency is by returning the requested, or critical, double-word first upon a read request. Subsequent double-words are then returned in a sequential fashion. Table 4-2 and Table 4-3 show the return ordering for 32- and 64-byte coherence granules. Sub-double-word data payloads due to I/O read operations start with the requested size as shown.

Table 4-2. Coherent 32-Byte Read Data Return Ordering

Requested Double-word	Double-word Return Ordering
0	0, 1, 2, 3
1	1, 2, 3, 0
2	2, 3, 0, 1
3	3, 0, 1, 2

Table 4-3. Coherent 64-Byte Read Data Return Ordering

Requested Double-word	Double-word Return Ordering
0	0, 1, 2, 3, 4, 5, 6, 7
1	1, 2, 3, 0, 4, 5, 6, 7
2	2, 3, 0, 1, 4, 5, 6, 7
3	3, 0, 1, 2, 4, 5, 6, 7
4	4, 5, 6, 7, 0, 1, 2, 3
5	5, 6, 7, 4, 0, 1, 2, 3
6	6, 7, 4, 5, 0, 1, 2, 3
7	7, 4, 5, 6, 0, 1, 2, 3

Data payloads for cache coherent write-type transactions are always linear starting with the specified address at the first double-word to be written, (including flush transactions that are not the size of the coherence granule). Data payloads that cross the coherence granule boundary can not be specified. This implies that all castout transactions start with the first double-word in the coherence granule. Table 4-4 and Table 4-5 show the cache-coherent write-data ordering for 32- and 64-byte coherence granules, respectively.

Table 4-4. Coherent 32-Byte Write Data Payload

Starting Double-word	Number of Double-words	Double-word Data Ordering Within Coherence Granule
0	1	0
0	2	0, 1
0	3	0, 1, 2
0	4	0, 1, 2, 3
1	1	1
1	2	1, 2
1	3	1, 2, 3
2	1	2
2	2	2, 3
3	1	3

Table 4-5. Coherent 64-Byte Write Data Payloads

Starting Double-word	Number of Double-words	Double-word Data Ordering Within Coherence Granule
0	1	0
0	2	0, 1
0	3	0, 1, 2
0	4	0, 1, 2, 3
0	5	0, 1, 2, 3, 4
0	6	0, 1, 2, 3, 4, 5
0	7	0, 1, 2, 3, 4, 5, 6
0	8	0, 1, 2, 3, 4, 5, 6, 7
1	1	1
1	2	1, 2
1	3	1, 2, 3
1	4	1, 2, 3, 4
1	5	1, 2, 3, 4, 5
1	6	1, 2, 3, 4, 5, 6
1	7	1, 2, 3, 4, 5, 6, 7
2	1	2
2	2	2, 3
2	3	2, 3, 4
2	4	2, 3, 4, 5
2	5	2, 3, 4, 5, 6
2	6	2, 3, 4, 5, 6, 7
3	1	3
3	2	3, 4

Table 4-5. Coherent 64-Byte Write Data Payloads (Continued)

Starting Double-word	Number of Double-words	Double-word Data Ordering Within Coherence Granule
3	3	3, 4, 5
3	4	3, 4, 5, 6
3	5	3, 4, 5, 6, 7
4	1	4
4	2	4, 5
4	3	4, 5, 6
4	4	4, 5, 6, 7
5	1	5
5	2	5, 6
5	3	5, 6, 7
6	1	6
6	2	6, 7
7	1	7

4.2.3 Field Definitions for All Request Packet Formats

Fields that are unique to type 1, type 2, and type 5 formats are defined in their sections. Bit fields that are defined as “reserved” shall be assigned to logic 0s when generated and ignored when received. Bit field encodings that are defined as “reserved” shall not be assigned when the packet is generated. A received reserved encoding is regarded as an error if a meaningful encoding is required for the transaction and function, otherwise it is ignored. Implementation-defined fields shall be ignored unless the encoding is understood by the receiving device. All packets described are bit streams from the first bit to the last bit, represented in the figures from left to right respectively.

The following field definitions in Table 4-6 apply to all of the request packet formats.

Table 4-6. General Field Definitions for All Request Packets

Field	Definition
ftype	Format type, represented as a 4-bit value; is always the first four bits in the logical packet stream.
wdptr	Word pointer, used in conjunction with the data size (rdsize and wrsize) fields—see Table 4-7, Table 4-8, and Section 3.4.
rdsize	Data size for read transactions, used in conjunction with the word pointer (wdptr) bit—see Table 4-7 and Section 3.4.
wrsize	Write data size for sub-double-word transactions, used in conjunction with the word pointer (wdptr) bit—see Table 4-8 and Section 3.4. For writes greater than one double-word, the size is the maximum payload.
rsrv	Reserved
srcTID	The packet’s transaction ID.

Table 4-6. General Field Definitions for All Request Packets (Continued)

Field	Definition
transaction	The specific transaction within the format class to be performed by the recipient; also called type or ttype.
extended address	Optional. Specifies the most significant 16 bits of a 50-bit physical address or 32 bits of a 66-bit physical address.
xamsbs	Extended address most significant bits. Further extends the address specified by the address and extended address fields by 2 bits. This field provides 34-, 50-, and 66-bit addresses to be specified in a packet with the xamsbs as the most significant bits in the address.
address	Least significant 29 bits (bits [0-28] of byte address [0-31]) of the double-word physical address

Table 4-7. Read Size (rdsize) Definitions

wdptr	rdsize	Number of Bytes	Byte Lanes	Comment
0b0	0b0000	1	0b10000000	I/O read only
0b0	0b0001	1	0b01000000	I/O read only
0b0	0b0010	1	0b00100000	I/O read only
0b0	0b0011	1	0b00010000	I/O read only
0b1	0b0000	1	0b00001000	I/O read only
0b1	0b0001	1	0b00000100	I/O read only
0b1	0b0010	1	0b00000010	I/O read only
0b1	0b0011	1	0b00000001	I/O read only
0b0	0b0100	2	0b11000000	I/O read only
0b0	0b0101	3	0b11100000	I/O read only
0b0	0b0110	2	0b00110000	I/O read only
0b0	0b0111	5	0b11111000	I/O read only
0b1	0b0100	2	0b00001100	I/O read only
0b1	0b0101	3	0b00000111	I/O read only
0b1	0b0110	2	0b00000011	I/O read only
0b1	0b0111	5	0b00011111	I/O read only
0b0	0b1000	4	0b11110000	I/O read only
0b1	0b1000	4	0b00001111	I/O read only
0b0	0b1001	6	0b11111100	I/O read only
0b1	0b1001	6	0b00111111	I/O read only
0b0	0b1010	7	0b11111110	I/O read only
0b1	0b1010	7	0b01111111	I/O read only
0b0	0b1011	8	0b11111111	I/O read only
0b1	0b1011	16		I/O read only
0b0	0b1100	32		
0b1	0b1100	64		
0b0-1	0b1101 0b1111			Reserved

Table 4-8. Write Size (wrsize) Definitions

wdptr	wrsize	Number of Bytes	Byte Lanes
0b0	0b0000	1	0b10000000
0b0	0b0001	1	0b01000000
0b0	0b0010	1	0b00100000
0b0	0b0011	1	0b00010000
0b1	0b0000	1	0b00001000
0b1	0b0001	1	0b00000100
0b1	0b0010	1	0b00000010
0b1	0b0011	1	0b00000001
0b0	0b0100	2	0b11000000
0b0	0b0101	3	0b11100000
0b0	0b0110	2	0b00110000
0b0	0b0111	5	0b11111000
0b1	0b0100	2	0b00001100
0b1	0b0101	3	0b00000111
0b1	0b0110	2	0b00000011
0b1	0b0111	5	0b00011111
0b0	0b1000	4	0b11110000
0b1	0b1000	4	0b00001111
0b0	0b1001	6	0b11111100
0b1	0b1001	6	0b00111111
0b0	0b1010	7	0b11111110
0b1	0b1010	7	0b01111111
0b0	0b1011	8	0b11111111
0b1	0b1011	16 maximum	
0b0	0b1100	32 maximum	
0b1	0b1100	64 maximum	
0b0-1	0b1101-1111	Reserved	

4.2.4 Type 0 Packet Format (Implementation-Defined)

The type 0 packet format is reserved for implementation-defined functions such as flow control.

4.2.5 Type 1 Packet Format (Intervention-Request Class)

Type 1 request packets never include data. They are the only request types that can cause an intervention, so the secondary domain, secondary ID, and secondary transaction ID fields are required. The total number of bits available for the secondary domain and secondary ID fields (shown in Figure 4-1) is determined by the size of the transport field defined in the appropriate transport layer specification, so the size (labeled *m* and *n*, respectively) of these fields are not specified. The division of the bits between the logical coherence domain and device ID fields is determined by the specific application. For example, an 8 bit transport field allows 16 coherence domains of 16 participants.

The type 1 packet format is used for the READ_OWNER, READ_TO_OWN_OWNER, and IO_READ_OWNER transactions that are specified in the transaction sub-field column defined in Table 4-9. Type 1 packets are issued only by a home memory controller to allow the third party intervention data transfer.

Definitions and encodings of fields specific to type 1 packets are displayed in Table 4-9. Fields that are not specific to type 1 packets are described in Table 4-6.

Table 4-9. Specific Field Definitions and Encodings for Type 1 Packets

Field	Encoding	Sub-Field	Definition
secID	—		Original requestor’s, or secondary, ID for intervention
secTID	—		Original requestor’s, or secondary, transaction ID for intervention
sec_domain	—		Original requestor’s, or secondary, domain for intervention
transaction	0b0000	READ_OWNER	
	0b0001	READ_TO_OWN_OWNER	
	0b0010	IO_READ_OWNER	
	0b0011–1111	Reserved	

Figure 4-1 displays a type 1 packet with all its fields. The field value 0b0001 in Figure 4-1 specifies that the packet format is of type 1.

