RapidIO™ Interconnect Specification Part 1: Input/Output Logical Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First public release	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, the following errata showings: 03-05-00006.001, 03-12-00001.001, 04-02-00001.002 and the following new features showings: 04-05-00005.001 Converted to ISO-friendly templates, re-formatted	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Overview

Chapter 2 System Models

Chapter 3 Operation Descriptions

Table of Contents

Chapter 4
Packet Format Descriptions

Chapter 5
Input/Output Registers

Table of Contents

Table of Contents

List of Figures

List of Figures

List of Tables

List of Tables

Chapter 1 Overview

1.1 Introduction

This chapter provides an overview of the *RapidIO Part 1: Input/Output Logical Specification*, including a description of the relationship between this specification and the other specifications of the RapidIO interconnect.

1.2 Overview

The *RapidIO Part 1: Input/Output 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 2: Message Passing Logical Specification* and *RapidIO Part 5: Globally Shared Memory 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.

1.3 Features of the Input/Output Specification

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

1.3.1 Functional Features

- A rich variety of transaction types, such as DMA-style read and writes, that allow efficient I/O systems to be built.
- System sizes from very small to very large are supported in the same or compatible packet formats—RapidIO plans for future expansion and requirements.
- Read-modify-write atomic operations are useful for synchronization between processors or other system elements.
- The RapidIO architecture supports 50- and 66-bit addresses as well as 34-bit local addresses for smaller systems.
- DMA devices can improve the interconnect efficiency if larger non-coherent data quantities can be encapsulated within a single packet, so RapidIO supports a variety of data sizes within the packet formats.

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.
- 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 Part 1: Input/Output Logical Specification:*

- Chapter 1, "Overview" (this chapter) provides an overview of the specification
- Chapter 2, "System Models," introduces some possible devices that could participate in a RapidIO system environment. Transaction ordering and deadlock prevention are discussed.
- Chapter 3, "Operation Descriptions," describes the set of operations and transactions supported by the RapidIO I/O protocols.
- Chapter 4, "Packet Format Descriptions," contains the packet format definitions for the I/O specification. The two basic types, request and response packets, with their sub-types and fields are defined.
- Chapter 5, "Input/Output Registers," describes the visible register set that allows an external processing element to determine the I/O capabilities, configuration, and status of a processing element using this logical specification. Only registers or register bits specific to the I/O 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.

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 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.

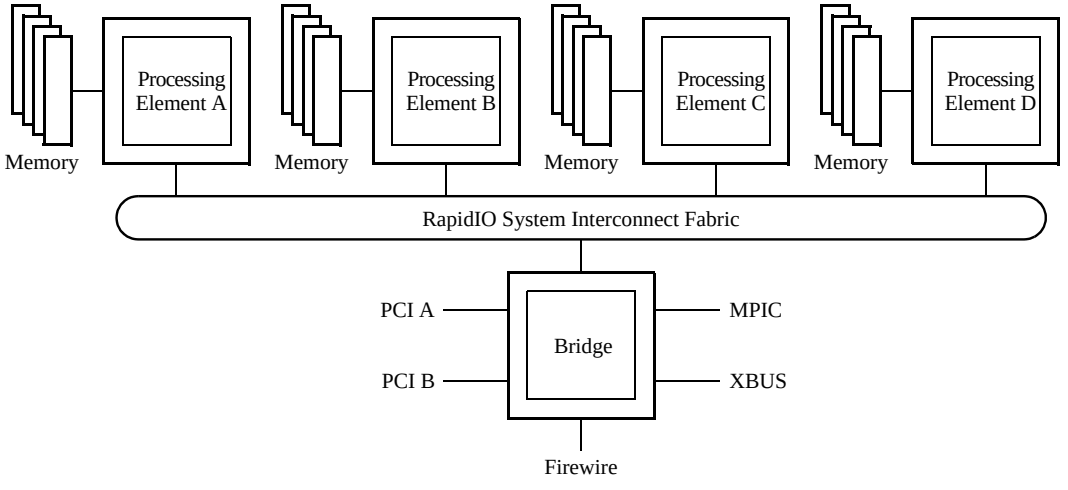


Figure 2-1. A Possible RapidIO-Based Computing System

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.

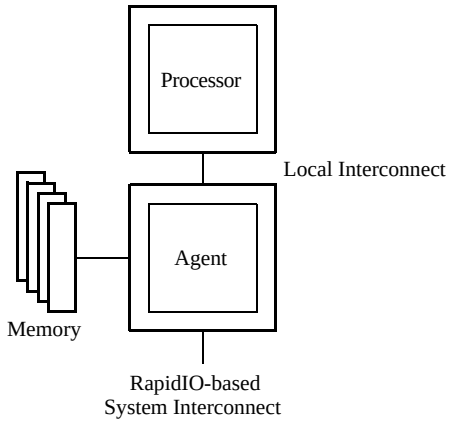


Figure 2-2. Processor-Memory Processing Element Example

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.

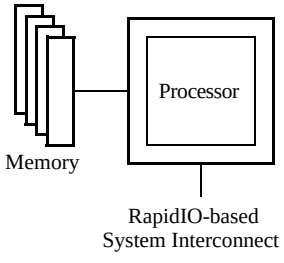


Figure 2-3. Integrated Processor-Memory Processing Element Example

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 shown in Figure 2-4. This type of device is much simpler than a processor; it only responds to requests from the external system, not to local requests as in the processor-based model. As such, its memory is remote for all processors in the system.

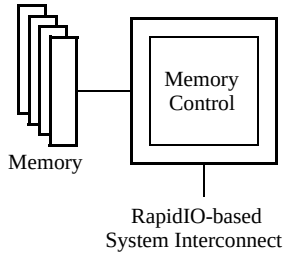


Figure 2-4. Memory-Only Processing Element Example

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.

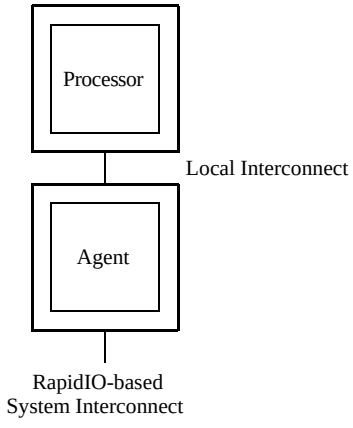


Figure 2-5. Processor-Only Processing Element Example

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 device. An I/O device only needs to move data into and out of local or remote memory.

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 hybrid processing element may combine a switch with end point functionality. A possible switch is shown in Figure 2-6. Behavior of the switches, and the interconnect fabric in general, is addressed in the *RapidIO Common Transport Specification.*

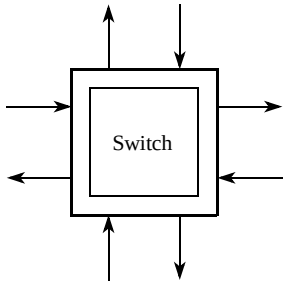


Figure 2-6. Switch Processing Element Example

2.3 System Issues

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

2.3.1 Operation Ordering

Most operations in an I/O system do not have any requirements as far as completion ordering. There are, however, several tasks that require events to occur in a specific order. As an example, a processing element may wish to write a set of registers in another processing element. The sequence in which those writes are carried out may be critical to the operation of the target processing element. Without some specific system rules there would be no guarantee of completion ordering of these operations. Ordering is mostly a concern for operations between a specific source and destination pair.

In certain cases a processing element may communicate with another processing element or set of processing elements in different contexts. A set or sequence of operations issued by a processing element may have requirements for completing in order at the target processing element. That same processing element may have another sequence of operations that also requires a completion order at the target processing element. However, the issuing processing element has no requirements for completion order between the two sequences of operations. Further, it may be desirable for one of the sequences of operations to complete at a higher priority than the other sequence. The term “transaction request flow” is defined as one of these sequences of operations.

A transaction request flow is defined as a ordered sequence of non-maintenance request transactions from a given source (as indicated by the source identifier) to a given destination (as indicated by the transaction destination identifier), where a maintenance request is a special system support request. Each packet in a transaction request flow has the same source identifier and the same destination identifier.

There may be multiple transaction request flows between a given source and destination pair. When multiple flows exist between a source and destination pair,

the flows are distinguished by a flow indicator (flowID). RapidIO allows multiple transaction request flows between any source and destination pair. The flows between each source and destination pair are identified with alphabetic characters beginning with A.

The flows between each source and destination pair are prioritized. The flow priority increases alphabetically with flowID A having the lowest priority, flowID B having the next to lowest priority, etc. When multiple transaction request flows exist between a given source and destination pair, transactions of a higher priority flow may pass transactions of a lower priority flow, but transactions of a lower priority flow may not pass transactions of a higher priority flow.

Maintenance transactions are not part of any transaction request flow. However, within a RapidIO fabric, maintenance transactions may not pass other maintenance transactions of the same or higher priority taking the same path through the fabric.

Response transactions are not part of any transaction request flow. There is no ordering between any pair of response transactions and there is no ordering between any response transaction and any request transaction that did not cause the generation of the response.

To support transaction request flows, all devices that support the RapidIO logical specification shall comply as applicable with the following Fabric Delivering Ordering and End point Completion Ordering rules.

Fabric Delivery Ordering Rules

1. **Non-maintenance request transactions within a transaction request flow (same source identifier, same destination identifier, and same flowID) shall be delivered to the logical layer of the destination in the same order that they were issued by the logical layer of the source.**
2. **Non-maintenance request transactions that have the same source (same source identifier) and the same destination (same destination identifier) but different flowIDs shall be delivered to the logical layer of the destination as follows.**
   - **A transaction of a higher priority transaction request flow that was issued by the logical layer of the source before a transaction of a lower priority transaction request flow shall be delivered to the logical layer of the destination before the lower priority transaction.**
   - **A transaction of a higher priority transaction request flow that was issued by the logical layer of the source after a transaction of a lower priority transaction request flow may be delivered to the logical layer of the destination before the lower priority transaction.**

3. **Request transactions that have different sources (different source identifiers) or different destinations (different destination identifiers) are unordered with respect to each other.**

End point Completion Ordering Rules

1. **Write request transactions in a transaction request flow shall be completed at the logical layer of the destination in the same order that the transactions were delivered to the logical layer of the destination.**
2. **A read request transaction with source A and destination B shall force the completion at the logical layer of B of all write requests in the same transaction request flow that were received by the logical layer of B before the read request transaction.**

Read request transactions need not be completed in the same order that they were received by the logical layer of the destination. As a consequence, read response transactions need not be issued by the logical layer of the destination in the same order that the associated read request transactions were received.

Write response transactions will likely be issued at the logical level in the order that the associated write request was received. However, since response transactions are not part of any flow, they are not ordered relative to one another and may not arrive at the logical level their destination in the same order as the associated write transactions were issued. Therefore, write response transactions need not be issued by the logical layer in the same order as the associated write request was received.

It may be necessary to impose additional rules in order to provide for inter operability with other interface standards or programming models. However, such additional rules are beyond the scope of this specification.

2.3.2 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.

2.3.2.1 Unordered Delivery System Issues

An unordered delivery system is defined as an interconnect fabric where transactions between a source and destination pair can arbitrarily pass each other during transmission through the intervening fabric.

Operations in the unordered system that are required to complete in a specific order shall be properly managed at the source processing element. For example, enforcing a specific sequence for writing a series of configuration registers, or preventing a subsequent read from bypassing a preceding write to a specific address are cases of ordering that may need to be managed at the source. The source of these transactions shall issue them in a purely serial sequence, waiting for completion notification for a write before issuing the next transaction to the interconnect fabric. The destination processing element shall guarantee that all outstanding non-coherent operations from that source are completed before servicing a subsequent non-coherent request from that source.

2.3.2.2 Ordered Delivery System Issues

Ordered delivery systems place additional implementation constraints on both the source and destination processing elements as well as any intervening hardware. Typically an ordered system requires that all transactions between a source/destination pair be completed in the order generated, not necessarily the order in which they can be accepted by the destination or an intermediate device. In one example, if several requests are sent before proper receipt is acknowledged the destination or intermediate device shall retry all following transactions until the first retried packet is retransmitted and accepted. In this case, the source shall “unroll” its outstanding transaction list and retransmit the first one to maintain the proper system ordering. In another example, an interface may make use of explicit transaction tags which allow the destination to place the transactions in the proper order upon receipt.

2.3.3 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), 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 ports. 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 read operations, that require responses to complete. These operations can lead to a dependency link between a processing element's input port and output port.

As an example of a 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.

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 their associated transactions supported by the I/O protocols of RapidIO. The transaction types, packet formats, and other necessary transaction information are described in Chapter 4, "Packet Format Descriptions."

The I/O operation protocols work using 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 another processing element sends an NREAD transaction in a request packet to that processing element, which reads its local memory at the requested address and returns the data in a DONE transaction in a response packet. Note that not all requests require responses; some requests assume that the desired activity will complete properly.

Two 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 it also returns data for read-type transactions as described above.
- 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 transport and physical layer specifications and are beyond the scope of this document.

For most transaction types, a 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.

Transaction IDs may also be used to indicate sequence information if ordered reception of transactions is required by the destination processing element and the interconnect fabric can reorder packets. The receiving device must accept and not complete the subsequent out-of-order requests until the missing transactions in the sequence have been received and completed.

3.2 I/O Operations Cross Reference

Table contains a cross reference of the I/O operations defined in this RapidIO specification and their system usage.

Table 2-1. I/O Operations Cross Reference

Operation	Transactions Used	Possible System Usage	Request Transaction Classification for Completion Ordering Rules	Description	Packet Format
Read	NREAD, RESPONSE	Read operation	Read	Section 3.3.1	Type 2 Section 4.1.5
Write	NWRITE	Write operation	Write	Section 3.3.2	Type 5 Section 4.1.7
Write-with-response	NWRITE_R, RESPONSE	Write operation	Write	Section 3.3.3	Type 5 Section 4.1.7
Streaming-write	SWRITE	Write operation	Write	Section 3.3.2	Type 6 Section 4.1.8
Atomic (read-modify-write)	ATOMIC, RESPONSE	Read-modify-write operation	Write	Section 3.3.4	Type 2 Section 4.1.5 Type 5 Section 4.1.7
Maintenance	MAINTENANCE	System exploration, initialization, and maintenance operation	not applicable	Section 3.4.1	Type 8 Section 4.1.10

3.3 I/O Operations

The operations described in this section are used for I/O accesses to physical addresses in the target of the operation. Examples are accesses to non-coherent memory, ROM boot code, or to configuration registers that do not participate in any globally shared system memory protocol. These accesses may be of any specifiable size allowed by the system.

All data payloads that are less than 8 bytes shall be padded and have their bytes aligned to their proper byte position within the double-word, as in the examples shown in Figure 3-6 through Figure 3-8.

The described behaviors are the same regardless of the actual target physical address.

3.3.1 Read Operations

The read operation, consisting of the NREAD and RESPONSE transactions (typically a DONE response) as shown in Figure 3-1, is used by a processing element that needs to read data from the specified address. The data returned is of the size requested.

If the read operation is to memory, data is returned from the memory regardless of the state of any system-wide cache coherence mechanism for the specified cache line or lines, although it may cause a snoop of any caches local to the memory controller.

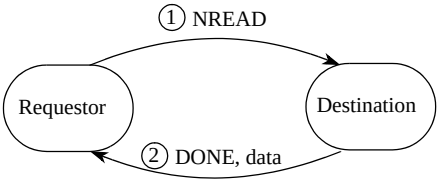


Figure 3-1. Read Operation

3.3.2 Write and Streaming-Write Operations

The write and streaming-write operations, consisting of the NWRITE and SWRITE transactions as shown in Figure 3-2, are used by a processing element that needs to write data to the specified address. The NWRITE transaction allows multiple double-word, word, half-word and byte writes with properly padded and aligned (to the 8-byte boundary) data payload. The SWRITE transaction is a double-word-only version of the NWRITE that has less header overhead. The write size and alignment for the NWRITE transaction are specified in Table 4-4. Non-contiguous and unaligned writes are not supported. It is the requestor's responsibility to break up a write operation into multiple transactions if the block is not aligned.

NWRITE and SWRITE transactions do not receive responses, so there is no notification to the sender when the transaction has completed at the destination.

If the write operation is to memory, data is written to the memory regardless of the state of any system-wide cache coherence mechanism for the specified cache line or lines, although it may cause a snoop of any caches local to the memory controller.

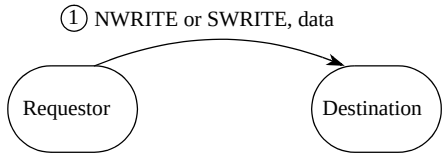


Figure 3-2. Write and Streaming-Write Operations

3.3.3 Write-With-Response Operations

The write-with-response operation, consisting of the NWRITE_R and RESPONSE transactions (typically a DONE response) as shown in Figure 3-3, is identical to the write operation except that it receives a response to notify the sender that the write has completed at the destination. This operation is useful for guaranteeing read-after-write and write-after-write ordering through a system that can reorder transactions and for enforcing other required system behaviors.

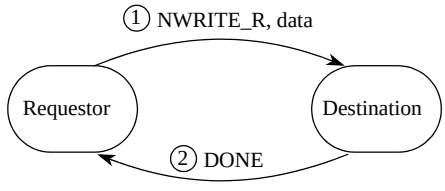


Figure 3-3. Write-With-Response Operation

3.3.4 Atomic (Read-Modify-Write) Operations

The read-modify-write operation, consisting of the ATOMIC and RESPONSE transactions (typically a DONE response) as shown in Figure 3-4, is used by a number of cooperating processing elements to perform synchronization using non-coherent memory. The allowed specified data sizes are one word (4 bytes), one half-word (2 bytes) or one byte, with the size of the transaction specified in the same way as for an NWRITE transaction. Double-word (8-byte) and 3, 5, 6, and 7 byte ATOMIC transactions may not be specified.

The atomic operation is a combination read and write operation. The destination reads the data at the specified address, returns the read data to the requestor, performs the required operation to the data, and then writes the modified data back to the specified address without allowing any intervening activity to that address. Defined operations are increment, decrement, test-and-swap, set, and clear (See bit settings in Table 5-9 and Table 5-10). Of these, only test-and-swap, compare-and-swap, and swap require the requesting processing element to supply data. The target data of an atomic operation may be initialized using an NWRITE transaction.

If the atomic operation is to memory, data is written to the memory regardless of the state of any system-wide cache coherence mechanism for the specified cache line or

lines, although it may cause a snoop of any caches local to the memory controller.

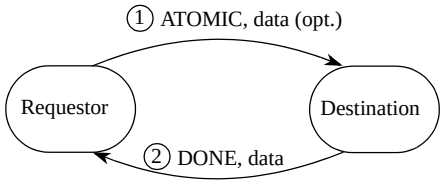


Figure 3-4. Atomic (Read-Modify-Write) Operation

3.4 System Operations

All data payloads that are less than 8 bytes shall be padded and have their bytes aligned to their proper byte position within the double-word, as in the examples shown in Figure 3-6 through Figure 3-8.

3.4.1 Maintenance Operations

The maintenance operation, which can consist of more than one MAINTENANCE transaction as shown in Figure 3-5, is used by a processing element that needs to read or write data to the specified CARs, CSRs, or locally-defined registers or data structures. If a response is required, MAINTENANCE requests receive a MAINTENANCE response rather than a normal response for both read and write operations. Supported accesses are in 32 bit quantities and may optionally be in double-word and multiple double-word quantities to a maximum of 64 bytes.

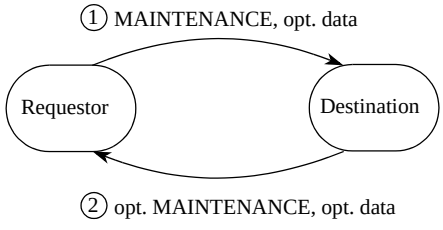
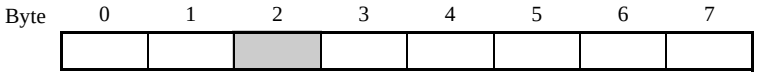


Figure 3-5. Maintenance Operation

3.5 Endian, Byte Ordering, and Alignment

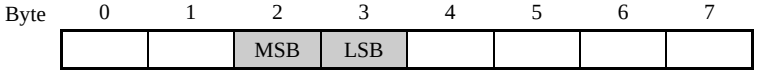
RapidIO has double-word (8-byte) aligned big-endian data payloads. This means that the RapidIO interface to devices that are little-endian shall perform the proper endian transformation to format a data payload.

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 3-6 through Figure 3-8.



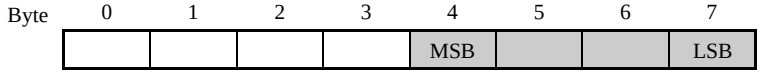
Byte address 0x0000_0002, the proper byte position is shaded.

Figure 3-6. Byte Alignment Example



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

Figure 3-7. Half-Word Alignment Example



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

Figure 3-8. 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 3-9 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.

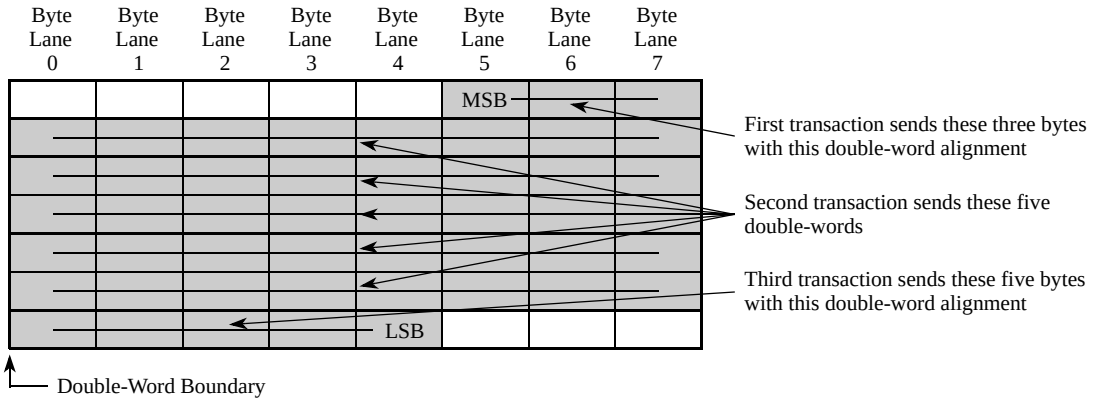


Figure 3-9. Data Alignment Example

Blank page

Chapter 4
Packet Format Descriptions

This chapter contains the packet format definitions for the *RapidIO Part 1: Input/Output Logical Specification*. Four types of I/O packet formats exist:

- 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.

4.1 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 I/O Logical Specification are shown in Table 4-1 below.

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 4.1.3
Type 1	—	Reserved	Section 4.1.4
Type 2	ATOMIC set	Read-write 1's to specified address	Section 4.1.5
	ATOMIC clear	Read-write 0's to specified address	
	ATOMIC increment	Read-increment-write to specified address	
	ATOMIC decrement	Read-decrement-write to specified address	
	NREAD	Read specified address	
Type 3-4	—	Reserved	Section 4.1.6

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

Request Packet Format Type	Transaction Type	Definition	Document Section No.
Type 5	ATOMIC test-and-swap	Read-test=0-swap-write to specified address	Section 4.1.7
	ATOMIC swap	Read-write to specified address	
	ATOMIC compare-and-swap	Read-test=first data-write second data to specified address	
	NWRITE	Write specified address	
	NWRITE_R	Write specified address, notify source of completion	
Type 6	SWRITE	Write specified address	Section 4.1.8
Type 7	—	Reserved	Section 4.1.9
Type 8	MAINTENANCE	Read or write device configuration registers and perform other system maintenance tasks	Section 4.1.10
Type 9-11	—	Reserved	Section 4.1.11

4.1.1 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.

All transactions are aligned to a byte, half-word, word, or double-word boundary. Read and write request addresses are aligned to any specifiable double-word boundary and are not aligned to the size of the data written or requested. Data payloads start at the first double-word and proceed linearly through the address space. Sub-double-word data payloads shall 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 shall break the operation into multiple request transactions. An example of this is shown in Section 3.5, “Endian, Byte Ordering, and Alignment.”

4.1.2 Field Definitions for All Request Packet Formats

Table 4-2 through Table 4-4 describe the field definitions for all request packet formats. 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.

Table 4-2. 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 4-3, Table 4-4 and Section 3.5.
rdsize	Data size for read transactions, used in conjunction with the word pointer (wdptr) bit—see Table 4-3 and Section 3.5.
wrsize	Write data size for sub-double-word transactions, used in conjunction with the word pointer (wdptr) bit—see Table 4-4 and Section 3.5. For writes greater than one double-word, the size is the maximum payload that should be expected by the receiver.
rsrv	Reserved
srcTID	The packet's transaction ID
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	Bits [0-28] of byte address [0-31] of the double-word physical address

Table 4-3. Read Size (rdsize) Definitions

wdptr	rdsize	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

Table 4-3. Read Size (rdsize) Definitions (Continued)

wdptr	rdsize	Number of Bytes	Byte Lanes
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	
0b0	0b1100	32	
0b1	0b1100	64	
0b0	0b1101	96	
0b1	0b1101	128	
0b0	0b1110	160	
0b1	0b1110	192	
0b0	0b1111	224	
0b1	0b1111	256	

Table 4-4. 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

Table 4-4. Write Size (wrsize) Definitions (Continued)

wdptr	wrsize	Number of Bytes	Byte Lanes
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	
00b	0b1101	reserved	
0b1	0b1101	128 maximum	
0b0	0b1110	reserved	
0b1	0b1110	reserved	
0b0	0b1111	reserved	
0b1	0b1111	256 maximum	

4.1.3 Type 0 Packet Format (Implementation-Defined)

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

4.1.4 Type 1 Packet Format (Reserved)

The type 1 packet format is reserved.

4.1.5 Type 2 Packet Format (Request Class)

The type 2 format is used for the NREAD and ATOMIC transactions as specified in the transaction field defined in Table 4-5. Type 2 packets never contain a data payload.

The data payload size for the response to an ATOMIC transaction is 8 bytes. The addressing scheme defined for the read portion of the ATOMIC transaction also controls the size of the atomic operation in memory so the bytes shall be contiguous and shall be of size byte, half-word (2 bytes), or word (4 bytes), and be aligned to that boundary and byte lane as with a regular read transaction. Double-word (8-byte), 3, 5, 6, and 7 byte ATOMIC transactions are not allowed.

Note that type 2 packets don't have any special fields.

Table 4-5. Transaction Fields and Encodings for Type 2 Packets

Encoding	Transaction Field
0b0000–0011	Reserved
0b0100	NREAD transaction
0b0101–1011	Reserved
0b1100	ATOMIC inc: post-increment the data
0b1101	ATOMIC dec: post-decrement the data
0b1110	ATOMIC set: set the data (write 0b11111...')
0b1111	ATOMIC clr: clear the data (write 0b00000...')

Figure 4-1 displays the type 2 packet with all its fields. The field value 0b0010 in Figure 4-1 specifies that the packet format is of type 2.

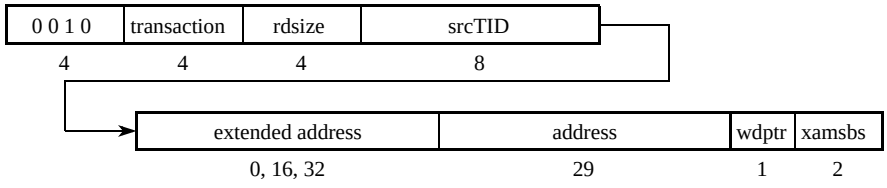


Figure 4-1. Type 2 Packet Bit Stream Format

4.1.6 Type 3–4 Packet Formats (Reserved)

The type 3–4 packet formats are reserved.

4.1.7 Type 5 Packet Format (Write Class)

Type 5 packets always contain a data payload. A data payload that consists of a single double-word or less has sizing information as defined in Table 4-4. The wrsize field specifies the maximum size of the data payload for multiple double-word transactions. The data payload may not exceed that size but may be smaller if desired. The ATOMIC, NWRITE, and NWRITE_R transactions use the type 5 format as defined in Table 4-6. NWRITE request packets do not require a response. Therefore, the transaction ID (srcTID) field for a NWRITE request is undefined and may have an arbitrary value.

The ATOMIC test-and-swap transaction is limited to one double-word (8 bytes) of data payload. The addressing scheme defined for the write transactions also controls the size of the atomic operation in memory so the bytes shall be contiguous and shall be of size byte, half-word (2 bytes), or word (4 bytes), and be aligned to that boundary and byte lane as with a regular write transaction. Double-word (8-byte) and 3, 5, 6, and 7 byte ATOMIC test-and-swap transactions are not allowed.

The ATOMIC swap transaction has the same addressing scheme and data payload

restrictions as the ATOMIC test-and-swap transaction.

The ATOMIC compare-and-swap operation is different from the other ATOMIC operations in that it requires two double-words (16 bytes) of data payload.

Note that type 5 packets don't have any special fields.

Table 4-6. Transaction Fields and Encodings for Type 5 Packets

Encoding	Transaction Field
0b0000–0011	Reserved
0b0100	NWRITE transaction
0b0101	NWRITE_R transaction
0b0110–1011	Reserved
0b1100	ATOMIC swap: read and return the data, unconditionally write with supplied data.
0b1101	ATOMIC compare-and-swap: read and return the data, if the read data is equal to the first 8 bytes of data payload, write the second 8 bytes of data to the memory location.
0b1110	ATOMIC test-and-swap: read and return the data, compare to 0, write with supplied data if compare is true
0b1111	Reserved

Figure 4-2 displays the type 5 packet with all its fields. The field value 0b0101 in Figure 4-2 specifies that the packet format is of type 5.

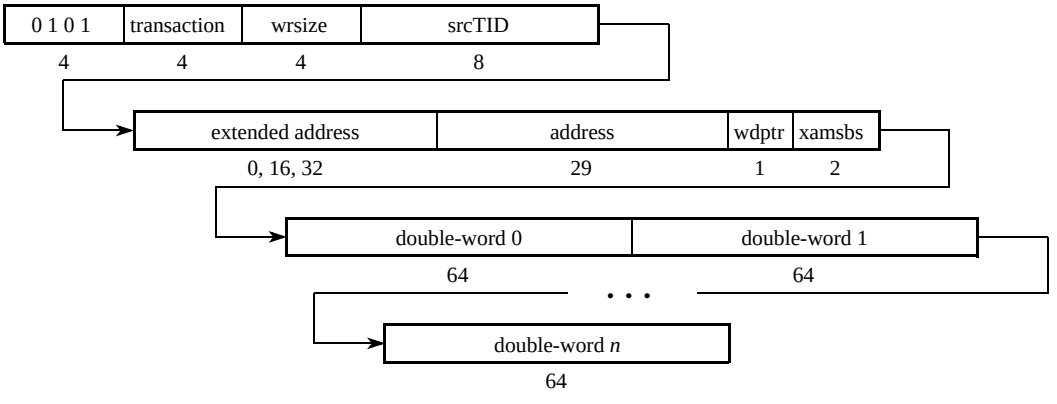


Figure 4-2. Type 5 Packet Bit Stream Format

4.1.8 Type 6 Packet Format (Streaming-Write Class)

The type 6 packet is a special-purpose type that always contains data. The data payload always contains a minimum of one complete double-word. Sub-double-word data payloads shall use the type 5 NWRITE transaction. Type 6 transactions may contain any number of double-words up to the maximum defined in Table 4-4.

Because the SWRITE transaction is the only transaction to use format type 6, there is no need for the transaction field within the packet. There are also no size or transaction ID fields.

Figure 4-3 displays the type 6 packet with all its fields. The field value 0b0110 in Figure 4-3 specifies that the packet format is of type 6.

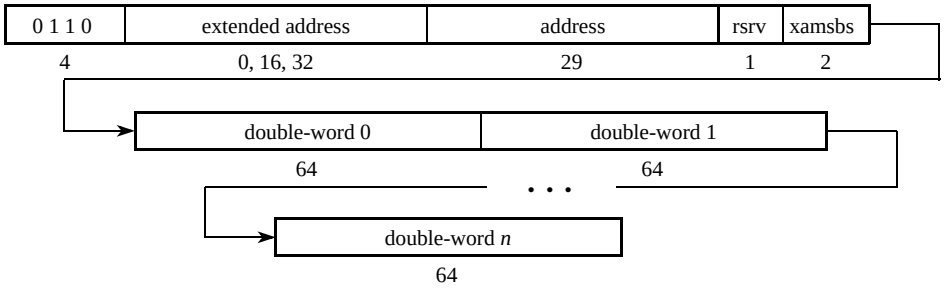


Figure 4-3. Type 6 Packet Bit Stream Format

4.1.9 Type 7 Packet Format (Reserved)

The type 7 packet format is reserved.

4.1.10 Type 8 Packet Format (Maintenance Class)

The type 8 MAINTENANCE packet format is used to access the RapidIO capability and status registers (CARs and CSRs) and data structures. Unlike other request formats, the type 8 packet format serves as both the request and the response format for maintenance operations. Type 8 packets contain no addresses and only contain data payloads for write requests and read responses. All configuration register read accesses are performed in word (4-byte), and optionally double-word (8-byte) or specifiable multiple double-word quantities up to a limit of 64 bytes. All register write accesses are also performed in word (4-byte), and optionally double-word (8-byte) or multiple double-word quantities up to a limit of 64 bytes.

Read and write data sizes are specified as shown in Table 4-3 and Table 4-4. The wrsize field specifies the maximum size of the data payload for multiple double-word transactions. The data payload may not exceed that size but may be smaller if desired. Both the maintenance read and the maintenance write request generate the appropriate maintenance response.

The maintenance port-write operation is a write operation that does not have guaranteed delivery and does not have an associated response. This maintenance operation is useful for sending messages such as error indicators or status information from a device that does not contain an end point, such as a switch. The data payload is typically placed in a queue in the targeted end point and an interrupt is typically generated to a local processor. A port-write request to a queue that is full or busy servicing another request may be discarded.

Definitions and encodings of fields specific to type 8 packets are provided in Table 4-7. Fields that are not specific to type 8 packets are described in Table 4-2.

Table 4-7. Specific Field Definitions and Encodings for Type 8 Packets

Type 8 Fields	Encoding	Definition
transaction	0b0000	Specifies a maintenance read request
	0b0001	Specifies a maintenance write request
	0b0010	Specifies a maintenance read response
	0b0011	Specifies a maintenance write response
	0b0100	Specifies a maintenance port-write request
	0b0101–1111	Reserved
config_offset	—	Double-word offset into the CAR/CSR register block for reads and writes
srcTID	—	The type 8 request packet's transaction ID (reserved for port-write requests)
targetTID	—	The corresponding type 8 response packet's transaction ID
status	0b0000	DONE—Requested transaction has completed successfully
	0b0001–0110	Reserved
	0b0111	ERROR—Unrecoverable error detected
	0b1000–1011	Reserved
	0b1100–1111	Implementation-defined—Can be used for additional information such as an error code

Figure 4-4 displays a type 8 request (read or write) packet with all its fields. The field value 0b1000 in Figure 4-4 specifies that the packet format is of type 8. The srcTID and config_offset fields are reserved for port-write requests.

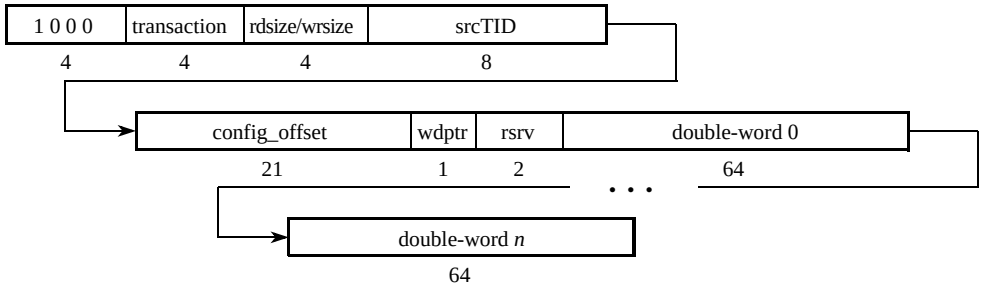


Figure 4-4. Type 8 Request Packet Bit Stream Format

Figure 4-5 displays a type 8 response packet with all its fields.

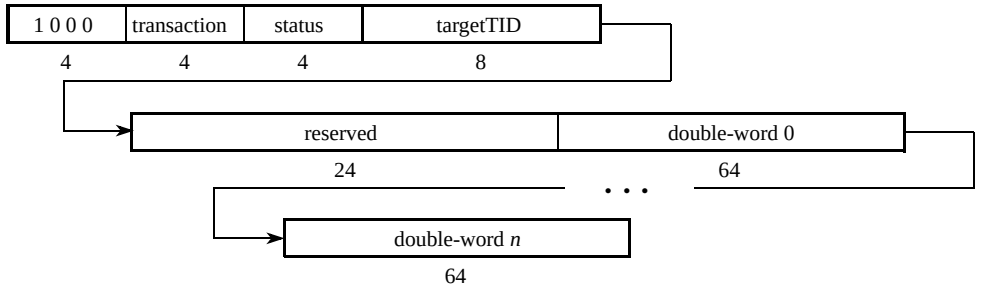


Figure 4-5. Type 8 Response Packet Bit Stream Format

4.1.11 Type 9–11 Packet Formats (Reserved)

The type 9–11 packet formats are reserved.

4.2 Response Packet Formats

A response transaction is issued by a processing element when it has completed a request made to it by a remote processing element. Response packets are always directed and are transmitted in the same way as request packets. Currently two packet format types exist, as shown in Table 4-8.

Table 4-8. Response Packet Type to Transaction Type Cross Reference

Response Packet Format Type	Transaction Type	Definition	Document Section Number
Type 12	—	Reserved	Section 4.2.2
Type 13	RESPONSE	Issued by a processing element when it completes a request by a remote element.	Section 4.2.3
Type 14	—	Reserved	Section 4.2.4
Type 15	Implementation-defined	Defined by the device implementation	Section 4.2.5

4.2.1 Field Definitions for All Response Packet Formats

The field definitions in Table 4-9 apply to more than one of the response packet formats.

Table 4-9. 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

Table 4-9. Field Definitions and Encodings for All Response Packets (Continued)

targetTID	—		The corresponding request packet's transaction ID
status	Type of status and encoding		
	0b0000	DONE	Requested transaction has been successfully completed
	0b0001–0110	—	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

4.2.2 Type 12 Packet Format (Reserved)

The type 12 packet format is reserved.

4.2.3 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 packets except maintenance and response-less writes.

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

Figure 4-6 illustrates the format and fields of type 13 packets. The field value 0b1101 in Figure 4-6 specifies that the packet format is of type 13.

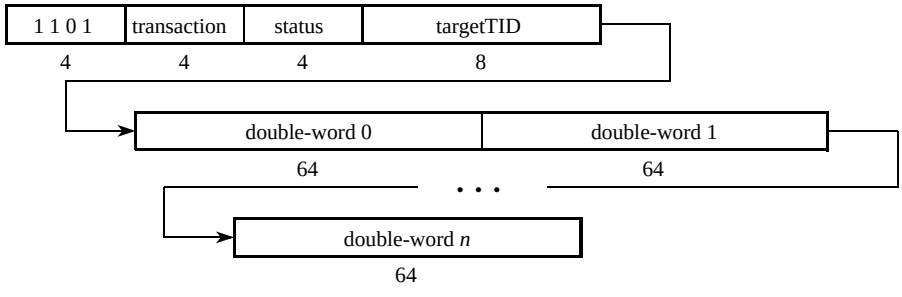


Figure 4-6. Type 13 Packet Bit Stream Format

4.2.4 Type 14 Packet Format (Reserved)

The type 14 packet format is reserved.

4.2.5 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
Input/Output Registers

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.

5.1 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 RapidIO 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. I/O Register Map

Configuration Space Byte Offset	Register Name
0x0	Device Identity CAR
0x4	Device Information CAR
0x8	Assembly Identity CAR
0xC	Assembly Information CAR
0x10	Processing Element Features CAR
0x14	Switch Port Information CAR

Table 5-1. I/O Register Map (Continued)

Configuration Space Byte Offset	Register Name
0x18	Source Operations CAR
0x1C	Destination Operations CAR
0x20–48	Reserved
0x4C	Processing Element Logical Layer Control CSR
0x50	Reserved
0x58	Local Configuration Space Base Address 0 CSR
0x5C	Local Configuration Space Base Address 1 CSR
0x60–FC	Reserved
0x100–FFFC	Extended Features Space
0x10000–FFFFFC	Implementation-defined Space

5.2 Reserved Register and Bit 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

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

Byte Offset	Space Name	Item	Initiator behavior	Target behavior
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
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.

5.3 Extended Features Data Structure

The RapidIO capability and command and status registers implement an extended capability data structure. If the extended features bit (bit 28) in the processing element features register is set, the extended features pointer is valid and points to the first entry in the extended features data structure. This pointer is an offset into the standard 16 Mbyte capability register (CAR) and command and status register (CSR) space and is accessed with a maintenance read operation in the same way as when accessing CARs and CSRs.

The extended features data structure is a singly linked list of double-word structures. Each of these contains a pointer to the next structure (EF_PTR) and an extended feature type identifier (EF_ID). The end of the list is determined when the next extended feature pointer has a value of logic 0. All pointers and extended features

blocks shall index completely into the extended features space of the CSR space, and all shall be aligned to a double-word boundary so the three least significant bits shall equal logic 0. Pointer values not in extended features space or improperly aligned are illegal and shall be treated as the end of the data structure. Figure 5-1 shows an example of an extended features data structure. It is required that the extended features bit is set to logic 1 in the processing element features register.

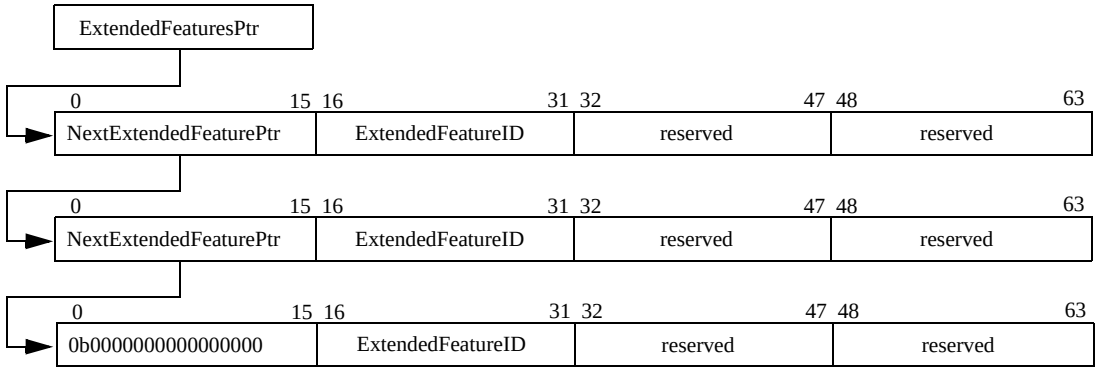


Figure 5-1. Example Extended Features Data Structure

5.4 Capability Registers (CARs)

Every processing element shall contain a set of registers that allows an external processing element to determine its capabilities through maintenance read operations. 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 Device Identity CAR (Configuration Space Offset 0x0)

The DeviceVendorIdentity field identifies the vendor that manufactured the device containing the processing element. A value for the DeviceVendorIdentity field is uniquely assigned to a device vendor by the registration authority of the RapidIO Trade Association.

The DeviceIdentity field is intended to uniquely identify the type of device from the vendor specified by the DeviceVendorIdentity field. The values for the DeviceIdentity field are assigned and managed by the respective vendor. See Table 5-3.

Table 5-3. Bit Settings for Device Identity CAR

Bit	Field Name	Description
0–15	DeviceIdentity	Device identifier
16–31	DeviceVendorIdentity	Device vendor identifier

5.4.2 Device Information CAR (Configuration Space Offset 0x4)

The DeviceRev field is intended to identify the revision level of the device. The value for the DeviceRev field is assigned and managed by the vendor specified by the DeviceVendorIdentity field. See Table 5-4.

Table 5-4. Bit Settings for Device Information CAR

Bit	Field Name	Description
0-31	DeviceRev	Device revision level

5.4.3 Assembly Identity CAR (Configuration Space Offset 0x8)

The AssyVendorIdentity field identifies the vendor that manufactured the assembly or subsystem containing the device. A value for the AssyVendorIdentity field is

uniquely assigned to a assembly vendor by the registration authority of the RapidIO Trade Association.

The AssyIdentity field is intended to uniquely identify the type of assembly from the vendor specified by the AssyVendorIdentity field. The values for the AssyIdentity field are assigned and managed by the respective vendor. See Table 5-5.

Table 5-5. Bit Settings for Assembly Identity CAR

Bit	Field Name	Description
0–15	AssyIdentity	Assembly identifier
16–31	AssyVendorIdentity	Assembly vendor identifier

5.4.4 Assembly Information CAR (Configuration Space Offset 0xC)

This register contains additional information about the assembly; see Table 5-6.

Table 5-6. Bit Settings for Assembly Information CAR

Bit	Field Name	Description
0–15	AssyRev	Assembly revision level
16–31	ExtendedFeaturesPtr	Pointer to the first entry in the extended features list

5.4.5 Processing Element Features CAR (Configuration Space Offset 0x10)

This register identifies the major functionality provided by the processing element; see Table 5-7.

Table 5-7. Bit Settings for Processing Element Features CAR

Bit	Field Name	Description
0	Bridge	PE can bridge to another interface. Examples are PCI, proprietary processor buses, DRAM, etc.
1	Memory	PE has physically addressable local address space and can be accessed as an end point through non-maintenance (i.e. non-coherent read and write) operations. This local address space may be limited to local configuration registers, or could be on-chip SRAM, etc.
2	Processor	PE physically contains a local processor or similar device that executes code. A device that bridges to an interface that connects to a processor does not count (see bit 0 above).
3	Switch	PE can bridge to another external RapidIO interface - an internal port to a local end point does not count as a switch port. For example, a device with two RapidIO ports and a local end point is a two port switch, not a three port switch, regardless of the internal architecture.
4–27	—	Reserved

Table 5-7. Bit Settings for Processing Element Features CAR (Continued)

Bit	Field Name	Description
28	Extended features	PE has extended features list; the extended features pointer is valid
29-31	Extended addressing support	Indicates the number address bits supported by the PE both as a source and target of an operation. All PEs shall at minimum support 34 bit addresses. 0b111 - PE supports 66, 50, and 34 bit addresses 0b101 - PE supports 66 and 34 bit addresses 0b011 - PE supports 50 and 34 bit addresses 0b001 - PE supports 34 bit addresses All other encodings reserved

5.4.6 Switch Port Information CAR (Configuration Space Offset 0x14)

This register defines the switching capabilities of a processing element. This register is only valid if bit 3 is set in the processing element features CAR; see Table 5-8.

Table 5-8. Bit Settings for Switch Port Information CAR

Bit	Field Name	Description
0–15	—	Reserved
16–23	PortTotal	The total number of RapidIO ports on the processing element 0b00000000 - Reserved 0b00000001 - 1 port 0b00000010 - 2 ports 0b00000011 - 3 ports 0b00000100 - 4 ports ... 0b11111111 - 255 ports
24–31	PortNumber	This is the port number from which the maintenance read operation accessed this register. Ports are numbered starting with 0x00.

5.4.7 Source Operations CAR (Configuration Space Offset 0x18)

This register defines the set of RapidIO IO logical operations that can be issued by this processing element; see Table 5-9. 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. For devices that have only switch functionality only bit 29 is valid. RapidIO switches shall be able to route any packet.

Table 5-9. Bit Settings for Source Operations CAR

Bit	Field Name	Description
0–13	—	Reserved
14–15	Implementation Defined	Defined by the device implementation
16	Read	PE can support a read operation
17	Write	PE can support a write operation

Table 5-9. Bit Settings for Source Operations CAR (Continued)

Bit	Field Name	Description
18	Streaming-write	PE can support a streaming-write operation
19	Write-with-response	PE can support a write-with-response operation
20-21	—	Reserved
22	Atomic (compare-and-swap)	PE can support an atomic compare-and-swap operation
23	Atomic (test-and-swap)	PE can support an atomic test-and-swap operation
24	Atomic (increment)	PE can support an atomic increment operation
25	Atomic (decrement)	PE can support an atomic decrement operation
26	Atomic (set)	PE can support an atomic set operation
27	Atomic (clear)	PE can support an atomic clear operation
28	Atomic (swap)	PE can support an atomic swap operation
29	Port-write	PE can support a port-write operation
30–31	Implementation Defined	Defined by the device implementation

5.4.8 Destination Operations CAR (Configuration Space Offset 0x1C)

This register defines the set of RapidIO I/O operations that can be supported by this processing element; see Table 5-10. It is required that all processing elements can respond to 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 5-10. Bit Settings for Destination Operations CAR

Bit	Field Name	Description
0-13	—	Reserved
14-15	Implementation Defined	Defined by the device implementation
16	Read	PE can support a read operation
17	Write	PE can support a write operation
18	Streaming-write	PE can support a streaming-write operation
19	Write-with-response	PE can support a write-with-response operation
20-21	—	Reserved
22	Atomic (compare-and-swap)	PE can support an atomic compare-and-swap operation
23	Atomic (test-and-swap)	PE can support an atomic test-and-swap operation
24	Atomic (increment)	PE can support an atomic increment operation
25	Atomic (decrement)	PE can support an atomic decrement operation
26	Atomic (set)	PE can support an atomic set operation
27	Atomic (clear)	PE can support an atomic clear operation
28	Atomic (swap)	PE can support an atomic swap operation

Table 5-10. Bit Settings for Destination Operations CAR (Continued)

Bit	Field Name	Description
29	Port-write	PE can support a port-write operation
30-31	Implementation Defined	Defined by the device implementation

5.5 Command and Status Registers (CSRs)

A processing element shall contain a set of command and status registers (CSRs) that allows an external processing element to control and determine the status of its internal hardware. All registers are 32 bits wide and are organized and accessed in the same way as the CARs. Refer to Table 5-2 for the required behavior for accesses to reserved registers and register bits.

5.5.1 Processing Element Logical Layer Control CSR (Configuration Space Offset 0x4C)

The Processing Element Logical Layer Control CSR is used for general command and status information for the logical interface.

Table 5-11. Bit Settings for Processing Element Logical Layer Control CSR

Bit	Field Name	Description
0–28	—	Reserved
29-31	Extended addressing control	Controls the number of address bits generated by the PE as a source and processed by the PE as the target of an operation. 0b100 - PE supports 66 bit addresses 0b010 - PE supports 50 bit addresses 0b001 - PE supports 34 bit addresses (default) All other encodings reserved

5.5.2 Local Configuration Space Base Address 0 CSR (Configuration Space Offset 0x58)

The local configuration space base address 0 register specifies the most significant bits of the local physical address double-word offset for the processing element's configuration register space. See Section 5.5.3 below for a detailed description.

Table 5-12. Bit Settings for Local Configuration Space Base Address 0 CSR

Bit	Field Name	Description
0	—	Reserved
1-16	LCSBA	Reserved for a 34-bit local physical address Reserved for a 50-bit local physical address Bits 0-15 of a 66-bit local physical address
17-31	LCSBA	Reserved for a 34-bit local physical address Bits 0-14 of a 50-bit local physical address Bits 16-30 of a 66-bit local physical address

5.5.3 Local Configuration Space Base Address 1 CSR (Configuration Space Offset 0x5C)

The local configuration space base address 1 register specifies the least significant bits of the local physical address double-word offset for the processing element's configuration register space, allowing the configuration register space to be physically mapped in the processing element. This register allows configuration and maintenance of a processing element through regular read and write operations rather than maintenance operations. The double-word offset is right-justified in the register.

Table 5-13. Bit Settings for Local Configuration Space Base Address 1 CSR

Bit	Field Name	Description
0	LCSBA	Reserved for a 34-bit local physical address Bit 15 of a 50-bit local physical address Bit 31 of a 66-bit local physical address
1-31	LCSBA	Bits 0-30 of a 34-bit local physical address Bits 16-46 of a 50-bit local physical address Bits 32-62 of a 66-bit local physical address

Blank page

Glossary of Terms and Abbreviations

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

A

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

B

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.

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

C

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

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

Cache line. A contiguous block of data that is the standard memory access size for a processor within a system.

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

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

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.

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

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

E

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

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

End point free device. A processing element which does not contain end point functionality.

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

External processing element. A processing element other than the processing element in question.

F

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

G

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

H

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

I

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

I/O. Input-output.

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

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.

MSB. Most significant byte.

N

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

O

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

P

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

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

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

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

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

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

R

Receiver. The RapidIO interface input port on a processing element.

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

ROM. Read-only memory.

S

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

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

SRAM. Static random access memory.

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.

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

W

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

Blank page

Blank page

RapidIO™ Interconnect Specification Part 2: Message Passing Logical Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First public release	03/08/2001
1.2	No technical changes	06/26/2002
1.3	Technical changes: incorporate Rev 1.2 errata 1 as applicable, the following errata showings: 03-05-00006.001, 03-07-00001.001, 04-02-00001.002, 04-05-00001.002 and the following new features showings: 02-05-00013.001 Converted to ISO-friendly templates; re-formatted	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Overview

Chapter 2 System Models

Chapter 3 Operation Descriptions

Table of Contents

Chapter 4 Packet Format Descriptions

Chapter 5 Message Passing Registers

Annex A Message Passing Interface

List of Figures

List of Figures

Blank page

List of Tables

List of Tables

Blank page

Chapter 1 Overview

1.1 Introduction

Part 2 is intended for users who need to understand the message passing architecture of the RapidIO interconnect.

1.2 Overview

The *RapidIO Part 2: Message Passing Logical Specification* is part of RapidIO's logical layer specifications that define the interconnect's overall protocol and packet formats. This layer contains the transaction protocols necessary for end points to process a transaction. Other RapidIO logical layer specifications include *RapidIO Part 1: Input/Output Logical Specification* and *RapidIO Part 5: Globally Shared Memory 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 encoding for the transport and physical layers lower in the RapidIO three-layer hierarchy.

RapidIO is targeted toward memory mapped distributed memory systems. A message passing programming model is supported to enable distributed I/O processing.

1.3 Features of the Message Passing Specification

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

1.3.1 Functional Features

- Many embedded systems are multiprocessor systems, not multiprocessing systems, and prefer a message passing or software-based coherency programming model over the traditional computer-style globally shared memory programming model in order to support their distributed I/O and processing requirements, especially in the networking and routing markets. RapidIO supports all of these programming models.

- System sizes from very small to very large are supported in the same or compatible packet formats—RapidIO plans for future expansion and requirements.
- Message passing devices can improve the interconnect efficiency if larger non-coherent data quantities can be encapsulated within a single packet, so RapidIO supports a variety of data sizes within the packet formats.
- Because the message passing programming model is fundamentally a non-coherent non-shared memory model, RapidIO can assume that portions of the memory space are only directly accessible by a processor or device local to that memory space. A remote device that attempts to access that memory space must do so through a local device controlled message passing interface.

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.
- 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 *RapidIO Part 2: Message Passing Logical Specification:*

- Chapter 1, “Overview” (this chapter) provides an overview of the specification

- Chapter 2, “System Models,” introduces some possible devices that might participate in a RapidIO message passing system environment. The chapter also explains the message passing model, detailing the data and doorbell message types used in a RapidIO system. System issues such as the lack of transaction ordering and deadlock prevention are presented.
- Chapter 3, “Operation Descriptions,” describes the set of operations and transactions supported by the RapidIO message passing protocols.
- Chapter 4, “Packet Format Descriptions,” contains the packet format definitions for the message passing specification. The two basic types, request and response packets, and their fields and sub-fields are explained.
- Chapter 5, “Message Passing Registers,” displays the RapidIO register map that allows an external processing element to determine the message passing capabilities, configuration, and status of a processing element using this logical specification. Only registers or register bits specific to the message passing 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.
- Annex A, “Message Passing Interface,” contains an informative discussion on possible programming models for the message passing logical layer.

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 system. The processing element is a computer device such as a processor attached to local memory and a RapidIO interconnect. The bridge part of the system provides I/O subsystem services such as high-speed PCI interfaces and Gbit ethernet ports, interrupt control, and other system support functions.

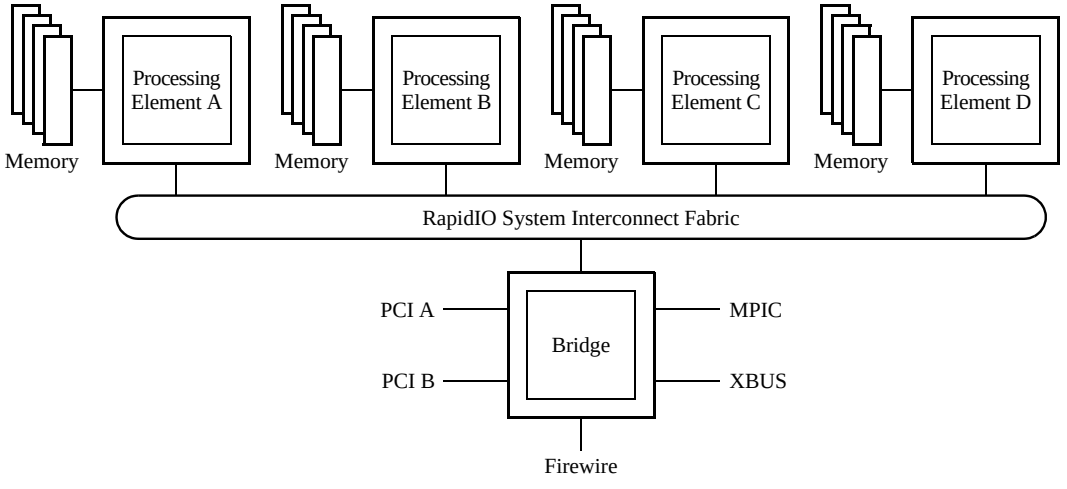


Figure 2-1. A Possible RapidIO-Based Computing System

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 local memory. It also provides an interface to the RapidIO interconnect to service message requests that are used for

communications with other processing elements.

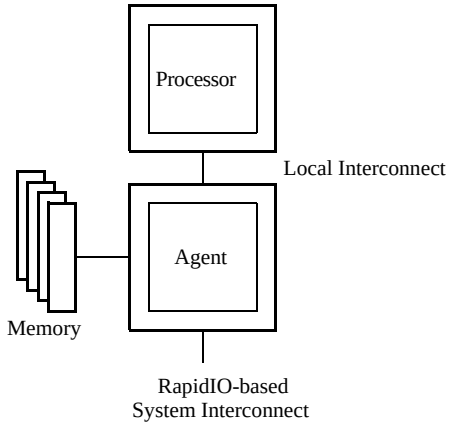


Figure 2-2. Processor-Memory Processing Element Example

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, 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.

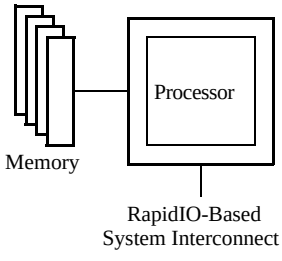


Figure 2-3. Integrated Processor-Memory Processing Element Example

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 in that 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.

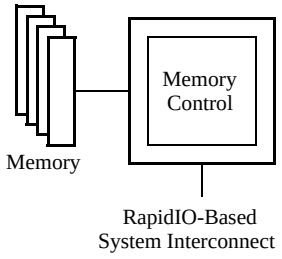


Figure 2-4. Memory-Only Processing Element Example

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.

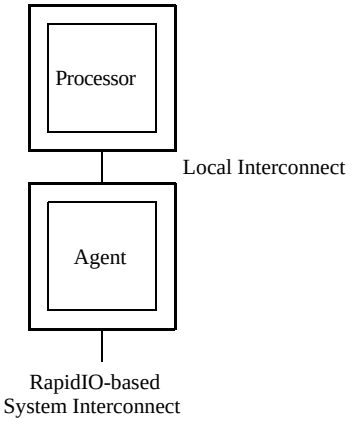


Figure 2-5. Processor-Only Processing Element Example

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.

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. Behavior of the switches, and the interconnect fabric in general, is

addressed in the *RapidIO Common Transport Specification*.

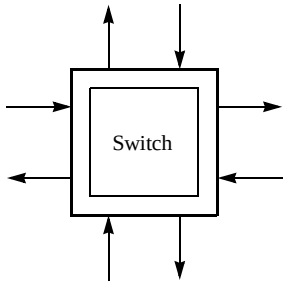


Figure 2-6. Switch Processing Element Example

2.3 Message Passing System Model

RapidIO supports a message passing programming model. Message passing is a programming model commonly used in distributed memory system machines. In this model, processing elements are only allowed to access memory that is local to themselves, and communication between processing elements is handled through specialized hardware manipulated through application or OS software. For two processors to communicate, the sending processor writes to a local message passing device that reads a section of the sender's local memory and moves that information to the receiving processor's local message passing device. The recipient message passing device then stores that information in local memory and informs the recipient processor that a message has arrived, usually via an interrupt. The recipient processor then accesses its local memory to read the message.

For example, referring to Figure 2-1, processing element A can only access the memory attached to it, and cannot access the memory attached to processing elements B, C, or D. Correspondingly, processing element B can only access the memory attached to it and cannot access the memory attached to processing element A, C, or D, and so on. If processing element A needs to communicate with processing element B, the application software accesses special message passing hardware (also called mailbox hardware) through operating system calls or API libraries and configure it to assemble the message and send it to processing element B. The message passing hardware for processing element B receives the message and puts it into local memory at a predetermined address, then notifies processing element B.

Many times a message is required to be larger than a single packet allows, so the source needs to break up the message into multiple packets before transmitting it. At times it may also be useful to have more than one message being transmitted at a time. RapidIO has facilities for both of these features.

2.3.1 Data Message Operations

A source may generate a single message operation of up to 16 individual packets containing as much as 256 data bytes per packet. A variety of data payload sizes exist, allowing a source to choose a smaller size data payload if needed for an application. RapidIO defines all data message packets as containing the same amount of data with the exception of the last one, which can contain a smaller data payload if desired. The packets are formatted with three fields:

- One field specifies the size of the data payload for all except the last packet for the data message operation.
- The second field specifies the size of the data payload for that packet, and
- The third field contains the packet sequence order information.

The actual packet formats are shown in Chapter 4, “Packet Format Descriptions.”

Because all packets except the last have the same data payload size, the receiver is able to calculate the local memory storage addresses if the packets are received out of order, allowing operation with an interconnect fabric that does not guarantee packet delivery ordering.

For multiple packet messages, a letter field and a mailbox field allow a source to simultaneously have up to four data message operations (or “letters”) in progress to each of four different mailboxes, allowing up to sixteen concurrent data message operations between a sender and a receiver. The mailbox field can be used to indicate the priority of a data message, allowing a higher priority message to interrupt a lower priority one at the sender, or it can be used as a simple mailbox identifier for a particular receiver if the receiver allows multiple mailbox addresses. If the mailbox number is used as a priority indicator, mailbox number 0 is the highest priority and mailbox 3 is the lowest.

For single packet messages, the letter and mailbox fields instead allow four concurrent letters to sixty-four possible mailboxes. As for multiple packet messages, if the mailbox number is used as a priority indicator, mailbox number 0 is the highest priority and mailbox 63 is the lowest.

The number of packets comprising a data message operation, the maximum data payload size, the number of concurrent letters, and the number of mailboxes that can be sent or received is determined by the implementation of a particular processing element. For example, a processing element could be designed to generate two concurrent letters of at most four packets with a maximum 64-byte data payload. That same processing element could also be designed to receive data messages in two mailboxes with two concurrent letters for each, all with the maximum data payload size and number of packets.

There is further discussion of the data message operation programming model and the necessary hardware support in Annex A, “Message Passing Interface”.

2.3.2 Doorbell Message Operations

RapidIO supports a second message type, the doorbell message operation. The doorbell message operation sends a small amount of software-defined information to the receiver and the receiver controls all local memory addressing as with the data message operation. It is the responsibility of the processor receiving the doorbell message to determine the action to undertake by examining the ID of the sender and the received data. All information supplied in a doorbell message is embedded in the packet header so the doorbell message never has a data payload.

The generation, transmission, and receipt of a doorbell message packet is handled in a fashion similar to a data message packet. If processing element A wants to send a doorbell message to processing element B, the application software accesses special doorbell message hardware through operating system calls or API libraries and configures it to assemble the doorbell message and send it to processing element B. The doorbell message hardware for processing element B receives the doorbell message and puts it into local memory at a predetermined address, then notifies processing element B, again, usually via an interrupt.

There is further discussion of the doorbell message operation programming model and the necessary hardware support in Annex A, “Message Passing Interface”.

2.4 System Issues

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

2.4.1 Operation Ordering

The *RapidIO Part 2: Message Passing Logical Specification* requires no special system operation ordering. Message operation completion is managed by the overlying system software.

It is important to recognize that systems may contain a mix of transactions that are maintained under the message passing model as well as under another model. As an example, I/O traffic may be interspersed with message traffic. In this case, the shared I/O traffic may require strong ordering rules to maintain coherency. This may set an operation ordering precedence for that implementation, especially in the case where the connection fabric cannot discern between one type of operation and another.

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.

A message operation may consist of several transactions. It is possible for these transactions to arrive at a target mailbox in an arbitrary order. A message transaction contains explicit tagging information to allow the message to be reconstructed as it arrives at the target processing element.

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 must 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 read 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 a input to output port dependency, consider a processing element where the output port queue is full. The processing element cannot 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.

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

Blank page

Chapter 3 Operation Descriptions

3.1 Introduction

This chapter describes the set of operations and transactions supported by the RapidIO message passing protocols. The opcodes and packet formats are described in Chapter 4, “Packet Format Descriptions”.

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 needs to send part of a message operation to another processing element sends a MESSAGE request packet to that processing element, which processes the message packet and returns a DONE response packet.

Three possible response transactions can be received by a requesting processing element:

- A DONE response indicates to the requestor that the desired transaction has completed.
- A RETRY response shall be generated for a message transaction that attempts to access a mailbox that is busy servicing another message operation, as can a doorbell transaction that encounters busy doorbell hardware. A transaction request which receives a RETRY response must be re-transmitted in order to complete the operation.
- 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.

3.2 Message Passing Operations Cross Reference

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

Table 3-1. Message Passing Operations Cross Reference

Operation	Transactions Used	Possible System Usage	Description	Packet Format
Doorbell	DOORBELL, RESPONSE		Section 3.3.1	Type 10 Section 4.2.4
Data Message	MESSAGE, RESPONSE		Section 3.3.2	Type 11 Section 4.2.5

3.3 Message Passing Operations

The two kinds of message passing transactions are described in this section and defined as follows:

- Doorbell
- Data Message

3.3.1 Doorbell Operations

The doorbell operation, consisting of the DOORBELL and RESPONSE transactions (typically a DONE response) as shown in Figure 3-1, is used by a processing element to send a very short message to another processing element through the interconnect fabric. The DOORBELL transaction contains the info field to hold information and does not have a data payload. This field is software-defined and can be used for any desired purpose; see Section 4.2.4, “Type 10 Packet Formats (Doorbell Class),” for information about the info field.

A processing element that receives a doorbell transaction takes the packet and puts it in a doorbell message queue within the processing element. This queue may be implemented in hardware or in local memory. This behavior is similar to that of typical message passing mailbox hardware. The local processor is expected to read the queue to determine the sending processing element and the info field and

determine what action to take based on that information.

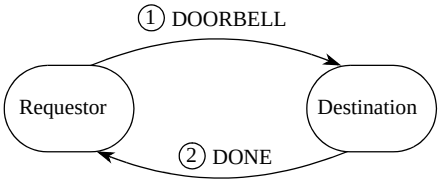


Figure 3-1. Doorbell Operation

3.3.2 Data Message Operations

The data message operation, consisting of the MESSAGE and RESPONSE transactions (typically a DONE response) as shown in Figure 3-2, is used by a processing element's message passing support hardware to send a data message to other processing elements. Completing a data message operation can consist of up to 16 individual MESSAGE transactions. MESSAGE transaction data payloads are always multiples of doubleword quantities.

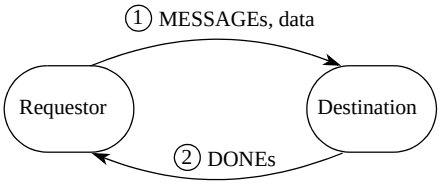


Figure 3-2. Message Operation

The processing element's message passing hardware that is the recipient of a data message operation examines a number of fields in order to place an individual MESSAGE packet data in local memory:

- Message length (msglen) field—Specifies the number of transactions that comprise the data message operation.
- Message segment (msgseg) field—Identifies which part of the data message operation is contained in this transaction. The message length and segment fields allow the individual packets of a data message to be sent or received out of order.
- Mailbox (mbox) field—Specifies which mailbox is the target of the data message.
- Letter (letter) field —Allows receipt of multiple concurrent data message operations from the same source to the same mailbox.
- Standard size (ssize) field—Specifies the data size of all of the transactions except (possibly) the last transaction in the data message.

From this information, the message passing hardware of the recipient processing element can calculate to which local memory address the

transaction data should be placed.

For example, assume that the mailbox starting addresses for the recipient processing element are at addresses 0x1000 for mailbox 0, 0x2000 for mailbox 1, 0x3000 for mailbox 2, and 0x4000 for mailbox 3, and that the processing element receives a message transaction with the following fields:

- message length of 6 packets
- message segment is 3rd packet
- mailbox is mailbox 2
- letter is 1
- standard size is 32 bytes
- data payload is 32 bytes (it shall be 32 bytes since this is not the last transaction)

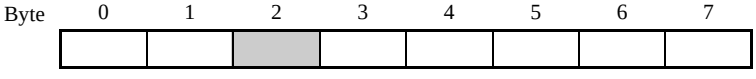
Using this information, the processing element's message passing hardware can determine that the 32 bytes contained in this part of the data message shall be put into local memory at address 0x3040.

The message passing hardware may also snoop the local processing element's caching hierarchy when writing local memory if the mailbox memory is defined as being cacheable by that processing element.

3.4 Endian, Byte Ordering, and Alignment

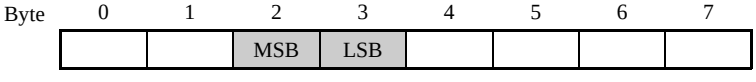
RapidIO has double-word (8-byte) aligned big-endian data payloads. This means that the RapidIO interface to devices that are little-endian shall perform the proper endian transformation at the output to format a data payload.

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 3-3 through Figure 3-5.



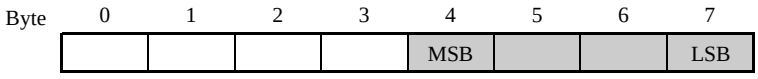
Byte address 0x0000_0002, the proper byte position is shaded.

Figure 3-3. Byte Alignment Example



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

Figure 3-4. Half-Word Alignment Example



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

Figure 3-5. Word Alignment Example

Blank page

Chapter 4 Packet Format Descriptions

4.1 Introduction

This chapter contains the packet format definitions for the *RapidIO Part 2: Message Passing Logical Specification.* There are four types of message passing 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 doorbell operation. The request packet format types and their transactions for the *RapidIO Part 2: Message Passing 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 Number
Type 0	Implementation-defined	Defined by the device implementation	Section 4.2.2
Type 1–9	—	Reserved	Section 4.2.3
Type 10	DOORBELL	Send a short message	Section 4.2.4
Type 11	MESSAGE	Send a message	Section 4.2.5

4.2.1 Field Definitions for All Request Packet Formats

The field definitions in Table 4-2 apply to all of the request packet formats. Fields that are unique to type 10 and type 11 formats are defined in the sections that describe each type. 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.

Table 4-2. 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.
rsrv	Reserved

4.2.2 Type 0 Packet Format (Implementation-Defined)

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

4.2.3 Type 1–9 Packet Formats (Reserved)

The type 1–9 formats are reserved.

4.2.4 Type 10 Packet Formats (Doorbell Class)

The type 10 packet format is the DOORBELL transaction format. Type 10 packets never have data payloads. The field value 0b1010 in Figure 4-1 specifies that the packet format is of type 10.

Definitions and encodings of fields specific to type 10 packets are provided in Table 4-3. Fields that are not specific to type 10 packets are described in Table 4-2.

Table 4-3. Specific Field Definitions for Type 10 Packets

Field	Encoding	Definition
info	—	Software-defined information field

Figure 4-1 displays a type 10 packet with all its fields.

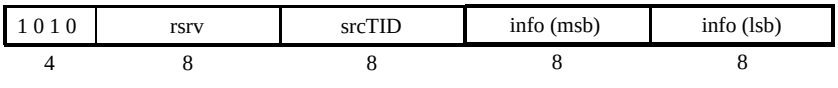


Figure 4-1. Type 10 Packet Bit Stream Format

4.2.5 Type 11 Packet Format (Message Class)

The type 11 packet is the MESSAGE transaction format. Type 11 packets always have a data payload. Sub-double-word messages are not specifiable and must be managed in software.

Definitions and encodings of fields specific to type 11 packets are provided in Table 4-4. Fields that are not specific to type 11 packets are described in Table 4-2.

Table 4-4. Specific Field Definitions and Encodings for Type 11 Packets

Field	Encoding	Definition
msglen	—	Total number of packets comprising this message operation. A value of 0 indicates a single-packet message. A value of 15 (0xF) indicates a 16-packet message, etc. See example in Section 3.3.2, “Data Message Operations”.
msgseg	—	For multiple packet data message operations, specifies the part of the message supplied by this packet. A value of 0 indicates that this is the first packet in the message. A value of 15 (0xF) indicates that this is the sixteenth packet in the message, etc. See example in Section 3.3.2, “Data Message Operations”.
xmbox	—	For single packet data message operations, specifies the upper 4 bits of the mailbox targeted by the packet. xmbox || mbox are specified as follows: 0000 00 - mailbox 0 0000 01 - mailbox 1 0000 10 - mailbox 2 0000 11 - mailbox 3 0001 00 - mailbox 4 .... 1111 11 - mailbox 63
ssize	—	Standard message packet data size. This field informs the receiver of a message the size of the data payload to expect for all of the packets for a single message operations except for the last packet in the message. This prevents the sender from having to pad the data field excessively for the last packet and allows the receiver to properly put the message in local memory. See example in Section 3.3.2, “Data Message Operations”.
	0b0000–1000	Reserved
	0b1001	8 bytes
	0b1010	16 bytes
	0b1011	32 bytes
	0b1100	64 bytes
	0b1101	128 bytes
	0b1110	256 bytes
	0b1111	Reserved
mbox	—	Specifies the recipient mailbox in the target processing element
letter	—	Identifies a slot within a mailbox. This field allows a sending processing element to concurrently send up to four messages to the same mailbox on the same processing element.

Figure 4-2 displays a type 11 packet with all its fields. The value 0b1011 in Figure 4-2 specifies that the packet format is of type 11.

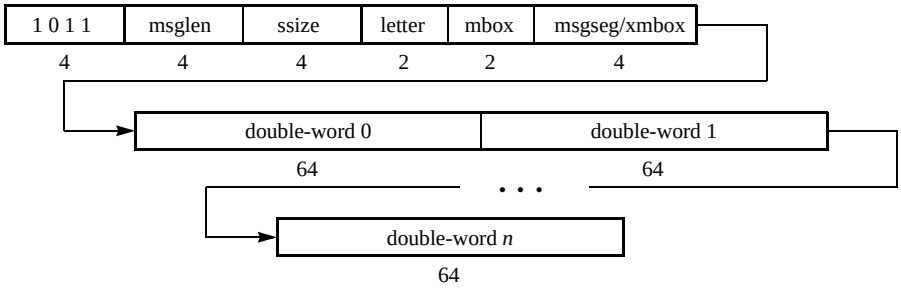


Figure 4-2. Type 11 Packet Bit Stream Format

The combination of the letter, mbox, and the msgseg or xmbox fields uniquely identifies the message packet in the system for each requestor and responder processing element pair in the same way as the transaction ID is used for other request types. Care must be taken to prevent aliasing of the combination of these values.

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 4-5.

Table 4-5. Response Packet Type to Transaction Type Cross Reference

Response Packet Format Type	Transaction Type	Definition	Document Section Number
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 4-6 apply to more than one of the response packet formats. Fields that are unique to the type 13 format are defined in Section 4.3.3,

"Type 13 Packet Format (Response Class)."

Table 4-6. Field Definitions and Encodings for All Response Packets

Field	Encoding	Sub-Field	Definition
transaction	0b0000		RESPONSE transaction with no data payload (including DOORBELL RESPONSE)
	0b0001		MESSAGE RESPONSE transaction
	0b0010–1111		Reserved
status	Type of status and encoding		
	0b0000	DONE	Requested transaction has been successfully completed
	0b0001–0010	—	Reserved
	0b0011	RETRY	Requested transaction is not accepted; re-transmission of the request is needed to complete the transaction
	0b0100–0110	—	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 and the requestor's transaction ID or message segment and mailbox information. The type 13 format is used for response packets to all request packets. Responses to message and doorbell packets never contain data.

Definitions and encodings of fields specific to type 13 packets are provided in Table 4-7. Fields that are not specific to type 13 packets are described in Table 4-6.

Table 4-7. Specific Field Definitions for Type 13 Packets

Field	Sub-Field	Definition
target_info	As shown in Figure 4-3, when the response is the target_info field, these three sub-fields are used:	
	msgseg	Specifies the part of the message supplied by the corresponding message packet. A value of 0 indicates that this is the response for the first packet in the message. A value of 15 (0xF) indicates that this is the response for the sixteenth (and last) packet in the message, etc.
	mbox	Specifies the recipient mailbox from the corresponding message packet.
	letter	Identifies the slot within the target mailbox. This field allows a sending processing element to concurrently send up to four messages to the same mailbox on the same processing element.
targetTID	—	Transaction ID of the request that caused this response (except for message responses defined in Figure 4-3).

Figure 4-3 shows the format of the target_info field for message responses.

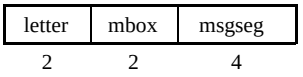


Figure 4-3. target_info Field for Message Responses

Figure 4-4 displays a type 13 packet with all its fields. The value 0b1101 in Figure 4-4 specifies that the packet format is of type 13.

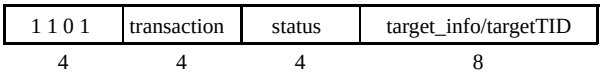


Figure 4-4. Type 13 Packet Bit Stream Format

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.

Chapter 5 Message Passing 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 *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. Message Passing 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. Message Passing Register Map (Continued)

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

5.3 Reserved Register and Bit 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.

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 message passing 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
0–13	—	Reserved
14–15	Implementation Defined	Defined by the device implementation
16–19	—	Reserved
20	Data message	PE can support a data message operation
21	Doorbell	PE can support a doorbell operation
22–29	—	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 message passing 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
0–13	—	Reserved
14–15	Implementation Defined	Defined by the device implementation
16–19	—	Reserved

Table 5-4. Bit Settings for Destination Operations CAR (Continued)

Bit	Field Name	Description
20	Data message	PE can support a data message operation
21	Doorbell	PE can support a doorbell operation
22–29	—	Reserved
30–31	Implementation Defined	Defined by the device implementation

5.5 Command and Status Registers (CSRs)

A processing element shall contain a set of command and status registers (CSRs) that allows an external processing element to control and determine the status of its internal hardware. All registers are 32 bits wide and are organized and accessed in the same way as the CARs. Refer to Table 5-2 for the required behavior for accesses to reserved registers and register bits.

Currently there are no CSRs defined by the message passing logical layer specification.

Annex A Message Passing Interface

A.1 Introduction

The *RapidIO Part 2: Message Passing Logical Specification* defines several packet formats that are useful for sending messages from a source device to a destination. These formats do not describe a specific programming model but are instantiated as an example packetizing mechanism. Because the actual programming models for message passing can vary greatly in both capability and complexity, they have been deemed beyond the scope of this specification. This appendix is provided as a reference model for message passing and is not intended to be all encompassing.

A.2 Definitions and Goals

A system may be made up of several processors and distributed memory elements. These processors may be tightly coupled and operating under a monolithic operating system in certain applications. When this is true the operating system is tasked with managing the pool of processors and memory to solve a set of tasks. In most of these cases, it is most efficient for the processors to work out of a common hardware-maintained coherent memory space. This allows processors to communicate initialization and completion of tasks through the use of semaphores, spin locks, and inter-process interrupts. Memory is managed centrally by the operating system with a paging protection scheme.

In other such distributed systems, processors and memory may be more loosely coupled. Several operating systems or kernels may be coexistent in the system, each kernel being responsible for a small part of the entire system. It is necessary to have a communication mechanism whereby kernels can communicate with other kernels in a system of this nature. Since this is a shared nothing environment, it is also desirable to have a common hardware and software interface mechanism to accomplish this communication. This model is typically called message passing.

In these message passing systems, two mechanisms typically are used to move data from one portion of memory space to another. The first mechanism is called direct memory access (DMA), the second is messaging. The primary difference between the two models is that DMA transactions are steered by the source whereas messages are steered by the target. This means that a DMA source not only requires access to a target but must also have visibility into the target's address space. The message source only requires access to the target and does not need visibility into the target's

address space. In distributed systems it is common to find a mix of DMA and messaging deployed.

The RapidIO architecture contains a packet transport mechanism that can aid in the distributed shared nothing environment. The RapidIO message passing model meets several goals:

- A message is constructed of one or more transactions that can be sent and received through a possibly unordered interconnect
- A sender can have a number of outstanding messages queued for sending
- A sender can send a higher priority message before a lower priority message and can also preempt a lower priority message to send a higher priority one and have the lower priority message resume when the higher is complete (prioritized concurrency)
- A sender requires no knowledge of the receiver's internal structure or memory map
- A receiver of a message has complete control over it's local address space
- A receiver can have a number of outstanding messages queued for servicing if desired
- A receiver can receive a number of concurrent multiple-transaction messages if desired

A.3 Message Operations

The *RapidIO Part 2: Message Passing Logical Specification* defines the type 11 packet as the MESSAGE transaction format. The transaction may be used in a number of different ways dependent on the specific system architecture. The transaction header contains the following field definitions:

mbox — Specifies the recipient mailbox in the target processing element. RapidIO allows up to four mailbox ports in each target device. This can be useful for defining blocks of different message frame sizes or different local delivery priority levels.

letter — A RapidIO message operation may be made up of several transactions. It may be desirable in some systems to have more than one multi-transaction message concurrently in transit to the target mailbox. The letter identifies the specific message within the mailbox. This field allows a sending of up to four messages to the same mailbox in the same target device.

multi-transaction fields — In cases where message operations are made up of multiple transactions, the following fields allow reconstruction of a message transported through an unordered interconnect fabric:

msglen — Specifies the total number of transactions comprising this message. A value of 0 indicates a single transaction message. A value of 15 (0xF) indicates a 16 transaction message, and so forth.

msgseg — Specifies the part of the message operation supplied by this transaction. A value of 0 indicates that this is the first transaction in the message. A value of 15 (0xF) indicates that this is the sixteenth transaction in the message, and so on.

ssize — Standard message transaction data size. This field tells the receiver to expect a message the size of the data field for all of the transactions except the last one. This prevents the sender from having to pad the data field excessively for the last transaction and allows the receiver to properly put the message in local memory; otherwise, if the last transaction is the first one received, the address calculations will be in error when writing the transaction to memory.

For a more detailed description of the message packet format, refer to Section 4.2.5, “Type 11 Packet Format (Message Class).”

The second type of message packet is the type 10 doorbell transaction packet. The doorbell transaction is a lightweight transaction that contains only a 16-bit information field that is completely software defined. The doorbell is intended to be an in-band mechanism to send interrupts between processors. In this usage the information field would be used to convey interrupt level and target information to the recipient. For a more detailed description of the doorbell packet format, refer to Section 4.2.4, “Type 10 Packet Formats (Doorbell Class).”

There are two transaction format models described in this appendix, a simple model and an extended model. The simple model is recommended for both the type 10 (doorbell) and type 11 (message) packet format messages. The extended model is only recommended for the type 11 (message) packet format messages.

A.4 Inbound Mailbox Structure

RapidIO provides two message transaction packet formats. By nature of having such formats it is possible for one device to pass a message to another device without a specific memory mapped transaction. The transaction allows for the concept of a memory map independent port. As mentioned earlier, how the transactions are generated and what is done with them at the destination is beyond the scope of the *RapidIO Part 2: Message Passing Logical Specification*. There are, however, a few examples as to how they could be deployed. First, look at the destination of the message.

A.4.1 Simple Inbox

Probably the most simple inbound mailbox structure is that of a single-register port or direct map into local memory space (see Figure A-1).

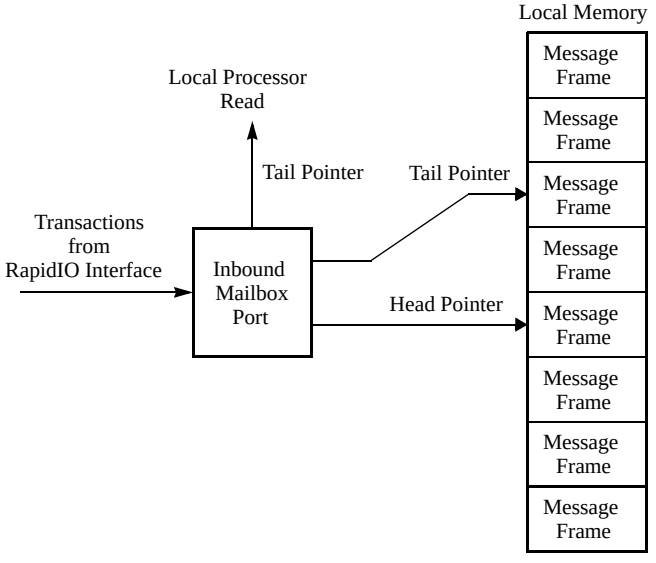


Figure A-1. Simple Inbound Mailbox Port Structure

In this structure, the inbound single transaction message is posted to either a register, set of registers, or circular queue in local memory. In the case of the circular queue, hardware maintains a head and tail pointer that points at a fixed window of pre-partitioned message frames in memory. Whenever the head pointer equals the tail pointer, no more messages can be accepted and they are retried on the RapidIO interface. When messages are posted, the local processor is interrupted. The interrupt service routine reads the mailbox port that contains the message located at the tail pointer. The message frame is equal to the largest message operation that can be received.

The RapidIO MESSAGE transaction allows up to four such inbound mailbox ports per target address. The DOORBELL transaction is defined as a single mailbox port.

A.4.2 Extended Inbox

A second more extensible structure similar to that used in the intelligent I/O (I_2O) specification, but managed differently, also works for the receiver (see Figure A-2).