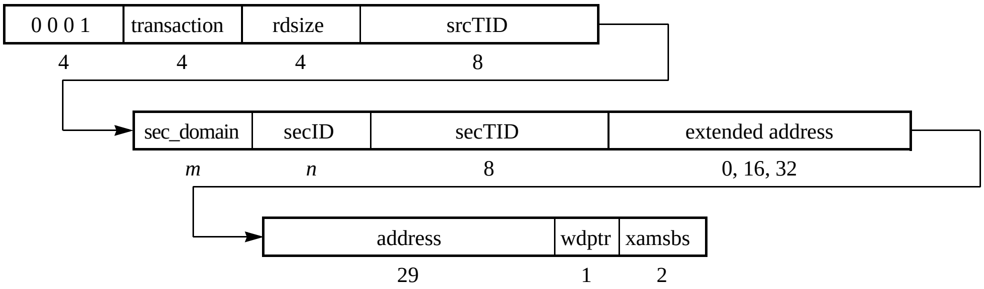


Figure 4-1. Type 1 Packet Bit Stream Format

4.2.6 Type 2 Packet Format (Request Class)

Type 2 request packets never include data. They cannot cause an intervention so the secondary domain and ID fields specified in the intervention-request format are not required. This format is used for the READ_HOME, IREAD_HOME, READ_TO_OWN_HOME, IO_READ_HOME, DKILL_HOME, DKILL_SHARER, IKILL_HOME, IKILL_SHARER, TLBIE, and TLBSYNC transactions as specified in the transaction field defined in Table 4-10. Type 2 packets for READ_HOME, IREAD_HOME, READ_TO_OWN_HOME, IO_READ_HOME, FLUSH without data, DKILL_HOME, and IKILL_HOME transactions are issued to home memory by a processing element. DKILL_SHARER and IKILL_SHARER transactions are issued by a home memory to the sharers of a coherence granule. DKILL_HOME, DKILL_SHARER, IKILL_HOME, IKILL_SHARER, FLUSH without data, and TLBIE are address-only transactions so the rdsize and wdptr fields are ignored and shall be set to logic 0. TLBSYNC is a transaction-type-only transaction so both the address, xamsbs, rdsize, and wdptr fields shall be set to logic 0.

The transaction field encodings for type 2 packets are displayed in Table 4-10. Fields that are not specific to type 2 packets are described in Table 4-6.

Table 4-10. Transaction Field Encodings for Type 2 Packets

Encoding	Transaction Field
0b0000	READ_HOME
0b0001	READ_TO_OWN_HOME
0b0010	IO_READ_HOME
0b0011	DKILL_HOME
0b0100	Reserved
0b0101	IKILL_HOME
0b0110	TLBIE
0b0111	TLBSYNC
0b1000	IREAD_HOME
0b1001	FLUSH without data
0b1010	IKILL_SHARER
0b1011	DKILL_SHARER
0b1100–1111	Reserved

Figure 4-2 displays a type 2 packet with all its fields. The field value 0b0010 in Figure 4-2 specifies that the packet format is of type 2.

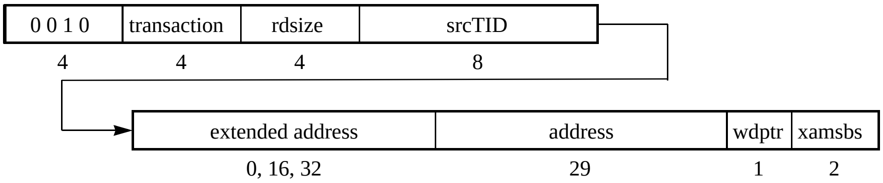


Figure 4-2. Type 2 Packet Bit Stream Format

4.2.7 Type 3–4 Packet Formats (Reserved)

The type 3–4 packet formats are reserved.

4.2.8 Type 5 Packet Format (Write Class)

Type 5 packets always contain data. A data payload that consists of a single double-word or less has sizing information as defined in Table 4-8. The wrsize field specifies the maximum size of the data payload for multiple double-word transactions. The FLUSH with data and CASTOUT transactions use type 5 packets as defined in Table 4-11. Note that type 5 transactions always contain data.

Fields that are not specific to type 5 packets are described in Table 4-6.

Table 4-11. Transaction Field Encodings for Type 5 Packets

Encoding	Transaction Field
0b0000	CASTOUT
0b0001	FLUSH with data
0b0010–1111	Reserved

Figure 4-3 displays a type 5 packet with all its fields. The field value 0b0101 in Figure 4-3 specifies that the packet format is of type 5.

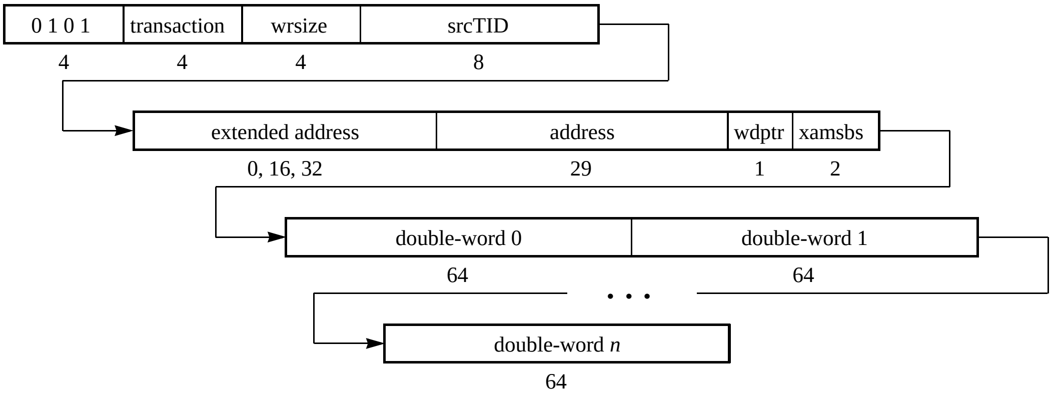


Figure 4-3. Type 5 Packet Bit Stream Format

4.2.9 Type 6–11 Packet Formats (Reserved)

The type 6–11 packet formats are reserved.

4.3 Response Packet Formats

A response transaction is issued by a processing element when it has completed a request made by a remote processing element. Response packets are always directed and are transmitted in the same way as request packets. Currently two response packet format types exist, as shown in Table 4-12.

Table 4-12. Request Packet Type to Transaction Type Cross Reference

Request Packet Format Type	Transaction Type	Definition	Document Section No.
Type 12	—	Reserved	Section 4.3.2
Type 13	RESPONSE	Issued by a processing element when it completes a request by a remote element.	Section 4.3.3
Type 14	—	Reserved	Section 4.3.4
Type 15	Implementation-defined	Defined by the device implementation	Section 4.3.5

4.3.1 Field Definitions for All Response Packet Formats

The field definitions in Table 4-13 apply to more than one of the response packet formats.

Table 4-13. Field Definitions and Encodings for All Response Packets

Field	Encoding	Sub-Field	Definition
transaction	0b0000		RESPONSE transaction with no data payload
	0b0001–0111		Reserved
	0b1000		RESPONSE transaction with data payload
	0b1001–1111		Reserved
targetTID	—		The corresponding request packet’s transaction ID

Table 4-13. Field Definitions and Encodings for All Response Packets (Continued)

status	Type of status and encoding		
	0b0000	DONE	Requested transaction has been successfully completed
	0b0001	DATA_ONLY	This is a data only response
	0b0010	NOT_OWNER	Not owner of requested coherence granule
	0b0011	RETRY	Requested transaction is not accepted; must retry the request
	0b0100	INTERVENTION	Update home memory with intervention data
	0b0101	DONE_INTERVENTION	Done for a transaction that resulted in an intervention
	0b0110	—	Reserved
	0b0111	ERROR	Unrecoverable error detected
	0b1000–1011	—	Reserved
	0b1100–1111	Implementation	Implementation defined—Can be used for additional information such as an error code

4.3.2 Type 12 Packet Format (Reserved)

The type 12 packet format is reserved.

4.3.3 Type 13 Packet Format (Response Class)

The type 13 packet format returns status, data (if required), and the requestor’s transaction ID. A RESPONSE packet with an “ERROR” status or a response that is not expected to have a data payload never has a data payload. The type 13 format is used for response packets to all request transactions.

Note that type 13 packets do not have any special fields.

Figure 4-4 illustrates the format and fields of type 13 packets. The field value 0b1101 in Figure 4-4 specifies that the packet format is of type 13.

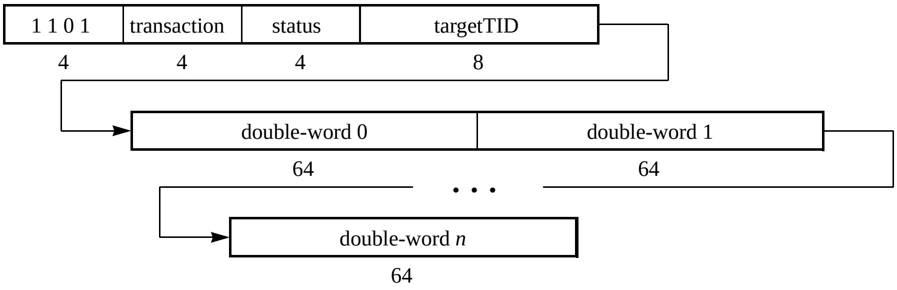


Figure 4-4. Type 13 Packet Bit Stream Format

4.3.4 Type 14 Packet Format (Reserved)

The type 14 packet format is reserved.

4.3.5 Type 15 Packet Format (Implementation-Defined)

The type 15 packet format is reserved for implementation-defined functions such as flow control.

Blank page

Chapter 5 Globally Shared Memory Registers

5.1 Introduction

This chapter describes the visible register set that allows an external processing element to determine the capabilities, configuration, and status of a processing element using this logical specification. This chapter only describes registers or register bits defined by this specification. Refer to the other RapidIO logical, transport, and physical specifications of interest to determine a complete list of registers and bit definitions. All registers are 32-bits and aligned to a 32-bit boundary.

5.2 Register Summary

Table 5-1 shows the register map for this RapidIO specification. These capability registers (CARs) and command and status registers (CSRs) can be accessed using the *RapidIO Part 1: Input/Output Logical Specification* maintenance operations. Any register offsets not defined are considered reserved for this specification unless otherwise stated. Other registers required for a processing element are defined in other applicable RapidIO specifications and by the requirements of the specific device and are beyond the scope of this specification. Read and write accesses to reserved register offsets shall terminate normally and not cause an error condition in the target device. Writes to CAR (read-only) space shall terminate normally and not cause an error condition in the target device.

Register bits defined as reserved are considered reserved for this specification only. Bits that are reserved in this specification may be defined in another RapidIO specification.

Table 5-1. GSM Register Map

Configuration Space Byte Offset	Register Name
0x0-14	Reserved
0x18	Source Operations CAR
0x1C	Destination Operations CAR
0x20-FC	Reserved

Table 5-1. GSM Register Map (Continued)

Configuration Space Byte Offset	Register Name
0x100-FFFC	Extended Features Space
0x10000-FFFFFC	Implementation-defined Space

5.3 Reserved Register, Bit and Bit Field Value Behavior

Table 5-2 describes the required behavior for accesses to reserved register bits and reserved registers for the RapidIO register space,

Table 5-2. Configuration Space Reserved Access Behavior

Byte Offset	Space Name	Item	Initiator behavior	Target behavior
0x0-3C	Capability Register Space (CAR Space - this space is read-only)	Reserved bit	read - ignore returned value[1]	read - return logic 0
			write -	write - ignored
		Implementation-defined bit	read - ignore returned value unless implementation-defined function understood	read - return implementation-defined value
			write -	write - ignored
		Reserved register	read - ignore returned value	read - return logic 0s
			write -	write - ignored
0x40-FC	Command and Status Register Space (CSR Space)	Reserved bit	read - ignore returned value	read - return logic 0
			write - preserve current value[2]	write - ignored
		Implementation-defined bit	read - ignore returned value unless implementation-defined function understood	read - return implementation-defined value
			write - preserve current value if implementation-defined function not understood	write - implementation-defined
		Reserved register	read - ignore returned value	read - return logic 0s
			write -	write - ignored

Table 5-2. Configuration Space Reserved Access Behavior (Continued)

Byte Offset	Space Name	Item	Initiator behavior	Target behavior
0x100-FFFC	Extended Features Space	Reserved bit	read - ignore returned value	read - return logic 0
			write - preserve current value	write - ignored
		Implementation-defined bit	read - ignore returned value unless implementation-defined function understood	read - return implementation-defined value
			write - preserve current value if implementation-defined function not understood	write - implementation-defined
		Reserved register	read - ignore returned value	read - return logic 0s
			write -	write - ignored
0x10000-FFFFFC	Implementation-defined Space	Reserved bit and register	All behavior implementation-defined	

[1] Do not depend on reserved bits being a particular value; use appropriate masks to extract defined bits from the read value.

[2] All register writes shall be in the form: read the register to obtain the values of all reserved bits, merge in the desired values for defined bits to be modified, and write the register, thus preserving the value of all reserved bits.

When a writable bit field is set to a reserved value, device behavior is implementation specific.

5.4 Capability Registers (CARs)

Every processing element shall contain a set of registers that allows an external processing element to determine its capabilities using the I/O logical maintenance read operation. All registers are 32 bits wide and are organized and accessed in 32-bit (4 byte) quantities, although some processing elements may optionally allow larger accesses. CARs are read-only. Refer to Table 5-2 for the required behavior for accesses to reserved registers and register bits.

CARs are big-endian with bit 0 the most significant bit.

5.4.1 Source Operations CAR (Configuration Space Offset 0x18)

This register defines the set of RapidIO GSM logical operations that can be issued by this processing element; see Table 5-3. It is assumed that a processing element can generate I/O logical maintenance read and write requests if it is required to access CARs and CSRs in other processing elements. RapidIO switches shall be able to route any packet.

Table 5-3. Bit Settings for Source Operations CAR

Bit	Field Name	Description
0	Read	PE can support a read operation
1	Instruction read	PE can support an instruction read operation
2	Read-for-ownership	PE can support a read-for-ownership operation
3	Data cache invalidate	PE can support a data cache invalidate operation
4	Castout	PE can support a castout operation
5	Data cache flush	PE can support a data cache flush operation
6	I/O read	PE can support an I/O read operation
7	Instruction cache invalidate	PE can support an instruction cache invalidate operation
8	TLB invalidate-entry	PE can support a TLB invalidate-entry operation
9	TLB invalidate-entry sync	PE can support a TLB invalidate-entry sync operation
10–13	—	Reserved
14–15	Implementation Defined	Defined by the device implementation
16–29	—	Reserved
30–31	Implementation Defined	Defined by the device implementation

5.4.2 Destination Operations CAR (Configuration Space Offset 0x1C)

This register defines the set of RapidIO GSM operations that can be supported by this processing element; see Table 5-4. It is required that all processing elements can respond to I/O logical maintenance read and write requests in order to access these registers. The Destination Operations CAR is applicable for end point devices only. RapidIO switches shall be able to route any packet.

Table 5-4. Bit Settings for Destination Operations CAR

Bit	Field Name	Description
0	Read	PE can support a read operation
1	Instruction read	PE can support an instruction read operation
2	Read-for-ownership	PE can support a read-for-ownership operation
3	Data cache invalidate	PE can support a data cache invalidate operation
4	Castout	PE can support a castout operation
5	Data cache flush	PE can support a flush operation
6	I/O read	PE can support an I/O read operation
7	Instruction cache invalidate	PE can support an instruction cache invalidate operation
8	TLB invalidate-entry	PE can support a TLB invalidate-entry operation
9	TLB invalidate-entry sync	PE can support a TLB invalidate-entry sync operation
10–13	—	Reserved
14-15	Implementation Defined	Defined by the device implementation
16-29	—	Reserved
30-31	Implementation Defined	Defined by the device implementation

5.5 Command and Status Registers (CSRs)

The RapidIO Globally Shared Memory Logical Specification does not define any command and status registers (CSRs).

Chapter 6 Communication Protocols

6.1 Introduction

This chapter contains the RapidIO globally shared memory (GSM) communications protocol definitions. Three state machines are required for a processing element on the RapidIO interface: one for local system accesses to local and remote space, one for remote accesses to local space, and one for handling responses made by the remote system to requests from the local system. The protocols are documented as pseudo-code partitioned by operation type. The RapidIO protocols as defined here assume a directory state definition that uses a modified bit with the local processor always sharing as described in Chapter 2, “System Models.” The protocols can be easily modified to use an alternate directory scheme that allows breaking the SHARED state into a REMOTE_SHARED and a REMOTE_AND_LOCAL_SHARED state pair.

Similarly, it may be desirable for an implementation to have an UNOWNED state instead of defaulting to LOCAL_SHARED or LOCAL_MODIFIED. These optimizations only affect the RapidIO transaction issuing behavior within a processing element, not the globally shared memory protocol itself. This flexibility allows a variety of local processor cache state coherence definitions such as MSI or MESI.

Some designs may not have a source of local system requests, for example, the memory only processing element described in Section 2.2.3, “Memory-Only Processing Element Model”. The protocols for these devices are much less complicated, only requiring the external request state machine and a portion of the response state machine. Similarly, a design may not have a local memory controller, which is also a much less complicated device, requiring only a portion of the internal request and response state machines. The protocols assume a processor element and memory processing element as described in Figure 2-2.

6.2 Definitions

The general definitions of Section 6.2.1 apply throughout the protocol, and the requests and responses of state machines are defined in Section 6.2.2, “Request and Response Definitions.”

6.2.1 General Definitions

address_collisionAn address match between the new request and an address currently being serviced by the state machines or some other address-based internal hazard. This frequently causes a retry of the new request.

assign_entry() Assign resources (such as a queue entry) to service a request, mark the address as able to participate in address collision detection (if appropriate), and assign a transaction ID

data Any data associated with the transaction; this field is frequently null

directory_state The memory directory state for the address being serviced

error() Signal an error (usually through an interrupt structure) to software, usually to indicate a coherence violation problem

free_entry() Release all resources assigned to this transaction, remove it from address collision detection, and deallocate the transaction ID

local Memory local to the processing element

local_request(m,n,...)A local request to a local processor caused by an incoming external request that requires a snoop of the processor’s caches

local_response(m,n,.)A local response to a local request; usually indicates the cache state for the requesting processor to mark the requested data

LOCAL_RTYPEThis is the response from the local agent to the local processor in response to a local request.

LOCAL_TTYPEThis is the transaction type for a request passed from the RapidIO interconnect to a local device.

(mask <= (mask ~= received_srcid))
“Assign the mask field to the old mask field with the received ID bit cleared.” This result is generated when a response to a multicast is received and it is not the last one expected.

((mask ~= (my_id OR received_id)) == 0)
“The mask field not including my ID or the received ID equals 0.” This result indicates that we have received all of the expected responses to a multicast request.

(mask ~= my_id)“The sharing mask not including my ID.” This result is used for multicast operations where the requestor is in the sharing list but does not need to be included in the multicast transaction because it is the source of the transaction.

(mask <= (participant_list ~= my_id))
“The sharing mask includes all participants except my ID.” This result is used for the IKILL operation, which does not

use the memory directory information.

(mask <= (participant_list ~= (received_srcid AND my_id)))
“The sharing mask includes all participants except the requestor’s and my IDs.” This result is used for the IKILL operation, which does not use the memory directory information.

(mask == received_srcid)
“The sharing mask only includes the requestor’s ID.” This result is used for the DKILL operation to detect a write-hit-on-shared case where the requestor has the only remote copy of the coherence granule.

original_srcid The ID of the initial requestor for a transaction, saved in the state associated with the transaction ID

received_data The response contained data

received_data_only_message
Flag set by set_received_data_only_message()

received_done_message
Flag set by set_received_done_message()

remote_request(m,n,...)
Make a request to the interconnect fabric

remote_response(m,n,...)
Send a response to the interconnect fabric

RESPONSE_TTYPE
This is the RapidIO transaction type for a response to a request

return_data() Return data to the local requesting processor, either from memory or from a interconnect fabric buffer; the source can be determined from the context

secondary_id The third party identifier for intervention responses; the processing element ID concatenated with the processing element domain.

set_received_data_only_message()
Remember that a DATA_ONLY response was received for this transaction ID

set_received_done_message()
Remember that a DONE response was received for this transaction ID

source_id The source device identifier; the processing element ID concatenated with the processing element domain

target_id The destination device identifier; the processing element ID concatenated with the processing element domain

TRANSACTIONThe RapidIO transaction type code for the request

update_memory()Write memory with data received from a response

update_state(m,n,...)Modify the memory directory state to reflect the new system status

6.2.2 Request and Response Definitions

Following are the formats used in the pseudocode to describe request and response transactions sent between processing elements and the formats of local requests and responses between the cache coherence controller and the local cache hierarchy and memory controllers.