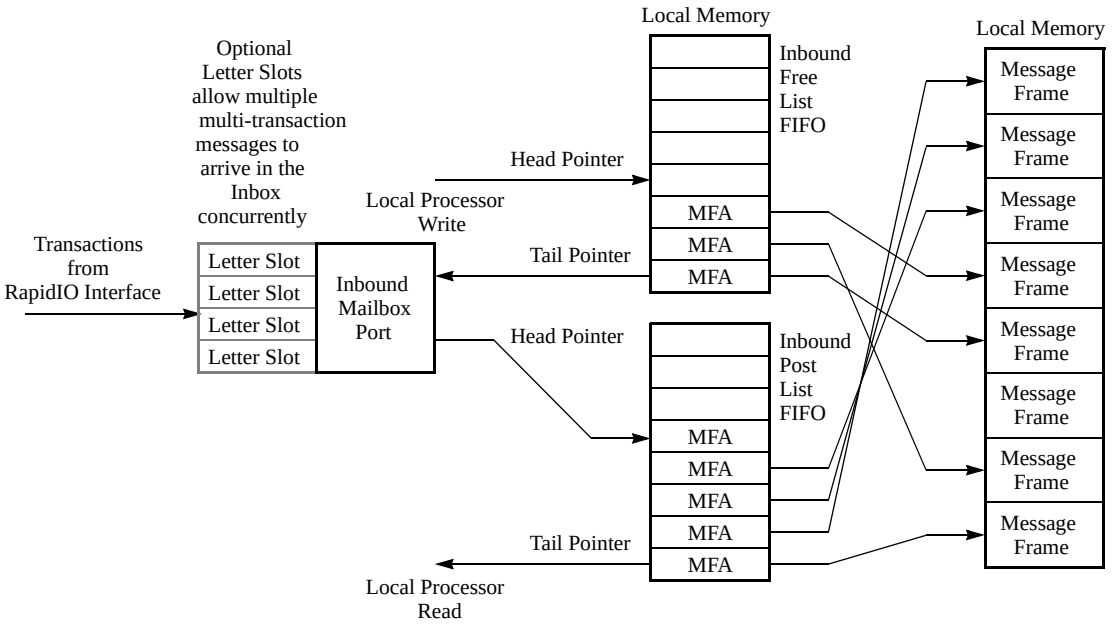


Figure A-2. Inbound Mailbox Structure

One of these structures is required for each priority level supported in an implementation. There are inbound post and free list FIFOs which function as circular queues of a fixed size. The message frames are of a size equal to the maximum message size that can be accepted by the receiver. Smaller messages can be accepted if allowed by the overlaying software. The sender only specifies the mailbox and does not request the frame pointer and perform direct memory access as with I_2O, although the I_2O model can be supported in software with this structure. All pointers are managed by the inbound hardware and the local processor. Message priority and letter number are managed by software.

The advantage of the extended structure is that it allows local software to service message frames in any order. It also allows memory regions to be moved in and out of the message structure instead of forcing software to copy the message to a different memory location.

A.4.3 Received Messages

When a message transaction is received, the inbound mailbox port takes the message frame address (MFA) pointed at by the inbound free list tail pointer and increments that pointer (this may cause a memory read to prefetch the next MFA), effectively taking the MFA from the free list. Subsequent message transactions from a different sender or with a different letter number are now retried until all of the transactions for this message operation have been received, unless there is additional hardware to handle multiple concurrent message operations for the same mailbox, differentiated by the letter slots.

The inbound mailbox port uses the MFA to write the transaction data into local memory at that base address with the exact address calculated as described in Section 2.3.1, “Data Message Operations” and Section 3.3.2, “Data Message Operations.” When the entire message is received and written into memory, the inbound post list pointer is incremented and the MFA is written into that location. If the queue was previously empty, an interrupt is generated to the local processor to indicate that there is a new message pending. This causes a window where the letter hardware is busy and cannot service a new operation between the receipt of the final transaction and the MFA being committed to the local memory.

When the local processor services a received message, it reads the MFA indicated by the inbound post FIFO tail pointer and increments the tail pointer. When the message has been processed (or possibly deferred), it puts a new MFA in the memory address indicated by the inbound free list head pointer and increments that pointer, adding the new MFA to the free list for use by the inbound message hardware.

If the free list head and tail pointer are the same, the FIFO is empty and there are no more MFAs available and all new messages are retried. If the post list head and tail pointers are the same, there are no outstanding messages awaiting service from the local processor. Underflow conditions are fatal since they indicate improper system behavior. This information can be part of an associated status register.

A.5 Outbound Message Queue Structure

Queuing messages in RapidIO is accomplished either through a simple or a more extended outbox.

A.5.1 Simple Outbox

Generation of a message can be as simple as writing to a memory-mapped descriptor structure either in local registers or memory. The outbound message queue (see Figure A-3) looks similar to the inbox.

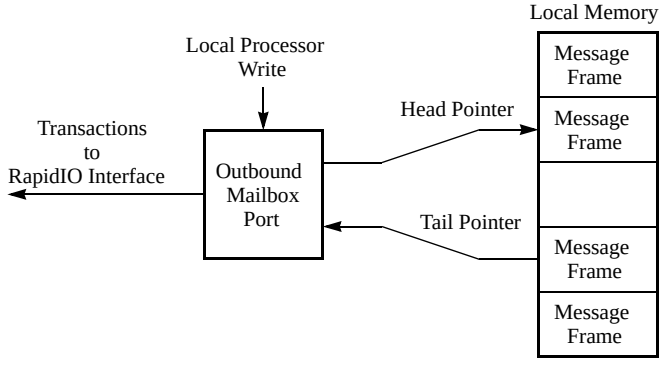


Figure A-3. Outbound Message Queue

The local processor reads a port in the outbound mailbox to obtain the position of a head pointer in local memory. If the read results in a pre-determined pattern the message queue is full. The processor then writes a descriptor structure and message to that location. When it is done, it writes the message port to advance the head point and mark the message as queued. The outbound mailbox hardware then reads the messages pointed to by the tail pointer and transfers them to the target device pointed at by the message descriptor.

One of these structures is required for each priority level of outbound messages supported.

A.5.2 Extended Outbox

A more extensible method of queueing messages is again a two-level approach (see Figure A-4). Multiple structures are required if concurrent operation is desired in an implementation. The FIFO is a circular queue of some fixed size. The message frames are of a size that is equal to the maximum message operation size that can be accepted by the receivers in the system. Smaller message operations can be sent if allowed by the hardware and the overlaying software. As with the receive side, the outbound slots can be virtual and any letter number can be handled by an arbitrary letter slot.

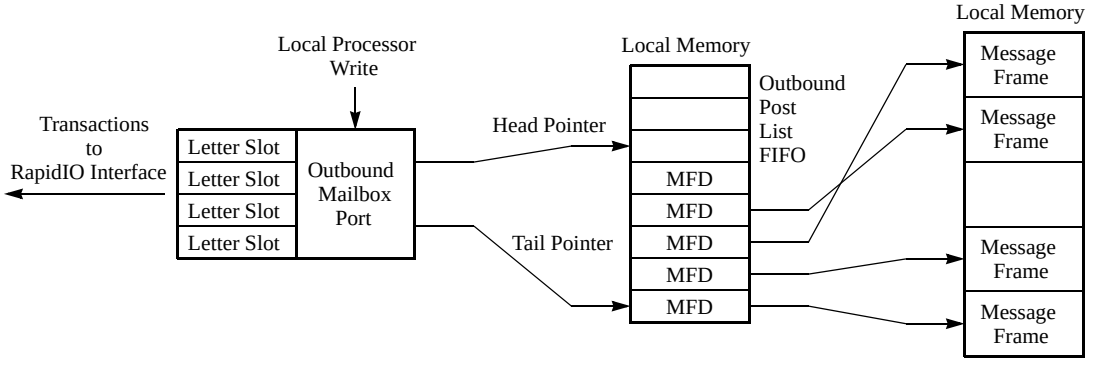


Figure A-4. Extended Outbound Message Queue

When the local processor wishes to send a message, it stores the message in local memory, writes the message frame descriptor (MFD) to the outbound mailbox port (which in-turn writes it to the location indicated by the outbound post FIFO head pointer), and increments the head pointer.

The advantage of this method is that software can have pre-set messages stored in local memory. Whenever it needs to communicate an event to a specific end point it writes the address of the message frame to the outbound mailbox, and the outbound mailbox generates the message transactions and completes the operation.

If the outbound post list FIFO head and tail pointers are not equal, there is a message waiting to be sent. This causes the outbound mailbox port to read the MFD pointed

to by the outbound post list tail pointer and then decrement the pointer (this may cause a memory read to prefetch the next MFD). The hardware then uses the information stored in the MFD to read the message frame, packetize it, and transmit it to the receiver. Multiple messages can be transmitted concurrently if there is hardware to support them, differentiated by the letter slots in Figure A-4.

If the free list head and tail pointer are the same, the FIFO is empty and there are no more MFDs to be processed. Underflow conditions are fatal because they indicate improper system behavior. This information can also be part of a status register.

Because the outbound and inbound hardware are independent entities, it is possible for more complex outbound mailboxes to communicate with less complex inboxes by simply reducing the complexity of the message descriptor to match. Likewise simple outboxes can communicate with complex inboxes. Software can determine the capabilities of a device during initial system setup. The capabilities of a devices message hardware are stored in the port configuration registers.

Glossary of Terms and Abbreviations

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

A

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

B

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.

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

C

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

CCITT. Consultive Communication for International Telegraph and Telephone.

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.

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.

Doorbell. A port on a device that is capable of generating an interrupt to a processor.

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

E

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

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

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

External processing element. A processing element other than the processing element in question.

F

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

FIFO. First in, first out.

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.

I

I_2O. Intelligent I/O architecture specification.

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

I/O. Input-output.

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

Mailbox. Dedicated hardware that receives messages.

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

MFA. Message frame address.

MFD. Message frame descriptor.

MSB. Most significant byte.

N

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

O

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

P

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

PCB. Printed circuit board.

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

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

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

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

R

Receiver. The RapidIO interface input port on a processing element.

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

S

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

Semaphore. A technique for coordinating activities in which multiple processing elements compete for the same resource, typically requiring atomic operations.

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

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

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.

W

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

Blank page

Blank page

RapidIO™ Interconnect Specification Part 3: Common Transport Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First public release	03/08/2001
1.2	No technical changes	06/26/2002
1.3	Technical changes: the following new features showings: 03-01-00002.006, 03-03-00002.002 Converted to ISO-friendly templates; re-formatted	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Overview

Chapter 2 Transport Format Description

Chapter 3 Common Transport Registers

Table of Contents

Table of Contents

Blank page

Table of Contents

List of Figures

List of Figures

List of Tables

List of Tables

Chapter 1 Overview

1.1 Introduction

This chapter provides an overview of the *RapidIO Part 3: Common Transport Specification*, including a description of the relationship between this specification and the other specifications of the RapidIO interconnect.

1.2 Overview

The *RapidIO Part 3: Common Transport Specification* defines a standard transport mechanism. In doing so, it specifies the header information added to a RapidIO logical packet and the way the header information is interpreted by a switching fabric. The RapidIO interconnect defines this mechanism independent of a physical implementation. The physical features of an implementation using RapidIO are defined by the requirements of the implementation, such as I/O signaling levels, interconnect topology, physical layer protocol, and error detection. These requirements are specified in the appropriate RapidIO physical layer specification.

This transport specification is also independent of any RapidIO logical layer specification.

1.3 Transport Layer Features

The transport layer functions of the RapidIO interconnect have been addressed by incorporating the following functional, physical, and performance features.

1.3.1 Functional Features

Functional features at the transport layer include the following:

- System sizes from very small to very large are supported in the same or compatible packet formats.
- Because RapidIO has only a single transport specification, compatibility among implementations is assured.
- The transport specification is flexible, so that it can be adapted to future applications.
- Packets are assumed, but not required, to be directed from a single source to a single destination.

1.3.2 Physical Features

The following are physical features of the RapidIO fabric that apply at the transport layer:

- The transport definition is independent of the width of the physical interface between devices in the interconnect fabric.
- No requirement exists in RapidIO for geographical addressing; a device's identifier does not depend on its location in the address map but can be assigned by other means.

1.3.3 Performance Features

Performance features that apply to the transport layer include the following:

- Packet headers are as small as possible to minimize the control overhead and are organized for fast, efficient assembly and disassembly.
- Broadcasting and multicasting can be implemented by interpreting the transport information in the interconnect fabric.
- Certain devices have bandwidth and latency requirements for proper operation. RapidIO does not preclude an implementation from imposing these constraints within the system.

1.4 Contents

RapidIO Part 3: Common Transport Specification contains three chapters:

- Chapter 1, "Overview" (this chapter) provides an overview of the specification
- Chapter 2, "Transport Format Description," describes the routing methods used in RapidIO for sending packets across the systems of switches described in this chapter.
- Chapter 3, "Common Transport Registers," describes the visible register set that allows an external processing element to determine the capabilities, configuration, and status of a processing element using this RapidIO transport layer definition.

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

Blank page

Chapter 2 Transport Format Description

2.1 Introduction

This chapter contains the transport format definition for the *RapidIO Part 3: Common Transport Specification*. Three transport fields are added to the packet formats described in the RapidIO logical specifications. The transport formats are intended to be fabric independent so the system interconnect can be anything required for a particular application; therefore all descriptions of the transport fields and their relationship with the logical packets are shown as bit streams.

2.2 System Topology

RapidIO is intended to be interconnect fabric independent. This section describes several of the possible system topologies and routing methodologies allowed by the processing element models described in the Models chapters of the different Logical Specifications.

2.2.1 Switch-Based Systems

A RapidIO system can be organized around the concept of switches. Figure 2-1 shows a small system in which five processing elements are interconnected through two switches. A logical packet sent from one processing element to another is routed through the interconnect fabric by the switches by interpreting the transport fields. Because a request usually requires a response, the transport fields must somehow indicate the return path from the requestor to the responder.

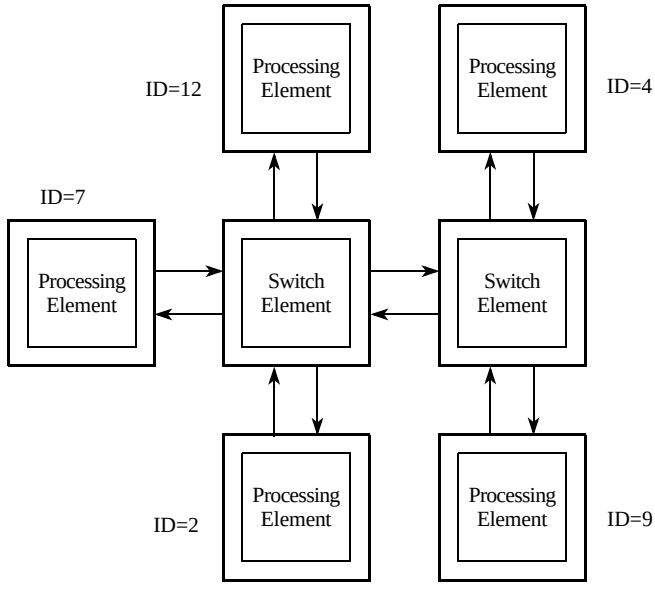


Figure 2-1. A Small Switch-Based System

2.2.2 Ring-Based Systems

A simplification of the switch structure is a ring as shown in Figure 2-2. A ring is a point-to-point version of a common bus; therefore, it is required to have a unique identifier for each processing element in the system. A packet put onto the ring contains the source and destination identifier in the transport fields. Each packet issued is examined by the downstream processing element. If that processing element's identifier matches that of the destination, it removes the packet from the ring for processing. If the destination identifier does not match the packet, it is passed to the next processing element in the ring.

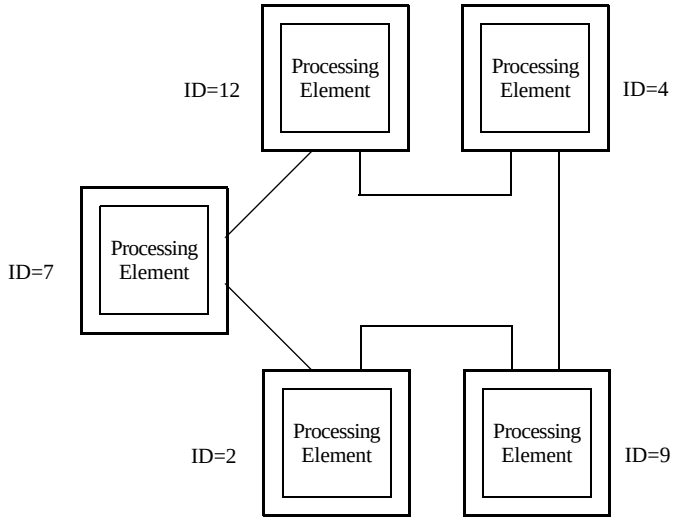


Figure 2-2. A Small Ring-Based System

2.3 System Packet Routing

There are many algorithms that can be used for routing through a system. The *RapidIO Part 3: Common Transport Specification* requires device identifier based packet routing. Each directly addressable device in the system shall have one or more unique device identifiers. When a packet is generated, the device ID of the destination of the packet is put in the packet header. The device ID of the source of the packet is also put in the packet header for use by the destination when generating response packets. When the destination of a request packet generates a response packet, it swaps the source and destination fields from the request, making the original source the new destination and itself the new source. Packets are routed through the fabric based on the destination device ID.

One method of routing packets in a switch fabric using device ID information incorporates routing tables. Each switch in the interconnect fabric contains a table that tells the switch how to route every destination ID from an input port to the proper output port. The simplest form of this method allows only a single path from every processing element to every other processing element. More complex forms of this method may allow adaptive routing for redundancy and congestion relief. However, the actual method by which packets are routed between the input of a switch and the output of a switch is implementation dependent.

2.4 Field Alignment and Definition

The *RapidIO Part 3: Common Transport Specification* adds a transport type (tt) field to the logical specification packet that allows four different transport packet types to be specified. The tt field indicates which type of additional transport fields are added

to the packet.

The three fields (tt, destinationID, and sourceID) added to the logical packets allow for two different sizes of the device ID fields, a large (16-bit), and a small (8-bit), as shown in Table 2-1. The two sizes of device ID fields allow two different system scalability points to optimize packet header overhead, and only affix additional transport field overhead if the additional addressing is required. The small device ID fields allow a maximum of 256 devices to be attached to the fabric. The large device ID fields allow systems with up to 65,536 devices.

Table 2-1. tt Field Definition

tt	Definition
0b00	8-bit deviceID fields
0b01	16-bit deviceID fields
0b10	Reserved
0b11	Reserved

Figure 2-3 shows the transport header definition bit stream. The shaded fields are the bits associated with the logical packet definition that are related to the transport bits. Specifically, the field labeled “Logical ftype” is the format type field defined in the logical specifications. This field comprises the first four bits of the logical packet. The second logical field shown (“Remainder of logical packet”) is the remainder of the logical packet of a size determined by the logical specifications, not including the logical ftype field which has already been included in the combined bit stream. The unshaded fields (tt=0b00 or tt=0b01 and destinationID and sourceID fields) are the transport fields added to the logical packet by the common transport specification.

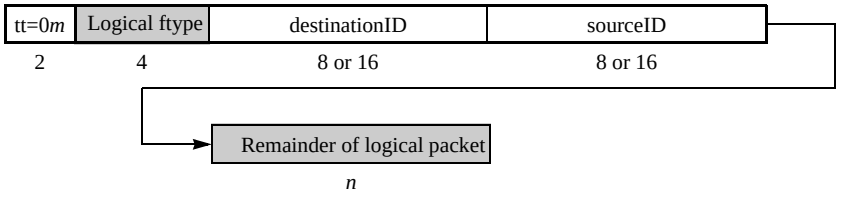


Figure 2-3. Destination-Source Transport Bit Stream

2.5 Routing Maintenance Packets

Routing maintenance packets in a switch-based network may be difficult because a switch processing element may not have its own device ID. An alternative method of addressing for maintenance packets for these devices uses an additional hop_count field in the packet to specify the number of switches (or hops) into the network from the issuing processing element that is being addressed. Whenever a switch processing element that does not have as associated device ID receives a maintenance packet it examines the hop_count field. If the received hop_count is

zero, the access is for that switch. If the hop_count is not zero, it is decremented and the packet is sent out of the switch according to the destinationID field. This method allows easy access to any intervening switches in the path between two addressable processing elements. However, since maintenance response packets are always targeted at an end point, the hop_count field shall always be assigned a value of 0xFF by the source of the packets to prevent them from being inadvertently accepted by an intervening device. Figure 2-4 shows the transport layer fields added to a maintenance logical packet. Maintenance logical packets can be found in the *RapidIO Part 1: Input/Output Logical Specification.*

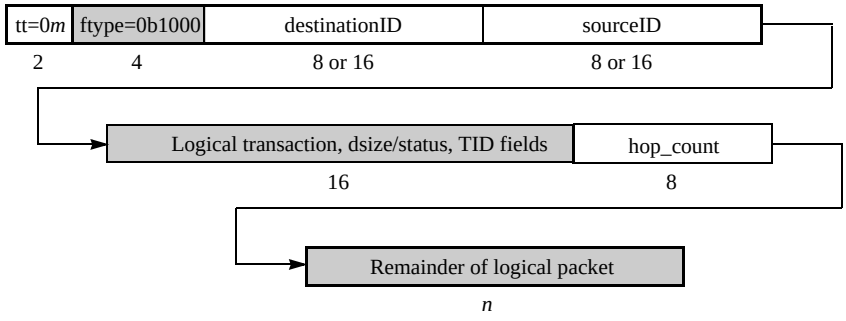


Figure 2-4. Maintenance Packet Transport Bit Stream

Blank page

Chapter 3 Common Transport Registers

3.1 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 transport layer definition. 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.

3.2 Register Summary

Table 3-1 shows the register address map for this RapidIO specification. These capability registers (CARs) and command and status registers (CSRs) can be accessed using *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 3-1. Common Transport Register Map

Configuration Space Byte Offset	Register Name
0x0-C	Reserved
0x10	Processing Element Features CAR
0x14–30	Reserved
0x34	Switch Route Table Destination ID Limit CAR
0x38-5C	Reserved

Table 3-1. Common Transport Register Map (Continued)

Configuration Space Byte Offset	Register Name
0x60	Base Device ID CSR
0x64	Reserved
0x68	Host Base Device ID Lock CSR
0x6C	Component Tag CSR
0x70	Standard Route Configuration Destination ID Select CSR
0x74	Standard Route Configuration Port Select CSR
0x78	Standard Route Default Port CSR
0x7C–FC	Reserved
0x100–FFFC	Extended Features Space
0x10000–FFFFFC	Implementation-defined Space

3.3 Reserved Register and Bit Behavior

Table 3-2 describes the required behavior for accesses to reserved register bits and reserved registers for the RapidIO register space,

Table 3-2. 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

Table 3-2. Configuration Space Reserved Access Behavior (Continued)

Byte Offset	Space Name	Item	Initiator behavior	Target behavior
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
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.

3.4 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 3-2 for the required behavior for accesses to reserved registers and register bits.

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

3.4.1 Processing Element Features CAR (Configuration Space Offset 0x10)

The processing element features CAR identifies the major functionality provided by the processing element. The bit settings are shown in Table 3-3.

Table 3-3. Bit Settings for Processing Element Features CAR

Bits	Name	Description
0–21	—	Reserved
22	Extended route table configuration support	0b0 - Switch PE does not support the extended route table configuration mechanism 0b1 - Switch PE supports the extended route table configuration mechanism (can only be set if bit 23 is set)
23	Standard route table configuration support	0b0 - Switch PE does not support the standard route table configuration mechanism 0b1 - Switch PE supports the standard route table configuration mechanism
22–26	—	Reserved
27	Common transport large system support	0b0 - PE does not support common transport large systems 0b1 - PE supports common transport large systems
28–31	—	Reserved

3.4.2 Switch Route Table Destination ID Limit CAR (Configuration Space Offset 0x34)

The Switch Route Table Destination ID Limit CAR specifies the maximum destination ID value that can be programmed with the standard route table configuration mechanism, and thereby indirectly defining the size of the route table. A route table access or extended route table access attempt to destination IDs greater than that specified in this register will have undefined results. This register is required if bit 23 of the Processing Element Features CAR is set. The bit settings are shown in Table 3-4.

Table 3-4. Bit Settings for Switch Route Table Destination ID Limit CAR

Bits	Name	Description
0–15	—	Reserved
16–31	Max_destID	Maximum configurable destination ID 0x00 - 1 destination ID 0x01 - 2 destinations IDs 0x02 - 3 destination IDs ... 0xFF - 65536 destination IDs

3.5 Command and Status Registers (CSRs)

A processing element shall contain a set of registers that allows an external processing element to control and determine status of its internal hardware. All registers are 32 bits wide and are organized and accessed in the same way as the CARs. Refer to Table 3-2 for the required behavior for accesses to reserved registers and register bits.

3.5.1 Base Device ID CSR (Configuration Space Offset 0x60)

The base device ID CSR contains the base device ID values for the processing element. A device may have multiple device ID values, but these are not defined in a standard CSR. The bit settings are shown in Table 3-5.

Table 3-5. Bit Settings for Base Device ID CSR

Bits	Name	Reset Value	Description
0-7	—		Reserved
8-15	Base_deviceID	see footnote[1]	This is the base ID of the device in a small common transport system (end point devices only)
16–31	Large_base_deviceID	see footnote[2]	This is the base ID of the device in a large common transport system (only valid for end point device and if bit 27 of the Processing Element Features CAR is set)

[1]The Base_deviceID reset value is implementation dependent

[2]The Large_base_deviceID reset value is implementation dependent

3.5.2 Host Base Device ID Lock CSR (Configuration Space Offset 0x68)

The host base device ID lock CSR contains the base device ID value for the processing element in the system that is responsible for initializing this processing element. The Host_base_deviceID field is a write-once/reset-able field which provides a lock function. Once the Host_base_deviceID field is written, all subsequent writes to the field are ignored, except in the case that the value written matches the value contained in the field. In this case, the register is re-initialized to 0xFFFF. After writing the Host_base_deviceID field a processing element must then read the Host Base Device ID Lock CSR to verify that it owns the lock before attempting to initialize this processing element. The bit settings are shown in Table 3-6.

Table 3-6. Bit Settings for Host Base Device ID Lock CSR

Bits	Name	Reset Value	Description
0-15	—		Reserved
16–31	Host_base_deviceID	0xFFFF	This is the base device ID for the PE that is initializing this PE.

3.5.3 Component Tag CSR (Configuration Space Offset 0x6C)

The component tag CSR contains a component tag value for the processing element and can be assigned by software when the device is initialized. It is especially useful for labeling and identifying devices that are not end points and do not have device ID registers. The bit settings are shown in Table 3-7.

Table 3-7. Bit Settings for Component ID CSR

Bits	Name	Reset Value	Description
0–31	component_tag	All 0s	This is a component tag for the PE.

3.5.4 Standard Route Configuration Destination ID Select CSR (Configuration Space Offset 0x70)

The Standard Route Configuration Destination ID Select CSR specifies the destination ID entry in the switch routing table to access when the Standard Route Configuration Port Select CSR is read or written.

The Ext_config_en bit controls whether the extended route table configuration mechanism is enabled. If the extended route table configuration mechanism is enabled, the specified destination ID and the next three sequential destination IDs are written or read when the Standard Route Configuration Port Select CSR accessed. Extended accesses that increment past the maximum specifiable destination ID (for example, starting an extended access at device ID 0xFF in a small transport system), has undefined results.

This register is required if bit 23 of the Processing Element Features CAR is set. The bit settings are shown in Table 3-8.

Table 3-8. Bit Settings for Standard Route Configuration Destination ID Select CSR

Bits	Name	Reset Value	Description
0	Ext_config_en	0b0	Extended Configuration Enable 0b0 - Extended configuration support is disabled 0b1 - Extended configuration support is enabled (only valid if bit 22 of the Processing Element Features CAR is set)
1-15	—		Reserved

Table 3-8. Bit Settings for Standard Route Configuration Destination ID Select CSR (Continued)

Bits	Name	Reset Value	Description
16-23	Config_destID_msb	0x00	Configuration destination ID most significant byte (only valid if bit 27 of the Processing Element Features CAR is set and the processing element is configured to operate in large transport mode)
24-31	Config_destID	0x00	Configuration destination ID

3.5.5 Standard Route Configuration Port Select CSR (Configuration Space Offset 0x74)

When written, the Standard Route Configuration Port Select CSR updates the switch output port configuration for packets with the destination ID selected by the Standard Route Configuration Destination ID Select CSR. When read, the Standard Route Configuration Port Select CSR returns the switch output port configuration for packets with the destination ID selected by the Standard Route Configuration Destination ID Select CSR.

If the extended route table configuration mechanism is enabled, when the Standard Route Configuration Port Select register is written the following route table configurations are carried out:

- destination ID Config_destID is routed to output port Config_output_port
- destination ID Config_destID+1 is routed to output port Config_output_port1
- destination ID Config_destID+2 is routed to output port Config_output_port2
- destination ID Config_destID+3 is routed to output port Config_output_port3

For reads of the Standard Route Configuration Port Select CSR, the configuration information is returned in the corresponding fashion.

After complete system initialization the switch output port route configuration information read may not be consistant with previously read values due to the capabilities and features of the particular switch. This register is required if bit 23 of the Processing Element Features CAR is set. The bit settings are shown in Table 3-9.

Table 3-9. Bit Settings for Standard Route Configuration Destination ID Select CSR

Bits	Name	Reset Value	Description
0-7	Config_output_port3	0x00	Configuration output port3 - This field is reserved if extended route table mechanism is not enabled
8-15	Config_output_port2	0x00	Configuration output port2 - This field is reserved if extended route table mechanism is not enabled
16-23	Config_output_port1	0x00	Configuration output port1 - This field is reserved if extended route table mechanism is not enabled
24-31	Config_output_port	see footnote[1]	Configuration output port

[1]The Config_output_port*n* reset values are implementation dependent

3.5.6 Standard Route Default Port CSR (Configuration Space Offset 0x78)

The Standard Route Default Port CSR specifies the port to which packets with destinations IDs that are greater than that specified in the Switch Route Table Destination ID Limit CAR are routed. This register is required if bit 23 of the Processing Element Features CAR is set. The bit settings are shown in Table 3-10.

Table 3-10. Bit Settings for Standard Route Default Port CSR

Bits	Name	Reset Value	Description
0-23	—		Reserved
24–31	Default_output_port	0x00	Default output port

Glossary of Terms and Abbreviations

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

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.

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

C

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

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

D

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.

E

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

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

External processing element. A processing element other than the processing element in question.

F

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

H

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

I

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

I/O. Input-output.

M

MSB. Most significant byte.

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

O

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

P

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

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

S

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

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

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.

Blank page

Blank page

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

Rev. 1.3, 06/2005

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

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Overview

Chapter 2 System Models

Chapter 3 Operation Descriptions

Table of Contents

Table of Contents

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.

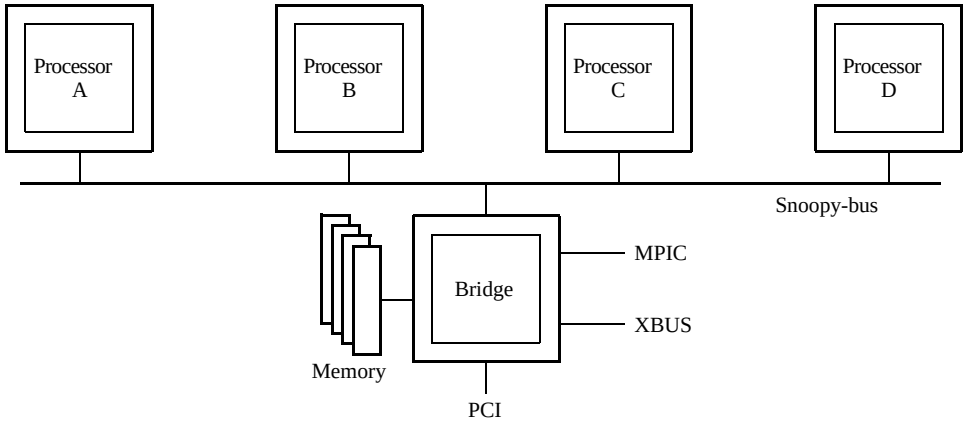


Figure 1-1. A Snoopy Bus-Based System

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.

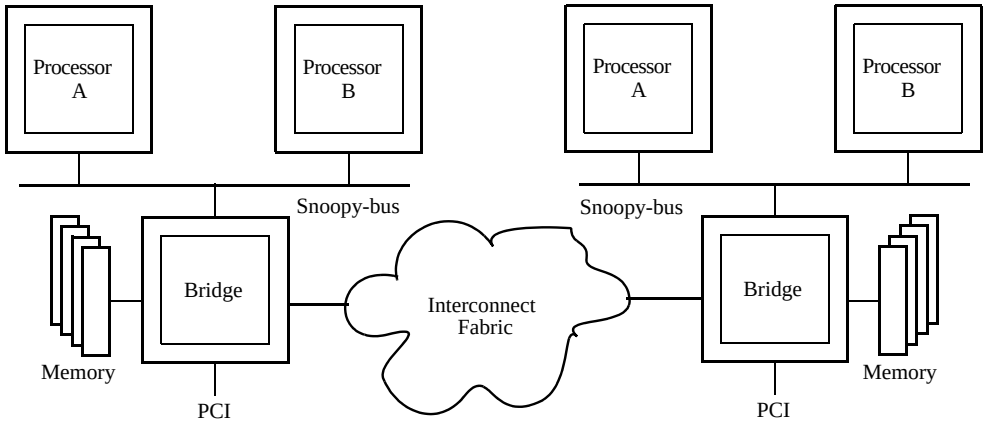


Figure 1-2. A Distributed Memory System

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”).

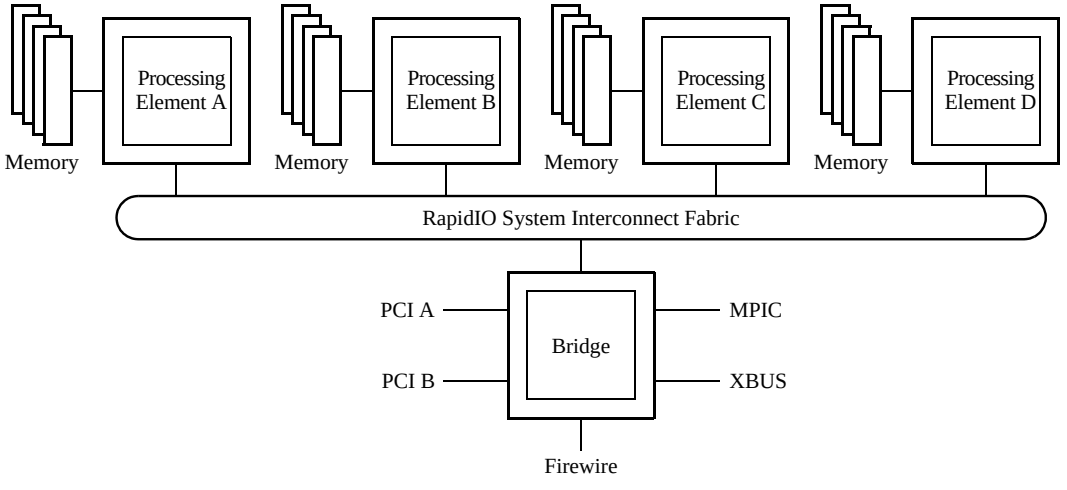


Figure 2-1. A Possible RapidIO-Based Computing System

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.

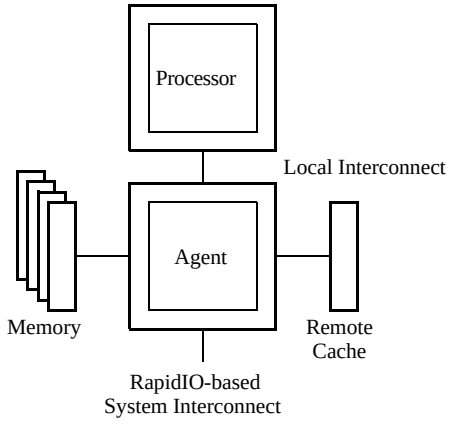


Figure 2-2. Processor-Memory Processing Element Example

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.

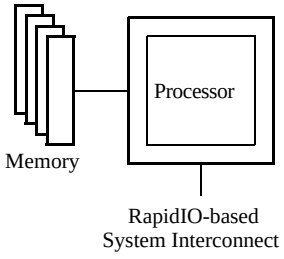


Figure 2-3. Integrated Processor-Memory Processing Element Example

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.

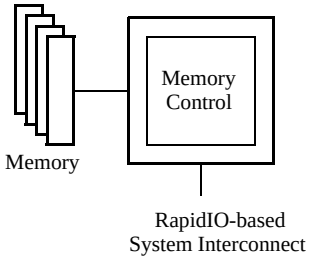


Figure 2-4. Memory-Only Processing Element Example

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.

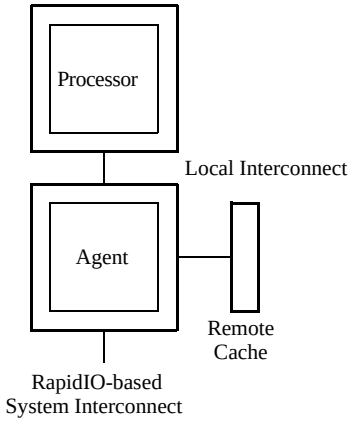


Figure 2-5. Processor-Only Processing Element Example

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.

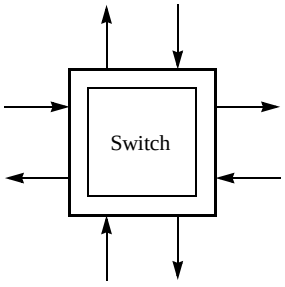


Figure 2-6. Switch Processing Element Example

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 page 35	Type 2 Section 4.2.6 page 51	Section 6.9 page 83
TLB invalidate-entry synchronize	TLBSYNC, RESPONSE	Software-maintained coherence of page table entries	Section 3.3.7 page 35	Type 2 Section 4.2.6 page 51	Section 6.9 page 83
Instruction cache invalidate	IKILL_HOME, IKILL_SHARER, RESPONSE,	Software-maintained coherence of instruction caches	Section 3.3.8 page 35	Type 2 Section 4.2.6 page 51	Section 6.7 page 79
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 page 36	Types 2 and 5: Section 4.2.6 page 51 and Section 4.2.8 page 52	Section 6.10 page 84
I/O read	IO_READ_HOME, IO_READ_ OWNER, INTERV, RESPONSE	CC-NUMA DMA, I/O DMA device support	Section 3.3.10 page 38	Types 1 and 2: Section 4.2.5 page 50 and Section 4.2.6 page 51	Section 6.11 page 88

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.

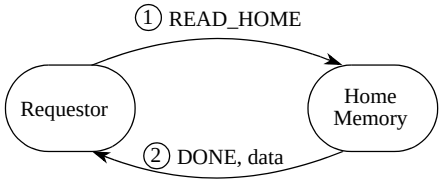


Figure 3-1. Read Operation to Remote Shared Coherence Granule

- 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.

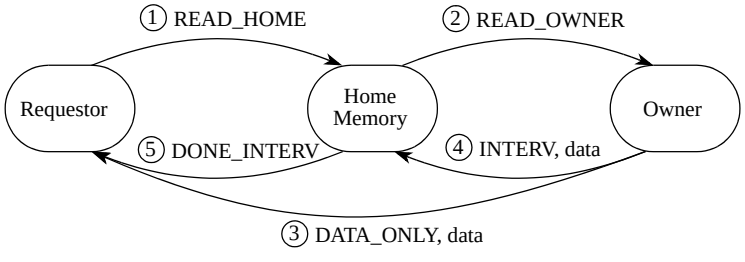


Figure 3-2. Read Operation to Remote Modified Coherence Granule

• 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.

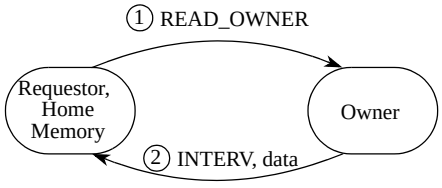


Figure 3-3. Read Operation to Local Modified Coherence Granule

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.

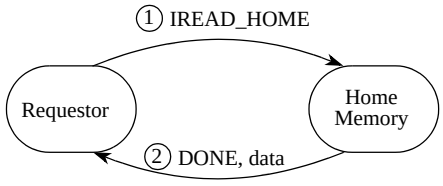


Figure 3-4. Instruction Read Operation to Remote Shared Coherence Granule

• 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.

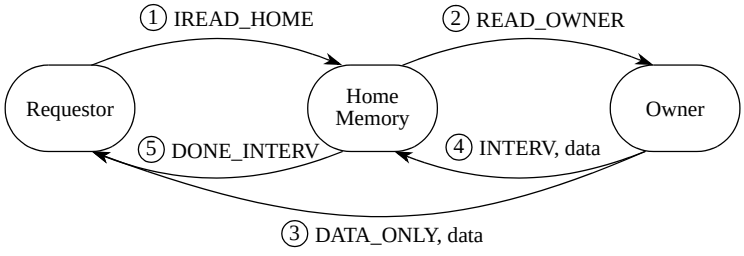


Figure 3-5. Instruction Read Operation to Remote Modified Coherence Granule

• 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.

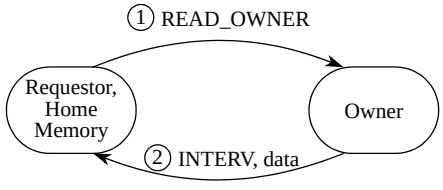


Figure 3-6. Instruction Read Operation to Local Modified Coherence Granule

• 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.

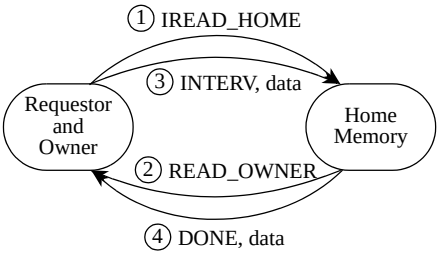


Figure 3-7. Instruction Read Operation Paradox Case

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.

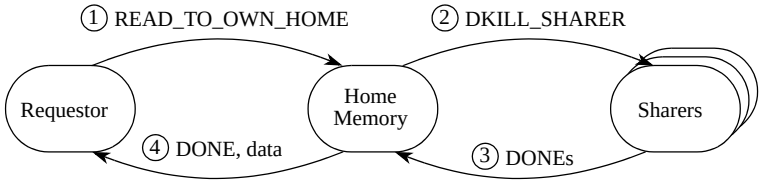


Figure 3-8. Read-for-Ownership Operation to Remote Shared Coherence Granule

- 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.

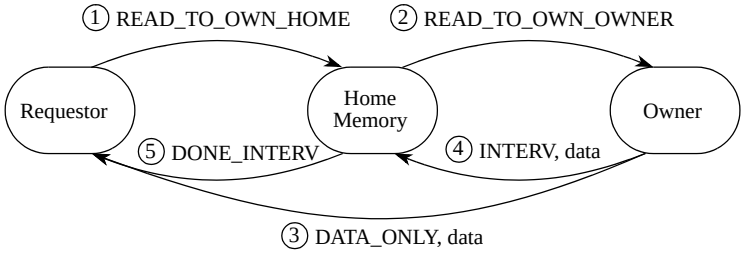


Figure 3-9. Read-for-Ownership Operation to Remote Modified Coherence Granule

- 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.

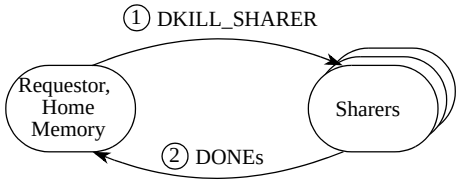


Figure 3-10. Read-for-Ownership Operation to Local Shared Coherence Granule

- 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.

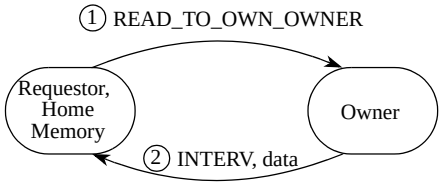


Figure 3-11. Read-for-Ownership Operation to Local Modified Coherence Granule

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.

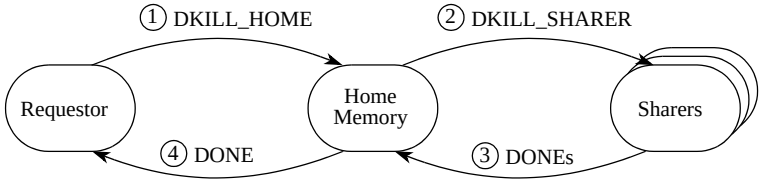


Figure 3-12. Data Cache Invalidate Operation to Remote Shared Coherence Granule

• 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).

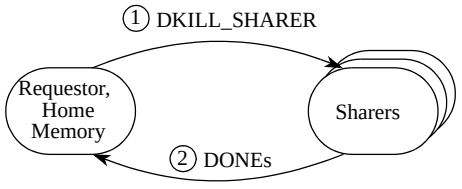


Figure 3-13. Data Cache Invalidate Operation to Local Shared Coherence Granule

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).

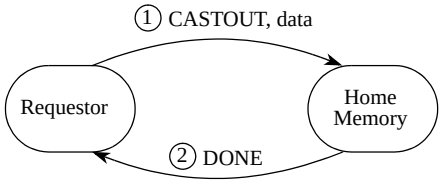


Figure 3-14. Castout Operation on Remote Modified Coherence Granule

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).

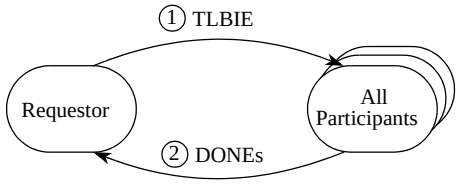


Figure 3-15. TLB Invalidate-Entry Operation

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).

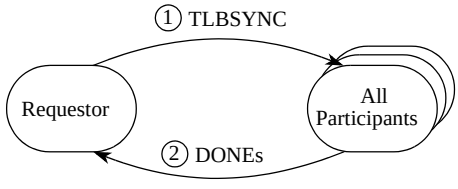


Figure 3-16. TLB Invalidate-Entry Synchronization Operation

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.)

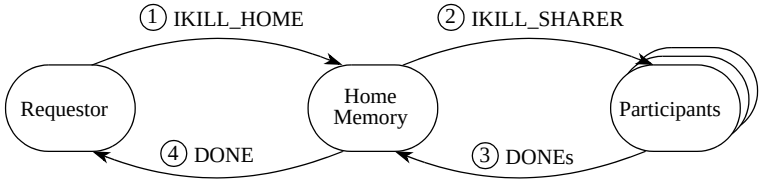


Figure 3-17. Instruction Cache Invalidate Operation to Remote Sharable Coherence Granule

- 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.

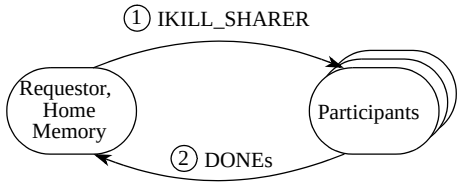


Figure 3-18. Instruction Cache Invalidate Operation to Local Sharable Coherence Granule

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).

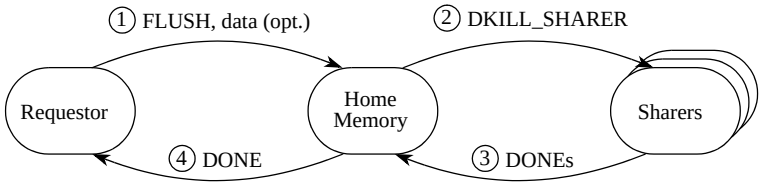


Figure 3-19. Flush Operation to Remote Shared Coherence Granule

• 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).

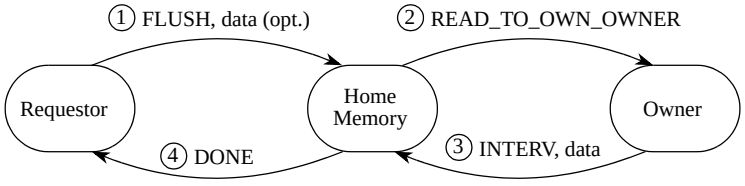


Figure 3-20. Flush Operation to Remote Modified Coherence Granule

• 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).

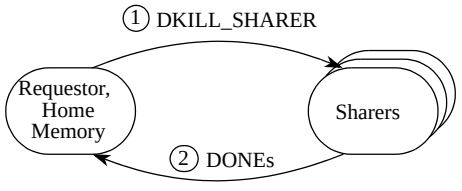


Figure 3-21. Flush Operation to Local Shared Coherence Granule

• 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).

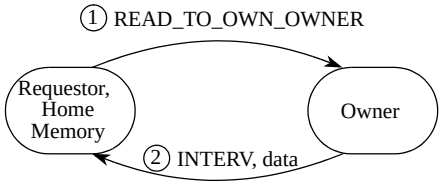


Figure 3-22. Flush Operation to Local Modified Coherence Granule

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).

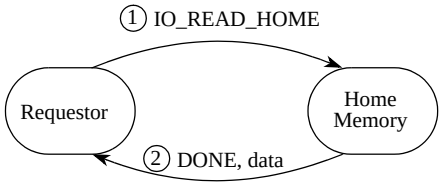


Figure 3-23. I/O Read Operation to Remote Shared Coherence Granule

- 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).

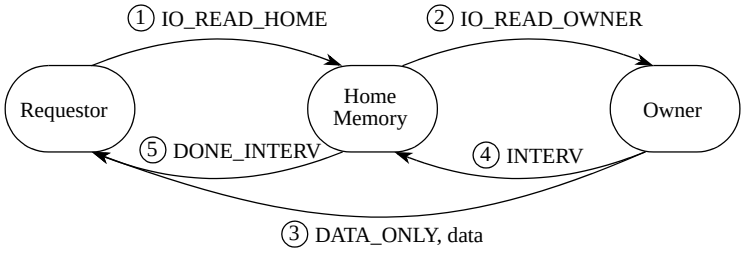


Figure 3-24. I/O Read Operation to Remote Modified Coherence Granule

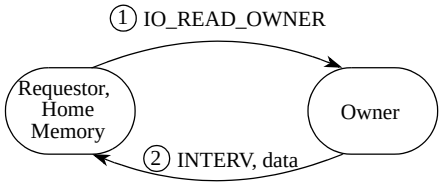
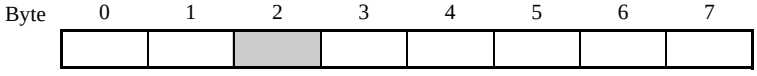


Figure 3-25. I/O Read Operation to Local Modified Coherence Granule

3.4 Endian, Byte Ordering, and Alignment

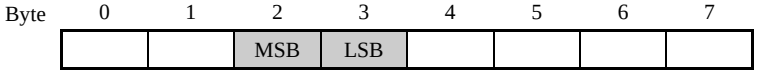
RapidIO has double-word (8-byte) aligned big-endian data payloads. This means that the RapidIO interface to devices that are little-endian shall perform the proper endian transformation to format a data payload.

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.



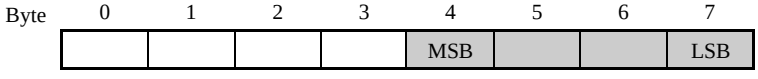
Byte address 0x0000_0002, the proper byte position is shaded.

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.

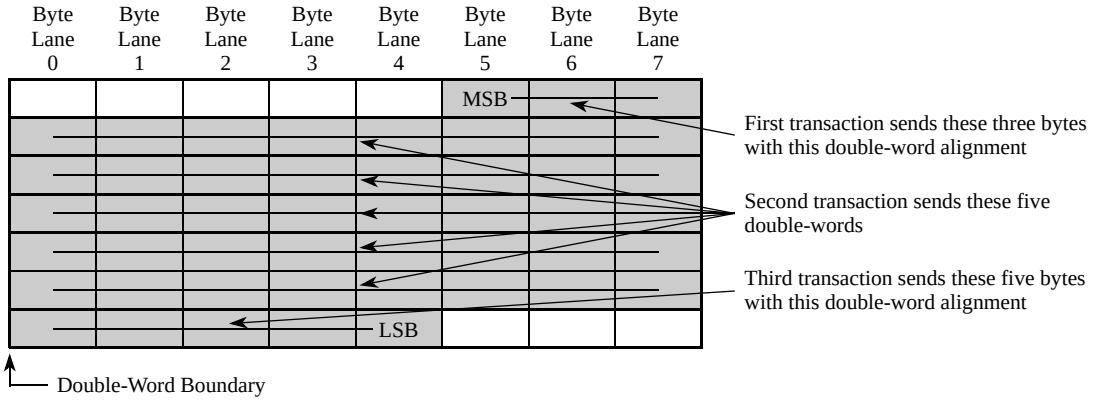


Figure 3-29. Data Alignment Example

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

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	2	3, 4
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

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

Field	Definition
srcTID	The packet's transaction ID.
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		

Table 4-7. Read Size (rdsize) Definitions (Continued)

wdptr	rdsize	Number of Bytes	Byte Lanes	Comment
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

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

Encoding	Transaction Field
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.

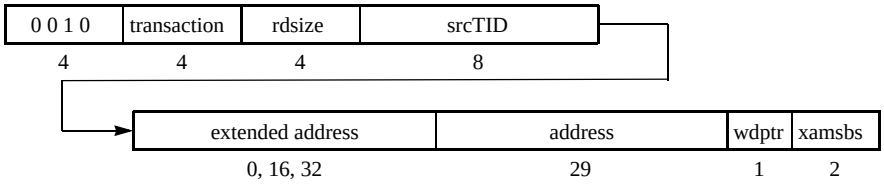


Figure 4-2. Type 2 Packet Bit Stream Format

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.

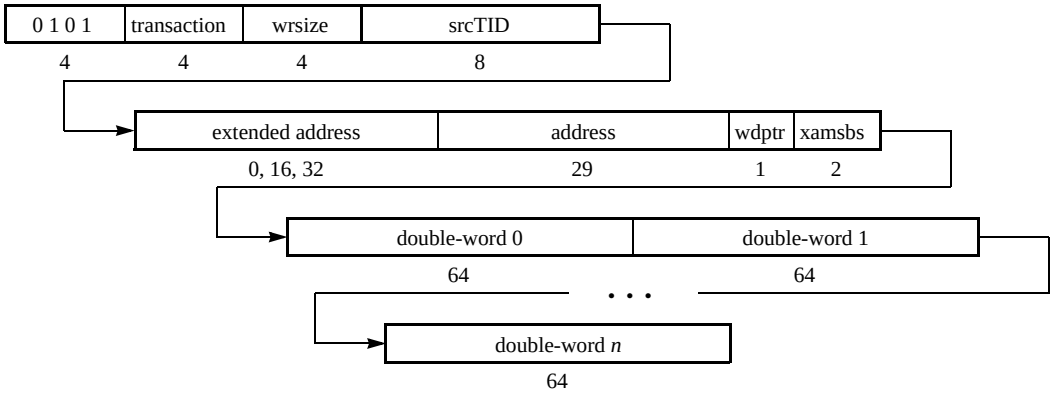


Figure 4-3. Type 5 Packet Bit Stream Format

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

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

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
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.

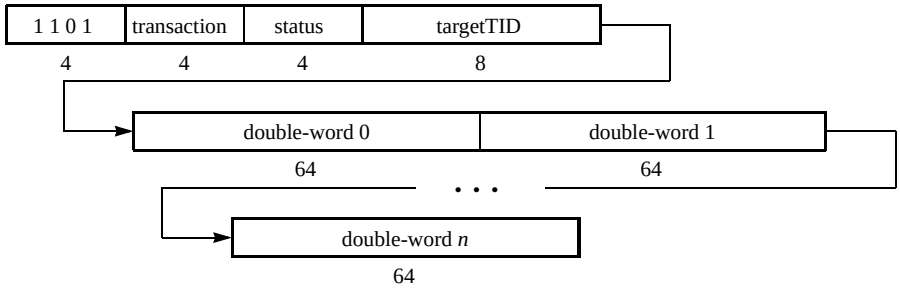


Figure 4-4. Type 13 Packet Bit Stream Format

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 and Bit 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.

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_collision An 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_RTYPE This is the response from the local agent to the local processor in response to a local request.

LOCAL_TTYPE This 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)
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).

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

Outstanding Request	Incoming Request	Resolution
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.

Blank page

Blank page

RapidIO™ Interconnect Specification Part 6: 1x/4x LP-Serial Physical Layer Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First release	12/17/2001
1.2	Technical changes: incorporate Rev. 1.1 errata rev. 1.1.1, errata 3	06/26/2002
1.3	Technical changes: incorporate Rev 1.2 errata 1 as applicable, the following errata showings: 03-03-00004.002, 03-07-00002.001, 03-12-00000.002, 03-12-00002.004, 04-02-00000.001, 04-05-00000.003, 04-05-00006.002 (partial), 04-05-00007.001 and the following new features showings: 02-03-0003.004, 02-06-00001.004, 04-08-00013.002, 04-09-00022.002 Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Overview

Chapter 2 Packets

Chapter 3 Control Symbols

Table of Contents

Chapter 4 PCS and PMA Layers

Table of Contents

Chapter 5 LP-Serial Protocol

Table of Contents

Chapter 6 LP-Serial Registers

Table of Contents

Table of Contents

Blank page

List of Figures

List of Figures

List of Tables

List of Tables

Chapter 1 Overview

1.1 Introduction

The *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification* addresses the physical layer requirements for devices utilizing an electrical serial connection medium. This specification defines a full duplex serial physical layer interface (link) between devices using unidirectional differential signals in each direction. Further, it allows ganging of four serial links for applications requiring higher link performance. It also defines a protocol for link management and packet transport over a link.

RapidIO systems are comprised of end point processing elements and switch processing elements. The RapidIO interconnect architecture is partitioned into a layered hierarchy of specifications which includes the Logical, Common Transport, and Physical layers. The Logical layer specifications define the operations and associated transactions by which end point processing elements communicate with each other. The Common Transport layer defines how transactions are routed from one end point processing element to another through switch processing elements. The Physical Layer defines how adjacent processing elements electrically connect to each other. RapidIO packets are formed through the combination of bit fields defined in the Logical, Common Transport, and Physical Layer specifications.

The RapidIO 1x/4x LP-Serial specification defines a protocol for packet delivery between serial RapidIO devices including packet and control symbol transmission, flow control, error management, and other device to device functions. A particular device may not implement all of the mode selectable features found in this document. See the appropriate user's manual or implementation specification for specific implementation details of a device.

The 1x/4x LP-Serial physical layer specification has the following properties:

- Embeds the transmission clock with data using an 8B/10B encoding scheme.
- Supports one serial differential pair, referred to as one lane, or four ganged serial differential pairs, referred to as four lanes, in each direction.
- Allows switching packets between RapidIO 1x/4x LP-Serial ports and *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification* ports without requiring packet manipulation.
- Employs similar retry and error recovery protocols as the RapidIO 8/16 LP-LVDS physical layer specification.

- Supports transmission rates of 1.25, 2.5, and 3.125 Gbaud (data rates of 1.0, 2.0, and 2.5 Gbps) per lane.

This specification first defines the individual elements that make up the link protocol such as packets, control symbols, and the serial bit encoding scheme. This is followed by a description of the link protocol. Finally, the control and status registers, signal descriptions, and electrical specifications are specified.

1.2 Packets

Chapter 2, “Packets” defines how a RapidIO 1x/4x LP-Serial packet is formed by prefixing a 10-bit physical layer header to the combined RapidIO transport and logical layer bit fields followed by an appended 16-bit CRC field.

This chapter shows the packet header format, the packet field definitions, the CRC error detection mechanism, and the packet alignment rules necessary to form LP-Serial packets.

1.3 Control Symbols

Chapter 3, “Control Symbols” defines the format of the two classes of control symbols (stype0 and stype1) used for packet acknowledgment, link utility functions, link maintenance, and packet delineation. A control symbol is a 24-bit entity (including a 5-bit CRC code). The control symbol is used for packet delineation and may also be embedded within a packet as well as sent when the link is idle.

Acknowledgment control symbols are used by processing elements to indicate packet transmission status. Utility control symbols are used to communicate buffer status and link recovery synchronization. Link maintenance control symbols are used by adjacent devices to communicate physical layer status, synchronization requests, and device reset.

1.4 PCS and PMA Layers

Chapter 4, “PCS and PMA Layers” describes the Physical Coding Sublayer (PCS) functionality as well as the Physical Media Attachment (PMA) functionality. The PCS layer functionality includes 8B/10B encoding scheme for embedding clock with data. It also gives transmission rules for the 1x and 4x interfaces and defines the link initialization sequence for clock synchronization.

The PMA (Physical Medium Attachment) function is responsible for serializing the 10-bit code-groups to and from the serial bitstream(s).

1.5 LP-Serial Protocol

Chapter 5, “LP-Serial Protocol” describes in detail how packets, control symbols, and the PCS/PMA layers are used to implement the physical layer protocol. This includes topics such as link initialization, link maintenance, error detection and recovery, flow control, and transaction delivery ordering.

1.6 LP-Serial Registers

Chapter 6, “LP-Serial Registers” describes the physical layer control and status register set. By accessing these registers a processing element may query the capabilities and status and configure another 1x/4x LP-Serial RapidIO processing element.

These registers utilize the Extended Features blocks and are accessed using *RapidIO Part 1: Input/Output Logical Specification* maintenance operations. Four types of RapidIO devices are defined in this section as follows:

- Generic End Point Processing Elements
- Generic End Point Processing Elements with software assisted error recovery
- Generic End Point Free Processing Elements (typically switch processing elements)
- Generic End Point Free Processing Elements with software assisted error recovery

1.7 Signal Descriptions

Chapter 7, “Signal Descriptions” contains the signal pin descriptions for a RapidIO LP-Serial end point device and shows connectivity between processing elements with 1x ports and processing elements with 4x ports.

1.8 AC Electrical Specifications

Chapter 8, “Electrical Specifications” describes the electrical specifications for the RapidIO 1x/4x LP-Serial device. This section defines two transmission types; short run and long run, as well as three speed grades (1.25 GHz, 2.5 GHz, and 3.125 GHz). This section also shows the required receiver eye diagrams for each link speed.

1.9 Interface Management

Annex A, “Interface Management (Informative)” contains information pertinent to interface management in a RapidIO system, including error recovery, link initialization, and packet retry state machines.

1.10 System Resources

Annex B, “Critical Resource Performance Limits (Informative)” contains a discussion on outstanding transactions and their relationship to transmission distance capability.

1.11 Manufacturability and Testability

Section Annex C, “Manufacturability and Testability (Informative)” recommends implementing to IEEE standard 1149.6 for improved manufacturing and manufacturing test.

Chapter 2 Packets

2.1 Introduction

This chapter specifies the LP-Serial packet format and the fields that are added by LP-Serial physical layer. These packets are fed into the PCS function explained in Chapter 4, “PCS and PMA Layers”.

2.2 Packet Field Definitions

This section specifies the bit fields added to a packet by the LP-Serial physical layer. These fields are required to implement the flow control, error management, and other specified system functions of the LP-Serial specification. The fields are specified in Table 2-1.

Table 2-1. Packet Field Definitions

Field	Description
ackID	Acknowledge ID is the packet identifier for acknowledgments back to the packet sender—see Section 5.4.2 for details concerning ackID functionality.
rsvd	The reserved bits are set to logic 0 when the packet is generated and ignored when a packet is received.
prio	Sets packet priority: 0b00 - lowest priority 0b01 - medium priority 0b10 - high priority 0b11 - highest priority See Section 5.4.3 for an explanation of prioritizing packets
CRF	Critical Request Flow is an optional bit that differentiates between flows of equal priority If Critical Request Flow is not supported, this bit is reserved See Section 5.4.3 for an explanation of prioritizing packets
CRC	Cyclic Redundancy Code used to detect transmission errors in the packet. See Section 2.4.1 for details on the CRC error detection scheme.

2.3 Packet Format

This section specifies the format of a LP-Serial packets. Figure 2-1 shows the format of the LP-Serial packet and how the physical layer ackID, rsvd, CRF, and prio fields are prefixed at the beginning of the packet and the 16-bit CRC field is appended to the end of the packet.

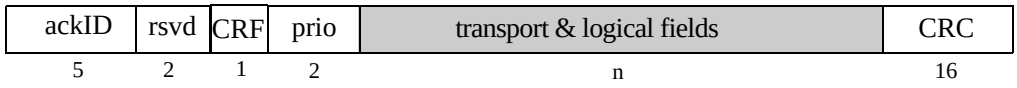


Figure 2-1. Packet Format

The unshaded fields are the fields added by the physical layer. The shaded field is the combined logical and transport layer bits and fields that are passed to the physical layer. The 2-bit rsvd field is required to make the packet length an integer multiple of 16 bits.

LP-Serial packets shall have a length that is an integer multiple of 32 bits. This sizing simplifies the design of port logic whose internal data paths are an integer multiple of 32 bits in width. Packets, as defined in this specification and the appropriate logical and transport layer specifications, have a length that is an integer multiple of 16 bits. This is illustrated in Figure 2-2. If the length of a packet defined by the above combination of specifications is an odd multiple of 16 bits, a 16-bit pad whose value is 0 (0x0000) shall be appended at the end of the packet such that the resulting padded packet is an integer multiple of 32 bits in length.

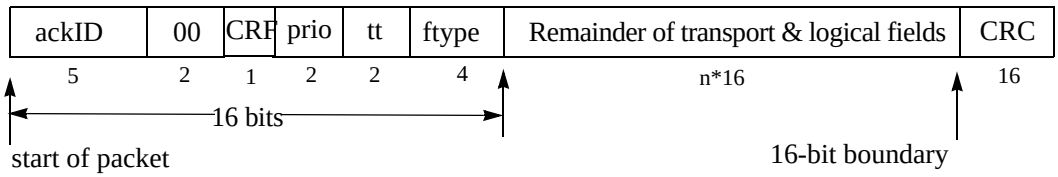


Figure 2-2. Packet Alignment

2.4 Packet Protection

A 16-bit CRC code is added to each packet by the LP-Serial physical layer to provide error detection. The code covers the entire packet except for the ackID field and one bit of the rsvd field, which are considered to be zero for the CRC calculations. Figure 2-3 shows the CRC coverage for the first 16 bits of the packet which contain the bits not covered by the code.

This structure allows the ackID to be changed on a link-by-link basis as the packet is transported across the fabric without requiring that the CRC be recomputed for each link. Since ackIDs on each link are assigned sequentially for each subsequent transmitted packet, an error in the ackID field is easily detected.

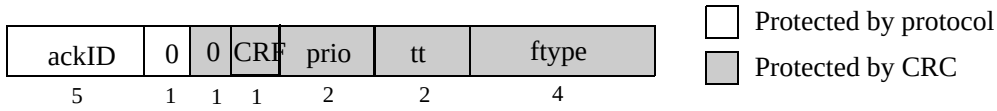


Figure 2-3. Error Coverage of First 16 Bits of Packet Header

2.4.1 Packet CRC Operation

The CRC is appended to a packet in one of two ways. For a packet whose length, exclusive of CRC, is 80 bytes or less, a single CRC is appended at the end of the logical fields. For packets whose length, exclusive of CRC, is greater than 80 bytes, a CRC is added after the first 80 bytes and a second CRC is appended at the end of the logical layer fields.

The second CRC value is a continuation of the first. The first CRC is included in the running calculation, meaning that the running CRC value is not reinitialized after it is inserted after the first 80 bytes of the packet. This allows intervening devices to regard the embedded CRC value as two bytes of packet payload for CRC checking purposes. If the CRC appended to the end of the logical layer fields does not cause the end of the resulting packet to align to a 32-bit boundary, a two byte pad of all logic 0s is postpended to the packet. The pad of logic 0s allows the CRC check to always be done at the 32-bit boundary. A corrupt pad may or may not cause a CRC error to be detected, depending upon the implementation.

The early CRC value can be used by the receiving processing element to validate the header of a large packet and start processing the data before the entire packet has been received, freeing up resources earlier and reducing transaction completion latency.

NOTE:

While the embedded CRC value can be used by a processing element to start processing the data within a packet before receiving the entire packet, it is possible that upon reception of the end of the packet the final CRC value for the packet is incorrect. This would result in a processing element that has processed data that may have been corrupted. Outside of the error recovery mechanism described in Section 5.11.2, the RapidIO Interconnect Specification does not address the occurrence of such situations nor does it suggest a means by which a processing element would handle such situations. Instead, the mechanism for handling this situation is left to be addressed by the device manufacturers for devices that implement the functionality of

early processing of packet data.

Figure 2-4 is an example of an unpadded packet of length less than or equal to 80 bytes.

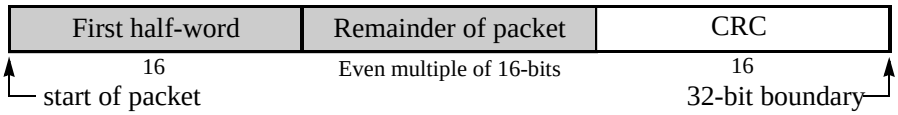


Figure 2-4. Unpadded Packet of Length 80 Bytes or Less

Figure 2-5 is an example of a padded packet of length less than or equal to 80 bytes.

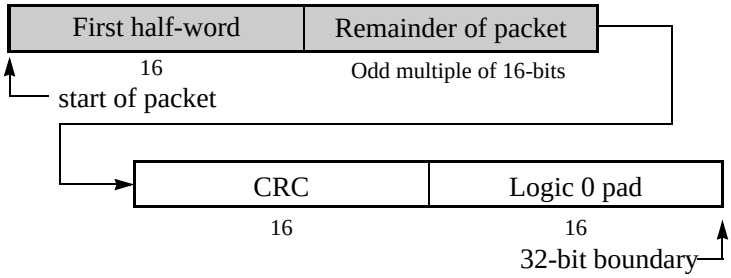


Figure 2-5. Padded Packet of Length 80 Bytes or Less

Figure 2-6 is an example of an unpadded packet of length greater than 80 bytes.

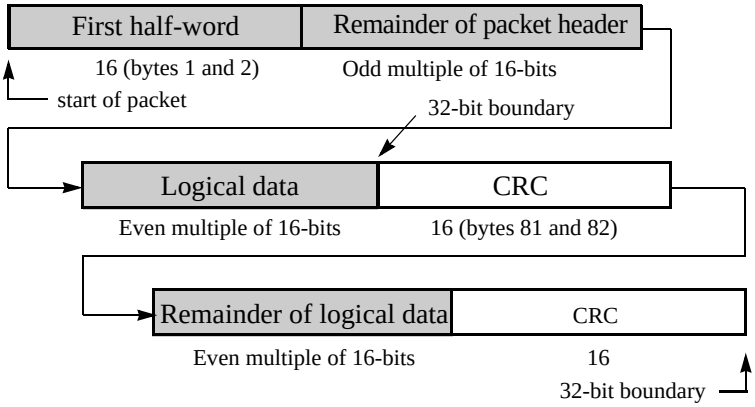


Figure 2-6. Unpadded Packet of Length Greater than 80 Bytes

Figure 2-7 is an example of a padded packet of length greater than 80 bytes.

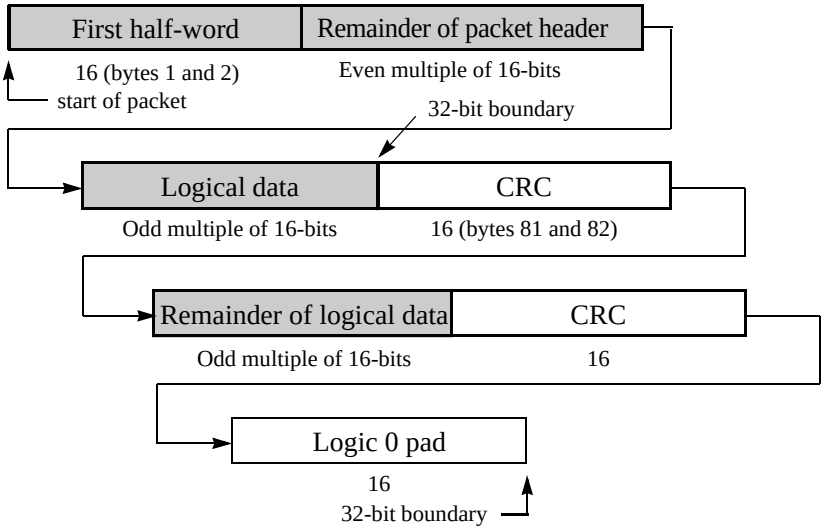


Figure 2-7. Padded Packet of Length Greater than 80 Bytes

2.4.2 16-Bit Packet CRC Code

The ITU polynomial $X^{16}+X^{12}+X^5+1$ shall be used to generate the 16-bit CRC for packets. The value of the CRC shall be initialized to 0xFFFF (all logic 1s) at the beginning of each packet. For the CRC calculation, the uncovered six bits are treated as logic 0s. As an example, a 16-bit wide parallel calculation is described in the equations in Table 2-2. Equivalent implementations of other widths can be employed.

Table 2-2. Parallel CRC Intermediate Value Equations

Check Bit	e00	e01	e02	e03	e04	e05	e06	e07	e08	e09	e10	e11	e12	e13	e14	e15
C00					x	x			x				x			
C01						x	x			x				x		
C02							x	x			x				x	
C03	x							x	x			x				x
C04	x	x			x	x				x						
C05		x	x			x	x				x					
C06	x		x	x			x	x				x				
C07	x	x		x	x			x	x				x			
C08	x	x	x		x	x			x	x				x		
C09		x	x	x		x	x			x	x				x	
C10			x	x	x		x	x			x	x				x
C11	x			x				x				x				
C12	x	x			x				x				x			

Table 2-2. Parallel CRC Intermediate Value Equations (Continued)

Check Bit	e00	e01	e02	e03	e04	e05	e06	e07	e08	e09	e10	e11	e12	e13	e14	e15
C13		x	x			x				x				x		
C14			x	x			x				x				x	
C15				x	x			x				x				x

where:

C00–C15 contents of the new check symbol

e00–e15 contents of the intermediate value symbol
e00 = d00 XOR c00
e01 = d01 XOR c01
through
e15 = d15 XOR c15

d00–d15 contents of the next 16 bits of the packet

c00–c15 contents of the previous check symbol

assuming the pipeline described in Figure 2-8

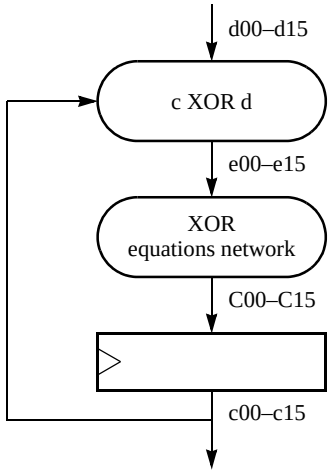


Figure 2-8. CRC Generation Pipeline

2.5 Maximum Packet Size

The maximum packet size permitted by the LP-Serial specification is 276 bytes. This includes all packet logical, transport, and physical layer header information, data payload, and required CRC bytes.

The maximum packet size of 276 bytes is achieved as shown below:

Table 2-3. Maximum Packet Size

Field	Size (bytes)	Layer	Notes
Header	2	Physical, Transport, Logical	
Source ID	2	Transport	
Destination ID	2	Transport	
Trans/wrsize	1	Logical	
srcTID	1	Logical	
Address	8	Logical	Includes Extended_address, Address, Wdptr, and Xambs
Payload	256	Logical	
CRC	4	Physical	Extra two CRC bytes for packets greater than 80 bytes
Total	276		

Blank page

Chapter 3 Control Symbols

3.1 Introduction

This chapter specifies RapidIO physical layer control symbols. Control symbols are the message elements used by ports connected by an LP-Serial link to manage all aspects of LP-Serial link operation. They are used for link maintenance, packet delimiting, packet acknowledgment, error reporting, and error recovery. For forward compatibility, control symbols received by a port with a reserved field encoding shall be ignored and not cause an error to be reported.

3.2 Control Symbol Field Definitions

This section describes the fields that make up the control symbols.

Table 3-1. Control Symbol Field Definitions

Field	Definition
stype0	Encoding for control symbols that make use of parameter0 and parameter1. Eight encodings are defined in Table 3-2.
parameter0	Used in conjunction with stype0 encodings. Reference Table 3-2 for the description of parameter0 encodings.
parameter1	Used in conjunction with stype0 encodings. Reference Table 3-2 for the description of parameter1 encodings.
stype1	Encoding for control symbols which make use of the cmd field. The eight encodings are defined in Table 3-6.
cmd	Used in conjunction with the stype1 field to define the link maintenance commands. Refer to Table 3-7 for the cmd field descriptions.
CRC	5-bit code used to detect transmission errors in control symbols. See Section 3.6 for details on the CRC error detection scheme.

3.3 Control Symbol Format

This section describes the general format of the LP-Serial control symbols. Figure 3-1 shows the control symbol format.

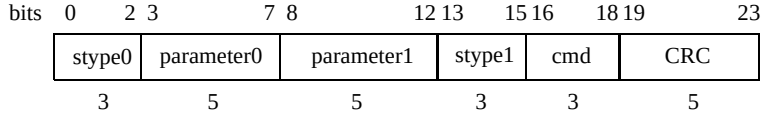


Figure 3-1. Packet-Retry Control Symbol Format

All control symbols follow the 24-bit control symbol format as detailed above. The

fields parameter0 and parameter1 are used by the functions encoded in the stype0 field. The cmd field is a modifier for the functions encoded in the stype1 field.

Control symbols can carry two functions, one encoded in the stype0 field and one encoded in the stype1 field. The functions encoded in stype0 are "status" functions that convey some type of status about the port transmitting the control symbol. The functions encoded in stype1 are requests to the receiving port or transmission delimiters.

A control symbol carrying one function is referred to using the name of the function it carries. A control symbol carrying two functions may be referred to using the name of either function that it carries. For example, a control symbol with stype0 set to packet-accepted and stype1 set to NOP is referred to a packet-accepted control symbol. A control symbol with stype0 set to packet-accepted and stype1 set to restart-from-retry is referred to as either a packet-accepted control symbol or a restart-from-retry control symbol depending on which name is appropriate for the context.

Control symbols are specified with the ability to carry two functions so that a packet acknowledgment and a packet delimiter can be carried in the same control symbol. Packet acknowledgment and packet delimiter control symbols constitute the vast majority of control symbol traffic on a busy link. Carrying an acknowledgment (or status) and a packet delimiter whenever possible in a single control symbol allows a significant reduction in link overhead traffic and an increase in the link bandwidth available for packet transmission.

3.4 Stype0 Control Symbols

The encoding and function of stype0 and the information carried in parameter0 and parameter1 for each stype0 encoding shall be as specified in Table 3-2.

Table 3-2. Stype0 Control Symbol Encoding

stype0	Function	Contents of		Reference
		Parameter0	Parameter1	
0b000	Packet-accepted	packet_ackID	buf_status	Section 3.4.1
0b001	Packet-retry	packet_ackID	buf_status	Section 3.4.2
0b010	Packet-not-accepted	packet_ackID	cause	Section 3.4.3
0b011	Reserved	-	-	-
0b100	Status	ackID_status	buf_status	Section 3.4.4
0b101	Reserved	-	-	-
0b110	Link-response	ackID_status	port_status	Section 3.4.5
0b111	Reserved	-	-	-

The status control symbol is the default stype0 encoding and is used when the control symbol does not convey another stype0 function. The following table

defines the parameters valid for stype0 control symbols.

Table 3-3. Stype0 Parameter Definitions

Parameter	Definition
packet_ackID	The ackID of the packet being acknowledged by an acknowledgment control symbol.
ackID_status	The value of ackID expected in the next packet the port receives. For example, a value of 0b00001 indicates the device is expecting to receive ackID 1.
buf_status	Specifies the number of maximum length packets that the port can accept without issuing a retry due to a lack of resources. The value of buf_status in a packet-accepted, packet-retry, or status control symbol is the number of maximum packets that can be accepted, inclusive of the effect of the packet being accepted or retried. Value 0-29: The encoding value specifies the number of new maximum sized packets the receiving device can receive. The value 0, for example, signifies that the downstream device has no available packet buffers (thus is not able to hold any new packets). Value 30: The value 30 signifies that the downstream device can receive 30 or more new maximum sized packets. Value 31: The downstream device can receive an undefined number of maximum sized packets, and relies on the retry protocol for flow control.

NOTE:

The following sections depict various control symbols. Since control symbols can contain one or two functions, shading in the figures is used to indicate which fields are applicable to that specific control symbol function.

3.4.1 Packet-Accepted Control Symbol

The packet-accepted control symbol indicates that the receiving device has taken responsibility for sending the packet to its final destination and that resources allocated by the sending device can be released. This control symbol shall be generated only after the entire packet has been received and found to be free of detectable errors. The packet-accepted control symbol format is displayed in Figure 3-2.

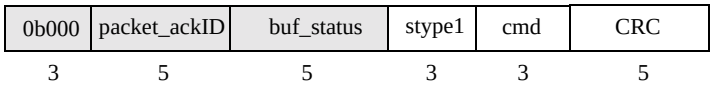


Figure 3-2. Packet-Accepted Control Symbol Format

3.4.2 Packet-Retry Control Symbol

A packet-retry control symbol indicates that the receiving device was not able to accept the packet due to some temporary resource conflict such as insufficient buffering and the sender should retransmit the packet. This control symbol format is displayed in Figure 3-3.

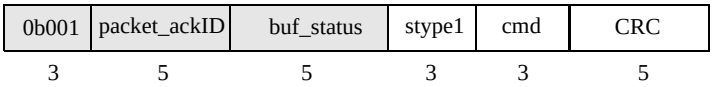


Figure 3-3. Packet-Retry Control Symbol Format

3.4.3 Packet-Not-Accepted Control Symbol

The packet-not-accepted control symbol is used to indicate to the sender of a packet why the packet was not accepted by the receiving port. As shown in Figure 3-4, the control symbol contains a cause field that indicates the reason for not accepting the packet and a packet_ackID field. If the receiving device is not able to specify the cause, or the cause is not one of defined options, the general error encoding shall be used.

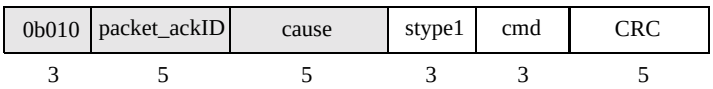


Figure 3-4. Packet-Not-Accepted Control Symbol Format

The cause field shall be used to display informational fields useful for debug. Table 3-4 displays the reasons a packet may not be accepted, indicated by the cause field.

Table 3-4. Cause Field Definition

Cause	Definition
0b00000	Reserved
0b00001	Received unexpected ackID on packet
0b00010	Received a control symbol with bad CRC
0b00011	Non-maintenance packet reception is stopped
0b00100	Received packet with bad CRC
0b00101	Received invalid character, or valid but illegal character
0b00110 - 0b11110	Reserved
0b11111	General error

3.4.4 Status Control Symbol

The status control symbol is the default stype0 encoding and is used when the control symbol does not convey another stype0 function. The status control symbol contains the ackID_status and the buf_status fields. The buf_status field indicates to the receiving port the number of maximum length packet buffers the sending port had available for packet reception at the time the control symbol was generated. The ackID_status field allows the receiving port to determine if it and the sending port are in sync with respect to the next ackID value the sending port expects to receive. The status control symbol format is shown in Figure 3-5 below.

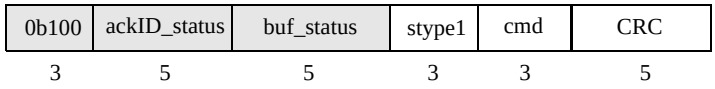


Figure 3-5. Status Control Symbol Format

3.4.5 Link-Response Control Symbol

The link-response control symbol is used by a device to respond to a link-request control symbol as described in the link maintenance protocol described in Section 5.5. The status reported in the status field is the status of the port at the time the associated input-status link-request control symbol was received.

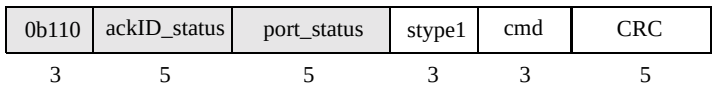


Figure 3-6. Link-Response Control Symbol Format

The port_status field of the link-response control symbol is defined in Table 3-5.

Table 3-5. Port_status Field Definitions

Port_status	Status	Description
0b00000		Reserved
0b00001		Reserved
0b00010	Error	The port has encountered an unrecoverable error and is unable to accept packets.
0b00011		Reserved
0b00100	Retry-stopped	The port has retried a packet and is waiting in the input retry-stopped state to be restarted.
0b00101	Error-stopped	The port has encountered a transmission error and is waiting in the input error-stopped state to be restarted.
0b00110 - 0b01111		Reserved
0b10000	OK	The port is accepting packets
0b10001 - 0b11111		Reserved

3.5 Stype1 Control Symbols

The encoding of stype1 and the function of the cmd field are defined in Table 3-6.

Table 3-6. Stype1 Control Symbol Encoding

stype1	stype1 Function	cmd	cmd Function	Packet Delimiter	Reference
0b000	Start-of-packet	0b000	-	yes	Section 3.5.1
0b001	Stomp	0b000	-	yes	Section 3.5.2
0b010	End-of-packet	0b000	-	yes	Section 3.5.3
0b011	Restart-from-retry	0b000	-	*	Section 3.5.4
0b100	Link-request	0b000 - 0b010	Reserved	*	-
		0b011	Reset-device		Section 3.5.5.1
		0b100	Input-status		Section 3.5.5.2
		0b101- 0b111	Reserved		-
0b101	Multicast-event	0b000	-	No	Section 3.5.6
0b110	Reserved	0b000	-	No	-
0b111	NOP (Ignore) **	0b000	-	No	-

Note: * denotes that restart-from-retry and link-request control symbols may only be packet delimiters if a packet is in progress.

Note: ** NOP (Ignore) is not defined as a control symbol, but is the default value when the control symbol does not convey another stype1 function.

NOTE:

The following sections depict various control symbols. Since control symbols can contain one or two functions, shading in the figures is used to indicate which fields are applicable to that specific control symbol function.

3.5.1 Start-of-Packet Control Symbol

The start-of-packet control symbol format is shown in Figure 3-7 below.

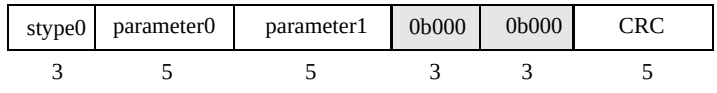


Figure 3-7. Start-of-Packet Control Symbol Format

3.5.2 Stomp Control Symbol

The stomp control symbol is used to cancel a partially transmitted packet. The protocol for packet cancelation is specified in Section 5.8. The stomp control symbol format is shown in Figure 3-8 below.

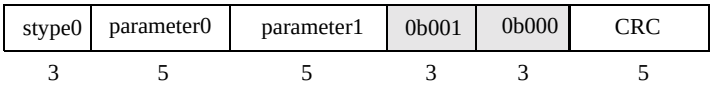


Figure 3-8. Stomp Control Symbol Format

3.5.3 End-of-Packet Control Symbol

The end-of-packet control symbol format is shown in Figure 3-9 below.

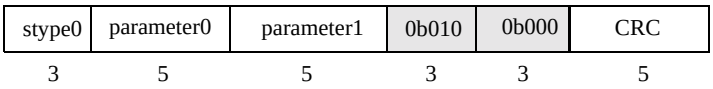


Figure 3-9. End-of-Packet Control Symbol Format

3.5.4 Restart-From-Retry Control Symbol

The restart-from-retry control symbol cancels a current packet and may also be transmitted on an idle link. This control symbol is used to mark the beginning of packet retransmission, so that the receiver knows when to start accepting packets after the receiver has requested a packet to be retried. The control symbol format is shown in Figure 3-10 below.

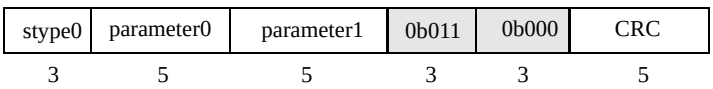


Figure 3-10. Restart-From-Retry Control Symbol Format

3.5.5 Link-Request Control Symbol

A link-request control symbol is used by a device to either issue a command to the connected device or request its input port status. A link-request control symbol always cancels a packet whose transmission is in progress and can also be sent between packets. Under error conditions, a link-request/input-status control symbol acts as a link-request/restart-from-error control symbol as described in Section 5.11.2.1, “Recoverable Errors.” This control symbol format is displayed in Figure 3-11.

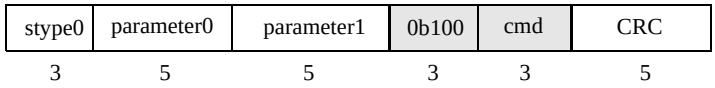


Figure 3-11. Link-Request Control Symbol Format

The cmd, or command, field of the link-request control symbol format is defined in Table 3-7 below.

Table 3-7. Cmd Field Definitions

cmd Encoding	Command Name	Description	Reference
0b000-0b010	-	Reserved	
0b011	Reset-device	Reset the receiving device	Section 3.5.5.1
0b100	Input-status	Return input port status; functions as a link request (restart-from-error) control symbol under error conditions	Section 3.5.5.2
0b101-0b111	-	Reserved	

3.5.5.1 Reset-Device Command

The reset-device command causes the receiving device to go through its reset or power-up sequence. All state machines and the configuration registers reset to the original power on states. The reset-device command does not generate a link-response control symbol.

Due to the undefined reliability of system designs it is necessary to put a safety lockout on the reset function of the link-request control symbol. A device receiving a reset-device command in a link-request control symbol shall not perform the reset function unless it has received four reset-device commands in a row without any other intervening packets or control symbols, except status control symbols. This will prevent spurious reset commands from inadvertently resetting a device.

When issuing a reset with four consecutive reset commands, care must be taken to account for all effects associated with the reset event. Consult *RapidIO Part 8: Error Management Extensions Specification* for more information.

3.5.5.2 Input-Status Command

The input-status command requests the receiving device to return the ackID value it expects to next receive from the sender on its input port and the current input port operational status for informational purposes. This command causes the receiver to flush its output port of all control symbols generated by packets received before the input-status command. Flushing the output port is implementation dependent and may result in either discarding the contents of the receive buffers or sending the control symbols on the link. The receiver then responds with a link-response control symbol.

3.5.6 Multicast-Event Control Symbol

The multicast-event control symbol differs from other control symbols in that it carries information not related to the link carrying the control symbol. The multicast-event control symbol allows the occurrence of a user-defined system event to be multicast throughout a system. Refer to Section 5.3.4 for more details on Multicast-Events.

The multicast-event control symbol format is shown in Figure 3-12 below.

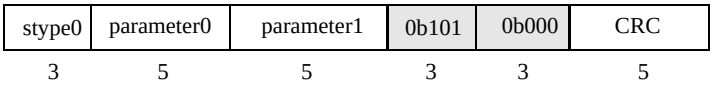


Figure 3-12. Multicast-Event Control Symbol Format

3.6 Control Symbol Protection

The 5-bit CRC shall be computed over control symbol bits 0 through 18 and provides 5-bit burst error detection for the entire 24-bit control symbol.

3.6.1 CRC-5 Code

The ITU polynomial $X^5+X^4+X^2+1$ shall be used to generate the 5-bit CRC for control symbols. The CRC check bits c0, c1, c2, c3, and c4 occupy the last 5 bits of a control symbol. It should be noted that the 5-bit CRC must be generated by each transmitter and verified by each receiver. Before the 5-bit CRC is computed, the CRC should be set to all 1's or 0b11111. In order to provide maximum implementation flexibility for all types of designs, a 20^{th} bit has been added. For all computations, the 20^{th} bit shall be the last bit applied and shall be set to a logic 0 (0b0).

3.6.2 CRC-5 Parallel Code Generation

Since it is often more efficient to implement a parallel CRC algorithm rather than a serial, examples of the equations for a complete, 19-bit single-stage parallel implementation is shown in shown in Table 3-8. Since only a single stage is used, the effect of both setting the initial CRC to all 1's (0b11111) and a 20^{th} bit set to logic 0 (0b0) have been included in the equations.

In Table 3-8, an "x" means that the data input should be an input to the Exclusive-OR necessary to compute that particular bit of the CRC. A "!x", means that bit 18 being applied to the CRC circuit must be inverted. Figure 3-13 shows the 19-bits that the CRC covers and how they should be applied to the circuit. As seen in Figure 3-13, bits are labeled with 0 on the left and 18 on the right. Bit 0, from the stype0 field, would apply to D0 in Table 3-8 and bit 18, from the cmd field, would apply to D18 in Table 3-8. Once completed, the 5-bit CRC is appended to the control symbol.

CRC Checksum	Control Symbol Data For CRC																	
Bits	**D 0**	**D 1**	**D 2**	**D 3**	**D 4**	**D 5**	**D 6**	**D 7**	**D 8**	**D 9**	**D 10**	**D 11**	**D 12**	**D 13**	**D 14**	**D 15**	**D 16**	**D 17**	**D 18**
4	x		x	x	x					x		x			x	x		x	x
3		x	x	x					x		x			x	x		x	x	!x
2		x		x	x			x				x	x	x	x		x		!x
1	x		x	x			x				x	x	x	x		x		x	!x
0	x	x		x	x	x					x		x			x	x		x

Table 3-8. Parallel CRC Equations

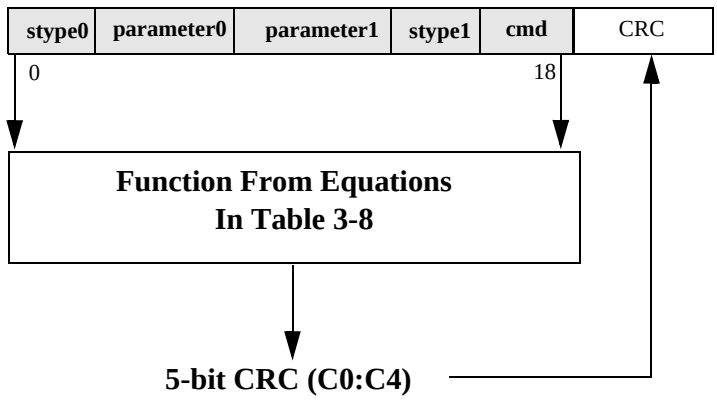


Figure 3-13. 5-bit CRC Implementation

Chapter 4 PCS and PMA Layers

4.1 Introduction

This chapter specifies the functions provided by the Physical Coding Sublayer (PCS) and Physical Media Attachment (PMA) sublayer. (The PCS and PMA terminology is adopted from IEEE 802.3). The topics include 8B/10B encoding, character representation, serialization of the data stream, code-groups, columns, link transmission rules, idle sequences, and link initialization.

The concept of lanes is used to describe the width of a LP-Serial link. A lane is defined as one unidirectional differential pair in each direction. RapidIO LP-Serial defines two link widths. The 1x LP-Serial link is a one-lane link and the 4x LP-Serial link is a 4-lane link. Wider links are possible, but are left for future work.

4.2 PCS Layer Functions

The Physical Coding Sublayer (PCS) function is responsible for idle sequence generation, lane striping, and encoding for transmission and decoding, lane alignment, and destriping on reception. The PCS uses an 8B/10B encoding for transmission over the link.

The PCS layer also provides mechanisms for determining the operational mode of the port as 4-lane or 1-lane operation, and means to detect link states. It provides for clock difference tolerance between the sender and receiver without requiring flow control.

The PCS layer performs the following transmit functions:

- Dequeues packets and delimited control symbols awaiting transmission as a character stream.
- Stripes the transmit character stream across the available lanes.
- Generates the idle sequence and inserts it into the transmit character stream for each lane when no packets or delimited control symbols are available for transmission.
- Encodes the character stream of each lane independently into 10-bit parallel code-groups.
- Passes the resulting 10-bit parallel code-groups to the PMA.

The PCS layer performs the following receive functions:

- Decodes the received stream of 10-bit parallel code-groups for each lane independently into characters.
- Marks characters decoded from invalid code-groups as invalid.
- If the link is using more than one lane, aligns the character streams to eliminate the skew between the lanes and reassembles (destripes) the character stream from each lane into a single character stream.
- Delivers the decoded character stream of packets and delimited control symbols to the higher layers.

4.3 PMA Layer Functions

The PMA (Physical Medium Attachment) function is responsible for serializing 10-bit parallel code-groups to/from a serial bitstream on a lane-by-lane basis. Upon receiving data, the PMA function provides alignment of the received bitstream to 10-bit code-group boundaries, independently on a lane-by-lane basis. It then provides a continuous stream of 10-bit code-groups to the PCS, one stream for each lane. The 10-bit code-groups are not observable by layers higher than the PCS.

4.4 Definitions

Definitions of terms used in this specification are provided below.

Byte: An 8-bit unit of information. Each bit of a byte has the value 0 or 1.

Character: A 9-bit entity comprised of an information byte and a control bit that indicates whether the information byte contains data or control information. The control bit has the value D or K indicating that the information byte contains respectively data or control information.

D-character: A character whose control bit has the value “D”.

K-character: A character whose control bit has the value “K”. Also referred to as a special character.

Code-group: A 10-bit entity that is the result of 8B/10B encoding a character.

Column: A group of four characters that are transmitted simultaneously on a 4x (4 lane) link.

Comma: A 7-bit pattern, unique to certain 8B/10B special code-groups, that is used by a receiver to determine code-group boundaries. See more in “Section 4.5.7.4, Sync (/K/)” on page 50 and Table 4-2, on page 48.

Idle sequence: The sequence of characters (code-groups after encoding) that is transmitted when a packet or control symbol is not being transmitted. The idle sequence allows the receiver to maintain bit synchronization and code-group alignment in between packets and control symbols.

Lane Alignment: The process of eliminating the skew between the lanes of a 4-lane LP-Serial link such that the characters transmitted as a column by the sender are output by

the alignment process of receiver as a column. Without lane alignment, the characters transmitted as a column might be scattered across several columns output by the receiver. The alignment process uses the columns of “A” special characters transmitted as part of the idle sequence.

Striping: The method used on a 4x link to send data across four lanes simultaneously. The character stream is *striped* across the lanes, on a character-by-character basis, starting with lane 0, to lane 1, to lane 2, to lane3, and wrapping back with the 5th character to lane 0.

4.5 8B/10B Transmission Code

The 8B/10B transmission code used by the PCS encodes 9-bit characters (8 bits of information and a control bit) into 10-bit code-groups for transmission and reverses the process on reception. Encodings are defined for 256 data characters and 12 special (control) characters.

The code-groups used by the code have either an equal number of ones and zeros (balanced) or the number of ones differs from the number of zeros by two (unbalanced). This selection of code-groups guarantees a minimum of two transitions, 0 to 1 or 1 to 0, within each code-group and it also eases the task of maintaining balance. Characters are encoded into either a single balanced code-group or a pair of unbalanced code-groups. The members of each code-group pair are the logical complement of each other. This allows the encoder, when selecting an unbalanced code-group, to select a code-group unbalanced toward ones or unbalanced toward zeros, depending on which is required to maintain the 0/1 balance of the encoder output code-group stream.

The 8B/10B code has the following properties.

- Sufficient bit transition density (3 to 8 transitions per code-group) to allow clock recovery by the receiver.
- Special code-groups that are used for establishing the receiver synchronization to the 10-bit code-group boundaries, delimiting control symbols and maintaining receiver bit and code-group boundary synchronization.
- Balanced. (can be AC coupled)
- Detection of single and some multiple-bit errors.

4.5.1 Character and Code-Group Notation

The description of 8B/10B encoding and decoding uses the following notation for characters, code-group and their bits.

The information bits ([0-7]) of an unencoded character are denoted with the letters “A” through “H” where the letter “H” denotes the most significant information bit (RapidIO bit 0) and the letter “A” denotes the least significant information bit (RapidIO bit 7). This is shown in Figure 4-1.

Each data character has a representation of the form Dx.y where x is the decimal value of the least significant 5 information bits EDCBA, and y is the decimal value of the most significant 3 information bits HGF as shown in Figure 4-1. Each special character has a similar representation of the form Kx.y.

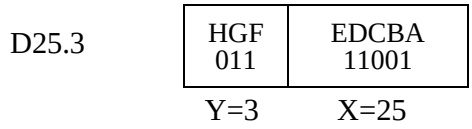


Figure 4-1. Character Notation Example (D25.3)

The output of the 8B/10B encoding process is a 10-bit code-group. The bits of a code-group are denoted with the letters “a” through “j”. The bits of a code-group are all of equal significance, there is no most significant or least significant bit. The ordering of the code-group bits is shown in Figure 4-2.

The code-groups corresponding to the data character Dx.y is denoted by /Dx.y/. The code-groups corresponding to the special character Kx.y is denoted by /Kx.y/.

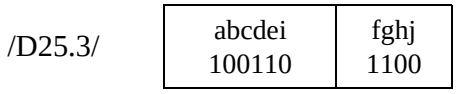


Figure 4-2. Code-Group Notation Example (/D25.3/)

4.5.2 Running Disparity

The 8B/10B encoding and decoding functions use a binary variable called running disparity. The variable can have a value of either positive (RD+) or negative (RD-). The encoder and decoder each have a running disparity variable for each lane which are all independent of each other.

The primary use of running disparity in the encoding process is to keep track of whether the decoder has output more ones or more zeros. The current value of encoder running disparity is used to select the which unbalanced code-group will be used when the encoding for a character requires a choice between two unbalanced code-groups.

The primary use of running disparity in the decoding process is to detect errors. Given a value of decoder running disparity, only (256 + 12) = 268 of the 1024 possible code-group values have defined decodings. The remaining 756 possible code-group values have no defined decoding and represent errors, either in that code-group or in an earlier code-group.

4.5.3 Running Disparity Rules

After power-up and before the port is operational, both the transmitter (encoder) and receiver (decoder) must establish current values of running disparity.

The transmitter shall use a negative value as the initial value for the running disparity for each lane.

The receiver may use either a negative or positive initial value of running disparity for each lane.

The following algorithm shall be used for calculating the running disparity for each lane. In the encoder, the algorithm operates on the code-group that has just been generated by the encoder. In the receiver, the algorithm operates on the received code-group that has just been decoded by the decoder.

Each code-group is divided to two sub-blocks as shown in Figure 4-2, where the first six bits (abcdei) form one sub-block (6-bit sub-block) and the second four bits (fghj) form a second sub-block (4-bit sub-block). Running disparity at the beginning of the 6-bit sub-block is the running disparity at the end of the previous code-group. Running disparity at the beginning of the 4-bit sub-block is the running disparity at the end of the 6-bit sub-block. Running disparity at the end of the code-group is the running disparity at the end of the 4-bit sub-block.

The sub-block running disparity shall be calculated as follows:

1. The running disparity is positive at the end of any sub-block if the sub-block contains more 1s than 0s. It is also positive at the end of a 4-bit sub-block if the sub-block has the value 0b0011 and at the end of a 6-bit sub-block if the sub-block has the value 0b000111.
2. The running disparity is negative at the end of any sub-block if the sub-block contains more 0s than 1s. It is also negative at the end of a 4-bit sub-block if the sub-block has the value 0b1100 and at the end of a 6-bit sub-block if the sub-block has the value 0b111000.
3. In all other cases, the value of the running disparity at the end of the sub-block is running disparity at the beginning of the sub-block (the running disparity is unchanged).

4.5.4 8B/10B Encoding

The 8B/10B encoding function encodes 9-bit characters into 10-bit code-groups.

The encodings for the 256 data characters (Dx.y) are specified in Table 4-1. The encodings for the 12 special characters (Kx.y) are specified in Table 4-2. Both tables have two columns of encodings, one marked RD- and one marked RD+. When encoding a character, the code-group in the RD- column is selected if the current value of encoder running disparity is negative and the code-group in the RD+ column is selected if the current value of encoder running disparity is positive.

Data characters (Dx.y) shall be encoded according to Table 4-1 and the current value of encoder running disparity. Special characters (Kx.y) shall be encoded according to Table 4-2 and the current value of encoder running disparity. After each character is encoded, the resulting code-group shall be used by the encoder to update the running disparity according to the rules in Section 4.5.3, “Running Disparity Rules.

4.5.5 Transmission Order

The parallel 10-bit code-group output of the encoder shall be serialized and transmitted with bit “a” transmitted first and a bit ordering of “abcdeifghj”. This is shown in Figure 4-3.

Figure 4-3 gives an overview of a character passing through the encoding, serializing, transmission, deserializing, and decoding processes. The left side of the figure shows the transmit process of encoding a character stream using 8B/10B encoding and the 10-bit serialization. The right side shows the reverse process of the receiver deserializing and using 8B/10B decoding on the received code-groups.

The dotted line shows the functional separation between the PCS layer, that provides 10-bit code-groups, and the PMA layer that serializes the code-groups.

The drawing also shows on the receive side the bits of a special character containing the comma pattern that is used by the receiver to establish 10-bit code-boundary synchronization.

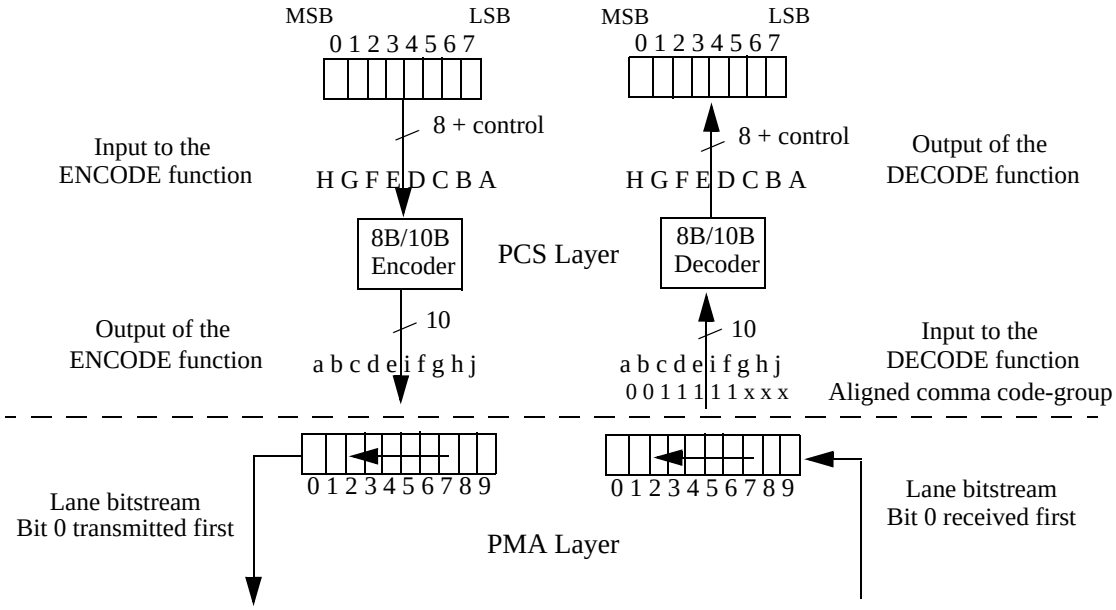


Figure 4-3. Lane Encoding, Serialization, Deserialization, and Decoding Process

4.5.6 8B/10B Decoding

The 8B/10B decoding function decodes received 10-bit code-groups into 9-bit characters, detects received code-groups that have no defined decoding and marks the resulting characters in the output stream of the decode as invalid character (INVALID).

The decoding function uses Table 4-1, Table 4-2 and the current value of the decoder running disparity. To decode a received code-group, the decoder shall select the RD- column of Table 4-1 and Table 4-2 if the current value of the decoder running disparity is negative or shall select the RD+ column if the value is positive. The decoder shall then compare the received code-group with the code-groups in the selected column of both tables. If a match is found, the code-group is decoded to the associated character. If no match is found, the code-group is decoded to a character that is flagged in some manner as invalid. After each code-group is decoded, the decoded code-group shall be used by the decoder to update the decoder running disparity according to the rules in Section 4.5.3, “Running Disparity Rules.

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D0.0	00	000 00000	100111 0100	011000 1011
D1.0	01	000 00001	011101 0100	100010 1011
D2.0	02	000 00010	101101 0100	010010 1011
D3.0	03	000 00011	110001 1011	110001 0100
D4.0	04	000 00100	110101 0100	001010 1011
D5.0	05	000 00101	101001 1011	101001 0100
D6.0	06	000 00110	011001 1011	011001 0100
D7.0	07	000 00111	111000 1011	000111 0100
D8.0	08	000 01000	111001 0100	000110 1011
D9.0	09	000 01001	100101 1011	100101 0100
D10.0	0A	000 01010	010101 1011	010101 0100
D11.0	0B	000 01011	110100 1011	110100 0100
D12.0	0C	000 01100	001101 1011	001101 0100
D13.0	0D	000 01101	101100 1011	101100 0100
D14.0	0E	000 01110	011100 1011	011100 0100
D15.0	0F	000 01111	010111 0100	101000 1011
D16.0	10	000 10000	011011 0100	100100 1011
D17.0	11	000 10001	100011 1011	100011 0100
D18.0	12	000 10010	010011 1011	010011 0100
D19.0	13	000 10011	110010 1011	110010 0100
D20.0	14	000 10100	001011 1011	001011 0100

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D21.0	15	000 10101	101010 1011	101010 0100
D22.0	16	000 10110	011010 1011	011010 0100
D23.0	17	000 10111	111010 0100	000101 1011
D24.0	18	000 11000	110011 0100	001100 1011
D25.0	19	000 11001	100110 1011	100110 0100
D26.0	1A	000 11010	010110 1011	010110 0100
D27.0	1B	000 11011	110110 0100	001001 1011
D28.0	1C	000 11100	001110 1011	001110 0100
D29.0	1D	000 11101	101110 0100	010001 1011
D30.0	1E	000 11110	011110 0100	100001 1011
D31.0	1F	000 11111	101011 0100	010100 1011
D0.1	20	001 00000	100111 1001	011000 1001
D1.1	21	001 00001	011101 1001	100010 1001
D2.1	22	001 00010	101101 1001	010010 1001
D3.1	23	001 00011	110001 1001	110001 1001
D4.1	24	001 00100	110101 1001	001010 1001
D5.1	25	001 00101	101001 1001	101001 1001
D6.1	26	001 00110	011001 1001	011001 1001
D7.1	27	001 00111	111000 1001	000111 1001
D8.1	28	001 01000	111001 1001	000110 1001
D9.1	29	001 01001	100101 1001	100101 1001
D10.1	2A	001 01010	010101 1001	010101 1001
D11.1	2B	001 01011	110100 1001	110100 1001
D12.1	2C	001 01100	001101 1001	001101 1001
D13.1	2D	001 01101	101100 1001	101100 1001
D14.1	2E	001 01110	011100 1001	011100 1001
D15.1	2F	001 01111	010111 1001	101000 1001
D16.1	30	001 10000	011011 1001	100100 1001
D17.1	31	001 10001	100011 1001	100011 1001
D18.1	32	001 10010	010011 1001	010011 1001
D19.1	33	001 10011	110010 1001	110010 1001
D20.1	34	001 10100	001011 1001	001011 1001
D21.1	35	001 10101	101010 1001	101010 1001
D22.1	36	001 10110	011010 1001	011010 1001
D23.1	37	001 10111	111010 1001	000101 1001
D24.1	38	001 11000	110011 1001	001100 1001

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D25.1	39	001 11001	100110 1001	100110 1001
D26.1	3A	001 11010	010110 1001	010110 1001
D27.1	3B	001 11011	110110 1001	001001 1001
D28.1	3C	001 11100	001110 1001	001110 1001
D29.1	3D	001 11101	101110 1001	010001 1001
D30.1	3E	001 11110	011110 1001	100001 1001
D31.1	3F	001 11111	101011 1001	010100 1001
D0.2	40	010 00000	100111 0101	011000 0101
D1.2	41	010 00001	011101 0101	100010 0101
D2.2	42	010 00010	101101 0101	010010 0101
D3.2	43	010 00011	110001 0101	110001 0101
D4.2	44	010 00100	110101 0101	001010 0101
D5.2	45	010 00101	101001 0101	101001 0101
D6.2	46	010 00110	011001 0101	011001 0101
D7.2	47	010 00111	111000 0101	000111 0101
D8.2	48	010 01000	111001 0101	000110 0101
D9.2	49	010 01001	100101 0101	100101 0101
D10.2	4A	010 01010	010101 0101	010101 0101
D11.2	4B	010 01011	110100 0101	110100 0101
D12.2	4C	010 01100	001101 0101	001101 0101
D13.2	4D	010 01101	101100 0101	101100 0101
D14.2	4E	010 01110	011100 0101	011100 0101
D15.2	4F	010 01111	010111 0101	101000 0101
D16.2	50	010 10000	011011 0101	100100 0101
D17.2	51	010 10001	100011 0101	100011 0101
D18.2	52	010 10010	010011 0101	010011 0101
D19.2	53	010 10011	110010 0101	110010 0101
D20.2	54	010 10100	001011 0101	001011 0101
D21.2	55	010 10101	101010 0101	101010 0101
D22.2	56	010 10110	011010 0101	011010 0101
D23.2	57	010 10111	111010 0101	000101 0101
D24.2	58	010 11000	110011 0101	001100 0101
D25.2	59	010 11001	100110 0101	100110 0101
D26.2	5A	010 11010	010110 0101	010110 0101
D27.2	5B	010 11011	110110 0101	001001 0101
D28.2	5C	010 11100	001110 0101	001110 0101

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D29.2	5D	010 11101	101110 0101	010001 0101
D30.2	5E	010 11110	011110 0101	100001 0101
D31.2	5F	010 11111	101011 0101	010100 0101
D0.3	60	011 00000	100111 0011	011000 1100
D1.3	61	011 00001	011101 0011	100010 1100
D2.3	62	011 00010	101101 0011	010010 1100
D3.3	63	011 00011	110001 1100	110001 0011
D4.3	64	011 00100	110101 0011	001010 1100
D5.3	65	011 00101	101001 1100	101001 0011
D6.3	66	011 00110	011001 1100	011001 0011
D7.3	67	011 00111	111000 1100	000111 0011
D8.3	68	011 01000	111001 0011	000110 1100
D9.3	69	011 01001	100101 1100	100101 0011
D10.3	6A	011 01010	010101 1100	010101 0011
D11.3	6B	011 01011	110100 1100	110100 0011
D12.3	6C	011 01100	001101 1100	001101 0011
D13.3	6D	011 01101	101100 1100	101100 0011
D14.3	6E	011 01110	011100 1100	011100 0011
D15.3	6F	011 01111	010111 0011	101000 1100
D16.3	70	011 10000	011011 0011	100100 1100
D17.3	71	011 10001	100011 1100	100011 0011
D18.3	72	011 10010	010011 1100	010011 0011
D19.3	73	011 10011	110010 1100	110010 0011
D20.3	74	011 10100	001011 1100	001011 0011
D21.3	75	011 10101	101010 1100	101010 0011
D22.3	76	011 10110	011010 1100	011010 0011
D23.3	77	011 10111	111010 0011	000101 1100
D24.3	78	011 11000	110011 0011	001100 1100
D25.3	79	011 11001	100110 1100	100110 0011
D26.3	7A	011 11010	010110 1100	010110 0011
D27.3	7B	011 11011	110110 0011	001001 1100
D28.3	7C	011 11100	001110 1100	001110 0011
D29.3	7D	011 11101	101110 0011	010001 1100
D30.3	7E	011 11110	011110 0011	100001 1100
D31.3	7F	011 11111	101011 0011	010100 1100
D0.4	80	100 00000	100111 0010	011000 1101

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D1.4	81	100 00001	011101 0010	100010 1101
D2.4	82	100 00010	101101 0010	010010 1101
D3.4	83	100 00011	110001 1101	110001 0010
D4.4	84	100 00100	110101 0010	001010 1101
D5.4	85	100 00101	101001 1101	101001 0010
D6.4	86	100 00110	011001 1101	011001 0010
D7.4	87	100 00111	111000 1101	000111 0010
D8.4	88	100 01000	111001 0010	000110 1101
D9.4	89	100 01001	100101 1101	100101 0010
D10.4	8A	100 01010	010101 1101	010101 0010
D11.4	8B	100 01011	110100 1101	110100 0010
D12.4	8C	100 01100	001101 1101	001101 0010
D13.4	8D	100 01101	101100 1101	101100 0010
D14.4	8E	100 01110	011100 1101	011100 0010
D15.4	8F	100 01111	010111 0010	101000 1101
D16.4	90	100 10000	011011 0010	100100 1101
D17.4	91	100 10001	100011 1101	100011 0010
D18.4	92	100 10010	010011 1101	010011 0010
D19.4	93	100 10011	110010 1101	110010 0010
D20.4	94	100 10100	001011 1101	001011 0010
D21.4	95	100 10101	101010 1101	101010 0010
D22.4	96	100 10110	011010 1101	011010 0010
D23.4	97	100 10111	111010 0010	000101 1101
D24.4	98	100 11000	110011 0010	001100 1101
D25.4	99	100 11001	100110 1101	100110 0010
D26.4	9A	100 11010	010110 1101	010110 0010
D27.4	9B	100 11011	110110 0010	001001 1101
D28.4	9C	100 11100	001110 1101	001110 0010
D29.4	9D	100 11101	101110 0010	010001 1101
D30.4	9E	100 11110	011110 0010	100001 1101
D31.4	9F	100 11111	101011 0010	010100 1101
D0.5	A0	101 00000	100111 1010	011000 1010
D1.5	A1	101 00001	011101 1010	100010 1010
D2.5	A2	101 00010	101101 1010	010010 1010
D3.5	A3	101 00011	110001 1010	110001 1010
D4.5	A4	101 00100	110101 1010	001010 1010

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D5.5	A5	101 00101	101001 1010	101001 1010
D6.5	A6	101 00110	011001 1010	011001 1010
D7.5	A7	101 00111	111000 1010	000111 1010
D8.5	A8	101 01000	111001 1010	000110 1010
D9.5	A9	101 01001	100101 1010	100101 1010
D10.5	AA	101 01010	010101 1010	010101 1010
D11.5	AB	101 01011	110100 1010	110100 1010
D12.5	AC	101 01100	001101 1010	001101 1010
D13.5	AD	101 01101	101100 1010	101100 1010
D14.5	AE	101 01110	011100 1010	011100 1010
D15.5	AF	101 01111	010111 1010	101000 1010
D16.5	B0	101 10000	011011 1010	100100 1010
D17.5	B1	101 10001	100011 1010	100011 1010
D18.5	B2	101 10010	010011 1010	010011 1010
D19.5	B3	101 10011	110010 1010	110010 1010
D20.5	B4	101 10100	001011 1010	001011 1010
D21.5	B5	101 10101	101010 1010	101010 1010
D22.5	B6	101 10110	011010 1010	011010 1010
D23.5	B7	101 10111	111010 1010	000101 1010
D24.5	B8	101 11000	110011 1010	001100 1010
D25.5	B9	101 11001	100110 1010	100110 1010
D26.5	BA	101 11010	010110 1010	010110 1010
D27.5	BB	101 11011	110110 1010	001001 1010
D28.5	BC	101 11100	001110 1010	001110 1010
D29.5	BD	101 11101	101110 1010	010001 1010
D30.5	BE	101 11110	011110 1010	100001 1010
D31.5	BF	101 11111	101011 1010	010100 1010
D0.6	C0	110 00000	100111 0110	011000 0110
D1.6	C1	110 00001	011101 0110	100010 0110
D2.6	C2	110 00010	101101 0110	010010 0110
D3.6	C3	110 00011	110001 0110	110001 0110
D4.6	C4	110 00100	110101 0110	001010 0110
D5.6	C5	110 00101	101001 0110	101001 0110
D6.6	C6	110 00110	011001 0110	011001 0110
D7.6	C7	110 00111	111000 0110	000111 0110
D8.6	C8	110 01000	111001 0110	000110 0110

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D9.6	C9	110 01001	100101 0110	100101 0110
D10.6	CA	110 01010	010101 0110	010101 0110
D11.6	CB	110 01011	110100 0110	110100 0110
D12.6	CC	110 01100	001101 0110	001101 0110
D13.6	CD	110 01101	101100 0110	101100 0110
D14.6	CE	110 01110	011100 0110	011100 0110
D15.6	CF	110 01111	010111 0110	101000 0110
D16.6	D0	110 10000	011011 0110	100100 0110
D17.6	D1	110 10001	100011 0110	100011 0110
D18.6	D2	110 10010	010011 0110	010011 0110
D19.6	D3	110 10011	110010 0110	110010 0110
D20.6	D4	110 10100	001011 0110	001011 0110
D21.6	D5	110 10101	101010 0110	101010 0110
D22.6	D6	110 10110	011010 0110	011010 0110
D23.6	D7	110 10111	111010 0110	000101 0110
D24.6	D8	110 11000	110011 0110	001100 0110
D25.6	D9	110 11001	100110 0110	100110 0110
D26.6	DA	110 11010	010110 0110	010110 0110
D27.6	DB	110 11011	110110 0110	001001 0110
D28.6	DC	110 11100	001110 0110	001110 0110
D29.6	DD	110 11101	101110 0110	010001 0110
D30.6	DE	110 11110	011110 0110	100001 0110
D31.6	DF	110 11111	101011 0110	010100 0110
D0.7	E0	111 00000	100111 0001	011000 1110
D1.7	E1	111 00001	011101 0001	100010 1110
D2.7	E2	111 00010	101101 0001	010010 1110
D3.7	E3	111 00011	110001 1110	110001 0001
D4.7	E4	111 00100	110101 0001	001010 1110
D5.7	E5	111 00101	101001 1110	101001 0001
D6.7	E6	111 00110	011001 1110	011001 0001
D7.7	E7	111 00111	111000 1110	000111 0001
D8.7	E8	111 01000	111001 0001	000110 1110
D9.7	E9	111 01001	100101 1110	100101 0001
D10.7	EA	111 01010	010101 1110	010101 0001
D11.7	EB	111 01011	110100 1110	110100 1000
D12.7	EC	111 01100	001101 1110	001101 0001

Table 4-1. Data Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +
			abcdei fghj	abcdei fghj
D13.7	ED	111 01101	101100 1110	101100 1000
D14.7	EE	111 01110	011100 1110	011100 1000
D15.7	EF	111 01111	010111 0001	101000 1110
D16.7	F0	111 10000	011011 0001	100100 1110
D17.7	F1	111 10001	100011 0111	100011 0001
D18.7	F2	111 10010	010011 0111	010011 0001
D19.7	F3	111 10011	110010 1110	110010 0001
D20.7	F4	111 10100	001011 0111	001011 0001
D21.7	F5	111 10101	101010 1110	101010 0001
D22.7	F6	111 10110	011010 1110	011010 0001
D23.7	F7	111 10111	111010 0001	000101 1110
D24.7	F8	111 11000	110011 0001	001100 1110
D25.7	F9	111 11001	100110 1110	100110 0001
D26.7	FA	111 11010	010110 1110	010110 0001
D27.7	FB	111 11011	110110 0001	001001 1110
D28.7	FC	111 11100	001110 1110	001110 0001
D29.7	FD	111 11101	101110 0001	010001 1110
D30.7	FE	111 11110	011110 0001	100001 1110
D31.7	FF	111 11111	101011 0001	010100 1110

Table 4-2. Special Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +	Notes
			abcdei fghj	abcdei fghj	
K28.0	1C	000 11100	001111 0100	110000 1011	
K28.1	3C	001 11100	001111 1001	110000 0110	1,2
K28.2	5C	010 11100	001111 0101	110000 1010	1
K28.3	7C	011 11100	001111 0011	110000 1100	
K28.4	9C	100 11100	001111 0010	110000 1101	1
K28.5	BC	101 11100	001111 1010	110000 0101	2
K28.6	DC	110 11100	001111 0110	110000 1001	1
K28.7	FC	111 11100	001111 1000	110000 0111	1,2
K23.7	F7	111 10111	111010 1000	000101 0111	1
K27.7	FB	111 11011	110110 1000	001001 0111	

Table 4-2. Special Character Encodings

Character Name	Character Value (hex)	Character Bits HGF EDCBA	Current RD –	Current RD +	Notes
			abcdei fghj	abcdei fghj	
K29.7	FD	111 11101	101110 1000	010001 0111	
K30.7	FE	111 11110	011110 1000	100001 0111	1

1 - Reserved code-groups.

2 - The code-groups /K28.5/, /K28.7/, and /K28.1/ contain a comma.

4.5.7 Special Characters and Columns

Table 4-3 defines the special characters and columns of special characters used by 1x and 4x LP-Serial links. Special characters are used for the following functions:

1. Alignment to code-group (10-bit) boundaries on lane-by-lane basis.
2. Alignment of the receive data stream across four lanes.
3. Clock rate compensation between receiver and transmitter.
4. Control symbol delimiting.

Table 4-3. Special Characters and Columns

Code-Group/Column Designation	Code-Group/Column Use	Number of Code-groups	Encoding
/PD/	Packet_Delimiter Control Symbol	1	/K28.3/
/SC/	Start_of_Control_Symbol	1	/K28.0/
/**I**/	Idle		
/K/	1x Sync	1	/K28.5/
/R/	1x Skip	1	/K29.7/
/A/	1x Align	1	/K27.7/
||**I**||	Idle column		
||K||	4x Sync column	4	/K28.5/K28.5/K28.5/K28.5/
||R||	4x Skip column	4	/K29.7/K29.7/K29.7/K29.7/
||A||	4x Align column	4	/K27.7/K27.7/K27.7/K27.7/

4.5.7.1 Packet Delimiter Control Symbol (/PD/)

PD and /PD/ are aliases for respectively the K28.3 character and the /K28.3/ code-group which are used to delimit the beginning of a control symbol that contains a packet delimiter.

4.5.7.2 Start of Control Symbol (/SC/)

SC and /SC/ are aliases for respectively the K28.0 character and the /K28.0/ code-group which are used to delimit the beginning of a control symbol that does not contain a packet delimiter.

4.5.7.3 Idle (/I/)

I and /I/ are aliases for respectively any of the idle sequence characters (A, K, or R) and idle sequence code-groups (/A/, /K/, or /R/).

4.5.7.4 Sync (/K/)

K and /K/ are aliases for respectively the K28.5 character and the /K28.5/ code-group which is used in the idle sequence to provide the receiver with the information it requires to achieve and maintain bit and 10-bit code-group boundary synchronization. The /K28.5/ code-group was selected as the Sync character for the following reasons:

1. It contains the comma pattern in bits **abcdeif** which can be easily found in the code-group bit stream and marks the code-group boundary.
2. The bits **ghj** provide the maximum number of transitions (i.e. 101 or 010).

A comma is a 7-bit string defined as either b'0011111' (comma+) or b'1100000' (comma-). Within the code-group set it is a singular bit pattern, which, in the absence of transmission errors, cannot appear in any other location of a code-group and cannot be generated across the boundaries of any two adjacent code-groups with the following exception:

> The /K28.7/ special code-group when followed by any of the data code-groups /D3.y/, /D11.y/, /D12.y/, /D19.y/, /D20.y/, /D28.y/, or /K28.y/, where y is an integer in the range 0 through 7, may (depending on the value of running disparity) cause a comma to be generated across the boundary of the two code-groups. A comma that is generated across the boundary between two adjacent code-groups may cause the receiver to change the 10-bit code-group alignment. As a result, the /K28.7/ special code-group may be used for test and diagnostic purposes only.

4.5.7.5 Skip (/R/)

R and /R/ are aliases for respectively the K29.7 character and the /29.7/ code-group which are used in the idle sequence and are also used in the clock compensation sequence.

4.5.7.6 Align (/A/)

A and /A/ are aliases for respectively the K27.7 character and the /27.7/ code-group which are used in the idle sequence and are used for lane alignment on 4x links.

4.5.8 Effect of Single Bit Code-Group Errors

Single bit code-group errors will be the dominant code-group error by many orders of magnitude. It is therefore useful to know the variety of code-group corruptions that can be caused by a single bit error.

Table 4-4 lists all possible code-group corruptions that can be caused by a single-bit error. The notation /X/ => /Y/ means that the code-group for the character X has been corrupted by a single-bit error into the code-group for the character Y. If the corruption results in a code-group that is invalid for the current receiver running disparity, the notation /X/ => /INVALID/ is used. The table provides the information required to deterministically detect all isolated single bit transmission errors.

Table 4-4. Code-Group Corruption Caused by Single Bit Errors

Corruption	Detection
/SC/ => /INVALID/	Detectable as an error when decoding the code-group. When this error occurs within a packet, it is indistinguishable from a /Dx.y/ => /INVALID/. When this error occurs outside of a packet, the type of error can be inferred from whether the /INVALID/ is followed by the three /Dx.y/ that comprise the control symbol data.
/PD/ => /INVALID/	Detectable as an error when decoding the code-group. When this error occurs within a packet, it is indistinguishable from a /Dx.y/ => /INVALID/. When this error occurs outside of a packet, the type of error can be inferred from whether the /INVALID/ is followed by the three /Dx.y/ that comprise the control symbol data.
/A/, /K/ or /R/ => /Dx.y/	Detectable as an error as /Dx.y/ is illegal outside of a packet or control symbol and /A/, /K/ and /R/ are illegal within a packet or control symbol.
/A/, /K/ or /R/ => /INVALID/	Detectable as an error when decoding the code-group.
/Dx.y/ => /A/, /K/ or /R/	Detectable as an error as /A/, /K/ and /R/ are illegal within a packet or control symbol and /Dx.y/ is illegal outside of a packet or control symbol.
/Dx.y/ => /INVALID/	Detectable as an error when decoding the code-group.
/Dx.y/ => /Du.v/	Detectable as an error by the packet or control symbol CRC. The error will also result in a subsequent unerrored code-group being decoded as INVALID, but that resulting INVALID code-group may occur an arbitrary number of code-groups after the errored code-group.

4.5.9 Idle Sequence

The idle sequence is a sequence of code-groups that shall be transmitted continuously over each lane of an LP-Serial link whenever packets or control symbols are not being transmitted. An idle sequence may not be inserted in a packet or delimited control symbol. The idle sequence is transmitted over each lane as part of the port initialization process as required in Section 4.7.3.5 and Section 4.7.3.6. An idle sequence containing a special clock compensation sequence shall be transmitted at least once every 5000 code-groups even when there are packets or

control symbols available to transmit to allow clock rate compensation.

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

1. The first code-group (column) of an idle sequence generated by a port operating in 1x mode (4x mode) shall be a /K/ (||K||). The first code-group (column) shall be transmitted immediately following the last code-group (column) of a packet or delimited control symbol.
2. At least once every 5000 code-groups (columns) transmitted by a port operating in 1x mode (4x mode), an idle sequence containing the /K/R/R/R/ code-group sequence (||K||R||R||R|| column sequence) shall be transmitted by the port. This sequence is referred to as the “compensation sequence”.
3. When not transmitting the compensation sequence, all code-groups (columns) following the first code-group (column) of an idle sequence generated by a port operating in 1x mode (4x mode) shall be a pseudo-randomly selected sequence of /A/, /K/, and /R/ (||A||, ||K||, and ||R||) based on a pseudo-random sequence generator of 7th order or greater and subject to the minimum and maximum /A/ (||A||) spacing requirements.
4. The number of non /A/ code-groups (non ||A|| columns) between /A/ code-groups (||A|| columns) in the idle sequence of a port operating in 1x mode (4x mode) shall be no less than 16 and no more than 32. The number shall be pseudo-randomly selected, uniformly distributed across the range and based on a pseudo-random sequence generator of 7th order or greater.

There are no requirements on the length of an idle sequence. An idle sequence may be of any length.

The idle sequence is transmitted on each lane of a link when neither packets nor delimited control symbols are being transmitted to allow each lane receiver to maintain bit and 10-bit code-group boundary synchronization.

The pseudo-random selection of code-groups in the idle sequence results in an idle sequence whose spectrum has no discrete lines which minimizes the EMI of long idle sequences.

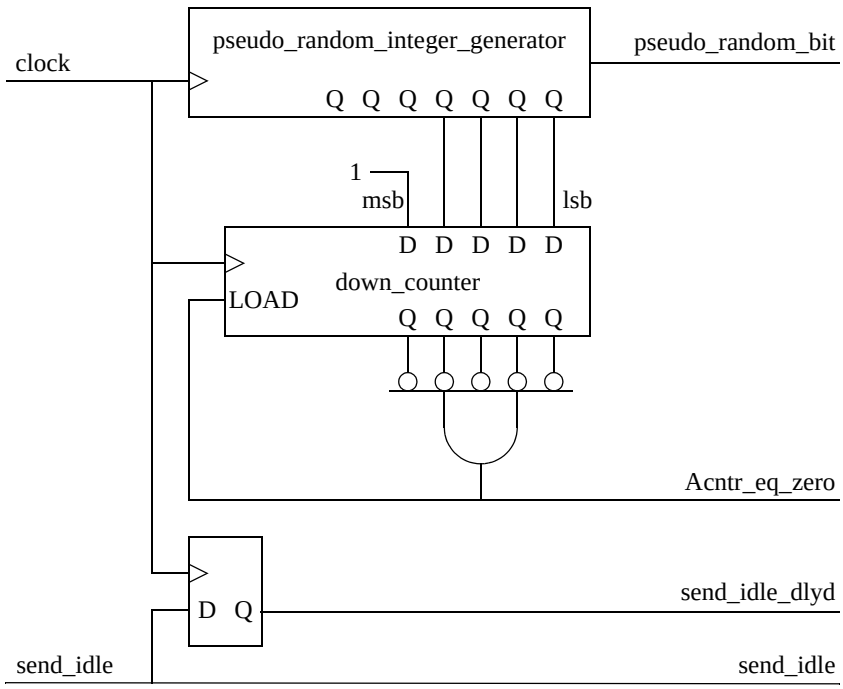
The compensation sequence allows retiming repeaters (discussed in Section 4.6) to compensate for up to a +/- 200 ppm difference between input bit rate and output bit rate, each of which have a +/-100 ppm tolerance. It may also be used to allow the input side of a port to compensate for up to a +/-200 ppm difference between the input bit rate and the bit rate of the device core which may be running off a different clock. This is done by dropping or adding an /R/ or ||R|| as needed to avoid overrun/underrun. Since a packet or delimited control symbol may not be interrupted by an idle sequence, designers must be careful to guarantee that no more

than 5000 code-groups are transmitted between compensation sequences.

4.5.9.1 Idle Sequence Generation

A primitive 7th order polynomial is recommended as the generating polynomial for the pseudo-random sequence that is used in the generation of the idle sequence. The polynomials $x^7 + x^6 + 1$ and $x^7 + x^3 + 1$ are examples of primitive 7th order polynomials which could be used as generator polynomials. The pseudo-random sequence generator is clocked (generates a new pseudo-random sequence value) once per idle sequence code-group (column). Four of the pseudo-random sequence generator state bits may be selected as the pseudo-random value for /A/ (||A||) spacing and any other state bit or logical function of state bits may be selected as the /K/ vs. /R/ selector.

Figure 4-4 shows an example circuit illustrating how this may be done. The clock ticks whenever a code-group or column is transmitted. Send_idle is asserted whenever an idle sequence begins. The equations indicate the states in which to transmit the indicated idle code-group, except when the compensation sequence is being transmitted. Any equivalent method is acceptable.



send_K = send_idle & (!send_idle_dlyd | send_idle_dlyd & !Acntr_eq_zero & pseudo_random_bit)
send_A = send_idle & send_idle_dlyd & Acntr_eq_zero
send_R = send_idle & send_idle_dlyd & !Acntr_eq_zero & !pseudo_random_bit

Figure 4-4. Example of a Pseudo-Random Idle Code-Group Generator

4.5.10 1x Link Transmission Rules

A 1x LP-Serial link has a single lane (differential pair) in each direction. A 1x LP-Serial port shall be encoded and transmit the character stream of delimited control symbols and packets received from the upper layers over the differential pair in the order the characters were received from the upper layers. When neither control symbols nor packets are available from the upper layers for transmission, the 1x idle sequence shall be fed to the input of the encoder for encoding and transmission. On reception, the code-group stream is decoded and passed to the upper layers.

When a 4x port is operating in 1x mode, the character stream from the upper layers is not striped across the lanes before encoding as is done when operating in 4x mode. The entire character stream from the upper layers is fed in parallel to both lanes 0 and 2. A 4x LP-Serial port operating in 1x mode shall encoded and transmit the character stream of delimited control symbols and packets received from the upper layers over both lane 0 and lane 2, with the following exception. A 4x port operating in 1x mode may elect to disable the output driver of the lane which was not selected by the initialization state machine. It is recommended that the mechanism for disabling the output driver be under software control.

When neither delimited control symbols nor packets are available from the upper layers for transmission, the 1x idle sequence shall be fed in parallel to the input of the lane 0 and lane 2 encoders for encoding and transmission on lanes 0 and 2. On reception, the code-group stream from either lane 0 or 2 is selected according to the state of the 1x/4x_Initialization state machine (Section 4.7.3.6, “1x/4x Mode Initialization State Machine), decoded and passed to the upper layers.

Figure 4-5 shows the encoding and transmission order for a control symbol transmitted over a 1x LP-Serial link.

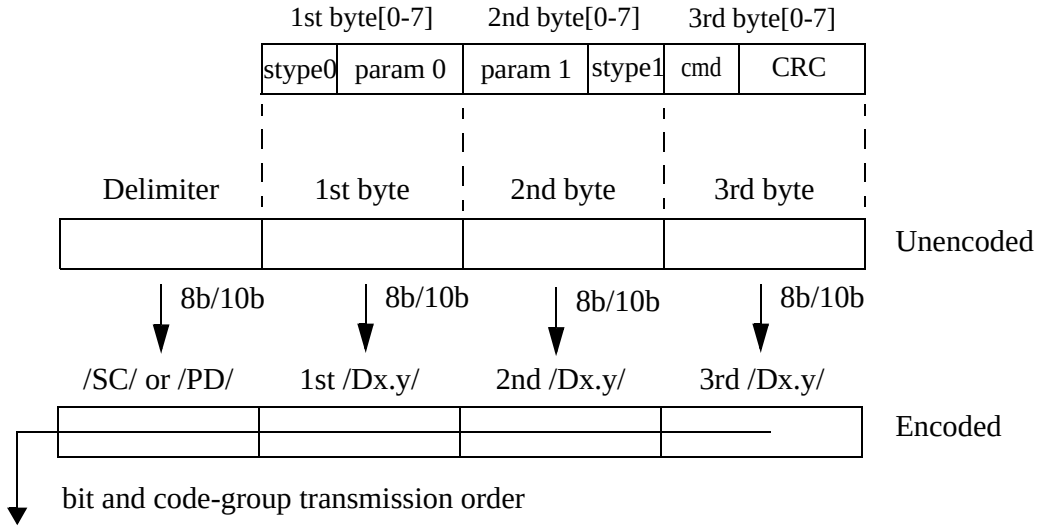


Figure 4-5. 1x Mode Control Symbol Encoding and Transmission Order

Figure 4-6 shows the encoding and transmission order for a packet transmitted over a 1x LP-Serial link.

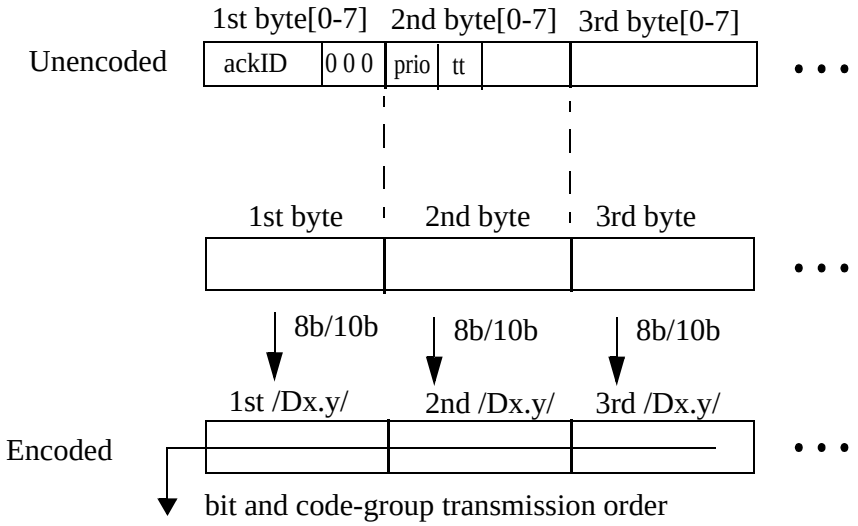


Figure 4-6. 1x Mode Packet Encoding and Transmission Order

Figure 4-7 shows an example of control symbol, packet, and idle sequence transmission on a 1x LP-Serial link.

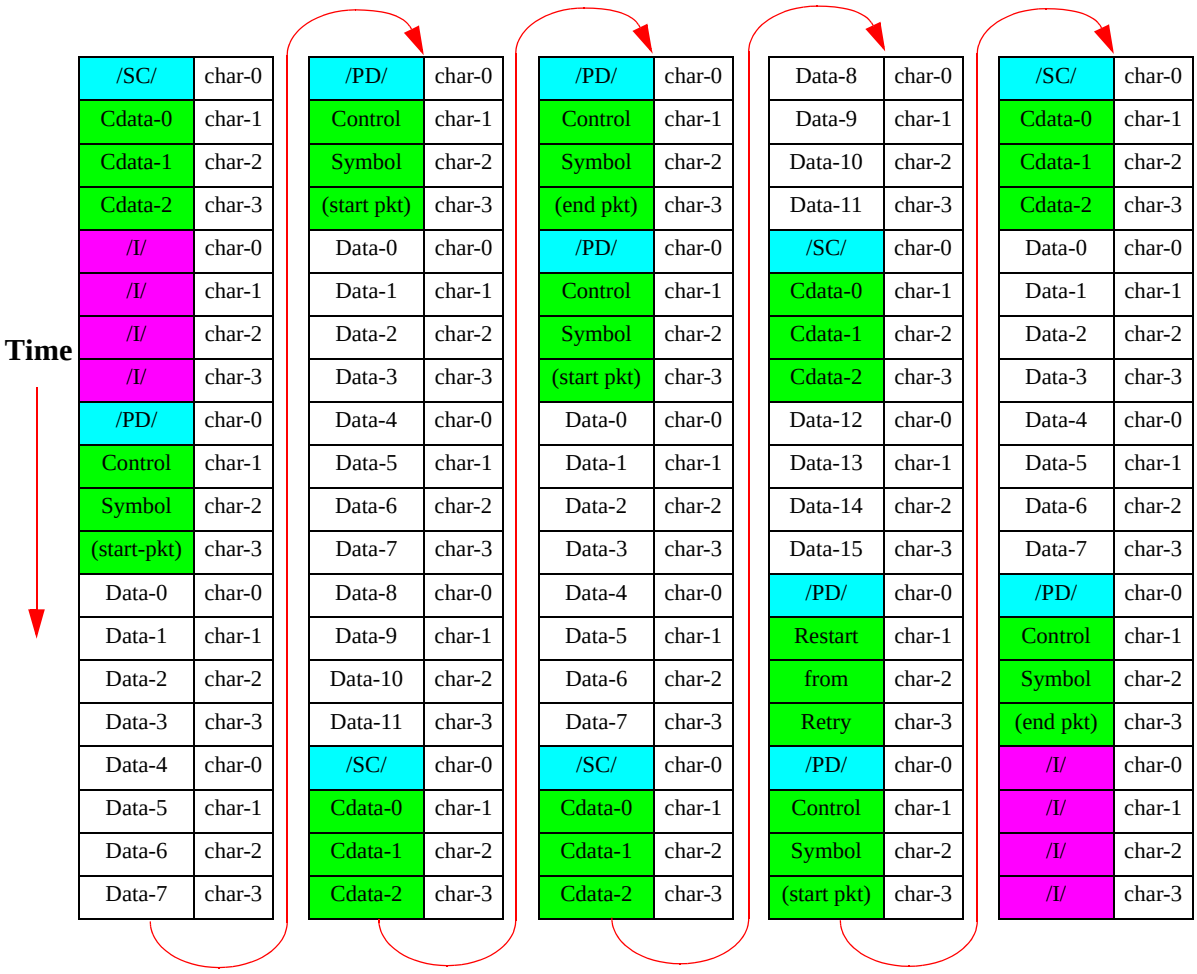


Figure 4-7. 1x Typical Data Flow

4.5.11 4x Link Striping and Transmission Rules

A LP-Serial port operating in 4x mode shall stripe the character stream of delimited control symbols and packets that it receives from the upper layers across the four lanes before 8B/10B encoding as follows:

Packets and delimited control symbols shall be striped across the four lanes beginning with lane 0. The first character of each packet, or delimited control symbol, is placed in lane 0, the second character is placed in lane1, the third character is placed in lane 2, and the fourth character is placed in lane 3. The fifth character and subsequent characters wrap around and continue beginning in lane 0.

As a result of their defined lengths, delimited control symbols will form a column after striping and a packet will form an integer number of contiguous columns.

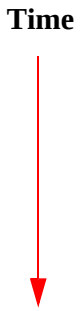
After striping, each of the 4 streams of characters shall be independently 8B/10B encoded and transmitted.

When neither delimited control symbols nor packets are available from the upper layers for transmission, the 4x idle sequence shall be transmitted. This can be achieved by feeding the 1x idle sequence in parallel to the inputs of the encoders for all four lanes for encoding and transmission on the four lanes. Feeding the 1x idle sequence in parallel to the input of all four lane encoders converts each 1x idle sequence character in to a 4x idle sequence column. The 1x sequence is not striped across the four lanes.

On reception, each lane shall be 8B/10B decoded. After decoding, the four lanes shall be aligned. The ||A|| columns transmitted as part of the 4x idle sequence provide the information needed to perform alignment. After alignment, the columns are destriped into a single character stream and passed to the upper layers.

The lane alignment process eliminates the skew between lanes so that after destriping, the ordering of characters in the received character stream is the same as the ordering of characters before striping and transmission. Since the minimum number of non ||A|| columns between ||A|| columns is 16, the maximum lane skew that can be unambiguously corrected is the time it takes for to transmit 7 code-groups on a lane.

Figure 4-8 shows an example of idle sequence, packet, and delimited control symbol transmission on a 4x link.



Lane-0	Lane-1	**Lane-2**	**Lane-3**
/SC/	Cdata-0	Cdata-1	Cdata-2
/I/	/I/	/I/	/I/
/PD/	Control Symbol (Start-of-packet)		
Data-0	Data-1	Data-2	Data-3
Data-4	Data-5	Data-6	Data-7
/PD/	Control Symbol (Start-of-packet)		
Data-0	Data-1	Data-2	Data-3
Data-4	Data-5	Data-6	Data-7
Data-8	Data-9	Data-10	Data-11
/SC/	Cdata-0	Cdata-1	Cdata-2
/PD/	Control Symbol (End-of-packet)		
/PD/	Control Symbol (Start-of-packet)		
Data-0	Data-1	Data-2	Data-3
Data-4	Data-5	Data-6	Data-7
/SC/	Cdata-0	Cdata-1	Cdata-2
Data-8	Data-9	Data-10	Data-11
/SC/	Cdata-0	Cdata-1	Cdata-2
Data-12	Data-13	Data-14	Data-15
/PD/	Control Symbol (Restart-from-retry)		
/PD/	Control Symbol (Start-of-packet)		
/SC/	Cdata-0	Cdata-1	Cdata-2
Data-0	Data-1	Data-2	Data-3
Data-4	Data-5	Data-6	Data-7
/PD/	Control Symbol (End-of-packet)		
/I/	/I/	/I/	/I/

Figure 4-8. Typical 4x Data Flow

4.6 Retimers and Repeaters

The LP-Serial Specification allows “retimers” and “repeaters”. Retimers amplify a weakened signal, but do not transfer jitter to the next segment. Repeaters also amplify a weakened signal, but transfer jitter to the next segment. Retimers allow greater distances between end points at the cost of additional latency. Repeaters support less distance between end points than retimers and only add a small amount of latency.

4.6.1 Retimers

A retimer shall comply with all AC specifications found in Chapter 8, “Electrical Specifications”. This includes resetting the jitter budget thus extending the transmission distance for the link. The retimer repeats the received code-groups after performing code-group synchronization and serializes the bitstream again on transmission, based on a local clock reference. Up to two retimers are allowed between two end nodes.

A retimer is not RapidIO protocol-aware or addressable in any way. The only awareness a retimer has is to the synchronization on the /K/ code-group and the function of /R/ insertion and removal. A retimer may insert up to one /R/ code-group immediately following a /K/ code-group sequence, or remove one /R/ code-group that immediately follows a /K/ code-group sequence. Note that the /R/ code-group is disparity neutral and therefore its insertion or deletion does not affect the running disparity.

A 4-lane retimer must perform lane synchronization and deskew, in exactly the same way a RapidIO device implementing this physical layer does when synchronizing inputs during initialization and startup. A 4-lane retimer will synchronize and align all lanes that are driven to it. Therefore, such a retimer allows for the degradation of an input 4x link to a 1x link on either lane 0 or 2. If any link drops out, the retimer must merely continue to pass the active links, monitoring for the compensation sequence and otherwise passing through whatever code-groups appear on its inputs. A retimer may optionally not drive any outputs whose corresponding inputs are not active.

Any insertion or removal of a /R/ code-groups in a 4-lane retimer must be done on a full column. A retimer may retime links operating at the same width only (i.e. cannot connect a link operating at 1x to a link operating at 4x). A retimer may connect a 1x link to a 4x link that is operating at 1x. Retimers perform clock tolerance compensation between the receive and transmit clock. The transmit clock is usually derived from a local reference.

Retimers do not check for code violations. Code-groups received on one port are transmitted on the other regardless of code violations or running disparity errors.

4.6.2 Repeaters

A repeater is used to amplify the signal, but does not retime the signal, and therefore can add additional jitter to the signal. It does not compensate for clock rate variation. The repeater repeats the received code-groups as the bits are received by sampling the incoming bits with a clock derived from the bit stream, and then retransmitting them based on that clock. Repeaters may be used with 4x links but lane-to-lane skew may be amplified. Repeaters do not interpret or alter the bit stream in any way.

4.7 Port Initialization

This section specifies the port initialization process. The process includes detecting the presence of a partner at the other end of the link (a link partner), establishing bit synchronization and code-group boundary alignment and if the port is capable of supporting both 1x and 4x modes (a 1x/4x port), discovering whether the link partner is capable of 4x mode (4-lane) operation, selecting 1x or 4x mode operation and if 1x mode is selected, selecting lane 0 or lane 2 for link reception.

The initialization process is controlled by several state machines. The number and type of state machines depends on whether the port supports only 1x mode (a 1x port) or supports both 1x and 4x modes (a 1x/4x port). In either case, there is a primary state machine and one or more secondary state machines. The use of multiple state machines results in a simpler overall design. As might be expected, the initialization process for a 1x port is simpler than and a subset of the initialization process for a 1x/4x port.

The initialization process for 1x and 1x/4x ports is both described in text and specified with state machine diagrams. **In the case of conflict between the text and a state machine diagram, the state machine diagram takes precedence.**

4.7.1 1x Mode Initialization

The initialization process for ports that support only 1x mode shall be controlled by two state machines, 1x_Initialization and Lane_Synchronization. 1x_Initialization is the primary state machine and Lane_Synchronization is the secondary state machine. The operation of these state machines is described and specified in Section 4.7.3.5 and Section 4.7.3.3 respectively.

After a 1x port has been initialized, the port is required to receive seven error-free status control symbols without intervening detected errors to verify link operation before transmitting any packets.

4.7.2 1x/4x Mode Initialization

The initialization process for ports that support both 1x and 4x modes shall be controlled by six state machines, 1x/4x_Initialization, Lane_Synchronization[0-3] (one for each of the four lanes) and Lane_Alignment. 1x/4x_Initialization is the primary state machine. Lane_Synchronization[0-3] and Lane_Alignment are the secondary state machines. The operation of these state machines is described and specified in Section 4.7.3.6, Section 4.7.3.3 and Section 4.7.3.4 respectively.

After a 4x port has been initialized, the port is required to receive seven error-free status control symbols without intervening detected errors to verify link operation before transmitting any packets.

4.7.3 State Machines

4.7.3.1 State Machine Conventions

A state machine state is persistent until an exit condition occurs.

A state machine variable that is listed in the body of a state but is not part of an assignment statement is asserted for the duration of that state only.

A state machine variable that is assigned a value in the body of a state retains that value until assigned a new value in another state.

A state machine function that is listed in the body of a state is executed once during the state.

4.7.3.2 State Machine Variables and Functions

A state machine variable is asserted when its value is 1 and deasserted when it value is 0.

The functions used in the state machines are defined as follows.

change( )

Asserted when the variable on which it operates changes state.

next_code_group( )

Gets the next 10 bit code-group for the lane when it becomes available.

next_column( )

Gets the next column of 4 code-groups when it becomes available.

The variables used in the state machines are defined as follows.

4x-mode

Asserted when the port is operating in 4x mode

||A||

Asserted when the current column contains all /A/s.

Acounter

A counter used in the Lane Alignment state machine to count received alignment columns (||A||s).

align_error

Asserted when the current column contains at least one /A/, but not all /A/s.

all_lane_sync

Asserted when all four lanes of a 4x mode receiver are in bit synchronization and

code-group boundary alignment. (all_lane_sync = lane_sync[0] & lane_sync[1] & lane_sync[2] & lane_sync[3])

discovery_timer_done

Asserted when discovery_timer_en has been continuously asserted for 12 +/- 4 msec and the state machine is in the DISCOVERY state. The assertion of discovery_timer_done causes discovery_timer_en to be deasserted. When the state machine is not in the DISCOVERY state, discovery_timer_done is deasserted.

discovery_timer_en

When asserted, the discovery_timer runs. When deasserted, the discovery_timer is reset to and maintains its initial value.

force_1x_mode

When asserted, forces the 1x/4x Initialization state machine to use 1x mode.

force_lane2

When asserted in 1x mode, forces the 1x/4x Initialization state machine to use lane 2 for reception.

force_reinit

When asserted, forces a 1x or 1x/4x Initialization state machine to reinitialize. The signal is set under software control and is cleared by the Initialization state machine.

Icounter

Counter used in the Lane_Synchronization state machine to count INVALID received code-groups. There is one Icounter for each lane in a 4x mode receiver.

/K28.5/

Asserted when the current code-group is /K28.5/

Kcounter

Counter used in the Lane_Synchronization state machine to count received /K.28.5/ code-groups. There is one Kcounter for each lane in a 4x mode receiver.

lane02_drvr_oe

The output enable for the lane 0 and the lane 2 output drivers of a 1x/4x mode port.

lane13_drvr_oe

The output enable for the lane 1 and the lane 3output drivers of a 1x/4x mode port.

lanes_aligned

Asserted by the Lane_Alignment state machine when it determines that all four

lanes are in sync and aligned.

lane_sync

Asserted by the Lane_Synchronization state machine when it determines that the lane it is monitoring is in bit synchronization and code-group boundary alignment. The is a lane_sync signal for each lane in a 4x mode receiver.

link_drvr_oe

When asserted, the output link driver of a 1x port is enabled.

Mcounter

Mcounter is used in the Lane_Alignment state machine to count columns received that contain at least one /A/, but not all /A/s.

Vcounter

Vcounter is used in the Lane_Synchronization state machine to count VALID received code-groups. There is one Vcounter for each lane in a 4x mode receiver.

port_initialized

When asserted, port_initialized indicates that the link is initialized and is available for transmitting control symbols and packets. When deasserted, the link is not initialized and is not available for transmitting control symbols and packets.

receive_lane2

In a 1x/4x capable port that is initialized and is operating in 1x mode (4x_mode deasserted), receive_lane2 indicates which lane the port has selected for input. When asserted, the port input is from lane 2. When deasserted the port input is from lane 0. When the port is operating in 4x mode (4x_mode asserted), receive_lane2 is undefined and shall be ignored.

signal_detect

Asserted when a lane receiver is enabled and a signal meeting an implementation defined criteria is present at the input of the receiver. The use of signal_detect is implementation dependent. It may be continuously asserted or it may be used to require that some implementation defined additional condition be met before the Lane_Synchronization state machine is allowed to exit the NO_SYNC state. Signal_detect might for example be used to ensure that the input signal to a lane receiver meet some minimum AC input power requirement so that the receiver can not lock up on crosstalk from an output of the same port.

silence_timer_done

Asserted when silence_timer_en has been continuously asserted for 120 +/- 40 µs and the state machine is in the SILENT state. The assertion of silence_timer_done causes silence_timer_en to be deasserted. When the state machine is not in the SILENT state, silence_timer_done is deasserted.

silence_timer_en

When asserted, the silence_timer runs. When deasserted, the silence_timer is reset to and maintains its initial value.

/VALID/

When asserted, /VALID/ indicates that the current code-group is a valid code-group given the current running disparity.

/INVALID/

When asserted, /INVALID/ indicates that the current code-group is an invalid code-group given the current running disparity.

4.7.3.3 Lane Synchronization State Machine

The Lane_Synchronization state machine monitors the bit synchronization and code-group boundary alignment for a lane receiver. A port that supports only 1x mode has one Lane_Synchronization state machine. A port that supports both 1x and 4x modes has four Lane_Synchronization state machines, one for each lane (Lane_Synchronization[0] through Lane_Synchronization[3]).

The state machine determines the bit synchronization and code-group boundary alignment state of a lane receiver by monitoring the received code-groups and looking for /K28.5/s, other valid code-groups and invalid code-groups. The code-group /K28.5/ contains the “comma” bit sequence that is used to establish code-group boundary alignment. When a lane is error free, the “comma” pattern occurs only in the /K28.5/ code-group. Several counters are used to provide hysteresis so that occasional bit errors do not cause spurious lane_sync state changes.

The state machine does not specify how bit synchronization and code-group boundary alignment is to be achieved. The methods used by a lane receiver to achieve bit synchronization and code-group boundary alignment are implementation dependent. However, isolated single bit errors shall not cause the code-group boundary alignment mechanism to change alignment. For example, a isolated single bit error that results in a “comma” pattern across a code-group boundary may not cause the code-group boundary alignment mechanism to change alignment.

The state machine starts in the NO_SYNC state and sets the variables Kcounter[0] and lane_sync[n] to 0 (lane n is out of code-group boundary sync). It then looks for a /K28.5/ code-group. When it finds one and the signal signal_detect[n] is asserted, the machine moves to the NO_SYNC_1 state.

The NO_SYNC_1 state in combination with the NO_SYNC_2 state looks for 127 /K28.5/ code-groups without any intervening /INVALID/ code-groups. When this condition is achieved, machine goes to state SYNC. If an intervening /INVALID/ code-group is detected, the machine goes back to the NO_SYNC state. The number

127 was selected because it is large enough that it would be highly unlikely that SYNC would be falsely achieved and also because it fits in a 7-bit counter. Since the /K28.5/ code-group comprises slightly less than half of the code-groups in the idle sequence, something more than 256 code-groups must be received after the first /K28.5/ to achieve the 128 /K28.5/ code-group criteria to transition to the SYNC state.

In the SYNC state, the machine sets the variable lane_sync[n] to 1 (lane n is in code-group boundary sync), set the variable Icounter[n] to 0 and begins looking for /INVALID/ code-groups. If an /INVALID/ code-group is detected, the machine goes to state SYNC_1.

The SYNC_1 state in combination with the SYNC_2, SYNC_3, and SYNC_4 states looks for 255 consecutive /VALID/ code-groups without any /INVALID/ code-groups. When 255 /VALID/ symbols are received, the Icounter[n] value is decremented in the transition through the SYNC_4 state. If it does not, it increments Icounter[n]. If Icounter[n] is decremented back to 0, the machine returns to the SYNC state. If Icounter[n] is incremented to 3, the machine goes to the NO_SYNC state and starts over. This algorithm tolerates isolated single bit errors in that an isolated single bit error will not cause the machine to change the variable lane_sync[n] from 1 to 0 (in sync to out of sync). Three errors (sometimes two errors when one of them causes a disparity error on the following code group) within 256 code-groups will result in an out-of-sync indication.

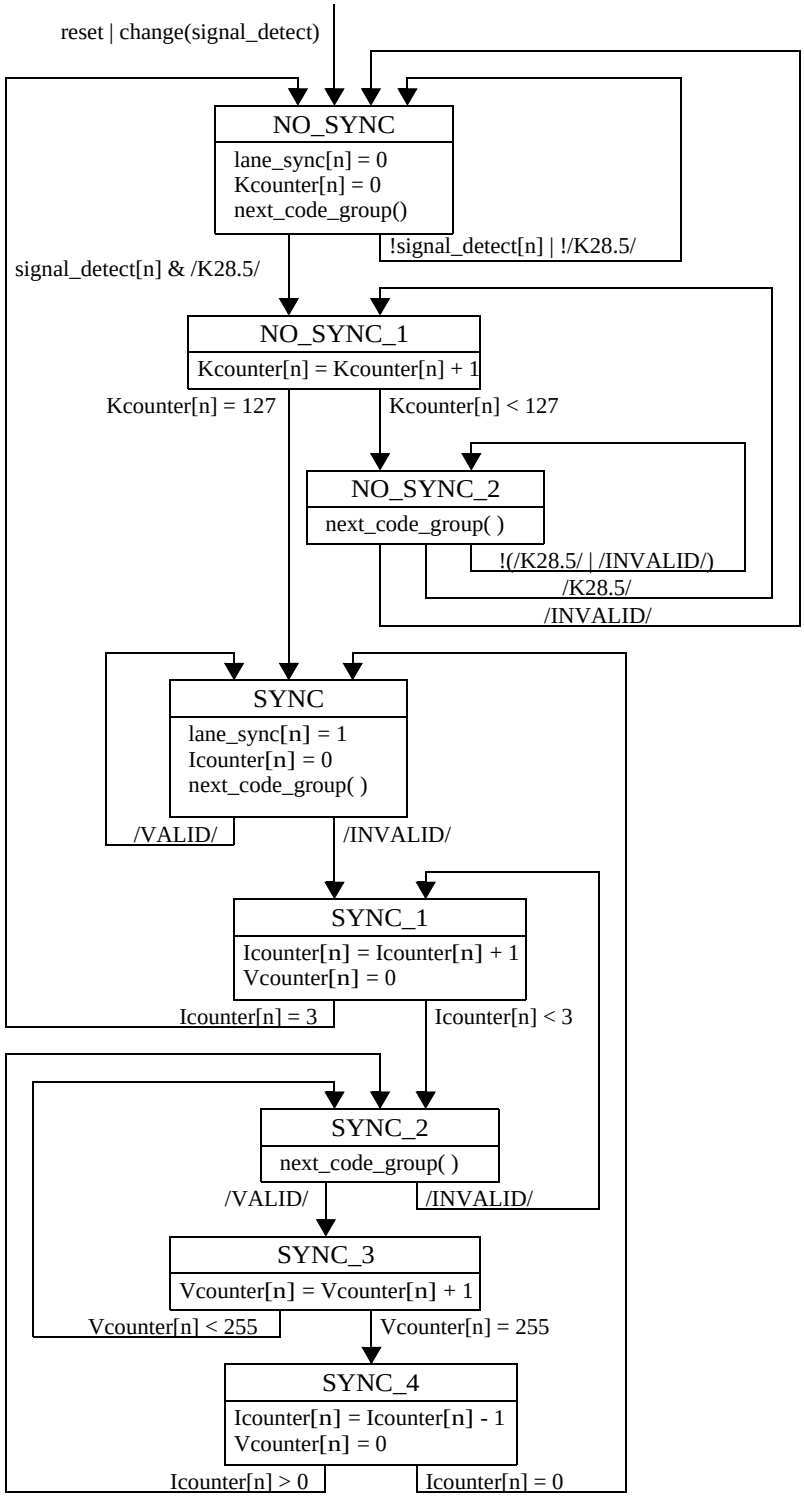


Figure 4-9. Lane_Synchronization State

4.7.3.4 Lane Alignment State Machine

The Lane_Alignment state machine monitors the alignment of the output of the four lane receivers in a port operating in 4x mode. A port supporting 4x mode has one Lane_alignment state machine, a port supporting only 1x mode does not have a Lane_Alignment state machine. (Lane alignment is required in a 4x port receiver to compensate for unequal propagation delays through the four lanes.)

The state machine determines the alignment state by monitoring the fours lanes for columns containing all /A/s (||A||), columns containing at least one but not all /A/s and columns containing no /A/s. Several counters are used to provide hysteresis so that occasional bit errors do not cause spurious lanes_aligned state changes.

The state machine does not specify how lane alignment is to be achieved. The methods used by a 4x port receiver to achieve lane alignment are implementation dependent. However, isolated single bit errors shall not cause the lane alignment mechanism to change lane alignment. For example, a isolated single bit error that results in a column that contains at least one /A/ but not all /A/s may not cause the lane alignment mechanism to change the lane alignment.

The state machine starts in the NOT_ALIGNED state where the variables Acounter and lanes_aligned are set to 0 (all lanes are not aligned). The machine then waits for all (four) lanes to achieve code-group boundary alignment (all_lanes_sync asserted) and the reception of an ||A|| (a column of all /A/s). When this obtains, the machine goes to NOT_ALIGNED_1 state.

The NOT_ALIGNED_1 state in combination with the NOT_ALIGNED_2 state looks for the reception of four ||A||s without the intervening reception of a misaligned column (a column with at least one /A/ but not all /A/s which causes the signal align_error to be asserted). When this obtains, the machine goes to the ALIGNED state. If an intervening misaligned column is received, the machine goes back to the NOT_ALIGNED state.

In the ALIGNED state, the machine sets the variable lanes_aligned to 1 (all lanes are aligned) and the variable Mcounter to 0 and looks for a misaligned column (align_error asserted). If a misaligned column is detected, the machine goes to the ALIGNED_1 state.

The ALIGNED_1 state in combination with the ALIGNED_2 and ALIGNED_3 states look for the reception of four ||A||s without the intervening reception of a misaligned column. If this condition obtains, the machine returns to the ALIGNED state. If an intervening misaligned column occurs, the machine goes to the NOT_ALIGNED state and starts over. This algorithm tolerates isolated single bit errors in that an isolated single bit error will not cause the machine to change the variable lanes_aligned from 1 to 0 (in lane alignment to out of lane alignment). The Lane_Alignment state machine is specified in Figure 4-10.

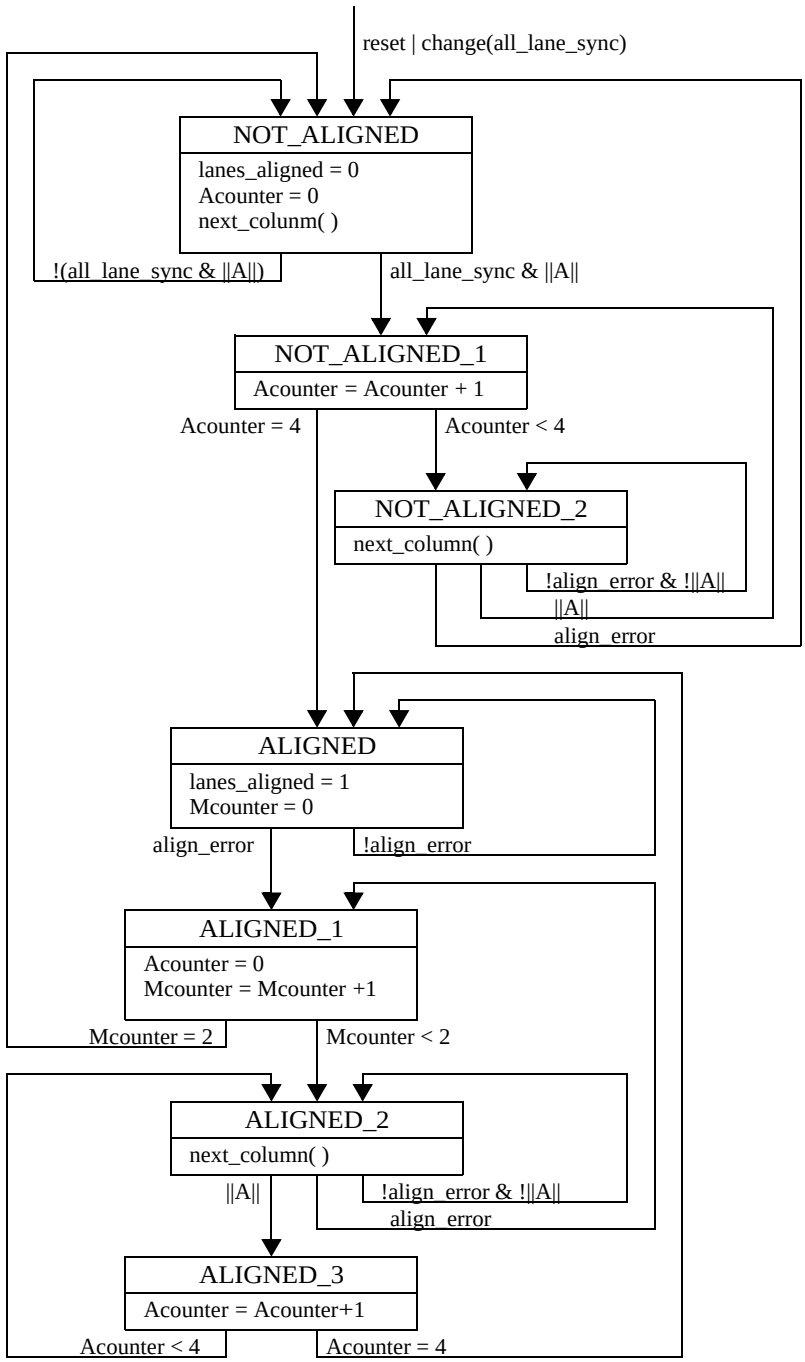


Figure 4-10. Lane_Alignment State Machine

4.7.3.5 1x Mode Initialization State Machine

The 1x_Initialization state machine shall be used by ports that support only 1x mode (1x ports).

The machine starts in the SILENT state. The link output driver is disabled to force the link partner to initialize regardless of its current state. The duration of the SILENT state is controlled by the silence_timer. The duration must be long enough to ensure that the link partner detects the silence (as a loss of lane_sync) and is forced to initialize but short enough that it is readily distinguished from a link break. When the silent interval is complete, the SEEK state is entered.

In the SEEK state, the link output driver is enabled, the idle sequence is transmitted, and the port waits for lane_sync to be asserted indicating the presence of a link partner. While lane_sync as defined indicates the bit and code-group boundary alignment synchronization state of the link receiver, it may also be thought of as indicating the presence of a link partner. When lane-sync is asserted, the 1X_MODE state is entered.

The input signal force_reinit allows the port to force link initialization at any time.

The variable port_initialized is asserted only in the 1X_MODE state. When port_initialized is deasserted, the port shall transmit a continuous idle sequence uninterrupted by control symbols or packets. To maintain receiver bit synchronization and code-group alignment, the port shall transmit the idle sequence when no control symbol or packet is being transmitted.

The 1x_Initialization state machine is specified in Figure 4-11.

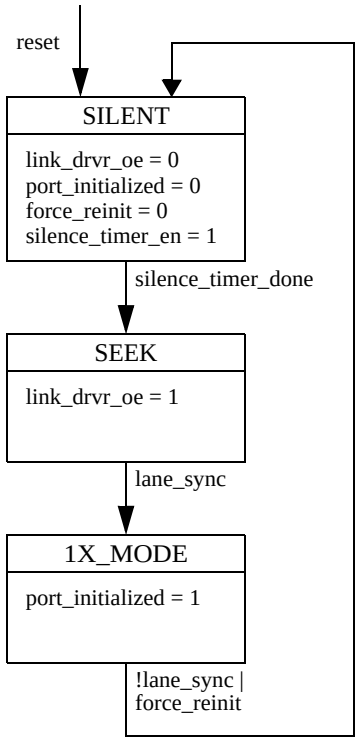


Figure 4-11. 1x_Initialization State Machine

4.7.3.6 1x/4x Mode Initialization State Machine

The 1x/4x_Initialization state machine shall be used by ports that support both 1x and 4x mode (1x/4x ports). In addition to determining when the link is initialized, the state machine controls whether the port receiver operates in 1x or 4x mode and in 1x mode whether lane 0 or lane 2 is selected as the inbound lane.

The machine starts in SILENT state. All four lane output drivers are disabled to force the link partner to initialize regardless of its current state. The duration of the SILENT state is controlled by the silence_timer. The duration must be long enough to ensure that the link partner detects the silence (as a loss of lane_sync) and is forced to initialize but short enough that it is readily distinguished from a link break. When the silent interval is complete, the SEEK state is entered.

In the SEEK state, a 1x/4x port transmits the idle sequence on lanes 0 and 2 (the lane 1 and lane 3 output drivers remain disabled to save power) and waits for an indication that a link partner is present. While lane_sync as defined indicates the bit and code-group boundary alignment synchronization state of a lane receiver, it is also used to indicate the presence of a link partner. A link partner is declared to be present when either lane_sync[0] or lane_sync[2] is asserted. If force_1x_mode is not asserted, the assertion of either lane_sync[0] or lane_sync[2] causes the state machine to enter the DISCOVERY state. If force_1x_mode is asserted, the state machine enters either the 1X_MODE_LANE0 or 1X_MODE_LANE2 state depending on whether lane_sync[0] or lane_sync[2] is asserted first and whether

force_lane2 is asserted.

In the DISCOVERY state, the port enables the output drivers for lanes 1 and 3 and transmits the idle sequence on all four lanes. The discovery_timer is also started. The discovery_timer allows time for the link partner to enter its DISCOVERY state and if the link partner is supporting 4x mode, for all four local lane receivers to acquire bit synchronization and code-group boundary alignment and for all four lanes to be aligned.

If lane alignment is achieved (lanes_aligned asserted) while in the DISCOVERY state, the machine enters the 4X_MODE state. It remains in this state until lane alignment or at least one lane_sync is lost (lanes_aligned deasserted) or reinitialization is forced (force_reinit is asserted).

At the end of the discovery period (discovery_timer_done is asserted), if lane alignment has not been achieved (lanes_aligned not asserted), the machine enters one of the 1x mode states. If code-group alignment has been achieved on lane 0 (lane_sync[0] asserted), the machine enters 1X_MODE_LANE0 and remains in that state until code-group boundary alignment is lost (lane0_sync deasserted) or reinitialization is forced (force_reinit is asserted). If code-group alignment has been achieved on lane 2 (lane_sync[2] asserted) but not on lane 0 (lane_sync[0] not asserted), the machine enters 1X_MODE_LANE2 and remains in that state until code-group boundary alignment is lost (lane_sync[2] deasserted) or reinitialization is forced (force_reinit is asserted).

If lane synchronization for both lane 0 and lane 2 is lost during the DISCOVERY state, it is not possible to successfully complete initialization and it is necessary to re-start the initialization process. Re-starting the initialization process is done by transitioning to the SILENT state. This results in the link partner also losing lane syncronization on both lanes 0 and 2, resulting in both ends of the link re-entering the SEEK state after the end of the respective silent periods (silence_timer_done asserted).