6.2.2.1 System Request

The system request format is:

```
remote_request(TRANSACTION, target_id, source_id, secondary_id, data)
```

which describes the necessary RapidIO request to implement the protocol.

6.2.2.2 Local Request

The local request format is:

```
local_request(LOCAL_TTYPE)
```

that is the necessary local processor request to implement the protocol; the pseudocode assumes a generic local bus. A local request also examines the remote cache as part of the processing element’s caching hierarchy. The local transactions are defined as:

DKILL Causes the processor to transition the coherence granule to invalid regardless of the current state; data is not pushed if current state is modified

IKILL Causes the processor to invalidate the coherence granule in the instruction cache

READ Causes the processor to transition the coherence granule to shared and push data if necessary

READ_LATESTCauses the processor to push data if modified but not transition the cache state

READ_TO_OWNCauses the processor to transition the coherence granule to invalid and push data

TLBIE Causes the processor to invalidate the specified translation look-aside buffer entry

TLBSYNC Causes the processor to indicate when all outstanding TLBIEs have completed

6.2.2.3 System Response

The system response format is:

```
remote_response(RESPONSE_TTYPE, target_id, source_id, data (opt.))
```

which is the proper response to implement the protocol.

6.2.2.4 Local Response

The local response format is:

```
local_response(LOCAL_RTYPE)
```

In general, a transaction ID (TID) is associated with each device ID in order to uniquely identify a request. This TID is frequently a queue index in the source processing element. These TIDs are not explicitly called out in the pseudocode below. The local responses are defined as:

EXCLUSIVE The processor has exclusive access to the coherence granule

OK The transaction requested by the processor has or will complete properly

RETRY Causes the processor to re-issue the transaction; this response may cause a local bus spin loop until the protocol allows a different response

SHARED The processor has a shared copy of the coherence granule

6.3 Operation to Protocol Cross Reference

Table 6-1 contains a cross reference of the operations defined in the *RapidIO Interconnect Globally Shared Memory Logical Specification* and their system usage.

Table 6-1. Operation to Protocol Cross Reference

Operations	Protocol
Read	Section 6.4
Instruction read	Section 6.4
Read for ownership	Section 6.6
Data cache invalidate	Section 6.7
Instruction cache invalidate	Section 6.7
Castout	Section 6.8
TLB invalidate entry	Section 6.9
TLB invalidate entry synchronize	Section 6.9
Data cache flush	Section 6.10
I/O read	Section 6.11

6.4 Read Operations

This operation is a coherent data cache read; refer to the description in Section 3.3.1.

6.4.1 Internal Request State Machine

This state machine handles requests to both local and 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
elseif (local)                                          // our local memory
            switch (directory_state)
            case LOCAL_MODIFIED:                        // local modified is OK if we default
                                                        // local memory to owned
                        local_response(EXCLUSIVE);
                        return_data();
            case LOCAL_SHARED,                          // local, owned by memory
            case SHARED:                                // shared local and remote
                        local_response(SHARED);
                        return_data();                  // keep directory state
                                                        // the way it was
            case REMOTE_MODIFIED:
                        local_response(SHARED);
                        assign_entry();                 // this means to assign
                                                        // a transaction ID,
                                                        // usually a queue entry
                        remote_request(READ_OWNER, mask_id, my_id, my_id);
            default:
                        error();
else                                                    // remote - we've got to go
                                                        // to another processing element
            assign_entry();
            local_response(RETRY);                      // can't guarantee data before a
                                                        // snoop yet
            remote_request(READ_HOME, mem_id, my_id);
endif;
```

6.4.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)                  // original requestor is home memory
            switch(remote_response)                     // matches my_id only for
                                                        // REMOTE_MODIFIED case
            case INTERVENTION:
                        update_memory();
                        update_state(SHARED, original_srcid);
                        return_data();
                        free_entry();
            case NOT_OWNER,                             // due to address collision or
            case RETRY:                                 // passing requests
                        switch (directory_state)
                        case LOCAL_MODIFIED:
                                    local_response(EXCLUSIVE);
                                                        // when processor re-requests
                                    return_data();
                                    free_entry();
                        case LOCAL_SHARED:
```

```
                        local_response(SHARED);
                                                // when processor re-requests
                        return_data();
                        free_entry();
                case REMOTE_MODIFIED:           // mask_id must match received_srcid

                                                //or error; spin or wait for castout
                        remote_request(READ_OWNER, received_srcid,
                                my_id, my_id);
                default:
                        error();
        default
                error();
elseif(my_id == mem_id ~== original_id          // i’m home memory working for
                                                // 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();
else                                            // my_id ~= mem_id - I’m
                                                // requesting a remote
                                                // memory location
        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(READ_HOME, received_srcid, my_id);
        default
                error();
endif;
```

6.4.3 External Request State Machine

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

```
if (address_collision)                                  // use collision tables in
                                                        // Chapter 7, “Address Collision Resolution
Tables”
elseif (READ_HOME)                                      // remote request to our local memory
        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();
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;
```

6.5 Instruction Read Operations

This operation is a partially coherent instruction cache read; refer to the description in Section 3.3.2.

6.5.1 Internal Request State Machine

This state machine handles requests to both local and 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
elseif (local)                                          // our local memory
            switch (directory_state)
            case LOCAL_MODIFIED:                        // local modified is OK if we default
                                                        // local memory to owned
                        local_response(EXCLUSIVE);
                        return_data();
            case LOCAL_SHARED,                          // local, owned by memory
            case SHARED:                                // shared local and remote
                        local_response(SHARED);
                        return_data();                  // keep directory state the way it was
            case REMOTE_MODIFIED:
                        local_response(SHARED);
                        assign_entry();                 // this means to assign a transaction
                                                        // ID, usually a queue entry
                        remote_request(READ_OWNER, mask_id, my_id, my_id);
            default:
                        error();
else                                                    // remote - we've got to go
                                                        // to another processing element
            assign_entry();
            local_response(RETRY);
                                                        // can't guarantee data before a
                                                        // snoop yet
            remote_request(IREAD_HOME, mem_id, my_id);
endif;
```

6.5.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)                  // original requestor is home memory
            error();
elseif(my_id == mem_id ~== original_id)                 // i'm home memory working for 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();
else                                        // my_id ~= mem_id - I’m requesting
                                            // a remote memory location
    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();
endif;
```

6.5.3 External Request State Machine

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

```
if (address_collision)                      // use collision tables in
                                            // Chapter 7, “Address Collision Resolution
Tables”
elseif(IREAD_HOME)                          // remote request to our local memory
    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;
```

6.6 Read for Ownership Operations

This is the coherent cache store miss operation.

6.6.1 Internal Request State Machine

This state machine handles requests to both local and 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
elseif (local)                                      // our local memory
        switch (directory_state
        case LOCAL_MODIFIED,                        // local modified is OK if we
                                                    // default memory to owned locally
        case LOCAL_SHARED:
                local_response(EXCLUSIVE);           // give ownership to processor
                return_data();
                if (directory_state == LOCAL_SHARED)
                        update_state(LOCAL_MODIFIED)
                endif;
        case REMOTE_MODIFIED:                       // owned by another, get a copy
                                                    // and ownership
                assign_entry();
                local_response(RETRY);              // retry
                remote_request(READ_TO_OWN_OWNER, mask_id, my_id, my_id);
        case SHARED:                                // invalidate the sharing list
                assign_entry();
                local_response(RETRY);   // retry
                remote_request(DKILL_SHARER, (mask ~= my_id), my_id, my_id);
        default:
                error();
else                                                // remote - we’ve got to go to another
                                                    // processing element
        assign_entry();
        local_response(RETRY);
        remote_request(READ_TO_OWN_HOME, mem_id, my_id);
endif;
```

6.6.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)              // original requestor is home memory
        switch (received_response)
        case DONE:                                  // SHARED, so invalidate case
                if ((mask ~= (my_id OR received_id)) == 0)
                                                    // this is the last DONE
                        local_response(EXCLUSIVE);
                        return_data();
                        update_state(LOCAL_MODIFIED);
                        free_entry();
                else
                        mask <= (mask ~= received_srcid);
                                                    // flip the responder’s shared
                                                    // bit and wait for next DONE
                endif;
        case NOT_OWNER:                             // due to address collision with
                                                    // CASTOUT or FLUSH
                switch(directory_state)
                case LOCAL_MODIFIED,:
                        local_response(EXCLUSIVE);
                        return_data();
                        free_entry();
                case LOCAL_SHARED:
                        local_response(EXCLUSIVE);
                        return_data();
                        update_state(LOCAL_MODIFIED);
                        free_entry();
                case REMOTE_MODIFIED:
                                                    // spin or wait for castout
                        remote_request(READ_TO_OWN_OWNER, mask_id,
                            my_id, my_id);
                default:
                        error();
        case INTERVENTION:                  // remotely owned
                local_response(EXCLUSIVE);
                return_data();
                update_state(LOCAL_MODIFIED);
                free_entry();
        case RETRY:
                switch (directory_state)
                case LOCAL_MODIFIED:
                        local_response(EXCLUSIVE);
                        return_data();
                        free_entry();
                case LOCAL_SHARED:
                        local_response(EXCLUSIVE);
                        return_data();
                        update_state(LOCAL_MODIFIED);
                        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();
elseif (my_id == mem_id ~= original_srcid)
                                                    // i’m home memory working
                                                    // for 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();
else                                     // my_id ~= mem_id - I’m requesting
                                         // a remote memory location
        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();
endif;
```

6.6.3 External Request State Machine

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

```
if (address_collision)                          // use collision tables
                                                // in Chapter 7, “Address Collision Resolution
Tables”
elseif (READ_TO_OWN_HOME)                       // remote request to our local memory
        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();
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();
endif;
```

6.7 Data Cache and Instruction Cache Invalidate Operations

This operation is used with coherent cache store-hit-on-shared, cache operations; refer to the description in Section 3.3.4.

6.7.1 Internal Request State Machine

This state machine handles requests to both local and 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
elseif (local)                              // our local memory and we won
        if (DKILL)                          // DKILL checks the directory
                switch (directory_state)
                case LOCAL_MODIFIED,        // local modified is OK if we default
                                            // memory to owned locally
                case LOCAL_SHARED:
                        local_response(EXCLUSIVE);
                        if (LOCAL_SHARED)
                                update_state(LOCAL_MODIFIED, my_id);
                        endif;
                case REMOTE_MODIFIED:       // cache paradox; DKILL is
                                            // write-hit-on-shared
                        error();
                case SHARED:
                        local_response(RETRY);
                        assign_entry();     // Multicast if possible otherwise
                                            // issue direct to each sharer
                        remote_request(DKILL_SHARER, (mask ~= my_id), my_id);
                default:
                        error();
        else                                // IKILL always goes to everyone
                remote_request(IKILL_SHARER,
                        (mask <= (participant_list ~= my_id)), my_id);
        endif;
else                                        // remote - we’ve got to go to another
                                            // processing element
        assign_entry();
        local_response(RETRY);
        remote_request({DKILL_HOME, IKILL_HOME}, mem_id, my_id);
endif;
```

6.7.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)      // original requestor is home memory
        switch (received_response)
        case DONE:                          // shared cases
                if ((mask ~= (my_id OR received_id)) == 0)
                                            // this is the last DONE
                        if (DKILL)          // don’t update state for IKILLs
                                update_state(LOCAL_MODIFIED);
                        endif;
                        free_entry();
                else
```

```
                        mask <= (mask ~= received_srcid);
                                                // flip the responder’s shared bit and
                endif;                          // wait for next DONE
        case RETRY:
                remote_request({DKILL_SHARER, IKILL_SHARER}, received_srcid,
                        my_id);                 // retry the transaction
        default:
                error();
elseif (my_id == mem_id ~= original_srcid)
                                                // i’m home memory working
                                                // for 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
                        if (DKILL)              // don’t update state for IKILLs
                                update_state(REMOTE_MODIFIED, original_srcid);
                        endif;
                        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();
else                                            // my_id ~= mem_id - I’m requesting
                                                // a remote memory location
        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();
endif;
```

6.7.3 External Request State Machine

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

```
if (address_collision)                          // use collision tables in
                                                // Chapter 7, “Address Collision Resolution
Tables”
elseif (DKILL_HOME || IKILL_HOME)               // remote request to our local memory
        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
```

```
                                if (DKILL)// don’t update state for (IKILLs)
                                        update_state(REMOTE_MODIFIED,
received_srcid);
                                endif;
                                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);
else                                            // DKILL_SHARER or IKILL_SHARER 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;
```

6.8 Castout Operations

This operation is used to return ownership of a coherence granule to home memory, leaving it invalid in the cache; refer to the description in Section 3.3.5.

6.8.1 Internal Request State Machine

A castout is always done to remote memory space. A castout may require local activity to flush all caches in the hierarchy.

```
if (local)                                          // our local memory
        switch (directory_state)
        case LOCAL_MODIFIED:                        // if the processor is doing a castout
                                                    // this is the only legal state
                local_response(OK);
                update_memory();
                update_state(LOCAL_SHARED);
        default:
                error();
else                                                // remote - we’ve got to go to another
                                                    // processing element
        assign_entry();
        local_response(OK);
        remote_request(CASTOUT, mem_id, my_id, data);
endif;
```

6.8.2 Response State Machine

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

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

6.8.3 External Request State Machine

This state machine handles requests from the system to the local memory or the local system. 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();
```

6.9 TLB Invalidate Entry, TLB Invalidate Entry Synchronize Operations

These operations are used for software coherence management of the TLBs; refer to the descriptions in Section 3.3.6 and Section 3.3.7.

6.9.1 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;
```

6.9.2 Response State Machine

This state machine handles responses to requests made to the RapidIO interconnect on behalf of the local system. 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();
```

6.9.3 External Request State Machine

This state machine handles requests from the system to the local memory or the local system. 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();
```

6.10 Data Cache Flush Operations

This operation returns ownership of a coherence granule to home memory and performs a coherent write; refer to the description in Section 3.3.9.

6.10.1 Internal Request State Machine

This state machine handles requests to both local and 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
elseif (local) // our local memory
            switch (directory_state)
            case LOCAL_MODIFIED,
            case LOCAL_SHARED:
                        local_response(OK);
                        update_memory();
            case REMOTE_MODIFIED:
                        assign_entry();
                        remote_request(READ_TO_OWN_OWNER, mask_id, my_id, my_id);
            case SHARED:
                        assign_entry();
                        remote_request(DKILL_SHARER, (mask ~= my_id), my_id);
            default:
                        error();
else                                                    // remote - we've got to go to
                                                        // another processing element
            assign_entry();
            remote_request(FLUSH, mem_id, my_id, data);
                                                        // data is optional
endif;
```

6.10.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)                  // original requestor is home memory
            switch (received_response)
            case DONE:
                        if ((mask ~= (my_id OR received_id)) == 0)
                                                        // this is the last DONE
                                    if (received_data)
                                                        // with local request or response
                                                update_memory();
                                    endif;
                                    update_state(LOCAL_SHARED);  // or LOCAL_MODIFIED
                                    local_response(OK);
                                    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:
                                    if (received_data)
                                                        // with local request from memory
```

```
                                        update_memory();
                                endif;
                                update_state(LOCAL_SHARED);  // or LOCAL_MODIFIED
                                local_response(OK);
                                free_entry();
                        case REMOTE_MODIFIED:
                                remote_request(READ_TO_OWN_OWNER,      mask_id,      my_id,
my_id);
                        default:
                                error();
                case RETRY:
                        switch (directory_state)
                        case LOCAL_MODIFIED,
                        case LOCAL_SHARED:
                                if (received_data)
                                                        // with local request
                                        update_memory();
                                                        // if there was some write data
                                endif;
                                update_state(LOCAL_SHARED);  // or LOCAL_MODIFIED
                                local_response(OK);
                                free_entry();
                        case REMOTE_MODIFIED:           // mask_id must match
                                                        // received_srcid or error
                                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();
elseif (my_id == mem_id ~= original_srcid)
                                                        // i'm home memory working for 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, my_id);
                        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, my_id);
        case SHARED:
            remote_request(DKILL_SHARER, received_srcid, my_id);
        default:
            error();
    default:
        error();
else                                    // my_id ~= mem_id - I’m requesting
                                        // a remote memory location
    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();
endif;
```

6.10.3 External Request State Machine

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

```
if (address_collision)                  // use collision table in
                                        // Chapter 7, “Address Collision Resolution
Tables”
elseif (FLUSH)                          // remote request to our local memory
    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,
                my_id);                 // secondary TID is a don’t care since data is
                                        // not forwarded to original requestor
        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;
```

6.11 I/O Read Operations

This operation is used for I/O reads of globally shared memory space; refer to the description in Section 3.3.10.

6.11.1 Internal Request State Machine

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

```
if (address_collision)                          // this is due to an external request
                                                // in progress or a cache index hazard
    local_response(RETRY);                      // from a previous request
elseif (local)                                  // our local memory
    local_response(OK);
    switch (directory_state)
    case LOCAL_MODIFIED:                        // local modified is OK if we default
                                                // local memory to owned
        local_request(READ_LATEST);
        return_data())                          // after possible push
    case LOCAL_SHARED,
    case SHARED:
        return_data();                          // keep directory state the way it was
    case REMOTE_MODIFIED:
        assign_entry();
        remote_request(IO_READ_OWNER, mask_id, my_id, my_id);
    default:
        error();
else                                            // remote - we’ve got to go to
                                                // another processing element
    assign_entry();
    local_response(OK);
    remote_request(IO_READ_HOME, mem_id, my_id);
endif;
```

6.11.2 Response State Machine

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

```
if (my_id == mem_id == original_srcid)
                                                // original requestor is home memory
    switch(remote_response)                     // matches my_id only for
                                                // REMOTE_MODIFIED case
    case INTERVENTION:
        return_data();
        free_entry();
    case NOT_OWNER,                             // due to address collision or
                                                // passing requests
    case RETRY:
        switch (directory_state)
        case LOCAL_MODIFIED:
        case LOCAL_SHARED
            return_data();
            free_entry();
        case REMOTE_MODIFIED:                   // mask_id must match received_srcid or
                                                // error; spin or wait for castout
            remote_request(IO_READ_OWNER, received_srcid, my_id,
                my_id);
        default:
            error();
    default
        error();
elseif(my_id == mem_id ~== original_id)             // i’m home memory working for 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();
else                                                // my_id ~= mem_id - I’m requesting a
                                                    // remote memory location
    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();
endif;
```