When in the 4X_MODE state, if lane alignment or at least one lane_sync is lost (lanes_aligned deasserted), the state machine transitions to either the SILENT state if both lane_sync[0] and land_sync[2] are deasserted or the DISCOVERY state if either lane_sync[0] or lane_sync[2] is asserted. This allows a 1x/4x port in the 4X_MODE state to return to 4X_MODE if lanes_aligned is deasserted due to multi-bit reception error, but also allows the port to switch to 1x mode if the connected 1x/4x port is not able to receive in 4x mode and has switched to 1x mode.

The input signals force_1x_mode and force_lane2 allow the state of the machine to be forced during initialization into 1x mode, and in 1x mode to be forced to receive on lane 2. The input signal force_reinit allows the port to force link initialization at any time.

The variable port_initialized is asserted only in the 1X_MODE_LANE0, 1X_MODE_LANE2 and 4X_MODE states. When port_initialized is deasserted, the

port shall transmit a continuous idle sequence uninterrupted by control symbols or packets. To maintain receiver bit synchronization and code-group alignment, the port shall transmit the idle sequence when no control symbol or packet is being transmitted.

The 1x/4x_Initialization state machine is specified in Figure 4-12.

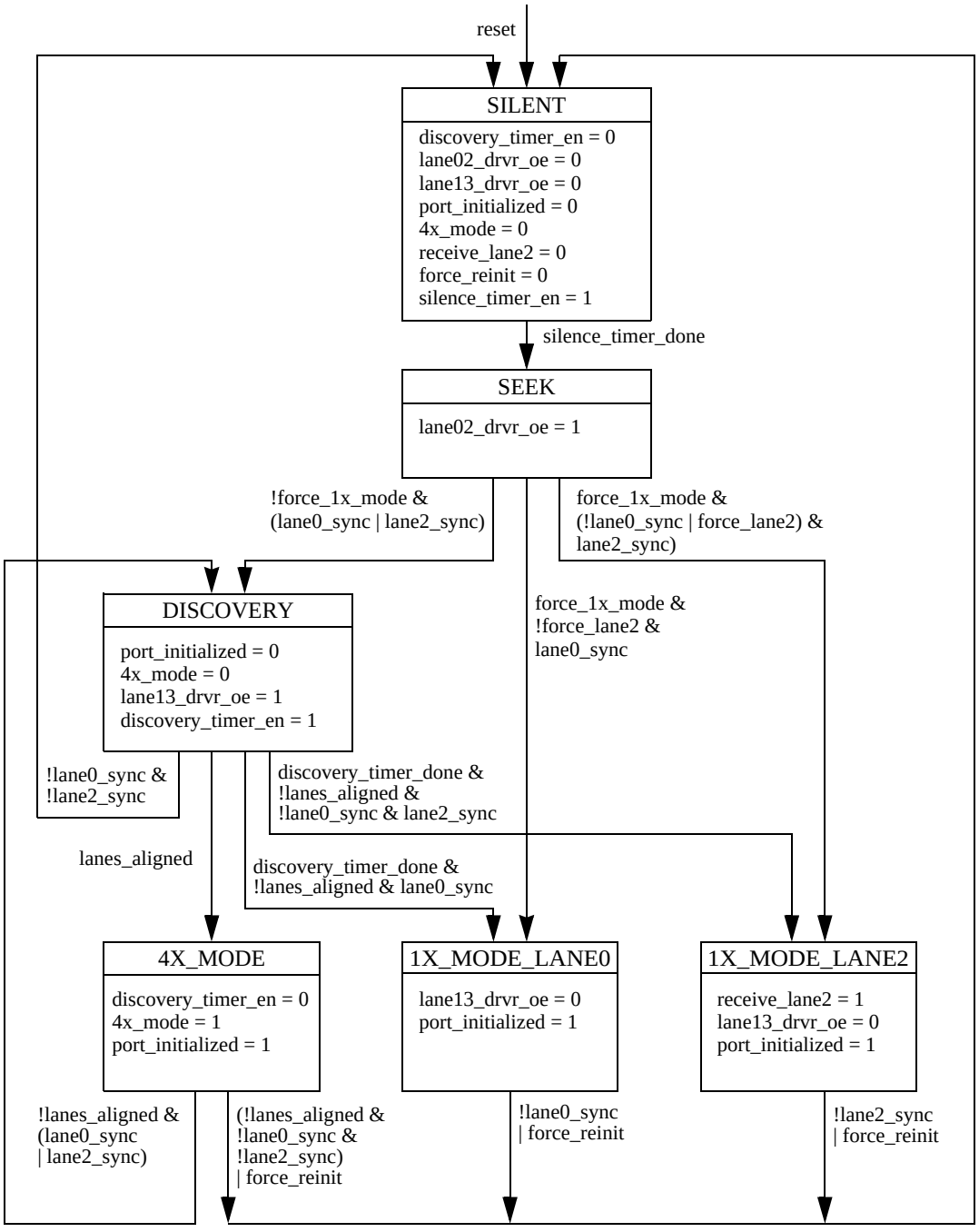


Figure 4-12. 1x/4x_Initialization State Machine

Chapter 5 LP-Serial Protocol

5.1 Introduction

This chapter specifies the LP-serial protocol. The protocol provides the reliable delivery of packets between two RapidIO devices that are connected by an LP-Serial link.

Packet priority, the mapping of transaction request flows onto packet priority, buffer management, and the use of control symbols in managing the delivery of packets between two devices is explained in this chapter.

5.2 Packet Exchange Protocol

This physical layer LP-Serial specification defines a protocol for devices connected by a LP-Serial link in which each packet transmitted by one device is acknowledged by control symbols transmitted by the other device. If a packet cannot be accepted for any reason, an acknowledgment control symbol indicates the reason and that the original packet and any transmitted subsequent packets must be resent. This behavior provides a flow control and error control mechanism between connected processing elements.

Figure 5-1 shows an example of transporting a request and response packet pair across an interconnect fabric with acknowledgments between the link transmitter/receiver pairs along the way. This allows flow control and error handling to be managed between each electrically connected device pair rather than between the original source and final target of the packet. An end point device shall transmit an acknowledgment control symbol for a request packet before transmitting the response packet corresponding to that request.

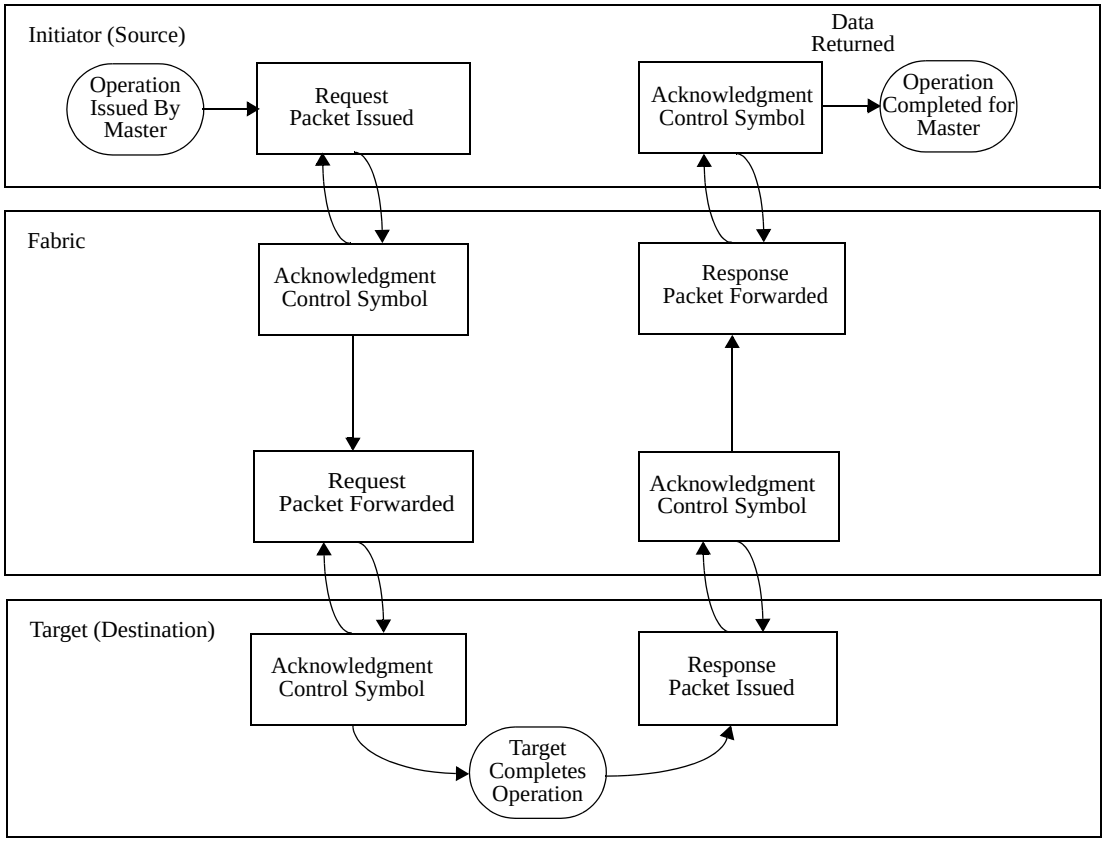


Figure 5-1. Example Transaction with Acknowledgment

5.3 Control Symbols

Control Symbols are the message elements used by ports connected by an LP-Serial link to manage all aspects of LP-Serial link operation. They are used for link maintenance, packet delimiting, packet acknowledgment, error reporting, and error recovery.

5.3.1 Control Symbol Delimiting

LP-Serial control symbols are delimited for transmission by a single 8B/10B special (control) character. The control character marks the beginning of the control symbol and immediately precedes the first bit of the control symbol. The control symbol delimiting special character is added to the control symbol before the control symbol is passed to the PCS sublayer for lane striping (if applicable) and 8B/10B encoding. Since control symbol length is constant and known, control symbols do not need end delimiters. The combined delimiter and control symbol is referred to as a delimited control symbol.

One of two special characters is used to delimit a control symbol. If the control

symbol contains a packet delimiter, the special character PD (K28.3) is used. If the control symbol does not contain a packet delimiter, the special character SC (K28.0) is used. This use of special characters provides the receiver with an “early warning” of the content of the control symbol.

5.3.2 Control Symbol Transmission

After an LP-Serial port is initialized, it shall transmit the idle sequence interrupted by one status control symbol every 1024 transmitted code-groups until the port has received an error free status control symbol from the connected port. The transmission of status control symbols indicates to the connected port that the port has completed initialization. The transmission of the idle sequence is required for the connected port to complete initialization.

After an initialized LP-Serial port has received an error free status control symbol from the connected port, the port shall transmit the idle sequence and a minimum of 15 status control symbols and shall receive an additional 6 error free status control symbols with no intervening detected errors before entering the normal operational state. Once in the normal operational state, the port may begin the transmission of packets and other (non-status) control symbols. This group of 15 status control symbols may be sent more rapidly than the minimum rate of one every 1024 code-groups. The requirement that a total of seven status control symbols be received provides a degree of link verification before packets and other control symbols are transmitted. The requirement that at least 15 status control symbols be transmitted is to minimize the probability that the connected port will not receive 7 control symbols even if a single bit error occurs.

When an LP-Serial port is in the normal operational state, it shall transmit a control symbol containing the buf_status field at least once every 1024 transmitted code-groups. To comply with this requirement, the port shall transmit a status control symbol if no other control symbol containing the buf_status field is available for transmission.

In each of the above cases, the time required to transmit 1024 code-groups shall be computed based on the aggregate unidirectional baud rate of the link. For example, a 4x link with each lane operating at 3.125 GBaud has an aggregate unidirectional baud rate of (4*3.125 =) 12.5 GBaud. In this example, the time required to transmit 1024 code-groups is (1024*10/12.5*10^9 =) 819.2 ns.

5.3.3 Embedded Control Symbols

Any control symbol that does not contain a packet delimiter may be embedded in a packet. An embedded control symbol may contain any defined encoding of stype0 and an stype1 encoding of “multicast-event” or “NOP”. Control symbols with stype1 encodings of start-of-packet, end-of-packet, stomp, restart-from-retry, or link-request cannot be embedded as they would terminate the packet.

When a control symbol is embedded in a packet, the control symbol delimiting special character shall begin on a 4-character boundary of the packet. That is, the number of packet characters between the start of the first packet character and the start of the delimited control symbol shall be a non-negative integer multiple of 4.

The manner and degree to which control symbol embedding is used on a link impacts both link and system performance. For example, embedding multicast-event control symbols allows their propagation delay and delay variation through switch processing elements to be minimized and is highly desirable for some multicast-event applications. On the other hand, embedding all packet acknowledgment control symbols rather than combining as many of them as possible with packet delimiter control symbols reduces the link bandwidth available for packet transmission and may be undesirable.

5.3.4 Multicast-Event Control Symbols

The Multicast-Event control symbol provides a mechanism through which end points are notified that some system defined event has occurred. This event can be selectively multicast through the system. Refer to Section 3.5.6 for the format of the multicast-event control symbol.

When a switch processing element receives a Multicast-Event control symbol, the switch shall forward the Multicast-Event by issuing a Multicast-Event control symbol from each port that is designated in the port's CSR as a Multicast-Event output port. A switch port shall never forward a Multicast-Event control symbol back to the device from which it received a Multicast-Event control symbol regardless of whether the port is designated a Multicast-Event output or not.

It is intended that at any given time, Multicast-Event control symbols will be sourced by a single device. However, the source device can change (in case of failover, for example). In the event that two or more Multicast-Event control symbols are received by a switch processing element close enough in time that more than one is present in the switch at the same time, at least one of the Multicast-Event control symbols shall be forwarded. The others may be forwarded or discarded (device dependent).

The system defined event whose occurrence Multicast-Event gives notice of has no required temporal characteristics. It may occur randomly, periodically, or anything in between. For instance, Multicast-Event may be used for a heartbeat function or for a clock synchronization function in a multiprocessor system.

In an application such as clock synchronization in a multiprocessor system, both the propagation time of the notification through the system and the variation in propagation time from Multicast-Event to Multicast-Event are of concern. For these reasons and the need to multicast, control symbols are used to convey Multicast-Events as control symbols have the highest priority for transmission on a link and can be embedded in packets.

While this specification places no limits on Multicast-Event forwarding delay or forwarding delay variation, switch functions should be designed to minimize these characteristics. In addition, switch functions shall include in their specifications the maximum value of Multicast-Event forwarding delay (the maximum value of Multicast-Event forwarding delay through the switch) and the maximum value of Multicast-Event forwarding delay variation (the maximum value of Multicast-Event forwarding delay through the switch minus the minimum value of Multicast-Event forwarding delay through the switch).

5.4 Packets

5.4.1 Packet Delimiting

LP-Serial packets are delimited for transmission by control symbols. Since packet length is variable, both start-of-packet and end-of-packet delimiters are required. The control symbol marking the end of a packet (packet termination) follows the end of the packet or the end of an embedded control symbol.

The following control symbols are used to delimit packets.

- Start-of-packet
- End-of-packet
- Stomp
- Restart-from-retry
- Any link-request

5.4.1.1 Packet Start

The beginning of a packet (packet start) shall be marked by a start-of-packet control symbol.

5.4.1.2 Packet Termination

A packet shall be terminated in one of the following three ways:

- The end of a packet is marked with an end-of-packet control symbol.
- The end of a packet is marked with a start-of-packet control symbol that also marks the beginning of a new packet.
- The packet is canceled by a restart-from-retry, stomp, or any link-request control symbol

5.4.2 Acknowledgment Identifier

Each packet requires an identifier to uniquely identify its acknowledgment control symbol. This identifier, the acknowledge ID (ackID), is five bits long, allowing for a range of one to thirty two outstanding unacknowledged request or

response packets between adjacent processing elements, however only up to thirty one outstanding unacknowledged packets are allowed at any one time. The first value of ackID assigned after a reset shall be 0b00000. Subsequent values of ackID shall be assigned sequentially (in increasing numerical order, wrapping back to 0 on overflow) to indicate the order of the packet transmission. The acknowledgments themselves are a number of control symbols defined in Chapter 3, "Control Symbols.

5.4.3 Packet Priority and Transaction Request Flows

Each packet has a priority, and optionally a critical request flow, that is assigned by the end point processing element that is the source of (initiates) the packet. The priority is carried in the prio field of the packet and has four possible values: 0, 1, 2, or 3. Packet priority increases with the priority value with 0 being the lowest priority and 3 being the highest. Packet priority is used in RapidIO for several purposes which include transaction ordering and deadlock prevention. The critical request flow is carried in the CRF bit. It allows a flow to be designated as a critical or preferred flow with respect to other flows of the same priority. Support for critical request flows is strongly encouraged.

When a transaction is encapsulated in a packet for transmission, the transaction request flow indicator (flowID) of the transaction is mapped into the prio field (and optionally the CRF bit) of the packet. If the CRF bit is not supported, transaction request flows A and B are mapped to priorities 0 and 1 respectively and transaction request flows C and above are mapped to priority 2 as specified in Table 5-1.

Table 5-1. Transaction Request Flow to Priority Mapping

Flow	System Priority	Request Packet Priority	Response Packet Priority
C or higher	Highest	2	3
B	Next	1	2 or 3
A	Lowest	0	1, 2, or 3

If the CRF bit is supported, the transaction request flows are mapped similarly as specified in Table 5-2. Devices that do not support the CRF bit treat it as reserved, setting it to logic 0 on transmit and ignoring it on receive.

Table 5-2. Transaction Request Flow to Priority and Critical Request Flow Mapping

Flow	System Priority	CRF Bit Setting	Request Packet Priority	Response Packet Priority
F or higher	Highest	1	2	3
E	Higher than A, B, C, D	0	2	3
D	Higher than A, B, C	1	1	2 or 3
C	Higher than A, B	0	1	2 or 3

Flow	System Priority	CRF Bit Setting	Request Packet Priority	Response Packet Priority
B	Higher than A	1	0	1, 2, or 3
A	Lowest	0	0	1, 2, or 3

The mapping of transaction request flows allows a RapidIO transport fabric to maintain transaction request flow ordering without the fabric having any knowledge of transaction types or their interdependencies. This allows a RapidIO fabric to be forward compatible as the types and functions of transactions evolve. A fabric can maintain transaction request flow ordering by simply maintaining the order of packets with the same priority and critical request flow for each path through the fabric and can maintain transaction request flow priority by never allowing a lower priority packet to pass a higher priority packet taking the same path through the fabric. In the case of congestion or some other restriction, a set CRF bit indicates that a flow of a priority can pass a flow of the same priority without the CRF bit set.

5.5 Link Maintenance Protocol

The link maintenance protocol involves a request and response pair between ports connected by an LP-Serial link. For software management, the request is generated through ports in the configuration space of the sending device. An external host write of a command to the link-request register with an I/O logical specification maintenance write transaction causes a link-request control symbol to be issued onto the output port of the device, but only one link-request can be outstanding on a link at a time.

The device that is linked to the sending device shall respond with an link-response control symbol if the link-request command required it to do so. The external host retrieves the link-response by polling the link-response register with I/O logical maintenance read transactions. A device with multiple RapidIO interfaces has a link-request and a link-response register pair for each corresponding RapidIO interface.

The automatic error recovery mechanism relies on the hardware generating link-request/input-status control symbols under the transmission error conditions described in Section 5.11.2.1, “Recoverable Errors and using the corresponding link-response information to attempt recovery.

Due to the undefined reliability of system designs, it is necessary to put a safety lockout on the reset function of the link-request/reset-device control symbol. A device receiving a link-request/reset-device control symbol shall not perform the reset function unless it has received four link-request/reset-device control symbols in a row without any intervening packets or other control symbols, except status control symbols. This will prevent spurious reset-device commands inadvertently resetting a device. The link-request/reset-device control symbol does not require a response.

The input-status command of the link-request/input-status control symbol is used by the hardware to recover from transmission errors. If the input port had stopped due to a transmission error that generated a packet-not-accepted control symbol back to the sender, the link-request/input-status control symbol acts as a link-request/restart-from-error control symbol, and the receiver is re-enabled to receive new packets after generating the link-response control symbol. The link-request/input-status control symbol may also be used to restart the receiving device if it is waiting for a restart-from-retry control symbol after retrying a packet. This situation can occur if transmission errors are encountered while trying to resynchronize the sending and receiving devices after the retry.

The link-request/input-status control symbol requires a response. A port receiving a link-request/input-status control symbol returns a link-response control symbol containing two pieces of information:

- port_status
- ackID_status

These status indicators are described in Table 3-5.

The retry-stopped state indicates that the port has retried a packet and is waiting to be restarted. This state is cleared when a restart-from-retry (or a link-request/input-status) control symbol is received. The error-stopped state indicates that the port has encountered a transmission error and is waiting to be restarted. This state is cleared when a link-request/input-status control symbol is received.

5.6 Packet Transmission Protocol

The LP-Serial protocol for packet transmission provides link level flow and error detection and recovery.

The LP-Serial link protocol uses control symbols to delimit packets when they are transmitted across an LP-Serial link as specified in Section 5.4.1, “Packet Delimiting.

The LP-Serial link protocol uses acknowledgment to monitor packet transmission. Each packet transmitted across an LP-Serial link shall be acknowledged by the receiving port with a packet acknowledgment control symbol. To associate packet acknowledgment control symbols with transmitted packets, each packet shall be assigned an ackID value that is carried in the ackID field of the packet and the packet_ackID field of the associated acknowledgment control symbol. AckID values are assigned to packets sequentially in increasing numerical order wrapping to 0 on overflow. The ackID value carried by packets indicates their order of transmission.

The LP-Serial link protocol uses retransmission to recover from packet transmission errors. To enable packet retransmission, a copy of each packet transmitted across an

LP-Serial link shall kept by the sending port until either a packet-accepted packet acknowledgment control symbol is received for the packet from the receiving port indicating that the port has received the packet without detected error and has accepted responsibility for the packet or the port determines that the packet has an encountered an unrecoverable error condition.

The LP-Serial link protocol uses the ackID value carried in each packet to ensure that no packets are lost due to transmission errors. A port shall accept packets from an LP-Serial link only in sequential ackID order, i.e. if the ackID value of the last packet accepted was N, the ackID value of the next packet that is accepted must be (N+1) modulo32.

An LP-Serial port accepts or rejects each error-free packet it receives depending on whether the port has input buffer space available at the priority level of the packet. The use of the packet-accepted, packet-retry, and restart-from-retry control symbols and the buf_status field in packet acknowledgment control symbols to control the flow of packets across an LP-Serial link is cover in Section 5.7, “Flow Control.

The LP-Serial link protocol allows a packet that is being transmitted to be canceled at any point during its transmission. Packet cancelation is covered in Section 5.8, “Canceling Packets.

The LP-Serial link protocol provides detection and recovery processes for both transmission errors and protocol violations. The enumeration of detectable errors, the detection of errors and the associated error recovery processes are covered in Section 5.11, “Error Detection and Recovery.

In order to prevent internal switch processing element internal errors, such as SRAM soft bit errors, from silently corrupting a packet and the system, switch processing elements shall maintain packet error detection coverage while a packet is passing through the switch. The simplest method for maintain packet error detection coverage is pass the packet CRC through the switch as part of the packet. This works well for all non-maintenance packets whose CRC does not change as the packets are transported from source to destination through the fabric. Maintaining error detection coverage is more complicated for maintenance packets as their hop_count and CRC change every time they pass through a switch.

In order to support transaction ordering requirements of the I/O Logical Layer specification, the LP-Serial protocol imposes packet delivery ordering requirements within the physical layer and transaction delivery ordering requirements between the physical layer and the transport layer in end point processing elements. These requirements are covered in Section 5.9, “Transaction and Packet Delivery Ordering Rules.

In order to prevent deadlock, the LP-Serial protocol imposes a set of deadlock prevention rules. These rules are covered in Section 5.10, “Deadlock Avoidance.

The LP-Serial specification does not require the use of fair bandwidth allocation

mechanisms within the transport fabric, therefore, it is possible that traffic associated with higher flow levels can starve traffic associated with lower flow levels. Any sort of starvation prevention, flow level bandwidth allocation, or fairness mechanisms are device and system dependent and are beyond the scope of this specification.

5.7 Flow Control

This section defines RapidIO LP-Serial link level flow control. The flow control operates between each pair of ports connected by an LP-Serial link. The purpose of link level flow control is to prevent the loss of packets due to a lack of buffer space in a link receiver.

The LP-Serial protocol defines two methods or modes of flow control. These are named receiver-controlled flow control and transmitter-controlled flow control. Every RapidIO LP-Serial port shall support receiver-controlled flow control. LP-Serial ports may optionally support transmitter-controlled flow control.

5.7.1 Receiver-Controlled Flow Control

Receiver-controlled flow control is the simplest and basic method of flow control. In this method, the input side of a port controls the flow of packets from its link partner by accepting or rejecting (retrying) packets on a packet by packet basis. The receiving port provides no information to its link partner about the amount of buffer space it has available for packet reception.

As a result, its link partner transmits packets with no *a priori* expectation as to whether a given packet will be accepted or rejected. A port signals its link partner that it is operating in receiver-controlled flow control mode by setting the buf_status field to all 1's in every control symbol containing the field that the port transmits. This method is named receiver-controlled flow control because the receiver makes all of the decisions about how buffers in the receiver are allocated for packet reception.

A port operating in receiver-controlled flow control mode accepts or rejects each inbound error-free packet based on whether the receiving port has enough buffer space available at the priority level of the packet. If there is enough buffer space available, the port accepts the packet and transmits a packet-accepted control symbol to its link partner that contains the ackID of the accepted packet in its packet_ackID field. This informs the port's link partner that the packet has been received without detected errors and that it has been accepted by the port. On receiving the packet-accepted control symbol, the link partner discards its copy of the accepted packet freeing buffer space in the partner.

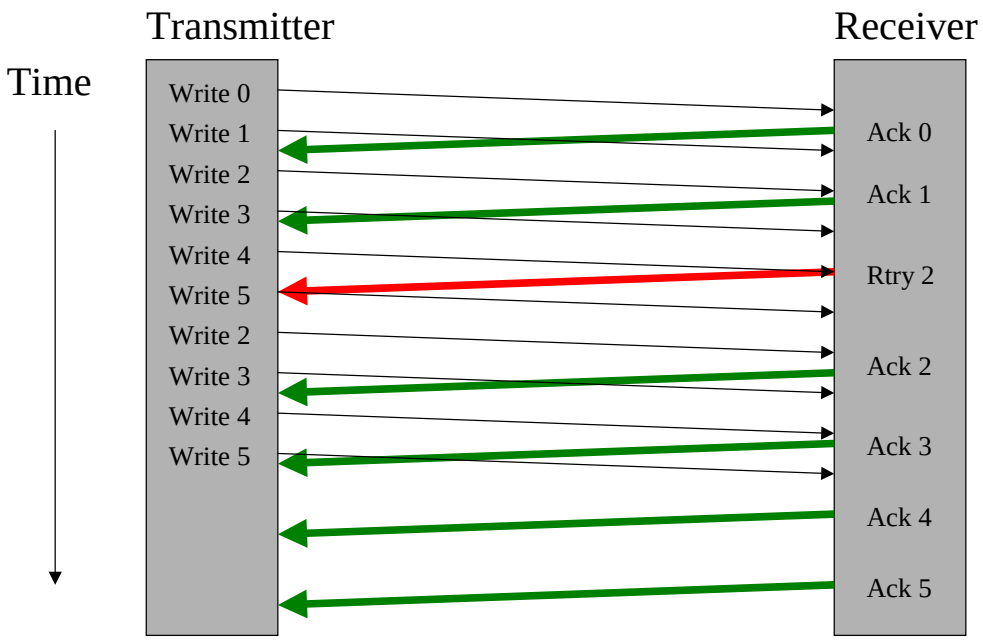
If buffer space is not available, the port rejects the packet. When a port rejects (retries) an error-free packet, it immediately enters the Input Retry-stopped state and follows the Input Retry-stopped recovery process specified in Section 5.7.2.1,

"Input Retry-Stopped Recovery Process. As part of the Input Retry-stopped recovery process, the port sends a packet-retry control symbol to its link partner indicating that the packet whose ackID is in the packet_ackID field of the control symbol and all packets subsequently transmitted by the port have been discarded by the link partner and must all be retransmitted. The control symbol also indicates that the link partner is temporarily out of buffers for packets of priority less than or equal to the priority of the retried packet.

A port that receives a packet-retry control symbol immediately enters the Output Retry-stopped state and follows the Output Retry-stopped recovery process specified in Section 5.7.2.2, "Output Retry-Stopped Recovery Process. As part of the Output Retry-stopped recovery process, the port receiving the packet-retry control symbol sends a restart-from-retry control symbol which causes its link partner to exit the Input Retry-stopped state and resume packet reception. The ackID assigned to that first packet transmitted after the restart-from-retry control symbol is the ackID of the packet that was retried.

Figure 5-2 shows an example of receiver-controlled flow control operation. In this example the transmitter is capable of sending packets faster than the receiver is able to absorb them. Once the transmitter has received a retry for a packet, the transmitter may elect to cancel any packet that is presently being transmitted since it will be discarded anyway. This makes bandwidth available for any higher priority packets that may be pending transmission.

Figure 5-2. Receiver-Controlled Flow Control



5.7.2 Transmitter-Controlled Flow Control

In transmitter-controlled flow control, the receiving port provides information to its link partner about the amount of buffer space it has available for packet reception. With this information, the sending port can allocate the use of the receiving port's receive buffers according to the number and priority of packets that the sending port has waiting for transmission without concern that one or more of the packets shall be forced to retry.

A port signals its link partner that it is operating in transmitter-controlled flow control mode by setting the buf_status field to a value different from all 1's in every control symbol containing the field that the port transmits. This method is named transmitter-controlled flow control because the transmitter makes almost all of the decisions about how the buffers in the receiver are allocated for packet reception.

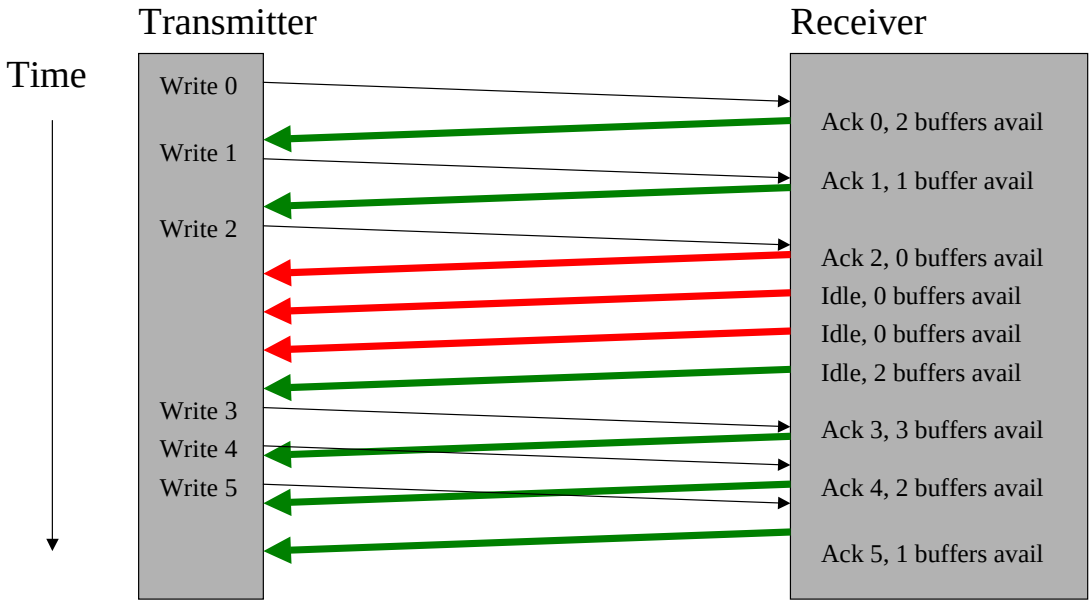
The number of free buffers that a port has available for packet reception is conveyed to its link partner by the value of the buf_status field in the control symbols that the port transmits. The value conveyed by the buf_status field is the number of maximum length packet buffers currently available for packet reception up to the limit that can be reported in the field. If a port has more buffers available than the maximum value that can be reported in the buf_status field, the port sets the field to that maximum value. A port may report a smaller number of buffers than it actually has available, but it shall not report a greater number.

A port informs its link partner when the number of free buffers available for packet reception changes. The new value of buf_status is conveyed in the buf_status field of a packet-accepted, packet-retry, or status control symbol. Each change in the number of free buffers a port has available for packet reception need not be conveyed to the link partner. However, a port shall send a control symbol containing the buf_status field to its link partner no less often than the minimum rate specified in Section 5.3.2, "Control Symbol Transmission.

A port whose link partner is operating in transmitter-control flow control mode should never receive a packet-retry control symbol from its link partner unless the port has transmitted more packets than its link partner has receive buffers, violated the rules that all input buffer may not be filled with low priority packets or there is some fault condition. If a port whose link partner is operating in transmitter-control flow control mode receives a packet-retry control symbol, the output side of the port immediately enters the Output Retry-stopped state and follows the Output Retry-stopped recovery process specified in Section 5.7.2.2, "Output Retry-Stopped Recovery Process.

A simple example of transmitter-controlled flow control is shown in Figure 5-3.

Figure 5-3. Transmitter-Controlled Flow Control



5.7.2.1 Input Retry-Stopped Recovery Process

When the input side of a port retries a packet, it immediately enters the Input Retry-stopped state. To recover from this state, the input side of the port takes the following actions.

- Discards the rejected or canceled packet without reporting a packet error and ignores all subsequently received packets while the port is in the Input Retry-stopped state.
- Causes the output side of the port to issue a packet-retry control symbol containing the ackID value of the retried packet in the packet_ackID field of the control symbol. (The packet-retry control symbol causes the output side of the link partner to enter the Output Retry-stopped state and send a restart-from-retry control symbol.)
- When a restart-from-retry control symbol is received, exit the Input Retry-stopped state and resume packet reception.

An example state machine with the behavior described in this section is included in Section A.2, “Packet Retry Mechanism.”

5.7.2.2 Output Retry-Stopped Recovery Process

To recover from the Output Retry-stopped state, the output side of a port takes the following actions.

- Immediately stops transmitting new packets. Resets the link packet acknowledgment timers for all transmitted but unacknowledged packets. (This prevents the generation of spurious time-out errors.)
- Transmits a restart-from-retry control symbol.
- Backs up to the first unaccepted packet (the retried packet) which is the packet whose ackID value is specified by the packet_ackID value contained in the packet-retry control symbol. (The packet_ackID value is also the value of ackID field the port retrying the packet expects in the first packet it receives after receiving the restart-from-retry control symbol.)
- Exits the Output Retry-stopped state and resumes transmission with either the retried packet or a higher priority packet which is assigned the ackID value contained in the packet_ackID field of the packet-retry control symbol.

An example state machine with the behavior described in this section is included in Section A.2, “Packet Retry Mechanism.”

5.7.2.3 Receive Buffer Management

In transmitter-controlled flow control, the transmitter manages the packet receive buffers in the receiver. This may be done in a number of ways, but the selected method shall not violate the rules in Section 5.10, “Deadlock Avoidance concerning the acceptance of packets by ports

One possible implementation to organize the buffers is establish watermarks and use them to progressively limit the packet priorities that can be transmitted as the effective number of free buffers in the receiver decreases. For example, RapidIO LP-Serial has four priority levels. Three non-zero watermarks are needed to progressively limit the packet priorities that may be transmitted as the effective number of free buffers decreases. Designate the three watermarks as WM0, WM1, and WM2 where WM0 > WM1 > WM2 > 0 and employ the following rules.

> If free_buffer_count >= WM0, all priority packets may be transmitted.
>
> If WM0 > free_buffer_count >= WM1, only priority 1, 2, and 3 packets may be transmitted.
>
> If WM1 > free_buffer_count >= WM2, only priority 2 and 3 packets may be transmitted.
>
> If WM2 > free_buffer_count, only priority 3 packets may be transmitted.

If this method is implemented, the initial values of the watermarks may be set by the hardware at reset as follows.

WM0 = 4

WM1 = 3

WM2 = 2

These initial values may be modified by hardware or software. The modified watermark values shall be based on the number of free buffers reported in the buf_status field of status control symbols received by the port following link initialization and before the start of packet transmission.

The three watermark values and the number of free buffers reported in the buf_status field of status control symbols received by the port following link initialization and before the start of packet transmission may be stored in a CSR. Since the maximum value of each of these four items is 30, each will fit in an 8-bit field and all four will fit in a single 32-bit CSR. If the watermarks are software setable, the three watermark fields in the CSR should be writable. For the greatest flexibility, a watermark register should be provided for each port on a device.

5.7.2.4 Effective Number of Free Receive Buffers

The number of buffers available in a port's link partner for packet reception is typically less than the value of the buf_status field most recently received from the link partner. The value in the buf_status field does not account for packets that have been transmitted by the port but not acknowledged by its link partner. The variable free_buffer_count is defined to be the effective number of free buffers available in the link partner for packet reception. The value of free_buffer_count shall be determined according to the following rules.

The port shall maintain a count of the packets that it has transmitted but that have not been acknowledged by its link partner. This count is named the outstanding_packet_count.

After link initialization and before the start of packet transmission,

```
If (received_buf_status < 31) {
        flow_control_mode = transmitter;
        free_buffer_count = received_buf_status;
        outstanding_packet_count = 0;
}
else
        flow_control_mode = receiver;
```

When a packet is transmitted by the port,

```
outstanding_packet_count =
                        outstanding_packet_count + 1;
```

When a status control symbol is received by the port,

```
free_buffer_count = received_buf_status -
```

```
                                        outstanding_packet_count;
```

When a packet-accepted control symbol is received by the port indicating that a packet has been accepted by the link partner,

```
Outstanding_packet_count =
                                    Outstanding_packet_count - 1;
free_buffer_count = received_buf_status -
                                        outstanding_packet_count;
```

When a packet-retry control symbol is received by the port indicating that a packet has been forced by the link partner to retry,

```
Outstanding_packet_count = 0;
free_buffer_count = received_buf_status;
```

When a packet-not-accepted control symbol is received by the port indicating that a packet has been rejected by the link partner because of one or more detected errors,

```
Outstanding_packet_count = 0;
free_buffer_count = 0;
```

The port then transmits a link-request/input-status (for input-status) control symbol and waits for the link partner to respond with a link-response control symbol. When the link-response control symbol is received,

```
free_buffer_count = received_buf_status;
```

5.7.2.5 Speculative Packet Transmission

A port whose link partner is operating in transmitter-controlled flow control mode may send more packets than the number of free buffers indicated by the link partner. Packets transmitted in excess of the free_buffer_count are transmitted on a speculative basis and are subject to retry by the link partner. The link partner accepts or rejects these packets on a packet by packet basis in exactly the same way it would if operating in receiver-controlled flow control mode. A port may use such speculative transmission in an attempt to maximize the utilization of the link. However, speculative transmission that results in a significant number of retries and discarded packets can reduce the effective bandwidth of the link.

5.7.3 Flow Control Mode Negotiation

Immediately following the initialization of a link, each port begins sending status control symbols to its link partner. The value of the buf_status field in these control symbols indicates to the link partner the flow control mode supported by the sending port.

The flow control mode negotiation rule is as follows:

> If the port and its link partner both support transmitter-controlled flow control, then both ports shall use transmitter-controlled flow control. Otherwise, both ports shall use receiver-controlled flow control.

5.8 Canceling Packets

When a port becomes aware of some condition that will require the packet it is currently transmitting to be retransmitted, the port may cancel the packet. This allows the port to avoid wasting bandwidth by not completing the transmission of a packet that the port knows must be retransmitted. Alternatively, the sending port may choose to complete transmission of the packet normally.

A port may cancel a packet if the port detects a problem with the packet as it is being transmitted or if the port receives a packet-retry or packet-not-accepted control symbol for a packet that is still being transmitted or that was previously transmitted. A packet-retry or packet-not-accepted control symbol can be transmitted by a port for a packet at any time after the port begins receiving the packet.

The sending device shall use the stomp control symbol, the restart-from-retry control symbol (in response to a packet-retry control symbol), or link-request/input-status control symbol (in response to a packet-not-accepted control symbol) or any link request control symbol to cancel a packet.

A port receiving a canceled packet shall drop the packet. The cancelation of a packet shall not result in the generation of any errors. If the packet was canceled because the sender received a packet-not-accepted control symbol, the error that caused the packet-not-accepted to be sent shall be reported in the normal manner.

The behavior of a port that receives a canceled packet depends on the control symbol that canceled the packet. A port that is not in an input stopped state (Retry-stopped or Error-stopped) while receiving the canceled packet and has not previously acknowledged the packet shall have the following behavior. If the packet is canceled by a link-request/input-status control symbol, the port shall drop the packet without reporting a packet error. If the packet is canceled by a restart-from-retry control symbol a protocol error has occurred and the port enters the Input Error-stopped state and follows the Input Error-stopped recovery process specified in Section 5.11.2.6, “Input Error-Stopped Recovery Process. If the packet was canceled by other than a restart-from-retry or link-request/input-status control symbol, the port shall immediately enter the Input Retry-Stopped state and follow the Input Retry-Stopped recovery process specified in Section 5.7.2.1, “Input Retry-Stopped Recovery Process. The Input Retry-Stopped recovery process includes the dropping of the canceled packet without reporting a packet error and the transmission of a packet-retry control symbol. In either case, if the packet was canceled before the packet ackID field was received by the port, the packet_ackID field of the associated retry control symbol will be set to the ackID the port expected

in the canceled packet. The packet sent following a canceled packet has the same ackID value as the canceled packet.

5.9 Transaction and Packet Delivery Ordering Rules

The rules specified in this section are required for the physical layer to support the transaction ordering rules specified in the logical layer specifications.

Transaction Delivery Ordering Rules:

1. **The physical layer of an end point processing element port shall encapsulate in packets and forwarded to the RapidIO fabric transactions comprising a given transaction request flow in the same order that the transactions were received from the transport layer of the processing element.**
2. **The physical layer of an end point processing element port shall ensure that a higher priority request transaction that it receives from the transport layer of the processing element before a lower priority request transaction with the same sourceID and the same destinationID is forwarded to the fabric before the lower priority transaction.**
3. **The physical layer of an end point processing element port shall deliver transactions to the transport layer of the processing element in the same order that the packetized transactions were received by the port.**

Packet Delivery Ordering Rules:

1. **A packet initiated by a processing element shall not be considered committed to the RapidIO fabric and does not participate in the packet delivery ordering rules until the packet has been accepted by the device at the other end of the link. (RapidIO does not have the concept of delayed or deferred transactions. Once a packet is accepted into the fabric, it is committed.)**
2. **A switch shall not alter the priority or critical request flow of a packet.**
3. **Packet forwarding decisions made by a switch processing element shall provide a consistent output port selection which is based solely on the value of the destinationID field carried in the packet.**
4. **A switch processing element shall not change the order of packets comprising a transaction request flow (packets with the same sourceID, the same destinationID, the same priority, same critical request flow and ftype != 8) as the packets pass through the switch.**
5. **A switch processing element shall not allow lower priority non-maintenance packets (ftype != 8) to pass higher priority**

non-maintenance packets with the same sourceID and destinationID as the packets pass through the switch.

6. **A switch processing element shall not allow a priority N maintenance packet (ftype = 8) to pass another maintenance packet of priority N or greater that takes the same path through the switch (same switch input port and same switch output port).**

5.10 Deadlock Avoidance

To allow a RapidIO protocol to evolve without changing the switching fabric, switch processing elements are not required, with the sole exception of ftype 8 maintenance transactions, to discern between packet types, their functions or their interdependencies. Switches, for instance, are not required to discern between packets carrying request transactions and packets carrying response transactions. As a result, it is possible for two end points, A and B to each fill all of their output buffers, the fabric connecting them and the other end point's input buffers with read requests. This would result in an input to output dependency loop in each end point in which there would be no buffer space to hold the responses necessary to complete any of the outstanding read requests.

To break input to output dependencies, end point processing elements must have the ability to issue outbound response packets even if outbound request packets awaiting transmission are congestion blocked by the connected device. Two techniques are provided to break input to output dependencies. First, a response packet (a packet carrying a response transaction) is always assigned an initial priority one priority level greater than the priority of the associated request packet (the packet carrying the associated request transaction).

This requirement is specified in Table 5-1. It breaks the dependency cycle at the request flow level. Second, the end point processing element that is the source of the response packet may additionally raise the priority of the response packet to a priority higher than the minimum required by Table 5-1 if necessary for the packet to be accepted by the connected device. This additional increase in response packet priority above the minimum required by Table 5-1 is called promotion. An end point processing element may promote a response packet only to the degree necessary for the packet to be accepted by the connected device.

The following rules define the deadlock prevention mechanism:

Deadlock Prevention Rules:

1. **A RapidIO fabric shall be dependency cycle free for all operations that do not require a response. (This rule is necessary as there are no mechanisms provided in the fabric to break dependency cycles for operations not requiring responses.)**

2. **A packet carrying a request transaction that requires a response shall not be issued at the highest priority. (This rule ensures that an end point processing element can issue a response packet at a priority higher then the priority of the associated request. This rule in combination with rule 3 are basis for the priority assignments in Table 5-1.)**
3. **A packet carrying a response shall have a priority at least one priority level higher than the priority of the associated request. (This rule in combination with rule 2 are basis for the priority assignments in Table 5-1.)**
4. **A switch processing element port shall accept an error-free packet of priority N if there is no packet of priority greater than or equal to N that was previously received by the port and is still waiting in the switch to be forwarded. (This rule has multiple implications which include but are not limited to the following. First, a switch processing element port must have at least as many maximum length packet input buffers as there are priority levels. Second, a minimum of one maximum length packet input buffer must be reserved for each priority level. A input buffer reserved for priority N might be restricted to only priority N packets or might be allowed to hold packets of priority greater than or equal to N, either approach complies with the rule.)**
5. **A switch processing element port that transmits a priority N packet that is forced to retry by the connected device shall select a packet of priority greater than N, if one is available, for transmission. (This guarantees that packets of a given priority will not block higher priority packets.)**
6. **An end point processing element port shall accept an error-free packet of priority N if the port has enough space for the packet in the input buffer space of the port allocated for packets of priority N. (Lack of input buffer space is the only reason an end point may retry a packet.)**
7. **The decision of an end point processing element to accept or retry an error-free packet of priority N shall not dependent on the ability of the end point to issue request packets of priority less than or equal to N from any of its ports. (This rule works in conjunction with rule 6. It prohibits a device's inability to issue packets of priority less than or equal to N, due to congestion in the connected device, from resulting in a lack of buffers to receive inbound packets of priority greater than or equal to N which in turn would result in packets of priority greater than or equal to N being forced to retry. The implications and some ways of complying with this rule are presented in the following paragraphs.)**

One implication of Rule 7 is that a port may not fill all of its buffers that can be used to hold packets awaiting transmission with packets carrying request transactions. If this situation was allowed to occur and the output was blocked due to congestion in

the connected device, read transactions could not be processed (no place to put the response packet), input buffer space would become filled and all subsequent inbound request packets would be forced to retry violating Rule 7.

Another implication is that a port must have a way of preventing output blockage at priority less than or equal to N, due to congestion in the connected device, from resulting in a lack of input buffer space for inbound packets of priority greater than or equal to N. There are multiple ways of doing this.

One way is to provide a port with input buffer space for at least four maximum length packets and reserve input buffer space for higher priority packets in a manner similar to that required by Rule 4 for switches. In this case, output port blockage at priority less than or equal to N will not result is blocking inbound packets of priority greater than or equal to N as any responses packets they generate will be of priority greater than N which is not congestion blocked. The port must however have the ability to select packets of priority greater than N for transmission from the packets awaiting transmission. This approach does not require the use of response packet priority promotion.

Alternatively, a port that does not have enough input buffer space for at least four maximum length packets or that does not reserve space for higher priority packets can use the promotion mechanism to increase the priority of response packets until they are accepted by the connected device. This allows output buffer space containing response packets to be freed even though all request packets awaiting transmission are congestion blocked.

As an example, suppose an end point processing element has a blocked input port because all available resources are being used for a response packet that the processing element is trying to send. If the response packet is retried by the downstream processing element, raising the priority of the response packet until it is accepted allows the processing element’s input port to unblock so the system can make forward progress.

5.11 Error Detection and Recovery

Error detection and recovery is becoming a more important issue for many systems. The LP-Serial specification provides extensive error detection and recovery by combining retry protocols with cyclic redundancy codes, the selection of delimiter control characters and response timers.

One feature of the error protection strategy is that with the sole exception of maintenance packets, the CRC value carried in a packet remains unchanged as the packet moves through the fabric. The CRC carried in a maintenance packet must be regenerated at each switch as the hop count changes.

5.11.1 Lost Packet Detection

Some types of errors, such as a lost request or response packet or a lost acknowledgment, result in a system with hung resources. To detect this type of error there shall be time-out counters that expire when sufficient time has elapsed without receiving the expected response from the system. Because the expiration of one of these timers should indicate to the system that there is a problem, this time interval should be set long enough so that a false time-out is not signaled. The response to this error condition is implementation dependent.

The RapidIO specifications require time-out counters for the physical layer, the port link time-out counters, and counters for the logical layer, the port response time-out counters. The interpretation of the counter values is implementation dependent, based on a number of factors including link clock rate, the internal clock rate of the device, and the desired system behavior.

The physical layer time-out occurs between the transmission of a packet and the receipt of an acknowledgment control symbol. This time-out interval is likely to be comparatively short because the packet and acknowledgment pair must only traverse a single link.

The logical layer time-out occurs between the issuance of a request packet that requires a response packet and the receipt of that response packet. This time-out is counted from the time that the logical layer issues the packet to the physical layer to the time that the associated response packet is delivered from the physical layer to the logical layer. Should the physical layer fail to complete the delivery of the packet, the logical layer time-out will occur. This time-out interval is likely to be comparatively long because the packet and response pair have to traverse the fabric at least twice and be processed by the target. Error handling for a response time-out is implementation dependent.

Certain GSM operations may require two response transactions, and both must be received for the operation to be considered complete. In the case of a device implementation with multiple links, one response packet may be returned on the same link where the operation was initiated and the other response packet may be returned on a different link. If this behavior is supported by the issuing processing element, the port response time-out implementation must look for both responses, regardless on which links they are returned.

5.11.2 Link Behavior Under Error

The LP-Serial link uses an error detection and retransmission protocol to protect against and recover from transmission errors. Transmission error detection is done at the input port, and all transmission error recovery is also initiated at the input port.

The protocol requires that each packet transmitted be acknowledged by the receiving port and that a copy of each transmitted packet be retained by the sender

until the sender receives a packet-accepted control symbol acknowledgment for the packet or the sending port determines that the packet has encountered an unrecoverable error. If the receiving port detects a transmission error in a packet, the port sends a packet-not-accepted control symbol acknowledgment back to the sender indicating that the packet was corrupted as received. After a link-request/input-status and link-response control symbol exchange, the sender begins retransmission with either the packet that was corrupted during transmission or a higher priority packet if one is awaiting transmission.

All packets corrupted in transmission are retransmitted. The number of times a packet may be retransmitted before the sending port determines that the packet has encountered an unrecoverable condition is implementation dependent.

5.11.2.1 Recoverable Errors

The following four basic types of errors are detected by an LP-Serial port:

- An idle sequence error
- A control symbol error
- A packet error
- A time-out waiting for an acknowledgment control symbol

5.11.2.2 Idle Sequence Errors

The idle sequence is comprised of A, K, and R (8B/10B special) characters. If an input port detects an invalid character or any valid character other then A, K, or R in an idle sequence, it shall enter the Input Error-stopped state and follow the Input Error-stopped recovery process specified in Section 5.11.2.6, “Input Error-Stopped Recovery Process.

To limit input port complexity, the port is no required to determine the specific error that resulted in an idle sequence error. Following are several examples of idle sequence errors.

- A single bit transmission error can change an /A/, /K/, or /R/ code-group into a /Dx.y/ (data) code-group which is illegal in an idle sequence.
- A single bit transmission error can change an /A/, /K/, or /R/ code-group into an invalid code-group.
- A single bit transmission error can change an /SP/ or /PD/ (control symbol delimiters) into an invalid code-group.

5.11.2.3 Control Symbol Errors

There are two types of detectable control symbol errors

- An uncorrupted control symbol that violates the link protocol
- A corrupted control symbol

5.11.2.3.1 Link Protocol Violations

The reception of a control symbol with no detected corruption that violates the link protocol shall cause the receiving port to immediately enter the appropriate Error-stopped state. Stype1 control symbol protocol errors shall cause the receiving port to immediately enter the Input Error-stopped state if not already in the Input Error-stopped state and follow the Error-stopped recovery process specified in Section 5.11.2.6, “Input Error-Stopped Recovery Process”. Stype0 control symbol protocol errors shall cause the receiving port to immediately enter the Output Error-stopped state if not already in the Output Error-stopped state and follow the Output Error-stopped recovery process specified in Section 5.11.2.7, “Output Error-Stopped Recovery Process”. If both stype0 and stype1 control symbols contain protocol errors, then the receiving port shall enter both Error-stopped states and follow both error recovery processes.

Link protocol violations include the following:

- Unexpected packet-accepted, packet-retry, or packet-not-accepted control symbol
- Packet acknowledgment control symbol with an unexpected packet_ackID value
- Link time-out while waiting for an acknowledgment control symbol

The following is an example of a link protocol violation and recovery. A sender transmits packets labeled ackID 2, 3, 4, and 5. It receives acknowledgments for packets 2, 4, and 5, indicating a probable error associated with ackID 3. The sender then stops transmitting new packets and sends a link-request/input-status (restart-from-error) control symbol to the receiver. The receiver then returns a link-response control symbol indicating which packets it has received properly. These are the possible responses and the sender’s resulting behavior:

- expecting ackID = 3 - sender must retransmit packets 3, 4, and 5
- expecting ackID = 4 - sender must retransmit packets 4 and 5
- expecting ackID = 5 - sender must retransmit packet 5
- expecting ackID = 6 - receiver got all packets, resume operation
- expecting ackID = anything else - fatal (non-recoverable) error

5.11.2.3.2 Corrupted Control symbols

The reception of a control symbol with detected corruption shall cause the receiving port to immediately enter the Input Error-stopped state and follow the Input Error-stopped recovery process specified in Section 5.11.2.6, “Input Error-Stopped Recovery Process. For this type of error, the packet-not-accepted control symbol sent by the output side of the port as part of the recovery process shall have an undefined packet_ackID value.

Input ports detect the following types of control symbol corruption.

- A control symbol containing invalid characters or valid but non-data characters
- A control symbol with an incorrect CRC value

5.11.2.4 Packet Errors

The reception of a packet with detected corruption shall cause the receiving port to immediately enter the Input Error-stopped state and follow the Input Error-stopped recovery process specified in Section 5.11.2.6, “Input Error-Stopped Recovery Process.

Input ports detect the following types of packet corruption

- Packet with an unexpected ackID value
- Packet with an incorrect CRC value
- Packet containing invalid characters or valid non-data characters
- Packet that overruns some defined boundary such as the maximum data payload.

An optional alternative behavior, error recovery suppression, can be enabled with the Re-transmit Suppression Mask field in the Port *n* Control CSR in systems that can withstand “lossy” transaction request flows (transaction request flows that are not required to have guaranteed data delivery). If this behavior is supported and enabled, when a processing element receives a corrupted packet as indicated by a bad CRC value, the processing element sends a packet-accepted control symbol, discards the corrupted packet, does not transition the “Input Error-stopped” state, and accepts new packets normally. The reporting mechanism for such an event is implementation dependent. Note that care must be taken when using this feature as it is possible that the CRF and/or the priority bits themselves could be corrupt.

5.11.2.5 Link Time-Out

A link time-out while waiting for an acknowledgment control symbol is handled as a link protocol violation as described in Section 5.11.2.3.1, “Link Protocol Violations

5.11.2.6 Input Error-Stopped Recovery Process

When the input side of a port detects a transmission error, it immediately enters the Input Error-stopped state. To recover from this state, the input side of the port takes the following actions.

- Record the error(s) that caused the port to enter the Input Error-stopped state.
- If the detected error(s) occurred in a control symbol or packet, discard the control symbol or packet.
- Ignore all subsequently received packets while the port is in the Input Error-stopped state.

- Cause the output side of the port to issue a packet-not-accepted control symbol. The packet_ackID field of the control symbol contains an undefined value. (The packet-not-accepted control symbol causes the output side of the receiving port to enter the Output Error-stopped state and send a link-request/input-status control symbol.)
- When an link-request/input-status control symbol is received, cause the output side of the port to issue a link-response control symbol, exit the Input Error-stopped state and resume packet reception.

An example state machine with the behavior described in this section is included in Section A.3, “Error Recovery”.

5.11.2.7 Output Error-Stopped Recovery Process

To recover from the Output Error-stopped state, the output side of a port takes the following actions.

- Immediately stops transmitting new packets.
- Resets the link packet acknowledgment timers for all transmitted but unacknowledged packets. (This prevents the generation of spurious time-out errors.)
- Transmits an input-status link-request/input-status (restart-from-error) control symbol. (The input status link-request/input-status control symbol causes the receiving port to transmit a link-response control symbol that contains the input_status and ackID_status of the input side of the port. The ackID_status is the ackID value that is expected in the next packet that the port receives.)
- When the link-response is received, the port backs up the first unaccepted packet, exits the Output Error-stopped state and resumes transmission with either the first unaccepted packet or a higher priority packet.

An example state machine with the behavior described in this section is included in Section A.3, “Error Recovery”.

5.12 Power Management

Power management is currently beyond the scope of this specification and is implementation dependent. A device that supports power management features can make these features accessible to the rest of the system using the device’s local configuration registers.

Chapter 6 LP-Serial Registers

6.1 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 physical layer 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.

There are four types of 1x/4x LP-Serial devices, an end point device, an end point device with additional software recovery registers, an end point free (or switch) device, and an end point free device with additional software recovery registers. Each has a different set of CSRs, specified in Section 6.5, Section 6.6, Section 6.7, and Section 6.8, respectively. All four device types have the same CARs, specified in Section 6.4.

6.2 Register Map

These registers utilize the Extended Features blocks and can be accessed using *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.

The Extended Features pointer (EF_PTR) defined in the RapidIO logical specifications contains the offset of the first Extended Features block in the Extended Features data structure for a device. The 1x/4x LP-Serial physical features block shall exist in any position in the Extended Features data structure and shall exist in any portion of the Extended Features Space in the register address map for the 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 6-1. 1x/4x LP-Serial Register Map

Configuration Space Byte Offset	Register Name
0x0-C	Reserved
0x10	Processing Element Features CAR
0x14–FC	Reserved
0x100–FFFC	Extended Features Space
0x10000–FFFFFC	Implementation-defined Space

6.3 Reserved Register and Bit Behavior

Table 6-2 describes the required behavior for accesses to reserved register bits and reserved registers for the RapidIO register space,

Table 6-2. 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 6-2. 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.

6.4 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 6-2 for the required behavior for accesses to reserved registers and register bits.

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

6.4.1 Processing Element Features CAR (Configuration Space Offset 0x10)

The processing element features CAR identifies the major functionality provided by the processing element. The bit settings are shown in Table 6-3.

Table 6-3. Bit Settings for Processing Element Features CAR

Bits	Name	Description
0–24	—	Reserved
25	Re-transmit Suppression Support	PE supports suppression of error recovery on packet CRC errors 0b0 - The error recovery suppression option is not supported by the PE 0b1 - The error recovery suppression option is supported by the PE
26	CRF Support	PE supports the Critical Request Flow (CRF) indicator 0b0 - Critical Request Flow is not supported 0b1 - Critical Request Flow is supported
27–31	—	Reserved

6.5 Generic End Point Devices

This section describes the 1x/4x LP-Serial registers for a general end point device. This Extended Features register block is assigned Extended Features block ID=0x0001.

6.5.1 Register Map

Table 6-4 shows the register map for generic RapidIO 1x/4x LP-Serial end point devices. The Block Offset is the offset relative to the 16-bit Extended Features Pointer (EF_PTR) that points to the beginning of the block.

The address of a byte in the block is calculated by adding the block byte offset to EF_PTR that points to the beginning of the block. This is denoted as [EF_PTR+xx] where xx is the block byte offset in hexadecimal.

This register map is currently only defined for devices with up to 16 RapidIO ports, but can be extended or shortened if more or less port definitions are required for a device. For example, a device with four RapidIO ports is only required to use register map space corresponding to offsets [EF_PTR + 0x00] through [EF_PTR + 0xBC]. Register map offset [EF_PTR + 0xC0] can be used for another Extended Features block.

Table 6-4. LP-Serial Register Map - Generic End Point Devices

	Block Byte Offset	Register Name
General	0x0	1x/4x LP-Serial Register Block Header
	0x4–1C	Reserved
	0x20	Port Link Time-Out Control CSR
	0x24	Port Response Time-Out Control CSR
	0x28-38	Reserved
	0x3C	Port General Control CSR
Port 0	0x40-54	Reserved
	0x58	Port 0 Error and Status CSR
	0x5C	Port 0 Control CSR
Port 1	0x60-74	Reserved
	0x78	Port 1 Error and Status CSR
	0x7C	Port 1 Control CSR
Ports 2-14	0x80–218	Assigned to Port 2-14 CSRs

Table 6-4. LP-Serial Register Map (Continued) - Generic End Point Devices

	Block Byte Offset	Register Name
Port 15	0x220-234	Reserved
	0x238	Port 15 Error and Status CSR
	0x23C	Port 15 Control CSR

6.5.2 Command and Status Registers (CSRs)

Refer to Table 6-2 for the required behavior for accesses to reserved registers and register bits.

6.5.2.1 1x/4x LP-Serial Register Block Header (Block Offset 0x0)

The 1x/4x LP-Serial register block header register contains the EF_PTR to the next extended features block and the EF_ID that identifies this as the generic end point 1x/4x LP-Serial register block header.

Table 6-5. Bit Settings for 1x/4x LP-Serial Register Block Header

Bit	Name	Reset Value	Description
0-15	EF_PTR		Hard wired pointer to the next block in the data structure, if one exists
16-31	EF_ID	0x0001	Hard wired Extended Features ID

6.5.2.2 Port Link Time-out Control CSR (Block Offset 0x20)

The port link time-out control register contains the time-out timer value for all ports on a device. This time-out is for link events such as sending a packet to receiving the corresponding acknowledge, and sending a link-request to receiving the corresponding link-response. The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-6. Bit Settings for Port Link Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.5.2.3 Port Response Time-out Control CSR (Block Offset 0x24)

The port response time-out control register contains the time-out timer count for all ports on a device. This time-out is for sending a request packet to receiving the corresponding response packet.The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-7. Bit Settings for Port Response Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.5.2.4 Port General Control CSR (Block Offset 0x3C)

The bits accessible through the Port General Control CSR are bits that apply to all ports on a device. There is a single copy of each such bit per device. These bits are also accessible through the Port General Control CSR of any other physical layers implemented on a device.

Table 6-8. Bit Settings for Port General Control CSRs

Bit	Name	Reset Value	Description
0	Host	see footnote[1]	A Host device is a device that is responsible for system exploration, initialization, and maintenance. Agent or slave devices are typically initialized by Host devices. 0b0 - agent or slave device 0b1 - host device
1	Master Enable	see footnote[2]	The Master Enable bit controls whether or not a device is allowed to issue requests into the system. If the Master Enable is not set, the device may only respond to requests. 0b0 - processing element cannot issue requests 0b1 - processing element can issue requests
2	Discovered	see footnote[3]	This device has been located by the processing element responsible for system configuration 0b0 - The device has not been previously discovered 0b1 - The device has been discovered by another processing element
3-31	—		Reserved

[1]The Host reset value is implementation dependent

[2]The Master Enable reset value is implementation dependent

[3]The Discovered reset value is implementation dependent

6.5.2.5 Port *n* Error and Status CSRs (Block Offsets 0x58, 78, ..., 238)

These registers are accessed when a local processor or an external device wishes to examine the port error and status information.

Table 6-9. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
0-10	—		Reserved
11	Output Retry-encountered	0b0	Output port has encountered a retry condition.This bit is set when bit 13 is set. Once set, remains set until written with a logic 1 to clear.
12	Output Retried	0b0	Output port has received a packet-retry control symbol and can not make forward progress. This bit is set when bit 13 is set and is cleared when a packet-accepted or a packet-not-accepted control symbol is received (read-only).
13	Output Retry-stopped	0b0	Output port has received a packet-retry control symbol and is in the “output retry-stopped” state (read-only).
14	Output Error-encountered	0b0	Output port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 15 is set. Once set, remains set until written with a logic 1 to clear.
15	Output Error-stopped	0b0	Output is in the “output error-stopped” state (read-only).
16-20	—		Reserved
21	Input Retry-stopped	0b0	Input port is in the “input retry-stopped” state (read-only).
22	Input Error-encountered	0b0	Input port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 23 is set. Once set, remains set until written with a logic 1 to clear.
23	Input Error-stopped	0b0	Input port is in the “input error-stopped” state (read-only).
24-26	—		Reserved
27	Port-write Pending	0b0	Port has encountered a condition which required it to initiate a Maintenance Port-write operation This bit is only valid if the device is capable of issuing a maintenance port-write transaction. Once set remains set until written with a logic 1 to clear.
28	—		Reserved
29	Port Error	0b0	Input or output port has encountered an error from which hardware was unable to recover. Once set, remains set until written with a logic 1 to clear.
30	Port OK	0b0	The input and output ports are initialized and the port is exchanging error-free control symbols with the attached device (read-only).
31	Port Uninitialized	0b1	Input and output ports are not initialized. This bit and bit 30 are mutually exclusive (read-only).

6.5.2.6 Port *n* Control CSR (Block Offsets 0x5C, 7C, ..., 23C)

The port *n* control registers contain control register bits for individual ports on a processing element.

Table 6-10. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
0-1	Port Width	see footnote[1]	Hardware width of the port (read-only): 0b00 - Single-lane port 0b01 - Four-lane port 0b10 - 0b11 - Reserved
2-4	Initialized Port Width	see footnote[2]	Width of the ports after initialized (read only): 0b000 - Single-lane port, lane 0 0b001 - Single-lane port, lane 2 0b010 - Four-lane port 0b011 - 0b111 - Reserved
5-7	Port Width Override	0b000	Soft port configuration to override the hardware size: 0b000 - No override 0b001 - Reserved 0b010 - Force single lane, lane 0 0b011 - Force single lane, lane 2 0b100 - 0b111 - Reserved If the port size is overridden, the port will re-initialize to change to the requested size.
8	Port Disable	0b0	Port disable: 0b0 - port receivers/drivers are enabled 0b1 - port receivers/drivers are disabled and are unable to receive/transmit any packets or control symbols
9	Output Port Enable	see footnote[3]	Output port transmit enable: 0b0 - port is stopped and not enabled to issue any packets except to route or respond to I/O logical MAINTENANCE packets. Control symbols are not affected and are sent normally. This is the recommended state after device reset. 0b1 - port is enabled to issue any packets
10	Input Port Enable	see footnote[4]	Input port receive enable: 0b0 - port is stopped and only enabled to route or respond I/O logical MAINTENANCE packets. Other packets generate packet-not-accepted control symbols to force an error condition to be signaled by the sending device. Control symbols are not affected and are received and handled normally. This is the recommended state after device reset. 0b1 - port is enabled to respond to any packet
11	Error Checking Disable	0b0	This bit disables all RapidIO transmission error checking 0b0 - Error checking and recovery is enabled 0b1 - Error checking and recovery is disabled Device behavior when error checking and recovery is disabled and an error condition occurs is undefined
12	Multicast-event Participant	see footnote[5]	Send incoming Multicast-event control symbols to this port (multiple port devices only)
13	—		Reserved

Table 6-10. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
14	Enumeration Boundary	see footnote[6]	An enumeration boundary aware system enumeration algorithm shall honor this flag. The algorithm, on either the ingress or the egress port, shall not enumerate past a port with this bit set. This provides for software enforced enumeration domains within the RapidIO fabric.
15-19	—		Reserved
20-27	Re-transmit Suppression Mask	0x00	Suppress packet re-transmission on CRC error. For devices that support CRF: 0b0000_0000 - Error recovery suppression disabled 0bxxxx_xxx1 - Suppress CRF=0, priority 0 re-transmission 0bxxxx_xx1x - Suppress CRF=0, priority 1 re-transmission 0bxxxx_x1xx - Suppress CRF=0, priority 2 re-transmission 0bxxxx_1xxx - Suppress CRF=0, priority 3 re-transmission 0bxxx1_xxxx - Suppress CRF=1, priority 0 re-transmission 0bxx1x_xxxx - Suppress CRF=1, priority 1 re-transmission 0bx1xx_xxxx - Suppress CRF=1, priority 2 re-transmission 0b1xxx_xxxx - Suppress CRF=1, priority 3 re-transmission For devices that do not support CRF: 0b0000_0000 - Error recovery suppression disabled 0b0000_xxx1 - Suppress priority 0 re-transmission 0b0000_xx1x - Suppress priority 1 re-transmission 0b0000_x1xx - Suppress priority 2 re-transmission 0b0000_1xxx - Suppress priority 3 re-transmission 0b0001_0000 - reserved ... 0b1111_1111 - reserved This field is only valid if bit 25 of the Processing Element Features CAR is set.
28-30	—		Reserved
31	Port Type		This indicates the port type (read only) 0b0 - Reserved 0b1 - Serial port

[1]The Port Width reset value is implementation dependent

[2]The Initialized Port Width reset value is implementation dependent

[3]The Output Port Enable reset value is implementation dependent

[4]The Input Port Enable reset value is implementation dependent

[5]The Multicast-event Participant reset value is implementation dependent

[6]The Enumeration Boundary reset value is implementation dependent. Provision shall be made to allow the reset value of this bit to be configurable on a per system basis if this feature is supported.

6.6 Generic End Point Devices, software assisted error recovery option

This section describes the 1x/4x LP-Serial registers for a general end point device that supports software assisted error recovery. This is most useful for devices that for whatever reason do not want to implement error recovery in hardware and to allow software to generate link-request control symbols and see the results of the responses. This Extended Features register block is assigned Extended Features block ID=0x0002.

6.6.1 Register Map

Table 6-11 shows the register map for generic RapidIO 1x/4x LP-Serial end point devices with software assisted error recovery. The Block Offset is the offset based on the Extended Features pointer (EF_PTR) to this block. This register map is currently only defined for devices with up to 16 RapidIO ports, but can be extended or shortened if more or less port definitions are required for a device. For example, a device with four RapidIO ports is only required to use register map space corresponding to offsets [EF_PTR + 0x00] through [EF_PTR + 0xBC]. Register map offset [EF_PTR + 0xC0] can be used for another Extended Features block.

Table 6-11. LP-Serial Register Map - Generic End Point Devices (SW assisted)

	Block Byte Offset	Register Name
General	0x0	1x/4x LP-Serial Register Block Header
	0x4–1C	Reserved
	0x20	Port Link Time-Out Control CSR
	0x24	Port Response Time-Out Control CSR
	0x28-38	Reserved
	0x3C	Port General Control CSR
Port 0	0x40	Port 0 Link Maintenance Request CSR
	0x44	Port 0 Link Maintenance Response CSR
	0x48	Port 0 Local ackID Status CSR
	0x4C-54	Reserved
	0x58	Port 0 Error and Status CSR
	0x5C	Port 0 Control CSR
Port 1	0x60	Port 1 Link Maintenance Request CSR
	0x64	Port 1 Link Maintenance Response CSR
	0x68	Port 1 Local ackID Status CSR
	0x6C-74	Reserved
	0x78	Port 1 Error and Status CSR
	0x7C	Port 1 Control CSR

Table 6-11. LP-Serial Register Map (Continued)- Generic End Point Devices (SW assisted)

	Block Byte Offset	Register Name
Ports 2-14	0x80–218	Assigned to Port 2-14 CSRs
Port 15	0x220	Port 15 Link Maintenance Request CSR
	0x224	Port 15 Link Maintenance Response CSR
	0x228	Port 15 Local ackID Status CSR
	0x22C-234	Reserved
	0x238	Port 15 Error and Status CSR
	0x23C	Port 15 Control CSR

6.6.2 Command and Status Registers (CSRs)

Refer to Table 6-2 for the required behavior for accesses to reserved registers and register bits.

6.6.2.1 1x/4x LP-Serial Register Block Header (Block Offset 0x0)

The 1x/4x LP-Serial register block header register contains the EF_PTR to the next extended features block and the EF_ID that identifies this as the generic end point 1x/4x LP-Serial register block header.

Table 6-12. Bit Settings for 1x/4x LP-Serial Register Block Header

Bit	Name	Reset Value	Description
0-15	EF_PTR		Hard wired pointer to the next block in the data structure, if one exists
16-31	EF_ID	0x0002	Hard wired Extended Features ID

6.6.2.2 Port Link Time-out Control CSR (Block Offset 0x20)

The port link time-out control register contains the time-out timer value for all ports on a device. This time-out is for link events such as sending a packet to receiving the corresponding acknowledge and sending a link-request to receiving the corresponding link-response. The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-13. Bit Settings for Port Link Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.6.2.3 Port Response Time-out Control CSR (Block Offset 0x24)

The port response time-out control register contains the time-out timer count for all ports on a device. This time-out is for sending a request packet to receiving the corresponding response packet.The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-14. Bit Settings for Port Response Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.6.2.4 Port General Control CSR (Block Offset 0x3C)

The port general control register contains control register bits applicable to all ports on a processing element.

Table 6-15. Bit Settings for Port General Control CSRs

Bit	Name	Reset Value	Description
0	Host	see footnote[1]	A Host device is a device that is responsible for system exploration, initialization, and maintenance. Agent or slave devices are initialized by Host devices. 0b0 - agent or slave device 0b1 - host device
1	Master Enable	see footnote[2]	The Master Enable bit controls whether or not a device is allowed to issue requests into the system. If the Master Enable is not set, the device may only respond to requests. 0b0 - processing element cannot issue requests 0b1 - processing element can issue requests
2	Discovered	see footnote[3]	This device has been located by the processing element responsible for system configuration 0b0 - The device has not been previously discovered 0b1 - The device has been discovered by another processing element
3-31	—		Reserved

[1]The Host reset value is implementation dependent

[2]The Master Enable reset value is implementation dependent

[3]The Discovered reset value is implementation dependent

6.6.2.5 Port *n* Link Maintenance Request CSRs (Block Offsets 0x40, 60, ..., 220)

The port link maintenance request registers are accessible both by a local processor and an external device. A write to one of these registers generates a link-request control symbol on the corresponding RapidIO port interface.

Table 6-16. Bit Settings for Port *n* Link Maintenance Request CSRs

Bit	Name	Reset Value	Description
0–28	—		Reserved
29-31	Command	0b000	Command to be sent in the link-request control symbol. If read, this field returns the last written value.

6.6.2.6 Port *n* Link Maintenance Response CSRs (Block Offsets 0x44, 64, ..., 224)

The port link maintenance response registers are accessible both by a local processor and an external device. A read to this register returns the status received in a link-response control symbol. The ackID_status and port_status fields are defined in Table 3-3 and Table 3-5. This register is read-only.

Table 6-17. Bit Settings for Port *n* Link Maintenance Response CSRs

Bit	Name	Reset Value	Description
0	response_valid	0b0	If the link-request causes a link-response, this bit indicates that the link-response has been received and the status fields are valid. If the link-request does not cause a link-response, this bit indicates that the link-request has been transmitted. This bit automatically clears on read.
1-21	—		Reserved
22-26	ackID_status	0b00000	ackID status field from the link-response control symbol
27-31	link_status	0b00000	link status field from the link-response control symbol

6.6.2.7 Port *n* Local ackID CSRs (Block Offsets 0x48, 68, ..., 228)

The port link local ackID status registers are accessible both by a local processor and an external device. A read to this register returns the local ackID status for both the out and input ports of the device.

Table 6-18. Bit Settings for Port *n* Local ackID Status CSRs

Bit	Name	Reset Value	Description
0	Clr_outstanding_ackIDs	0b0	Writing 0b1 to this bit causes all outstanding unacknowleged packets to be discarded. This bit should only be written when trying to recover a failed link. This bit is always logic 0 when read.
1-2	—		Reserved
3-7	Inbound_ackID	0b00000	Input port next expected ackID value

Table 6-18. Bit Settings for Port *n* Local ackID Status CSRs (Continued)

Bit	Name	Reset Value	Description
8-18	—		Reserved
19-23	Outstanding_ackID	0x00000	Output port unacknowledged ackID status. Next expected acknowledge control symbol ackID field that indicates the ackID value expected in the next received acknowledge control symbol.
24-26	—		Reserved
27-31	Outbound_ackID	0b00000	Output port next transmitted ackID value. Software writing this value can force retransmission of outstanding unacknowledged packets in order to manually implement error recovery.

6.6.2.8 Port *n* Error and Status CSRs (Block Offset 0x58, 78, ..., 238)

These registers are accessed when a local processor or an external device wishes to examine the port error and status information.

Table 6-19. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
0-10	—		Reserved
11	Output Retry-encountered	0b0	Output port has encountered a retry condition.This bit is set when bit 13 is set. Once set, remains set until written with a logic 1 to clear.
12	Output Retried	0b0	Output port has received a packet-retry control symbol and can not make forward progress. This bit is set when bit 13 is set and is cleared when a packet-accepted or a packet-not-accepted control symbol is received (read-only).
13	Output Retry-stopped	0b0	Output port has received a packet-retry control symbol and is in the “output retry-stopped” state (read-only).
14	Output Error-encountered	0b0	Output port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 15 is set. Once set, remains set until written with a logic 1 to clear.
15	Output Error-stopped	0b0	Output is in the “output error-stopped” state (read-only).
16-20	—		Reserved
21	Input Retry-stopped	0b0	Input port is in the “input retry-stopped” state (read-only).
22	Input Error-encountered	0b0	Input port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 23 is set. Once set, remains set until written with a logic 1 to clear.
23	Input Error-stopped	0b0	Input port is in the “input error-stopped” state (read-only).
24-26	—		Reserved
27	Port-write Pending	0b0	Port has encountered a condition which required it to initiate a Maintenance Port-write operation This bit is only valid if the device is capable of issuing a maintenance port-write transaction. Once set remains set until written with a logic 1 to clear.
28	—		Reserved

Table 6-19. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
29	Port Error	0b0	Input or output port has encountered an error from which hardware was unable to recover. Once set, remains set until written with a logic 1 to clear.
30	Port OK	0b0	The input and output ports are initialized and the port is exchanging error-free control symbols with the attached device (read-only).
31	Port Uninitialized	0b1	Input and output ports are not initialized. This bit and bit 30 are mutually exclusive (read-only).

6.6.2.9 Port *n* Control CSR (Block Offsets 0x5C, 7C, ..., 23C)

The port *n* control registers contain control register bits for individual ports on a processing element.

Table 6-20. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
0-1	Port Width	see footnote[1]	Hardware width of the port (read-only): 0b00 - Single-lane port 0b01 - Four-lane port 0b10 - 0b11 - Reserved
2-4	Initialized Port Width	see footnote[2]	Width of the ports after initialized (read only): 0b000 - Single-lane port, lane 0 0b001 - Single-lane port, lane 2 0b010 - Four-lane port 0b011 - 0b111 - Reserved
5-7	Port Width Override	0b000	Soft port configuration to override the hardware size: 0b000 - No override 0b001 - Reserved 0b010 - Force single lane, lane 0 0b011 - Force single lane, lane 2 0b100 - 0b111 - Reserved If the port size is overridden, the port will re-initialize to change to the requested size.
8	Port Disable	0b0	Port disable: 0b0 - port receivers/drivers are enabled 0b1 - port receivers/drivers are disabled and are unable to receive/transmit any packets or control symbols
9	Output Port Enable	see footnote[3]	Output port transmit enable: 0b0 - port is stopped and not enabled to issue any packets except to route or respond to I/O logical MAINTENANCE packets. Control symbols are not affected and are sent normally. This is the recommended state after device reset. 0b1 - port is enabled to issue any packets

Bit	Name	Reset Value	Description
10	Input Port Enable	see footnote[4]	Input port receive enable: 0b0 - port is stopped and only enabled to route or respond I/O logical MAINTENANCE packets. Other packets generate packet-not-accepted control symbols to force an error condition to be signaled by the sending device. Control symbols are not affected and are received and handled normally. This is the recommended state after device reset. 0b1 - port is enabled to respond to any packet
11	Error Checking Disable	0b0	This bit disables all RapidIO transmission error checking 0b0 - Error checking and recovery is enabled 0b1 - Error checking and recovery is disabled Device behavior when error checking and recovery is disabled and an error condition occurs is undefined
12	Multicast-event Participant	see footnote[5]	Send incoming Multicast-event control symbols to this port (multiple port devices only)
13	—		Reserved
14	Enumeration Boundary	see footnote[6]	An enumeration boundary aware system enumeration algorithm shall honor this flag. The algorithm, on either the ingress or the egress port, shall not enumerate past a port with this bit set. This provides for software enforced enumeration domains within the RapidIO fabric.
15-19	—		Reserved
20-27	Re-transmit Suppression Mask	0x00	Suppress packet re-transmission on CRC error. For devices that support CRF: 0b0000_0000 - Error recovery suppression disabled 0bxxxx_xxx1 - Suppress CRF=0, priority 0 re-transmission 0bxxxx_xx1x - Suppress CRF=0, priority 1 re-transmission 0bxxxx_x1xx - Suppress CRF=0, priority 2 re-transmission 0bxxxx_1xxx - Suppress CRF=0, priority 3 re-transmission 0bxxx1_xxxx - Suppress CRF=1, priority 0 re-transmission 0bxx1x_xxxx - Suppress CRF=1, priority 1 re-transmission 0bx1xx_xxxx - Suppress CRF=1, priority 2 re-transmission 0b1xxx_xxxx - Suppress CRF=1, priority 3 re-transmission For devices that do not support CRF: 0b0000_0000 - Error recovery suppression disabled 0b0000_xxx1 - Suppress priority 0 re-transmission 0b0000_xx1x - Suppress priority 1 re-transmission 0b0000_x1xx - Suppress priority 2 re-transmission 0b0000_1xxx - Suppress priority 3 re-transmission 0b0001_0000 - reserved ... 0b1111_1111 - reserved This field is only valid if bit 25 of the Processing Element Features CAR is set.
28-30	—		Reserved
31	Port Type		This indicates the port type (read only) 0b0 - Reserved 0b1 - Serial port

[1]The Port Width reset value is implementation dependent

[2]The Initialized Port Width reset value is implementation dependent

[3]The Output Port Enable reset value is implementation dependent

[4]The Input Port Enable reset value is implementation dependent

[5]The Multicast-Event Participant reset value is implementation dependent

[6]The Enumeration Boundary reset value is implementation dependent. Provision shall be made to allow the reset value of this bit to be configurable on a per system basis if this feature is supported.

6.7 Generic End Point Free Devices

This section describes the 1x/4x LP-Serial registers for a general devices that do not contain end point functionality (i.e. switches). This Extended Features register block uses extended features block ID=0x0003.

6.7.1 Register Map

Table 6-21 shows the register map for generic RapidIO 1x/4x LP-Serial end point-free devices. The Block Offset is the offset based on the Extended Features pointer (EF_PTR) to this block. This register map is currently only defined for devices with up to 16 RapidIO ports, but can be extended or shortened if more or less port definitions are required for a device. For example, a device with four RapidIO ports is only required to use register map space corresponding to offsets [EF_PTR + 0x00] through [EF_PTR + 0xBC]. Register map offset [EF_PTR + 0xC0] can be used for another Extended Features block.

Table 6-21. LP-Serial Register Map - Generic End Point Free Devices

	Block Byte Offset	Register Name
General	0x0	1x/4x LP-Serial Register Block Header
	0x4–1C	Reserved
	0x20	Port Link Time-Out Control CSR
	0x24-38	Reserved
	0x3C	Port General Control CSR
Port 0	0x40-54	Reserved
	0x58	Port 0 Error and Status CSR
	0x5C	Port 0 Control CSR
Port 1	0x60-74	Reserved
	0x78	Port 1 Error and Status CSR
	0x7C	Port 1 Control CSR
Ports 2-14	0x80–218	Assigned to Port 2-14 CSRs
Port 15	0x220-234	Reserved
	0x238	Port 15 Error and Status CSR
	0x23C	Port 15 Control CSR

6.7.2 Command and Status Registers (CSRs)

Refer to Table 6-2 for the required behavior for accesses to reserved registers and register bits.

6.7.2.1 1x/4x LP-Serial Register Block Header (Block Offset 0x0)

The 1x/4x LP-Serial register block header register contains the EF_PTR to the next extended features block and the EF_ID that identifies this as the generic end point 1x/4x LP-Serial register block header.

Table 6-22. Bit Settings for 1x/4x LP-Serial Register Block Header

Bit	Name	Reset Value	Description
0-15	EF_PTR		Hard wired pointer to the next block in the data structure, if one exists
16-31	EF_ID	0x0003	Hard wired Extended Features ID

6.7.2.2 Port Link Time-out Control CSR (Block Offset 0x20)

The port link time-out control register contains the time-out timer value for all ports on a device. This time-out is for link events such as sending a packet to receiving the corresponding acknowledge and sending a link-request to receiving the corresponding link-response. The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-23. Bit Settings for Port Link Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.7.2.3 Port General Control CSR (Block Offset 0x3C)

The port general control register contains control register bits applicable to all ports on a processing element.

Table 6-24. Bit Settings for Port General Control CSRs

Bit	Name	Reset Value	Description
0-1	—		Reserved
2	Discovered	0b0	This device has been located by the processing element responsible for system configuration 0b0 - The device has not been previously discovered 0b1 - The device has been discovered by another processing element
3-31	—		Reserved

6.7.2.4 Port *n* Error and Status CSRs (Block Offsets 0x58, 78, .., 238)

These registers are accessed when a local processor or an external device wishes to examine the port error and status information.

Table 6-25. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
0-10	—		Reserved
11	Output Retry-encountered	0b0	Output port has encountered a retry condition.This bit is set when bit 13 is set. Once set, remains set until written with a logic 1 to clear.
12	Output Retried	0b0	Output port has received a packet-retry control symbol and can not make forward progress. This bit is set when bit 13 is set and is cleared when a packet-accepted or a packet-not-accepted control symbol is received (read-only).
13	Output Retry-stopped	0b0	Output port has received a packet-retry control symbol and is in the “output retry-stopped” state (read-only).
14	Output Error-encountered	0b0	Output port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 15 is set. Once set, remains set until written with a logic 1 to clear.
15	Output Error-stopped	0b0	Output is in the “output error-stopped” state (read-only).
16-20	—		Reserved
21	Input Retry-stopped	0b0	Input port is in the “input retry-stopped” state (read-only).
22	Input Error-encountered	0b0	Input port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 23 is set. Once set, remains set until written with a logic 1 to clear.
23	Input Error-stopped	0b0	Input port is in the “input error-stopped” state (read-only).
24-26	—		Reserved

Table 6-25. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
27	Port-write Pending	0b0	Port has encountered a condition which required it to initiate a Maintenance Port-write operation This bit is only valid if the device is capable of issuing a maintenance port-write transaction. Once set remains set until written with a logic 1 to clear.
28	—		Reserved
29	Port Error	0b0	Input or output port has encountered an error from which hardware was unable to recover. Once set, remains set until written with a logic 1 to clear.
30	Port OK	0b0	The input and output ports are initialized and the port is exchanging error-free control symbols with the attached device (read-only).
31	Port Uninitialized	0b1	Input and output ports are not initialized. This bit and bit 30 are mutually exclusive (read-only).