6.11.3 External Request State Machine

This state machine handles requests from the system to the local memory or the local system. 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
                                            // memory
                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;
```

Chapter 7 Address Collision Resolution Tables

7.1 Introduction

Address collisions are conflicts between incoming cache coherence requests to a processing element and outstanding cache coherence requests within it. A collision is usually due to a match between the associated addresses, but also may be because of a conflict for some internal resource such as a cache index. Within a processing element, actions taken in response to an address collision vary depending upon the outstanding request and the incoming request. These actions are described in Table 7-1 through Table 7-17. Non-cache coherent transactions (transactions specified in other RapidIO logical specifications) do not cause address collisions.

Some of the table entries specify that an outstanding request should be canceled at the local processor and that the incoming transaction then be issued immediately to the processor. This choosing between transactions is necessary to prevent deadlock conditions between multiple processing elements vying for ownership of a coherence granule.

7.2 Resolving an Outstanding READ_HOME Transaction

Table 7-1 describes the address collision resolution for an incoming transaction that collides with an outstanding READ_HOME transaction.

Table 7-1. Address Collision Resolution for READ_HOME

Outstanding Request	Incoming Request	Resolution
READ_HOME	READ_HOME	Generate “ERROR” response
READ_HOME	IREAD_HOME	Generate “ERROR” response
READ_HOME	READ_OWNER	Generate “NOT_OWNER” response
READ_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
READ_HOME	READ_TO_OWN_OWNER	Generate “NOT_OWNER” response
READ_HOME	DKILL_HOME	Generate “ERROR” response
READ_HOME	DKILL_SHARER	If outstanding request, wait for all expected responses. If final response is “DONE”, return data if necessary and forward DKILL_SHARER to processor then generate a “DONE” response. If final response is “RETRY”, cancel the read at the processor and forward DKILL_SHARED to processor then generate a “DONE” response If no outstanding request, cancel the read at the processor and forward DKILL_SHARER to processor then generate a “DONE” response (this case should be very rare).
READ_HOME	CASTOUT	Generate “ERROR” response
READ_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_HOME	IKILL_HOME	Generate “ERROR” response
READ_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
READ_HOME	FLUSH	Generate “ERROR” response
READ_HOME	IO_READ_HOME	Generate “ERROR” response
READ_HOME	IO_READ_OWNER	Generate “NOT_OWNER” response

7.3 Resolving an Outstanding IREAD_HOME Transaction

Table 7-2 describes the address collision resolution for an incoming transaction that collides with an outstanding IREAD_HOME transaction.

Table 7-2. Address Collision Resolution for IREAD_HOME

Outstanding Request	Incoming Request	Resolution
IREAD_HOME	READ_HOME	Generate “ERROR” response
IREAD_HOME	IREAD_HOME	Generate “ERROR” response
IREAD_HOME	READ_OWNER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IREAD_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
IREAD_HOME	READ_TO_OWN_OWNER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IREAD_HOME	DKILL_HOME	Generate “ERROR” response
IREAD_HOME	DKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IREAD_HOME	CASTOUT	Generate “ERROR” response
IREAD_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IREAD_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IREAD_HOME	IKILL_HOME	Generate “ERROR” response
IREAD_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IREAD_HOME	FLUSH	Generate “ERROR” response
IREAD_HOME	IO_READ_HOME	Generate “ERROR” response
IREAD_HOME	IO_READ_OWNER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)

7.4 Resolving an Outstanding READ_OWNER Transaction

Table 7-3 describes the address collision resolution for an incoming transaction that collides with an outstanding READ_OWNER transaction.

Table 7-3. Address Collision Resolution for READ_OWNER

Outstanding Request	Incoming Request	Resolution
READ_OWNER	READ_HOME	Generate “RETRY” response
READ_OWNER	IREAD_HOME	Generate “RETRY” response
READ_OWNER	READ_OWNER	Generate “ERROR” response
READ_OWNER	READ_TO_OWN_HOME	Generate “RETRY” response
READ_OWNER	READ_TO_OWN_OWNER	Generate “ERROR” response
READ_OWNER	DKILL_HOME	Generate “RETRY” response
READ_OWNER	DKILL_SHARER	Generate “ERROR” response
READ_OWNER	CASTOUT	No collision, update directory state, generate “DONE” response (CASTOUT bypasses address collision detection)
READ_OWNER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_OWNER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_OWNER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
READ_OWNER	IKILL_SHARER	Generate “ERROR” response
READ_OWNER	FLUSH	Generate “RETRY” response
READ_OWNER	IO_READ_HOME	Generate “RETRY” response
READ_OWNER	IO_READ_OWNER	Generate “ERROR” response

7.5 Resolving an Outstanding READ_TO_OWN_HOME Transaction

Table 7-4 describes the address collision resolution for an incoming transaction that collides with an outstanding READ_TO_OWN_HOME transaction.

Table 7-4. Address Collision Resolution for READ_TO_OWN_HOME

Outstanding Request	Incoming Request	Resolution
READ_TO_OWN_HOME	READ_HOME	Generate “ERROR” response
READ_TO_OWN_HOME	IREAD_HOME	Generate “ERROR” response
READ_TO_OWN_HOME	READ_OWNER	If outstanding request, wait for all expected responses. If final response is “DONE”, return data if necessary and forward READ_OWNER to processor and generate an “DONE_INTERVENTION” with data response and a “DATA_ONLY” to originator as in Section 3.3.1. If final response is “RETRY” generate an “ERROR” response If no outstanding request generate an “NOT_OWNER” response.
READ_TO_OWN_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
READ_TO_OWN_HOME	READ_TO_OWN_OWNER	If outstanding request, wait for all expected responses. If final response is “DONE”, return data if necessary and forward READ_TO_OWN_OWNER to processor and generate an “DONE_INTERVENTION” with data response and a “DATA_ONLY” to originator as in Section 3.3.3. If final response is “RETRY” generate an “ERROR” response
READ_TO_OWN_HOME	DKILL_HOME	Generate “ERROR” response
READ_TO_OWN_HOME	DKILL_SHARER	If outstanding request, wait for all expected responses. If final response is “DONE” generate an “ERROR” response (we own the coherence granule and should never see a DKILL). If final response is “RETRY” generate a “DONE” response and continue the READ_TO_OWN_HOME. If no outstanding request generate a “DONE” response.
READ_TO_OWN_HOME	CASTOUT	Generate “ERROR” response
READ_TO_OWN_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_TO_OWN_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_TO_OWN_HOME	IKILL_HOME	Generate “ERROR” response
READ_TO_OWN_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)

Table 7-4. Address Collision Resolution for READ_TO_OWN_HOME (Continued)

Outstanding Request	Incoming Request	Resolution
READ_TO_OWN_HOME	FLUSH	If outstanding request, wait for all expected responses. If final response is "DONE", return data if necessary and forward FLUSH to processor and generate a "DONE" with data response as in Section 3.3.9. If final response is "RETRY" generate an "ERROR" response (we didn't own the data and we lost at home memory) If no outstanding request generate an "ERROR" response (we didn't own the data).
READ_TO_OWN_HOME	IO_READ_HOME	Generate "ERROR" response
READ_TO_OWN_HOME	IO_READ_OWNER	If outstanding request, wait for all expected responses. If final response is "DONE", return data if necessary and forward IO_READ_OWNER to processor then generate a "DONE" with data response, etc. as in Section 3.3.10. If final response is "RETRY" generate an "ERROR" response (we didn't own the data and we lost at home memory) If no outstanding request generate an "NOT_OWNER" response.

7.6 Resolving an Outstanding READ_TO_OWN_OWNER Transaction

Table 7-5 describes the address collision resolution for an incoming transaction that collides with an outstanding READ_TO_OWN_OWNER transaction.

Table 7-5. Address Collision Resolution for READ_TO_OWN_OWNER

Outstanding Request	Incoming Request	Resolution
READ_TO_OWN_OWNER	READ_HOME	Generate “RETRY” response
READ_TO_OWN_OWNER	IREAD_HOME	Generate “RETRY” response
READ_TO_OWN_OWNER	READ_OWNER	Generate “ERROR” response
READ_TO_OWN_OWNER	READ_TO_OWN_HOME	Generate “RETRY” response
READ_TO_OWN_OWNER	READ_TO_OWN_OWNER	Generate “ERROR” response
READ_TO_OWN_OWNER	DKILL_HOME	Generate “RETRY” response
READ_TO_OWN_OWNER	DKILL_SHARER	Generate “ERROR” response
READ_TO_OWN_OWNER	CASTOUT	No collision, update directory state, generate “DONE” response (CASTOUT bypasses address collision detection)
READ_TO_OWN_OWNER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_TO_OWN_OWNER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
READ_TO_OWN_OWNER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
READ_TO_OWN_OWNER	IKILL_SHARER	Generate “ERROR” response
READ_TO_OWN_OWNER	FLUSH	Generate “RETRY” response
READ_TO_OWN_OWNER	IO_READ_HOME	Generate “RETRY” response
READ_TO_OWN_OWNER	IO_READ_OWNER	Generate “ERROR” response

7.7 Resolving an Outstanding DKILL_HOME Transaction

Table 7-6 describes the address collision resolution for an incoming transaction that collides with an outstanding DKILL_HOME transaction.

Table 7-6. Address Collision Resolution for DKILL_HOME

Outstanding Request	Incoming Request	Resolution
DKILL_HOME	READ_HOME	Generate “ERROR” response
DKILL_HOME	IREAD_HOME	Generate “ERROR” response
DKILL_HOME	READ_OWNER	If outstanding request, wait for all expected responses. If final response is “DONE”, return data if necessary and forward READ_OWNER to processor and generate a “DONE_INTERVENTION” with data response and a “DATA_ONLY” to originator as in Section 3.3.1. If final response is “RETRY” generate an “ERROR” response (we didn’t own the data and we lost at home memory) If no outstanding request generate an “ERROR” response (we didn’t own the data).
DKILL_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
DKILL_HOME	READ_TO_OWN_OWNER	If outstanding request, wait for all expected responses. If final response is “DONE” forward READ_TO_OWN_OWNER to processor and generate a “DONE_INTERVENTION” with data response and a “DATA_ONLY” to originator as in Section 3.3.3. If final response is “RETRY” generate an “ERROR” response (we didn’t own the data and we lost at home memory) If no outstanding request generate an “ERROR” response (we didn’t own the data).
DKILL_HOME	DKILL_HOME	Generate “ERROR” response
DKILL_HOME	DKILL_SHARER	If outstanding request, wait for all expected responses. If final response is “DONE” generate an “ERROR” response (we should never see a DKILL_SHARER if we own the coherence granule). If final response is “RETRY” cancel the data cache invalidate at the processor and forward DKILL_SHARER to processor then generate a “DONE” response If no outstanding request, cancel the data cache invalidate at the processor and forward DKILL_SHARER to processor then generate a “DONE” response.
DKILL_HOME	CASTOUT	Generate “ERROR” response (cache paradox, can’t have a SHARED granule also MODIFIED in another processing element)
DKILL_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
DKILL_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
DKILL_HOME	IKILL_HOME	Generate “ERROR” response
DKILL_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)

Table 7-6. Address Collision Resolution for DKILL_HOME (Continued)

Outstanding Request	Incoming Request	Resolution
DKILL_HOME	FLUSH	Generate “ERROR” response
DKILL_HOME	IO_READ_HOME	Generate “ERROR” response
DKILL_HOME	IO_READ_OWNER	If outstanding request, wait for all expected responses. If final response is “DONE” forward IO_READ_OWNER to processor then generate a “DONE” with data response, etc. as in Section 3.3.10. If final response is “RETRY” generate an “ERROR” response (we didn’t own the data and we lost at home memory) If no outstanding request generate an “ERROR” response (we didn’t own the data).

7.8 Resolving an Outstanding DKILL_SHARER Transaction

Table 7-7 describes the address collision resolution for an incoming transaction that collides with an outstanding DKILL_SHARER transaction.

Table 7-7. Address Collision Resolution for DKILL_SHARER

Outstanding Request	Incoming Request	Resolution
DKILL_SHARER	READ_HOME	Generate “RETRY” response
DKILL_SHARER	IREAD_HOME	Generate “RETRY” response
DKILL_SHARER	READ_OWNER	Generate “ERROR” response
DKILL_SHARER	READ_TO_OWN_HOME	Generate “RETRY” response
DKILL_SHARER	READ_TO_OWN_OWNER	Generate “ERROR” response
DKILL_SHARER	DKILL_HOME	Generate “RETRY” response
DKILL_SHARER	DKILL_SHARER	Generate “ERROR” response
DKILL_SHARER	CASTOUT	Generate “ERROR” response (cache paradox, can’t have a SHARED granule also MODIFIED in another processing element)
DKILL_SHARER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
DKILL_SHARER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
DKILL_SHARER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
DKILL_SHARER	IKILL_SHARER	Generate “ERROR” response
DKILL_SHARER	FLUSH	Generate “RETRY” response
DKILL_SHARER	IO_READ_HOME	If processing element is HOME: generate a “RETRY” response If processing element is not HOME: If outstanding request, wait for all expected responses. If final response is “DONE” forward IO_READ to processor then generate a “DONE” with data response, etc. as in Section 3.3.10. If final response is “RETRY” generate an “ERROR” response (we didn’t own the data and we lost at home memory) If no outstanding request generate an “ERROR” response (we didn’t own the data).
DKILL_SHARER	IO_READ_OWNER	Generate “ERROR” response

7.9 Resolving an Outstanding IKILL_HOME Transaction

Table 7-8 describes the address collision resolution for an incoming transaction that collides with an outstanding IKILL_HOME transaction.

Table 7-8. Address Collision Resolution for IKILL_HOME

Outstanding Request	Incoming Request	Resolution
IKILL_HOME	READ_HOME	Generate “ERROR” response
IKILL_HOME	IREAD_HOME	Generate “ERROR” response
IKILL_HOME	READ_OWNER	No collision, process normally
IKILL_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
IKILL_HOME	READ_TO_OWN_OWNER	No collision, process normally
IKILL_HOME	DKILL_HOME	Generate “ERROR” response
IKILL_HOME	DKILL_SHARER	No collision, process normally
IKILL_HOME	CASTOUT	No collision, process normally
IKILL_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IKILL_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IKILL_HOME	IKILL_HOME	Generate “ERROR” response
IKILL_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IKILL_HOME	FLUSH	Generate “ERROR” response
IKILL_HOME	IO_READ_HOME	Generate “ERROR” response
IKILL_HOME	IO_READ_OWNER	No collision, process normally

7.10 Resolving an Outstanding IKILL_SHARER Transaction

Table 7-9 describes the address collision resolution for an incoming transaction that collides with an outstanding IKILL_SHARER transaction.

Table 7-9. Address Collision Resolution for IKILL_SHARER

Outstanding Request	Incoming Request	Resolution
IKILL_SHARER	READ_HOME	No collision, process normally
IKILL_SHARER	IREAD_HOME	No collision, process normally
IKILL_SHARER	READ_OWNER	Generate “ERROR” response
IKILL_SHARER	READ_TO_OWN_HOME	No collision, process normally
IKILL_SHARER	READ_TO_OWN_OWNER	Generate “ERROR” response
IKILL_SHARER	DKILL_HOME	No collision, process normally
IKILL_SHARER	DKILL_SHARER	Generate “ERROR” response
IKILL_SHARER	CASTOUT	No collision, process normally
IKILL_SHARER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IKILL_SHARER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IKILL_SHARER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
IKILL_SHARER	IKILL_SHARER	Generate “ERROR” response
IKILL_SHARER	FLUSH	No collision, process normally
IKILL_SHARER	IO_READ_HOME	If processing element is HOME: generate a “RETRY” response If processing element is not HOME: If outstanding request, wait for all expected responses. If final response is “DONE” forward IO_READ to processor then generate a “DONE” with data response, etc. as in Section 3.3.10. If final response is “RETRY” generate an “ERROR” response (we didn’t own the data and we lost at home memory) If no outstanding request generate an “ERROR” response (we didn’t own the data).
IKILL_SHARER	IO_READ_OWNER	Generate “ERROR” response

7.11 Resolving an Outstanding CASTOUT Transaction

Table 7-10 describes the address collision resolution for an incoming transaction that collides with an outstanding CASTOUT transaction.

Table 7-10. Address Collision Resolution for CASTOUT

Outstanding Request	Incoming Request	Resolution
CASTOUT	READ_HOME	Generate “ERROR” response
CASTOUT	IREAD_HOME	Generate “ERROR” response
CASTOUT	READ_OWNER	Generate “RETRY” response; the CASTOUT will bypass address collision at home memory and modify the directory state
CASTOUT	READ_TO_OWN_HOME	Generate “ERROR” response
CASTOUT	READ_TO_OWN_OWNER	Generate “RETRY” response; the CASTOUT will bypass address collision at home memory and modify the directory state
CASTOUT	DKILL_HOME	Generate “ERROR” response
CASTOUT	DKILL_SHARER	Generate “ERROR” response
CASTOUT	CASTOUT	Generate “ERROR” response
CASTOUT	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
CASTOUT	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
CASTOUT	IKILL_HOME	Generate “ERROR” response
CASTOUT	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
CASTOUT	FLUSH	Generate “ERROR” response
CASTOUT	IO_READ_HOME	Generate “ERROR” response
CASTOUT	IO_READ_OWNER	Generate “RETRY” response; the CASTOUT will bypass address collision at home memory and modify the directory state

7.12 Resolving an Outstanding TLBIE or TLBSYNC Transaction

Table 7-11 describes the address collision resolution for an incoming transaction that collides with an outstanding TLBIE or TLBSYNC transaction.

Table 7-11. Address Collision Resolution for Software Coherence Operations

Outstanding Request	Incoming Request	Resolution
TLBIE, TLBSYNC	ANY	No collision, process request as described in Chapter 6, “Communication Protocols”

7.13 Resolving an Outstanding FLUSH Transaction

The flush operation has two distinct versions. The first is for processing elements that participate in the coherence protocol such as a processor and it’s associated agent, which may also have a local I/O device. The second is for processing elements that do not participate in the coherence protocols such as a pure I/O device that does not have a corresponding bit in the directory sharing mask. Table 7-12 describes the address collision resolution for an incoming transaction that collides with an outstanding participant FLUSH transaction.

Table 7-12. Address Collision Resolution for Participant FLUSH

Outstanding Request	Incoming Request	Resolution
FLUSH	READ_HOME	Generate “ERROR” response
FLUSH	IREAD_HOME	Generate “ERROR” response
FLUSH	READ_OWNER	Generate “NOT_OWNER” response (we are not allowed to issue FLUSH to an owned coherence granule - should be a CASTOUT)
FLUSH	READ_TO_OWN_HOME	Generate “ERROR” response
FLUSH	READ_TO_OWN_OWNER	Generate “NOT_OWNER” response (we are not allowed to issue FLUSH to an owned coherence granule - should be a CASTOUT)
FLUSH	DKILL_HOME	Generate “ERROR” response
FLUSH	DKILL_SHARER	If outstanding request, wait for all expected responses. If final response is “DONE” generate an “ERROR” response (we should never see a DKILL_SHARER if we own the coherence granule). If final response is “RETRY” cancel the flush at the processor and forward DKILL_SHARER to processor then generate a “DONE” response If no outstanding request, cancel the data cache invalidate at the processor and forward DKILL_SHARER to processor then generate a “DONE” response.
FLUSH	CASTOUT	Generate “ERROR” response
FLUSH	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
FLUSH	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
FLUSH	IKILL_HOME	Generate “ERROR” response
FLUSH	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
FLUSH	FLUSH	Generate “ERROR” response
FLUSH	IO_READ_HOME	Generate “ERROR” response
FLUSH	IO_READ_OWNER	Generate “NOT_OWNER” response (we are not allowed to issue FLUSH to an owned coherence granule - should be a CASTOUT)

Table 7-13 describes the address collision resolution for an incoming transaction that collides with an outstanding non-participant FLUSH transaction.

Table 7-13. Address Collision Resolution for Non-participant FLUSH

Outstanding Request	Incoming Request	Resolution
FLUSH	READ_HOME	Generate “ERROR” response (should never receive coherent operation)
FLUSH	IREAD_HOME	Generate “ERROR” response (should never receive coherent operation)
FLUSH	READ_OWNER	Generate “ERROR” response (should never receive coherent operation)
FLUSH	READ_TO_OWN_HOME	Generate “ERROR” response (should never receive coherent operation)
FLUSH	READ_TO_OWN_OWNER	Generate “ERROR” response (should never receive coherent operation)
FLUSH	DKILL_HOME	Generate “ERROR” response (should never receive coherent operation)
FLUSH	DKILL_SHARER	Generate “ERROR” response (should never receive coherent operation)
FLUSH	CASTOUT	Generate “ERROR” response (should never receive coherent operation)
FLUSH	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - non-participant may have page table hardware.
FLUSH	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - non-participant may have page table hardware.
FLUSH	IKILL_HOME	Generate “ERROR” response
FLUSH	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence) - non-participant may have software coherence.
FLUSH	FLUSH	Generate “ERROR” response (should never receive coherent operation)
FLUSH	IO_READ_HOME	Generate “ERROR” response (should never receive coherent operation)
FLUSH	IO_READ_OWNER	Generate “ERROR” response (should never receive coherent operation)

7.14 Resolving an Outstanding IO_READ_HOME Transaction

The I/O read operation is used by processing elements that do not want to participate in the coherence protocol but do want to get current copies of cached data. There are two versions of this operation, one for processing elements that have both processors and I/O devices, the second for pure I/O devices that do not have a corresponding bit in the directory sharing mask. Table 7-14 describes the address collision resolution for an incoming transaction that collides with an outstanding participant IO_READ_HOME transaction.

Table 7-14. Address Collision Resolution for Participant IO_READ_HOME

Outstanding Request	Incoming Request	Resolution
IO_READ_HOME	READ_HOME	Generate “ERROR” response
IO_READ_HOME	IREAD_HOME	Generate “ERROR” response
IO_READ_HOME	READ_OWNER	Generate “NOT_OWNER” response (we don’t own the data otherwise we could have obtained a copy locally)
IO_READ_HOME	READ_TO_OWN_HOME	Generate “ERROR” response
IO_READ_HOME	READ_TO_OWN_OWNER	Generate “NOT_OWNER” response (we don’t own the data otherwise we could have obtained a copy locally)
IO_READ_HOME	DKILL_HOME	Generate “ERROR” response
IO_READ_HOME	DKILL_SHARER	If outstanding request, wait for all expected responses. If final response is “DONE”, return data if necessary and forward DKILL_SHARER to processor then generate a “DONE” response. If final response is “RETRY” forward DKILL_SHARED to processor then generate a “DONE” response If no outstanding request forward DKILL_SHARER to processor then generate a “DONE” response
IO_READ_HOME	CASTOUT	Generate “ERROR” response
IO_READ_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IO_READ_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IO_READ_HOME	IKILL_HOME	Generate “ERROR” response
IO_READ_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence)
IO_READ_HOME	FLUSH	Generate “ERROR” response
IO_READ_HOME	IO_READ_HOME	Generate “ERROR” response