6.7.2.5 Port *n* Control CSR (Block Offsets 0x5C, 7C, ..., 23C)

The port *n* control registers contain control register bits for individual ports on a processing element.

Table 6-26. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
0-1	Port Width	see footnote[1]	Hardware width of the port (read-only): 0b00 - Single-lane port 0b01 - Four-lane port 0b10 - 0b11 - Reserved
2-4	Initialized Port Width	see footnote[2]	Width of the ports after initialized (read only): 0b000 - Single-lane port, lane 0 0b001 - Single-lane port, lane 2 0b010 - Four-lane port 0b011 - 0b111 - Reserved
5-7	Port Width Override	0b000	Soft port configuration to override the hardware size: 0b000 - No override 0b001 - Reserved 0b010 - Force single lane, lane 0 0b011 - Force single lane, lane 2 0b100 - 0b111 - Reserved If the port size is overridden, the port will re-initialize to change to the requested size.
8	Port Disable	0b0	Port disable: 0b0 - port receivers/drivers are enabled 0b1 - port receivers/drivers are disabled and are unable to receive/transmit any packets or control symbols
9	Output Port Enable	see footnote[3]	Output port transmit enable: 0b0 - port is stopped and not enabled to issue any packets except to route or respond to I/O logical MAINTENANCE packets. Control symbols are not affected and are sent normally. This is the recommended state after device reset. 0b1 - port is enabled to issue any packets

Bit	Name	Reset Value	Description
10	Input Port Enable	see footnote[4]	Input port receive enable: 0b0 - port is stopped and only enabled to route or respond I/O logical MAINTENANCE packets. Other packets generate packet-not-accepted control symbols to force an error condition to be signaled by the sending device. Control symbols are not affected and are received and handled normally. This is the recommended state after device reset. 0b1 - port is enabled to respond to any packet
11	Error Checking Disable	0b0	This bit disables all RapidIO transmission error checking 0b0 - Error checking and recovery is enabled 0b1 - Error checking and recovery is disabled Device behavior when error checking and recovery is disabled and an error condition occurs is undefined
12	Multicast Event Participant	see footnote[5]	Send incoming Multicast-event control symbols to this port (multiple port devices only)
13	—		Reserved
14	Enumeration Boundary	see footnote[6]	An enumeration boundary aware system enumeration algorithm shall honor this flag. The algorithm, on either the ingress or the egress port, shall not enumerate past a port with this bit set. This provides for software enforced enumeration domains within the RapidIO fabric.
15-19	—		Reserved
20-27	Re-transmit Suppression Mask	0x00	Suppress packet re-transmission on CRC error. For devices that support CRF: 0b0000_0000 - Error recovery suppression disabled 0bxxxx_xxx1 - Suppress CRF=0, priority 0 re-transmission 0bxxxx_xx1x - Suppress CRF=0, priority 1 re-transmission 0bxxxx_x1xx - Suppress CRF=0, priority 2 re-transmission 0bxxxx_1xxx - Suppress CRF=0, priority 3 re-transmission 0bxxx1_xxxx - Suppress CRF=1, priority 0 re-transmission 0bxx1x_xxxx - Suppress CRF=1, priority 1 re-transmission 0bx1xx_xxxx - Suppress CRF=1, priority 2 re-transmission 0b1xxx_xxxx - Suppress CRF=1, priority 3 re-transmission For devices that do not support CRF: 0b0000_0000 - Error recovery suppression disabled 0b0000_xxx1 - Suppress priority 0 re-transmission 0b0000_xx1x - Suppress priority 1 re-transmission 0b0000_x1xx - Suppress priority 2 re-transmission 0b0000_1xxx - Suppress priority 3 re-transmission 0b0001_0000 - reserved ... 0b1111_1111 - reserved This field is only valid if bit 25 of the Processing Element Features CAR is set.
28-30	—		Reserved
31	Port Type		This indicates the port type (read only) 0b0 - Reserved 0b1 - Serial port

[1]The Port Width reset value is implementation dependent

[2]The Initialized Port Width reset value is implementation dependent

[3]The Output Port Enable reset value is implementation dependent

[4]The Input Port Enable reset value is implementation dependent

[5]The Multicast-event Participant reset value is implementation dependent

[6]The Enumeration Boundary reset value is implementation dependent. Provision shall be made to allow the reset value of this bit to be configurable on a per system basis if this feature is supported.

6.8 Generic End Point Free Devices, software assisted error recovery option

This section describes the 1x/4x LP-Serial registers for a general device that does not contain end point functionality with software assisted error recovery.Typically these devices are switches. This is most useful for devices that for whatever reason do not want to implement error recovery in hardware and to allow software to generate link-request control symbols and see the results of the responses. This Extended Features register block is assigned Extended Features block ID=0x0009.

6.8.1 Register Map

Table 6-27 shows the register map for generic RapidIO 1x/4x LP-Serial end point-free devices with software assisted error recovery. The Block Offset is the offset based on the Extended Features pointer (EF_PTR) to this block. This register map is currently only defined for devices with up to 16 RapidIO ports, but can be extended or shortened if more or less port definitions are required for a device. For example, a device with four RapidIO ports is only required to use register map space corresponding to offsets [EF_PTR + 0x00] through [EF_PTR + 0xBC]. Register map offset [EF_PTR + 0xC0] can be used for another Extended Features block.

Table 6-27. LP-Serial Register Map - Generic End Point-free Devices (SW assisted)

	Block Byte Offset	Register Name
General	0x0	1x/4x LP-Serial Register Block Header
	0x4–1C	Reserved
	0x20	Port Link Time-Out Control CSR
	0x24-38	Reserved
	0x3C	Port General Control CSR
Port 0	0x40	Port 0 Link Maintenance Request CSR
	0x44	Port 0 Link Maintenance Response CSR
	0x48	Port 0 Local ackID Status CSR
	0x4C-54	Reserved
	0x58	Port 0 Error and Status CSR
	0x5C	Port 0 Control CSR
Port 1	0x60	Port 1 Link Maintenance Request CSR
	0x64	Port 1 Link Maintenance Response CSR
	0x68	Port 1 Local ackID Status CSR
	0x6C-74	Reserved
	0x78	Port 1 Error and Status CSR
	0x7C	Port 1 Control CSR

Table 6-27. LP-Serial Register Map (Continued)- Generic End Point-free Devices (SW assisted)

	Block Byte Offset	Register Name
Ports 2-14	0x80–218	Assigned to Port 2-14 CSRs
Port 15	0x220	Port 15 Link Maintenance Request CSR
	0x224	Port 15 Link Maintenance Response CSR
	0x228	Port 15 Local ackID Status CSR
	0x22C-234	Reserved
	0x238	Port 15 Error and Status CSR
	0x23C	Port 15 Control CSR

6.8.2 Command and Status Registers (CSRs)

Refer to Table 6-2 for the required behavior for accesses to reserved registers and register bits.

6.8.2.1 1x/4x LP-Serial Register Block Header (Block Offset 0x0)

The 1x/4x LP-Serial register block header register contains the EF_PTR to the next extended features block and the EF_ID that identifies this as the generic end point 1x/4x LP-Serial register block header.

Table 6-28. Bit Settings for 1x/4x LP-Serial Register Block Header

Bit	Name	Reset Value	Description
0-15	EF_PTR		Hard wired pointer to the next block in the data structure, if one exists
16-31	EF_ID	0x0009	Hard wired Extended Features ID

6.8.2.2 Port Link Time-out Control CSR (Block Offset 0x20)

The port link time-out control register contains the time-out timer value for all ports on a device. This time-out is for link events such as sending a packet to receiving the corresponding acknowledge and sending a link-request to receiving the corresponding link-response. The reset value is the maximum time-out interval, and represents between 3 and 6 seconds.

Table 6-29. Bit Settings for Port Link Time-out Control CSR

Bit	Name	Reset Value	Description
0–23	time-out value	All 1s	time-out interval value
24-31	—		Reserved

6.8.2.3 Port General Control CSR (Block Offset 0x3C)

The port general control register contains control register bits applicable to all ports on a processing element.

Table 6-30. Bit Settings for Port General Control CSRs

Bit	Name	Reset Value	Description
0-1	—		Reserved
2	Discovered	0b0	This device has been located by the processing element responsible for system configuration 0b0 - The device has not been previously discovered 0b1 - The device has been discovered by another processing element
3-31	—		Reserved

6.8.2.4 Port *n* Link Maintenance Request CSRs (Block Offsets 0x40, 60, ..., 220)

The port link maintenance request registers are accessible both by a local processor and an external device. A write to one of these registers generates a link-request control symbol on the corresponding RapidIO port interface.

Table 6-31. Bit Settings for Port *n* Link Maintenance Request CSRs

Bit	Name	Reset Value	Description
0–28	—		Reserved
29-31	Command	0b000	Command to be sent in the link-request control symbol. If read, this field returns the last written value.

6.8.2.5 Port *n* Link Maintenance Response CSRs (Block Offsets 0x44, 64, ..., 224)

The port link maintenance response registers are accessible both by a local processor and an external device. A read to this register returns the status received in a link-response control symbol. The ackID_status and port_status fields are defined in Table 3-3 and Table 3-5. This register is read-only.

Table 6-32. Bit Settings for Port *n* Link Maintenance Response CSRs

Bit	Name	Reset Value	Description
0	response_valid	0b0	If the link-request causes a link-response, this bit indicates that the link-response has been received and the status fields are valid. If the link-request does not cause a link-response, this bit indicates that the link-request has been transmitted. This bit automatically clears on read.
1-21	—		Reserved
22-26	ackID_status	0b00000	ackID status field from the link-response control symbol
27-31	link_status	0b00000	link status field from the link-response control symbol

6.8.2.6 Port *n* Local ackID CSRs (Block Offsets 0x48, 68, ..., 228)

The port link local ackID status registers are accessible both by a local processor and an external device. A read to this register returns the local ackID status for both the out and input ports of the device.

Table 6-33. Bit Settings for Port *n* Local ackID Status CSRs

Bit	Name	Reset Value	Description
0	Clr_outstanding_ackIDs	0b0	Writing 0b1 to this bit causes all outstanding unacknowleged packets to be discarded. This bit should only be written when trying to recover a failed link. This bit is always logic 0 when read.
1-2	—		Reserved
3-7	Inbound_ackID	0b00000	Input port next expected ackID value
8-18	—		Reserved
19-23	Outstanding_ackID	0x00000	Output port unacknowledged ackID status. Next expected acknowledge control symbol ackID field that indicates the ackID value expected in the next received acknowledge control symbol.
24-26	—		Reserved
27-31	Outbound_ackID	0b00000	Output port next transmitted ackID value. Software writing this value can force retransmission of outstanding unacknowledged packets in order to manually implement error recovery.

6.8.2.7 Port *n* Error and Status CSRs (Block Offset 0x58, 78, ..., 238)

These registers are accessed when a local processor or an external device wishes to examine the port error and status information.

Table 6-34. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
0-10	—		Reserved
11	Output Retry-encountered	0b0	Output port has encountered a retry condition.This bit is set when bit 13 is set. Once set, remains set until written with a logic 1 to clear.
12	Output Retried	0b0	Output port has received a packet-retry control symbol and can not make forward progress. This bit is set when bit 13 is set and is cleared when a packet-accepted or a packet-not-accepted control symbol is received (read-only).
13	Output Retry-stopped	0b0	Output port has received a packet-retry control symbol and is in the “output retry-stopped” state (read-only).
14	Output Error-encountered	0b0	Output port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 15 is set. Once set, remains set until written with a logic 1 to clear.
15	Output Error-stopped	0b0	Output is in the “output error-stopped” state (read-only).
16-20	—		Reserved
21	Input Retry-stopped	0b0	Input port is in the “input retry-stopped” state (read-only).

Table 6-34. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
22	Input Error-encountered	0b0	Input port has encountered (and possibly recovered from) a transmission error. This bit is set when bit 23 is set. Once set, remains set until written with a logic 1 to clear.
23	Input Error-stopped	0b0	Input port is in the “input error-stopped” state (read-only).
24-26	—		Reserved
27	Port-write Pending	0b0	Port has encountered a condition which required it to initiate a Maintenance Port-write operation This bit is only valid if the device is capable of issuing a maintenance port-write transaction. Once set remains set until written with a logic 1 to clear.
28	—		Reserved
29	Port Error	0b0	Input or output port has encountered an error from which hardware was unable to recover. Once set, remains set until written with a logic 1 to clear.
30	Port OK	0b0	The input and output ports are initialized and the port is exchanging error-free control symbols with the attached device (read-only).
31	Port Uninitialized	0b1	Input and output ports are not initialized. This bit and bit 30 are mutually exclusive (read-only).

6.8.2.8 Port *n* Control CSR (Block Offsets 0x5C, 7C, ..., 23C)

The port *n* control registers contain control register bits for individual ports on a processing element.

Table 6-35. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
0-1	Port Width	see footnote[1]	Hardware width of the port (read-only): 0b00 - Single-lane port 0b01 - Four-lane port 0b10 - 0b11 - Reserved
2-4	Initialized Port Width	see footnote[2]	Width of the ports after initialized (read only): 0b000 - Single-lane port, lane 0 0b001 - Single-lane port, lane 2 0b010 - Four-lane port 0b011 - 0b111 - Reserved
5-7	Port Width Override	0b000	Soft port configuration to override the hardware size: 0b000 - No override 0b001 - Reserved 0b010 - Force single lane, lane 0 0b011 - Force single lane, lane 2 0b100 - 0b111 - Reserved If the port size is overridden, the port will re-initialize to change to the requested size.
8	Port Disable	0b0	Port disable: 0b0 - port receivers/drivers are enabled 0b1 - port receivers/drivers are disabled and are unable to receive/transmit any packets or control symbols

Bit	Name	Reset Value	Description
9	Output Port Enable	see footnote[3]	Output port transmit enable: 0b0 - port is stopped and not enabled to issue any packets except to route or respond to I/O logical MAINTENANCE packets. Control symbols are not affected and are sent normally. This is the recommended state after device reset. 0b1 - port is enabled to issue any packets
10	Input Port Enable	see footnote[4]	Input port receive enable: 0b0 - port is stopped and only enabled to route or respond I/O logical MAINTENANCE packets. Other packets generate packet-not-accepted control symbols to force an error condition to be signaled by the sending device. Control symbols are not affected and are received and handled normally. This is the recommended state after device reset. 0b1 - port is enabled to respond to any packet
11	Error Checking Disable	0b0	This bit disables all RapidIO transmission error checking 0b0 - Error checking and recovery is enabled 0b1 - Error checking and recovery is disabled Device behavior when error checking and recovery is disabled and an error condition occurs is undefined
12	Multicast-event Participant	see footnote[5]	Send incoming Multicast-event control symbols to this port (multiple port devices only)
13	—		Reserved
14	Enumeration Boundary	see footnote[6]	An enumeration boundary aware system enumeration algorithm shall honor this flag. The algorithm, on either the ingress or the egress port, shall not enumerate past a port with this bit set. This provides for software enforced enumeration domains within the RapidIO fabric.
15-19	—		Reserved
20-27	Re-transmit Suppression Mask	0x00	Suppress packet re-transmission on CRC error. For devices that support CRF: 0b0000_0000 - Error recovery suppression disabled 0bxxxx_xxx1 - Suppress CRF=0, priority 0 re-transmission 0bxxxx_xx1x - Suppress CRF=0, priority 1 re-transmission 0bxxxx_x1xx - Suppress CRF=0, priority 2 re-transmission 0bxxxx_1xxx - Suppress CRF=0, priority 3 re-transmission 0bxxx1_xxxx - Suppress CRF=1, priority 0 re-transmission 0bxx1x_xxxx - Suppress CRF=1, priority 1 re-transmission 0bx1xx_xxxx - Suppress CRF=1, priority 2 re-transmission 0b1xxx_xxxx - Suppress CRF=1, priority 3 re-transmission For devices that do not support CRF: 0b0000_0000 - Error recovery suppression disabled 0b0000_xxx1 - Suppress priority 0 re-transmission 0b0000_xx1x - Suppress priority 1 re-transmission 0b0000_x1xx - Suppress priority 2 re-transmission 0b0000_1xxx - Suppress priority 3 re-transmission 0b0001_0000 - reserved ... 0b1111_1111 - reserved This field is only valid if bit 25 of the Processing Element Features CAR is set.

Bit	Name	Reset Value	Description
28-30	—		Reserved
31	Port Type		This indicates the port type (read only) 0b0 - Reserved 0b1 - Serial port

[1]The Port Width reset value is implementation dependent

[2]The Initialized Port Width reset value is implementation dependent

[3]The Output Port Enable reset value is implementation dependent

[4]The Input Port Enable reset value is implementation dependent

[5]The Multicast-Event Participant reset value is implementation dependent

[6]The Enumeration Boundary reset value is implementation dependent. Provision shall be made to allow the reset value of this bit to be configurable on a per system basis if this feature is supported.

Chapter 7 Signal Descriptions

7.1 Introduction

This chapter contains the signal pin descriptions for a RapidIO 1x/4x LP-Serial port. The interface is defined either as a single- or four-lane, full duplex, point-to-point interface using differential signaling. A single-lane implementation consists of 4 wires and a four-lane implementation consists of 16 wires. The electrical details are described in Chapter 8, “Electrical Specifications.”

7.2 Signal Definitions

Table 7-1 provides a summary of the RapidIO 1x/4x LP-Serial signal pins as well as a short description of their functionality.

Table 7-1. 1x/4x LP-Serial Signal Description

Signal Name	I/O	Signal Meaning	Timing Comments
TD[0-3]	O	Transmit Data - The transmit data is a unidirectional point to point bus designed to transmit the packet information. The TD bus of one device is connected to the RD bus of the receiving device. TD[0] is used in 1x mode.	Clocking is embedded in data using 8B/10B encoding.
$\overline{\text{TD[0-3]}}$	O	Transmit Data complement—These signals are the differential pairs of the TD signals.	
RD[0-3]	I	Receive Data - The receive data is a unidirectional point to point bus designed to receive the packet information. The RD bus of one device is connected to the TD bus of the receiving device. RD[0] is used in 1x mode.	
$\overline{\text{RD[0-3]}}$	I	Receive Data complement—These signals are the differential pairs of the RD signals.	

7.3 Serial RapidIO Interface Diagrams

Figure 7-1 shows the signal interface diagram connecting two 1x devices together with the RapidIO 1x/4x LP-Serial interconnect.

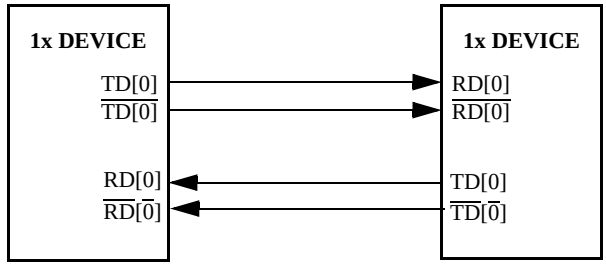


Figure 7-1. RapidIO 1x Device to 1x Device Interface Diagram

Figure 7-2 shows the signal interface diagram connecting two 4x devices together with the RapidIO 1x/4x LP-Serial interconnect.

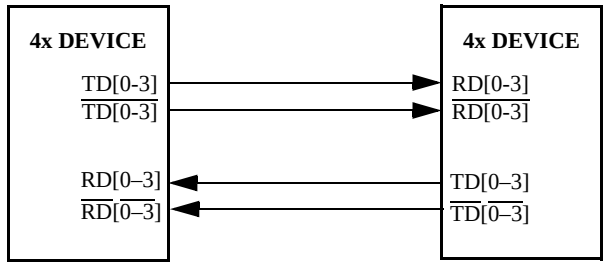


Figure 7-2. RapidIO 4x Device to 4x Device Interface Diagram

Figure 7-3 shows the connections between a 4x LP-Serial device and a 1x LP-Serial device.

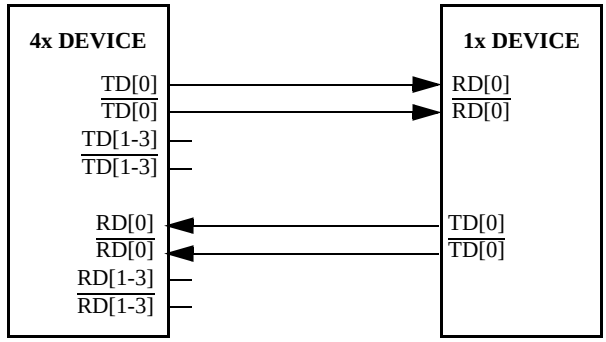


Figure 7-3. RapidIO 4x Device to 1x Device Interface Diagram

Chapter 8 Electrical Specifications

8.1 Introduction

The chapter defines the electrical specifications for the LP-Serial physical layer. The electrical specifications covers both single and multiple-lane links. Two transmitters (short run and long run) and a single receiver are specified for each of three baud rates, 1.25, 2.50, and 3.125 GBaud.

Two transmitter specifications allow for solutions ranging from simple board-to-board interconnect to driving two connectors across a backplane. A single receiver specification is given that will accept signals from both the short run and long run transmitter specifications.

The short run transmitter should be used mainly for chip-to-chip connections on either the same printed circuit board or across a single connector. This covers the case where connections are made to a mezzanine (daughter) card. The minimum swings of the short run specification reduce the overall power used by the transceivers.

The long run transmitter specifications use larger voltage swings that are capable of driving signals across backplanes. This allows a user to drive signals across two connectors and a backplane. The specifications allow a distance of at least 50 cm at all baud rates.

All unit intervals are specified with a tolerance of +/- 100 ppm. The worst case frequency difference between any transmit and receive clock will be 200 ppm.

To ensure interoperability between drivers and receivers of different vendors and technologies, AC coupling at the receiver input must be used.

8.2 Signal Definitions

LP-Serial links use differential signaling. This section defines terms used in the description and specification of differential signals. Figure 8-1 shows how the signals are defined. The figures shows waveforms for either a transmitter output (TD and $\overline{TD}$) or a receiver input (RD and $\overline{RD}$). Each signal swings between A Volts and B Volts where A > B. Using these waveforms, the definitions are as follows:

5. The transmitter output signals and the receiver input signals TD, $\overline{TD}$, RD and $\overline{RD}$ each have a peak-to-peak swing of A - B Volts
6. The differential output signal of the transmitter, V_{OD}, is defined as V_{TD}-$V_{\overline{TD}}$.
7. The differential input signal of the receiver, V_{ID}, is defined as V_{RD}-$V_{\overline{RD}}$.
8. The differential output signal of the transmitter and the differential input signal of the receiver each range from A - B to -(A - B) Volts
9. The peak value of the differential transmitter output signal and the differential receiver input signal is A - B Volts
10. The peak-to-peak value of the differential transmitter output signal and the differential receiver input signal is 2 * (A - B) Volts

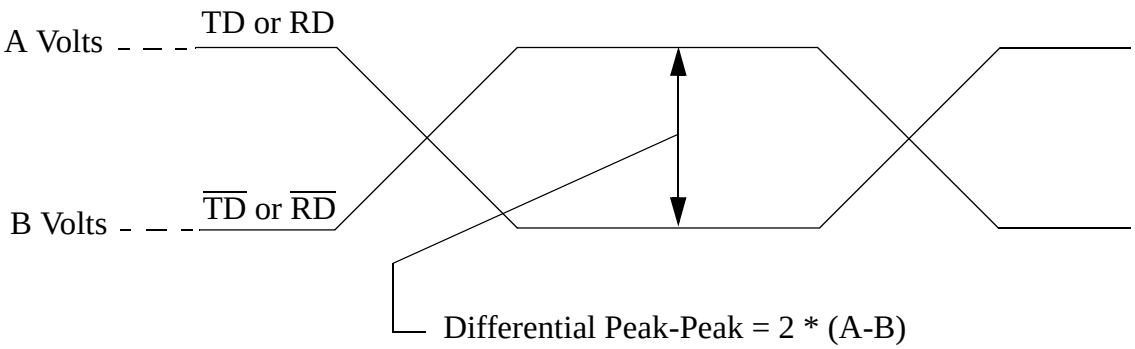


Figure 8-1. Differential Peak-Peak Voltage of Transmitter or Receiver

To illustrate these definitions using real values, consider the case of a CML (Current Mode Logic) transmitter that has a common mode voltage of 2.25 V and each of its outputs, TD and $\overline{TD}$, has a swing that goes between 2.5V and 2.0V. Using these values, the peak-to-peak voltage swing of the signals TD and $\overline{TD}$ is 500 mV p-p. The differential output signal ranges between 500 mV and -500 mV. The peak differential voltage is 500 mV. The peak-to-peak differential voltage is 1000 mV p-p.

8.3 Equalization

With the use of high speed serial links, the interconnect media will cause degradation of the signal at the receiver. Effects such as Inter-Symbol Interference (ISI) or data dependent jitter are produced. This loss can be large enough to degrade the eye opening at the receiver beyond what is allowed in the specification. To negate a portion of these effects, equalization can be used. The most common equalization techniques that can be used are:

- Pre-emphasis on the transmitter
- A passive high pass filter network placed at the receiver. This is often referred to as passive equalization.
- The use of active circuits in the receiver. This is often referred to as adaptive equalization.

8.4 Explanatory Note on Transmitter and Receiver Specifications

AC electrical specifications are given for transmitter and receiver. Long run and short run interfaces at three baud rates (a total of six cases) are described.

The parameters for the AC electrical specifications are guided by the XAUI electrical interface specified in Clause 47 of IEEE 802.3ae-2002.

XAUI has similar application goals to serial RapidIO, as described in Section 8.1. The goal of this standard is that electrical designs for serial RapidIO can reuse electrical designs for XAUI, suitably modified for applications at the baud intervals and reaches described herein.

8.5 Transmitter Specifications

LP-Serial transmitter electrical and timing specifications are stated in the text and tables of this section.

The differential return loss, S11, of the transmitter in each case shall be better than

-10 dB for (Baud Frequency)/10 < Freq(f) < 625 MHz, and

-10 dB + 10log(f/625 MHz) dB for 625 MHz <= Freq(f) <= Baud Frequency

The reference impedance for the differential return loss measurements is 100 Ohm resistive. Differential return loss includes contributions from on-chip circuitry, chip packaging and any off-chip components related to the driver. The output impedance requirement applies to all valid output levels.

It is recommended that the 20%-80% rise/fall time of the transmitter, as measured at the transmitter output, in each case have a minimum value 60 ps.

It is recommended that the timing skew at the output of an LP-Serial transmitter between the two signals that comprise a differential pair not exceed 25 ps at 1.25 GB, 20 ps at 2.50 GB and 15 ps at 3.125 GB.

Table 8-1. Short Run Transmitter AC Timing Specifications - 1.25 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	500	1000	mV p-p	
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	800	800	ps	+/- 100 ppm

Table 8-2. Short Run Transmitter AC Timing Specifications - 2.5 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	500	1000	mV p-p	

Table 8-2. Short Run Transmitter AC Timing Specifications - 2.5 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple Output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	400	400	ps	+/- 100 ppm

Table 8-3. Short Run Transmitter AC Timing Specifications - 3.125 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	500	1000	mV p-p	
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	320	320	ps	+/- 100 ppm

Table 8-4. Long Run Transmitter AC Timing Specifications - 1.25 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	800	1600	mV p-p	
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	800	800	ps	+/- 100 ppm

Table 8-5. Long Run Transmitter AC Timing Specifications - 2.5 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	800	1600	mV p-p	
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	400	400	ps	+/- 100 ppm

Table 8-6. Long Run Transmitter AC Timing Specifications - 3.125 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Output Voltage,	V_O	-0.40	2.30	Volts	Voltage relative to COMMON of either signal comprising a differential pair
Differential Output Voltage	V_{DIFFPP}	800	1600	mV p-p	
Deterministic Jitter	J_D		0.17	UI p-p	
Total Jitter	J_T		0.35	UI p-p	
Multiple output skew	S_{MO}		1000	ps	Skew at the transmitter output between lanes of a multilane link
Unit Interval	UI	320	320	ps	+/- 100 ppm

For each baud rate at which an LP-Serial transmitter is specified to operate, the output eye pattern of the transmitter shall fall entirely within the unshaded portion of the Transmitter Output Compliance Mask shown in Figure 8-2 with the parameters specified in Table 8-7 when measured at the output pins of the device and the device is driving a 100 Ohm +/- 5% differential resistive load. The output eye pattern of a LP-Serial transmitter that implements pre-emphasis (to equalize the link and reduce inter-symbol interference) need only comply with the Transmitter Output Compliance Mask when pre-emphasis is disabled or minimized.

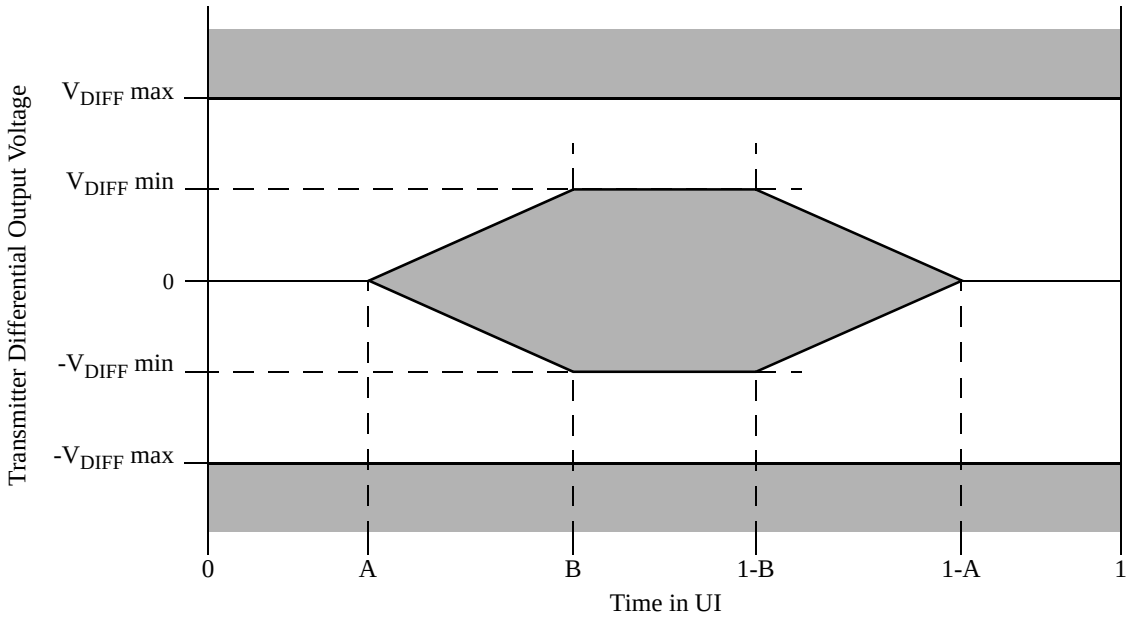


Figure 8-2. Transmitter Output Compliance Mask

Table 8-7. Transmitter Differential Output Eye Diagram Parameters

Transmitter Type	V_{DIFF}min (mV)	V_{DIFF}max (mV)	A (UI)	B (UI)
1.25 GBaud short range	250	500	0.175	0.39
1.25 GBaud long range	400	800	0.175	0.39
2.5 GBaud short range	250	500	0.175	0.39
2.5 GBaud long range	400	800	0.175	0.39
3.125 GBaud short range	250	500	0.175	0.39
3.125 GBaud long range	400	800	0.175	0.39

8.6 Receiver Specifications

LP-Serial receiver electrical and timing specifications are stated in the text and tables of this section.

Receiver input impedance shall result in a differential return loss better that 10 dB and a common mode return loss better than 6 dB from 100 MHz to (0.8)*(Baud Frequency). This includes contributions from on-chip circuitry, the chip package and any off-chip components related to the receiver. AC coupling components are included in this requirement. The reference impedance for return loss measurements is 100 Ohm resistive for differential return loss and 25 Ohm resistive for common mode.

Table 8-8. Receiver AC Timing Specifications - 1.25 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Differential Input Voltage	V_{IN}	200	1600	mV p-p	Measured at receiver
Deterministic Jitter Tolerance	J_D	0.37		UI p-p	Measured at receiver
Combined Deterministic and Random Jitter Tolerance	J_{DR}	0.55		UI p-p	Measured at receiver
Total Jitter Tolerance[1]	J_T	0.65		UI p-p	Measured at receiver
Multiple Input Skew	S_{MI}		24	ns	Skew at the receiver input between lanes of a multi-lane link
Bit Error Ratio	BER		10^{-12}		
Unit Interval	UI	800	800	ps	+/- 100 ppm

Notes
1. Total jitter is composed of three components, deterministic jitter, random jitter and single frequency sinusoidal jitter. The sinusoidal jitter may have any amplitude and frequency in the unshaded region of Figure 8-3. The sinusoidal jitter component is included to ensure margin for low frequency jitter, wander, noise, crosstalk and other variable system effects.

Table 8-9. Receiver AC Timing Specifications - 2.5 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Differential Input Voltage	V_{IN}	200	1600	mV p-p	Measured at receiver
Deterministic Jitter Tolerance	J_D	0.37		UI p-p	Measured at receiver
Combined Deterministic and Random Jitter Tolerance	J_{DR}	0.55		UI p-p	Measured at receiver
Total Jitter Tolerance[1]	J_T	0.65		UI p-p	Measured at receiver
Multiple Input Skew	S_{MI}		24	ns	Skew at the receiver input between lanes of a multi-lane link

Table 8-9. Receiver AC Timing Specifications - 2.5 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Bit Error Ratio	BER		10^{-12}		
Unit Interval	UI	400	400	ps	+/- 100 ppm

Notes
1. Total jitter is composed of three components, deterministic jitter, random jitter and single frequency sinusoidal jitter. The sinusoidal jitter may have any amplitude and frequency in the unshaded region of Figure 8-3. The sinusoidal jitter component is included to ensure margin for low frequency jitter, wander, noise, crosstalk and other variable system effects.

Table 8-10. Receiver AC Timing Specifications - 3.125 GBaud

Characteristic	Symbol	Range		Unit	Notes
		Min	Max		
Differential Input Voltage	V_{IN}	200	1600	mV p-p	Measured at receiver
Deterministic Jitter Tolerance	J_D	0.37		UI p-p	Measured at receiver
Combined Deterministic and Random Jitter Tolerance	J_{DR}	0.55		UI p-p	Measured at receiver
Total Jitter Tolerance[1]	J_T	0.65		UI p-p	Measured at receiver
Multiple Input Skew	S_{MI}		22	ns	Skew at the receiver input between lanes of a multi-lane link
Bit Error Ratio	BER		10^{-12}		
Unit Interval	UI	320	320	ps	+/- 100 ppm

Notes
1. Total jitter is composed of three components, deterministic jitter, random jitter and single frequency sinusoidal jitter. The sinusoidal jitter may have any amplitude and frequency in the unshaded region of Figure 8-3. The sinusoidal jitter component is included to ensure margin for low frequency jitter, wander, noise, crosstalk and other variable system effects.

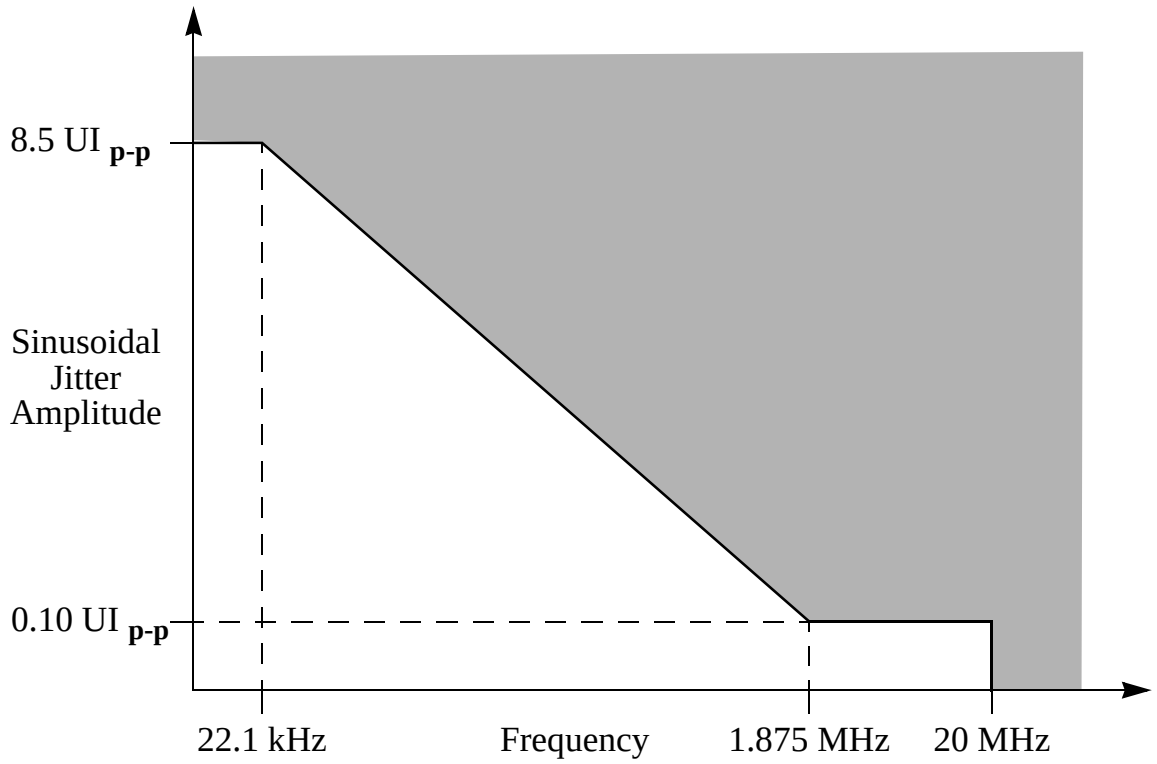


Figure 8-3. Single Frequency Sinusoidal Jitter Limits

8.7 Receiver Eye Diagrams

For each baud rate at which an LP-Serial receiver is specified to operate, the receiver shall meet the corresponding Bit Error Ratio specification (Table 8-8, Table 8-9, Table 8-10) when the eye pattern of the receiver test signal (exclusive of sinusoidal jitter) falls entirely within the unshaded portion of the Receiver Input Compliance Mask shown in Figure 8-4 with the parameters specified in Table 8-11. The eye pattern of the receiver test signal is measured at the input pins of the receiving device with the device replaced with a 100 Ohm +/- 5% differential resistive load.

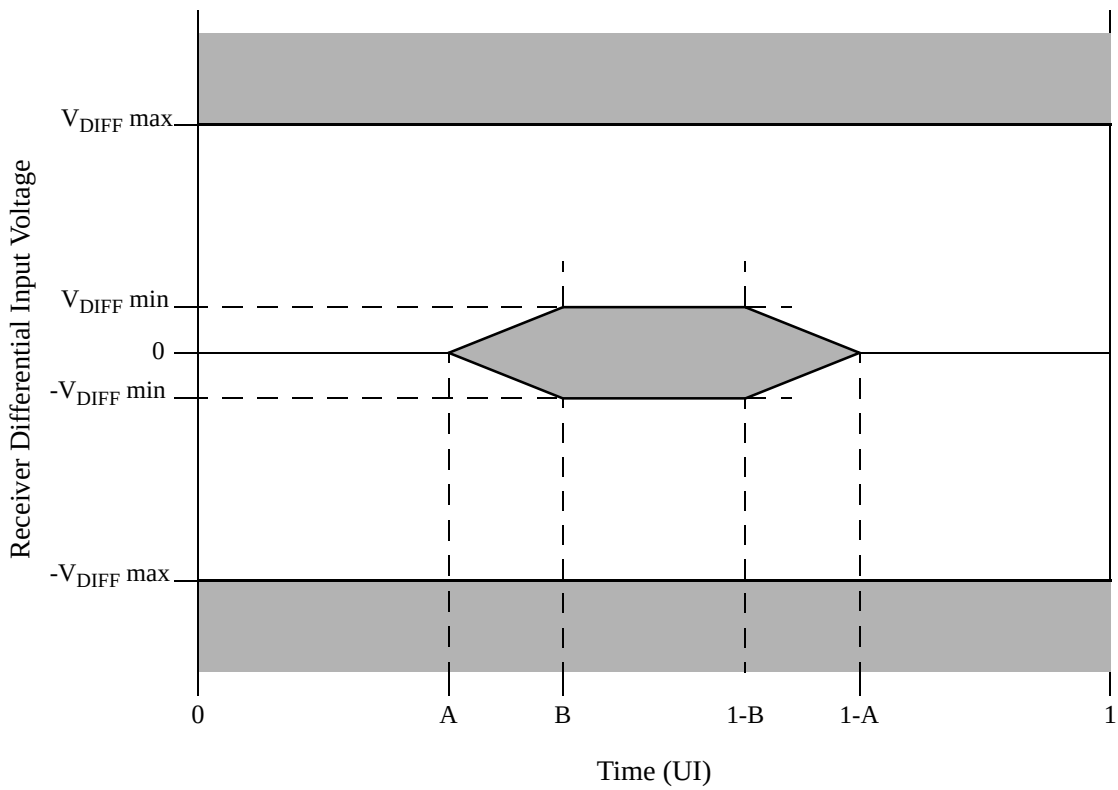


Figure 8-4. Receiver Input Compliance Mask

Table 8-11. Receiver Input Compliance Mask Parameters exclusive of Sinusoidal Jitter

Receiver Type	V_{DIFF}min (mV)	V_{DIFF}max (mV)	A (UI)	B (UI)
1.25 GBaud	100	800	0.275	0.400
2.5 GBaud	100	800	0.275	0.400
3.125 GBaud	100	800	0.275	0.400

8.8 Measurement and Test Requirements

Since the LP-Serial electrical specification are guided by the XAUI electrical interface specified in Clause 47 of IEEE 802.3ae-2002, the measurement and test requirements defined here are similarly guided by Clause 47. In addition, the CJPAT test pattern defined in Annex 48A of IEEE802.3ae-2002 is specified as the test pattern for use in eye pattern and jitter measurements. Annex 48B of IEEE802.3ae-2002 is recommended as a reference for additional information on jitter test methods.

8.8.1 Eye template measurements

For the purpose of eye template measurements, the effects of a single-pole high pass filter with a 3 dB point at (Baud Frequency)/1667 is applied to the jitter. The data pattern for template measurements is the Continuous Jitter Test Pattern (CJPAT) defined in Annex 48A of IEEE802.3ae. All lanes of the LP-Serial link shall be active in both the transmit and receive directions, and opposite ends of the links shall use asynchronous clocks. Four lane implementations shall use CJPAT as defined in Annex 48A. Single lane implementations shall use the CJPAT sequence specified in Annex 48A for transmission on lane 0. The amount of data represented in the eye shall be adequate to ensure that the bit error ratio is less than 10^{-12}. The eye pattern shall be measured with AC coupling and the compliance template centered at 0 Volts differential. The left and right edges of the template shall be aligned with the mean zero crossing points of the measured data eye. The load for this test shall be 100 Ohms resistive +/- 5% differential to 2.5 GHz.

8.8.2 Jitter test measurements

For the purpose of jitter measurement, the effects of a single-pole high pass filter with a 3 dB point at (Baud Frequency)/1667 is applied to the jitter. The data pattern for jitter measurements is the Continuous Jitter Test Pattern (CJPAT) pattern defined in Annex 48A of IEEE802.3ae. All lanes of the LP-Serial link shall be active in both the transmit and receive directions, and opposite ends of the links shall use asynchronous clocks. Four lane implementations shall use CJPAT as defined in Annex 48A. Single lane implementations shall use the CJPAT sequence specified in Annex 48A for transmission on lane 0. Jitter shall be measured with AC coupling and at 0 Volts differential. Jitter measurement for the transmitter (or for calibration of a jitter tolerance setup) shall be performed with a test procedure resulting in a BER curve such as that described in Annex 48B of IEEE802.3ae.

8.8.3 Transmit jitter

Transmit jitter is measured at the driver output when terminated into a load of 100 Ohms resistive +/- 5% differential to 2.5 GHz.

8.8.4 Jitter tolerance

Jitter tolerance is measured at the receiver using a jitter tolerance test signal. This signal is obtained by first producing the sum of deterministic and random jitter defined in Section 8.6 and then adjusting the signal amplitude until the data eye contacts the 6 points of the minimum eye opening of the receive template shown in Figure 8-4 and Table 8-11. Note that for this to occur, the test signal must have vertical waveform symmetry about the average value and have horizontal symmetry (including jitter) about the mean zero crossing. Eye template measurement requirements are as defined above. Random jitter is calibrated using a high pass filter with a low frequency corner at 20 MHz and a 20 dB/decade rolloff below this. The required sinusoidal jitter specified in Section 8.6 is then added to the signal and the test load is replaced by the receiver being tested.

Blank page

Annex A Interface Management (Informative)

A.1 Introduction

This appendix contains state machine descriptions that illustrate a number of behaviors that are described in the *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*. They are included as examples and are believed to be correct, however, actual implementations should not use the examples directly.

A.2 Packet Retry Mechanism

This section contains the example packet retry mechanism state machine referred to in Section 5.6, “Packet Transmission Protocol”.

Packet retry recovery actually requires two inter-dependent state machines in order to operate, one associated with the input port and the other with the output port on the two connected devices. The two state machines work together to attempt recovery from a retry condition.

A.2.1 Input port retry recovery state machine

If a packet cannot be accepted by a receiver for reasons other than error conditions, such as a full input buffer, the receiver follows the state sequence shown in Figure A-1.

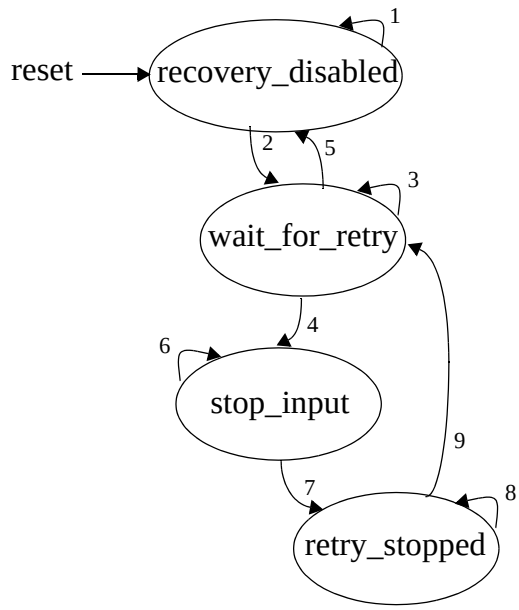


Figure A-1. Input Port Retry Recovery State Machine

Table A-1 describes the state transition arcs for Figure A-1. The states referenced in the comments in quotes are the RapidIO 1x/4x LP-Serial defined status states, not states in this state machine.

Table A-1. Input Port Retry Recovery State Machine Transition Table

Arc	Current State	Next state	cause	Comments
1	recovery_disabled	recovery_disabled	Remain in this state until the input port is enabled to receive packets.	This is the initial state after reset. The input port can't be enabled before the initialization sequence has been completed, and may be controlled through other mechanisms as well, such as a software enable bit.
2	recovery_disabled	wait_for_retry	Input port is enabled.	
3	wait_for_retry	wait_for_retry	Remain in this state until a packet retry situation has been detected.	
4	wait_for_retry	stop_input	A packet retry situation has been detected.	Usually this is due to an internal resource problem such as not having packet buffers available for low priority packets.
5	wait_for_retry	recovery_disabled	Input port is disabled.	
6	stop_input	stop_input	Remain in this state until described input port stop activity is completed.	Send a packet-retry control symbol with the expected ackID, discard the packet, and don't change the expected ackID. This will force the attached device to initiate recovery starting at the expected ackID. Clear the "Port Normal" state and set the "Input Retry-stopped" state.
7	stop_input	retry_stopped	Input port stop activity is complete.	

Table A-1. Input Port Retry Recovery State Machine Transition Table (Continued)

Arc	Current State	Next state	cause	Comments
8	retry_stopped	retry_stopped	Remain in this state until a restart-from-retry or link request (restart-from-error) control symbol is received or an input port error is encountered.	The "Input Retry-stopped" state causes the input port to silently discard all incoming packets and not change the expected ackID value.
9	retry_stopped	wait_for_retry	Received a restart-from-retry or a link request (restart-from-error) control symbol or an input port error is encountered.	Clear the "Input Retry-stopped" state and set the "Port Normal" state. An input port error shall cause a clean transition between the retry recovery state machine and the error recovery state machine.

A.2.2 Output port retry recovery state machine

On receipt of an error-free packet-retry control symbol, the attached output port follows the behavior shown in Figure A-2. The states referenced in the comments in quotes are the RapidIO 8/16 LP-LVDS defined status states, not states in this state machine.

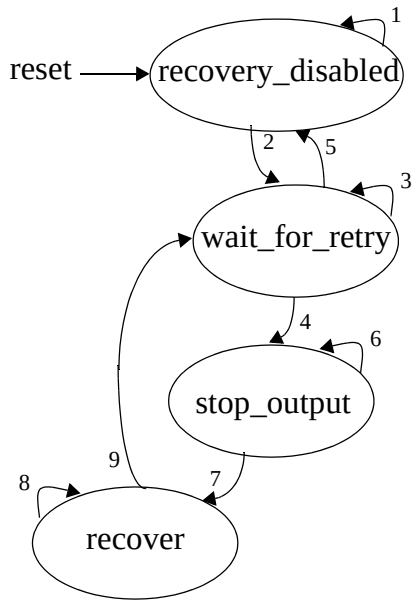


Figure A-2. Output Port Retry Recovery State Machine

Table A-2 describes the state transition arcs for Figure A-2.

Table A-2. Output Port Retry Recovery State Machine Transition Table

Arc	Current State	Next state	cause	Comments
1	recovery_disabled	recovery_disabled	Remain in this state until the output port is enabled to receive packets.	This is the initial state after reset. The output port can't be enabled before the initialization sequence has been completed, and may be controlled through other mechanisms as well, such as a software enable bit.
2	recovery_disabled	wait_for_retry	Output port is enabled.	
3	wait_for_retry	wait_for_retry	Remain in this state until a packet-retry control symbol is received.	The packet-retry control symbol shall be error free.
4	wait_for_retry	stop_output	A packet-retry control symbol has been received.	Start the output port stop procedure.
5	wait_for_retry	recovery_disabled	Output port is disabled.	
6	stop_output	stop_output	Remain in this state until the output port stop procedure is completed.	Clear the "Port Normal" state, set the "Output Retry-stopped" state, and stop transmitting new packets.
7	stop_output	recover	Output port stop procedure is complete.	
8	recover	recover	Remain in this state until the internal recovery procedure is completed.	The packet sent with the ackID value returned in the packet-retry control symbol and all subsequent packets shall be retransmitted. Output port state machines and the outstanding ackID scoreboard shall be updated with this information, then clear the "Output Retry-stopped" state and set the "Port Normal" state to restart the output port. Receipt of a packet-not-accepted control symbol or other output port error during this procedure shall cause a clean transition between the retry recovery state machine and the error recovery state machine. Send restart-from-retry control symbol.
9	recover	wait_for_retry	Internal recovery procedure is complete.	Retransmission has started, so return to the wait_for_retry state to wait for the next packet-retry control symbol.

A.3 Error Recovery

This section contains the error recovery state machine referred to in Section 5.11.2, “Link Behavior Under Error.”

Error recovery actually requires two inter-dependent state machines in order to operate, one associated with the input port and the other with the output port on the two connected devices. The two state machines work together to attempt recovery.

A.3.1 Input port error recovery state machine

There are a variety of recoverable error types described in detail in Section 5.11.2, “Link Behavior Under Error”. The first group of errors are associated with the input port, and consists mostly of corrupt packet and control symbols. An example of a corrupt packet is a packet with an incorrect CRC. An example of a corrupt control symbol is a control symbol with error on the 5-bit CRC control symbol. The recovery state machine for the input port of a RapidIO link is shown in Figure A-3.

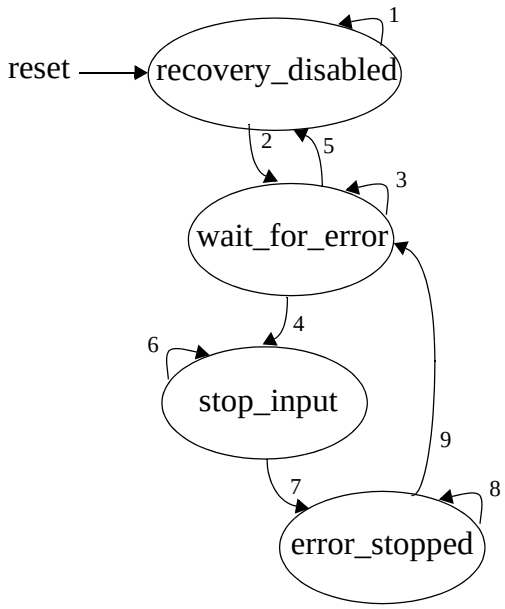


Figure A-3. Input Port Error Recovery State Machine

Table A-3 describes the state transition arcs for Figure A-3. The states referenced in the comments in quotes are the RapidIO 1x/4x LP-Serial defined status states, not states in this state machine.

Table A-3. Input Port Error Recovery State Machine Transition Table

Arc	Current State	Next state	cause	Comments
1	recovery_disabled	recovery_disabled	Remain in this state until error recovery is enabled.	This is the initial state after reset. Error recovery can't be enabled before the initialization sequence has been completed, and may be controlled through other mechanisms as well, such as a software enable bit.
2	recovery_disabled	wait_for_error	Error recovery is enabled.	
3	wait_for_error	wait_for_error	Remain in this state until a recoverable error is detected.	Detected errors and the level of coverage is implementation dependent.
4	wait_for_error	stop_input	A recoverable error has been detected.	An output port associated error will not cause this transition, only an input port associated error.
5	wait_for_error	recovery_disabled	Error recovery is disabled.	
6	stop_input	stop_input	Remain in this state until described input port stop activity is completed.	Send a packet-not-accepted control symbol and, if the error was on a packet, discard the packet and don't change the expected ackID value. This will force the attached device to initiate recovery. Clear the "Port Normal" state and set the "Input Error-stopped" state.
7	stop_input	error_stopped	Input port stop activity is complete.	
8	error_stopped	error_stopped	Remain in this state until a link request (restart-from-error) control symbol is received.	The "Input Error-stopped" state causes the input port to silently discard all subsequent incoming packets and ignore all subsequent input port errors.
9	error_stopped	wait_for_error	Received a link request (restart-from-error) control symbol.	Clear the "Input Error-stopped" state and set the "Port Normal" state, which will put the input port back in normal operation.

A.3.2 Output port error recovery state machine

The second recoverable group of errors described in Section 5.11.2, "Link Behavior Under Error" is associated with the output port, and is comprised of control symbols that are error-free and indicate that the attached input port has detected a transmission error or some other unusual situation has occurred. An example of this situation is indicated by the receipt of a packet-not-accepted control symbol. The state machine for the output port is shown in Figure A-4.

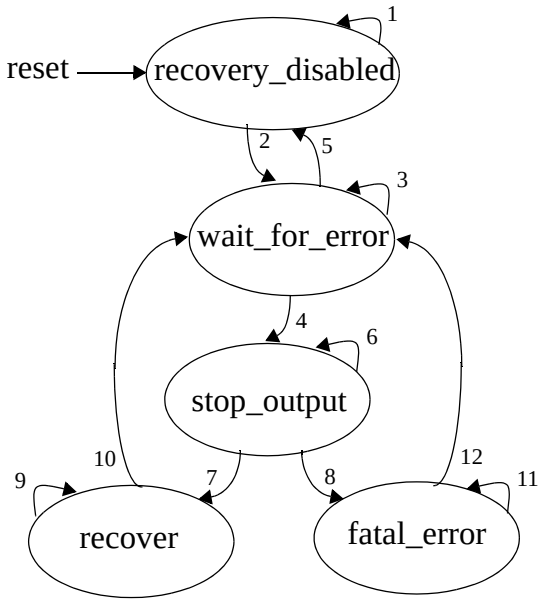


Figure A-4. Output Port Error Recovery State Machine

Table A-4 describes the state transition arcs for Figure A-4. The states referenced in the comments in quotes are the RapidIO 8/16 LP-LVDS defined status states, not states in this state machine.

Table A-4. Output Port Error Recovery State Machine Transition Table

Arc	Current State	Next state	cause	Comments
1	recovery_disabled	recovery_disabled	Remain in this state until error recovery is enabled.	This is the initial state after reset. Error recovery can't be enabled before the initialization sequence has been completed, and may be controlled through other mechanisms as well, such as a software enable bit.
2	recovery_disabled	wait_for_error	Error recovery is enabled.	
3	wait_for_error	wait_for_error	Remain in this state until a recoverable error is detected.	Detected errors and the level of coverage is implementation dependent.
4	wait_for_error	stop_output	A recoverable error has been detected.	An input port associated error will not cause this transition, only an output port associated error.
5	wait_for_error	recovery_disabled	Error recovery is disabled.	

Table A-4. Output Port Error Recovery State Machine Transition Table (Continued)

Arc	Current State	Next state	cause	Comments
6	stop_output	stop_output	Remain in this state until an exit condition occurs.	Clear the "Port Normal" state, set the "Output Error-stopped" state, stop transmitting new packets, and send a link-request/input-status control symbol. Ignore all subsequent output port errors. The input on the attached device is in the "Input Error-stopped" state and is waiting for a link-request/input-status in order to be re-enabled to receive packets. An implementation may wish to time-out several times before regarding a time-out as fatal using a threshold counter or some other mechanism.
7	stop_output	recover	The link-response is received and returned an outstanding ackID value	An outstanding ackID is a value sent out on a packet that has not been acknowledged yet. In the case where no ackID is outstanding the returned ackID value shall match the next expected/next assigned ackID value, indicating that the devices are synchronized. Recovery is possible, so follow recovery procedure.
8	stop_output	fatal_error	The link-response is received and returned an ackID value that is not outstanding, or timed out waiting for the link-response.	Recovery is not possible, so start error shutdown procedure.
9	recover	recover	Remain in this state until the internal recovery procedure is completed.	The packet sent with the ackID value returned in the link-response and all subsequent packets shall be retransmitted. All packets transmitted with ackID values preceding the returned value were received by the attached device, so they are treated as if packet-accepted control symbols have been received for them. Output port state machines and the outstanding ackID scoreboard shall be updated with this information, then clear the "Output Error-stopped" state and set the 'Port Normal" state to restart the output port.
10	recover	wait_for_error	The internal recovery procedure is complete.	retransmission (if any was necessary) has started, so return to the wait_for_error state to wait for the next error.

Table A-4. Output Port Error Recovery State Machine Transition Table (Continued)

Arc	Current State	Next state	cause	Comments
11	fatal_error	fatal_error	Remain in this state until error shutdown procedure is completed.	Clear the "Output Error-stopped" state, set the "Port Error" state, and signal a system error.
12	fatal_error	wait_for_error	Error shutdown procedure is complete.	Return to the wait_for_error state.

Blank page

Annex B Critical Resource Performance Limits (Informative)

The RapidIO LP-Serial layer is intended for use over links whose length ranges from centimeters to tens of meters. The shortest length links will almost certainly use copper printed circuit board traces. The longer lengths will require the use of fiber optics (optical fiber and electro-optical converters) to overcome the high frequency losses of long copper printed circuit board traces or cable. The longer lengths will also have significant propagation delay which can degrade the usable bandwidth of a link.

The serial protocol is a handshake protocol. Each packet transmitted by a port is assigned an ID (the ackID) and a copy of the packet is retained by the port in a holding buffer until the packet is accepted by the port's link partner. The number of packets that a port can transmit without acknowledgment is limited to the lesser of the number of distinct ackIDs and the number of buffers available to hold unacknowledged packets. Which ever is the limiting resource, ackIDs or holding buffers, will be called the "critical resource".

The concern is the time between the assignment of a critical resource to a packet and the release of that resource as a consequence of the packet being accepted by the link partner. Call this time the resource_release_delay. When the resource_release_delay is less than the time it takes to transmit a number of packets equal to the number of distinct critical resource elements, there is no degradation of link performance. When the resource_release_delay is greater than the time it takes to transmit a number of packets equal to the number of distinct critical resource elements, the transmitter may have to stall from time to time waiting for a free critical resource. This will degraded the usable link bandwidth. The onset of degradation will depend on the average length of transmitted packets and the physical length of the link as reflected in the resource_release_delay.

The following example provides some idea of the impact on link performance of the interaction between link length and a critical resource. For purposes of this example, the following assumptions are made.

1. The link is a 4 lane (4x) link.
2. The link uses optical fiber and electro-optical transceivers to allow link lengths of tens of meters. The propagation delay of the optical fiber is 0.45c.
3. The width of the data path within the port is 4 bytes.

4. The data path and logic within the port run at a clock rate equal to the aggregate unidirectional data rate of the link divided by 32. This is referred to as the logic clock. One cycle of this clock is referred to a one logic clock cycle. (If the aggregate unidirectional baud rate of the link was used to compute the logic clock, the baud rate would be divided by 40. With 8B/10B encoding, the baud rate is 1.25 times the data rate.)
5. The minimum length packet header is used. Write request packets have a length of 12 bytes plus a payload containing an integer multiple of 8 bytes. Read request packets have a length of 12 bytes. Read response packets have a length of 8 bytes plus a payload containing an integer multiple of 8 bytes.
6. The beginning and end of each packet is delimited by a control symbol. A single control symbol may delimit both the end of one packet and the beginning of the next packet.
7. Packet acknowledgments are carried in packet delimiter control symbols when ever possible to achieve the efficiency provided by the dual stype control symbol. This implies that a packet acknowledgment must wait for an end-of-packet control symbol if packet transmission is in progress when the packet acknowledgment becomes available.
8. The logic and propagation delay in the packet transmission direction is comprised of the following components.

Table B-12. Packet Transmission Delay Components

Item	Time required
Generate start-of-packet control symbol (critical resource is available)	1 logic clock cycle
Generate start-of-packet control symbol CRC	1 logic clock cycle
8B/10B encode delimiter and start-of-packet control symbol	1 logic clock cycle
Serialize and transmit delimiter and start-of-packet control symbol	1 logic clock cycle
PCB copper and electro-optical transmitter delay	2 ns
Optical fiber delay	fiber_length/0.45c
Electro-optical receiver and pcb copper delay	2 ns
Receive and deserialize delimiter and start-of-packet control symbol	0.5 logic clock cycles
Receive and deserialize packet	depends on packet
Receive and deserialize delimiter and end-of-packet control symbol	1 logic clock cycle
8B/10B decode delimiter and end-of-packet control symbol	1 logic clock cycle
Check CRC of end-of-packet control symbol	1 logic clock cycle
Make packet acceptance decision	1 logic clock cycle

9. The logic and propagation delay in the packet acknowledgment direction is comprised of the following.

Table B-13. Packet Acknowledgment Delay Components

Item	Time required
Wait for end-of-packet if packet transmission is in progress, generate packet-acknowledgment control symbol and control symbol CRC	depends on packet >= 2 logic clock cycles
8B/10B encode delimiter and packet-acknowledgment control symbol	1 logic clock cycle
Serialize and transmit delimiter and packet-acknowledgment control symbol	1 logic clock cycle
PCB copper and electro-optical transmitter delay	2 ns
Optical fiber delay	fiber_length/0.45c
Electro-optical receiver and pcb copper delay	2 ns
Receive and deserialize delimiter and packet-acknowledgment control symbol	0.5 logic clock cycles
8B/10B decode delimiter and packet-acknowledgment control symbol	1 logic clock cycle
Check CRC of packet-acknowledgment control symbol	1 logic clock cycle
Make decision to free critical resource	1 logic clock cycle

The packet times in the above tables depend on packet length which in turn depends on packet type and payload size. Since packet traffic will typically involve a mixture of packet types and payload sizes, the traffic in each direction will be assumed to contain an equal number of read, write and response packets and average payloads of 8, 32, and 64 bytes.

The number of logic clock cycles required to transmit or receive a packet is given in the following table as a function of packet type and payload size.

Table B-14. Packet Delays

Packet Type	Packet Header bytes	Data Payload bytes	Transmit/Receive Time logic clock cycles
Read	12	0	3
Response	8	8	4
		32	10
		64	18
Write	12	8	5
		32	11
		64	19

Using the above table and the assumed equal number of read, write and response packets, the average number of logic clock cycles to transmit or received a packet is 4, 8, and 13.3 respectively for packet payloads of 8, 32, and 64 bytes. The average wait for the completion of a packet being transmitted is assumed to be 1/2 the transmit time.

The following table gives the maximum length of the optical fiber before the packet transmission rate becomes limited by the critical resource for a 4x link operating at unidirectional data rates of 4.0, 8.0 and 10.0 Gb/s.

Table B-15. Maximum Transmission Distances

Number of Critical Resources Available	Data Payload (Bytes)	Maximum Fiber Length Before Critical Resource Limited (Meters)		
		4.0 Gb/s link	8.0 Gb/s link	10.0 Gb/s link
4	8	-	-	-
	32	4.3	1.9	1.4
	64	11.4	5.5	4.3
8	8	9.7	4.6	3.5
	32	23.6	11.5	9.1
	64	42.2	20.8	16.6
16	8	31.1	15.3	12.1
	32	62.2	30.8	24.6
	64	103.7	51.6	41.1
24	8	52.5	26.0	20.7
	32	100.8	50.2	40.0
	64	165.2	82.3	65.7
32	8	74.0	36.7	29.3
	32	139.5	69.5	55.5
	64	226.7	113.1	90.3

Annex C Manufacturability and Testability (Informative)

It is not possible in many cases for assembly vendors to verify the integrity of soldered connections between components and the printed circuit boards to which they are attached. Alternative methods to direct probing are needed to insure high yields for printed circuit assemblies which include LP-Serial RapidIO devices.

It is recommended that component vendors support IEEE Std. 1149.6 (commonly known as “AC-JTAG”) on all connections to LP-Serial RapidIO links. (Note: IEEE Std. 1149.6 is needed, in addition to IEEE Std. 1149.1, due the fact that RapidIO LP-Serial lanes are AC-coupled.) This provides boundary scan capability on all TD, TDN, RD, and RDN pins on a component which supports one or more LP-Serial RapidIO ports.

The IEEE Std. 1149.6 is available from the IEEE.

Blank page

Glossary of Terms and Abbreviations

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

A

AC Coupling. A method of connecting two devices together that does not pass DC.

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

ANSI. American National Standards Institute.

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.

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

C

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

Code-group. A 10-bit entity produced by the 8B/10B encoding process and the input to the 8B/10B decoding process.

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

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

CRC. Cyclic redundancy code

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.

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

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

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

Device ID. The identifier of a processing element connected to the RapidIO interconnect.

Direct Memory Access (**DMA**). A process element that can independently read and write system memory.

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

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

E

EMI. Electromagnetic Interference.

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

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

End point free device. A processing element which does not contain end point functionality.

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

External processing element. A processing element other than the processing element in question.

F

Fabric. A series of interconnected switch devices, typically used in reference to a switch fabric.

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

FIFO. First in, first out.

Full-duplex. Data can be transmitted in both directions between connected processing elements at the same time.

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.

Header. Typically the first few bytes of a packet, containing control information.

I

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

I/O. Input-output.

IP. Intellectual Property

ITU. International Telecommunication Union.

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.

LP. Link Protocol

LSB. Least significant byte.

LVDS. Low voltage differential signaling.

M

Message passing. An application programming model that allows processing elements to communicate through special hardware instead of through memory as with the globally shared memory programming model.

MSB. Most significant byte.

N

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

O

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

P

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

Payload. The user data embedded in the RapidIO packet.

PCB. Printed circuit board.

PCS. Physical Coding Sublayer.

PMA. Physical Media Attachment.

Port-write. An address-less write operation.

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

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

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

R

Receiver. The RapidIO interface input port on a processing element.

S

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

Semaphore. A technique for coordinating activities in which multiple processing elements compete for the same resource.

Serializer. A device which converts parallel data (such as 8-bit data) to a single bit-wide datastream.

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

SRAM. Static random access memory.

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.

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

W

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

Write port. Hardware within a processing element that is the target of a port-write operation.

Blank page

Blank page

Blank page

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

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.1	First public release	04/06/2001
1.2	Technical changes: incorporate Rev. 1.1 errata rev. 1.1.1, errata 3	06/26/2002
1.3	Technical changes: incorporate Rev 1.2 errata 1 as applicable, the following errata showings: 004-05-00002.002 Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Blank page

List of Figures

List of Figures

Blank page

List of Tables

List of Tables

Blank page

Chapter 1 Overview

1.1 Introduction

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

1.2 Overview

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

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

Chapter 2, “System Exploration and Initialization”

Chapter 3, “RapidIO Device Class Requirements”

Chapter 4, “PCI Considerations”

Chapter 5, “Globally Shared Memory Devices”

Blank page

Chapter 2 System Exploration and Initialization

2.1 Introduction

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

2.2 Boot code access

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

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

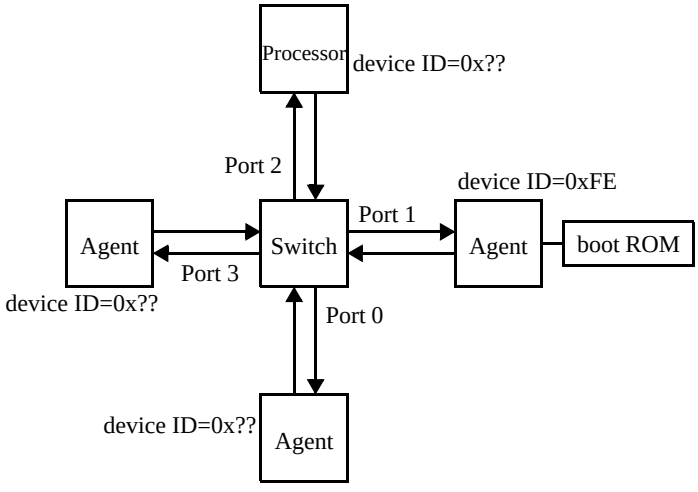


Figure 2-1. Example system with boot ROM

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

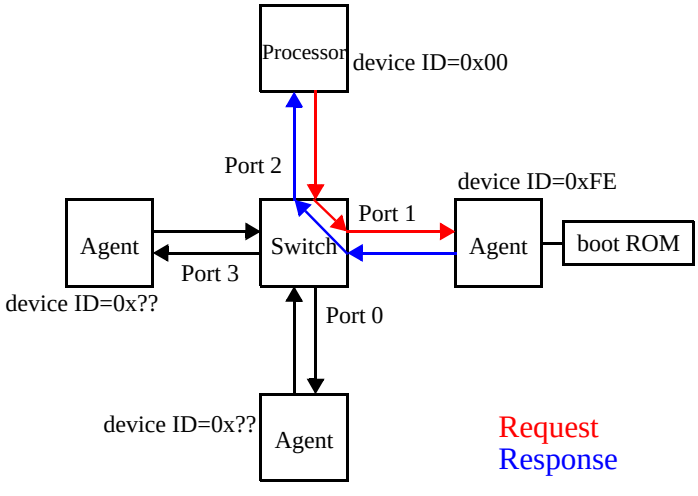


Figure 2-2. Automatically finding the boot ROM

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

2.3 Exploration and initialization

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

2.3.1 Exploration and initialization rules

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

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

2.3.2 Exploration and initialization algorithm

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

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

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

2.3.3 Exploration and initialization example

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

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

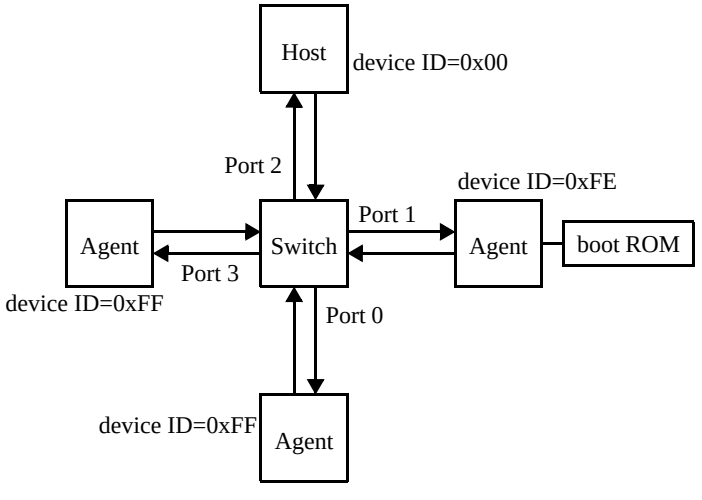


Figure 2-3. Example system

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

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

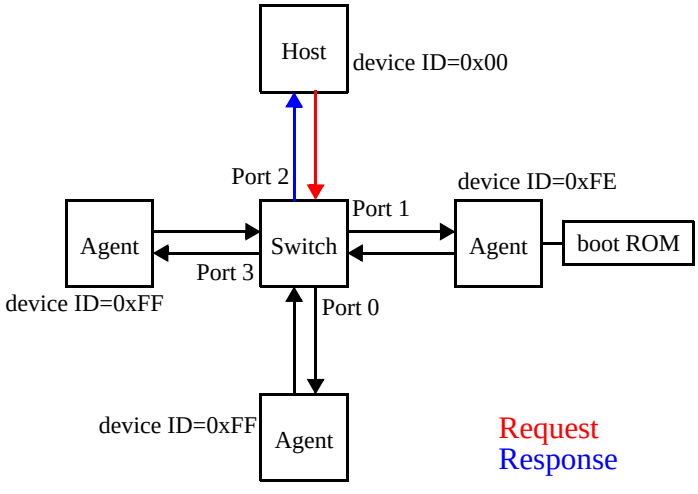


Figure 2-4. Finding the adjacent device

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

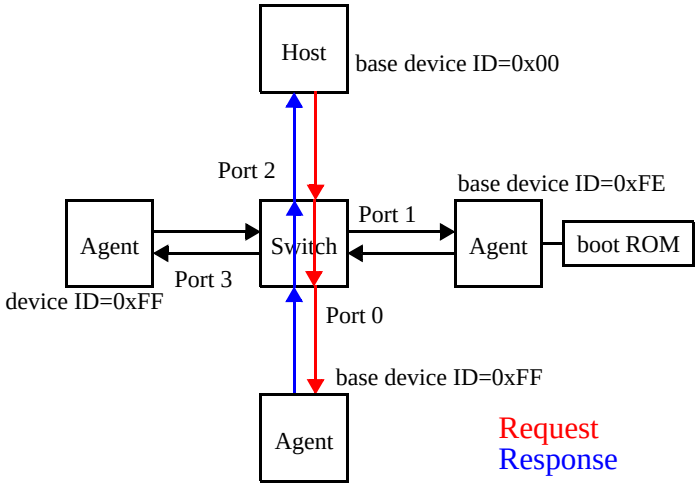


Figure 2-5. Finding the device on switch port 0

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

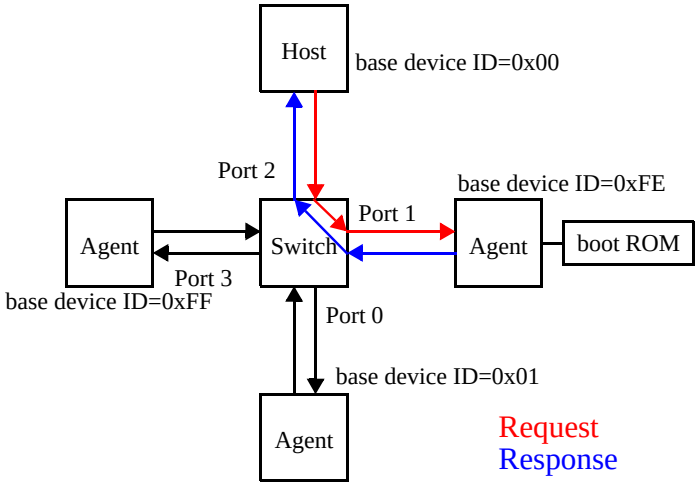


Figure 2-6. Finding the device on switch port 1

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

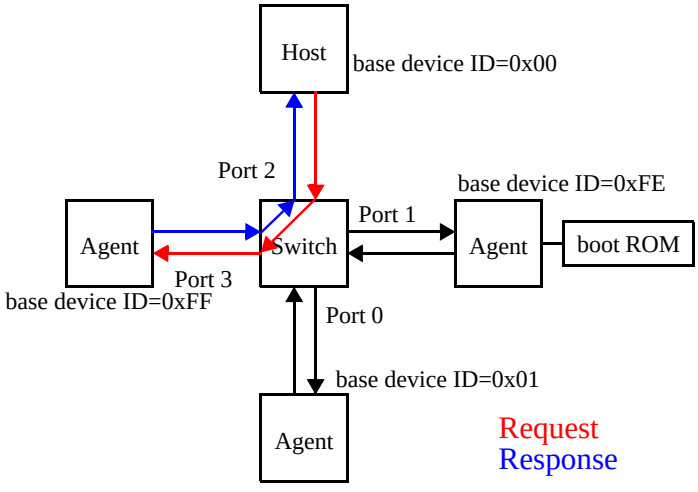


Figure 2-7. Finding the device on switch port 3

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

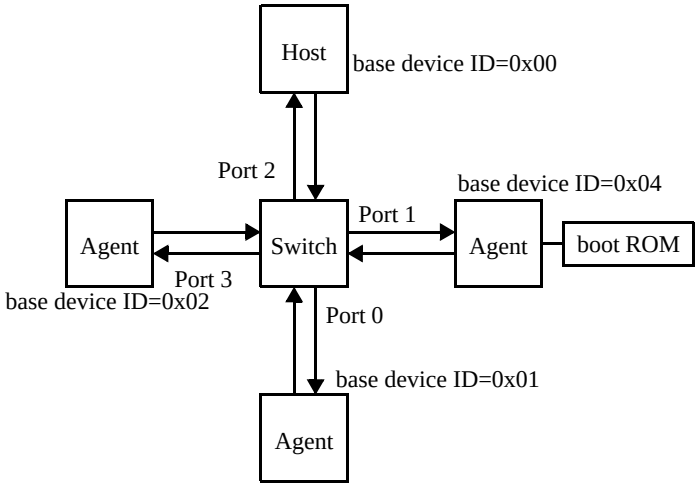


Figure 2-8. Final initialized system state

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

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

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

Chapter 3 RapidIO Device Class Requirements

3.1 Introduction

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

3.2 Class Partitioning

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

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

3.2.1 Generic: All devices

3.2.1.1 General requirements

- One or more 8/16 LP-LVDS and/or 1x/4x LP-Serial ports
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification* and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*)
- Support for small (8-bit) transport device ID fields
  - (refer to *RapidIO Part 3: Common Transport Specification*, Section 1.3)
- Support for recovery from a single corrupt packet or control symbol
  - (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.3.5) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.10.2)
- Support for packet retry protocol

— (refer to *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.2.4) and/or *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.6)

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

3.2.1.2 Operation support as target

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

— (switch targeted by hop_count transport field)

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.3.1,
Section 3.1.10)

3.2.1.3 Operation support as source

• <none>

3.2.2 Class 1: Simple target device

3.2.2.1 General requirements

• all Generic requirements

• Support for 34-bit address packet formats

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 4.4.5)

3.2.2.2 Operation support as target

• all Generic requirements

• Write

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.7)

• Streaming-write

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.2, Section 3.1.8)

• Write-with-response

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.3, Section 3.1.7)

• Read

— (refer to *RapidIO Part 1: Input/Output Logical Specification*, Section 2.2.1, Section 3.1.5)

3.2.2.3 Operation support as source

• all Generic requirements

3.2.3 Class 2: Simple mastering device

3.2.3.1 General requirements

• all Class 1 requirements

3.2.3.2 Operation support as target

• all Class 1 requirements

3.2.3.3 Operation support as source

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

3.2.4 Class 3: Complex mastering device

3.2.4.1 General requirements

- all Class 2 requirements

3.2.4.2 Operation support as target

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

3.2.4.3 Operation support as source

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

Blank page

Chapter 4 PCI Considerations

4.1 Introduction

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

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

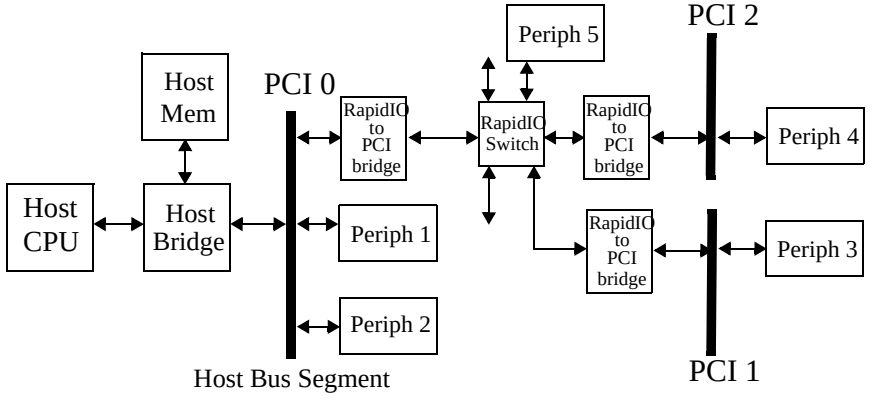


Figure 4-1. Example System with PCI and RapidIO

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

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

4.2 Address Map Considerations

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

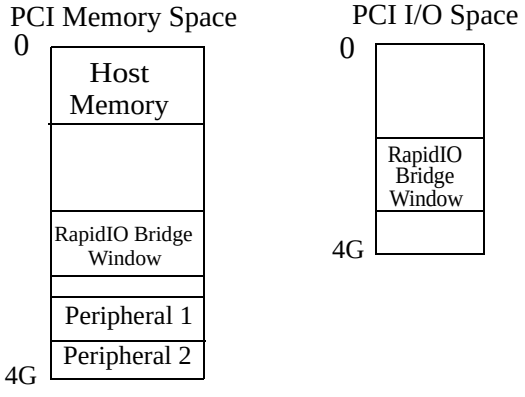


Figure 4-2. Host segment PCI Memory Map Example

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

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

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

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

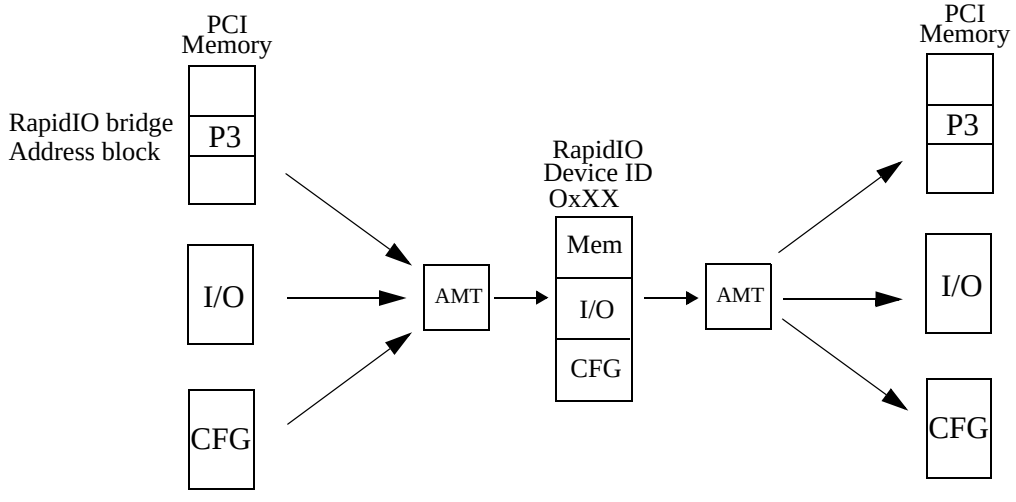


Figure 4-3. AMT and Memory Mapping

4.3 Transaction Flow

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

4.3.1 PCI 2.2 Transaction Flow

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

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

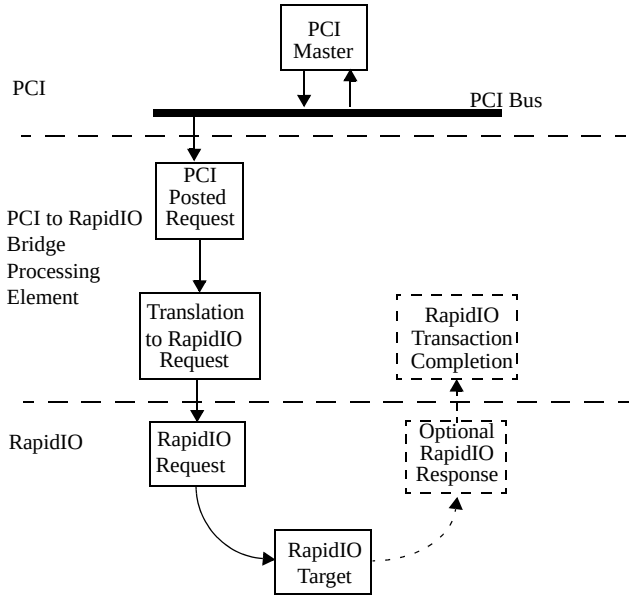


Figure 4-4. PCI Mastered Posted Write Transaction Flow Diagram

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

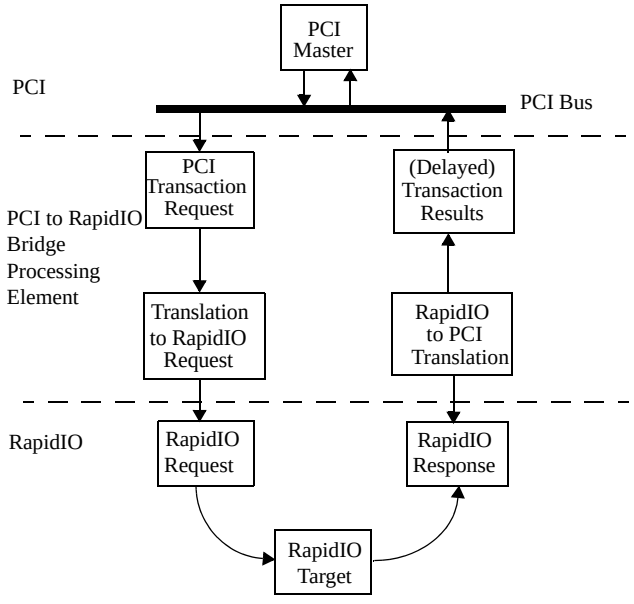


Figure 4-5. PCI Mastered non-posted (delayed) Transaction Flow Diagram

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

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

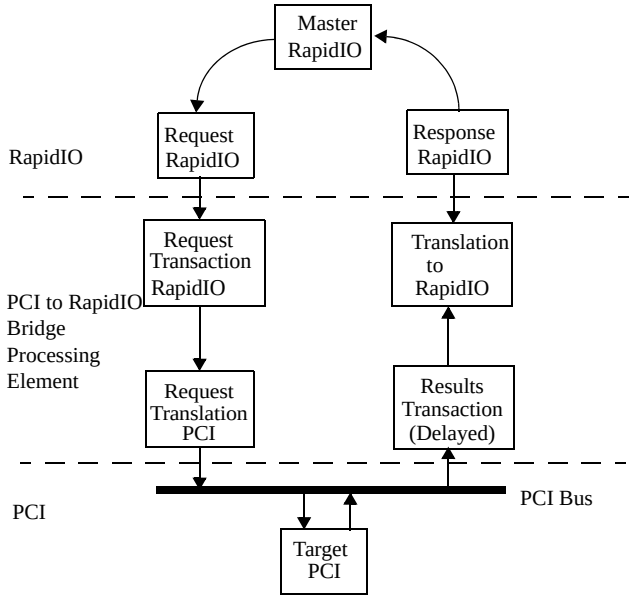


Figure 4-6. RapidIO Mastered Transaction

4.3.2 PCI-X Transaction Flow

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

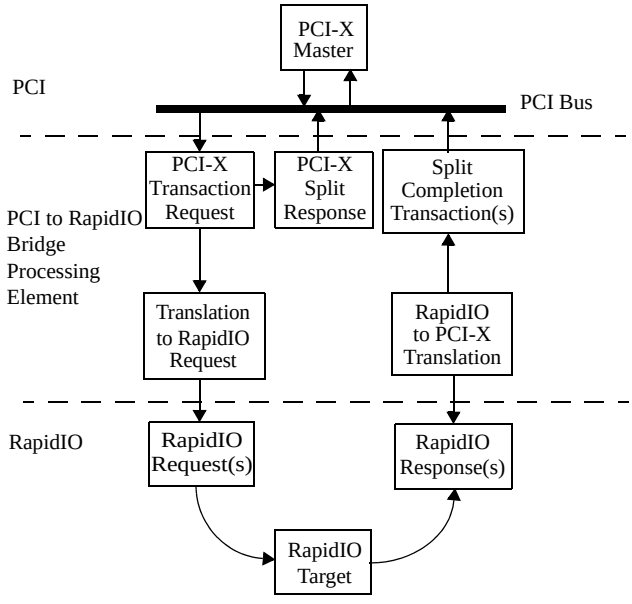


Figure 4-7. PCI-X Mastered Split Response Transaction

4.4 RapidIO to PCI Transaction Mapping

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

Table 4-1. PCI 2.2 to RapidIO Transaction Mapping

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

Table 4-1. PCI 2.2 to RapidIO Transaction Mapping

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

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

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

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

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

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

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

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

4.5 Operation Ordering and Transaction Delivery

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

4.5.1 Operation Ordering

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

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

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

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

4.5.2 Transaction Delivery Ordering

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

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

Table 4-3. Packet priority assignments for PCI ordering

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

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

4.5.3 PCI-X Relaxed Ordering Considerations

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

function.