IO_READ_HOME	IO_READ_OWNER	Generate “NOT_OWNER” response (we don’t own the data otherwise we could have obtained a copy locally)

Table 7-15 describes the address collision resolution for an incoming transaction that collides with an outstanding non-participant IO_READ_HOME transaction.

Table 7-15. Address Collision Resolution for Non-participant IO_READ_HOME

Outstanding Request	Incoming Request	Resolution
IO_READ_HOME	READ_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	IREAD_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	READ_OWNER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	READ_TO_OWN_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	READ_TO_OWN_OWNER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	DKILL_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	DKILL_SHARER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	CASTOUT	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - broadcast operation and non-participant may have page table hardware.
IO_READ_HOME	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - broadcast operation and non-participant may have page table hardware.
IO_READ_HOME	IKILL_HOME	Generate “ERROR” response
IO_READ_HOME	IKILL_SHARER	No collision, forward to processor then generate “DONE” response (software must maintain instruction cache coherence) - broadcast operation and non-participant may have software coherence.
IO_READ_HOME	FLUSH	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	IO_READ_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_HOME	IO_READ_OWNER	Generate “ERROR” response (should never receive coherent operation)

7.15 Resolving an Outstanding IO_READ_OWNER Transaction

The I/O read operation is used by processing elements that do not want to participate in the coherence protocol but do want to get current copies of cached data. There are two versions of this operation, one for processing elements that have both processors and I/O devices, the second for pure I/O devices that do not have a corresponding bit in the directory sharing mask. Table 7-16 describes the address collision resolution for an incoming transaction that collides with an outstanding IO_READ_OWNER transaction.

Table 7-16. Address Collision Resolution for Participant IO_READ_OWNER

Outstanding Request	Incoming Request	Resolution
IO_READ_OWNER	READ_HOME	Generate “RETRY” response
IO_READ_OWNER	IREAD_HOME	Generate “RETRY” response
IO_READ_OWNER	READ_OWNER	Generate “ERROR” response
IO_READ_OWNER	READ_TO_OWN_HOME	Generate “RETRY” response
IO_READ_OWNER	READ_TO_OWN_OWNER	Generate “ERROR” response
IO_READ_OWNER	DKILL_HOME	Generate “RETRY” response
IO_READ_OWNER	DKILL_SHARER	Generate “ERROR” response
IO_READ_OWNER	CASTOUT	No collision, update directory state and memory, generate DONE response (CASTOUT bypasses address collision detection)
IO_READ_OWNER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IO_READ_OWNER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence)
IO_READ_OWNER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
IO_READ_OWNER	IKILL_SHARER	Generate “ERROR” response
IO_READ_OWNER	FLUSH	Generate “RETRY” response
IO_READ_OWNER	IO_READ_HOME	Generate “RETRY” response
IO_READ_OWNER	IO_READ_OWNER	Generate “ERROR” response (we don’t own the data otherwise we could have obtained a copy locally)

Table 7-17 describes the address collision resolution for an incoming transaction that collides with an outstanding non-participant IO_READ_OWNER transaction.

Table 7-17. Address Collision Resolution for Non-participant IO_READ_OWNER

Outstanding Request	Incoming Request	Resolution
IO_READ_OWNER	READ_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	IREAD_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	READ_OWNER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	READ_TO_OWN_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	READ_TO_OWN_OWNER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	DKILL_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	DKILL_SHARER	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	CASTOUT	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	TLBIE	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - non-participant may have page table hardware.
IO_READ_OWNER	TLBSYNC	No collision, forward to processor then generate “DONE” response (software must maintain TLB entry coherence) - non-participant may have page table hardware.
IO_READ_OWNER	IKILL_HOME	No collision, forward to processor, send IKILL_SHARER to all participants except requestor (software must maintain instruction cache coherence)
IO_READ_OWNER	IKILL_SHARER	Generate “ERROR” response
IO_READ_OWNER	FLUSH	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	IO_READ_HOME	Generate “ERROR” response (should never receive coherent operation)
IO_READ_OWNER	IO_READ_OWNER	Generate “ERROR” response (should never receive coherent operation)

Glossary of Terms and Abbreviations

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

A

Address collision. An address based conflict between two or more cache coherence operations when referencing the same coherence granule.

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

Asychronous transfer mode (**ATM**). A standard networking protocol which dynamically allocates bandwidth using a fixed-size packet.

B

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most significant byte.

Block flush. An operation that returns the latest copy of a block of data from caches within the system to memory.

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.

Broadcast. The concept of sending a packet to all processing elements in a system.

Bus-based snoopy protocol. A broadcast cache coherence protocol that assumes that all caches in the system are on a common bus.

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. Also referred to as memory coherence.

Cache coherent-non uniform memory access (**CC-NUMA**). A cache coherent system in which memory accesses have different latencies depending upon the physical location of the accessed address.

Cache paradox. A circumstance in which the caches in a system have an undefined or disallowed state for a coherence granule, for example, two caches have the same coherence granule marked “modified”.

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

Castout operation. An operation used by a processing element to relinquish its ownership of a coherence granule and return it to home memory.

Coherence domain. A logically associated group of processing elements that participate in the globally shared memory protocol and are able to maintain cache coherence among themselves.

Coherence granule. A contiguous block of data associated with an address for the purpose of guaranteeing cache coherence.

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.

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.

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.

Direct Memory Access (**DMA**). The process of accessing memory in a device by specifying the memory address directly.

Distributed memory. System memory that is distributed throughout the system, as opposed to being centrally located.

Domain. A logically associated group of processing elements.

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.

Ethernet. A common local area network (LAN) technology.

Exclusive. A processing element has the only cached copy of a sharable coherence granule. The exclusive state allows the processing element to modify the coherence granule without notifying the rest of the system.

F

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

Flush operation. An operation used by a processing element to return the ownership and current data of a coherence granule to home memory.

G

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

H

Half-word. A two byte or 16 bit quantity, aligned on two byte boundaries.

Home memory. The physical memory corresponding to the physical address of a coherence granule.

I

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

Instruction cache. High-speed memory containing recently accessed instructions (subset of main memory) associated with a processor.

Instruction cache invalidate operation. An operation that is used if the instruction cache coherence must be maintained by software.

Instruction read operation. An operation used to obtain a globally shared copy of a coherence granule specifically for an instruction cache.

Instruction set architecture (ISA). The instruction set for a certain processor or family of processors.

Intervention. A data transfer between two processing elements that does not go through the coherence granule’s home memory, but directly between the requestor of the coherence granule and the current owner.

Invalidate operation. An operation used to remove a coherence granule from caches within the coherence domain.

I/O. Input-output.

I/O read operation. An operation used by an I/O processing element to obtain a globally shared copy of a coherence granule without disturbing the coherence state of the granule.

L

Little-endian. A byte-ordering method in memory where the address n of a word corresponds to the least significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most significant byte.

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

LSB. Least significant byte.

M

Memory coherence. Memory is 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. Also referred to as cache coherence.

Memory controller. The point through which home memory is accessed.

Memory directory. A table of information associated with home memory that is used to track the location and state of coherence granules cached by coherence domain participants.

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

Modified. A processing element has written to a locally cached coherence granule and so has the only valid copy of the coherence granule in the system.

Modified exclusive shared invalid (**MESI**). A standard 4 state cache coherence definition.

Modified shared invalid (**MSI**). A standard 3 state cache coherence definition.

Modified shared local (**MSL**). A standard 3 state cache coherence definition.

MSB. Most significant byte.

Multicast. The concept of sending a packet to more than one processing elements in a system.

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.

Ownership. A processing element has the only valid copy of a coherence granule and is responsible for returning it to home memory.

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.

Priority. The relative importance of a packet; in most systems a higher priority 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

Read operation. An operation used to obtain a globally shared copy of a coherence granule.

Read-for-ownership operation. An operation used to obtain ownership of a coherence granule for the purposes of performing a write operation.

Remote access. An access by a processing element to memory located in another processing element.

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

S

Shared. A processing element has a cached copy of a coherence granule that may be cached by other processing elements and is consistent with the copy in home memory.

Sharing mask. The state associated with a coherence granule in the memory directory that tracks the processing elements that are sharing the coherence granule.

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

Sub-double-word. Aligned on eight byte boundaries.

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

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.

Translation look-aside buffer (TLB). Part of a processor's memory management unit; a TLB contains a set of virtual to physical page address translations, along with a set of attributes that describe access behavior for that portion of physical memory.

W

Write-through. A cache policy that passes all write operations through the caching hierarchy directly to home memory.

Word. A four byte or 32 bit quantity, aligned on four byte boundaries.