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

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

4.6 Interactions with Globally Shared Memory

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

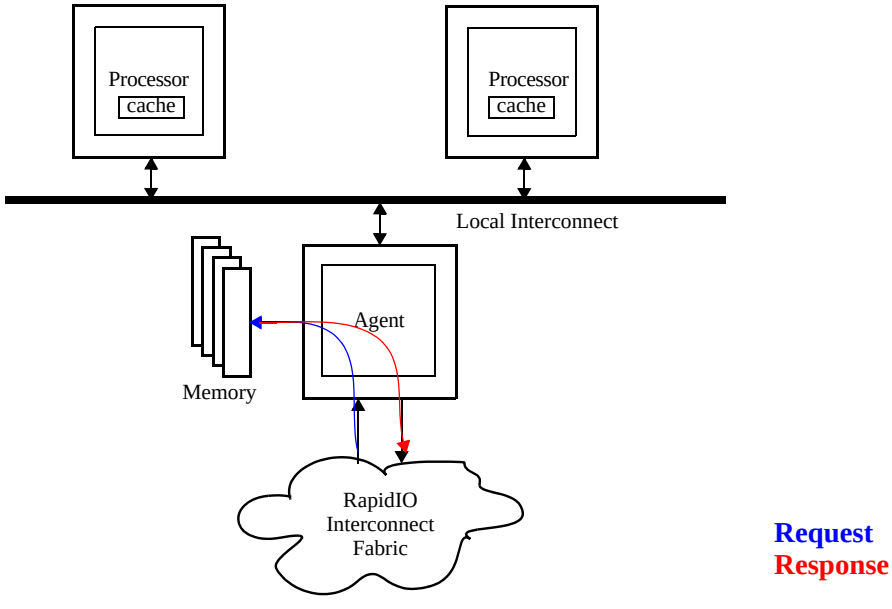


Figure 4-8. Traditional Non-coherent I/O Access Example

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

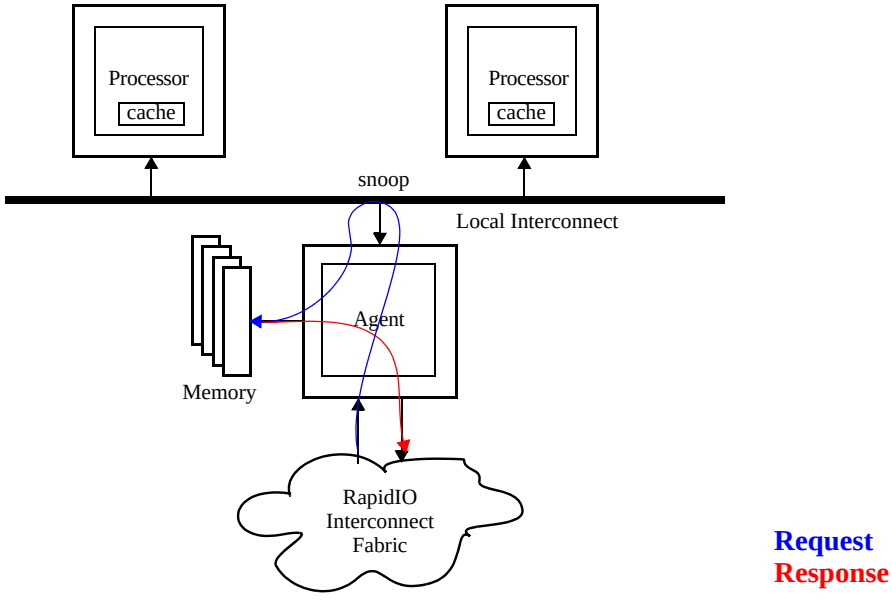


Figure 4-9. Traditional Globally Coherent I/O Access Example

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

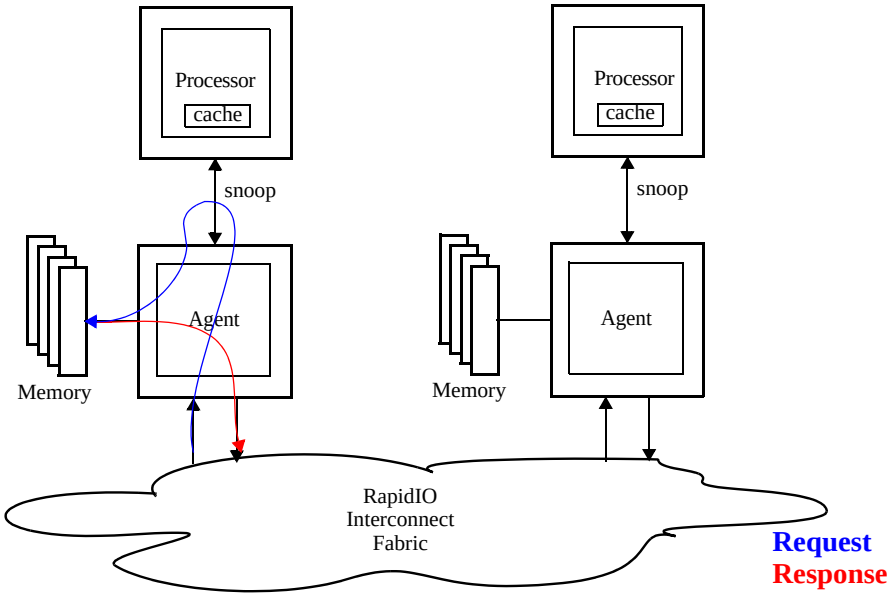


Figure 4-10. RapidIO Locally Coherent I/O Access Example

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

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

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

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

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

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

4.6.1 I/O Read Operation Details

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

4.6.1.1 Internal Request State Machine

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

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

4.6.1.2 Response State Machine

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

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

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

4.6.2 Data Cache Flush Operation Details

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

4.6.2.1 Internal Request State Machine

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

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

4.6.2.2 Response State Machine

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

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

4.7 Byte Lane and Byte Enable Usage

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

4.8 Error Management

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

Blank page

Chapter 5 Globally Shared Memory Devices

5.1 Introduction

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

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

5.2 Processing Element Behavior

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

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

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

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

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

5.2.1 Processor-Memory Processing Element

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

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

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

5.2.1.1 I/O Read Operations

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

5.2.1.1.1 Response State Machine

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

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

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

5.2.1.1.2 External Request State Machine

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

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

5.2.2 Memory-only Processing Element

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

5.2.2.1 Read Operations

This operation is a coherent data cache read.

5.2.2.1.1 Response State Machine

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

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

5.2.2.1.2 External Request State Machine

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

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

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

5.2.2.2 Instruction Read Operations

This operation is a partially coherent instruction cache read.

5.2.2.2.1 Response State Machine

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

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

5.2.2.2.2 External Request State Machine

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

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

5.2.2.3 Read for Ownership Operations

This is the coherent cache store miss operation.

5.2.2.3.1 Response State Machine

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

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

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

5.2.2.3.2 External Request State Machine

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

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

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

5.2.2.4 Data Cache and Instruction Cache Invalidate Operations

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

5.2.2.4.1 Response State Machine

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

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

5.2.2.4.2 External Request State Machine

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

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

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

5.2.2.5 Castout Operations

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

5.2.2.5.1 External Request State Machine

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

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

5.2.2.6 Data Cache Flush Operations

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

5.2.2.6.1 Response State Machine

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

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

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

5.2.2.6.2 External Request State Machine

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

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

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

5.2.2.7 I/O Read Operations

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

5.2.2.7.1 Response State Machine

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

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

5.2.2.7.2 External Request State Machine

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

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

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

5.2.3 Processor-only Processing Element

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

5.2.3.1 Read Operations

This operation is a coherent data cache read.

5.2.3.1.1 Internal Request State Machine

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

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

5.2.3.1.2 Response State Machine

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

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

5.2.3.1.3 External Request State Machine

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

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

5.2.3.2 Instruction Read Operations

This operation is a partially coherent instruction cache read.

5.2.3.2.1 Internal Request State Machine

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

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

5.2.3.2.2 Response State Machine

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

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

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

5.2.3.2.3 External Request State Machine

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

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

5.2.3.3 Read for Ownership Operations

This is the coherent cache store miss operation.

5.2.3.3.1 Internal Request State Machine

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

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

5.2.3.3.2 Response State Machine

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

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

5.2.3.3.3 External Request State Machine

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

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

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

endif;
```

5.2.3.4 Data Cache and Instruction Cache Invalidate Operations

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

5.2.3.4.1 Internal Request State Machine

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

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

5.2.3.4.2 Response State Machine

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

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

5.2.3.4.3 External Request State Machine

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

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

5.2.3.5 Castout Operations

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

5.2.3.5.1 Internal Request State Machine

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

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

5.2.3.5.2 Response State Machine

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

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

5.2.3.6 TLB Invalidate Entry, TLB Invalidate Entry Synchronize Operations

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

5.2.3.6.1 Internal Request State Machine

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

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

5.2.3.6.2 Response State Machine

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

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

5.2.3.6.3 External Request State Machine

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

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

5.2.3.7 Data Cache Flush Operations

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

5.2.3.7.1 Internal Request State Machine

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

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

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

5.2.3.7.2 Response State Machine

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

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

5.2.3.7.3 External Request State Machine

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

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

5.2.3.8 I/O Read Operations

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

5.2.3.8.1 External Request State Machine

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

processor. This may require making further external requests.

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

5.2.4 I/O Processing Element

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

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

5.2.4.1 I/O Read Operations

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

5.2.4.1.1 Internal Request State Machine

This state machine handles requests to remote memory.

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

5.2.4.1.2 Response State Machine

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

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

5.2.4.2 Data Cache Flush Operations

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

5.2.4.2.1 Internal Request State Machine

This state machine handles requests to remote memory.

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

5.2.4.2.2 Response State Machine

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

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

5.2.5 Switch Processing Element

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

5.3 Transaction to Priority Mappings

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

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

Table 5-1. Transaction to Priority Mapping

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

Glossary of Terms and Abbreviations

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

A

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

B

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

C

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

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

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

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

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

D

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

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

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

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

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

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

E

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

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

F

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

G

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

H

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

I

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

I/O. Input-output.

L

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

LVDS. Low voltage differential signaling.

M

Mailbox. Dedicated hardware that receives messages.

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

N

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

O

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

P

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

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

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

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

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

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

R

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

ROM. Read-only memory.

S

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

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

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

Symbol. A 16-bit quantity.

T

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

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

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

Blank page

Blank page

Blank page

RapidIO™ Interconnect Specification Part 8: Error Management Extensions Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.2	First public release	09/13/2002
1.3	Technical changes: the following errata showings: 04-02-00002.001 and the following new features showings: 04-09-00022.002 Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Chapter 1 Error Management Extensions

Chapter 2 Error Management Registers

Table of Contents

Annex A Error Management Discussion (Informative)

List of Tables

List of Tables

Blank page

Chapter 1 Error Management Extensions

1.1 Introduction

The error management extensions describe added requirements in all physical and logical layers. These extensions add definitions to bits that were previously reserved in the Port *n* Control CSR and add new registers that are contained within the Error Management Extended Features Block. This chapter describes the behavior of a device when an error is detected and how the new registers and bits are managed by software and hardware.

1.2 Physical Layer Extensions

The following registers and register bit extensions allow software to monitor and control the reporting of transmission errors:

- (Extensions to the) Port *n* Control CSR defined in Section 2.2
- (Extensions to the) Port *n* Error and Status CSR defined in Section 2.2
- Port-write Target deviceID CSR defined in Section 2.3.2.8
- Port *n* Error Detect CSR defined in Section 2.3.2.10
- Port *n* Error Rate Enable CSR defined in Section 2.3.2.11
- Port *n* Attributes Capture CSR defined in Section 2.3.2.12
- Port *n* Packet/Control Symbol Capture 0 CSR defined in Section 2.3.2.13
- Port *n* Packet Capture 1-3 CSRs defined in Section 2.3.2.14 through Section 2.3.2.16
- Port *n* Error Rate CSR defined in Section 2.3.2.17
- Port *n* Error Rate Threshold CSR defined in Section 2.3.2.18

1.2.1 Port Error Detect, Enable and Capture CSRs

The occurrence of a transmission error shall be logged by hardware by setting the appropriate error indication bit in the Port *n* Error Detect CSR. Transmission errors that are enabled for error capture and error counting will have the corresponding bit set by software in the Port *n* Error Rate Enable CSR. When the Capture Valid Info status bit is not set in the Port *n* Error Capture Attributes CSR, information about the next enabled transmission error shall be saved to the Port *n* Error Capture CSRs. The Info Type and Error Type fields shall be updated and the Capture Valid Info status

bit shall be set by hardware in the Port *n* Error Capture Attributes CSR to lock the error capture registers. The first 16 bytes of the packet header or the 4 bytes of the control symbol that have a detected error are saved in the capture CSRs. Packets smaller than 16 bytes are captured in their entirety. The Port *n* Error Capture CSRs and the Port *n* Error Capture Attributes CSR are not overwritten by hardware with error capture information for subsequent errors until software writes a zero to the Capture Valid Info bit.

The Port *n* Error Detect CSR does not lock so subsequent error indications shall also be logged there by hardware. By reading the register, software may see the types of transmission errors that have occurred. The Port *n* Error Detect CSR is cleared by writing it with all logic 0s.

1.2.2 Error Reporting Thresholds

Transmission errors are normally hidden from system software since they may be recovered with no loss of data and without software intervention. Two thresholds are defined in the Port *n* Error Rate Threshold CSR which can be set to force a report to system software when the link error rate reaches a level that is deemed by the system to be either degraded or unacceptable. The two thresholds are respectively the Degraded Threshold and the Failed Threshold. These thresholds are used as follows.

When the error rate counter is incremented, the Error Rate Degraded Threshold Trigger provides a threshold value that, when equal to or exceeded by the value in the Error Rate Counter in the Port *n* Error Rate register, shall cause the error reporting logic to set the Output Degraded-encountered bit in the Port *n* Error and Status CSR, and notify the system software as described in Section 1.4.

The Error Rate Failed Threshold Trigger, if enabled, shall be larger than the degraded threshold trigger. It provides a threshold value that, when equal to or exceeded by the value in the Error Rate Counter, shall trigger the error reporting logic to set the Output Failed-encountered bit in the Port *n* Error and Status CSR, and notify system software as described in Section 1.4.

No action shall be taken if the Error Rate Counter continues to exceed either threshold value after initial notification when additional errors are detected. No action shall be taken when the Error Rate Counter drops below either threshold.

1.2.3 Error Rate Control and Status

The fields in the Port *n* Error Rate CSR are used to monitor the error rate of the link connected to port *n*.

The Error Rate Bias field determines the rate at which the Error Rate Counter is decremented and defines the acceptable error rate of the link for error reporting purposes. In the absence of additional counted link errors, this mechanism allows the system to recover from both Failed and Degraded levels of operation without a

software reset of the Error Rate Counter. If the link error rate is less than the decrement rate specified in the Error Rate Bias field, the value of the Error Rate counter will rarely be greater than 0x01 or 0x02.

The Error Rate Counter shall increment when a physical layer error is detected whose associated enable bit is set in the Port *n* Error Rate Enable register. The Error Rate Counter shall decrement at the rate specified by the Error Rate Bias field of the Port *n* Error Rate CSR. The Error Rate Counter shall not underflow (shall not decrement when equal to 0x00) and shall not overflow (shall not increment when equal to 0xFF). The incrementing and decrementing of the Error Rate Counter are in no way affected by the values in the Degraded and Failed thresholds. Software may reset the Error Rate Counter at any time.

The Error Rate Recovery field defines how far above the Error Rate Failed Threshold Trigger in the Port *n* Error Rate Threshold Register the Error Rate Counter is allowed to count. In the absence of additional counted errors, this allows software to control the length of time required for the value of the Error Rate Counter to drop below both the Failed and Degraded Thresholds.

The Peak Error Rate field shall contain the largest value encountered by the Error Rate Counter. This field is loaded whenever the current value of the Peak Error Rate field is exceeded by the value of the Error Rate Counter.

1.2.4 Port Behavior When Error Rate Failed Threshold is Reached

The behavior of a port when the Error Rate Counter in the Port *n* Error Rate CSR reaches the Error Rate Failed Threshold and the threshold is enabled depends upon the values of the Stop on Port Failed-encountered Enable and the Drop Packet Enable bits in the Port *n* Control CSR. The Table 1-1 below defines the required behavior.

Table 1-1. Port Behavior when Error Rate Failed Threshold has been hit

Stop on Port Failed Encountered Enable	Drop Packet Enable	Port Behavior	Comments
0	0	The port shall continue to attempt to transmit packets to the connected device if the Output Failed-encountered bit is set and/or if the Error Rate Failed threshold has been met or exceeded.	All devices
0	1	The port shall discard packets that receive a Packet-not-accepted control symbol when the Error Rate Failed Threshold has been met or exceeded. Upon discarding a packet, the port shall set the Output Packet-dropped bit in the Port *n* Error and Status CSR. If the output port "heals", the Error Rate Counter falls below the Error Rate Failed Threshold, the output port shall continue to attempt to forward all packets.	Switch Device Only
1	0	The port shall stop attempting to send packets to the connected device when the Output Failed-encountered bit is set. The output port will congest.	All devices.
1	1	The port shall discard all output packets without attempting to send when the port's Output Failed-encountered bit is set. Upon discarding a packet, the port shall set Output Packet-dropped bit in the Port *n* Error and Status CSR.	All devices.

1.2.5 Packet Timeout Mechanism in a Switch Device

In some systems, it is either desirable or necessary to bound the length of time a packet can remain in a switch. To enable this functionality, a switch shall monitor the length of time each packet accepted by one of its ports has been in the switch. The acceptance of a packet by a port is signaled by the port issuing a packet-accepted control symbol for the packet. The timing begins when the port accepts the packet.

If a packet remains in a switch longer than the Time-to-Live time specified by the Time-to-Live field of the Packet Time-to-live CSR as defined in Section 2.3.2.9, the packet shall be discarded rather than forwarded, the Output Packet-Dropped bit shall be set in the Port *n* Error and Status CSR and the system shall be notified as described in Section 1.4.

1.3 Logical and Transport Layer Extensions

While the RapidIO link may be working properly, an end point processing element may encounter logical or transport layer errors, or other errors unrelated to its RapidIO ports, while trying to complete a transaction. The "ERROR" status response transaction is the mechanism for the target device to indicate to the source that there is a problem completing the request. Experiencing a time-out waiting for a response is also a symptom of an end point or switch fabric with a problem. These

types of errors are logged and reporting enabled with a set of registers that are separate from those used for the Physical Layer errors.

- Logical/Transport Layer Error Detect CSR defined in Section 2.3.2.2
- Logical/Transport Layer Error Enable CSR defined in Section 2.3.2.3
- Logical/Transport Layer Capture CSRs defined in Section 2.3.2.4 to Section 2.3.2.7

1.3.1 Logical/Transport Error Detect, Enable and Capture CSRs

When a logical or transport layer error is detected, the appropriate error bit shall be set by the hardware in the Logical/Transport Layer Error Detect CSR. If the corresponding bit is also set in the Logical/Transport Layer Error Enable CSR, the detect register shall lock, the appropriate information is saved in the Logical/Transport Layer Capture registers, all resources held by the transaction are freed, and system software is notified of the error as described in Section 1.4. If multiple enabled errors occur during the same clock cycle, multiple bits will be set in the detect register and the contents of the Logical/Transport Layer Capture registers are implementation dependent. Once locked, subsequent errors will not set another error detect bit. The contents of the Logical/Transport Capture CSRs are valid if the bitwise AND of the Logical/Transport Layer Error Detect CSR and the Logical/Transport Layer Error Detect Enable CSR is not equal to zero (0x00000000).

Software shall write the Logical/Transport Detect register with all logic 0s to clear the error detect bits or a corresponding enable bit to unlock the register. Any other recovery actions associated with these types of errors are system dependent and outside the scope of this specification.

1.3.2 Message Passing Error Detection

Message passing is a special case of logical layer error recovery requiring error detection at both the source and destination ends of the message. The source of the message has the request-to-response time-out (defined in the Port Response Time-out Control CSR in the RapidIO Physical Layer specifications) to detect lost request or response packets in the switch fabric. However, in order to not hang the recipient mailbox in the case of a lost request packet for a multiple packet message, the recipient mailbox shall have an analogous response-to-request time-out. This time-out is for sending a response packet to receiving the next request packet of a given message operation, and has the same value as the request-to-response time-out that is already specified. The Logical/Transport Layer Control Capture CSR contains the ‘msg info’ field to capture the critical information of the last received (or sent) message segment before time-out.

1.4 System Software Notification of Error

System software is notified of logical, transport, and physical layer errors in two ways. An interrupt is issued to the local system by a device, the method of which is not defined in this specification, or a Maintenance port-write operation is issued by a device. Maintenance port-write operations are sent to a predetermined system host (defined in the Port-write Target deviceID CSR in Section 2.3.2.8). The sending device sets the Port-write Pending status bit in the Port *n* Error and Status CSR. A 16 byte data payload of the Maintenance Port-write packet contains the contents of several CSRs, the port on the device that encountered the error condition (for port-based errors), and some optional implementation specific additional information as shown in Table 1-2. Software indicates that it has seen the port-write operation by clearing the Port-write Pending status bit.

The Component Tag CSR is defined in the *RapidIO Part 3: Common Transport Specification*, and is used to uniquely identify the reporting device within the system. A Port ID field, the Logical/Transport Layer Detect CSR defined in Section 2.3.2.2, and the Port *n* Error Detect CSR defined in Section 2.3.2.10 are used to describe the encountered error condition.

Table 1-2. Port-write Packet Data Payload for Error Reporting

Data Payload Byte Offset	Word	
0x0	Component Tag CSR	
0x4	Port *n* Error Detect CSR	
0x8	Implementation specific	Port ID (byte)
0xC	Logical/Transport Layer Error Detect CSR	

1.5 Mechanisms for Software Debug

In most systems, it is difficult to verify the error handling software. The Error management extensions make some registers writable for easier debug.

The Logical/Transport Layer Error Detect register and the Logical/Transport Layer Error Capture registers are writable by software to allow software debug of the system error recovery mechanisms. For software debug, software must write the Logical/Transport Layer Error capture registers with the desired address and device id information then write the Logical/Transport Layer Error Detect register to set an error bit and lock the registers. When an error detect bit is set, the hardware will inform the system software of the error using its standard error reporting mechanism. After the error has been reported, the system software may read and clear registers as necessary to complete its error handling protocol testing.

The Port *n* Error Detect register and the Port *n* Error Capture registers are also

writable by software to allow software debug of the system error recovery and thresholding mechanism. For debug, software must write the Port *n* Attributes Error Capture CSR to set the Capture Valid Info bit and then the packet/control symbol information in the other capture registers. Each write of a non-zero value to the Port *n* Error Detect CSR shall cause the Error Rate Counter to increment if the corresponding error bit is enabled in the Port *n* Error Rate Enable CSR. When a threshold is reached, the hardware will inform the system software of the error using its standard error reporting mechanism. After the error has been reported, the system software may read and clear registers as necessary to complete its error handling protocol testing.

Blank page

Chapter 2 Error Management Registers

2.1 Introduction

This section describes the Error Management Extended Features block, and adds a number of new bits to the existing standard physical layer registers. ‘End-point only’ and ‘switch only’ register bits shall be considered reserved when the registers are implemented on devices for which these bits are not required.

2.2 Additions to Existing Registers

The following bits are added to the parallel and serial logical layer specification Port *n* Control CSRs.

Table 2-1. Bit Settings for Port *n* Control CSRs

Bit	Name	Reset Value	Description
28	Stop on Port Failed-encountered Enable	0b0	This bit is used with the Drop Packet Enable bit to force certain behavior when the Error Rate Failed Threshold has been met or exceeded. See Section 1.2.4 of the Part 8: Error Management Extensions for detailed requirements.
29	Drop Packet Enable	0b0	This bit is used with the Stop on Port Failed-encountered Enable bit to force certain behavior when the Error Rate Failed Threshold has been met or exceeded. See Section 1.2.4 of the Part 8: Error Management Extensions for detailed requirements.
30	Port Lockout	0b0	When this bit is cleared, the packets that may be received and issued are controlled by the state of the Output Port Enable and Input Port Enable bits in the Port *n* Control CSR. When this bit is set, this port is stopped and is not enabled to issue or receive any packets; the input port can still follow the training procedure and can still send and respond to link-requests; all received packets return packet-not-accepted control symbols to force an error condition to be signaled by the sending device

The following bits are added to the parallel and serial specification Port *n* Error and Status CSRs.

Table 2-2. Bit Settings for Port *n* Error and Status CSRs

Bit	Name	Reset Value	Description
5	Output Packet-dropped	0b0	Output port has discarded a packet. Once set remains set until written with a logic 1 to clear.
6	Output Failed-encountered	0b0	Output port has encountered a failed condition, meaning that the port's failed error threshold has been reached in the Port *n* Error Rate Threshold register. Once set remains set until written with a logic 1 to clear.
7	Output Degraded-encountered	0b0	Output port has encountered a degraded condition, meaning that the port's degraded error threshold has been reached in the Port *n* Error Rate Threshold register. Once set remains set until written with a logic 1 to clear.

2.3 New Error Management Registers

This section describes the Extended Features block (EF_ID=0h0007) that allows an external processing element to manage the error status and reporting for a processing element. This chapter only describes registers or register bits defined by this extended features block. All registers are 32-bits and aligned to a 32-bit boundary.

Table 2-3 describes the required behavior for accesses to reserved register bits and reserved registers for the RapidIO Extended Features register space,

Table 2-3. Extended Feature Space Reserved Access Behavior

Byte Offset	Space Name	Item	Initiator behavior	Target behavior
0x100–FFFC	Extended Features Space	Reserved bit	read - ignore returned value[1]	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

[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.

2.3.1 Register Map

Table 2-4 shows the register map for the error management registers. This register map is currently only defined for devices with up to 16 RapidIO ports, but can be extended or shortened if more or less port definitions are required for a device. For example, a device with four RapidIO ports is only required to use register map space corresponding to offsets [EF_PTR+0x00] through [EF_PTR+0x13C]. Register map offset [EF_PTR+0x140] can be used for another Extended Features block.

Table 2-4. Error Management Extensions Register Map

	Block Byte Offset	Register Name
General	0x0	Error Management Extensions Block Header
	0x4	Reserved
	0x8	Logical/Transport Layer Error Detect CSR
	0xC	Logical/Transport Layer Error Enable CSR
	0x10	Logical/Transport Layer High Address Capture CSR
	0x14	Logical/Transport Layer Address Capture CSR
	0x18	Logical/Transport Layer Device ID Capture CSR
	0x1C	Logical/Transport Layer Control Capture CSR
	0x20-24	Reserved
	0x28	Port-write Target deviceID CSR
	0x2C	Packet Time-to-live CSR
	0x30-3C	Reserved
Port 0	0x40	Port 0 Error Detect CSR
	0x44	Port 0 Error Rate Enable CSR
	0x48	Port 0 Attributes Capture CSR
	0x4C	Port 0 Packet/Control Symbol Capture 0 CSR
	0x50	Port 0 Packet Capture 1 CSR
	0x54	Port 0 Packet Capture 2 CSR
	0x58	Port 0 Packet Capture 3 CSR
	0x5C-64	Reserved
	0x68	Port 0 Error Rate CSR
	0x6C	Port 0 Error Rate Threshold CSR
	0x70-7C	Reserved

Table 2-4. Error Management Extensions Register Map

	Block Byte Offset	Register Name
Port 1	0x80	Port 1 Error Detect CSR
	0x84	Port 1 Error Rate Enable CSR
	0x88	Port 1 Attributes Capture CSR
	0x8C	Port 1 Packet/Control Symbol Capture 0 CSR
	0x90	Port 1 Packet Capture 1 CSR
	0x94	Port 1 Packet Capture 2 CSR
	0x98	Port 1 Packet Capture 3 CSR
	0x9C-A4	Reserved
	0xA8	Port 1 Error Rate CSR
	0xAC	Port 1 Error Rate Threshold CSR
	0xB0-BC	Reserved
Ports 2-14	0xC0–3FC	Assigned to Port 2-14 CSRs
Port 15	0x400	Port 15 Error Detect CSR
	0x404	Port 15 Error Rate Enable CSR
	0x408	Port 15 Attributes Capture CSR
	0x40C	Port 15 Packet/Control Symbol Capture 0 CSR
	0x410	Port 15 Packet Capture 1 CSR
	0x414	Port 15 Packet Capture 2 CSR
	0x418	Port 15 Packet Capture 3 CSR
	0x41C-424	Reserved
	0x428	Port 15 Error Rate CSR
	0x42C	Port 15 Error Rate Threshold CSR
	0x430-43C	Reserved

2.3.2 Command and Status Registers (CSRs)

Refer to Table 2-3 for the required behavior for access to reserved registers and register bits.

2.3.2.1 Error Management Extensions Block Header (Block Offset 0x0)

The error management extensions block header register contains the EF_PTR to the next EF_BLK and the EF_ID that identifies this as the error management extensions block header.

Table 2-5. Bit Settings for Error Management Extensions Block Header

Bit	Name	Reset Value	Description
0-15	EF_PTR		Hard wired pointer to the next block in the data structure, if one exists
16-31	EF_ID	0x0007	Hard wired Extended Features ID

2.3.2.2 Logical/Transport Layer Error Detect CSR (Block Offset 0x08)

This register indicates the error that was detected by the Logical or Transport logic layer. Multiple bits may get set in the register if simultaneous errors are detected during the same clock cycle that the errors are logged.

Table 2-6. Bit Settings for Logical/Transport Layer Error Detect CSR

Bit	Name	Reset Value	Description
0	IO error response	0b0	Received a response of 'ERROR' for an IO Logical Layer Request. (end point device only)
1	Message error response	0b0	Received a response of 'ERROR' for an MSG Logical Layer Request. (end point device only)
2	GSM error response	0b0	Received a response of 'ERROR' for a GSM Logical Layer Request. (end point device only)
3	Message Format Error	0b0	Received MESSAGE packet data payload with an invalid size or segment (MSG logical) (end point device only)
4	Illegal transaction decode	0b0	Received illegal fields in the request/response packet for a supported transaction (IO/MSG/GSM logical) (switch or endpoint device)
5	Illegal transaction target error	0b0	Received a packet that contained a destination ID that is not defined for this end point. End points with multiple ports and a built-in switch function may not report this as an error (Transport) (end point device only)
6	Message Request Time-out	0b0	A required message request has not been received within the specified time-out interval (MSG logical) (end point device only)
7	Packet Response Time-out	0b0	A required response has not been received within the specified time out interval (IO/MSG/GSM logical) (end point device only)
8	Unsolicited Response	0b0	An unsolicited/unexpected Response packet was received (IO/MSG/GSM logical; only Maintenance response for switches) (switch or endpoint device)

Table 2-6. Bit Settings for Logical/Transport Layer Error Detect CSR

Bit	Name	Reset Value	Description
9	Unsupported Transaction	0b0	A transaction is received that is not supported in the Destination Operations CAR (IO/MSG/GSM logical; only Maintenance port-write for switches) (switch or endpoint device)
10-23	—		Reserved
24-31	Implementation Specific error	0x00	An implementation specific error has occurred. (switch or end point device)

2.3.2.3 Logical/Transport Layer Error Enable CSR (Block Offset 0x0C)

This register contains the bits that control if an error condition locks the Logical/Transport Layer Error Detect and Capture registers and is reported to the system host.

Table 2-7. Bit Settings for Logical/Transport Layer Error Enable CSR

Bit	Name	Reset Value	Description
0	IO error response enable	0b0	Enable reporting of an IO error response. Save and lock original request transaction information in all Logical/Transport Layer Capture CSRs. (end point device only)
1	Message error response enable	0b0	Enable reporting of a Message error response. Save and lock original request transaction information in all Logical/Transport Layer Capture CSRs. (end point device only)
2	GSM error response enable	0b0	Enable reporting of a GSM error response. Save and lock original request transaction capture information in all Logical/Transport Layer Capture CSRs. (end point device only)
3	Message Format Error enable	0b0	Enable reporting of a message format error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs. (end point device only)
4	Illegal transaction decode enable	0b0	Enable reporting of an illegal transaction decode error Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs. (switch or end-point device)
5	Illegal transaction target error enable	0b0	Enable reporting of an illegal transaction target error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs. (end point device only)
6	Message Request time-out enable	0b0	Enable reporting of a Message Request time-out error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs for the last Message request segment packet received. (end point device only)
7	Packet Response Time-out error enable	0b0	Enable reporting of a packet response time-out error. Save and lock original request address in Logical/Transport Layer Address Capture CSRs. Save and lock original request Destination ID in Logical/Transport Layer Device ID Capture CSR. (end point device only)

Table 2-7. Bit Settings for Logical/Transport Layer Error Enable CSR

Bit	Name	Reset Value	Description
8	Unsolicited Response error enable	0b0	Enable reporting of an unsolicited response error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs. (switch or end-point device)
9	Unsupported Transaction error enable	0b0	Enable reporting of an unsupported transaction error. Save and lock transaction capture information in Logical/Transport Layer Device ID and Control Capture CSRs. (switch or end-point device)
10-23	—		Reserved
24-31	Implementation Specific error enable	0x00	Enable reporting of an implementation specific error has occurred. Save and lock capture information in appropriate Logical/Transport Layer Capture CSRs.

2.3.2.4 Logical/Transport Layer High Address Capture CSR (Block Offset 0x10)

This register contains error information. It is locked when a Logical/Transport error is detected and the corresponding enable bit is set. This register is only required for end point devices that support 66 or 50 bit addresses.

Table 2-8. Bit Settings for Logical/Transport Layer High Address Capture CSR

Bit	Name	Reset Value	Description
0-31	address[0-31]	All 0s	Most significant 32 bits of the address associated with the error (for requests, for responses if available)

2.3.2.5 Logical/Transport Layer Address Capture CSR (Block Offset 0x14)

This register contains error information. It is locked when a Logical/Transport error is detected and the corresponding enable bit is set.

Table 2-9. Bit Settings for Logical/Transport Layer Address Capture CSR

Bit	Name	Reset Value	Description
0-28	address[32-60]	All 0s	Least significant 29 bits of the address associated with the error (for requests, for responses if available)
29	—		Reserved
30-31	xamsbs	0b00	Extended address bits of the address associated with the error (for requests, for responses if available)

2.3.2.6 Logical/Transport Layer Device ID Capture CSR (Block Offset 0x18)

This register contains error information. It is locked when an error is detected and the corresponding enable bit is set.

Table 2-10. Bit Settings for Logical/Transport Layer Device ID Capture CSR

Bit	Name	Reset Value	Description
0-7	MSB destinationID	0x00	Most significant byte of the destinationID associated with the error (large transport systems only)
8-15	destinationID	0x00	The destinationID associated with the error
16-23	MSB sourceID	0x00	Most significant byte of the sourceID associated with the error (large transport systems only)
24-31	sourceID	0x00	The sourceID associated with the error

2.3.2.7 Logical/Transport Layer Control Capture CSR (Block Offset 0x1C)

This register contains error information. It is locked when a Logical/Transport error is detected and the corresponding enable bit is set.

Table 2-11. Bit Settings for Logical/Transport Layer Control Capture CSR

Bit	Name	Reset Value	Description
0-3	ftype	0x0	Format type associated with the error
4-7	ttype	0x0	Transaction type associated with the error
8-15	msg info	0x00	letter, mbox, and msgseg for the last Message request received for the mailbox that had an error (Message errors only)
16-31	Implementation specific	0x0000	Implementation specific information associated with the error

2.3.2.8 Port-write Target deviceID CSR (Block Offset 0x28)

This register contains the target deviceID to be used when a device generates a Maintenance port-write operation to report errors to a system host.

Table 2-12. Bit Settings for Port-write Target deviceID CSR

Bit	Name	Reset Value	Description
0-7	deviceID_msb	0x00	This is the most significant byte of the port-write target deviceID (large transport systems only)
8-15	deviceID	0x00	This is the port-write target deviceID
16	large_transport	0b0	deviceID size to use for a port-write 0b0 - use the small transport deviceID 0b1 - use the large transport deviceID
17-31	—		Reserved

2.3.2.9 Packet Time-to-live CSR (Block Offset 0x2C)

The Packet Time-to-live register specifies the length of time that a packet is allowed to exist within a switch device. The maximum value of the Time-to-live variable (0xFFFF) shall correspond to 100 msec. +/-34%. The resolution (minimum step size) of the Time-to-live variable shall be (maximum value of Time-to-live)/(2^{16}-1). The reset value is all logic 0s, which disables the Time-to-live function so that a packet never times out. This register is not required for devices without switch functionality.

Table 2-13. Bit Settings for Packet Time-to-live CSR

Bit	Name	Reset Value	Description
0-15	Time-to-live value	0x0000	Maximum time that a packet is allowed to exist within a switch device
16-31	—		Reserved

2.3.2.10 Port *n* Error Detect CSR (Block Offset 0x40, 80,..., 400)

The Port *n* Error Detect Register indicates transmission errors that are detected by the hardware.

Table 2-14. Bit Settings for Port *n* Error Detect CSR

Bit	Name	Reset Value	Description
0	Implementation specific error	0b0	An implementation specific error has been detected
1-7	—		Reserved
8	Received S-bit error	0b0	Received a packet/control symbol with an S-bit parity error (parallel)
9	Received corrupt control symbol	0b0	Received a control symbol with a bad CRC value (serial) Received a control symbol with a true/complement mismatch (parallel)
10	Received acknowledge control symbol with unexpected ackID	0b0	Received an acknowledge control symbol with an unexpected ackID (packet-accepted or packet_retry)
11	Received packet-not-accepted control symbol	0b0	Received packet-not-accepted acknowledge control symbol
12	Received packet with unexpected ackID	0b0	Received packet with unexpected ackID value - out-of-sequence ackID
13	Received packet with bad CRC	0b0	Received packet with a bad CRC value
14	Received packet exceeds 276 Bytes	0b0	Received packet which exceeds the maximum allowed size
15-25	—		Reserved
26	Non-outstanding ackID	0b0	Link_response received with an ackID that is not outstanding
27	Protocol error	0b0	An unexpected packet or control symbol was received
28	Frame toggle edge error	0b0	FRAME signal toggled on falling edge of receive clock (parallel)

Table 2-14. Bit Settings for Port *n* Error Detect CSR

Bit	Name	Reset Value	Description
29	Delineation error	0b0	FRAME signal toggled on non-32-bit boundary (parallel) Received unaligned /SC/ or /PD/ or undefined code-group (serial)
30	Unsolicited acknowledge control symbol	0b0	An unexpected acknowledge control symbol was received
31	Link time-out	0b0	An acknowledge or link-response control symbol is not received within the specified time-out interval

2.3.2.11 Port *n* Error Rate Enable CSR (Block Offset 0x44, 84,..., 404)

This register contains the bits that control when an error condition is allowed to increment the error rate counter in the Port *n* Error Rate Threshold Register and lock the Port *n* Error Capture registers.

Table 2-15. Bit Settings for Port *n* Error Rate Enable CSR

Bit	Name	Reset Value	Description
0	Implementation specific error enable	0b0	Enable error rate counting of implementation specific errors
1-7	—		Reserved
8	Received S-bit error enable	0b0	Enable error rate counting of a packet/control symbol with an S-bit parity error (parallel)
9	Received control symbol with bad CRC enable	0b0	Enable error rate counting of a corrupt control symbol
10	Received out-of-sequence acknowledge control symbol enable	0b0	Enable error rate counting of an acknowledge control symbol with an unexpected ackID
11	Received packet-not-accepted control symbol enable	0b0	Enable error rate counting of received packet-not-accepted control symbols
12	Received packet with unexpected ackID enable	0b0	Enable error rate counting of packet with unexpected ackID value - out-of-sequence ackID
13	Received packet with Bad CRC enable	0b0	Enable error rate counting of packet with a bad CRC value
14	Received packet exceeds 276 Bytes enable	0b0	Enable error rate counting of packet which exceeds the maximum allowed size
15-25	—		Reserved
26	Non-outstanding ackID enable	0b0	Enable error rate counting of link-responses received with an ackID that is not outstanding
27	Protocol error enable	0b0	Enable error rate counting of protocol errors
28	Frame toggle edge error enable	0b0	Enable error rate counting of frame toggle edge errors

Table 2-15. Bit Settings for Port *n* Error Rate Enable CSR

Bit	Name	Reset Value	Description
29	Delineation error	0b0	Enable error rate counting of delineation errors
30	Unsolicited acknowledge control symbol	0b0	Enable error rate counting of unsolicited acknowledge control symbol errors
31	Link time-out	0b0	Enable error rate counting of link time-out errors

2.3.2.12 Port *n* Attributes Capture CSR (Block Offset 0x48, 88,..., 408)

The error capture attribute register indicates the type of information contained in the Port *n* error capture registers. In the case of multiple detected errors during the same clock cycle one of the errors must be reflected in the Error type field. The error that is reflected is implementation dependent.

Table 2-16. Bit Settings for Port *n* Attributes Capture CSR

Bit	Name	Reset Value	Description
0-1	Info type	0b00	Type of information logged 00 - packet 01 - control symbol (only error capture register 0 is valid) 10 - implementation specific (capture register contents are implementation specific) 11 - undefined (S-bit error), capture as if a packet (parallel physical layer only)
2	—		Reserved
3-7	Error type	0x00	The encoded value of the bit in the Port *n* Error Detect CSR that describes the error captured in the Port *n* Error Capture CSRs.
8-27	Implementation Dependent	All 0s	Implementation Dependent Error Information
28-30	—		Reserved
31	Capture valid info	0b0	This bit is set by hardware to indicate that the Packet/control symbol capture registers contain valid information. For control symbols, only capture register 0 will contain meaningful information.

2.3.2.13 Port *n* Packet/Control Symbol Capture 0 CSR (Block Offset 0x4C, 8C,..., 40C)

Captured control symbol information includes the true and complement of the control symbol. This is exactly what arrives on the RapidIO interface with bits 0-7 of the capture register containing the least significant byte of the 32-bit quantity. This register contains the first 4 bytes of captured packet symbol information.

Table 2-17. Bit Settings for Port *n* Packet/Control Symbol Capture 0 CSR

Bit	Name	Reset Value	Description
0-31	Capture 0	All 0s	True and Complement of Control Symbol (parallel) or Control Character and Control Symbol (serial) or Bytes 0 to 3 of Packet Header

2.3.2.14 Port *n* Packet Capture 1 CSR (Block Offset 0x50, 90,..., 410)

Error capture register 1 contains bytes 4 through 7 of the packet header.

Table 2-18. Bit Settings for Port *n* Packet Capture 1 CSR

Bit	Name	Reset Value	Description
0-31	Capture 1	All 0s	Bytes 4 thru 7 of the packet header.

2.3.2.15 Port *n* Packet Capture 2 CSR (Block Offset 0x54, 94,..., 414)

Error capture register 2 contains bytes 8 through 11 of the packet header.

Table 2-19. Bit Settings for Port *n* Packet Capture 2 CSR

Bit	Name	Reset Value	Description
0-31	Capture 2	All 0s	Bytes 8 thru 11of the packet header.

2.3.2.16 Port *n* Packet Capture 3 CSR (Block Offset 0x58, 98,..., 418)

Error capture register 3 contains bytes 12 through 15 of the packet header.

Table 2-20. Bit Settings for Port *n* Packet Capture 3 CSR

Bit	Name	Reset Value	Description
0-31	Capture 3	All 0s	Bytes 12 thru 15 of the packet header.

2.3.2.17 Port *n* Error Rate CSR (Block Offset 0x68, A8,..., 428)

The Port *n* Error Rate register is a 32-bit register used with the Port *n* Error Rate Threshold register to monitor and control the reporting of transmission errors, shown in Table 2-21.

Table 2-21. Bit Settings for Port *n* Error Rate CSR

Bit	Name	Reset Value	Description
0-7	Error Rate Bias	0x80	These bits provide the error rate bias value 0x00 - do not decrement the error rate counter 0x01 - decrement every 1ms (+/-34%) 0x02 - decrement every 10ms (+/-34%) 0x04 - decrement every 100ms (+/-34%) 0x08 - decrement every 1s (+/-34%) 0x10 - decrement every 10s (+/-34%) 0x20 - decrement every 100s (+/-34%) 0x40 - decrement every 1000s (+/-34%) 0x80 - decrement every 10000s (+/-34%) other values are reserved
8-13	—		Reserved
14-15	Error Rate Recovery	0b00	These bits limit the incrementing of the error rate counter above the failed threshold trigger. 0b00 - only count 2 errors above 0b01 - only count 4 errors above 0b10 - only count 16 error above 0b11 - do not limit incrementing the error rate count
16-23	Peak Error Rate	0x00	This field contains the peak value attained by the error rate counter.
24-31	Error Rate Counter	0x00	These bits maintain a count of the number of transmission errors that have been detected by the port, decremented by the Error Rate Bias mechanism, to create an indication of the link error rate.

2.3.2.18 Port *n* Error Rate Threshold CSR (Block Offset 0x6C, AC,..., 42C)

The Port *n* Error Rate Threshold register is a 32-bit register used to control the reporting of the link status to the system host.

Table 2-22. Bit Settings for Port *n* Error Rate Threshold CSR

Bit	Name	Reset Value	Description
0-7	Error Rate Failed Threshold Trigger	0xFF	These bits provide the threshold value for reporting an error condition due to a possibly broken link. 0x00 - Disable the Error Rate Failed Threshold Trigger 0x01 - Set the error reporting threshold to 1 0x02 - Set the error reporting threshold to 2 ... 0xFF - Set the error reporting threshold to 255

Table 2-22. Bit Settings for Port *n* Error Rate Threshold CSR

Bit	Name	Reset Value	Description
8-15	Error Rate Degraded Threshold Trigger	0xFF	These bits provide the threshold value for reporting an error condition due to a degrading link. 0x00 - Disable the Error Rate Degraded Threshold Trigger 0x01 - Set the error reporting threshold to 1 0x02 - Set the error reporting threshold to 2 ... 0xFF - Set the error reporting threshold to 255
16-31	—		Reserved

Annex A Error Management Discussion (Informative)

A.1 Introduction

This section is intended to provide useful information/background on the application of the error management capabilities. This section is a guideline, not part of the specification.

A.2 Limitations of Error Management Discussion

The RapidIO hardware that implements the Error Management extensions is able to log transmission errors and errors that occur at a higher level. Some error scenarios require no software intervention and recovery procedures are done totally by the hardware.

Some error scenarios detected require fault management software for recovery to be successful. For example, some types of logical layer errors on a Read or Write operation may be recoverable by killing the software process using the affected memory space and removing the memory space from the available system resource pool. It may also be possible for software to retry the operation, possibly through a different path in the switch fabric. Since such fault management software is typically tightly coupled to a particular system and/or implementation, it is considered outside of the scope of this specification.

Another area of fault recovery that requires fault management software to be implemented is correcting of system state after an error during an atomic operation. The swap style Atomic operations are possibly recoverable through software and require software convention to uniquely identify attempts to take locks. For example, if the request is lost and times out, software can examine the current lock value to determine if the request or the associated response was the transaction that was lost in the switch fabric. For all other Atomic operations (such as the Atomic set operation), it is impossible to correct the system state in the presence of a ‘lost packet’ type of error.

The use of RapidIO message packets relies on the use of higher layer protocols for error management. Since end points that communicate via messaging are typically running a variety of higher layer protocols, error reporting of both request and response time-outs is done locally by the message queue management controller.

Note that side effect errors can occur, for example, ERROR responses or RETRY responses during an active (partially completed) message, which may complicate the recovery procedure. The recovery strategies for messages lost in this manner are outside of the scope of this specification.

Globally Shared Memory systems that encounter a logical or transport layer error are typically not recoverable by any mechanism as this usually means that the processor caches are no longer coherent with the main memory system. Historically, recovery from such errors requires a complete reboot of the machine after the component that caused the error is repaired or replaced.

A.3 Hot-insertion/extraction Discussion

Hot-insertion can be regarded as an error condition in which a new part of the system is detected, therefore, hot-insertion of a Field Replaceable Unit (FRU) can be handled utilizing the above described mechanisms. This section describes two approaches for hot insertion. The first generally applies to high availability systems, or systems where FRUs need to brought into the system in a controlled manner. The second generally applies to systems where availability is less of a concern, for example, a trusted system or a system without a system host.

At system boot time, the system host identifies all of the unattached links in the machine through system discovery and puts them in a locked mode, whereby all incoming packets are to be rejected, leaving the drivers and receivers enabled. This is done by setting the Discovered bit in the Port General Control CSR and the Port Lockout bit in the Port *n* Control CSR. Note that whenever an FRU is removed, the port lockout bit should be used to ensure that whatever new FRU is inserted cannot access the system until the system host allows it. When a FRU is hot-inserted connecting to a switch device, the now connected link will automatically start the training sequence. When training is complete (the Port OK bit in the Port *n* Error and Status CSR is now set), the locked port generates a Maintenance port-write operation to notify the system host of the new connection, and sets the Port-write Pending bit.

On receipt of the port-write, the system host is responsible for bringing the inserted FRU into the system in a controlled manner. The system host can communicate with the inserted FRU using Maintenance operations after clearing all error conditions, if any, clearing the Port Lockout bit and clearing the Output and Input Port Enable bits in the Port *n* Control CSR. This procedure allows the system host to access the inserted FRU safely, without exposing itself to incorrect behavior by the inserted FRU.

In order to issue Maintenance operations to the inserted FRU, the system host must first make sure that the ackID values for both ends are consistent. Since the inserted FRU has just completed a power-up reset sequence, both it’s transmit and receive ackID values are the reset value of 0x00. The system host can set the switch device’s

transmit and receive ackID values to also be 0x00 through the Port *n* Local ackID Status CSR if they are not already in that state, and can then issue packets normally.

The second method for hot insertion would allow the replaced FRU to bring itself into the system, which is necessary for a system in which the FRU is the system host itself. In this approach, the Port Lockout bit is not set and instead the Output and Input Port Enable bits are set for any unconnected port, allowing inserted FRUs free access to the system without reliance on a system host. Also, a port-write operation is not generated when the training sequence completes and the link is active, so a host is not notified of the event. However, this method leaves the system vulnerable to corruption from a misbehaving hot-inserted FRU.

As with the first case, the system host must make the ackID values for both link partners match in order to begin sending packets. In order to accomplish this, the system host generates a link-request/link-status to the attached device to obtain it's expected receiver value using the Port *n* Link Maintenance Request and Response CSRs. It can then set its transmit ackID value to match. Next, the system host generates a Maintenance write operation to set the attached device's Port *n* Local ackID Status CSR to set the transmit ackID value to match the receive ackID value in the system host. Upon receipt of the maintenance write, the attached device sets it's transmit ackID value as instructed, and generates the maintenance response using the new value. Packet transmission can now proceed normally.

Hot extraction from a port's point of view behaves identically to a very rapidly failing link and therefore can utilize the above described error reporting mechanism. Hot extraction is ideally done in a controlled fashion by taking the FRU to be removed out of the system as a usable resource through the system management software so that extraction does not cause switch fabric congestion or result in a loss of data.

The required mechanical aspects of hot-insertion and hot-extraction are not addressed in this specification.

A.4 Port-write Discussion

The error management specification includes only one destination for port-write operations, while designers of reliable systems would assume that two is the minimum number. This section explains the rationale for only having one port-write destination.

It is assumed that in the event of an error on a link that both ends of the link will see the error. Thus, there are two parties who can be reporting on any error. In the case that the sole link between an end point and a switch fails completely, the switch is expected to see and report the error. When one of a set of redundant links between an end point and a switch device fails, it is expected that the switch and possibly the end point will report the failure.

When a link between two switches fails, it is assumed that there are multiple paths to the controlling entity available for the port-write to travel. The switches will be able to send at least one, and possibly two, reports to the system host. It is assumed that it is possible to set up a switch's routing parameters such that the traffic to the system host will follow separate paths from each switch.

In some reliable systems, the system host is implemented as multiple redundant subsystems. It is assumed in RapidIO that only one subsystem is actually in control at any one time, and so should be the recipient of all port-writes. If the subsystem that should be in control is detected to be insane, it is the responsibility of the rest of the control subsystem to change the destination for port-writes to be the new subsystem that is in control.

A.5 Physical Layer Fatal Error Recovery Discussion

Recovery from a fatal error under software control at the physical layer may be possible under certain circumstances. An example of this would be if the transmitter and receiver have lost synchronization of their ackIDs. This could occur if one end of the link experienced a spurious reset. In this case a loss of packets may occur as there may be outstanding unacknowledged packets between the transmitter and the receiver.

Such an event would cause an error to be detected given the appropriate initial conditions at the transmitter, and, eventually a port-write to the system host to be generated if the system is properly configured:

- The reset state of the Input Port Enable bit in the Port *n* Control CSR set to disabled throughout the system.
- All defined errors in the Port *n* Error Detect CSRs are enabled and will increment the error rate counter throughout the system.

If a device experiences a reset event, numerous errors will be detected by the transmitter over time, and eventually an error threshold is reached as described in Section 1.2.2, "Error Reporting Thresholds", and the system host is notified as described in Section 1.4, "System Software Notification of Error". The most likely errors that will be detected are bits 12 (Received packet-not-accepted control symbol) and 26 (Non-outstanding ackID) in the Port *n* Error Detect CSRs, but others could be encountered depending upon the state of the link at the time of the reset event.

Re-synchronizing the ackIDs must be done from the transmitter side as it is not possible to communicate with the receiver with maintenance transactions in this situation. This can be done by resetting some of the physical layer state by writing the Port *n* Local ackID Status CSR with the appropriate ackID values. It may be necessary to have the transmitter drop outstanding packets using that CSR as well, depending upon the situation. It may not be desirable, or it might not be possible, to resend the packets, depending upon the state of the overall system and the

transmitter implementation.

Therefore, the following sequence of events occur:

1. The system is configured as described above and is operating.
2. The receiver of a transmitter/receiver pair experiences a reset.
3. The transmitter enters error recovery mode and attempts to re-train the link.
4. Eventually the receiver comes back and link re-training completes.
5. The transmitter starts the error recovery sequence and begins to encounter large numbers of errors due to a bad ackID for a link-response (which may immediately cause a port-write transaction to be sent to the system host to report the condition) or having all packets receive packet-not-accepted control symbols. As noted earlier, other errors may also be detected.
6. At some point, an error threshold is reached and the system host is sent a port-write maintenance transaction to report the condition, if one has not already been sent.
7. The system host cleans up the machine using maintenance transactions, including resetting ackIDs in the transmitter and rediscovering and reconfiguring the lost portion of the machine. This may be a very complex and time-consuming task.

Note that it may be useful to implement resetting the ackIDs and restarting the link in hardware for lab debug or for applications where frequent resets are expected and software intervention is not required.

A.6 Persistence of Error Management Registers

Under some conditions, a device may be unable to accept any packets because it is in an undefined, ‘broken’ condition. It is unable to accept Maintenance packets to access any of its Error Detect and capture registers so it cannot be queried by software. Only a device ‘reset’ is able to bring the device back. The meaning of a link-request/reset condition may be modified for some implementations of the Error Management Extensions to be a ‘soft reset’ condition. A device that supports a soft reset will still cause a hardware reset however the Port *n* Error Detect register, the Port *n* Error Capture registers, the Logical/Transport Error Detect register, and the Logical/Transport Error Capture registers may retain their previous values.

Blank page

Glossary of Terms and Abbreviations

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

D

Degraded threshold. Bits 8-15 of the Port n Error Rate Threshold CSR. An application-specific level that indicates an unacceptable error rate resulting in degraded throughput, when equal to the error rate count.

F

Failed threshold. Bits 0-7 of the Port n Error Rate Threshold CSR. An application-specific level that indicates an error rate due to a broken link, when equal to the error rate count.

H

Hot-insertion. Hot-insertion is the insertion of a processing element into a powered-up system.

Hot-extraction. Hot-extraction is the removal of a processing element from a powered-up system.

L

Logical/Transport error. A logical/transport error is one that cannot be resolved using the defined transmission error recovery sequence, results in permanent loss of data or causes system corruption. Recovery may possible under software control.

N

Non-reporting processing element. A non-reporting processing element depends upon an attached device (usually a switch) to report its logged errors to the system host on its behalf.

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

Physical error. A physical error occurs only in the physical layer.

Port healing. The process whereby software resets the error rate count, or allows it to decrement as required by the error rate bias field of the Port n Error Rate CSR.

R

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

Reporting processing element. A reporting processing element is capable of reporting its logged errors to the system host.

S

Switch processing element. One of three processing elements, a switch processing element, or switch, is capable of logging and reporting errors to the host system.

T

Transmission error. A transmission error is one that can be resolved using the defined transmission error recovery sequence, results in no permanent loss of data and does not cause system corruption. Recovery may also be possible under software control using mechanisms outside of the scope of this specification.

Blank page

Blank page

RapidIO™ Interconnect Specification Part 9: Flow Control Logical Layer Extensions Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.0	First release	06/18/2003
1.3	No technical changes, revision changed for consistancy with other specifications Converted to ISO-friendly templates	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Table of Contents

Blank page

List of Figures

List of Figures

List of Tables

List of Tables

Chapter 1 Flow Control Overview

1.1 Introduction

A switch fabric based system can encounter several types of congestion, differentiated by the duration of the event:

- Ultra short term
- Short term
- Medium term
- Long term

Congestion can be detected inside a switch, at the connections between the switch, and other switches and end points. Conceptually, the congestion is detected at an output port that is trying to transmit data to the connected device, but is receiving more information than it is able to transmit. This excess data can possibly "pile up" until the switch is out of storage capacity, and then the congestion spreads to other devices that are connected to the switch's inputs, and so on. Therefore, contention for a particular connection in the fabric can affect the ability of the fabric to transmit data unrelated to the contested connection. This is highly undesirable behavior for many applications.

The length of time that the congestion lasts determines the magnitude of the effect the congestion has upon the system overall.

Ultra short term congestion events are characterized as lasting a very small length of time, perhaps up to 500 or so nanoseconds. In a RapidIO type system these events are adequately handled by a combination of buffering within the devices on either end of a link and the retry based link layer mechanism defined in the RapidIO Part 4: 8/16 LP-LVDS Physical Layer and RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specifications. This combination adds "elasticity" to each link in the system. The impact of ultra short term events on the overall system is minor, if noticeable at all.

Short term congestion events last much longer than ultra short term events, lasting up into the dozens or hundreds of microseconds. These events can be highly disruptive to the performance of the fabric (and the system overall), in both aggregate bandwidth and end to end latency. Managing this type of congestion requires some means of detecting when an ultra short term event has turned into a short term event, and then using some mechanism to reduce the amount of data being

injected by the end points into the congested portion of the fabric. If this can be done in time, the congestion stays localized until it clears, and does not adversely affect other parts of the fabric.

Medium term congestion is typically a frequent series of short term congestion events over a long period of time, such as seconds or minutes. This type of event is indicative of an unbalanced data load being sent into the fabric. Alleviating this type of congestion event requires some sort of software based load balancing mechanism to reconfigure the fabric.

Long term congestion is a situation in which a system does not have the raw capacity to handle the demands placed upon it. This situation is corrected by upgrading (or replacing) the system itself.

This specification addresses the problem of short term congestion.

1.2 Requirements

The flow control mechanism shall fulfill the following goals:

- Simple - excess complexity will not gain acceptance
- React quickly - otherwise the solution won't work
- Robust - same level of protection and recovery as the rest of RapidIO
- Scalable - must be able to extend to multi-layer switch systems
- Compatibility with all physical layers

1.3 Problem Illustration

The *RapidIO Part 1: Input/Output Logical Specification* defines a transaction request flow as a series of packets that have a common source identifier and a common destination identifier at some given priority. On a link, packets of a single transaction request flow can be interleaved with packets from one or more other transaction request flows.

No assumptions are made on the underlying switch architecture for this discussion of the short term congestion problem. Also for the purposes of this discussion, an idealized output queued switch is assumed, which in literature is also used to compare the performance of a particular switch under study. Packet buffers are associated with the output of the switch. An example switch topology showing output buffers is illustrated in Figure 1-1 below. A point of congestion is therefore associated with an output buffer of such a switch.

The problem that is to be addressed by this specification is caused by multiple independent transaction request flows, each with burst and spatial locality characteristics that typically do not exceed the bandwidth capacity of links or end points. Due to the statistical combination of such transaction request flows, usually

in the middle of multistage topologies, the demand for bandwidth through a particular link exceeds the link's capacity for some period of time, for example, Data Flows a, b, and c for an output port of Switch 3 as shown in Figure 1-1. As a result, the output buffer for this port will fill up, causing the link layer flow control to be activated on the links of the preceding switch stages. The output packet buffers for Switches 1 and 2 then also fill up. Packets for transaction request flows, such as data flow d, in these same output buffers not destined for the output port with the full buffer in Switch 3 are now also waiting, causing additional system performance loss. This phenomenon is known as higher order head of line blocking.

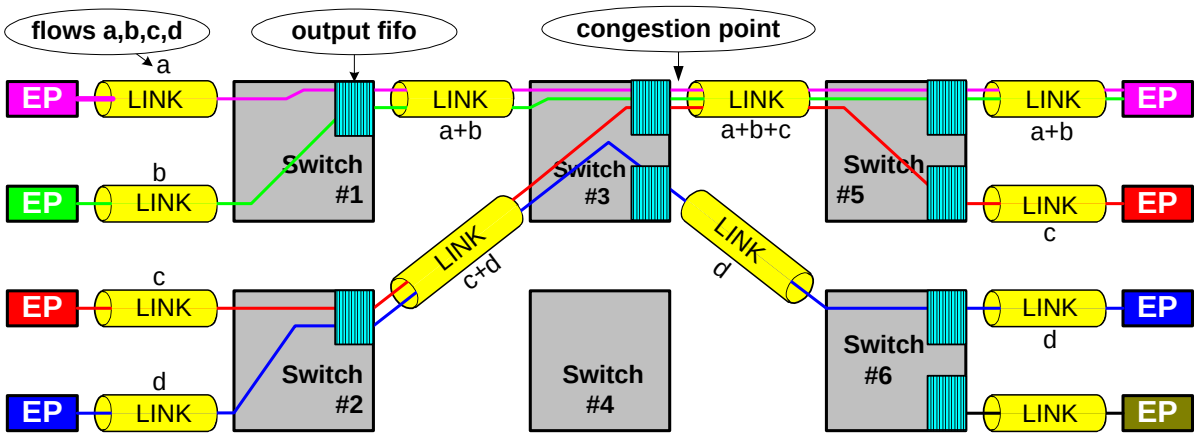


Figure 1-1. Interconnect Fabric Congestion Example

A second problem, less frequently a contributor to system performance loss, occurs when an end point cannot process the incoming bandwidth and employs link layer flow control to stop packets from coming in. This results in a similar sequence of events as described above.

The problem described in this section is very well known in the literature. The aggregate throughput of the fabric is reduced with increased load when congestion control is not applied (see reference [1]). Such non-linear behavior is known as 'performance-collapse'. It is the objective of this specification to provide a logical layer flow control mechanism to avoid this collapse. Research also shows that relatively simple "XON/XOFF" controls on transaction request flows can be adequate to control congestion in fabrics of significant size.

The reason for the described non-linear behavior is illustrated with a saturation tree. The point at which a single transaction request flow that causes link bandwidth to be exceeded and causes buffer overflow is referred to as the root of the saturation tree. This tree grows backward towards the sources of all transaction request flows going through these buffers, and all buffers that these transaction request flows pass through in preceding stages, causing even more transaction request flows to be affected.

An important design factor for interconnect fabrics is the latency between a

congestion control action being initiated and the transaction request flow source acting in response. This latency determines, among other factors, the required buffer sizes for the switches. To keep such buffers small, the latency of a congestion control mechanism must be minimized. For example, 10 data flows contribute to a buffer overflow (forming what is known as a “hotspot”). If it takes 10 packet transmission times for the congestion notification to reach the sources and the last packets sent from the sources to reach the point of congestion after the sources react to the congestion notification, up to 100 packets could be added to the congested buffer. The number of packets added may be much smaller depending on the rate of oversubscription of the congested port.

Reference

[1] “Tree saturation control in the AC3 velocity cluster interconnect”, W. Vogels et.al., Hot Interconnects 2000, Stanford.

Chapter 2 Logical Layer Flow Control Operation

2.1 Introduction

This chapter describes the logical layer flow control mechanism.

2.2 Fabric Link Congestion

In compliant devices, logical layer flow control methods shall be employed within a fabric or destination end point for the purpose of short term congestion abatement at the point in time and location at which excessive congestion is detected. This remediation scheme shall be enacted via explicit flow control messages referred to as transmit off (XOFF) and transmit on (XON) congestion control packets (CCPs) which, like any other packet, require link-level packet acknowledgements. The XOFF CCPs are sent to shut off select flows at their source end points. Later, when the congestion event has passed, XON CCPs are sent to the same source end points to restore those flows.

The method used to detect congestion is implementation specific and is heavily dependent upon the internal packet buffering structure and capacity of the particular switch device. In the example output port buffered switch from “Section 1.3, Problem Illustration” on page 10, congestion occurs when some output buffer watermark is exceeded, but this is not the only way of detecting congestion. Several possible implementation methods are described in Appendix A. These described methods are purely exemplary and are not intended to be an exhaustive list of possible methods.

2.3 Flow Control Operation

The flow control operation consists of a single FLOW CONTROL transaction as shown in Figure 2-1. The FLOW CONTROL transaction is issued by a switch or end point to control the flow of data. This mechanism is backward compatible with

RapidIO legacy devices in the same system.

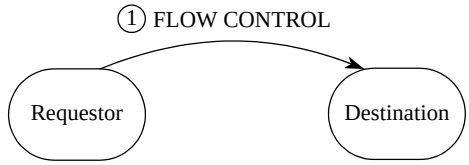


Figure 2-1. Flow Control Operation

2.4 Physical Layer Requirements

This section describes requirements put upon the system physical layers in order to support efficient logical layer flow control.

2.4.1 Fabric Topology

The interconnect fabric for a system utilizing the logical layer flow control extensions must have a topology such that a flow control transaction can be sent back to any transaction request flow source. This path through the fabric may be back along the path taken by the transaction request flow to the congestion point or it may be back along a different path, depending upon the requirements of the particular system.

2.4.2 Flow Control Transaction Transmission

Flow control transactions are regarded as independent traffic flows. They are the most important traffic flow defined by the system. Flow control transactions are always transmitted at the first opportunity at the expense of all other traffic flows if possible. For the 8/16 LP-LVDS and 1x/4x LP-Serial physical layer specifications, this requires marking flow control packets with a “prio” field value of 0b11, and a “crf” bit value of 0b1, if supported. These transactions use a normal packet format for purposes of error checking and format.

Because an implicit method of flow restoration was simulated and found to be impractical for RapidIO fabrics due to lack of system knowledge in the end point, an explicit restart mechanism using an XON transaction is used. In the CCP flow back to the source end point, XOFF and XON CCPs may be dropped on input ports of downstream elements in the event of insufficient buffer space.

2.4.2.1 Orphaned XOFF Mechanism

Due to the possibility of XON flow control packets being lost in the fabric, there shall be an orphaned XOFF mechanism for the purpose of restarting orphaned flows which were XOFF’d but never XON’d in end points. Details of this mechanism are implementation specific, however the end point shall have sufficient means to avoid

abandonment of orphaned flows. A typical implementation of such a mechanism would be some sort of counter. A description of a possible implementation is given in Appendix A. The Orphaned XOFF Mechanism is intended to work with the rest of the XON/XOFF CCPs to handle the short term congestion problem as previously described, and so shall operate such that software intervention is not required or inadvertently invoked.

2.4.2.2 Controlled Flow List

It is required that elements which send XOFFs keep a list of flows they have stopped, along with whatever flow-specific information is needed to select flows for restart, such as per-flow XON watermark level, or relative shut off order. This information shall be stored along with flow identification information in a “controlled flow list”, a memory structure associated with the controlling element. It shall be permissible in the time following the sending of an XOFF CCP for the flow control -initiating element to re-evaluate system resources and modify the flow restart ordering or expected XON watermark level within the controlled flow list to better reflect current system state. It shall not however be permissible to abandon the controlled flow by “forgetting” it, either due to lack of controlled flow list resources or other factors. In the event that limited controlled flow list resources cause the congested element to have insufficient room to issue another XOFF CCP which is deemed more important than a previously-XOFF’d controlled flow, then that previously-XOFF’d controlled flow may be prematurely XON’d and removed from the controlled flow list. The new, more important flow may be XOFF’d and take its place in the controlled flow list.

Details of the controlled flow list are implementation specific, though at the very least it shall contain entries for each currently XOFF’d flow, including flow identification information. It is likely that some state information will be required, such as expected time of flow restart, or per-flow restart watermark levels. The controlled flow list size is selected to provide coverage for short term congestion events only. Remediation for medium and greater -term congestion events is beyond the scope of logical layer flow control as these events likely indicate systemic under-provisioning in the fabric.

2.4.2.3 XOFF/XON Counters

XOFF/XON counters shall be instantiated for some number of output flows at the end point. Since the number of flows may be large or unpredictable, the number of counters and how flows are aggregated to a particular counter is implementation dependant. However, all flows must be associated with a counter. For simplicity, the following behavioral description assumes a single flow associated with a single counter. The counter is initialized to zero at start up or when a new DestinationID and given Priority is initialized. The counter increments by one for each associated XOFF CCP and decrements by one for each associated XON CCP, stopping at zero. Only when this counter is equal to zero is the flow enabled. In no event shall the counter wrap upon terminal count. If the orphaned XOFF mechanism activates, the

counter is reset to zero and the flow is restarted.

2.4.3 Priority to Transaction Request Flow Mapping

When a switch or end point determines that it is desirable to generate a flow control transaction, it must determine the associated flowID for the (non-maintenance and non-flow control) packet that caused the flow control event to be signalled. Maintenance and flow control transaction request flows must never cause the generation of a flow control transaction. For the 8/16 LP-LVDS and the 1x/4x LP-Serial physical layer specifications, the flowID of a transaction request flow is mapped to the “prio” bits as summarized in Table 1-3 of the 8/16 LP-LVDS specification and Table 5-1 of the 1x/4x LP-Serial specification. Determining the original transaction request flow for the offending packet requires the switch to do a reverse mapping.

It is recognized that mapping a particular response to a particular transmission request may be inaccurate because the end point that generated the response is permitted in the physical layer to promote the response to a priority higher than would normally be assigned. Deadlock avoidance rules permit this promotion. For this reason the choice of which flow to XOFF is preferably made using request packets, not response packets, as responses release system resources, which also may help alleviate system congestion.

Additionally, the crf (critical request flow) bit should also be used in conjunction with flowID to decide whether or not a particular transaction request flow should be targeted with a XOFF flow control transaction. A switch may select for shut off a packet with crf=0 over a packet with crf=1 if there are two different flows of otherwise equal importance. Correspondingly, an end point may choose to ignore a flow control XOFF request for a transaction request flow that it regards as critical.

The reverse mappings from the transaction request flow prio field to the CCP flowID field for the 8/16 LP-LVDS and 1x/4x LP-Serial physical layers are summarized in Table 2-1.

Table 2-1. Prio field to flowID Mapping

Transaction Request flow prio Field	Transaction Type	System Priority	CCP flowID
0b00	request	Lowest	A
0b00	response	Illegal	
0b01	request	Next	B

Table 2-1. Prio field to flowID Mapping

0b01	response	Lowest	A
0b10	request	Highest	C or higher
0b10	response	Lowest or Next	A or B
0b11	request	Illegal	
0b11	response	Lowest or Next or Highest	A, B, C or higher

2.4.4 Flow Control Transaction Ordering Rules

The ordering rules for flow control transactions within a system are analogous to those for maintenance transactions.

1. Ordering rules apply only between the source (the original issuing switch device or destination end point) of flow control transactions and the destination of flow control transactions.
2. There are no ordering requirements between flow control transactions and maintenance or non-maintenance request transactions.
3. A switch processing element must pass through flow control transactions between an input and output port pair in the order they are received.
4. An end point processing element must process flow control transactions from the same source (the destination of the packet that caused the flow control event) in the order they are received.

2.4.5 End Point Flow Control Rules

There are a number of rules related to flow control that are required of an end point that supports the logical layer flow control extensions.

1. An XOFF flow control transaction stops all transaction request flows of the specified priority and lower targeted to the specified destination and increments the XON/XOFF counter associated with the specified flowID.
2. A XON flow control transaction decrements the XON/XOFF counter associated with the specified flowID. If the resulting value is zero, the transaction request flows for that flowID and flowIDs of higher priority are restarted.
3. An end point must be able to identify an orphaned XOFF'd flow and restart it.
4. A destination end point issuing an XOFF Flow Control transaction must maintain the information necessary to restart the flow with an XON flow control transaction when congestion abates.
5. Upon detection of congestion within one of its ports, the destination end point shall send required CCP(s) as quickly as possible to reduce latency back to the source end point.

2.4.6 Switch Flow Control Rules

There are a number of rules related to flow control that are required of a switch that supports the logical layer flow control extensions.

1. Upon detection of congestion within a port, the switch shall send a CCP (XOFF) for each congested flow to their respective end points.
2. If a switch runs out of packet buffer space, it is permitted to drop CCPs.
3. A switch issuing an XOFF Flow Control transaction must maintain the information necessary to restart the flow with an XON flow control transaction when congestion abates.

Chapter 3 Packet Format Descriptions

3.1 Introduction

This chapter contains the definitions of the flow control packet format.

3.2 Logical Layer Packet Format

The type 7 FLOW CONTROL packet formats (Flow Control Class) are used by a RapidIO switch or end point processing element to stop (XOFF) and start (XON) the flow of traffic to it from a targeted RapidIO end point processing element. A single transaction request flow is targeted with a CCP. Type 7 packets do not have a data payload and do not generate response packets. The origin of a flow control packet shall set the SOC (Source of Congestion) bit to (SOC=0) if it is a switch or (SOC=1) if it is an end point. The SOC bit is informational only but may be useful for system software in identifying a failing end point.

Definitions and encodings of fields specific to type 7 packets are provided in Table 3-1.

Table 3-1. Specific Field Definitions and Encodings for Type 7 Packets

Type 7 Fields	Encoding	Definition
XON/XOFF	0b0	Stop issuing requests for the specified and lower priority transaction request flows
	0b1	Start issuing requests for the specified and higher priority transaction request flows
flowID	—	Highest priority affected transaction request flow 0b0000000 - transaction request flow A 0b0000001 - transaction request flow B 0b0000010 - transaction request flows C and higher Remaining encodings are reserved for the 8/16 LP-LVDS and the 1x/4x LP-Serial physical layers.
destinationID	—	Indicates which end point the CCP is destined for (sourceID of the packet which caused the generation of the CCP).
tgtdestinationID	—	Combined with the flowID field, indicates which transaction request flows need to be acted upon (destinationID field of the packet which caused the generation of the CCP).
SOC	0b0	Source Of Congestion is a Switch
	0b1	Source Of Congestion is an End Point
rsrv	—	Reserved

Figure 3-1 displays a CCP packet with all its fields. The field value 0b0111 in Figure 3-1 specifies that the packet format is of type 7. Small (tt=0b00) and Large (tt=0b01) Transport Formats are shown in the figure.

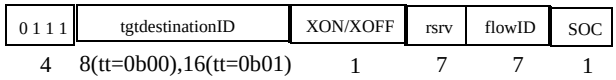


Figure 3-1. Type 7 Packet Bit Stream Logical Layer Format

3.3 Transport and Physical Layer Packet Format

Figure 3-2 shows a complete flow control packet, including all transport and 1x/4x LP-Serial physical layer fields except for delineation characters. The destinationID field of the CCP packet is the sourceID field from packets associated with the congestion event, and is the target of the flow control transaction. The tgtdestinationID field is the destinationID field from packets associated with the congestion event, and was the target of those packets. The tgtdestinationID field is used by the target of the flow control packet to identify the transaction request flow that needs to be acted upon. For all undefined flowID encodings, there is no action required and the tgtdestinationID is ignored. Field size differences for 8 bit address Small Transport Format (tt=0b00) vs. 16 bit address Large Transport Format (tt=0b01) are shown. Note: when tt=0b01 there will be a pad after the CRC.

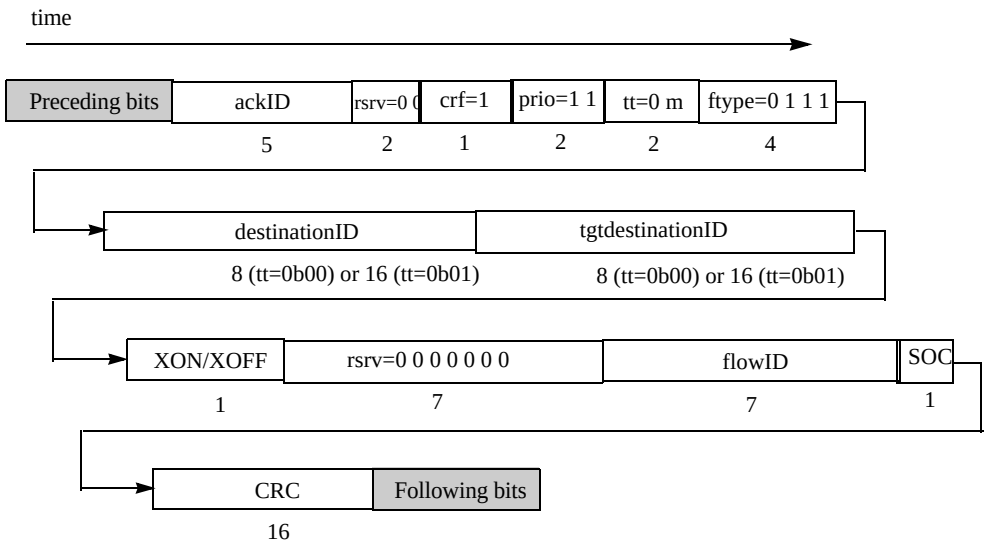


Figure 3-2. 1x/4x LP-Serial Flow Control Packet

Figure 3-3 shows the corresponding 8/16 LP-LVDS physical layer small transport packet.

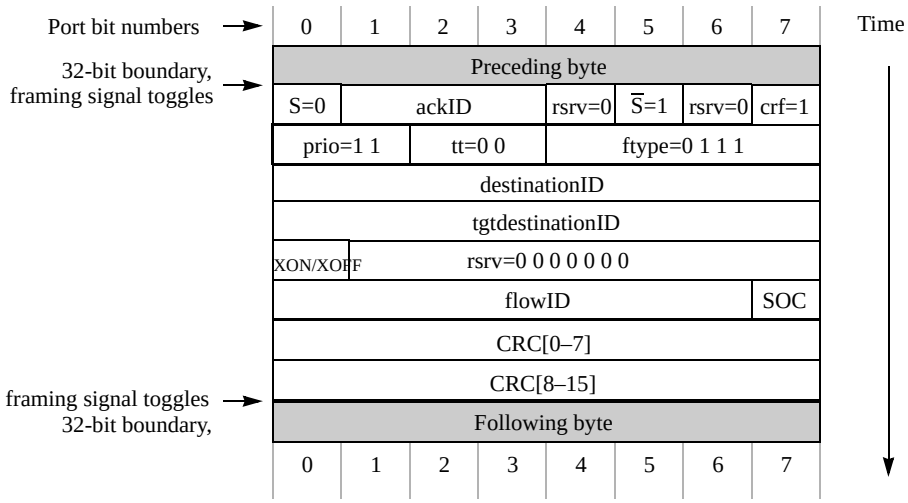


Figure 3-3. 8/16 LP-LVDS Small Transport Flow Control Packet

Blank page

Chapter 4 Logical Layer Flow Control Extensions Register Bits

4.1 Introduction

This section describes the Logical Layer Flow Control Extensions CAR and CSR bits that allow an external processing element to determine if a switch or end point device supports the flow control extensions defined in this specification, and to manage the transmission of flow control transactions for a switch processing element. This chapter only describes registers or register bits defined by this specification. Refer to the other RapidIO logical, transport, physical, and extension specifications of interest to determine a complete list of registers and bit definitions for a device. All registers are 32-bits and aligned to a 32-bit boundary.

4.2 Processing Elements Features CAR (Configuration Space Offset 0x10)

The Processing Elements Features CAR contains 31 processing elements features bits defined in various RapidIO specifications, as well as the Flow Control Support bit, defined here.

Table 4-1. Bit Settings for Processing Elements Features CAR

Bit	Name	Reset Value	Description
0-23	-		Reserved (defined elsewhere)
24	Flow Control Support		* Support for flow control extensions 0b0 - Does not support flow control extensions 0b1 - Supports flow control extensions
25-31	-		Reserved (defined elsewhere)

* Implementation dependant

4.3 Port *n* Control CSR (Block Offset 0x08)

The Port *n* Control CSR contains 31 bits specifying individual port controls defined in various RapidIO specifications, as well as the Flow Control Participant bit, defined here.

Table 4-2. Bit Settings for Port *n* Control CSR

Bit	Name	Reset Value	Description
0-9 (parallel) 0-12 (serial)	-		Reserved (defined elsewhere)
10 (parallel) 13 (serial)	Flow Control Participant	0b0	Enable flow control transactions 0b0 - Do not route or issue flow control transactions to this port 0b1 - Route or issue flow control transactions to this port
11-31 (parallel) 14-31 (serial	-		Reserved (defined elsewhere)

Annex A Flow Control Examples (Informative)

A.1 Congestion Detection and Remediation

The method used to detect congestion is implementation specific and is heavily dependent upon the internal packet buffering structure and capacity of the particular switch device. In the example output port buffered switch from “Section 1.3, Problem Illustration” on page 10, congestion occurs when some output buffer watermark is exceeded. As long as the watermark is exceeded the output port is said to be in a congested state. The watermark can have different levels when entering the congested state and leaving the congested state.

Fabric elements should monitor their internal packet buffer levels, comparing them on a packet by packet basis to pre-established, locally-defined watermark levels. These levels likely would be configurable depending upon the local element's position within the fabric relative to source endpoints and its particular architecture. On the high watermark side, a level should be selected which is low enough that the remaining buffer space is adequate to provide ample storage for packets in-flight, given a worse-case latency for XOFF CCPs to travel back to the source endpoint and shut off the flow in the endpoint. On the low watermark side (if a watermark is used for XON), a yet-lower level should be selected which meets the following criteria;

a) Provides sufficient hysteresis. When considered in context with the high watermark, it should not be so close as to provide a high flow of XON/XOFF CCP traffic back to the source endpoint.

b) Is set high enough that the switch output buffer does not run dry (underflow) in the typical live-flow case (one or more packets are present in the source endpoint output buffer waiting to be sent when the flow is restarted), given the latency of XON CCP travel back to the source endpoint and restoration of the shut-off flow in the endpoint.

The following two examples are provided to show possible methods for detecting and reacting to congestion:

1. Histogram analysis:
   — The switch keeps track of packet quantities for the different transaction request flows for which packets are stored in its output buffer.
   — The switch sorts the transaction request flows according to the number of packets.
   — The switch selects the 1 to 5 transaction request flows with the most

packets stored in the buffers.

— The switch sends an XOFF flow control request to those transaction request flow sources when the watermark threshold is exceeded, as long as flow control transaction routing is enabled on that switch port. Handling of system critical flows intending to bypass the flow control operation is outside the scope of this document.

— The CCP-targeted sources stop transmitting packets for the indicated transaction request flow and all lower priority transaction request flows.

— The switch sends a flow control XON request to those transaction request flow sources when the watermark drops below the threshold.

— The CCP-targeted sources begin to transmit packets for the indicated transaction request flow and all higher priority transaction request flows.

2. Simple threshold:

— The switch sends an XOFF flow control to the source of every new transaction flow it receives as long as the watermark is exceeded, provided flow control transaction routing is enabled on that switch port. Handling of system critical flows intending to bypass the flow control operation is outside the scope of this document.

— The CCP-targeted sources stop transmitting packets for the indicated transaction request flow and all lower priority transaction request flows.

— The switch sends a flow control XON request to those transaction request flow sources when the watermark drops below the threshold.

— The CCP-targeted sources begin to transmit packets for the indicated transaction request flow and all higher priority transaction request flows.

Note that the first method is reasonably fair in that it targets the source of the data flows that are consuming most of the link bandwidth, and that the second method is unfair in that it indiscriminately targets any source unfortunate enough to have a packet be transmitted while the link is congested.

A.2 Orphaned XOFF Mechanism Description

This timer may take the form of a low precision counter in the end point which monitors the oldest XOFF’d flow at any given time. When a flow first becomes the oldest flow (reaches top of an XOFF’d flow FIFO list within the end point) the timer is reset to its programmed value and begins to count down with time. If it is allowed to elapse without a change to the oldest XOFF’d flow, that flow will be presumed to be orphaned due to lost XON CCP and be restarted as if an XON CCP had been received, with the orphaned flow entry removed from the top of the list and the counter reset to count down for the next oldest XOFF’d flow. The length of the count should be long enough to insure that significant degradation of the flow control function does not occur, on the order of several times the width of the fabric expressed in terms of packet transit time, yet not so large that it would fail to elapse

between uncorrelated congestion events. The length of this count shall be programmable through an implementation-dependent register in the end point. The orphaned XOFF mechanism is intended solely as a last-resort mechanism for restarting orphaned flows. It will not be adequate for the purpose of implicit controlled flow reinstatement owing to inherent fairness issues as well as burstyness due to uncontrolled simultaneous multi-flow restart.

Blank page

Glossary of Terms and Abbreviations

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

C

Congestion. A condition found in output ports of switch and bridge elements characterized by excessive packet buildup in the buffer, when packet entry rate into the buffer exceeds packet exit rate for a long enough period of time.

CCP (Congestion Control Packet). A packet sent from the point of congestion in the fabric back to the source endpoint of particular flows instructing the source to either turn on or off the flow.

Controlled Flow List. A memory structure associated with controlling elements which holds a list of currently controlled flows, used by the element to turn back on controlled flows.

crf. Critical Request Flow. For packets or packets of a given priority, this bit further defines which packet or notice should be moved first from the input queue to the output queue (see *RapidIO Part 4: 8/16 LP-LVDS Physical Layer Specification*, Section 1.2.2 and *RapidIO Part 6: 1x/4x LP-Serial Physical Layer Specification*, Section 5.3.3).

F

flowID. Transaction request flow indicator (see *RapidIO Part 1: Input/Output Logical Specification*, Section 1.2.1).

L

Long Term Congestion. A severe congestion event in which a system does not have the raw capacity to handle the demands placed upon it in actual use.

M

Medium Term Congestion. A congestion event in which a frequent series of short term congestion events occur over a long period of time such as seconds or minutes, handled in RapidIO systems by reconfiguration of the fabric by system-level software.

O

Orphaned XOFF Mechanism. A mechanism in an end point which is used to restart the oldest controlled flow within the end point after a certain period of time has elapsed without the flow being XON'd.

P

Performance Collapse. Non-linear behavior found in non- congestion controlled fabrics, whereby reduced aggregate throughput is exhibited with increased load.

S

Saturation Tree. A pattern of congestion identified within the fabric which grows backward from the root buffer overflow towards the sources of all transaction request flows passing through this buffer.

Short Term Congestion. A congestion event lasting up into the dozens or hundreds of microseconds, handled in RapidIO by Logical Layer Flow Control.

T

Topology. The structure represented by the physical interconnections of a switch fabric.

Transaction Request Flow. A series of packets that have a common source identifier and a common destination identifier at some given priority.

U

Ultra Short Term Congestion. A congestion event lasting from dozens to hundreds of nanoseconds, handled in RapidIO by Link Level Flow Control.

Underflow. A condition within output buffers of switches in which the buffer runs dry.

W

Watermark. A predetermined buffer occupancy level indicating either congestion (high watermark) or abatement of congestion (low watermark).

X

XOFF (Transmit Off). A congestion control packet sent from the point of congestion back to the source of a particular flow, telling the source endpoint to shut off the flow.

XON (Transmit On). A congestion control packet sent from the point of congestion back to the source of a particular flow, telling the source endpoint to restart a controlled flow.

Blank page

Blank page

RapidIO™ Interconnect Specification Part 10: Data Streaming Logical Specification

Rev. 1.3.a, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.3	First release	06/09/2004
1.3.a	No technical changes Converted to ISO-friendly templates	02/23/2005
1.3.a	Removed confidentiality markings for public release	06/07/2005

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

Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Table of Contents

Chapter 5 Data Streaming Registers

Annex A VSID Usage Examples

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 10: Data Streaming Logical Specification*. The goal of the specification is to combine the need for efficiency, flexibility, and protocol independence in order to minimize the resources necessary to support a data plane interconnect fabric, and to maintain compatibility and fully inter-operate with the rest of the RapidIO specifications.

The rationale for this optimization is based upon the assumption that platforms are expected to produce many times more revenue than the initial cost of the platform. For example, a platform is expected to produce 10 times the revenue vs. its initial capital costs. If that same platform could cost 10% more but allow 10% more resources for producing revenue rather than doing fabric support, the result would be a significant net gain on the investment. Therefore, enabling more intelligence within the system fabric and relieving the system processing resources to produce revenue, even if that fabric is more expensive, is believed to be a good trade-off.

The features of the data streaming specification define virtual mechanisms in simple forms for building cost sensitive systems and also provides for complex high functioning fabrics for more demanding applications.

It is assumed that the reader has a thorough understanding of the other RapidIO specifications and of data plane equipment and applications in general.

1.2 Overview

Standard encapsulation schemes have been developed for the transmission of datagrams over most popular LANs. A number of different proposals currently exist for the encapsulation of one protocol over another protocol [RFC1226, RFC1234, RFC1701]. The data streaming logical specification defines a mechanism for transporting an arbitrary protocol over a standard RapidIO interface, and addresses interconnection between elements in an end-to-end data communications circuit. The protocol has been carefully designed to provide complete compatibility and inter-operability with existing RapidIO specifications.

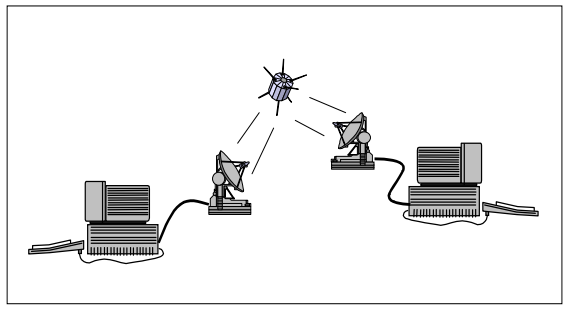


Figure 1-1. End to End Communication Circuit

The defined encapsulation methodology provides for the multiplexing of different network-layer protocols simultaneously over the same link and provides a common solution for easy connection of a wide variety of hosts, bridges and switches. It is envisioned that a RapidIO system will be capable of carrying a wide variety of data types, supporting a diverse set of protocol regimens concurrently.

1.3 Features of the Data Streaming Specification

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

1.3.1 Functional Features

- Protocol encapsulation, independent of the protocol being encapsulated.
- Support for Protocol Data Units (PDUs) of up to 64k bytes through Segmentation and Reassembly (SAR).
- Support for hundreds of traffic classes.
- Support for thousands of data streams between end points.
- Support for concurrent interleaved PDUs between end points.
- Seamless inter-operability with other RapidIO specifications.

1.3.2 Physical Features

- Packet definition is independent of the choice of physical layer interconnection 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.
- No dependencies exist on the bandwidth or latency of the physical fabric.

- The protocol requires in-order packet transmission and reception; out-of-order packet delivery is not tolerated.
- Certain devices have bandwidth and latency requirements for proper operation. The data streaming logical layer specification does not preclude an implementation from imposing these constraints within the system.

1.3.3 Performance Features

- Packet headers are small to minimize the control overhead and be organized for fast, efficient assembly and disassembly.
- Multiple transactions are allowed concurrently in the system, otherwise a majority of the potential system throughput is wasted.
- Multiple end point to end point concurrent data streams are supported for high fabric utilization.

1.4 Contents

Following are the contents of the *RapidIO Part 10: Data Streaming Logical Specification:*

- Chapter 1, “Overview,” is an overview of the data streaming logical specification.
- Chapter 2, “Data Streaming Systems,” introduces system issues such as transaction ordering and deadlock prevention.
- Chapter 3, “Operation Descriptions,” describes the set of operations and transactions supported by the RapidIO data streaming protocol.
- Chapter 4, “Packet Format Descriptions,” contains the packet format definitions for the data streaming specification.
- Chapter 5, “Data Streaming Registers,” describes the visible register set that allows an external processing element to determine the data streaming capabilities, configuration, and status of a processing element using this logical specification. Only registers or register bits specific to the data streaming 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.
- Annex A, “VSID Usage Examples,” contains a number of examples of how the virtual stream identifier can be used in a system.

1.5 Terminology

The data streaming logical specification introduces some new terms:

Protocol Data Unit - (PDU) A self contained unit of data transfer comprised of data and protocol information that defines the treatment of that data.

Virtual Stream ID - (VSID) an identifier comprised of several fields in the protocol to identify individual data streams.

Virtual input Queue (ViQ), Virtual output Queue (VoQ) - an intermediate point in the system where one or more virtual streams may be concentrated.

Class of service - (cos) a term used to describe different treatment (quality of service) for different data streams. Support for class of service is provided by a class of service field in the data streaming protocol. The class of service field is used in the virtual stream ID and in identifying a virtual queue.

StreamID - a specific field in the data streaming protocol that is combined with the data streams's transaction request flow ID and the source ID or destination ID from the underlying packet transport fabric to form the virtual stream ID.

Segment - A portion of a PDU.

Segmentation - a process by which a PDU is transferred as a series of smaller segments.

Segmentation context - Information that allows a receiver to associate a particular packet with the correct PDU.

Ingress - Ingress is the device or node where traffic enters the system. The ingress node also becomes the source for traffic into the RapidIO fabric. The terms ingress and source may or may not be used interchangeably when considering a single end to end connection.

Egress - Egress is the device or node where traffic exits the system. The egress node also becomes the destination for traffic out of the RapidIO fabric. The terms egress and destination may or may not be used interchangeably when considering a single end to end connection.

Refer to the Glossary at the back of this document for additional definitions.

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.

1.7 Useful References

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC791, September 1981

[RFC1226] Kantor, B. "Internet Protocol Encapsulation of AX.25 Frames", RFC1226, University of California, San Diego, May 1991.

[RFC1234] Provan, D. "Tunneling IPX Traffic through IP Networks", RFC 1234, Novell, Inc., June 1991.

[RFC1700] J. Reynolds and J. Postel, "Assigned Numbers", RFC1700, October 1994.

[RFC2460] S. Deering, R. Hinden, "Internet Protocol, Version 6 (IPv6)", RFC2460, December 1998.

[RFC1884] Hinden, R., and S. Deering, Editors, "IP Version 6 Addressing Architecture", RFC1884, Ipsilon Networks, Xerox PARC, December 1995.

[RFC2004] C. Perkins, "Minimal Encapsulation within IP", RFC2004, October 1996.

Blank page

Chapter 2 Data Streaming Systems

2.1 Introduction

This overview introduces the role of the data streaming logical layer in an overall system. It provides some possible use examples. See Annex A, “VSID Usage Examples”, for more example details.

2.2 System Example

Figure 2-1 shows a block diagram of an example RapidIO-based networking system in which protocol encapsulation is required. A number of typical data path type devices are connected with a variety of proprietary and/or somewhat standard interfaces and the entire system is tied together with a RapidIO switching fabric of some topology.

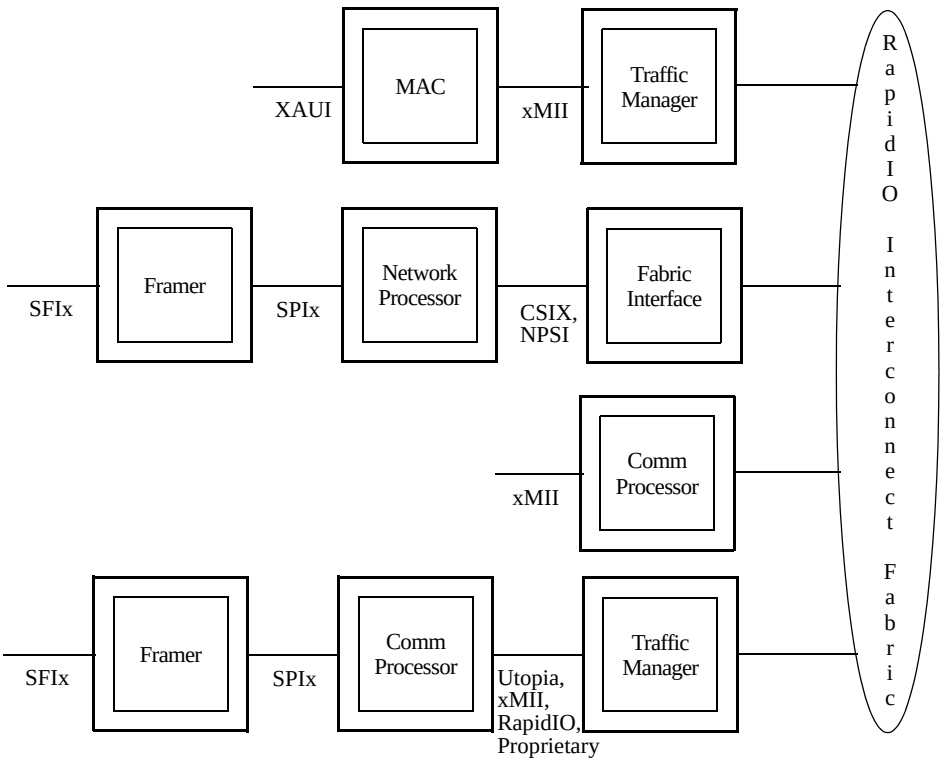


Figure 2-1. Example of a RapidIO-Based Networking System

Data “streams” represent logical connections between an ingress port and an egress port. A connection spans the transfer of multiple PDUs. The transfer of PDUs may be separated by discrete intervals of time, based on the arrival of data at the ingress. Transfer between an ingress process and an egress process is unidirectional. An I/O device may be bi-directional, containing both an ingress process and an egress process. These processes are usually completely independent consisting of separate streams in each direction.

A given ingress may service hundreds, thousands, even millions of streams at any given time depending on how specifically a PDU is classified. Traffic may be lumped into a single stream, or classified by user and application to form millions of data streams.

Data streaming transactions differ from most other RapidIO transactions in two ways: they must accommodate larger variably sized data transfers, and the transactions are not acknowledged with a response packet. The *data streaming logical layer* is intended to support data from a variety of hardware and processing devices. These devices have a variety of different interfaces, protocols, and degrees of sophistication. This specification is intended to enable these kinds of devices to exist on the RapidIO interconnect.

2.3 Traffic Streams

A stream identifier identifies independent streams of traffic between the end producer (for example, a web server) and end consumer (for example, a home personal computer) of the encapsulated data. Stream identifiers vary with protocol and may include multiple fields from the various networking layers included in the protocol. A unit of data that contains a discrete identifier is called a Protocol Data Unit, or PDU. A PDU may or may not have an ordering relationship with another PDU being transmitted between that same producer and consumer, depending upon the higher layer protocol being carried. A traffic stream is a series of PDUs that have an ordering relationship between each other. A PDU has no ordering relationship with a PDU from different producers and consumers pairs.

The data streaming logical layer uses a **virtual stream identifier** (VSID) to allow multiple end to end traffic streams of PDUs to be uniquely identified and managed concurrently within the RapidIO system. Creation of a VSID is done by performing a protocol specific classification process on a PDU. The complexity of the classification process is directly proportional to the sophistication of the system as required by the application. The VSID allows the traffic to be reassociated with an appropriate application at the egress without having to perform a second protocol-specific classification. A VSID is comprised of fields from the data streaming protocol: **source** or **destination ID** from the underlying packet transport fabric, **class of service**, and **streamID**.

2.4 Operation Ordering

A transaction request flow is defined as an ordered sequence of request transactions comprising a specific PDU from a given source (as indicated by the transaction source identifier from the underlying packet transport fabric) to a given destination (as indicated by the transaction destination identifier). Each packet in a transaction request flow has the same source identifier and the same destination identifier. All traffic streams are mapped onto transaction request flows. These flows may also be shared with other RapidIO logical layers transactions, and therefore the relationship between streams, traffic classes, virtual queues, and all RapidIO transaction request flows are implementation specific.

There may be multiple transaction request flows between a given source and destination pair. When multiple flows exist between a source and destination pair, the flows are distinguished by a flow indicator referred to as a “flowID”, introduced in the *RapidIO Part 1: Input/Output Logical Specification.* RapidIO allows multiple transaction request flows between any source and destination pair. Any number of transaction request flows may exist between the two end points. The flowID represents the lowest level of traffic management in a RapidIO system as that is the construct mapped directly on to the switch fabric itself.

The transaction request flows between each source and destination end point pair may be allocated to different virtual channels in the underlying fabric and may also be prioritized within a channel. The flows are labeled and identified alphabetically as in the other logical layer specifications, and the channels labeled and identified numerically with channel then priority, starting with 0 as first channel or lowest priority, then 1 as second channel or next lowest priority, etc. For example, flowID 0A is channel 0 flow A, flowID 1C is channel 1 flow C, flowID 3E is channel 3 flow E, and so on. This flow information provides class of service information when mapped by the application to the switch fabric.

Allocation of transaction request flows to virtual channels and the relative priority within each channel is application dependent. A special case is a single virtual channel application which must follow the same prioritization of flows and labeling as the other logical layers (flowID A, flowID B, flowID C, etc.). The channel label (0) is dropped. This channel may include traffic from the other logical layers.

At the link level, when multiple transaction request flows within the same virtual channel exist between a given connected source and destination pair, transactions of a higher priority flow may pass transactions of a lower priority flow, but transactions of a lower priority flow may not pass transactions of a higher priority flow. There are no ordering rules for flows in different channels. A traffic stream being transmitted between a source and a destination end point pair must utilize the same flowID value so that the ordering of the traffic stream is maintained. As a class of service indicator, the flowID is used by the underlying RapidIO fabric to determine how to treat a packet with respect to other packets with respect to priority and

ordering. It is expected that in a mixed control and data plane application that both I/O logical and data streaming transaction request flows will exist in a RapidIO system simultaneously, possibly between the same end point devices.

To support transaction request flows, all devices that support the RapidIO data streaming logical specification shall comply as applicable with the following Fabric Delivering Ordering and End point Completion Ordering rules. Note that these rules are very similar and complementary to the rules specified in *RapidIO Part 1: Input/Output Logical Specification*.

Fabric Delivery Ordering Rules

1. **Transactions within a transaction request flow (same source identifier, same destination identifier, same flowID, same PDU) shall be delivered to the logical layer of the destination in the same order that they were issued by the logical layer of the source.**
2. **Request transactions that have the same source (same source identifier) and the same destination (same destination identifier) within the same virtual channel but with different flowIDs shall be delivered to the logical layer of the destination as follows.**
   - **A transaction of a higher priority transaction request flow that was issued by the logical layer of the source before a transaction of a lower priority transaction request flow shall be delivered to the logical layer of the destination before the lower priority transaction.**
   - **A transaction of a higher priority transaction request flow that was issued by the logical layer of the source after a transaction of a lower priority transaction request flow may be delivered to the logical layer of the destination before the lower priority transaction.**
3. **Request transactions that have different sources (different source identifiers) or different destinations (different destination identifiers) or different virtual channels are unordered with respect to each other.**

End point Completion Ordering Rules

1. **Request transactions in a transaction request flow shall be completed at the logical layer of the destination in the same order that the transactions were delivered to the logical layer of the destination.**

It may be necessary to impose additional rules in order to provide for inter-operability with other interface standards or programming models. However, such additional rules are beyond the scope of this specification.

2.5 Class of Service and Virtual Queues

Data streaming systems may support thousands, even millions of active data streams. These streams are eventually interleaved onto the single underlying packet transport fabric. The process for deciding which streams may share common resources is sometimes referred to as virtual queuing. To facilitate virtual queuing at the ingress and/or egress of the fabric, and to provide for more sophisticated management of traffic streams, the data streaming logical layer provides a class of service (cos) identifier. The cos field exists to provide a common semantic as to how the traffic stream is to be treated. The relationship between the ingress/egress cos and the end to end flowID assigned to the traffic stream is implementation specific.

At the ingress to the fabric, thousands of streams may be combined into fewer virtual output queues (VoQs) using just the **destination ID** and the **class of service** portions of the VSID as shown in Figure 2-2. The cos field defined by this specification is comprised of one byte. The number of bits utilized by a particular device depends upon the number of data buffering structures implemented, but are always from the most significant bit of the cos field to the least significant bit. For example, a device with two buffering structures (or "bins") maps a packet to a bin using bit 0, a device with four bins maps a packet to a bin using bits 0 and 1, and so on.

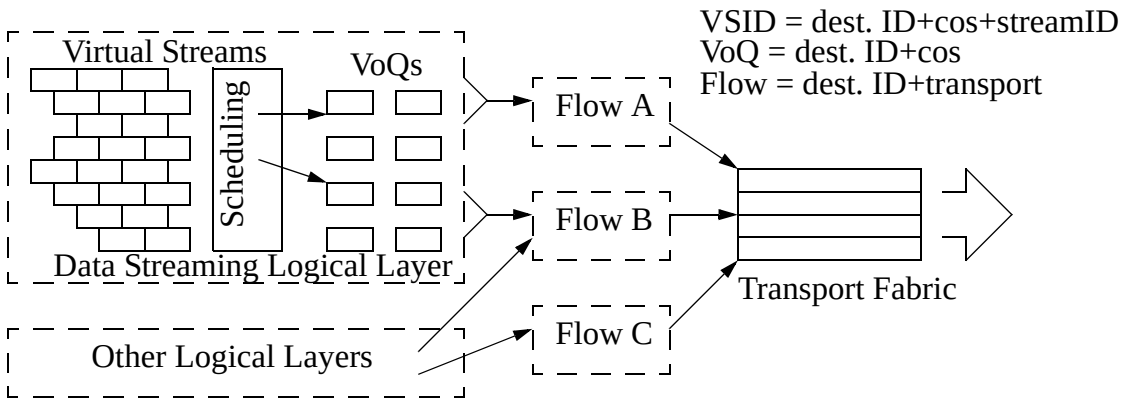


Figure 2-2. Mapping Virtual Streams at the System Ingress

As shown in Figure 2-2, as the virtual output queues are mapped on to the flowIDs and then on to the underlying packet transport fabric, they may be intermingled with other logical layer transactions. The use of the transport fabric must account for the needs of the total environment and is application and implementation specific. End points designed to support a wide variety of applications for data streaming should offer some flexibility in how virtual queues are mapped down on to the transport fabric in the implementation.

A reverse process (virtual input queueing) may or may not occur at the destination. If there is a critical resource needed to process traffic on egress from the fabric, the system designer may choose to fan the traffic back out into virtual queues. This allows the fabric egress processing to re-prioritize utilization of the critical resource.

This is illustrated in Figure 2-3.

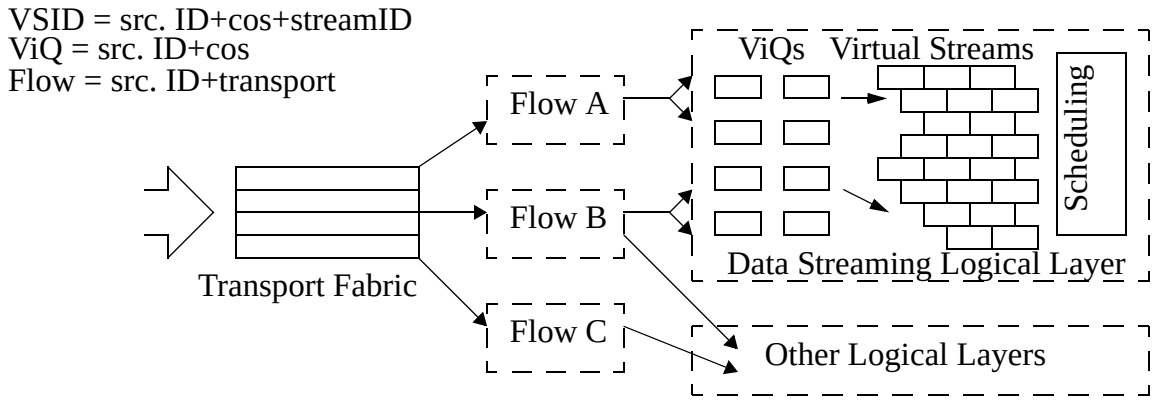


Figure 2-3. Mapping Virtual Streams at the System Egress

A switch device may choose to utilize the information carried in the cos field by acting as a "virtual" end point, removing the traffic streams from the underlying packet transport fabric, reassembling the individual PDUs, and fanning the streams back out into some larger number of queues. It then re-injects the traffic streams back into the underlying transport fabric re-ordering the traffic using the cos. This permits intervening devices to participate in the overall assurance of quality of service in the system.

2.6 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 data streaming logical specification does not have any dependency loops since the defined operations do not require responses. However, a real RapidIO system is required to support the I/O logical maintenance operation, and will very likely require the use of other logical operations for control functions. Support for these other logical operations may have significant deadlock considerations for processing element and system designs.

Chapter 3 Operation Descriptions

3.1 Introduction

This chapter describes the RapidIO data streaming protocol. The field encodings and packet formats are described in Chapter 4, “Packet Format Descriptions.”

Data path data movement through a machine has requirements that are significantly different than those for control path and traditional DMA functions. Many times this data is encapsulated data, which also many times contains further encapsulated data. For example, the data moving through the system may be encapsulated Ethernet packets, which may in turn be encapsulating TCP/IP packets.

This style of data movement is typically not address-based as with DMA type I/O, and consequently follows a queue based message passing paradigm. Data path data movement also has much more complex requirements in the area of class (or quality) of service than control path communications, and generally requires managing a number of queues at the egress of the system. There is also a need to be able to identify and manage many thousands of data traffic streams that pass through a RapidIO based data path system. The data being passed through the RapidIO system may not be directly generated or consumed by the device connected to the RapidIO portion of the machine, but instead by a distant end user, such as a personal computer attached to a LAN. This necessitates the addition of a new protocol to the RapidIO logical layers, the data streaming protocol.

The RapidIO data streaming protocol uses request transactions through the interconnect fabric as with other RapidIO operation protocols. Since many data movement protocols guarantee data delivery in an upper layer protocol, the generation of responses indicating completion are not needed. Such upper layer protocols may also allow data to be discarded if necessary, for example, under error or fabric congestion conditions.

3.2 Data Streaming Protocol

This section describes the RapidIO data streaming protocol.

3.2.1 Data Streaming Operation

A data stream represents a logical connection between a source and a destination pair. A stream may consist of multiple transactions and requires the allocation of

resources at both the source and the destination. This may be done in advance of any data transfer, or in response to receiving a new transaction. Since streams are virtual constructs between source and destination pairs, they may be reused for different data transfers at any time as long as the source and destination pair are both synchronized as to the stream usage.

A data streaming operation consists of individual data streaming transactions, as shown in Figure 3-1. A series of transactions is used to send PDUs between two end points. The data streaming protocol is completely independent of the PDU's native protocol.

Data streaming transactions do not receive responses, so there is no notification to the sender when the transaction has completed at the destination.

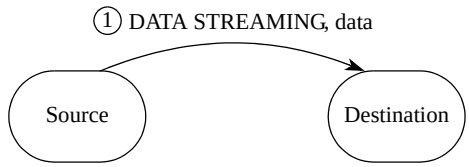


Figure 3-1. Data Streaming Operation

3.2.2 Virtual Streams

A stream is represented by a unique virtual stream indentifier, or VSID. This identifier represents the handling of all PDUs within the stream for the duration of a PDU's transit of the RapidIO fabric. The identifier is created by performing some form of protocol specific classification of the PDU. The classification can be as complex or as simple as the application warrants. The VSID allows this protocol specific classification to take place one time at the ingress to the fabric. After that, the handling of the PDU is protocol independent.

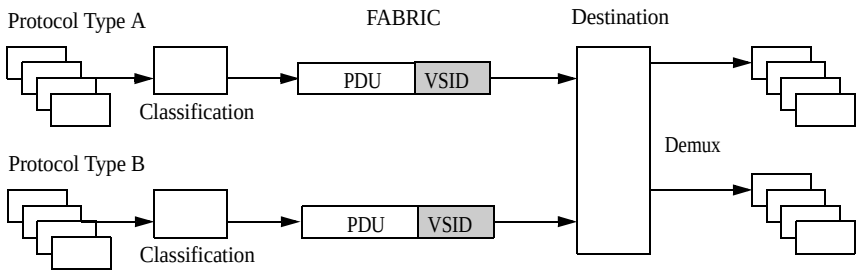


Figure 3-2. Virtual Streams

The VSID is used at the destination to "reclassify" the PDU. This sorts the data back into contexts that can now be protocol specific again. This virtual addressing model eliminates the need for the source and the destination to align the use of buffers and other resources. Therefore, the VSID can be used to carry a wide variety of information about a stream through the system, such as the protocol being encapsulated, demultiplexing exit port IDs instructions, very fine grained buffer

management, etc., as required for a specific application.

The VSID is a “key” comprised of multiple fields. These fields are the source/destination ID, cos, and streamID.
From the source’s viewpoint: **destination ID+cos+streamID** represents a unique stream.
From the destination’s viewpoint: **source ID+cos+streamID** represents a unique stream.

By using the complete key, each source and destination pair is free to allocate the use of these fields independently. Some examples of how the VSID may be applied in a system are described in Appendix A, “VSID Usage Examples,” on page 39.

3.2.3 PDU Sequences Within Streams

As described earlier, a traffic stream may consist of a sequence of related PDUs that have ordering requirements between each other. A stream of PDUs is transmitted one PDU at a time to preserve the required ordering. PDUs that do not have an ordering relationship may be separated into different streams or may be interleaved in common streams. A stream is identified by the interconnect fabric by the combination of the destination ID and either the cos field or the flowID, depending upon the complexity of the fabric, as described in “Section 2.5, Class of Service and Virtual Queues” on page 19.

Only one PDU from any given stream will be transmitted at a time at the source, but fabric conditions may result in multiple PDUs in transit. The fabric must guarantee that delivery of PDUs (and segments of PDUs as described below) remain in order. A fabric may load balance traffic through multiple paths on a stream by stream basis.

3.2.4 Segments within a PDU

The basic mechanism of segmentation defines a general methodology to provide for larger PDUs than are accommodated by the standard 256 byte limit on a RapidIO data payload. The standard industry term for this function is “Segmentation and Reassembly”, or SAR. A PDU that is to be transmitted from the initial producer to the final consumer is broken up (segmented) into a series of blocks of data. The consumer “reassembles” that data back into the original PDU. The maximum size of a PDU that a particular destination can accept is specified in a CAR (see Chapter 5, “Data Streaming Registers”). The system must be configured with in accordance to these limitations.

The block size used for the segmentation process is specified by the Maximum Transmission Unit, or MTU, parameter. The MTU is defined in Chapter 5, “Data Streaming Registers”. The MTU is a system-wide parameter agreed to by all processing elements participating in the SAR process. By managing the MTU size for the system, the variability in latency for the system can be controlled.

A data streaming transaction is also referred to as a segment. The transmission of a PDU for any given stream may result in one or more transactions (segments). A typical sequence is made up of three types of transactions, a start segment, some number of continuation segments, and an end segment. Start segments and continuation segments are always filled to the MTU size. End segments are variable in size containing the remainder of the PDU. If a PDU is equal to or less than the MTU size, it is carried in a single segment. A single segment may also be variable in size, matching the PDU payload. Since flowIDs and the cos are assigned on a PDU basis, all segments of a PDU must also have that same flowID and cos assignments.

A start segment contains the necessary fields to identify the VSID and "open" a *segmentation context*. The segmentation context for a stream is defined as the combination of the source ID and the flowID, and is used by a receiver to reassociate the segments of a particular PDU. Using **source ID+flowID** allows each source and destination pair to have one PDU for each flowID that is explicitly supported by the system interleaved in the fabric at any one point in time. The VSID is used when opening a segmentation process at the destination to associate the PDU with its stream since the continuation and end segments do not carry that information. After the receipt of the end segment, the segmentation context is "closed" (the sending processing element has an analogous definition for open and closed). The stream and PDU associated with a segmentation context is not permitted to change during the time that the context is open.

Since there may be a large number of PDU sources and concurrent contexts per source, the amount of context state that a destination may have to handle can potentially get very large. The number of contexts that can be supported by a particular destination end point is specified in a CAR (see Chapter 5, "Data Streaming Registers"). These segmentation contexts must be allocated to sources by system software.

For efficiency, information as to which block of the PDU is contained in a specific packet is not included in the header. This requires that the transmitter issue the sequence starting with the first block of the PDU and proceeding sequentially through the PDU, and requires the underlying transport fabric to deliver the sequence to the data streaming logical layer in the issued order.

Figure 3-3 shows a 24 byte PDU that is to be segmented for transmission, with an eight byte MTU (note that an eight byte MTU is not permitted in this specification; it is used to simplify the illustration). Since the PDU is divisible by the size specified as the MTU, all data payloads are exactly that size and no padding is necessary. The sender takes byte 0 (the first byte of the PDU) through byte 7 as the data payload to transmit in the start segment. The second data payload consists of bytes 8 through 15, which is transmitted in a continuation segment. The last data payload consists of bytes 16 through 23, which is transmitted in the end segment. Since the data payloads are required to be delivered to the receiver's data management hardware in

order of transmission, the receiver can correctly reassemble the original PDU when all three packets have arrived.

To guarantee the packet ordering, all packets making up an individual PDU and all PDUs in a stream must be in the same transaction request flow, as described in "Section 2.4, Operation Ordering" on page 17.

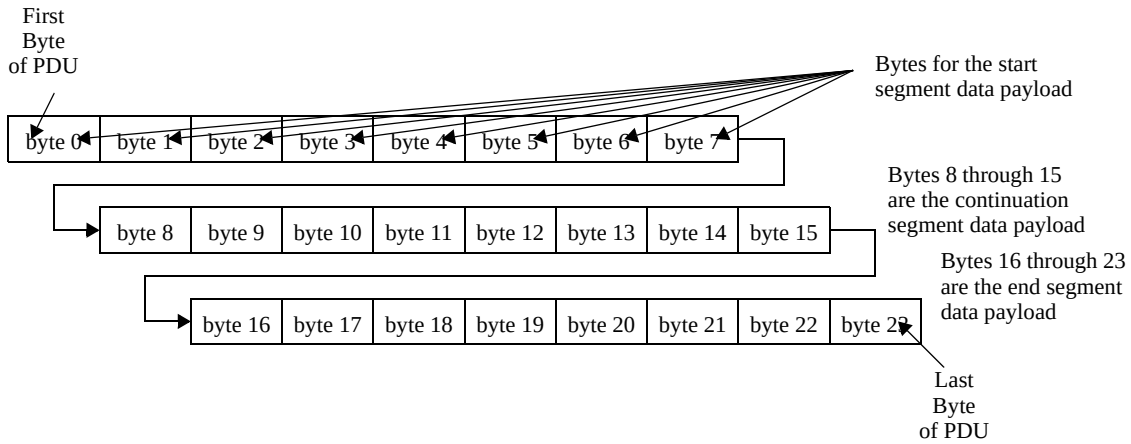


Figure 3-3. PDU Segmentation and Reassembly Example 1

Figure 3-4 shows an example of a similar situation, except that this time the PDU is 21 bytes. In this case, the end segment has a data payload that is less than the specified MTU, and also has a pad byte to round out the data payload to be a multiple of half-words. A bit in the end segment (the "P" bit) indicates the presence of the pad byte. An additional bit (the "O" bit) indicates that the data payload has an odd number of half-words and is therefore oddly aligned. The number of half-words in the data payload as well as the presence of a pad byte can be determined from a PDU length field contained in the end segment header.

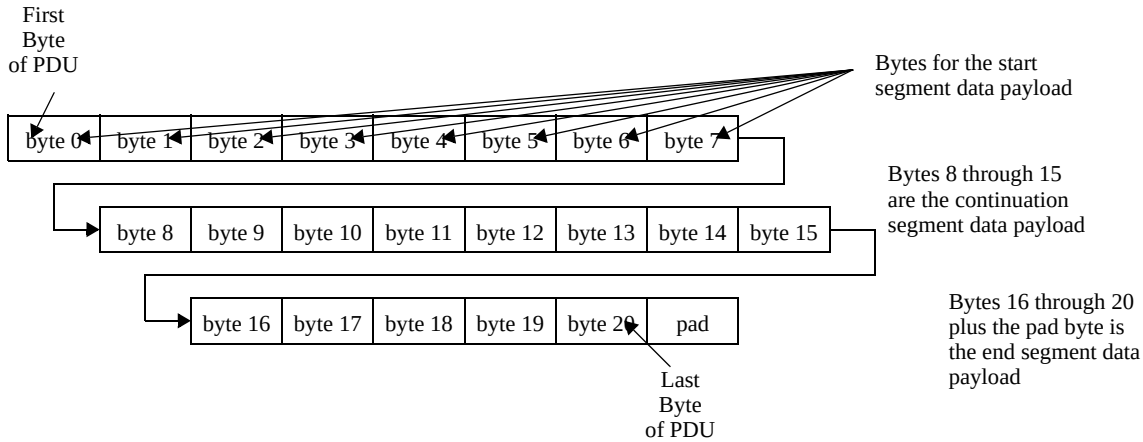


Figure 3-4. PDU Segmentation and Reassembly Example 2

3.2.5 Rules for Segmentation and Reassembly

Segmentation (source)

1. In order to limit implementation complexity due to possible PDU ordering issues, only one PDU from a given stream may be segmented at a time.
2. Segments are filled with bytes from the PDU in order as shown in Figure 3-3 and Figure 3-4.
3. The first segment is marked as start segment (see section 4).
4. The start segment is filled to the end of the PDU data or to the MTU size.
5. If the end of the PDU data is encountered, the start segment then re-marked as a single segment.
6. If the start segment reaches MTU size (and there is remaining PDU data), the start segment is encapsulated, and a continuation segment is opened.
7. Continuation segments are filled to MTU size from the PDU data, in order.
8. When the end of PDU data is encountered, the segment is marked as the end segment. The end segment data payload size may be less than or equal to the MTU size.
9. If the source wishes to abort a PDU transmission, it sends an end segment with no data payload and with the length field set to zero.

Reassembly (destination)

1. Upon receiving a segment with a start bit, the reassembly unit opens a “context” containing the virtual stream ID and associates it with the segmentation context (consisting of the source ID and the flowID).
2. The reassembly process transfers the entire payload into the reassembly buffer in order. The amount of data transferred is counted for comparison to the length field.
3. If the packet is a single segment, the amount of payload data must be equal to or less than the MTU size or the PDU is defective.
4. If the packet is a start segment and the payload data does not match the MTU size the PDU is defective.
5. Reassembly continues with continuation packets. All continuation packets must match the MTU size or the PDU is defective. All data transferred to the reassembly buffer is counted.
6. An end segment terminates the reassembly process. An end segment may be received immediately after a start segment. The data payload size must be less than or equal to the MTU size or the PDU is defective. The data from the end segment is transferred according to the data payload size and counted.
7. Once all the data has been reassembled, the length (provided by the end segment packet header) is checked against the received data count. A mismatch indicates a lost continuation segment and the PDU is defective.

8. Receiving a continuation or end segment on a closed context indicates a lost start segment and the PDU is defective.
9. Receiving a start or single segment on an open context indicates a lost end segment and the PDU is defective. The existing context is closed, and the new context is opened.

In all cases, a defective PDU results in discarding the entire PDU. The method used for reporting the discard event is beyond the scope of this specification. It may be desirable for a destination to have a time-out as part of the lost packet detection mechanism, but the definition and time interval are also outside of the scope of this specification.

3.3 Class of Service and Traffic Streams

A virtual stream ID is partitioned into three pieces as previously discussed: port (identified by the source/destination ID), class (the cos field), and the stream identifier (the streamID field). These fields form a specific hierarchy for transitioning packets from highly individualized streams to coarser groupings of traffic. At the fabric ingress, egress, and potentially at interim points (where competition for resources may occur) the traffic may be resegregated and queued by class. In the packet transport fabric, switching is done by destination ID and the mapped flowID, as described in Section 2.4. The full class of service identifier (CoS ID) is a subset of the VSID. It consists of the source/destination ID (or ingress/egress port) plus the cos field.
Ingress queuing should be based on: **destination ID+cos**
Egress queuing should be based on: **source ID+cos**
as shown in Figure 3-5.

Including the source or destination ID in the CoS ID allows the class of service to be specific to the source and destination pairing.

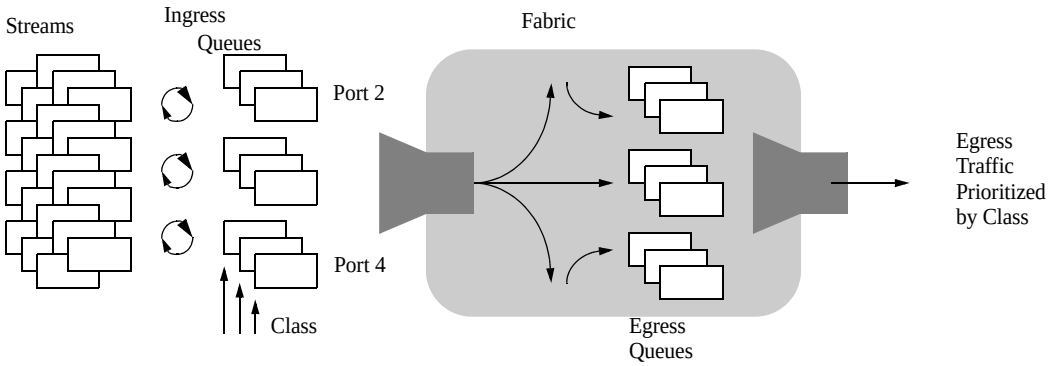


Figure 3-5. Traffic Sorting Based on CoS ID

The cos field shall be used beginning with the MSB (bit 0) using the necessary number of bits for the number of classes supported.

Bit 0 - 2 Classes of Service

Bits 0, 1 - 4 Classes of Service

Bits 0, 1, 2 - 8 Classes of Service supported

etc.

Chapter 4 Packet Format Descriptions

4.1 Introduction

This chapter contains the definition of the data streaming packet format.

4.2 Type 9 Packet Format (Data-Streaming Class)

The type 9 packet format is the DATA STREAMING transaction format. Type 9 packets always have a data payload, unless terminating the PDU. Unlike other RapidIO logical specifications, the data payload length is defined as a multiple of half-words rather than double-words. A pad bit allows a sender to transmit an odd number of bytes in a packet. An odd bit indicates that the data payload has an odd number of half-words. This bit makes it possible for the destination to determine the end of a data payload if packet padding is done by the underlying transport. An extended header bit allows future expansion of the functionality of the type 9 packet format.

Definitions and encodings of fields specific to type 9 packets are provided in Table 4-1.

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

Field	Definition
cos	class of service - This field defines the class of service to be applied by the destination end point (and possibly intervening switch processing elements) to the specified traffic stream.
S	Start - If set, this packet is the first segment of a new PDU that is being transmitted. The new PDU is identified by the combination of the source of the packet and the flowID.
E	End - If set, this packet is the last segment of a PDU that is being transmitted. Both S and E set indicates that the PDU is fully contained in a single packet.
rsrv	Reserved - Assigned to logic 0s by the sender, ignored by the receiver
xh	Extended header - There is an extended header on this packet. Currently there are no defined extended header formats. It is always assigned to 0b0 for type 9 packets.
O	Odd - If set, the data payload has an odd number of half-words
P	Pad - If set, a pad byte was used to pad to a half-word boundary

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

Field	Definition
streamID	traffic stream identifier - This is an end to end (producer to consumer) traffic stream identifier.
length	PDU length - This is the length in bytes of the segmented PDU. 0x0000 - 64kbytes 0x0001 - 1 byte 0x0002 - 2 bytes 0x0003 - 3 bytes ... 0xFFFF - 64kbytes - 1

Table 4-1 details the O and P bit combinations.

Table 4-2. Specific Field Definitions and Encodings for Type 9 Packets

O bit	P bit	Definition
0b0	0b0	Even number of half-words and no pad byte
0b0	0b1	Even number of half-words and a pad byte
0b1	0b0	Odd number of half-words and no pad byte
0b1	0b1	Odd number of half-words and a pad byte

There are three type 9 packet headers, determined by the value of the Start and End bits, which determine if the header is a Start/Single header, a Continuation header, or an End header. The following set of figures shows examples of type 9 packets. Field sizes are specified in bits.

Figure 4-1 is an example of a Single Segment type 9 packet with all of its fields. The data payload size may or may not match the MTU size, so *n* and *m* are determined by the size of the PDU itself. In this example, the data payload is un-padded and there are an even number of half-words. The value 0b1001 in Figure 4-1 specifies that the packet format is of type 9. This is the only type 9 packet that has the xh field.

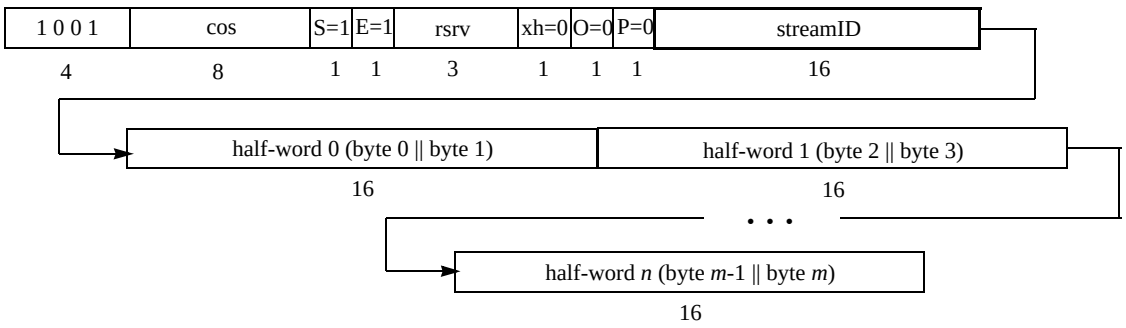


Figure 4-1. Single Segment Type 9 Packet Bit Stream Format Example

Figure 4-2 is an example of a Start Segment type 9 packet with all of its fields. The data payload that matches the MTU, so *n* and *m* are determined by the MTU size. The value 0b1001 in Figure 4-2 specifies that the packet format is of type 9.

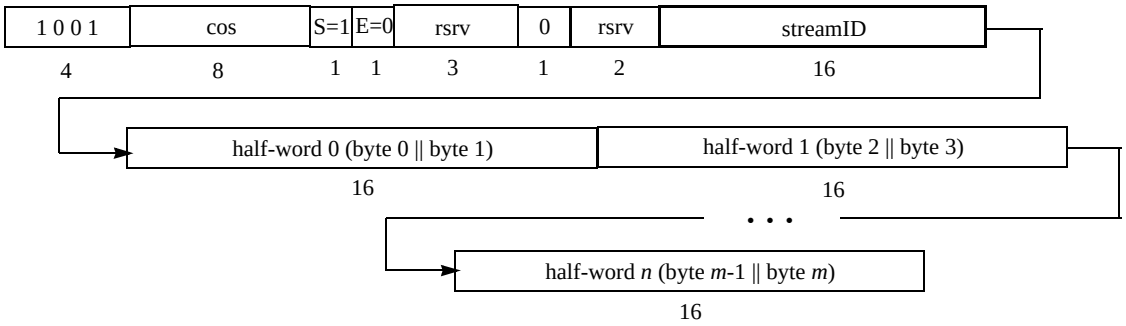


Figure 4-2. Start Segment Type 9 Packet Bit Stream Format Example

Figure 4-3 is an example of a Continuation Segment type 9 packet with all of its fields. The size of the data payload must match the MTU size. The half-words (and correspondingly, bytes) are contiguous in the manner shown in the preceding examples. The value 0b1001 in Figure 4-3 specifies that the packet format is of type 9.

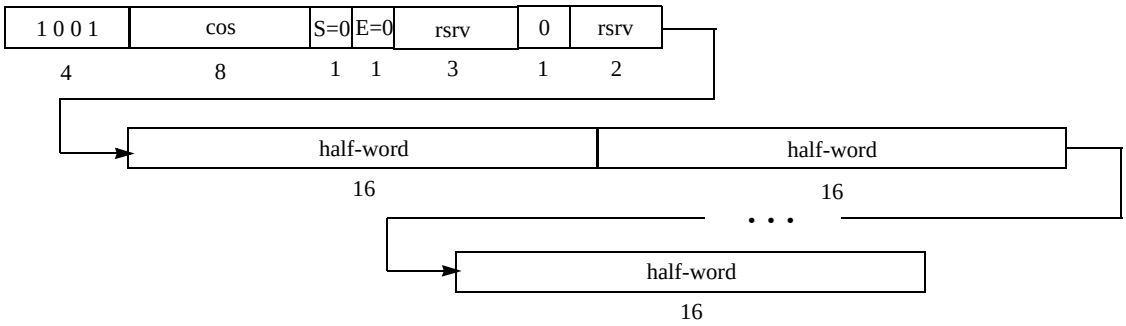


Figure 4-3. Continuation Segment Type 9 Packet Bit Stream Format Example

Figure 4-4 is an example of an End Segment type 9 packet with all of its fields. The size of the data payload is determined by the remainder of the size of the PDU (the length field) divided by the size of the MTU. For convenience at the destination, the O and P bits are used as they are for a single segment. In this example, the data payload size does not match the PDU size, has a pad byte, and is an odd number of half-words. The half-words (and correspondingly, bytes) are contiguous in the manner shown in the preceding examples. A length value of 0 and no data payload can be used to force the PDU to be discarded. The value 0b1001 in Figure 4-4 specifies that the packet format is of type 9.

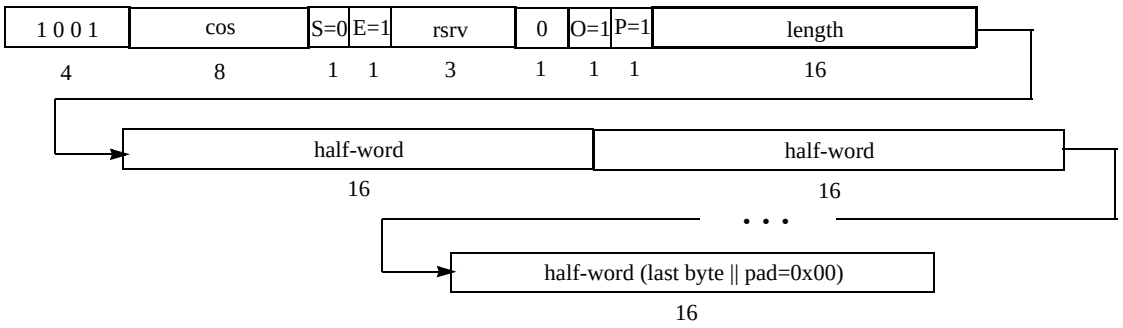


Figure 4-4. End Segment Type 9 Packet Bit Stream Format

4.3 Type 9 Extended Packet Format (Extended Data-Streaming Class)

The type 9 extended packet format is intended to be used for advanced traffic management and other possible future features of the data streaming specification. The format is undefined.

Chapter 5 Data Streaming 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, physical, and extension 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 *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. Data Streaming Register Map

Configuration Space Byte Offset	Register Name
0x0-14	Reserved
0x18	Source Operations CAR
0x1C	Destination Operations CAR
0x20–38	Reserved
0x3C	Data Streaming Information CAR

Table 5-1. Data Streaming Register Map (Continued)

Configuration Space Byte Offset	Register Name
0x40–44	Reserved
0x48	Data Streaming Logical Layer Control CSR
0x4C–FC	Reserved
0x100–FFFC	Extended Features Space
0x10000–FFFFFC	Implementation-defined Space

5.3 Reserved Register and Bit 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.

5.4 Capability Registers (CARs)

Every processing element shall contain a set of registers that allows an external processing element to determine its capabilities through maintenance read operations. 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 and Word 0 respectively the most significant bit and word.

5.4.1 Source Operations CAR (Configuration Space Offset 0x18)

This register defines the set of RapidIO data streaming 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. The Source Operations CAR is applicable for end point devices only. RapidIO switches shall be able to route any packet.

Table 5-3. Bit Settings for Source Operations CAR

Bit	Field Name	Description
0–12	—	Reserved
13	Data-streaming	PE can support a data streaming operation
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 data streaming operations that can be supported by this processing element; see Table 5-4. It is required that all processing elements can respond to 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
0-12	—	Reserved
13	Data-streaming	PE can support a data streaming operation
14-15	Implementation Defined	Defined by the device implementation

Table 5-4. Bit Settings for Destination Operations CAR (Continued)

Bit	Field Name	Description
16-29	—	Reserved
30-31	Implementation Defined	Defined by the device implementation

5.4.3 Data Streaming Information CAR (Configuration Space Offset 0x3C)

This register defines the data streaming capabilities of a processing element. It is required for destination end point devices.

Table 5-5. Bit Settings for Data Streaming Information CAR

Bit	Field Name	Description
0–15	MaxPDU	Maximum PDU - The maximum PDU size in bytes supported by the destination end point 0x0000 - 64kbytes 0x0001 - 1 byte 0x0002 - 2 bytes ... 0xFFFF - 64kbytes - 1
16–31	SegSupport	Segmentation Support - The number of segmentation contexts supported by the destination end point 0x0000 - 64k segmentation contexts 0x0001 - 1 segmentation context 0x0002 - 2 segmentation contexts ... 0xFFFF - 64k - 1 segmentation contexts

5.5 Command and Status Registers (CSRs)

A processing element shall contain a set of command and status registers (CSRs) that allows an external processing element to control and determine the status of its internal hardware. All registers are 32 bits wide and are organized and accessed in the same way as the CARs. Refer to Table 5-2 for the required behavior for accesses to reserved registers and register bits.

5.5.1 Data Streaming Logical Layer Control CSR (Configuration Space Offset 0x48)

The Data Streaming Logical Layer Control CSR is used for general command and status information for the logical interface.

Table 5-6. Bit Settings for Data Streaming Logical Layer Control CSR

Bit	Field Name	Description
0-23	—	Reserved
24-31	MTU	Maximum Transmission Unit - controls the data payload size for segments of an encapsulated PDU. Only single segment PDUs and end segments are permitted to have a data payload that is less this value. The MTU can be specified in increments of 4 bytes. Support for the entire range is required. 0b0000_0000 - reserved ... 0b0000_0111 - reserved 0b0000_1000 - 32 byte block size 0b0000_1001 - 36 byte block size 0b0000_1010 - 40 byte block size ... 0b0100_0000 - 256 byte block size 0b0100_0001 - Reserved ... 0b1111_1111 - Reserved All other encodings reserved

Annex A VSID Usage Examples

A.1 Introduction

The virtual stream identification (VSID) mechanism provides multiple features condensed in a single 32 bit key. These features include:

- A mechanism to manage traffic for ingress to the fabric
- A mechanism to manage traffic in transit within the fabric
- A protocol independent tag to reclassify packets on fabric egress
- A flexible "sub-port" addressing mechanism
- Independence in buffer management

A.2 Background

The VSID is a composite of the port, class, and streamID fields as described in Section 3.2.2. The port address used in the VSID is either the destination ID or the source ID depending on which side of the fabric the packet is on. At the ingress to the fabric (source) the destination IDs are unique. At the egress from the fabric, the source IDs are unique.

By including the source/destination IDs in the VSID, these keys are unique for each source and destination pairing. This allows the other fields (class and streamID) to be set up independently without consideration of how these fields are used with any other port pairings.

The usage of the VSID can vary depending on the sophistication of the fabric and the demands of the application, from very simplistic port or queue steering to conveying significant amounts of information (requiring intensive computation) as to the content of the PDU.

A.3 Packet Classification

All PDUs require some form of classification for ingress to the fabric. Fields in the PDU specific to the protocol are examined and routing information is produced. The VSID produced is a 32 bit tag as opposed to just a port address. At the destination, this 32 bit tag can be used to re-associate the PDU with a target buffer. This can be done by direct addressing, or using a single key table lookup.

This mechanism provides a finely grained and protocol independent way to sort traffic, and a virtual mechanism for buffer pool management. Without a virtual tag, the packet would have to undergo a re-classification based on the protocol specific portion of the PDU. In multi-service platforms, this could involve numerous and elaborate processes, duplicating what was already done at the source.

The following sections illustrate in degrees of increasing complexity, the versatility of the VSID scheme.

A.3.1 Sub-port Addressing at the Destination

The simplest use of the VSID is to de-multiplex the traffic into coarse sub-ports at the destination. These may be to separate traffic by protocol, or into multiple sub-ports of the same protocol.

A.3.1.1 DSLAM application

Assume that each line card contain 128 user ports. The system could expose each of these as independent destinations to the RapidIO fabric, requiring the use of an excessively large number of destination IDs in the system, and imposing the associated cost in overhead. Alternatively the ATM traffic can be encapsulated into 128 VSIDs, one for each port. The line card would then expose a single port to the RapidIO fabric. The VSID would be used as the address to fan out the traffic on various UTOPIA busses to the user ports. This also has an advantage for fault recovery. Should a line card fail, a single port entry in routing tables in the fabric needs to be updated rather than all 128 sub-ports.

A.3.1.2 VOIP application

The VSID can be used to separate the traffic into just 2 channels, one destined for a control processor to handle control messages and one channel that goes to a network processor to be distributed to DSPs. The VSID could contain the address of the target DSP, to further off-load the network processor on distribution. The VSID could also contain the user channel within the DSP de-multiplexing the traffic even further.

A.3.2 Virtual Output Queuing - Fabric On-ramp

Applications involving larger numbers of flows can use the class field to regulate the ingress to the fabric (known as virtual output queuing). For example, the RapidIO fabric interface could contain 256 queues for 64 destination ports with 4 traffic classes. Traffic for each destination of the same class is fairly weighted. The weighting between classes can be application unique.

The traffic is kept sorted by destination. If traffic was just dumped into 4 queues, and a destination port was to fail, the traffic could head of line block the traffic to the other ports, or it would have to be discarded while the port is being recovered or

re-routed. By keeping the traffic sorted by destination at the fabric ingress, that destination can be re-routed with minimal traffic loss.

Virtual output queuing can be expanded to 2K or even 16K buffers depending on how large the fabric is, and how many different traffic classes are involved. This fabric ingress management can be a simple mechanism to add some quality of service to a system using the destination ID and the class portion of the VSID. Note that this can be done separately from the use of the streamID at the destination for de-multiplexing.

A.4 System Requirements

The use of the VSID is determined by all three elements in a system, the source, the fabric, and the destination. This section contains descriptions of some example source devices.

A.4.1 UTOPIA to RapidIO ATM bridge

The UTOPIA to RapidIO ATM bridge classifies traffic using the VPI field as the destination port, and the VCI as a sub-port address. It maps all (type 9) traffic to a single RapidIO flow, setting the class to 0 and the streamID to the VCI. The fabric switches on flows. The destination uses the streamID portion of the VSID as a hard-wired sub-port address.

A.4.2 Network processor

The network processor (NP) contains a OC-48 link aggregating traffic to and from multiple 1MB/s ports distributed on line cards. The NP classifies traffic for each user into two classes: high priority for voice (using RTP) and low priority for all others. It sets the class field to 0 or 1, the port to the proper line card, and the streamID to the desired sub-port.

A.4.3 CSIX to RapidIO interface

The CSIX packet contains the destination and class fields (the source is a preset parameter in the interface chip). The streamID is the first 16 bits of the CSIX payload. The RapidIO packet is easily constructed from this information. The fabric interface contains multiple virtual output queues, 2 per destination port. Since the CSIX to NP interface is also a segmented interface, PDUs are reassembled in the virtual queues until enough information is available to form the required MTU on the RapidIO fabric.

The fabric maps the class to a higher or lower priority flow. The destination uses the streamID to map the traffic to the correct user sub-port. Each sub-port contains two class queues to collect traffic as it is reassembled.

A.4.4 10Gb Metropolitan Area Network interface

A specialized classification processor creates the 32 bit VSID based on IP, TCP/UDP, and application information. The tag is prepended to a SPI4.2 packet. The interface to the fabric is a SPI4.2 to RapidIO bridge, which contains virtual output queues.

The destination is a processor that only supports memory and IO logical transactions. The RapidIO to processor interface bridge contains the segmentation and reassembly buffers and look up tables and associated engines that maps the VSID to a DMA buffer address (and vice-versa).

The system contains multiple of these processing cards to support address translation, encryption, or firewall processing. The source classifies traffic based on which of these applications applies. A connection is created by allocating a buffer address in the destination, and assigning a streamID. The source table is created with the search tree requirements for the protocol, and setting up the VSID result.

Destinations may use the VSID in a hard-wired method, or it may be a flexible mapping to virtual buffers. In either case, the source must be flexible to assign the VSID according the destination's needs. This is normally not an issue as the source needs to classify the packet to determine the destination anyway. The use of the VSID can be to separate the traffic by protocol, sub-port, service class, or into as many virtual queues as necessary. If the destination is managing a large number of buffers, the VSID allows the destination to use a single protocol independent key to re-map the traffic and completely abstract any buffer management.

Glossary of Terms and Abbreviations

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

B

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

C

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

Class of service - (cos) a term used to describe different treatment (quality of service) for different data streams. Support for class of service is provided by a class of service field in the data streaming protocol. The class of service field is used in the virtual stream ID and in identifying a virtual queue.

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.

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

Double-word. An eight byte or 64 bit quantity, aligned on eight byte boundaries.

E

Egress - Egress is the device or node where traffic exits the system. The egress node also becomes the destination for traffic out of the RapidIO fabric. The terms egress and destination may or may not be used interchangeably when considering a single end to end connection.

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

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

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

External processing element. A processing element other than the processing element in question.

F

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

H

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

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

I

Ingress - Ingress is the device or node where traffic enters the system. The ingress node also becomes the source for traffic into the RapidIO fabric. The terms ingress and source may or may not be used interchangeably when considering a single end to end connection.

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

I/O. Input-output.

O

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

P

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

PDU. Protocol Data Unit, the OSI term for a packet.

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

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

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

R

Receiver. The RapidIO interface input port on a processing element.

S

SAR. Segmentation and Reassembly, a mechanism for encapsulating a PDU within multiple packets.

Segmentation. A process by which a PDU is transferred as a series of smaller *segments*.

Segmentation Context. Information that allows a receiver to associate a particular packet with the correct PDU.

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

Sequence. Sequentially ordered, uni-directional group of messages that constitute the basic unit of data delivered from one end point to another.

StreamID. A specific field in the data streaming protocol that is combined with the data stream's transaction request flow ID and the sourceID or destinationID from the underlying packet transport fabric to form the virtual stream ID.

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

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

T

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

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

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

V

Virtual Stream ID (VSID). An identifier comprised of several fields in the protocol to identify individual data streams.

Virtual input Queue (ViQ), Virtual output Queue (VoQ). An intermediate point in the system where one or more virtual streams may be concentrated.

W

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

Blank page

Blank page

RapidIO™ Interconnect Specification Part 11: Multicast Extensions Specification

Rev. 1.3.1, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.3	First release	06/09/2004
1.3.1	Technical changes: the following new features showings: 04-08-00015.003 Converted to ISO-friendly templates	02/23/2005
1.3.1	Removed confidentiality markings for public release	06/07/2005

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

Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

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 11: Multicast Extensions Specification*. The goal of this specification is to add a simple mechanism to the existing RapidIO specifications that provides multicast functionality to a system. This specification assumes that the reader has a working understanding of the other RapidIO specifications.

1.1 Overview

The concept of duplicating a single message and sending it to multiple selected destinations is known as ‘multicast’, and is found to be useful in many computing systems. This can be accomplished by a variety of means. The most efficient and highest performance method is to have hardware support for the duplication of messages.

Within a RapidIO system, the ability to duplicate messages should scale with the number of end points in a system. Since the number of end points scales with the number of switches in the system, the multicast extensions are defined for switches only and end points are largely unaffected. Possible end point design considerations are described in Annex A.

The multicast specification is limited to request transactions that do not require responses, for example, *RapidIO Part 1: Input/Output Logical Specification* SWRITE transactions. This is because implementing support for collecting the response transactions within a switch device, which are typically not aware of RapidIO logical layer protocols, is problematic and complex.

The ability for a switch to send a single message to a variety of destinations can be implemented in a wide variety of ways, depending on system needs. There are two reasons, however, that motivate definition of a common interface and behavior for multicast in a system. Without a standard interface and behavioral definition, the wide variety of possible implementations would not allow a common multicast software driver to exist. The second reason is that without a standard definition for interface and behavior it is impossible to guarantee inter-operability of different components which support multicast.

In defining a common interface for a wide variety of implementations, it is necessary

to define the standard interface with some level of abstraction in order to avoid limiting implementation flexibility. Therefore, several examples of the use of the interface have been included.

1.2 Requirements

The multicast mechanism shall fulfill the following goals:

- Simple - excess complexity will not gain acceptance
- Compact - Does not cost excessive silicon area in a switch
- Robust - same level of protection and recovery as the rest of RapidIO
- Scalable - must be able to extend to multi-layer switch systems
- Compatibility with all physical layers

Chapter 2 Multicast Extensions Behavior

2.1 Introduction

This chapter describes the multicast extensions rules of operation in a RapidIO system. A RapidIO switch which does not support multicast can co-exist in a RapidIO fabric with other switches that do support multicast. The only requirement is that the switch be capable of routing the destination IDs used for multicast transactions.

2.2 Packet Replication

A RapidIO multicast operation consists of the replication of a single packet so that it can be received by multiple end points. This replication is performed by the switch devices in the fabric rather than by the end point itself, so that the capability to replicate packets expands with the number of switches (and hence possible end points) in a system. Each switch may be individually programmed to control which egress ports of the switch the replicated packets are sent to, and thus indirectly which specific set of end point devices receive the replicated packet. The packets themselves are not modified by the replication process, merely transmitted out through the appropriate ports.

This specification only addresses multicasting request packets for transactions which do not require responses. This greatly simplifies multicast support for RapidIO switches, which will therefore have no need to aggregate responses from other types of RapidIO operations. Examples of transactions which can be multicast are I/O logical specification NWRITE and SWRITE transactions. Multicasting transactions which require responses have implementation defined behavior.

2.3 Multicast Operation

Multicast operations have two control value types - **multicast masks** and **multicast groups**. The set of target end points which all receive a particular multicast packet is known as a **multicast group**. Each multicast group is associated with a unique destination ID. The destination ID of a received packet allows a RapidIO switch device to determine that a packet is to be replicated for a multicast.

A **multicast mask** is a value that controls which egress ports one or more multicast groups are associated with. Conceptually, a multicast mask is a register with one

enable bit for each possible switch egress port. There is one set of multicast masks for the entire switch. All multicast masks in a switch are assigned unique sequential ID numbers beginning with 0. Figure 2-1 shows an example of the use of multicast in a RapidIO system.

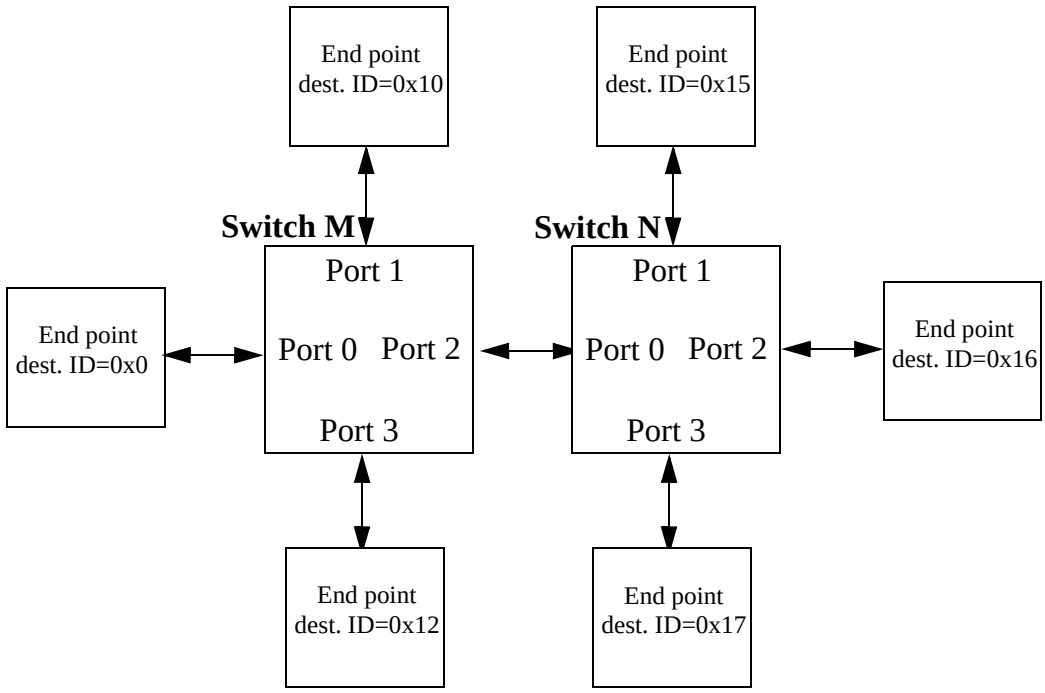


Figure 2-1. Multicast System Example

In this example, the end point assigned destination ID 0x0 uses destination ID 0x80 to perform multicast operations to the multicast group comprised of end points 0x10, 0x15, 0x16, and 0x17, arbitrarily called group A. Software configures the switch devices in the fabric to **associate** the destination IDs that represent multicast groups with multicast masks. For Figure 2-1 switch M associates destination ID 0x80 with egress ports 1 and 2, and switch N associates destination ID 0x80 with ports 1, 2, and 3. Figure 2-2 shows a possible relationship between the multicast group, the multicast masks for the switches, and the global system address map.

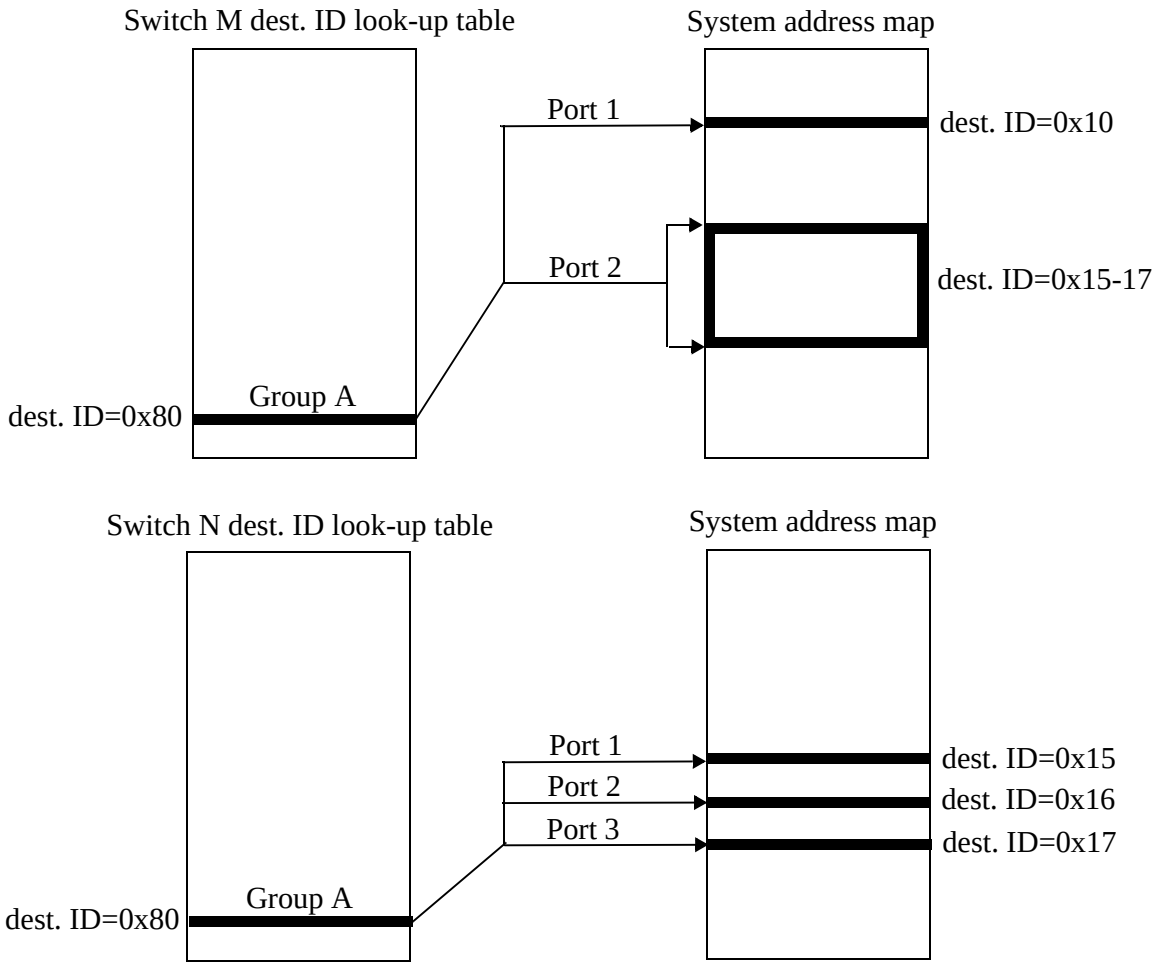


Figure 2-2. Multicast Association Example

Configuring a RapidIO switch to replicate packets for a multicast group is a two-step process. First, a list of egress ports is set in a multicast mask list. Second, one or more destination IDs which represent the multicast groups are associated with the multicast mask in the switch. During normal system operation, any time a switch receives a packet with a destination ID which has been associated with a multicast mask it will send that packet to all egress ports enabled by that multicast mask.

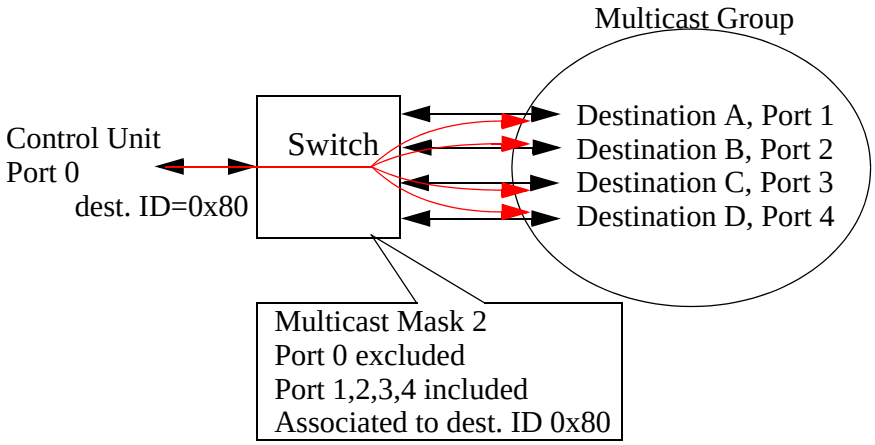


Figure 2-3. Multicast Configuration Example

Figure 2-3 shows a control unit connected to switch port 0 which needs to multicast to destinations A, B, C and D. A multicast mask, in this case arbitrarily picked as multicast mask 2, is set up to select which ports in the switch are part of the multicast group of destinations A, B, C, and D. A destination ID, in this case arbitrarily assigned 0x80, is associated with multicast mask 2 as the destination ID that the control unit should use to multicast to the multicast group. The associate operation is done using the CSRs defined in Chapter 3, “Multicast Extensions Registers”.

The defined CSRs allow a switch to associate destination IDs with multicast masks using a small number of maintenance write operations. The number of unique destination IDs that can be associated with a multicast mask is also defined in a CSR.

While each destination ID is associated with a unique multicast group, the programming model allows a destination ID to be mapped to a different multicast mask for each port on the switch. However, for each port a destination ID can be associated with at most one multicast mask. The last association operation performed for a specific port and destination ID dictates which multicast mask the destination ID is associated with. It is also possible to map a given destination ID to the same multicast mask for all ports.

A RapidIO switch may be capable of supporting large numbers of multicast groups by dedicating a sequential range of destination ID’s to an equal number of sequentially numbered multicast masks. A switch may also be designed which does not require all multicast destination IDs to be sequential. The programming model supports both of these implementations.

A packet will never be multicast back out of the port it was received on even if it is included in the multicast mask for that destination ID. This allows a group of end points which need to multicast to each other to share the same multicast mask. Packets using a multicast mask which has no egress ports selected will be dropped without error notification. A device may have implementation specific error notification in this situation, depending on system requirements.

The default state after a reset for multicast masks is that all multicast masks have no ports selected. Additionally, after reset no associations exist between any multicast group/destination ID and the multicast masks. However, implementation specific capabilities may modify the multicast mask values and associations after reset without software intervention.

For more information and examples on the use of the programming model for multicast refer to Annex B, “Multicast Applications (Informative)”.

2.4 Multicast Transaction Ordering Requirements

RapidIO packets which are in the same multicast group (the same destination ID) with the same flowID and are received on the same ingress port must be multicast on the egress ports in the same order that they were received. There are no ordering requirements between multicast packets and non-multicast packets, or between multicast packets in different multicast groups. Maintaining ordering between transactions in the same transaction request flow for a multicast group allows an application to multicast a completion flag at the end of a potentially large data transfer which was sent to the same multicast group.

Blank page

Chapter 3 Multicast Extensions Registers

3.1 Introduction

This section describes the Multicast Extensions CAR and CSR registers that allow an external processing element to determine if a switch supports the multicast extensions defined in this specification, and to manage the configuration of multicast groups for a switch processing element. This chapter only describes registers or register bits defined by this specification. Refer to the other RapidIO logical, transport, physical, and extension specifications of interest to determine a complete list of registers and bit definitions for a device. All registers are 32-bits and aligned to a 32-bit boundary. The behavior of reserved register bits and register offsets and access rules and requirements are described in the *RapidIO Part 1: Input/Output Logical Specification*.

Table 3-1. Multicast Register Map

Configuration Space Byte Offset	Register Name
0x0-C	Reserved
0x10	Processing Element Features CAR
0x14-2C	Reserved
0x30	Switch Multicast Support CAR
0x34	Reserved
0x38	Switch Multicast Information CAR
0x3C-7C	Reserved
0x80	Multicast Mask Port CSR
0x84	Multicast Associate Select CSR
0x88	Multicast Associate Operation CSR
0x8C–FC	Reserved
0x100–FFFC	Extended Features Space
0x10000–FFFFFC	Implementation-defined Space

3.2 Processing Elements Features CAR (Configuration Space Offset 0x10)

The Processing Elements Features CAR contains 31 processing elements features bits defined in various RapidIO specifications, as well as the Multicast Support bit, defined here.

Table 3-2. Bit Settings for Processing Elements Features CAR

Bit	Name	Reset Value	Description
0-20	-		Reserved (defined elsewhere)
21	Multicast Support	*	Support for multicast extensions 0b0 - Does not support multicast extensions 0b1 - Supports multicast extensions
22-31	-		Reserved (defined elsewhere)

* Implementation dependant

3.3 Switch Multicast Support CAR (Configuration Space Offset 0x30)

The Switch Multicast Support CAR defines support for a simple multicast model and the additional limits on multicast mask resources.

Table 3-3. Bit Settings for Switch Multicast Support CAR

Bit	Name	Reset Value	Description
0	Simple_Assoc	*	Support for a simple multicast association model 0b0 - Does not support simple association 0b1 - Supports simple association If this bit is set, the Block_Assoc bit in the Switch Multicast Information CAR must also be set.
1-31	-		Reserved (defined elsewhere)

* Implementation dependant

3.4 Switch Multicast Information CAR (Configuration Space Offset 0x38)

The Switch Multicast Information CAR defines the methods for associating destination IDs with multicast masks supported by a RapidIO switch device. It also defines the limits on multicast mask resources.

Table 3-4. Bit Settings for Switch Multicast Information CAR

Bits	Name	Description
0	Block_Assoc	Block association support - allows equal sized blocks of destination IDs and multicast masks to be associated with each other with a single operation rather than one at a time. 0b0 - block association is not supported 0b1 - block association is supported If the Simple_Assoc bit in the Switch Multicast Support CAR is set, this bit must also be set.
1	Per_Port_Assoc	Per ingress port association support - allows a destination ID to be associated with a multicast mask on a per-ingress port basis rather than a single association for the entire switch. 0b0 - per port association is not supported 0b1 - per port association is supported
2-15	MaxDestIDAssoc	The maximum number of destination IDs associations per multicast mask 0x0000 - 1 destination ID 0x0001 - 2 destination IDs ... 0x3FFF - 16384 destination IDs
16-31	MaxMcastMasks	The number of multicast egress port masks available. This field also defines the largest block of destination IDs that can be block associated. 0x0000 - [reserved] 0x0001 - 1 multicast mask 0x0002 - 2 multicast masks ... 0xFFFF - 65535 multicast masks

3.5 Multicast Mask Port CSR (Configuration Space Offset 0x80)

The Multicast Mask Port CSR allows configuration of the egress port list for each of the switch's multicast masks.

Writing the Write_to_Verify command sets up a Mcast_Mask and Egress_Port_Num pair to verify. The presence of the specified egress port in the specified multicast mask is indicated by the Port_Present bit on a subsequent read of the register.

Writing the Add_Port or Delete_Port commands adds or deletes the specified egress port to or from the specified multicast mask.

Writing the Add_All_Ports or Delete_All_Ports commands adds or deletes all of the egress ports in the specified multicast mask.

The result of illegal values or combinations for an operation is implementation dependent. For examples of how to use this register, refer to Section 4.2, "Configuring Multicast Masks**".**

Table 3-5. Bit Settings for Multicast Mask Port CSR

Bits	Name	Reset Value	Description
0-15	Mcast_Mask	0x0000	Specifies the multicast mask which is to be modified or queried as determined by the Mask_Cmd field.
16-23	Egress_Port_Num	0x00	Specifies the port number to be added, deleted, or queried with the Mask_Cmd field.
24	-	0b0	Reserved
25-27	Mask_Cmd	0b000	Specifies the mask action on a write. 0b000 - Write_to_Verify 0b001 - Add_Port 0b010 - Delete_Port 0b011 - reserved 0b100 - Delete_All_Ports 0b101 - Add_All_Ports 0b110 - reserved 0b111 - reserved
28–30	-	0b000	Reserved
31	Port_Present	0b0	Indicates the existence of the egress port and multicast mask pair as a result of the last preceding Write_to_Verify command. 0b0 - Port was not enabled as an egress port in the specified multicast mask 0b1 - Port was enabled as an egress port in the specified multicast mask. This bit is reserved on a write.

3.6 Multicast Associate Select CSR (Configuration Space Offset 0x84)

This register specifies the destination ID and multicast mask number for a subsequent associate operation controlled with the Multicast Associate Operation CSR. If block association is supported, this register specifies the start of the block to associate. For examples of how this register is used, refer to Section 4.4, “Configuring Associations”.

Table 3-6. Bit Settings for Multicast Associate Select CSR

Bits	Name	Reset Value	Description
0-7	Large_DestID	0x00	Selects the most significant byte of a large transport destination ID for an association operation
8-15	DestID	0x00	Selects the destination ID for an association operation
16-31	Mcast_Mask_Num	0x0000	Selects the multicast mask number for an association operation

3.7 Multicast Associate Operation CSR (Configuration Space Offset 0x88)

The Multicast Associate Operation CSR specifies three operations for associating destination IDs with multicast masks. The affected destination ID and multicast mask is specified in the Multicast Associate Select CSR. The specified operation is executed when this register is written. When this register is read and the Assoc_Cmd field it set to Write_to_Verify the specified operation is executed and the updated register state is returned. If this register is read and the Assoc_Cmd field is not set to Write_to_Verify the resulting behavior is implementation dependent. Block association operations assign associations sequentially starting with the destination ID and multicast mask specified in the Multicast Associate Select CSR.

Writing the Write_To_Verify command checks for an association between the destination ID and multicast mask specified in the Multicast Associate Select CSR. The result of the check is indicated by the state of the Assoc_Present bit on a read of this register. This command cannot be executed on a block.

Writing the Add_Assoc or Delete_Assoc command adds or deletes the association between the destination ID and the multicast mask (or block of associations, if block association is supported) specified in the Multicast Associate Select CSR.

The result of illegal values or field combinations for an association operation is implementation dependent. For examples of how this register is used, refer to Section 4.4, “Configuring Associations.

Table 3-7. Bit Settings for Multicast Associate Operation CSR

Bits	Name	Reset Value	Description
0-15	Assoc_Blksize	0x0000	This field specifies the number of sequential DestinationIDs to be associated with an equal number of sequential multicast mask numbers if block association is supported. This field is ignored on a Write_to_Verify command. 0x0000 - one association 0x0001 - two sequential associations ... 0xFFFF - 65536 sequential associations
16-23	Ingress_Port	0x00	This field specifies the ingress port association to affect if per-port ingress association is supported
24	Large_Transport	0b0	0b0 - the association is for small transport destination IDs 0b1 - the association is for large transport destination IDs
25-26	Assoc_Cmd	0b00	This field specifies the command to execute when this register is written. 0b00 - Write_To_Verify 0b01 - reserved 0b10 - Delete_Assoc 0b11 - Add_Assoc

Table 3-7. Bit Settings for Multicast Associate Operation CSR (Continued)

Bits	Name	Reset Value	Description
27-30	-	0b0000	reserved
31	Assoc_Present	0b0	This bit contains the result of the last Write_to_Verify command executed. 0b0 - no association present 0b1 - association present This bit is reserved on write.

Chapter 4 Configuration Examples

4.1 Introduction

This chapter provides several examples of how to use the multicast programming interface. The given examples build upon each other while proceeding through the sections. References to the order of operations within the examples run from the top of a list to the bottom unless otherwise stated.

Initially assume a switch with 8 ports which supports 4 or more multicast masks with two or more destination IDs allowed per multicast group so that a total of 8 destination IDs minimum can be associated with the multicast masks. The system has the following requirements:

- Three sources of traffic (ports 0, 1, and 2) must be multicast to two destinations (ports 6 and 7).
- Three ports (ports 3, 4 and 5) need to multicast signals between each other.
- All ports occasionally need to multicast to every other port.

Assume that the switch does not require any other multicast functions and therefore multicast masks 0, 1, and 2 will be used.

4.2 Configuring Multicast Masks

This section discusses assigning an egress port list to a multicast mask.

4.2.1 Clearing Multicast Masks

Suppose that the state of the multicast masks is unknown, and therefore the masks must be cleared before being configured. In order to clear the masks the following register accesses are made. (The accesses to the Multicast Mask Port CSR can be performed in any order.)

- Remove all ports from multicast mask 0
  - — Write the value 0x0000_0040 to the Multicast Mask Port CSR
- Remove all ports from multicast mask 1
  - — Write the value 0x0001_0040 to the Multicast Mask Port CSR
- Remove all ports from multicast mask 2
  - — Write the value 0x0002_0040 to the Multicast Mask Port CSR

4.2.2 Assigning Ports to Multicast Masks

To configure mask 0 to multicast to ports 6 and 7, mask 1 to multicast to ports 3, 4 and 5, and mask 2 to multicast to every port, requires the following series of register accesses. (The accesses to the Multicast Mask Port CSR can be performed in any order.)

- Add port 6 to multicast mask 0
  - — Write the value 0x0000_0610 to the Multicast Mask Port CSR
- Add port 7 to multicast mask 0
  - — Write the value 0x0000_0710 to the Multicast Mask Port CSR
- Add port 3 to multicast mask 1
  - — Write the value 0x0001_0310 to the Multicast Mask Port CSR
- Add port 4 to multicast mask 1
  - — Write the value 0x0001_0410 to the Multicast Mask Port CSR
- Add port 5 to multicast mask 1
  - — Write the value 0x0001_0510 to the Multicast Mask Port CSR
- Add all ports to multicast mask 2
  - — Write the value 0x0002_0050 to the Multicast Mask Port CSR

4.2.3 Removing a Port from a Multicast Mask

Suppose that the device attached to port 4 needs to be removed from the system. The following register accesses are used to modify multicast masks 1 and 2 to stop port 4 from being a multicast destination. (The accesses may be performed in any order.)

- Remove port 4 from multicast mask 1
  - — Write the value 0x0001_0420 to the Multicast Mask Port CSR
- Remove port 4 from multicast mask 2
  - — Write the value 0x0002_0420 to the Multicast Mask Port CSR

4.2.4 Querying a Multicast Mask

In this section suppose that a system designer needs to determine which of the 8 ports are included in multicast mask 2. The following accesses are to be performed to provide this information. (In each case, the write operation setting up the ‘Write to Verify’ operation must be performed before the subsequent read to check the Port Present bit status. The individual multicast masks may be queried in any order.)

- Verify that port 0 is included in mask 2
  - — Write the value 0x0002_0000 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0001 from the Multicast Mask Port CSR

- Verify that port 1 is included in mask 2
  - — Write the value 0x0002_0100 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0101 from the Multicast Mask Port CSR.
- Verify that port 2 is included in mask 2
  - — Write the value 0x0002_0200 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0201 from the Multicast Mask Port CSR
- Verify that port 3 is included in mask 2
  - — Write the value 0x0002_0300 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0301 from the Multicast Mask Port CSR
- Verify that port 4 is not included in mask 2
  - — Write the value 0x0002_0400 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0400 from the Multicast Mask Port CSR
- Verify that port 5 is included in mask 2
  - — Write the value 0x0002_0500 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0501 from the Multicast Mask Port CSR
- Verify that port 6 is included in mask 2
  - — Write the value 0x0002_0600 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0601 from the Multicast Mask Port CSR
- Verify that port 7 is included in mask 2
  - — Write the value 0x0002_0700 to the Multicast Mask Port CSR
  - — Read the value 0x0002_0701 from the Multicast Mask Port CSR

4.3 Simple Association

If the Simple_Assoc bit is set in the Switch Multicast Support CAR, the device supports the simple multicast programming model. This model allows for basic multicast support for devices with a limited number of multicast masks, and requires a fixed relationship between those masks and sequential multicast groups.

4.3.1 Restrictions on Block Size

If the Simple_Assoc bit is set the device has a limited number of masks. Therefore, the number of sequential associations equals the maximum number of masks.

The Assoc_BlkSize field in the Multicast Associate Operation CSR must be set to the value of (MaxMCastMasks - 1). The MaxMCastMasks field is in the Switch Multicast Information CAR.

4.3.2 Restrictions on Block Associate

If the Simple_Assoc bit is set, non-block associations are precluded.

4.3.3 Restrictions on Associations

If the Simple_Assoc bit is set the device requires a fixed relationship between the sequential mask numbers and sequential destination IDs. This must be taken into account when the masks are associated.

The Multicast Associate Select CSR is set with the Mcast_Mask_num value set to 0x0000 and the Large_DestID and DestID fields set to an integer multiple of the MaxMCastMasks value.

Hardware that sets the new Simple_Assoc CAR bit could implement a single block associate for all of the masks that it supports with the requirement that they all be sequential destination IDs.

4.4 Configuring Associations

This section describes how to associate destination IDs with multicast masks, including examples of how to use the block association and per-port association functions.

4.4.1 Basic Association

For the assumed system it is now necessary to associate a destination ID with each multicast mask from the preceding examples. How this can be accomplished may vary depending on the capabilities of the switch. For this section, assume that neither block association nor per-ingress-port association is supported by the switch.

Following upon the previous example, assume the following additional system requirements.

- the 16 bit destination ID 0x1234 needs to be associated with multicast mask 0.
- the 8 bit destination ID 0x44 needs to be associated with multicast mask 1.
- the 16 bit destination ID 0xFEED needs to be associated with multicast mask 2.

In order to accomplished the desired associations, the following register accesses are required. (The individual association operations can be performed in any order.)

- Set up the operation to associate destination ID 0x1234 with multicast mask 0
  - — Write the value 0x1234_0000 to the Multicast Associate Select CSR
- Associate destination ID 0x1234 with multicast mask 0
  - — Write the value 0x0000_00E0 to the Multicast Associate Operation CSR
- Set up the operation to associate destination ID 0x44 with multicast mask 1

— Write the value 0x0044_0001 to the Multicast Associate Select CSR

- Associate destination ID 0x44 with multicast mask 1

  — Write the value 0x0000_0060 to the Multicast Associate Operation CSR

- Set up the operation to associate destination ID 0xFEED with multicast mask 2

  — Write the value 0xFEED_0002 to the Multicast Associate Select CSR

- Associate destination ID 0xFEED with multicast mask 2

  — Write the value 0x0000_00E0 to the Multicast Associate Operation CSR

4.4.2 Using Per-Ingress Port Association

For the associations discussed in the preceding section, if the switch supports per-ingress-port association (destination IDs are associated with multicast masks on a per ingress port basis), the required programming operations change. The associations for each multicast mask are grouped into a write to the Multicast Associate Select CSR, followed by a write to the Multicast Associate Operation CSR for each ingress port that must be aware of the association. (The writes to the Multicast Associate Operation CSR can occur in any order but must occur after the related writes to the Multicast Associate Select CSR. The individual association operations can be performed in any order.)

- Set up the operation to associate destination ID 0x1234 with multicast mask 0

  — Write the value 0x1234_0000 to the Multicast Associate Select CSR

- Associate destination ID 0x1234 with multicast mask 0 on ingress port 0

  — Write the value 0x0000_00E0 to the Multicast Associate Operation CSR

- Associate destination ID 0x1234 with multicast mask 0 on ingress port 1

  — Write the value 0x0000_01E0 to the Multicast Associate Operation CSR

- Associate destination ID 0x1234 with multicast mask 0 on ingress port 2

  — Write the value 0x0000_02E0 to the Multicast Associate Operation CSR

- Set up the operation to associate destination ID 0x44 with multicast mask 1

  — Write the value 0x0044_0001 to the Multicast Associate Select CSR

- Associate destination ID 0x44 with multicast mask 1 on ingress port 3

  — Write the value 0x0000_0360 to the Multicast Associate Operation CSR

- Associate destination ID 0x44 with multicast mask 1 on ingress port 4

  — Write the value 0x0000_0460 to the Multicast Associate Operation CSR

- Associate destination ID 0x44 with multicast mask 1 on ingress port 5

  — Write the value 0x0000_0560 to the Multicast Associate Operation CSR

- Set up the operation to associate destination ID 0xFEED with multicast mask 2

  — Write the value 0xFEED_0002 to the Multicast Associate Select CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 0

— Write the value 0x0000_00E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 1

  — Write the value 0x0000_01E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 2

  — Write the value 0x0000_02E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 3

  — Write the value 0x0000_03E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 4

  — Write the value 0x0000_04E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 5

  — Write the value 0x0000_05E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 6

  — Write the value 0x0000_06E0 to the Multicast Associate Operation CSR

- Associate destination ID 0xFEED with multicast mask 2 on ingress port 7

  — Write the value 0x0000_07E0 to the Multicast Associate Operation CSR

4.4.3 Using Block Association

In this section assume that the switch supports block association rather than per-ingress-port association. With this feature sequential destination IDs can be quickly associated to sequential multicast masks. In order to take advantage of this feature, different destination IDs assignments are required for the system than for the preceding examples. The starting destination 0xFF00 is arbitrarily selected.

- the 16 bit destination ID 0xFF00 is used to multicast from ports 0, 1 and 2 to ports 6 and 7, so destination ID 0xFF00 needs to be associated with multicast mask 0.
- the 16 bit destination ID 0xFF01 identifies the multicast group including ports 3, 4 and 5, so destination ID 0xFF01 needs to be associated with multicast mask 1.
- the 16 bit destination ID 0xFF02 identifies the multicast group that includes all ports, so destination ID 0xFF02 needs to be associated with multicast mask 2.

Note that the number of accesses needed to accomplish the desired associations is reduced to two. (The accesses must be performed in the order given.)

- Set up the associate operation starting with destination ID 0xFF00 and multicast mask 0

  — Write the value 0xFF00_0000 to the Multicast Associate Select CSR

- Associate three sequential destination IDs starting at 0xFF00 with three sequential multicast masks starting at 0

  — Write the value 0x0002_00E0 to the Multicast Associate Operation CSR

4.4.4 Using Per-Ingress Port and Block Association

Next, if both block association and per-ingress port association are supported by the switch, then the following sequence of operations is required. (The write to the Multicast Associate Select CSR must occur before the corresponding write to the Multicast Associate Operation CSR. The individual association operations can be performed in any order.)

- Set up the associate operations starting with destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF00_0000 to the Multicast Associate Select CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 0
  - — Write the value 0x0002_00E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 1
  - — Write the value 0x0002_01E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 2
  - — Write the value 0x0002_02E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 3
  - — Write the value 0x0002_03E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 4
  - — Write the value 0x0002_04E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 5
  - — Write the value 0x0002_05E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 6
  - — Write the value 0x0002_06E0 to the Multicast Associate Operation CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 7
  - — Write the value 0x0002_07E0 to the Multicast Associate Operation CSR

For this example, suppose that ingress port 4 needs a second destination ID to be mapped to each of the three multicast masks and the switch also has this capability. The second destination would be added to port 4 with the following association operation. (The write to the Multicast Associate Select CSR must occur before the write to the Multicast Associate Operation CSR.

- Set up the associate operations starting with destination ID 0xFF03 and multicast mask 0
  - — Write the value 0xFF03_0000 to the Multicast Associate Select CSR
- Associate three sequential destination IDs with three sequential multicast masks for ingress port 4
  - — Write the value 0x0002_04E0 to the Multicast Associate Operation CSR

4.4.5 Removing a Destination ID to Multicast Mask Association

Now assume that packets from destination ID 0xFF02 on port 4 should no longer be allowed to multicast to all nodes (multicast mask 2). To remove destination ID 0xFF02 from being associated with multicast mask 2 on port 4, the following register accesses need to be performed in order.

- Set up the operation to remove the association between destination ID 0xFF02 and multicast mask 2
  - — Write the value 0xFF02_0002 to the Multicast Associate Select CSR
- Remove the association between destination ID 0xFF02 and multicast mask 2 on ingress port 4
  - — Write the value 0x0000_04C0 to the Multicast Associate Operation CSR

4.4.6 Querying an Association

There are three scenarios for querying destination ID to multicast mask associations in a switch. For the first scenario assume that a system designer wants to know which multicast masks are associated with destination ID 0xFF01 on port 4. Note that since a read of the Multicast Associate Operation CSR causes the last command written to be executed, that register is only written at the beginning of the sequence. (The individual associations can be queried in any order.)

- Set up the associate operations for destination ID 0xFF01 and multicast mask 0
  - — Write the value 0xFF01_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF01 is not associated with multicast mask 0 for port 4
  - — Write the value 0x0000_0480 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF01 and multicast mask 1
  - — Write the value 0xFF01_0001 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF01 is not associated with multicast mask 1 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR

- Set up the associate operations for destination ID 0xFF01 and multicast mask 2
  - — Write the value 0xFF01_0002 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF01 is associated with multicast mask 2 for port 4
  - — Read the value 0x0000_0481 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF01 and multicast mask 3
  - — Write the value 0xFF01_0003 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF01 is not associated with multicast mask 3 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR

For the second scenario assume that the system designer wants to know which destination IDs from 0xFF00 through 0xFF07 are associated with multicast mask 0 on Port 4. (The individual associations may be queried in any order.)

- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF00_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF00 is associated with multicast mask 0 for port 4
  - — Write the value 0x0000_0480 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF01_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF01 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF02_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF02 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF03_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR

- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF04_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF04 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF05_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF05 is associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0481 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF06_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF06 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Set up the associate operations for destination ID 0xFF00 and multicast mask 0
  - — Write the value 0xFF07_0000 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF07 is not associated with multicast mask 0 for port 4
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR

For the last scenario assume that the system designer now wants to know whether or not destination ID 0xFF03 is mapped to multicast mask 3 on all ports. (The individual associations may be queried in any order.)

- Set up the associate operations for destination ID 0xFF03 and multicast mask 3
  - — Write the value 0xFF03_0003 to the Multicast Associate Select CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 0
  - — Write the value 0x0000_0080 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0080 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 1
  - — Write the value 0x0000_0180 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0180 from the Multicast Associate Operation CSR

- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 2
  - — Write the value 0x0000_0280 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0280 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is associated with multicast mask 3 for port 3
  - — Write the value 0x0000_0380 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0381 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 4
  - — Write the value 0x0000_0480 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0480 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 5
  - — Write the value 0x0000_0580 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0580 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 6
  - — Write the value 0x0000_0680 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0680 from the Multicast Associate Operation CSR
- Verify that destination ID 0xFF03 is not associated with multicast mask 3 for port 7
  - — Write the value 0x0000_0780 to the Multicast Associate Operation CSR
  - — Read the value 0x0000_0780 from the Multicast Associate Operation CSR

Blank page

Annex A End Point Considerations (Informative)

A.1 Introduction

This appendix provides implementation considerations for end points that are intended to be used in a multicast RapidIO system.

A.2 Multicast Destination ID

If an end point does validation of the destination ID of a received packet against it's own deviceID or IDs, then that end point must be able to disable the comparison or have a deviceID assignment that allows validation of the multicast packet.

A.3 End Point Multicast Channels

It may be valuable for an end point to have support for one or more multicast channels. Multicast channels are address ranges in RapidIO address space for which an end point may accept a multicast packet and possibly translate the RapidIO write address to another local address region. This is necessary if the recipient of a multicast transaction does not have valid address space at the address received. The size and quantity of multicast channels depend on the requirements of the application. It may also be necessary to link multicast channels to particular multicast groups.

A multicast channel valid bit can be implemented to control whether an address out-of-range error occurs for a received address which falls inside a multicast channel address range. A multicast channel enable bit can control whether an end point silently ignores the packet when an address is received which falls inside the channel address range. The enable bit allows software finer control over which end points for a particular multicast ID will actually process the multicast write without modifying switch settings in the fabric.

Blank page

Annex B Multicast Applications (Informative)

B.1 Introduction

In a multi-switch RapidIO fabric, each switch which supports multicast in the fabric will have it's own set of multicast masks. The particular multicast mask in each switch device associated with a multicast group is very likely to have a different egress port pattern enabled, depending upon where that switch is in the switch fabric topology.

As an example, refer to the following system, where data streams entering switch A1 need to be sent to a set of destinations. There are several possible approaches to implementing this system. The first example is based on the number of different multicast groups that must be supported. Destinations are linked to a destination ID which in turn is associated with static multicast mask values. In the second example, a specific multicast mask in each switch is associated with each possible destination. The destinations are linked statically to destination IDs.

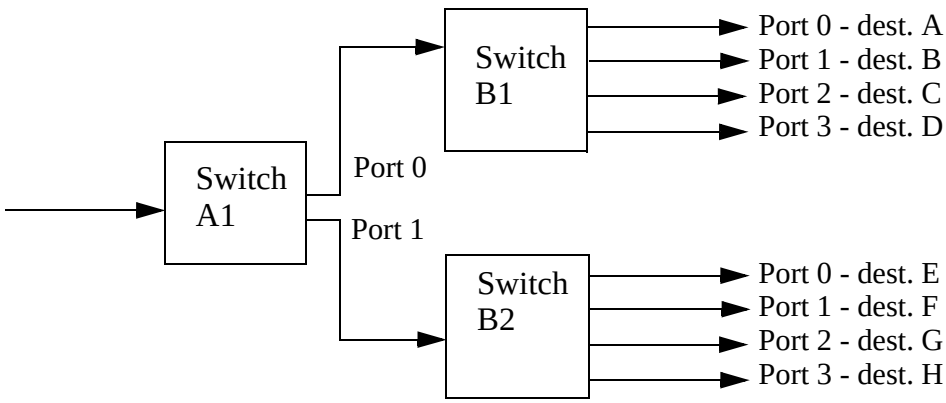


Figure 4-1. Example System using Multicast

B.2 Example 1 - Static Multicast Masks

If there are 256 combinations of destinations to receive a data stream, multicast requires 256 multicast groups, associated with 256 destination IDs. This means that an 8 bit destination ID could be used, but then there would be no destination IDs left over for control traffic in the system. As a result, this example assumes that the system needs to use 16 bit destination IDs in order to support multicast.

It is possible to use the least significant 4 bits of the 16 bit destination ID to identify which ports in Switch B1 need to be multicast to, and the next most significant 4 bits for the ports on B2. Arbitrarily selecting the value of 0x04 for the upper byte of the destination ID, then all multicast destination IDs have a format of 0x04XY, where X selects the ports in switch B2 and Y selects the ports in switch B1.

Switch A1 therefore needs two multicast masks as shown in Table 4-1.

Table 4-1. Multicast Masks for Switch A1

Multicast Mask Index	Egress Ports	Description
0	None	Associated with destination ID 0x0400, indicating that no destination is to receive this data stream. Packets multicast with destination ID of 0x0400 are dropped without notification.
1	Port 0 and port 1	Associated with destination IDs 0x04XY, where both X and Y is not 0. These represent all destination IDs which need only be multicast to both Switch B1 and switch B2.

Destination IDs of the form 0x040Y, where Y is non-zero, or 0x04X0, where X is non zero, do not have to be replicated. They can be routed directly to either port 0 (for 0x040Y) or port 1 (0x04X0) using the standard switch routing tables since there is only a single egress port.

Because Multicast Mask 1 must have 223 ((256 total) - (16 for X) - (16 for Y) - (1 for none)) destination IDs associated with it, the Switch Multicast Information CAR MaxDestIDAssociations field must contain a value of at least 222. In this particular case, the easiest internal implementation for the selection of packets to be multicast may be the use of a non-existent port in the routing table. For example, since Switch A1 has three ports, make use of a non-existent port value in the routing table to signify that the packet is subject to multicast.

Switches B1 and B2 must have 16 multicast masks, each associated with a particular combination of their egress ports 0 through 3. Each multicast mask may have 16 destination IDs associated with it, so the Switch Multicast Information CAR MaxDestIDAssociations field must contain a value of at least 15.

Table 4-2 describes which destination IDs must be associated with each multicast group for Switches B1. Note that for index 0, if the routing tables in Switch A1 are set up correctly, no packets with those multicast groups should reach switch B1.

Table 4-2. Multicast Masks for Switch B1

Multicast Mask Index	Egress Ports	Description
0	None	Associated with the following destination IDs: 0x0400 0x0410 0x0420 ... 0x04E0 0x04F0
1	Port 0	Associated with the following destination IDs: 0x0401 0x0411 0x0421 ... 0x04E1 0x04F1
2	Port 1	Associated with the following destination IDs: 0x0402 0x0412 0x0422 ... 0x04E2 0x04F2
3	Ports 1and 0	Associated with the following destination IDs: 0x0403 0x0413 ... 0x04E3 0x04F3
4	Port 2	Associated with the following destination IDs: 0x0404 0x0414 0x0424 ... 0x04E4 0x04F4
5	Ports2 and 0	Associated with the following destination IDs: 0x0405 0x0415 0x0425 ... 0x04E5 0x04F5
6	Ports2 and 1	Associated with the following destination IDs: 0x0406 0x0416 0x0426 ... 0x04E6 0x04F6

Table 4-2. Multicast Masks for Switch B1

Multicast Mask Index	Egress Ports	Description
7	Ports 2, 1 and 0	Associated with the following destination IDs: 0x0407 0x0417 0x0427 ... 0x04E7 0x04F7
8	Port 3	Associated with the following destination IDs: 0x0408 0x0418 0x0428 ... 0x04E8 0x04F8
9	Ports 3 and 0	Associated with the following destination IDs: 0x0409 0x0419 0x0429 ... 0x04E9 0x04F9
10	Ports 3 and 1	Associated with the following destination IDs: 0x040A 0x041A 0x042A ... 0x04EA 0x04FA
11	Ports 3, 1 and 0	Associated with the following destination IDs: 0x040B 0x041B 0x042B ... 0x04EB 0x04FB
12	Ports3 and2	Associated with the following destination IDs: 0x040C 0x041C 0x042C ... 0x04EC 0x04FC
13	Ports 3, 2 and 0	Associated with the following destination IDs: 0x040D 0x041D 0x042D ... 0x04ED 0x04FD

Table 4-2. Multicast Masks for Switch B1

Multicast Mask Index	Egress Ports	Description
14	Ports 3, 2 and 1	Associated with the following destination IDs: 0x040E 0x041E 0x042E ... 0x04EE 0x04FE
15	Ports 3, 2, 1 and 0	Associated with the following destination IDs: 0x040F 0x041F 0x042F ... 0x04EF 0x04FF

Table 4-2 describes which destination IDs must be associated with each multicast group for Switches B1. Note that for index 0, if the routing tables in Switch A1 are set up correctly, no packets with those multicast groups should reach switch B2.

Table 4-3. Multicast Masks for Switch B2

Multicast Mask Index	Egress Ports	Description
0	None	Associated with the following destination IDs: 0x0400 0x0401 0x0402 ... 0x040E 0x040F
1	Port 0	Associated with the following destination IDs: 0x0410 0x0411 0x0412 ... 0x041E 0x041F
2	Port 1	Associated with the following destination IDs: 0x0420 0x0421 0x0422 ... 0x042E 0x042F
3	Ports 1and 0	Associated with the following destination IDs: 0x0430 0x0431 0x0432 ... 0x043E 0x043F

Table 4-3. Multicast Masks for Switch B2

Multicast Mask Index	Egress Ports	Description
4	Port 2	Associated with the following destination IDs: 0x0440 0x0441 0x0442 ... 0x044E 0x044F
5	Ports2 and 0	Associated with the following destination IDs: 0x0450 0x0451 0x0452 ... 0x045E 0x045F
6	Ports2 and 1	Associated with the following destination IDs: 0x0460 0x0461 0x0462 ... 0x046E 0x046F
7	Ports 2, 1 and 0	Associated with the following destination IDs: 0x0470 0x0471 0x0472 ... 0x047E 0x047F
8	Port 3	Associated with the following destination IDs: 0x0480 0x0481 0x0482 ... 0x048E 0x048F
9	Ports 3 and 0	Associated with the following destination IDs: 0x0490 0x0491 0x0492 ... 0x049E 0x049F
10	Ports 3 and 1	Associated with the following destination IDs: 0x04A0 0x04A1 0x04A2 ... 0x04AE 0x04AF

Table 4-3. Multicast Masks for Switch B2

Multicast Mask Index	Egress Ports	Description
11	Ports 3, 1 and 0	Associated with the following destination IDs: 0x04B0 0x04B1 0x04B2 ... 0x04BE 0x04BF
12	Ports3 and2	Associated with the following destination IDs: 0x04C0 0x04C1 0x04C2 ... 0x04CE 0x04CF
13	Ports 3, 2 and 0	Associated with the following destination IDs: 0x04D0 0x04D1 0x04D2 ... 0x04DE 0x04DF
14	Ports 3, 2 and 1	Associated with the following destination IDs: 0x04E0 0x04E1 0x04E2 ... 0x04EE 0x04EF
15	Ports 3, 2, 1 and 0	Associated with the following destination IDs: 0x04F0 0x04F1 0x04F2 ... 0x04FE 0x04FF

It is up to the application whether either of the switch routing tables should be used for the destination IDs associated with multicast masks 1, 2, 4, and 8, as packets for these destination IDs do not have to be replicated.

Configuring each of the 16 multicast masks for switches B1 and B2 should require a maximum of 2 writes to the Multicast Mask Load CSR. Multicast masks with one or two ports require a number of register writes equal to the number of ports. Multicast masks with three egress ports to be selected should add all of the ports and then remove the port which doesn't belong in the multicast mask, thus requiring a maximum of two register writes. The multicast mask with all ports selected requires 1 register write. Thus, to configure all 16 of the multicast masks requires a maximum of (0 + (5*1) + (10*2))=25 register write operations.

For the destination ID to multicast mask association operations for Switch B1, it

would make sense to implement block association operations since this would greatly reduce the amount of effort required to associate destination IDs with multicast masks. This feature makes possible in this example to associate a sequential block of 16 destination IDs with the 16 multicast masks with only 32 register writes. Refer to Table 4.4.3, “Using Block Association,” on page 30 for details of the pair of writes required for each block of 16 destination IDs.

For the destination ID to multicast mask association operations for Switch B2 there is no pattern that leverages the programming model to speed the association of destination IDs to multicast masks. In Switch B2, it would make sense to use the regular switch routing tables rather than a multicast mask for the destination IDs associated with multicast masks 1, 2, 4 and 8 in order to minimize the number of writes required. The remaining 12 multicast groups each require 32 register write operations to complete their associations with the appropriate destination IDs, for a total of 384 writes. Designers who prefer speed of initialization over reliability may reduce this to 352 register writes by ignoring the destination IDs associated with multicast mask 0.

For switch B2, it may make sense in some systems to implement application specific configuration registers to reduce the number of operations required for configuration.

There can be significant limitations to using static multicast masks. Assume, of the 8 destinations, destinations A, B, C, D, and E are receiving one data stream using destination ID 0x041F, and destinations F, G, and H are receiving a second data stream using destination ID 0x0420.

If destination E switches wishes to change to the second data stream, two things must happen. The destination ID for the first data stream must change from 0x041F to 0x040F in order to have the proper multicast mask for switches B1 and B2, and the destination ID for the second data stream must correspondingly change from 0x0420 to 0x0430.

Because the destination IDs have changed, the switches are now allowed to reorder packets sent to destination IDs 0x041F, 0x040F, 0x0420 and 0x0430, which may change the behavior of the system in unexpected or undesirable ways.

Another issue with static multicast masks is that the latency difference for a data stream between different destinations depends upon whether the data stream is routed using the regular switch routing table or multicast through a particular switch. The different destinations will see different performance characteristics.

These characteristics could have undesirable side effects for latency and jitter sensitive applications like Voice over Internet Protocol (VoIP).

B.3 Example 2 - Linking Multicast Masks to Destination IDs

As an alternative implementation, again suppose that there are 256 possible destinations which need to be multicast, numbered 0 through 255. Each destination has a number of data streams it can receive, up to 256, which is always associated with a 16 bit destination ID of the form 0x04<destination stream>. This requires 256 multicast masks in switches A1, B1, and B2.

When a destination changes the data stream it wants to receive, the multicast masks for that data stream need to be changed. First, the multicast mask in each switch associated with the stream currently being received needs to be modified to stop multicasting to this destination. Next, the multicast mask for the new data stream needs to be modified in each switch to enable multicast to that destination.

Depending on system requirements, there are many ways to implement the multicast capabilities in this system. For example, switch A1 could always multicast all data streams to both switch B1 and switch B2. In this case, switch A1 would require 1 multicast mask that could have all 256 destination IDs associated with it. Switch B1 and B2 may receive a lot of undesired traffic in this case.

Initial programming of the multicast masks is not a requirement as with example 1. No ports should be selected in any mask after reset. Multicast masks will be modified during system operation as destinations request to receive a particular data stream. Removing the data streams from one multicast mask and adding a data stream to a multicast mask can be performed in two register writes for each switch.

The destination ID associations with multicast masks can be done far more effectively in this example if the switch devices support block associate operations. Refer to Section 3.4, “Switch Multicast Information CAR (Configuration Space Offset 0x38)”, and the programming examples in Section 4.4, “Configuring Associations” for more information and examples.

Blank page

Glossary of Terms and Abbreviations

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

A

Associate, Association. A defined relationship between a destination ID and a group of end point devices, or, in a switch, a defined relationship between a destination ID and a multicast mask.

M

Multicast. The concept of sending a single message to multiple destinations in a system.

Multicast group. The group of end point devices in a system that is the target of a multicast operation.

Multicast mask. The group of egress ports in a switch that are the targets of a replicated multicast packet.

Blank page

Blank page

Blank page

RapidIO™ Interconnect Specification Annex 1: Software/System Bring Up Specification

Rev. 1.3, 06/2005

RapidIO Trade Association

Revision History

Revision	Description	Date
1.0	First release	12/17/2003
1.3	Technical changes: the following errata showings: 04-09-00020.001, 04-09-00023.001 Converted to ISO-friendly templates Revision bumped to align with the rest of the specification stack	02/23/2005
1.3	Removed confidentiality markings for public release	06/07/2005

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

RapidIO Trade Association
Suite 325, 3925 W. Braker Lane
Austin, TX 78759
512-305-0070 Tel.
512-305-0009 FAX.

Table of Contents

Table of Contents

List of Figures

List of Figures

Blank page

Chapter 1 Overview

1.1 Introduction

This chapter provides an overview of the *RapidIO Annex 1: Software/System Bring Up Specification Rev. 1.3* document. This document assumes that the reader is familiar with the RapidIO specifications, conventions, and terminology.

1.2 Overview

The RapidIO Architectural specifications establish a framework that enables a wide variety of implementations. The *RapidIO Part 7: System and Device Inter-operability Specification* provides a standard set of device and system design solutions to support inter-operability. This document builds upon the inter-operability specification to define a standard set of software API functions for use in system bring up.

Each chapter addresses a different bring up topic. This revision of the *RapidIO Annex 1: Software/System Bring Up Specification Rev. 1.3* document covers the following issues:

Chapter 2, “Requirements for System Bring Up”

Chapter 3, “Hardware Abstraction Layer”

Chapter 4, “Standard Bring Up Functions”

Chapter 5, “Routing-Table Manipulation Functions”

Chapter 6, “Device Access Routine Interface”

Annex A, “System Bring Up Guidelines”

1.3 Scope

Although RapidIO networks provide many features and capabilities, there are a few assumptions and restrictions that this specification relies on to simplify the bring up process and narrow the specification scope. These assumptions and restrictions are:

- Only two hosts may simultaneously enumerate a network. Two hosts may be needed on a network for fault tolerance purposes. System integrators must determine which hosts can perform this function.

- Only one host actually completes the network enumeration (this is referred to as the *winning host*). The second host must retreat and wait for the enumeration to complete or, assuming the winning host has failed, for enumeration to time out. If a time out occurs, the second host re-enumerates the network.
- After enumeration, other hosts in the system must passively discover the network to gather topology information such as routing tables and memory maps.

1.4 System Enumeration API

System enumeration API functions may be divided into two categories:

- Standard RapidIO functions that use hardware resources defined by the RapidIO specifications. These functions should rely on the support functions provided by the Hardware Abstraction Layer (HAL) to ensure portability between different platforms.
- Device-specific (vendor-specific) functions defined by a device manufacturer that use hardware resources outside of the scope of the RapidIO specifications. The main purpose of these functions is to provide Hardware Abstraction Layer (HAL) support to the standard RapidIO functions.

An important goal of this software API specification is to minimize the number of device-specific functions required for enumeration so that the portability of the API across hardware platforms is maximized.

1.5 Terminology

This document uses terms such as *local port, local configuration registers,* etc. to refer to hardware resources associated with a RapidIO end point device attached to (or combined with) the host processor that performs RapidIO system enumeration and initialization.

1.6 Software Conventions

To describe the software API functions, this document uses syntactic and notational conventions consistent with the C programming language. The conventions for naming functions and variables used by these APIs are outside of scope of this document.

Chapter 2 Requirements for System Bring Up

2.1 Introduction

This section describes basic requirements for system bring up and discovery. An overview of the system bring up process, including a system bring up example, is presented in Annex A, “System Bring Up Guidelines”.

2.2 Boot Requirements

The following system state is required for proper system bring up:

After the system is powered on, the state necessary for system enumeration to occur using multiple host processors is automatically initialized as follows (These initial state requirements are specified in the *RapidIO Part 7: System and Device Inter-operability Specification*):

- System devices are initialized with the following Base Device IDs:
  - — Non-boot-code and non-host device IDs are set to 0xFF (0xFFFF for 16-bit deviceID systems).
  - — Boot code device IDs are set to 0xFE (0x00FE for 16-bit deviceID systems).
  - — Host device IDs are set to 0x00 (0x0000 for 16-bit deviceID systems).
- Physical layer link initialization of end points is complete.
- The default routing state of all switches between the boot code device and the host device is set to route all requests for device ID 0xFE (0x00FE for 16-bit deviceID systems) to the appropriate boot code device. All response packets are routed back to the host from the boot code device.
- Any host that participates in discovery must change its destination ID to a unique ID value before starting the system initialization process. This value is used by a device’s Host Base Device ID Lock CSR to ensure only one host can manipulate a device at a time. The allowed ID values for a discovering host are 0x00 (0x0000) and 0x01 (0x0001). A host with an ID of 0x00 (0x0000) has a lower priority than a host with an ID of 0x01 (0x0001). Host devices must be configured to accept maintenance packets with a destination ID of 0xFF (0xFFFF for 16-bit deviceID systems) as well as the unique host ID.

- All host devices have their Master Enable bit (Port General Control CSR) set to 1. Switch devices do not have a Master enable bit.

2.3 Enumeration Completion

One or two hosts can perform system enumeration in a RapidIO network. If two hosts are present, an algorithm is needed to determine which host has the priority to proceed with enumeration. The host with the higher priority is the *winning host* and the other host is the *losing host*. The enumeration algorithm suggested in Appendix A, “System Bring Up Guidelines,” on page 41 sets priority based on the value of the power-on device ID.

Enumeration is complete when the winning host releases the lock on the losing host. It is the losing host’s responsibility to detect that it has been locked by the winning host and to later detect that the lock has been released by the winning host. The methods used to release locks on nodes other than the host nodes is outside the scope of this document.

2.4 Enumeration Time-Out

As mentioned in the previous section, two hosts can be used to enumerate the RapidIO network. The algorithm in Appendix A assumes the host with the higher power-on host device ID has priority over the other host. Because of this pre-defined priority, only one host (the one with higher priority) can win the enumeration task. In this case, the losing host enters a wait state.

If the winning host fails to enumerate the entire network, the losing host’s wait state times out. When this occurs, the losing host attempts to enumerate the network. In an open 8-bit deviceID system, the losing host must wait 15 seconds before timing out and restarting the enumeration task. The length of the time-out period in a closed or a 16-bit deviceID system may differ from that of an open system.

To develop the 15 second timeout value, the following assumptions are made about the network maximal size:

NUMDEV = 256 devices

NUMSWITCHES = 256 switches

NUMFTE = 256 routing table entries per switch

It is assumed that a separate maintenance write packet is required to program each routing table entry for each switch. Since we need to establish a time base for operations, we assume:

CWTime = 100 microseconds per configuration write packet

Now we can estimate that the number of configuration writes it takes to program all

of the switch routing table entries is (256 switches)*(256 routing table entries), or;

=> 256*256*CWTIME microsecs =

=> ~6.6 seconds.

Given these rough approximations, a 15 second timeout value is seen as appropriate and conservative for open systems. The chosen value must be such that if a timeout were to occur, it must be guaranteed that failure HAS occured, and hence choosing a conservative value is necessary.

2.5 Function Return Codes

The following return codes and their constant values are defined for use by the system bring up functions.

```
typedef     unsigned int            STATUS;

#define     RIO_SUCCESS                             0x0     // Success status code
#define     RIO_WARN_INCONSISTENT                   0x1     // Used by
                                                            // rioRouteGetEntry—indicates
                                                            // that the routeportno returned is
                                                            // not the same for all ports
#define     RIO_ERR_SLAVE                           0x1001  // Another host has a higher
                                                            // priority
#define     RIO_ERR_INVALID_PARAMETER               0x1002  // One or more input parameters
                                                            // had an invalid value
#define     RIO_ERR_RIO                             0x1003  // The RapidIO fabric returned a
                                                            // Response Packet with ERROR
                                                            // status reported
#define     RIO_ERR_ACCESS                          0x1004  // A device-specific hardware
                                                            // interface was unable to generate
                                                            // a maintenance transaction and
                                                            // reported an error
#define     RIO_ERR_LOCK                            0x1005  // Another host already acquired
                                                            // the specified processor element
#define     RIO_ERR_NO_DEVICE_SUPPORT               0x1006  // Device Access Routine does not
                                                            // provide services for this device
#define     RIO_ERR_INSUFFICIENT_RESOURCES          0x1007  // Insufficient storage available in
                                                            // Device Access Routine private
                                                            // storage area
#define     RIO_ERR_ROUTE_ERROR                     0x1008  // Switch cannot support
                                                            // requested routing
#define     RIO_ERR_NO_SWITCH                       0x1009  // Target device is not a switch
#define     RIO_ERR_FEATURE_NOT_SUPPORTED           0x100A  // Target device is not capable of
                                                            // per-input-port routing
```

Blank page

Chapter 3 Hardware Abstraction Layer

3.1 Introduction

The Hardware Abstraction Layer (HAL) provides a standard software interface to the device-specific hardware resources needed to support RapidIO system configuration transactions. Configuration read and write operations are used by the HAL functions to access RapidIO device registers. The HAL functions are accessed by the RapidIO enumeration API during system bring up.

This section describes the HAL functions and how they can be used to access local and remote RapidIO device registers. These functions must be implemented by every new device-specific host-processing element to support RapidIO system enumeration and initialization. The HAL functions assume the following:

- All configuration read and write operations support only single word (4-byte) accesses.
- As required by the device, the size of the 8-bit or 16-bit deviceID field is considered by the device implementation (see section 2.4 of the *RapidIO Part 3: Common Transport Specification* for more information).
- An enumerating processor device may have more than one RapidIO end point (local port).

3.2 Device Addressing

One purpose of the HAL is to provide a unified software interface to configuration registers in both local and remote RapidIO processing elements. This is done using a universal device-addressing scheme. Such a scheme enables HAL functions to distinguish between accesses to local and remote RapidIO end points without requiring an additional parameter. The result is that only one set of HAL functions must be implemented to support local and remote configuration operations.

All HAL functions use the **destid** and **hopcount** parameters to address a RapidIO device. The HAL reserves **destid**=0xFFFFFFFF and hopcount of 0 for addressing configuration registers within the local RapidIO end point. A **destid**= 0xFFFFFFFF and hopcount of 0 value *must* be used to address the local processing end point regardless of the actual destination ID value. This reserved combination does not conflict with the address of other RapidIO devices. The **localport** parameter is used by the HAL functions to identify a specific local port within RapidIO devices containing multiple ports.

3.3 HAL Functions

The functions that form the RapidIO initialization HAL are described in the following sections.

3.3.1 Types and Definitions

/* The HOST_REGS value below is a destination ID used to specify that the registers of the processor/platform on which the code is running are to be accessed. */

#define HOST_REGS 0xFFFFFFFF

3.3.2 rioGetNumLocalPorts

Prototype:

```
INT32 rioGetNumLocalPorts (
        void
)
```

Arguments:

None

Return Value:

0	Error
n	Number of RapidIO ports supported

Synopsis:

rioGetNumLocalPorts() returns the total number of local RapidIO ports supported by the HAL functions. The number **n** returned by this function should be equal to or greater than 1. A returned value of 0 indicates an error.

3.3.3 rioConfigurationRead

Prototype:

```
STATUS rioConfigurationRead (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      offset,
        UINT32      *readdata
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the target device [IN]
hopcount	Hop count [IN]
offset	Word-aligned (four byte boundary) offset—in bytes—of the CAR or CSR [IN]
*readdata	Pointer to storage for received data [OUT]

Return Value:

RIO_SUCCESS	The read operation completed successfully and valid data was placed into the specified location.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioConfigurationRead() performs a configuration read transaction from CAR and/or CSR register(s) belonging to a local or remote RapidIO device. The function uses a device-specific hardware interface to generate maintenance transactions to remote devices. This hardware sends a configuration read request to the remote device (specified by **destid** and/or **hopcount**) and waits for a corresponding configuration read response. After the function receives a configuration read response it returns data and/or status to the caller. The method for accessing registers in a local device is device-specific.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

3.3.4 rioConfigurationWrite

Prototype:

```
STATUS rioConfigurationWrite (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      offset,
        UINT32      *writedata
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the target device [IN]
hopcount	Hop count [IN]
offset	Word-aligned (four byte boundary) offset—in bytes—of the CAR or CSR [IN]
*writedata	Pointer to storage for data to be written [IN]

Return Value:

RIO_SUCCESS	The write operation completed successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioConfigurationWrite() performs a configuration write transaction to CAR and/or CSR register(s) belonging to a local or remote RapidIO device. The function uses a device-specific hardware interface to generate maintenance transactions to remote devices. This hardware sends a configuration write request to the remote device (specified by

destid and/or **hopcount**) and waits for a corresponding configuration write response. After the function receives a configuration write response it returns status to the caller. The method for accessing registers in a local device is device-specific.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Chapter 4 Standard Bring Up Functions

4.1 Introduction

This section describes the RapidIO functions that must be implemented to support system bring up. Functions are defined only for device registers used during the RapidIO enumeration and initialization process, not for all possible RapidIO device registers. These functions can be implemented using the HAL functions. Many of the functions can also be implemented as macros that specify predefined parameters for the HAL functions. The standard RapidIO bring up functions can be combined into a library if they are implemented as a set of subroutines.

4.2 bring up Functions

The functions defined for the RapidIO enumeration and initialization process are described in the following sections.

4.3 Data Structures

typedef ADDR_MODE UINT32;

```
#define ADDR_MODE_34BIT_SUPPORT 0x1
#define ADDR_MODE_50_34BIT_SUPPORT 0x3
#define ADDR_MODE_66_34BIT_SUPPORT 0x5
#define ADDR_MODE_66_50_34BIT_SUPPORT 0x7
```

4.3.1 rioInitLib

Prototype:

```
STATUS rioInitLib (
        void
)
```

Arguments:

None

Return Value:

RIO_SUCCESS	Initialization completed successfully.
RIO_ERROR	Generic error report. Unable to initialize library.

Synopsis:

rioInitLib() initializes the RapidIO API library. No routines defined in this chapter may be called unless and untile rioInitLib has been invoked. If rioInitLib returns RIO_ERROR, no routines defined in this chapter may be called.

4.3.2 rioGetFeatures

Prototype:

```
STATUS rioGetFeatures (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *features
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*features	Pointer to storage containing the received features [OUT]

Return Value:

RIO_SUCCESS	The features were retrieved successfully and placed into the location specified by *features.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetFeatures() uses the HAL *rioConfigurationRead()* function to read from the Processing Element Features CAR of the specified processing element. Values read are placed into the location referenced by the ***features** pointer. Reported status is similar to *rioConfigurationRead()*

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.3 rioGetSwitchPortInfo

Prototype:

```
STATUS rioGetSwitchPortInfo (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *portinfo
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]

*portinfo	Pointer to storage containing the received port information [OUT]

Return Value:

RIO_SUCCESS	The port information was retrieved successfully and placed into the location specified by *portinfo.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetSwitchPortInfo() uses the HAL *rioConfigurationRead()* function to read from the Switch Port Information CAR of the specified processing element. Values read are placed into the location referenced by the ***portinfo** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.4 rioGetExtFeaturesPtr

Prototype:

```
STATUS rioGetExtFeaturesPtr (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      *extfptr
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*extfptr	Pointer to storage containing the received extended feature information [OUT]

Return Value:

RIO_SUCCESS	The extended feature information was retrieved successfully and placed into the location specified by *extfptr.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetExtFeaturesPtr() uses the HAL *rioConfigurationRead()* function to read the pointer to the first entry in the extended features list from the Assembly Information CAR of the specified processing element. That pointer is placed into the location referenced by the ***extfptr** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Note that if the EF_PTR field of *extfptr is 0, no extended features are available.

4.3.5 rioGetNextExtFeaturesPtr

Prototype:

```
STATUS rioGetNextExtFeaturesPtr (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      currfptr,
        UINT32      *extfptr
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
currfptr	Pointer to the last reported extended feature [IN]
*extfptr	Pointer to storage containing the received extended feature information [OUT]

Return Value:

RIO_SUCCESS	The extended feature information was retrieved successfully and placed into the location specified by *extfptr.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetNextExtFeaturesPtr() uses the HAL *rioConfigurationRead()* function to read the pointer to the next entry in the extended features. That pointer is placed into the location referenced by the ***extfptr** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Note that if the EF_PTR field of *extfptr is 0, no further extended features are available. Invoking rioGetNextExtFeaturesPtr when currfptr has an EF_PTR field value of 0 will result in a return code of RIO_ERR_INVALID_PARAMETER.

4.3.6 rioGetSourceOps

Prototype:

```
STATUS rioGetSourceOps (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *srcops
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*srcops	Pointer to storage containing the received source operation information [OUT]

Return Value:

RIO_SUCCESS	The source operation information was retrieved successfully and placed into the location specified by *srcops.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetSourceOps() uses the HAL *rioConfigurationRead()* function to read from the Source Operations CAR of the specified processing element. Values read are placed into the location referenced by the ***srcops** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.7 rioGetDestOps

Prototype:

```
STATUS rioGetDestOps (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *dstops
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*dstops	Pointer to storage containing the received destination operation information [OUT]

Return Value:

RIO_SUCCESS	The destination operation information was retrieved successfully and placed into the location specified by *dstops.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetDestOps() uses the HAL *rioConfigurationRead()* function to read from the Destination Operations CAR of the specified processing element. Values read are placed into the location referenced by the ***dstops** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.8 rioGetAddressMode

Prototype:

```
STATUS rioGetAddressMode (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        ADDR_MODE       *amode
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*amode	Pointer to storage containing the received address mode (34-bit, 50-bit, or 66-bit address) information [OUT]

Return Value:

RIO_SUCCESS	The address mode information was retrieved successfully and placed into the location specified by *amode.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetAddressMode() uses the HAL *rioConfigurationRead()* function to read from the PE Logical Layer CSR of the specified processing element. The number of address bits generated by the PE (as the source of an operation) and

processed by the PE (as the target of an operation) are placed into the location referenced by the ***amode** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.9 rioGetBaseDeviceId

Prototype:

```
STATUS rioGetBaseDeviceId (
        UINT8           localport,
        UINT32          *deviceid
)
```

Arguments:

localport	Local port number [IN]
*deviceid	Pointer to storage containing the base device ID [OUT]

Return Value:

RIO_SUCCESS	The base device ID information was retrieved successfully and placed into the location specified by *deviceid.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetBaseDeviceId() uses the HAL *rioConfigurationRead()* function to read from the Base Device ID CSR of the local processing element (the **destid** and **hopcount** parameters used by *rioConfigurationRead()* must be set to HOST_REGS and zero, respectively). Values read are placed into the location referenced by the ***deviceid** pointer. Reported status is similar to *rioConfigurationRead()*. This function is useful only for local end-point devices.

4.3.10 rioSetBaseDeviceId

Prototype:

```
STATUS rioSetBaseDeviceId (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          newdeviceid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
newdeviceid	New base device ID to be set [IN]

Return Value:

RIO_SUCCESS	The base device ID was updated successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioSetBaseDeviceId() uses the HAL *rioConfigurationWrite()* function to write the base device ID in the Base Device ID CSR of the specified processing element (end point devices only). Reported status is similar to *rioConfigurationWrite()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.11 rioAcquireDeviceLock

Prototype:

```
STATUS rioAcquireDeviceLock (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT16      hostdeviceid,
        UINT16      *hostlockid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
hostdeviceid	Host base device ID for the local processing element [IN]
*hostlockid	Device ID of the host holding the lock if ERR_LOCK is returned [OUT]

Return Value:

RIO_SUCCESS	The device lock was acquired successfully.
RIO_ERR_LOCK	Another host already acquired the specified processor element. ID of the device holding the lock is contained in the location referenced by the *hostlockid parameter.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioAcquireDeviceLock() tries to acquire the hardware device lock for the specified processing element on behalf of the requesting host. The function uses the HAL *rioConfigurationWrite()* function to write the requesting host device ID into the Host Base Lock Device ID CSR of the specified processing element. After the write completes, this function uses the HAL *rioConfigurationRead()* function to read the value back from the Host Base Lock Device ID CSR. The written and read values are compared. If they are equal, the lock was acquired successfully. Otherwise, another host acquired this lock and the device ID for that host is reported.

This function assumes unique host-based device identifiers are assigned to discovering hosts. For more details, refer to Annex A, "System Bring Up Guidelines".

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.12 rioReleaseDeviceLock

Prototype:

```
STATUS rioReleaseDeviceLock (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT16          hostdeviceid,
        UINT16          *hostlockid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
hostdeviceid	Host base device ID for the local processing element [IN]
*hostlockid	Device ID of the host holding the lock if ERR_LOCK is returned [OUT]

Return Value:

RIO_SUCCESS	The device lock was released successfully.
RIO_ERR_LOCK	Another host already acquired the specified processor element.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioReleaseDeviceLock() tries to release the hardware device lock for the specified processing element on behalf of the requesting host. The function uses the HAL *rioConfigurationWrite()* function to write the requesting host device ID into the Host Base Lock Device ID CSR of the specified processing element. After the write completes, this function uses the HAL *rioConfigurationRead()* function to read the value back from the Host Base Lock Device ID CSR. The written and read values are compared. If they are equal, the lock was acquired successfully. Otherwise, another host acquired this lock and the device ID for that host is reported.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.13 rioGetComponentTag

Prototype:

```
STATUS rioGetComponentTag (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      *componenttag
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*componenttag	Pointer to storage containing the received component tag information [OUT]

Return Value:

RIO_SUCCESS	The component tag information was retrieved successfully and placed into the location specified by *componenttag.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetComponentTag() uses the HAL *rioConfigurationRead()* function to read from the Component Tag CSR of the specified processing element. Values read are placed into the location referenced by the ***componenttag** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.14 rioSetComponentTag

Prototype:

```
STATUS rioSetComponentTag (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      componenttag
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
componenttag	Component tag value to be set [IN]

Return Value:

RIO_SUCCESS	The component tag was updated successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioSetComponentTag() uses the HAL *rioConfigurationWrite()* function to write the component tag into the Component Tag CSR of the specified processing element. Reported status is similar to *rioConfigurationWrite()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.15 rioGetPortErrStatus

Prototype:

```
STATUS rioGetPortErrStatus (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT16          extfoffset,
        UINT8           portnum,
        UINT32          *porterrorstatus
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
extfoffset	Offset from the previously reported extended features pointer [IN]
portnum	Port number to be accessed [IN]
*porterrorstatus	Pointer to storage for the returned value [OUT]

Return Value:

RIO_SUCCESS	The read completed successfully and valid data was placed into the location specified by *porterrorstatus.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had aninvalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetPortErrStatus() uses the HAL *rioConfigurationRead()* function to read the contents of the Port *n* Error and Status CSR of the specified processing element. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Blank page

Chapter 5 Routing-Table Manipulation Functions

5.1 Introduction

This section describes the RapidIO functions that must be provided to support routing tables used within the switch fabric. The RapidIO common transport specification requires implementing device-identifier-based packet routing. The detailed implementation of routing tables is beyond the scope of this specification.

The routing-table manipulation functions assume the following:

- The destination ID of the device that receives a packet routed by the switch is the *route destination ID.*
- The specific port at the route destination ID that receives a packet routed by the switch is the *route port number.*
- The software paradigm used for routing tables is a linear routing table indexed by the route destination ID.
- Switches may implement a global routing table, “per port” routing tables, or a combination of both.

5.2 Routing Table Functions

The functions defined for RapidIO routing-table manipulation are described in the following sections.

5.2.1 rioRouteAddEntry

Prototype:

```
STATUS rioRouteAddEntry (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT8       tableidx,
        UINT16      routedestid,
        UINT8       routeportno
)
```

Arguments:

localport	Local port number (RapidIO switch) [IN]
destid	Destination ID of the processing element (RapidIO

	switch) [IN]
hopcount	Hop count [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
routeportno	Route port number—value written to the selected routing table entry [IN]

Return Value:

RIO_SUCCESS	The routing table entry was added successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.
RIO_WARN_INCONSISTENT	Used by rioRouteGetEntry—indicates that the routeportno returned is not the same for all ports.

Synopsis:

rioRouteAddEntry() adds an entry to a routing table for the RapidIO switch specified by the **destid** and **hopcount** parameters. The **tableidx** parameter is used to select a specific routing table in the case of implementations with "per port" routing tables. A value of **tableidx**=0xFFFFFFFF specifies a global routing table for the RapidIO switch. The **routeportno** parameter is written to the routing table entry selected by the **routedestid** parameter.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

5.2.2 rioRouteGetEntry

Prototype:

```
STATUS rioRouteGetEntry (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT8       tableidx,
        UINT16      routedestid,
        UINT8       *routeportno
)
```

Arguments:

localport	Local port number (RapidIO switch) [IN]
destid	Destination ID of the processing element (RapidIO switch) [IN]
hopcount	Hop count [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
*routeportno	Route port number—pointer to value read from the selected routing table entry [OUT]

Return Value:

RIO_SUCCESS	The routing table entry was added successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.
RIO_WARN_INCONSISTENT	Used by rioRouteGetEntry—indicates that the routeportno returned is not the same for all ports.

Synopsis:

rioRouteGetEntry() reads an entry from a routing table for the RapidIO switch specified by the **destid** and **hopcount** parameters. The **tableidx** parameter is used to select a specific routing table in the case of implementations with "per port" routing tables. A value of **tableidx**=0xFF specifies a global routing table for the RapidIO switch. The value in the routing table entry selected by the **routedestid** parameter is read from the table and placed into the location referenced by the ***routeportno** pointer.

Reads from the global routing table may be undefined in the case where per-port routing tables exist.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Blank page

Chapter 6 Device Access Routine Interface

6.1 Introduction

This section defines the device access routine (DAR) interface that must be provided for RapidIO device configuration. The client for this interface is the boot loader responsible for RapidIO network enumeration and initialization. By using a standard DAR interface, the firmware does not need to include knowledge of device-specific configuration operations. Thus, enumeration and initialization firmware can operate transparently with devices from many component vendors.

6.2 DAR Packaging

For each processor type supported by a DAR provider, linkable object files for DARs shall be supplied using ELF format. Device-specific configuration DARs shall be supplied using C-language source code format.

6.3 Execution Environment

The functions provided by device-specific configuration DARs must be able to link and execute within a minimal execution context (e.g., a system-boot monitor or firmware). In general, configuration DARs should not call an external function that is not implemented by the DAR, unless the external function is passed to the configuration DAR by the initialization function. Also, configuration DAR functions may not call standard C-language I/O functions (e.g., *printf*) or standard C-language library functions that might manipulate the execution environment (e.g., *malloc* or *exit*).

6.4 Type Definitions

The following type definitions are to be used by the DAR functions in Section 6.5.

```
typedef struct RDCDAR_PLAT_OPS_STRUCT {
        UINT32 specversion;
        UINT32 (*rioConfigurationRead)      (   UINT8     localport,
                                                UINT16    destid,
                                                UINT8     hopcount,
                                                UINT32    offset,
                                                UINT32    *readdata);
        UINT32 (*rioConfigurationWrite)     (   UINT8     localport,
                                                UINT16    destid,
                                                UINT8     hopcount,
```

```
                                                    UINT32          offset,
                                                    UINT32          *writedata);
} RDCDAR_PLAT_OPS;

typedef struct RDCDAR_OPS_STRUCT {
        UINT32 specversion;
        UINT32 (*rioDarInitialize)                  (...);
        UINT32 (*rioDarTerminate)                   (...);
        UINT32 (*rioDarTestMatch)                   (...);
        UINT32 (*rioDarRegister)                    (...);
        UINT32 (*rioDarGetSwitchInfo)               (...);
        UINT32 (*rioDarSetPortRoute)                (...);
        UINT32 (*rioDarGetPortRoute)                (...);
        UINT32 (*rioDarGetMemorySize)               (...);
} RDCDAR_OPS

typedef struct RDCDAR_DATA_STRUCT {
        UINT32 databytesallocated;
        CHAR   *data;
} RDCDAR_DATA

typedef struct RDCDAR_SWITCH_INFO_STRUCT {
        BOOL   useslutmodel;
        BOOL   separatelutperinputport;
        UINT32 maxlutentries;
} RDCDAR_SWITCH_INFO
```

6.5 DAR Functions

The functions that must be provided for a RapidIO device-specific configuration DAR are described in the following sections. For the *rioDar_name*GetFunctionTable functions, the *rioDar_name* portion of the function name shall be replaced by an appropriate name for the implemented driver.

6.5.1 *rioDar_name*GetFunctionTable

Prototype:

```
UINT32 rioDar_nameGetFunctionTable(
        UINT32                  specversion,
        RDCDAR_OPS_STRUCT       *darops,
        UINT32                  maxdevices,
        UINT32                  *darspecificdatabytes
)
```

Arguments:

specversion	Version number of the DAR interface specification indicating the caller's implementation of the type definition structures [IN]
*darops	Pointer to a structure of DAR functions that are allocated by the caller and filled in by the called function (see Section 6.4) [OUT]
maxdevices	Maximum expected number of RapidIO devices that must be serviced by thisconfiguration DAR [IN]
*darspecificdatabytes	Number of bytes needed by the DAR for the DAR private data storage area [OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDar_name*GetFunctionTable()* is called by a client to obtain the list of functions implemented by a RapidIO device-specific configuration DAR module. It shall be called once before enumerating the RapidIO network.

The **specversion** parameter is the version number defined by the revision level of the specification from which the DAR type definition structures are taken (see Section 6.4).

The **maxdevices** parameter is an estimate of the maximum number of RapidIO devices in the network that this DAR must service. The DAR uses this estimate to determine the size required for the DAR private data storage area. The storage size is returned to the location referenced by the ***darspecificdatabytes** pointer. After the client calls this function, the client shall allocate a DAR private data storage area of a size no less than that indicated by ***darspecificdatabytes**. The client shall provide that private data storage area to *rioDarInitialize()*.

6.5.2 rioDarInitialize

Prototype:

```
UINT32 rioDarInitialize (
        UINT32                  specversion,
        UINT32                  maxdevices,
        RDCDAR_PLAT_OPS         *platops,
        RDCDAR_DATA             *privdata
)
```

Arguments:

specversion	Version number of the DAR interface specification indicating the caller's implementation of the type definition structures [IN]
maxdevices	Maximum expected number of RapidIO devices that must be serviced by this configuration DAR [IN]
*platops	Pointer to a structure of platform functions for use by the DAR (see Section 6.4) [IN]
*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDarInitialize() is called by a client to initialize a RapidIO device-specific configuration DAR module. This function shall be called once after calling the rioDar_name*GetFunctionTable()* functions and before enumerating the RapidIO network.

The **specversion** parameter is the version number defined by the revision level of the specification from which the DAR type definition structures are taken (see Section 6.4).

The **maxdevices** parameter is an estimate of the maximum number of RapidIO devices in the network that this DAR must service. The **maxdevices** value must be equal to the value used in the corresponding rioDar_name*GetFunctionTable()* function call. The client is responsible for allocating the structure referenced by ***privdata**. The client is also responsible for allocating a DAR private data storage area at least as large as that specified by the rioDar_name*GetFunctionTable()* call. The client must initialize the structure referenced by ***privdata** with the number of bytes allocated to the DAR private data storage area and with the pointer to the storage area. After calling *rioDarInitialize()*, the client may not deallocate the DAR private data storage area until after the *rioDarTerminate()* function has been called.

6.5.3 rioDarTerminate

Prototype:

```
UINT32 rioDarTerminate (
        RDCDAR_DATA             *privdata
```

)

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDarTerminate() is invoked by a client to terminate a RapidIO device-specific configuration DAR module. This function shall be called once after all use of the DAR services is completed. After calling this function, the client may deallocate the DAR private data storage area in the structure referenced by ***privdata**.

6.5.4 rioDarTestMatch

Prototype:

```
UINT32 rioDarTestMatch (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]

Return value:

RIO_SUCCESS	Device DAR does provide services for this device
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR does not provide services for this device.

Synopsis:

rioDarTestMatch() is invoked by a client to determine whether or not a RapidIO device-specific configuration DAR module provides services for the device specified by **destid**. The DAR interrogates the device (using the platform functions supplied during DAR initialization), examines the device identity and any necessary device registers, and determines whether or not the device is handled by the DAR.

The DAR does not assume that a positive match (return value of 0) means the DAR will actually provide services for the device. The client must explicitly register the device with *rioDARregister()* if the client will be requesting services.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.5 rioDarRegister

Prototype:

```
UINT32 rioDarRegister (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]

Return value:

RIO_SUCCESS	Device DAR successfully registered this device.
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR does not provide services for this device.
RIO_ERR_INSUFFICIENT_RESOURCES	Insufficient storage available in DAR private storage area

Synopsis:

rioDarRegister() is invoked by a client to register a target device with a RapidIO device-specific configuration DAR. The client must call this function once for each device serviced by the DAR. The client should first use the *rioDarTestMatch()* function to verify that the DAR is capable of providing services to the device.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.6 rioDarGetMemorySize

Prototype:

```
UINT32 rioDarGetMemorySize (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          regionix,
        UINT32          *nregions,
        UINT32          *regbytes[2],
        UINT32          *startoffset[2]
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]
regionix	Index of the memory region being queried (0, 1, 2, 3, ...) [IN]
*nregions	Number of memory regions provided by the target device [OUT]
*regbytes	Size (in bytes) of the queried memory region [OUT]
*startoffset	Starting address offset for the queried memory region [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully returned memory size

	information for the target device.
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR could not determine memory size information for the target device.

Synopsis:

rioDarGetMemorySize() is invoked by a client to determine the number of, the sizes of, and the offsets for the memory regions supported by a RapidIO target device. The function is intended to support the mapping of PCI or other address windows to RapidIO devices. If the **regionix** parameter is greater than the number of regions provided by the device (***nregions**), the DAR should return a value of zero for the ***regbytes** and ***startoffset** parameters, and indicate a "successful" (0) return code.

rioDarGetMemorySize always returns at least one region. The first index, index 0, always refers to the region controlled by the Local Configuration Space Base Address Registers.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.7 rioDarGetSwitchInfo

Prototype:

```
UINT32 rioDarGetSwitchInfo (
        RDCDAR_DATA             *privdata,
        UINT8                   localport,
        UINT32                  destid,
        UINT8                   hopcount,
        RDCDAR_SWITCH_INFO      *info
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
*info	Pointer to switch information data structure (see Section 6.4) [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully retrieved the information for RDCDAR_PLAT_OPS_STRUCT.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_NO_SWITCH	Target device is not a switch.

Synopsis:

rioDarGetSwitchInfo() is invoked by a client to retrieve the data necessary to initialize the RDCDAR_SWITCH_INFO structure.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.8 rioDarSetPortRoute

Prototype:

```
UINT32 rioDarSetPortRoute (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT8           tableidx,
        UINT16          routedestid,
        UINT8           routeportno
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
inport	Target switch device input port [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
routeportno	Route port number—value written to the selected routing table entry [IN]

Return value:

RIO_SUCCESS	Device DAR successfully modified the packet routing configuration for the target switch device.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_ROUTE_ERROR	Switch cannot support requested routing.
RIO_ERR_NO_SWITCH	Target device is not a switch.
RIO_ERR_FEATURE_NOT_SUPPORTED	Target device is not capable of per-input-port routing.

Synopsis:

rioDarSetPortRoute() is invoked by a client to modify the packet routing configuration for a RapidIO target switch device.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.9 rioDarGetPortRoute

Prototype:

```
UINT32 rioDarGetPortRoute (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT8           tableidx,
        UINT16          routedestid,
        UINT8           *routeportno
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
*routeportno	Route port number—pointer to value read from the selected routing table entry [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully modified the packet routing configuration for the target switch device.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_ROUTE_ERROR	Switch cannot support requested routing.
RIO_ERR_NO_SWITCH	Target device is not a switch.

Synopsis:

rioDarGetPortRoute() is invoked by a client to read the packet routing configuration for a RapidIO target switch device.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Annex A System Bring Up Guidelines

A.1 Introduction

The *RapidIO Annex 1: Software/System Bring Up Specification Rev. 1.3* defines a standard set of software API functions for use in system enumeration and initialization. These API functions enable up to two RapidIO hosts to cooperatively enumerate and configure a RapidIO network.

This appendix is provided as a reference model for the system bring up process. An algorithm is presented that enables up to two cooperating host processors in a Rapid IO system to enumerate the entire network, set up a route to every system node, and enable the booting software to start the next boot-process phase. The actual implementation of the algorithm used to bring up a RapidIO network can vary greatly from this model in both capability and complexity.

A.2 Overview of the System Bring Up Process

This section presents a high-level overview of the system bring up process.

1. The system is powered on. Refer to Chapter 2, “Requirements for System Bring Up” for the system power-on requirements.
2. The host processor fetches the initial boot code (if necessary). If two processors are present, both can fetch the initial boot code.
3. The system exploration and enumeration algorithm is started. The algorithm for this process is outlined in Section A.3 on page 42.
4. All devices have been enumerated and stored in the device database, and routes have been set up between the host device and all end point devices. The enumeration process may optionally choose to do the following:

   a) Compute and configure optimal routes between the host device and end point devices, and between different end point devices.

   b) Configure the switch devices with the optimal route information.

   c) Store the optimal route and alternate route information in the device database.
5. The address space is mapped.

The host may access the network across a host-RapidIO bridge or host-PCI bridge. The address-space mapping across this bridge must be done when devices are enumerated and stored in the device database. This allows the address of a found device to be retrieved later and presented to the device access routines during operating system (OS) initialization. The pseudocode for this process is as follows:

```
1   ACQUIRE the host bridge address-space requirement
2   MAP the address space into a host address partition X
3   FOR every device in the database
```

```
4       IF the component is a RapidIO device
5           ACQUIRE the device's address-space requirement
6           MAP the address space into a new host address partition
7           EXPAND the partition X window to cover the new partition
8           UPDATE the device database with the new host address
9       ELSE IF the component is a PCI bridge
10          ACQUIRE the bridge's PCI bus ID
11          ACQUIRE the bridge's address-space requirement
12          // All devices that appear behind this PCI bridge must have their address spaces
            // mapped within the region specified for this bridge.
13          MAP the address space into a new host address partition
14          EXPAND the partition X window to cover the new partition
15          UPDATE the device database with the new host address
16      ENDIF
17  ENDFOR
```

After discovery has been concluded, it is expected that the majority of systems will then attempt to load in a software image from a boot device.

A.3 System Enumeration Algorithm

The system enumeration algorithm is designed for use by one or two host processors. The outline of the algorithm is as follows:

1. Access the RapidIO network. This step may involve generating special transaction cycles to ensure that the RapidIO network is accessible.
2. Discover the host and assign a device ID to it.
3. Discover the neighbor, if present.
4. If necessary, repeat the previous step recursively to discover additional devices.
5. Clear up.

When a host begins exploring, it must acquire the Host Base Device ID Lock before it can proceed. Once acquired, it can set its device ID and discover its neighbor (if necessary).

If two hosts are used, both can execute the enumeration algorithm. However, only one host (the one with higher priority) can win the enumeration task. The losing host enters a wait state. The guidelines for prioritizing hosts to enumerate the network and restarting enumeration should the winning host fail to complete the task are described in Chapter 2, “Requirements for System Bring Up,” on page 9.

The enumeration algorithm described below sets priority based on the value of the power-on device ID. The winning host is the device with the higher power-on host device ID. The losing host has the lower power-on host device ID. The losing host enters a wait state until the winning host completes enumeration or until the wait state times out.

The prioritization mechanism never results in a deadlock if the priorities of both host

processors are unique. The enumeration process is initially performed in parallel by both hosts until they meet at a device. When a meeting occurs, prioritization guarantees one winning host—the other host retreats (enters a wait state).

The enumeration algorithm described below uses a recursive, depth-first graph traversal to discover the network. It may be possible to improve the algorithm using non-recursive or breadth-first graph traversal. However, those improvements and optimizations are implementation dependent and beyond the scope of this document.

A.3.1 Data Structures, Constants, and Global Variables

This section outlines the data structures, constants, and global variables used by the system enumeration algorithm pseudocode.

The example system is composed of only 8 bit capable devices.

Data Structures

```
struct rioRouteTable {

        // The switch routing table is implemented as a linear routing table for destination IDs. The table is
        // indexed using the destination ID and the table index range is equal to the maximum destination ID
        // value. The value of a table entry indicates the output port number used to route messages for the
        // destination ID. The table entry default value is implementation dependent. Table entries must be
        // initialized to support FLASH memory accesses. The algorithm pseudocode described in this
        // document assumes the device ID is equal to the RapidIO protocols destination ID. This assignment
        // is not a general requirement.

        UINT8   LFT[MAX_DEVICEID];
}

struct rioSwitch {

        …

        UINT16 SwitchIdentity;                  // Switch Identity
        UINT16 hopCount;                        // Hop Count to reach this switch
        UINT16 DeviceID;                        // Associated Device ID in the path to this switch
        struct rioRouteTable RouteTable;        // Switch Routing Table

        …

}
```

Constants

```
        RIO_GEN_DFLT_DID 0x00FFFFFF     // RIO_GEN_DFLT_DID is the general default device
                                        // ID assigned to non-host and non-boot code end
                                        // points
        RIO_BOOT_DFLT_DID 0x0000FFFE    // RIO_BOOT_DFLT_DID is the default device ID
                                        // assigned to boot code devices
        RIO_HOST_DFLT_DID 0x00000000    // RIO_HOST_DFLT_DID is the default device ID
                                        // assigned to host devices
```

Global Variables

```
        UINT16 DeviceID = 0;            // Currently available Device ID to be assigned to the
                                        // end point device
        UINT16 SwitchID = 0;            // Currently available Switch ID. This is used
                                        // internally by the to index
                                        // switches that have been discovered.
```

```
                                        // The following global arrays are used to store device
                                        // information
                                        // collected from rioGetFeatures and
                                        // rioGetSwitchPortInfo. They are
                                        // also used to store the hopCount and DeviceID
                                        // assigned to switches.

    struct rioSwitch Switches[MAX_SWITCHES];
```

A.3.2 Pseudocode

This section outlines the detailed pseudocode for the system enumeration algorithm.

```
1   //*************************************************************************
2   // System enumeration and initialization using the power-on device ID as the hostDeviceID
3   // —Discover the host first
4   // —Discover the host's neighbor recursively
5
6   STATUS rioSystemEnumerate (hostDeviceID)
7   {
8       // Discover the host first.
9       status = rioEnumerateHost (hostDeviceID);
10
11      if (status == ERR_SLAVE) {
12          rioClearUp (hostDeviceID);
13          return ERR_SLAVE;
14      }
15
16      // Discover the host neighbor
17      status = rioEnumerateNeighbor (hostDeviceID, hopCount = 1);
18
19      if (status == ERR_SLAVE) {
20          rioClearUp (hostDeviceID);
21          return ERR_SLAVE;
22      }
23
24      // If the code advances to this point successfully, the host must acquire the
25      // HostBaseDeviceIdLock for all devices in the system. When this is done, the Discovered bit
26      // Master Enable bit, etc. can be set for all devices.
27
28  }   // end rioSystemEnumerate
29
30  //*************************************************************************
31  // System Delay
32  // —Wait for other host to release the lock
33
34  rioDelay () {
35  }   // end rioDelay
36
37  //*************************************************************************
38  // Host enumeration and initialization
39
40  STATUS rioEnumerateHost (hostDeviceID)
41  {
42      // Try to acquire the lock
43      rioAcquireDeviceLock (0, hostDeviceID, 0, hostDeviceID);
44
```

```
45      while (HostBaseDeviceIdLockCSR.HostBaseDeviceID < hostDeviceID) {
46          // Delay for a while
47          rioDelay ();
48
49          // Retry lock acquisition
50          rioAcquireDeviceLock (0, hostDeviceID, 0, hostDeviceID, &lockingHost);
51      }
52
53      // Check to see if there is a master with a larger host device ID
54      if (HostBaseDeviceIdLock.HostBaseDeviceID > hostDeviceID) {
55          // Release the current lock
56          rioReleaseDeviceLock (0, hostDeviceID, 0, hostDeviceID);
57
58          return ERR_SLAVE;
59      }
60
61      // Lock has been acquired so enumeration can begin
62
63      // Assign the default host ID to the host
64      rioSetBaseDeviceId (0, hostDeviceID, hostDeviceID);
65
66      // Increment the available device ID
67      if (DeviceID == hostDeviceID) {
68          DeviceID ++;
69      }
70
71      return RIO_SUCCESS;
72  }   // end rioEnumerateHost
73
74  //************************************************************************
75  // Neighbor enumeration
76
77  STATUS rioEnumerateNeighbor (hostDeviceID, hopCount)
78  {
79      // The host has already discovered this node if it currently owns the lock
80      rioGetCurHostLock (0, 0, 0, &owner_device_id);
81      if (owner_device_id == hostDeviceID) {
82          return RIO_SUCCESS;
83      }
84
85      // Try to acquire the lock
86      rioAcquireDeviceLock (0, RIO_GEN_DFLT_DID, hopCount, hostDeviceID, &lockingHost);
87
88      while (HostBaseDeviceIdLockCSR.HostBaseDeviceID < hostDeviceID) {
89          // Delay for a while
90          rioDelay ();
91
92          // Retry lock acquisition
93          rioAcquireDeviceLock(0, RIO_GEN_DFLT_DID, hopCount, hostDeviceID,
                &lockingHost);
94      }
95
96      // Check to see if there is a master with a larger host device ID
97      if (HostBaseDeviceIdLock.HostBaseDeviceID > hostDeviceID) {
98          return ERR_SLAVE;
99      }
```

```
100
101     // Lock has been acquired so enumeration can begin
102
103     // Check Source Operation CAR and Destination Operation CAR to see if a Device ID can be
104     // assigned
105
106     rioGetSourceOps (0, RIO_GEN_DFLT_DID, hopCount, &SourceOperationCAR);
107     rioGetDestOps (0, RIO_GEN_DFLT_DID, hopCount, &DestinationOperationCAR);
108
109     if ( (SourceOperationCAR.Read || Write || Atomic) &&
110         (DestinationOperationCAR.Read || Write || Atomic)) {
111
112         // Set the device ID
113         rioSetBaseDeviceId (0, RIO_GEN_DFLT_DID, DeviceID);
114
115         // Increment the available device ID
116         DeviceID ++;
117         if (DeviceID == hostDeviceID) {
118             DeviceID ++;
119         }
120     }
121
122     // Check to see if the device is a switch
123     rioGetFeatures (0, RIO_GEN_DFLT_DID, hopCount, &ProcessingElementFeatureCAR);
124     if (ProcessingElementFeatureCAR.Switch == TRUE) {
125
126         // Read the switch information
127         rioGetSwitchPortInfo (0, RIO_GEN_DFLT_DID, hopCount,
                &SwitchPortInformationCAR);
128
129         // Record the switch device identity
130         Switches[SwitchID].SwitchIdentity = DeviceIdentityCAR.DeviceIdentity;
131
132         // Bookkeeping for the current switch ID
133         curSwitchID = SwitchID;
134
135         // Increment the available switch ID
136         SwitchID ++;
137
138         // Initialize the current switch routing table to add entries for all previously discovered
139         // devices so that they are routed correctly. Start with the host device ID (0x00) and end with
140         // DeviceID-1.
141         for (each deviceID in [0..DeviceID-1]) {
142             rioRouteAddEntry (0, RIO_GEN_DFLT_DID, hopCount, RIO_GEN_DFLT_DID,
                        deviceID,
143                 SwitchPortInformationCAR.PortNumber, NULL);
144         }
145
146         // Synchronize the current switch routing table with the global table
147         for (each deviceID in [0.. DeviceID-1]) {
148             Switches[curSwitchID].RouteTable.LFT[deviceID] =
149                 SwitchPortInformationCAR.PortNumber;
150         }
151
152         // Update the hopCount to reach the current switch
153         Switches[curSwitchID].HopCount = hopCount;
```

```
154
155         for (each portNum in SwitchPortInformationCAR.PortTotal) {
156             if (SwitchPortInformationCAR.PortNumber == portNum) {
157                 continue;
158             }
159
160             // Bookkeeping for the current available device ID
161             curDeviceID = DeviceID;
162
163             rioGetPortErrStatus (0, RIO_GEN_DFLT_DID, hopCount,
                    &PortErrorStatusCSR[portNum]);
164
165             // Check if it is possible to have a neighbor
166             if (PortErrorStatusCSR[portNum].PortUninitialized == TRUE) {
167                 continue;
168             }
169
170             else if (PortErrorStatusCSR[portNum].PortOK == TRUE) {
171
172                 // Check if it is an enumeration boundary port
173                 if (PortControlCSR[portNum].PortEnumerationBoundary == TRUE) {
174                     continue;
175                 }
176                 rioRouteAddEntry(0, RIO_GEN_DFLT_DID, hopCount, RIO_GEN_DFLT_DID, 0,
                        portNumber, NULL);
177
178                 // Discover the neighbor recursively
179                 if (status = rioEnumerateNeighbor(hopCount + 1) != RIO_SUCCESS) {
180                     return status;
181                 }
182
183                 // If more than one end point device was found, update the current switch routing table
184                 // entries beginning with the curDeviceID entry and ending with the DeviceID-1
185                 // entry.
186                 if (DeviceID > curDeviceID) {
187                     for (each deviceID in [curDeviceID..DeviceID-1]) {
188                         rioRouteAddEntry(0, RIO_GEN_DFLT_DID, hopCount, deviceID,
                                portNumber);
189                     }
190
191                     // Synchronize the current switch routing table with the global table
192                     for (each deviceID in [curDeviceID..DeviceID-1]) {
193                         Switches[curSwitchID].RouteTable.LFT[deviceID] = portNumber;
194                     }
195
196                     // Update the associated Device ID in the path.
197                     Switches[curSwitchID].DeviceID = curDeviceID;
198                 }   // end if
199             }   // end else if
200         }   // end for
201     } // end if (ProcessingElementFeatureCAR.Switch == TRUE)
202
203     return RIO_SUCCESS;
204
205 }   // end rioEnumerateNeighbor
206
```

```
// ************************************************************************
// System clear up
// —Reset the previously acquired lock because a master exists elsewhere. Use hostDeviceID to
// reset the lock

STATUS rioClearUp (hostDeviceID) {

    // Clear the host lock
    if (hostDeviceID > DeviceID –1) {
        rioReleaseDeviceLock (0, hostDeviceID, 0, hostDeviceID);
    }

    // Clear the discovered end point device lock
    while (DeviceID >= 1) {
        rioReleaseDeviceLock (0, DeviceID-1, 0, hostDeviceID);
        DeviceID --;
    }

    // Clear the discovered switch device lock
    while (SwitchID >= 1) {
        rioReleaseDeviceLock (0, Switches[SwitchID–1].DeviceID,
                              Switches[SwitchID-1].hopCount, hostDeviceID);
        SwitchID --;
    }

    return RIO_SUCCESS;
}   // end rioClearUp
```

A.4 System Bring Up Example

This section walks-through a system bring up example. The system described in this example is shown in Figure A-1.

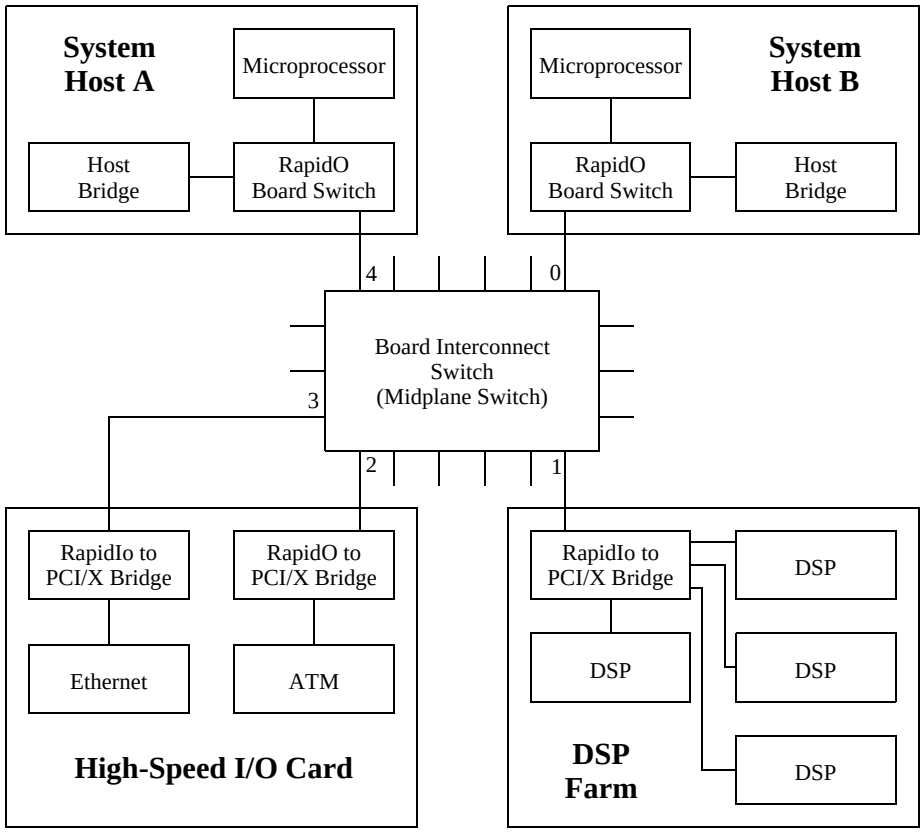
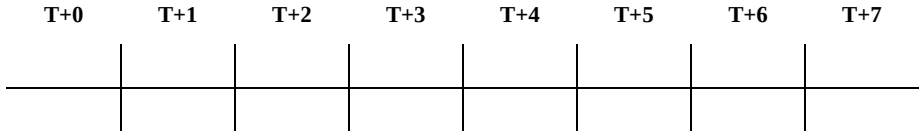


Figure A-1. Example System

Referring to Figure A-1, system Host A is preloaded with device ID 0x00 and system Host B is preloaded with device ID 0x01. Host A is configured to accept maintenance packets with destination IDs of 0x00 and 0xFF. Host B is configured to accept maintenance packets with destination IDs of 0x01 and 0xFF. System Bring Up advances through time slots along the following timeline:



The time slots shown above are defined as follows:

- **T+0:** Host A begins RapidIO enumeration.
- **T+1:** Host B begins RapidIO enumeration and Host A continues RapidIO enumeration.
- **T+2:** Host B discovers another host in the system (Host A) and waits.
- **T+3:** Host A discovers a higher priority host in the system (Host B) and retreats.

- **T+4:** Host B assumes sole enumeration of the system.
- **T+5:** Host B enumerates the PE on switch port 1.
- **T+6:** Host B enumerates the PEs on switch ports 2, 3 and 4.
- **T+7:** System enumeration is complete.

The following describes the actions taken during each time slot in more detail:

Time T+0

Host A attempts to acquire the lock from its Host Base Device ID Lock CSR by writing 0x00 to the CSR. Host A confirms it has acquired the lock when it reads the value of 0x00 (the host device ID) from the Lock CSR. Host A continues by reading the Processing Element Features CAR and adding the information from the CAR to its RapidIO device database. Host A updates its Base Device ID CSR with the host device ID (0x00).

Time T+1

Host B attempts to acquire the lock from its Host Base Device ID Lock CSR by writing 0x01 to the CSR. Host B confirms it has acquired the lock when it reads the value of 0x01 (the host device ID) from the Lock CSR. Host B continues by reading the Processing Element Features CAR and adding the information from the CAR to its RapidIO device database. Host B updates its Base Device ID CSR with the host device ID (0x01).

Host A begins neighbor enumeration. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x00), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. Host A confirms it has acquired the lock when it reads the value of 0x00 (the host device ID) from the Lock CSR.

Time T+2

Host B begins neighbor enumeration. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. However, after Host B issues a maintenance read from the Lock CSR it finds that the device was already locked by host device ID 0x00. Because Host B has a higher priority than the current lock holder (0x01 is greater than 0x00), Host B spins in a delay loop and repeatedly attempts to acquire the lock.

Time T+3

Host A continues neighbor enumeration. It issues a maintenance read cycle to the Device Identity CAR of the Board Interconnect Switch and looks for a matching entry in the device database. Device configuration continues because no match is found (Host A has not enumerated the device). Host A reads the Source Operations and Destination Operations CARs for the device. It is determined that the device

does not support read/write/atomic operations and does not require a device ID. Host A reads the Processing Element Feature CAR for the device and determines that it is a switch element.

Because the device is a switch, Host A reads the Switch Port Information CAR and records the device identity in the switch database. Next, Host A adds a set of entries to the switch's routing table. For each previously discovered device ID, an entry is created containing a target ID (0xFF), hop count (0), and the route port number (from the Switch Port Information CAR). The switch database is updated with the same routing information. Host A reads the Port Error Status CSR for switch port 0, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch's routing table containing target ID (0xFF), hop count (0), and the route port number (0).

Host A continues neighbor enumeration using a hop count of 1. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the neighbor PE on port 0. A maintenance write of the host device ID (0x00), the destination device ID (0xFF), and the hop count (1) is issued for the Lock CSR. However, after Host B issues a maintenance read from the Lock CSR it finds that the device was already locked by host device ID 0x01. Because Host A has a lower priority than the current lock holder (0x00 is less than 0x01), Host A retreats. It begins the process of backing out all enumeration and configuration changes it has made.

Host A checks its device and switch databases to find all host locks it obtained within the system (System Host A and the Board Interconnect Switch). It issues a maintenance write transaction to their Host Base Device ID Lock CSRs to release the locks.

Time T+4

As Host B spins in its delay loop, it attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. Because Host A released the lock, Host B is able to confirm it has acquired the lock when it reads the value of 0x01 from the Lock CSR.

Host B continues neighbor enumeration. It issues a maintenance read cycle to the Device Identity CAR of the Board Interconnect Switch and looks for a matching entry in the device database. Device configuration continues because no match is found (Host B has not enumerated the device). Host B reads the Source Operations and Destination Operations CARs for the device. It is determined that the device does not support read/write/atomic operations and does not require a device ID. Host B reads the Processing Element Feature CAR for the device and determines that it is a switch element.

Because the device is a switch, Host B reads the Switch Port Information CAR and records the device identity in the switch database. Next, Host B adds a set of entries to the switch's routing table. For each previously discovered device ID, an entry is

created containing a target ID (0xFF), hop count (0), and the route port number (from the Switch Port Information CAR). The switch database is updated with the same routing information. Host B reads the Port Error Status CSR for switch port 0, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch's routing table containing target ID (0xFF), hop count (0), and the route port number (0). Host B detects that it is attached to port 0. Because Host B has already been enumerated, neighbor enumeration continues on the next port.

Time T+5

Host B reads the Port Error Status CSR for switch port 1, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch's routing table containing target ID (0xFF), hop count (0), and the route port number (1).

Host B continues neighbor enumeration using a hop count of 1. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the neighbor PE on port 1. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (1) is issued for the Lock CSR. Host B confirms it has acquired the lock when it reads the value of 0x01 from the Lock CSR.

Host B issues a maintenance read cycle to the Device Identity CAR of the DSP Farm and looks for a matching entry in the device database. Device configuration continues because no match is found (Host B has not enumerated the device). Host B reads the Source Operations and Destination Operations CARs for the device. It is determined that the device supports read/write/atomic operations. A maintenance write is used to update the Base Device ID CSR with the value of 0x00 (the first available device ID). DeviceID is incremented and compared with the Host B device ID. Because they are equal, deviceID is assigned the next available device ID.

Time T+6

The process described in the previous step (Time T+5) is repeated on switch ports 2–4. Device IDs 0x02, 0x03, and 0x04 are assigned to the PEs on switch ports 2, 3 and 4, respectively.

Time T+7

Host A detects that its Host Base Device Lock CSR has been acquired by another host device, indicating it has been enumerated. Host A can initiate passive discovery to build a local system database.

Glossary of Terms and Abbreviations

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

A

Application programming interface (API.). A standard software interface that promotes portability of application programs across multiple devices.

C

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

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

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.

Discovery. The passive exploration of a RapidIO network fabric. This process involves walking an already enumerated RapidIO fabric to determine network topology and resource allocations.

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

E

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

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

Enumeration. The active exploration of a RapidIO network fabric. This process involves configuring device identifiers and maintaining proper host locking.

H

Hardware abstraction layer (HAL). A a standard software interface to device-specific hardware resources.

I

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

O

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

P

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

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.

S

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

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

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

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.

W

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

Write port. Hardware within a processing element that is the target of a portwrite operation.

Blank page

Blank page